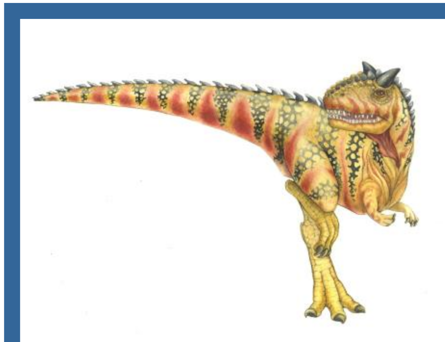
Virtualization



Chapter 16: Virtual Machines

- Overview
- History
- Benefits and Features
- Building Blocks
- Types of Virtual Machines and Their Implementations
- Containers
- Virtualization and Operating-System Components

Chapter Objectives

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

What is Virtualization?

- Fundamental component of cloud computing (especially in IaaS)

- Allows creation of isolated execution environment for multi-user environments

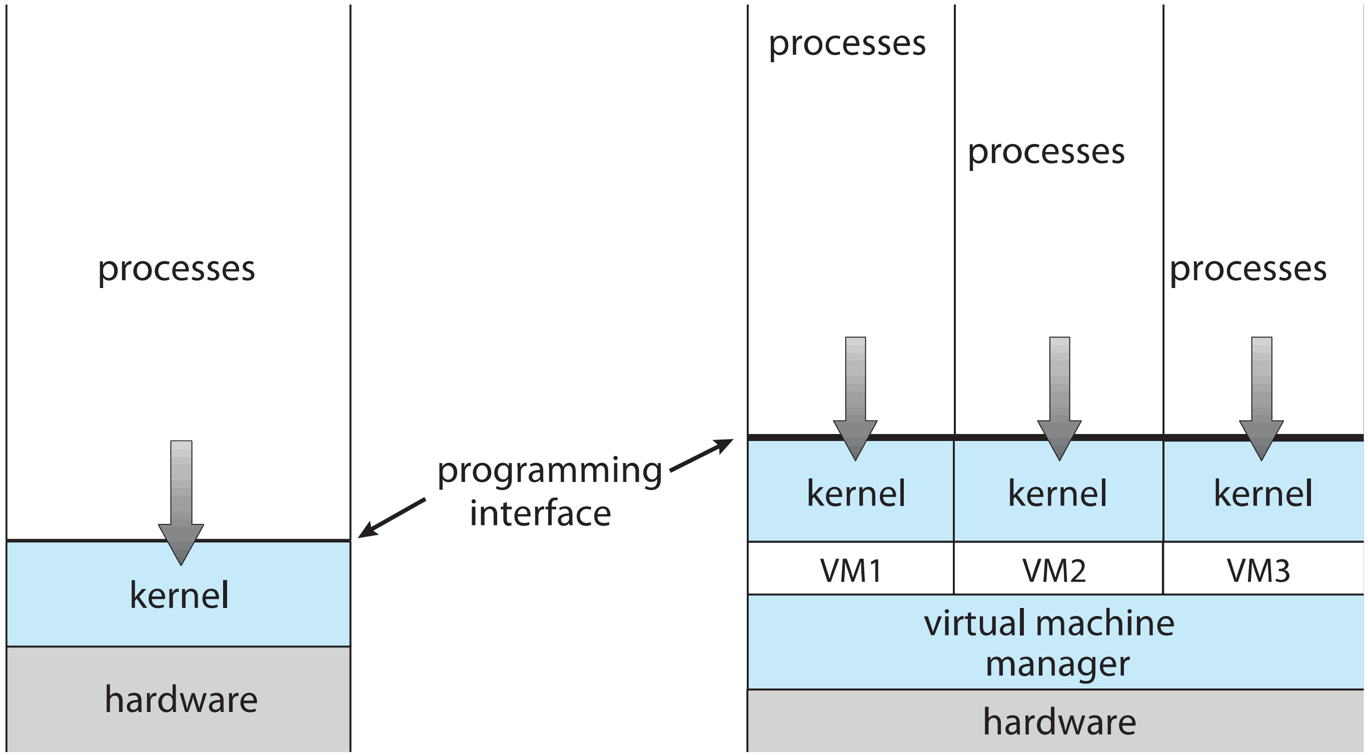
- Basic idea: ability of a computer program (software and hardware) to emulate an executing environment separate from the one that hosts such programs.

- Layer of indirection to run multiple machines (VMs) on hardware abstraction

Components

- Three main components
  - **Host** – underlying hardware system
  - **Virtual machine manager** (**VMM**) or **hypervisor** – creates and runs virtual machines by providing interface that is ***identical*** to the host
    - (Except in the case of paravirtualization)
  - **Guest** – process provided with virtual copy of the host
    - Usually an operating system
- Single physical machine can run multiple operating systems concurrently, each in its own virtual machine

System Models



Non-virtual machine

Virtual machine

Implementation Levels of Virtualization

- Instruction Set Architecture (ISA) Level
  - Emulate guest ISA by the ISA of host machine – mostly by code interpretation
  - Instructions in guest ISA are interpreted to host ISA one at a time
  - Dynamic binary translation for translating blocks of code makes it faster
  - e.g., MIPS binary code can run on x86 machine with emulation
- Hardware Abstraction Level
  - Provide a general virtual hardware environment for a VM and manage the underlying hardware
  - Virtualize a computer's hardware (CPU, storage, memory) and allow any guest OS to function
  - e.g., Xen hypervisor virtualizes x86 machine to run any OS

Implementation Levels of Virtualization

- Operating System Level
  - Virtualization layer between OS and applications to create isolation units for application execution
  - For example, Containers have own of set of processes, file system, etc.
  - Different applications are isolated in containers and share same OS
- Library Support Level
  - Application Programming Interface (API) is virtualized through controlling the API call to OS interfaces
  - e.g., WINE to execute Windows applications in Linux
- Application Level
  - Also known as process-level virtualization, where virtualization layer can run any high level language program compiled for a particular OS
  - e.g., Java Virtual Machine (JVM), *application sandboxing*

Virtualization at Hardware Level

- Virtual Machine (VM)
  - a software-based implementation of some real (hardware-based) computer
  - supports booting and execution of unmodified OSs and apps
  - managed by another software (VMM/Hypervisor)
  - essentially a set of files corresponding to disk, logs, activity, etc.

- Virtual machine monitor (VMM)
  - the software that creates and manages the execution of virtual machines
  - runs on bare-metal hardware or host OS
  - a VMM/Hypervisor is essentially a simple operating system

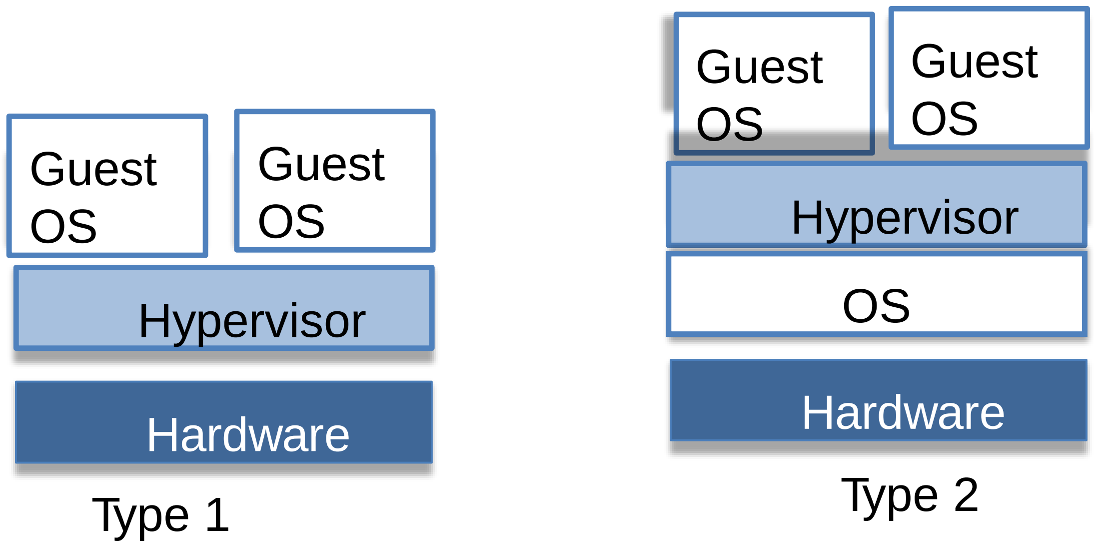
Implementation of VMMs

- Vary greatly, with options including:
  - **Type 0 hypervisors -** Hardware-based solutions that provide support for virtual machine creation and management via firmware
    - IBM LPARs and Oracle LDOMs are examples
  - **Type 1 hypervisors** - Operating-system-like software built to provide virtualization
    - Including VMware ESX, and Citrix XenServer
  - **Type 1 hypervisors** – Also includes general-purpose operating systems that provide standard functions as well as VMM functions
    - Including Microsoft Windows Server with HyperV and RedHat Linux with KVM
  - **Type 2 hypervisors** - Applications that run on standard operating systems but provide VMM features to guest operating systems
    - Including VMware Workstation and Fusion, and Oracle VirtualBox

Type 1 and Type 2 Hypervisor

- Type 1: Hypervisor runs directly on hardware – Bare metal or Native Hypervisor
  - –Examples: VMware ESX, Xen, Microsoft Hyper-V

- Type 2: Hypervisor runs on hardware – Hosted Hypervisor
  - – Examples: VMware Workstation, QEMU, Microsoft Virtual PC, Virtual Box



Criteria for an ideal Hypervisor

- **Equivalence**: same behavior as when it is executed directly on the physical host when it was running under control of VMM.

- **Resource control**: VMM should be in complete control of virtualized resources.

- **Efficiency**: a statistically dominant fraction of the machine instructions should be executed without intervention from the VMM.

Virtualization – a Brief History

- Invented by IBM in 1960's (System/360):
  - Sharing resources on expensive mainframes
  - CP: a "control program" that created and managed virtual machines
  - CMS: the "Cambridge monitor system" -- a lightweight, single-user OS
- 1970's - 1990's:
  - Cheap hardware and multiprocess OS became popular
  - Motivation for virtualization became unclear
  - Virtualization became unpopular

Virtualization – a Brief History

- VMware co-founded by Mendel Rosenblum and Diane Green in 1998
  - Brought virtualization to PC computers

- Their initial market was software developers
  - often need to develop and test software on multiple OSs (windows, linux, …)
  - using multiple PCs is very inconvenient
  - instead, run multiple OSs simultaneously in separate VMs

Virtualization Now

- Big companies (datacenters)

– operate many services: mail servers, file servers, Web servers, search services

– want to run at most one service per machine (administrative best practices)

– leads to low utilization, lots of machines, high power bills, administrative hassles

- Instead, run one service per virtual machine

– and consolidate many VMs per physical machine

– leads to better utilization, easier management

- Much larger market when cloud computing started

Virtualization Now

- Large-scale, hosted cloud computing (e.g., Amazon EC2)
  - the cloud provider buys a millions of computers and operates a data center
  - your run your software in a VM on their computers, and pay them rent
- run large number of VMs for a day and pay only for usage

Benefits and Features

- Host system protected from VMs, VMs protected from each other
  - i.e. A virus less likely to spread
  - Sharing is provided though via shared file system volume, network communication
- Freeze, **suspend**, running VM
  - Then can move or copy somewhere else and **resume**
  - Snapshot of a given state, able to restore back to that state
    - Some VMMs allow multiple snapshots per VM
  - **Clone** by creating copy and running both original and copy
- Great for OS research, better system development efficiency
- Run multiple, different OSes on a single machine
  - **Consolidation**, app dev, …

Benefits and Features (cont.)

- **Templating** – create an OS + application VM, provide it to customers, use it to create multiple instances of that combination
- **Live migration** – move a running VM from one host to another!
  - No interruption of user access
- All those features taken together -> **cloud computing**
  - Using APIs, programs tell cloud infrastructure (servers, networking, storage) to create new guests, VMs, virtual desktops

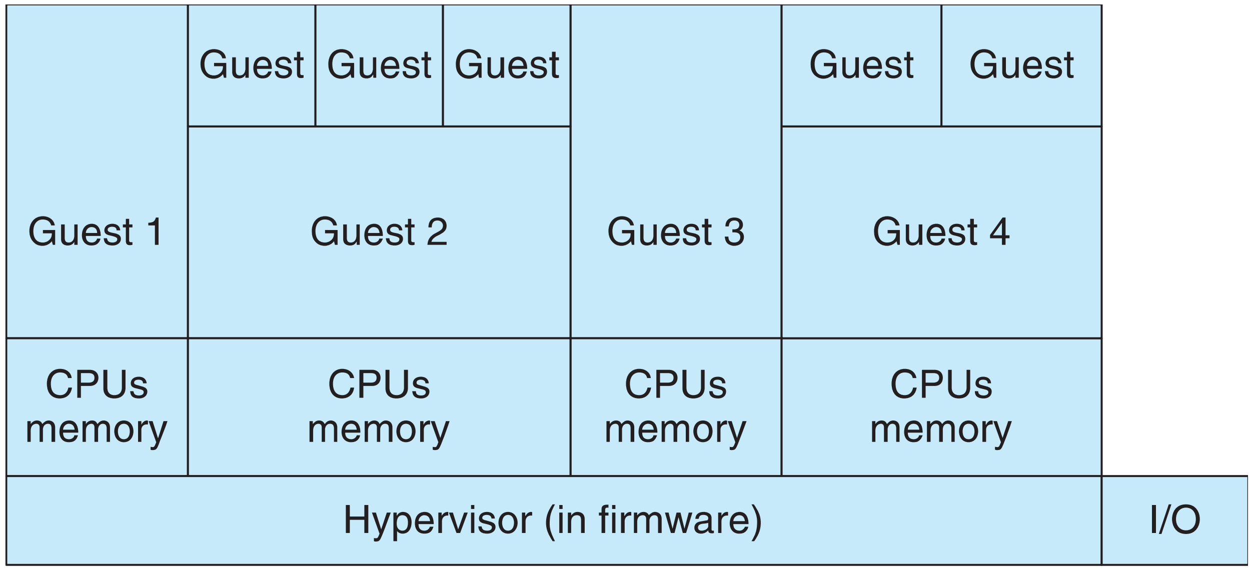
Types of Virtual Machines and Implementations

- Many variations as well as HW details
  - Assume VMMs take advantage of HW features
    - HW features can simplify implementation, improve performance
- Whatever the type, a VM has a lifecycle
  - Created by VMM
  - Resources assigned to it (number of cores, amount of memory, networking details, storage details)
  - Types: dedicated (type 0) or shared resources, or a mix
  - When no longer needed, VM can be deleted, freeing resources
- Steps simpler, faster than with a physical machine install
  - Can lead to **virtual machine sprawl** with lots of VMs, history and state difficult to track

Types of VMs – Type 0 Hypervisor

- Old idea, under many names by HW manufacturers
  - “partitions”, “domains”
  - A HW feature implemented by VMM in firmware
  - Smaller feature set than other types
  - Each guest has dedicated HW
- I/O a challenge as difficult to have enough devices, controllers to dedicate to each guest
- Sometimes VMM implements a **control partition** running daemons that other guests communicate with for shared I/O
- Can provide virtualization-within-virtualization (guest itself can be a VMM with guests
  - Other types have difficulty doing this

Type 0 Hypervisor



Types of VMs – Type 1 Hypervisor

- Commonly found in company datacenters
  - In a sense becoming "datacenter operating systems"
    - Datacenter managers control and manage OSes in new, sophisticated ways by controlling the Type 1 hypervisor
    - Consolidation of multiple OSes and apps onto less HW
    - Move guests between systems to balance performance
- Special purpose operating systems that run natively on HW
  - Rather than providing system call interface, create run and manage guest OSes
  - Run in kernel mode
  - Guests generally don't know they are running in a VM
  - Implement device drivers for host HW because no other component can
  - Also provide other traditional OS services like CPU and memory management

Types of VMs – Type 1 Hypervisor (cont.)

- Another variation is a general purpose OS that also provides VMM functionality
  - RedHat Enterprise Linux with KVM, Windows with Hyper-V, Oracle Solaris
  - Perform normal duties as well as VMM duties
  - Typically less feature rich than dedicated Type 1 hypervisors
- In many ways, treat guests OSes as just another process
  - Albeit with special handling when guest tries to execute special instructions

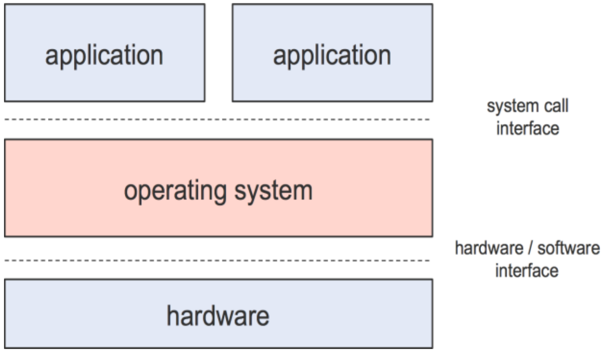
Types of VMs – Type 2 Hypervisor

- Less interesting from an OS perspective
  - Very little OS involvement in virtualization
  - VMM is simply another process, run and managed by host
    - Even the host doesn't know they are a VMM running guests
  - Tend to have poorer overall performance because can't take advantage of some HW features
  - But also a benefit because require no changes to host OS

What’s an application?

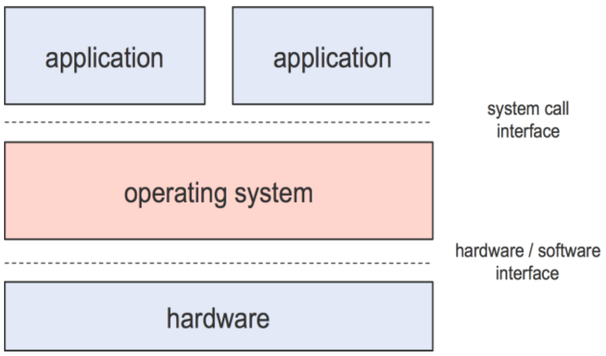
A program that relies on the system call interface

- While executing it, the CPU runs in unprivileged (user) mode
- a special instruction (“intc” on x86) lets a program call into the OS
  - the OS uses this to expose system calls
  - the program uses system calls to manipulate file system, network stack, etc.
- OS provides a program with the illusion of its own memory
  - MMU hardware lets the OS define the “virtual address space” of the program



What’s an application?

- Is this safe?
  - most instructions run directly on the CPU (fast)
    - but sensitive instructions cause the CPU to throw an exception to the OS
  - address spaces prevent program from accessing OS memory and each other‘s as well
  - it’s as though each program runs in its own, private machine (the “process”)



Protection Rings

- Protection ring is a level or hierarchical layer of privilege in a computer architecture (x86 CPU has 0,1,2,3 levels)
- Only Ring 0 can execute privileged instructions
  - Normally OS runs in Ring 0
- More privileged rings can access memory of less privileged ones
- Calling across rings can only happen with hardware enforcement

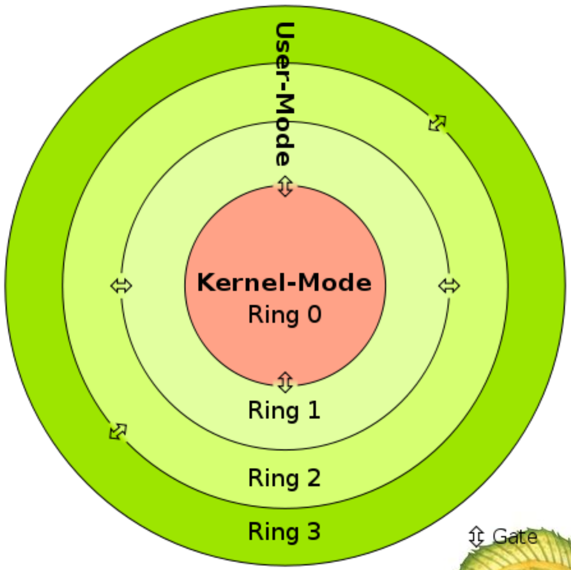


Image Source: https://commons.wikimedia.org/wiki/File:CPU_ring_scheme.svg

Classes of Instructions

Popek and Goldberg (1974) defined two classes of instructions

- privileged instructions: those that trap when CPU is in user-mode
- sensitive instructions: those that modify hardware configuration or resources (control sensitive) and
- those whose behavior depends on H/W configuration, i.e., user or kernel mode (behavior sensitive)

e.g., control sensitive instructions in x86 are PUSHF, POPF, SMSW

e.g., behavior sensitive instructions are POP, PUSH, JMP

Popek, G. J.; Goldberg, R. P. (July 1974). "Formal requirements for virtualizable third generation architectures". Communications of the ACM. ***17*** *(7): 412–421.*

Challenges in Virtualization

Until 2005, the Intel architecture did not meet Goldberg’s requirement

- 17 instructions were not virtualizable
  - they do not trap, and they behave differently in supervisor vs. user mode
  - some leak processor mode (e.g., SMSW, or store machine status word)
  - some behave differently (e.g., CALL or JMP to addresses that reference the protection mode of the destination)

Popek-Goldberg Theorem

- Theorem: A VMM can be constructed efficiently and safely if the set of sensitive instructions is a subset of the set of privileged instructions

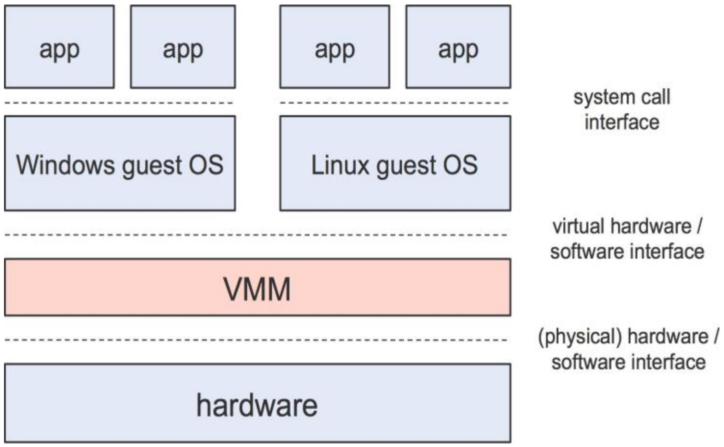
OR

- An architecture is virtualizable if its sensitive instructions is a subset of its privileged instructions

*Popek, G. J.; Goldberg, R. P. (July 1974). "Formal requirements for virtualizable third generation architectures". Communications of the ACM. **17** (7): 412–421. doi:10.1145/361011.361073*

What happens with a VM?

- What if we run the guest OS as a user-level program?
  - Guest OS is forced to run in Ring 1, with VMM or host OS in Ring 0
- What happens when Windows executes a sensitive instruction?
  - Traps to VMM for privileged operations
- What (virtual) hardware devices should guest OS see?
- How do you prevent guest OSes from hurting each other or the VMM?



Building Blocks

- Generally difficult to provide an ***exact*** duplicate of underlying machine
  - Especially if only dual-mode operation available on CPU
  - But getting easier over time as CPU features and support for VMM improves
  - Most VMMs implement **virtual CPU** (**VCPU**) to represent state of CPU per guest as guest believes it to be
    - When guest context switched onto CPU by VMM, information from VCPU loaded and stored (like PCB)
  - Several techniques, as described in next slides

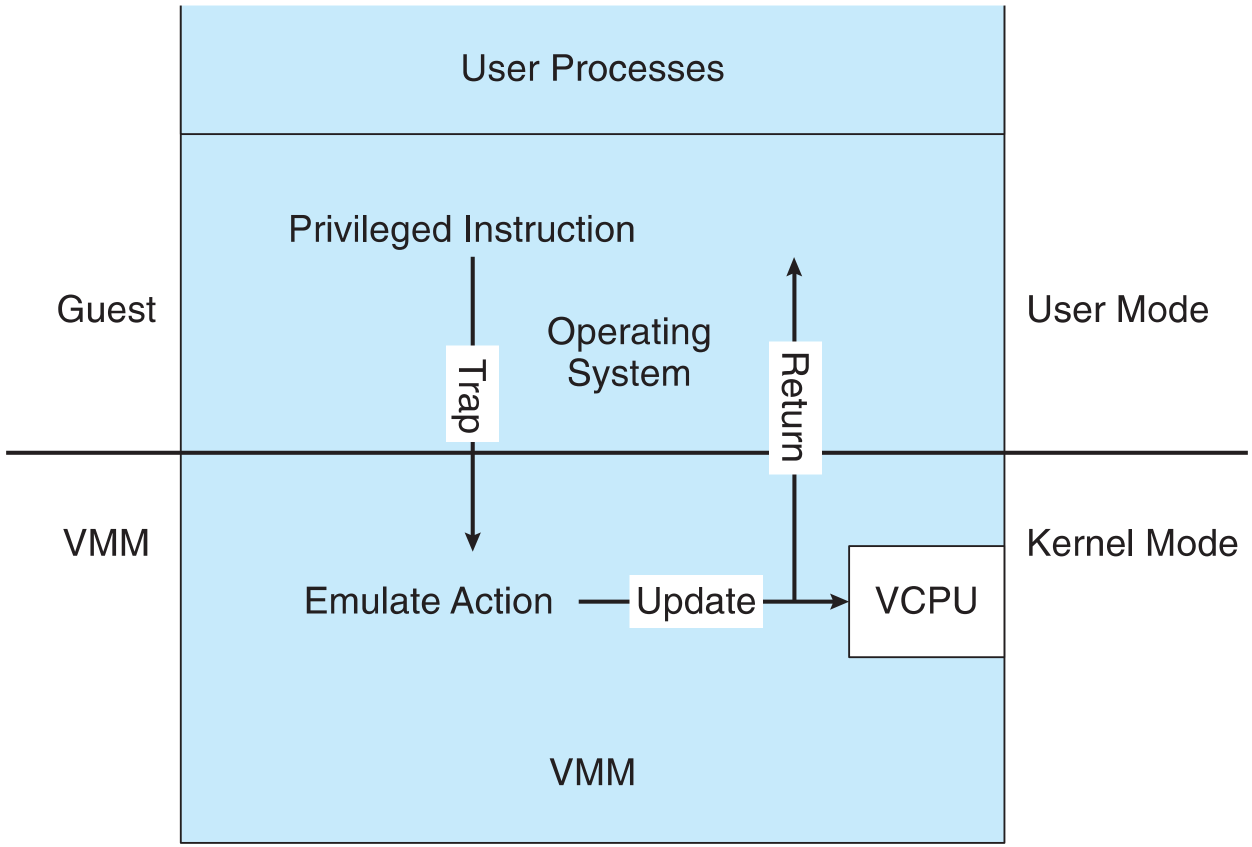
Building Block – Trap and Emulate

- Dual mode CPU means guest executes in user mode
  - Kernel runs in kernel mode
  - Not safe to let guest kernel run in kernel mode too
  - So VM needs two modes – virtual user mode and virtual kernel mode
    - Both of which run in real user mode
  - Actions in guest that usually cause switch to kernel mode must cause switch to virtual kernel mode

Trap-and-Emulate (cont.)

- How does switch from virtual user mode to virtual kernel mode occur? (**trap-and-emulate**)
  - Attempting a privileged instruction in user mode causes an error -> **trap**
  - VMM gains control, analyzes error, executes operation as attempted by guest
  - Returns control to guest in user mode
  - Widely used in most virtualization products
- User mode code in guest runs at same speed as if not a guest
- But kernel mode code runs slower due to trap-and-emulate
  - Especially a problem when multiple guests running, each needing trap-and-emulate
- Bypass VMM for non-sensitive cases, guest CPU state storing etc..
- CPUs adding hardware support, more CPU modes to improve virtualization performance

Trap-and-Emulate Virtualization Implementation



Building Block – Binary Translation

- Some CPUs don’t have clean separation between privileged and nonprivileged instructions
  - Earlier Intel x86 CPUs are among them (all sensitive instructions)
  - Backward compatibility means difficult to improve
  - Consider Intel x86 **`popf`** instruction
    - Loads CPU flags register from contents of the stack
    - If CPU in privileged mode -> all flags replaced
    - If CPU in user mode -> on some flags replaced
  - In trap and emulate, **`popf`** like instructions do not work.
    - Similar instructions exist (behavior sensitive)

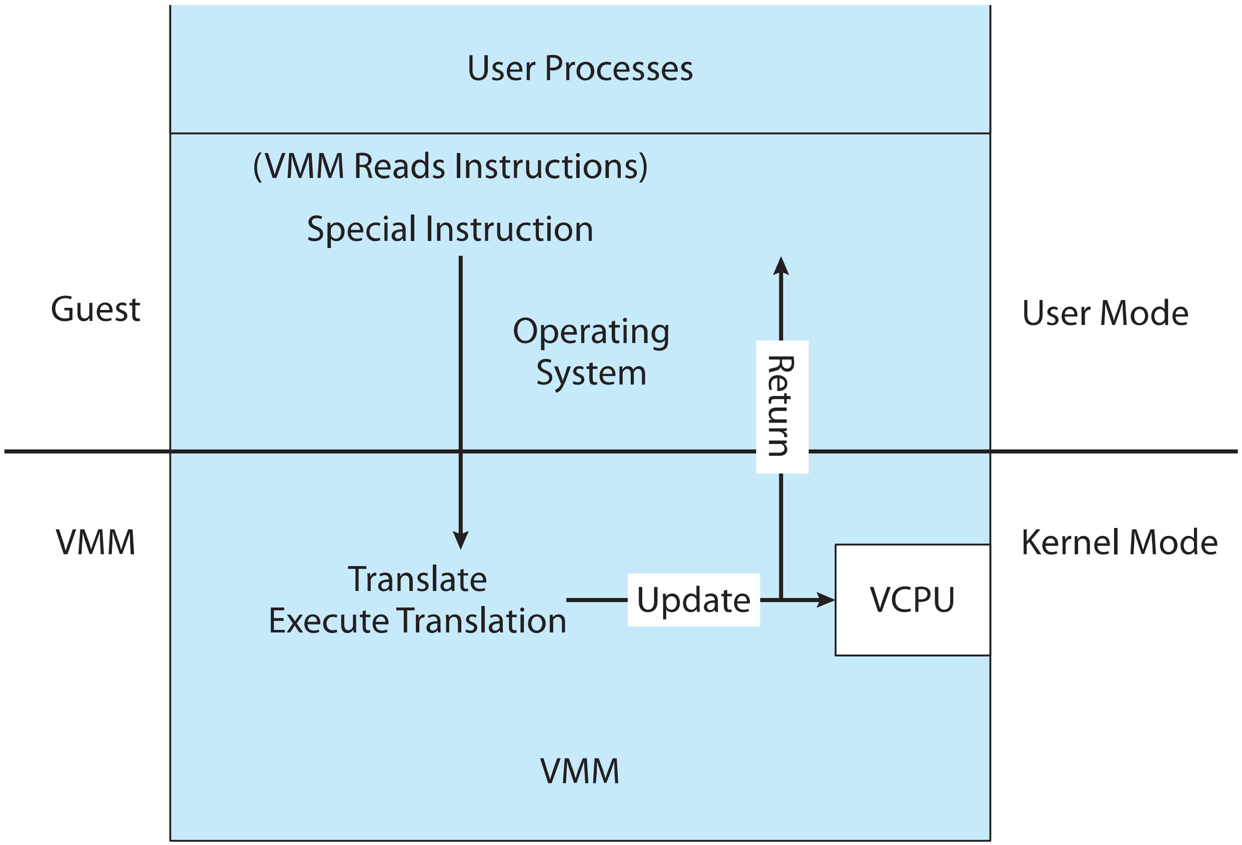
Binary Translation (cont.)

- Binary translation solves the problem
  - Basics are simple, but implementation very complex
  - If guest VCPU is in user mode, guest can run instructions natively
  - If guest VCPU in kernel mode (guest believes it is in kernel mode)
    - VMM examines every instruction guest is about to execute by reading a few instructions ahead of program counter
    - Non-special-instructions run natively
    - Special instructions translated into new set of instructions that perform equivalent task (for example changing the flags in the VCPU)

Binary Translation (cont.)

- Implemented by translation of code within VMM
- Code reads native instructions dynamically from guest, on demand, generates native binary code that executes in place of original code
- Performance of this method would be poor without optimizations
  - Products like VMware use caching
    - Translate once, and when guest executes code containing special instruction cached translation used instead of translating again
    - Testing showed booting Windows XP as guest caused 950,000 translations, at 3 microseconds each, or 3 second (5 %) slowdown over native

Binary Translation Virtualization Implementation



Nested Page Tables

- Memory management another general challenge to VMM implementations
- How can VMM keep page-table state for both guests believing they control the page tables and VMM that does control the tables?
- Common method (for trap-and-emulate and binary translation) is **nested page tables** (**NPTs**)
  - Each guest maintains page tables to translate virtual to physical addresses
  - VMM maintains per guest NPTs to represent guest's page-table state
    - Just as VCPU stores guest CPU state
  - When guest on CPU -> VMM makes that guest's NPTs the active system page tables
  - Guest tries to change page table -> VMM makes equivalent change to NPTs and its own page tables
  - Can cause many more TLB misses -> much slower performance

Building Blocks – Hardware Assistance

- All virtualization needs some HW support
- More support -> more feature rich, stable, better performance of guests
- Intel added new **VT-x** instructions in 2005 and AMD the **AMD-V** instructions in 2006
  - CPUs with these instructions remove need for binary translation
  - Generally define more CPU modes – “guest/non-root” and “host/root”
  - VMM can enable host mode, define characteristics of each guest VM, switch to guest mode and guest(s) on CPU(s)
  - In guest mode, guest OS thinks it is running natively, sees devices (as defined by VMM for that guest)
    - Access to virtualized device, priv instructions cause trap to VMM
    - CPU maintains VCPU, context switches it as needed
- HW support for Extended Page Tables, DMA, interrupts as well

Intel VT-x

- Guest OS runs in Ring 0 (unmodified)
- Trap all exceptions and privileged instructions by forcing a transition from the guest OS to the VMM (VM Exit)
- VMM Runs at higher level Ring -1 (VMX Root mode)
- VMCS (VM Control Structure)
  - Manages VMX transitions
  - Guest and host states saved and loaded during transitions

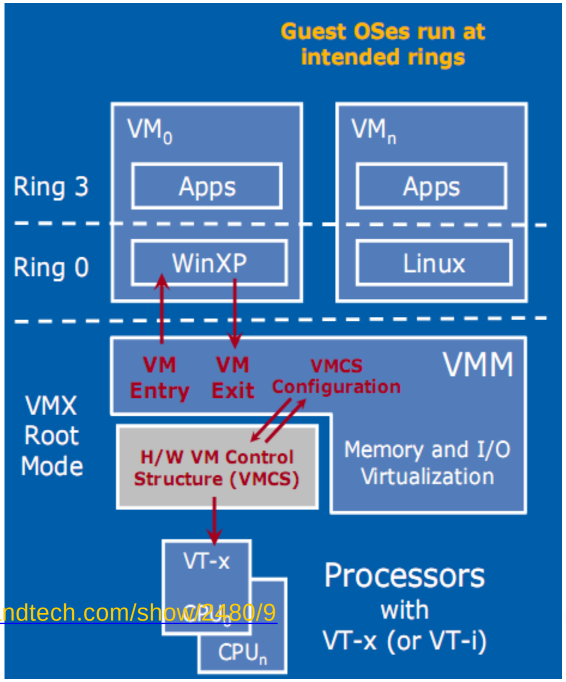
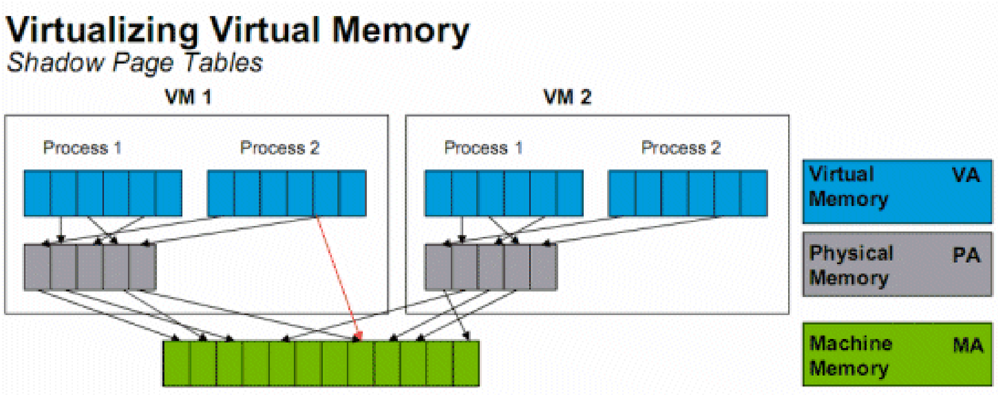


Image source: https://www.anandtech.com/show/2480/9

Memory Virtualization

- Virtual memory in OS
  - memory references from virtual space are translated to physical addresses
- How is it handled by VMM?
  - VMM maintains combined mapping of VA->PA->MA
  - Uses shadow page tables to accelerate the translation
- Hardware assistance
  - Extended page tables (EPT): MMU hardware is aware of virtualization,
  - takes pointers to two separate page tables
  - address translation walks both page tables



Ref: https://www.anandtech.com/show/2480/10

I/O Virtualization

- Guest OS needs to access I/O devices, but cannot give full control of I/O to any one guest OS
- Two main techniques for I/O virtualization
  - Emulation: guest OS I/O operations trap to VMM, emulated by doing I/O in VMM/host OS
  - Direct I/O: assign a slice of a device directly to each VM

Storage Virtualization

- Storage in form of clusters of disks are pooled and distributed across VMs
  - Physical storage devices are partitioned into *logical unit numbers* (*LUN*s)
  - LUNs are used to create a *datastore*
- VMs are stored as files (.*vmdk*) on datastore
- VM configuration files are stored as .*vmx* files
- Storage Area Network (SAN) uses a network-accessible device through a highspeed network connection to provide storage facilities

Network Virtualization

- Physical components that make up a network are virtualized
- Combine hardware and software network resources, as well as network functionality into a software-based virtual network

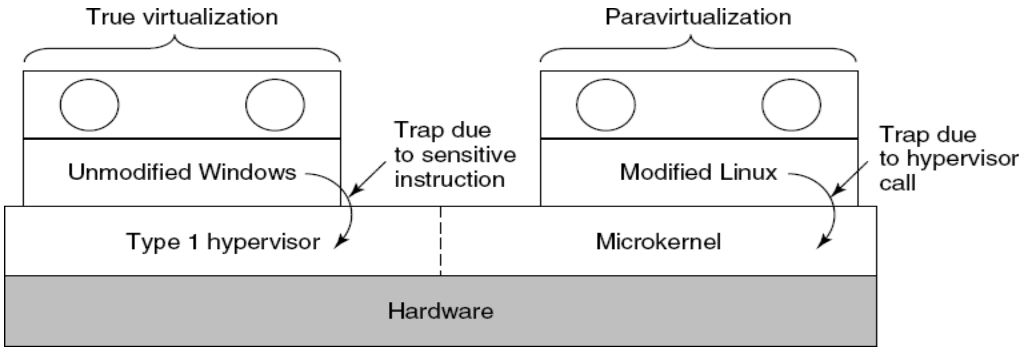
- External network virtualization
  - Combine many networks, or parts of networks, into a virtual unit (VLANs)
- Internal network virtualization
  - Provide network switch-like functionality to the VMs on a single system (vSwitch)

Network Virtualization

- Properties of virtual switch
- Virtual switch in a hypervisor facilitates multiple VMs to share a physical Ethernet network interface
  - Detects which VMs are logically connected to each of its virtual ports and uses that information to forward traffic to the correct virtual machines.

Types of VMs – Paravirtualization

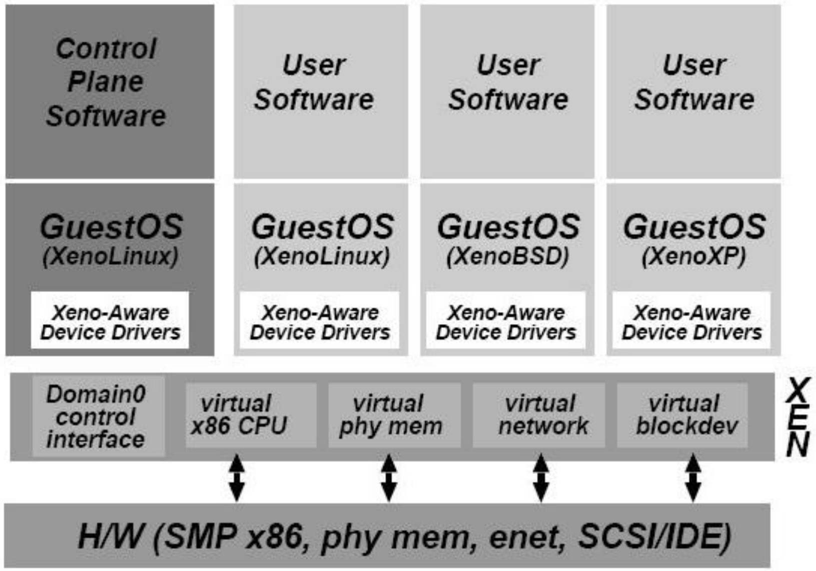
- Both type 1 and 2 hypervisors work on unmodified OS
- Para virtualization: modify OS kernel to replace all sensitive instructions with hyper calls
- OS behaves like a user program making system calls
- Hypervisor executes the privileged operations invoked by *hypercalls*.
- Less needed as hardware support for VMs grows



Source: Book by A.S. Tanenbaum, "Modern Operating Systems" 3e, Prentice Hall

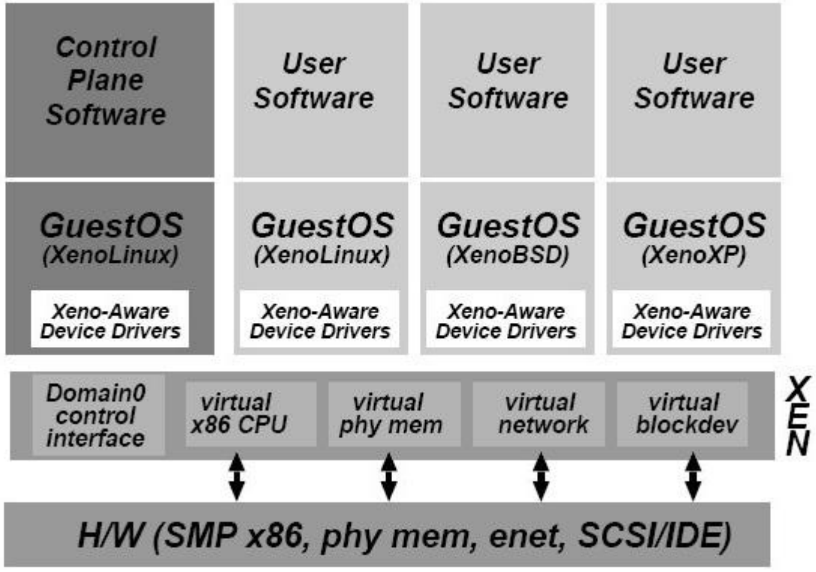
Xen Architecture

- Xen, leader in paravirtualized space, adds several techniques
- Runs directly over the hardware (even without virtualization support)
- Trap and emulate architecture
  - Xen runs in Ring 0, while guest OSes run in Ring 1
  - Guest OS traps to Xen for privileged instructions
- VM in which guest OS runs is termed as domain
  - Dom 0 is a special VM that runs the control/management software

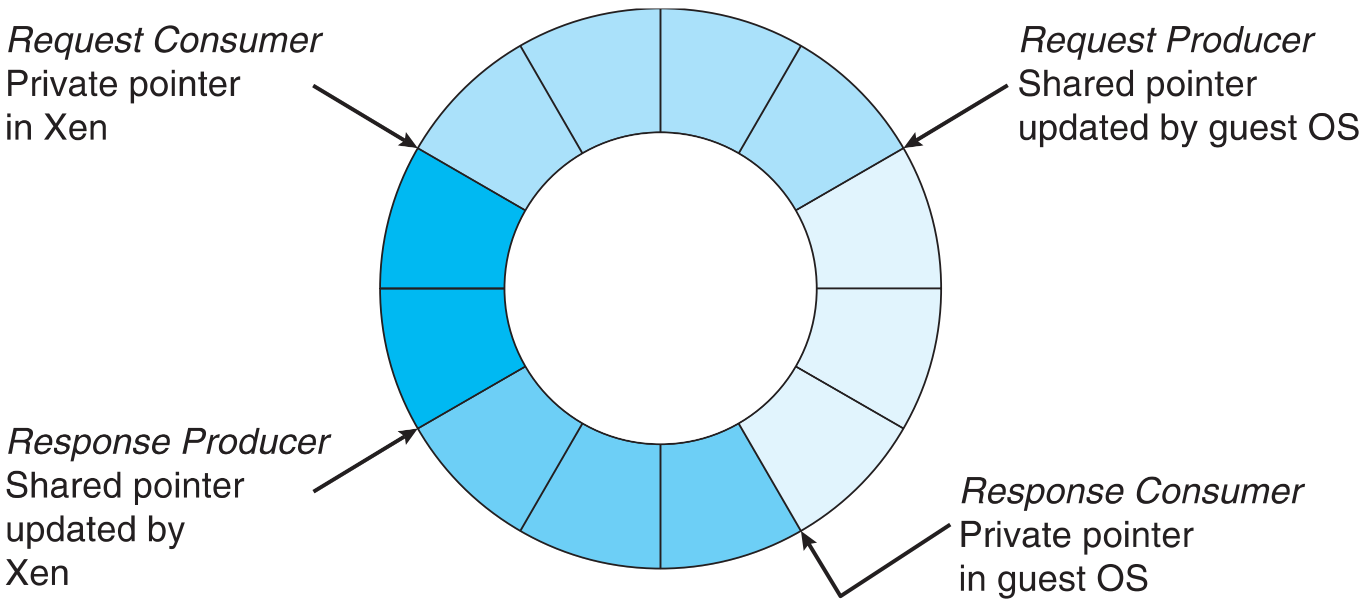


Xen Architecture

  - Guest OS pauses while hypercall is serviced
  - Minimize number of privilege transitions into Xen
- Xen sends asynchronous interrupts to Dom 0- avoids device interrupts
- Most frequent exceptions: system calls and page faults
  - Use ‘fast’ handler bypassing hypervisor
  - direct calls from an application into its guest OS avoids indirection through Xen
- I/O transfer done directly between guest OS and Dom 0



Xen I/O via Shared Circular Buffer



Request queue - Descriptors queued by the VM but not yet accepted by Xen

Outstanding descriptors - Descriptor slots awaiting a response from Xen

Response queue - Descriptors returned by Xen in response to serviced requests

Unused descriptors

Xen Architecture

- Memory management does not include nested page tables
  - Each guest has own read-only tables
  - Guest uses **hypercall** (call to hypervisor) when page-table changes needed
- Paravirtualization allowed virtualization of older x86 CPUs (and others) without binary translation
- Guest had to be modified to use run on paravirtualized VMM
- But on modern CPUs Xen no longer requires guest modification -> no longer paravirtualization

Virtualization and Operating-System Components

- Now look at operating system aspects of virtualization
  - CPU scheduling, memory management, I/O, storage, and unique VM migration feature
    - How do VMMs schedule CPU use when guests believe they have dedicated CPUs?
    - How can memory management work when many guests require large amounts of memory?

OS Component – CPU Scheduling

- Even single-CPU systems act like multiprocessor ones when virtualized
  - One or more virtual CPUs per guest
- Generally VMM has one or more physical CPUs and number of threads to run on them
  - Guests configured with certain number of VCPUs
    - Can be adjusted throughout life of VM
  - When enough CPUs for all guests -> VMM can allocate dedicated CPUs, each guest much like native operating system managing its CPUs
  - Usually not enough CPUs -> CPU **overcommitment**
    - VMM can use standard scheduling algorithms to put threads on CPUs
    - Some add fairness aspect

OS Component – CPU Scheduling (cont.)

- Cycle stealing by VMM and oversubscription of CPUs means guests don’t get CPU cycles they expect
    - Poor response times for users of guest
    - Time-of-day clocks incorrect
- Some VMMs provide application to run in each guest to fix time-of-day and provide other integration features

OS Component – Memory Management

- Also suffers from oversubscription -> requires extra management efficiency from VMM
- For example, VMware ESX guests have a configured amount of physical memory, then ESX uses 3 methods of memory management
  - Double-paging, in which the guest page table indicates a page is in a physical frame but the VMM moves some of those pages to backing store
  - Install a **pseudo-device driver** in each guest (it looks like a device driver to the guest kernel but really just adds kernel-mode code to the guest)
    - **Balloon** memory manager communicates with VMM to allocate or deallocate physical memory
  - De-duplication by VMM determining if same page loaded more than once, memory mapping the same page into multiple guests

OS Component – I/O

- Easier for VMMs to integrate with guests due to device drivers
- Some I/O complications in VMMs
  - Many short paths for I/O in standard OSes for improved performance
  - Possibilities include direct device access, DMA pass-through, direct interrupt delivery
    - Again, HW support needed for these
- Networking also complex as VMM and guests all need network access
  - VMM can **bridge** guest to network (allowing direct access)
  - And / or provide **network address translation** (**NAT**)
    - NAT address local to machine on which guest is running, VMM provides address translation to guest to hide its address

OS Component – Storage Management

- Both boot disk and general data access need be provided by VMM
- Need to support potentially dozens of guests per VMM (so standard disk partitioning not sufficient)
- Type 1 – storage guest root disks and config information within file system provided by VMM as a **disk image**
- Type 2 – store as files in file system provided by host OS
- Duplicate file -> create new guest
- Move file to another system -> move guest
- **File system across disks**
  - **Physical-to-virtual** (**P-to-V**) convert native disk blocks into VMM format
- VMM also needs to provide access to network attached storage (just networking) and other disk images, disk partitions, disks, etc

VM Migration

- Migrate an entire VM from one physical host to another
  - Typically within same virtualization layer (like Xen)
  - All user processes and kernel state
  - Without having to shut down the machine (live migration)

Why VM Migration?

- Why migrate VMs?
  - Distribute VM load efficiently across servers in a cloud
  - System maintenance (high availability)
- Easier than migrating processes
  - VM has a much thinner interface than a process
- Two main techniques: pre-copy and post copy

VM Migration – What is migrated?

- Migrate only CPU context of VM, contents of main memory
- Disk: assume NAS (network attached storage) that is accessible from both hosts, or local disk is mirrored
  - We do not consider migrating disk data
- Network: assume both hosts on same LAN
  - Network packets redirected to new location (with transient losses)
- I/O devices are provisioned at target
  - Virtual I/O devices easier to migrate

Steps to Migrate a VM

- Suppose we are migrating a VM from host A to host B

1. Setup target host B, reserve resources for the VM
2. Push phase: push some memory pages of VM from A to B
3. Stop-and-copy: stop the VM at A, copy CPU context, and some memory pages
4. Pull phase: Start VM at host B, pull any further memory pages required from A
5. Clean up state from host A, migration complete

- Total migration time : time for steps 2,3,4
- Service downtime : time for step 3

Challenges in VM migration

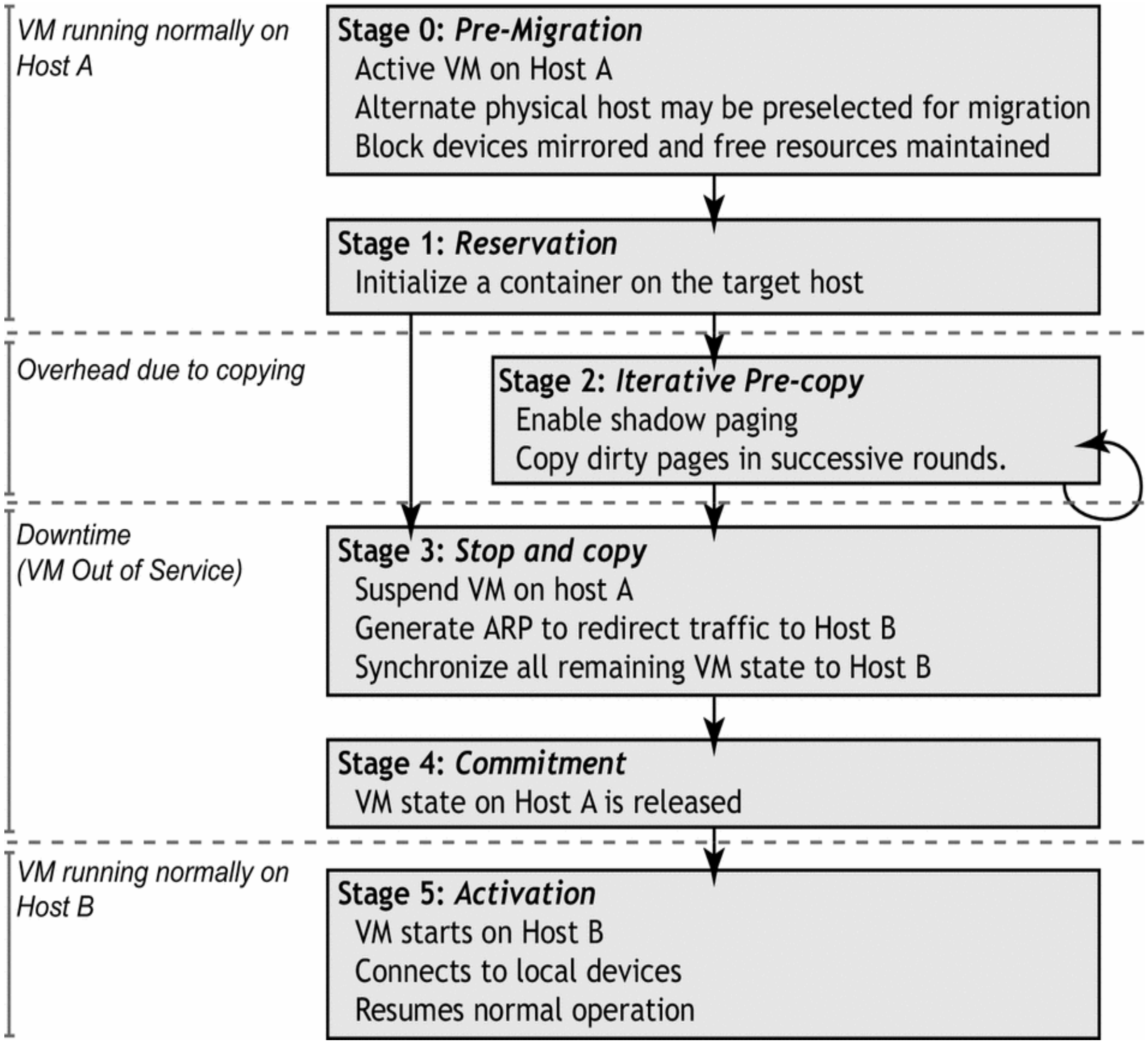
- VMs have lots of state in memory
- Some VMs have soft real-time requirements
  - e.g., web servers, databases, game servers
  - May be members of a cluster quorum
  - Minimize down-time
- Performing relocation requires resources

Popular approaches

- Pre-copy : most state is transferred in the push phase, followed by a brief stop and copy phase

- Post copy : VM stopped, bare minimum state required to run the VM is transferred to the target host. Remaining state is pulled on demand while the VM is running at the new location.

Pre-copy Live VM migration in Xen



Issues with VM-based Applications

- Each VM in a virtualized platform (VMM) stills requires
  - CPU allocation
  - Storage
  - RAM
  - An entire guest operating system replica
- The more VMs you run, the more resources you need
  - Guest OS means wasted resources
- Application portability not guaranteed across hypervisors

VM-based application deployment issues

- User application contains components with different requirements, in terms of libraries, runtimes, kernel features
- Applications in VM are coupled to the version of host OS
- Scaling of application is related to scaling of VM
- Application developer also becomes system administrator
- Solution: Use OS level virtualization construct - Containers

Operating System Level

- To create different and separated execution environments for applications that are managed concurrently in a safe manner
- No VMM or hypervisor required
- Virtualization is within a single OS
  - OS kernel allows for multiple isolated user space instances.
  - OS kernel is responsible for sharing the system resources among instances.
- A user space instance has an independent view of the file system (isolated), separate IP addresses, software configurations, and access to devices.

Operating System Level

- Uses constructs like namespaces and cgroups in Linux
- Compared to H/W virtualization, no overhead because applications directly use OS system calls and there is no need for emulation.
- No need to modify applications nor any specific hardware
- e.g., OpenVZ, Solaris Zones, Containers (LXC, Docker),

Containers

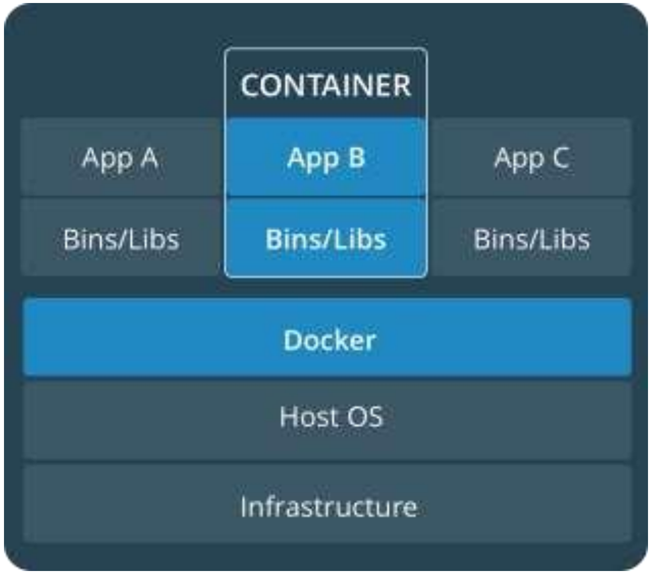
- What is the problem we are trying to solve?
- Different cloud payloads like applications and programming environments
  - e.g., webapps, distributed stores, databases,
  - e.g., Go, Python, Node.js, Ruby
- Different target operating systems in use
  - Linux, BSD, Windows
- Different environments for deployment and testing
  - Own and team's dev. environment, Staging server, production server, bare metal, VMs,

- Analogy: Containers used in shipping

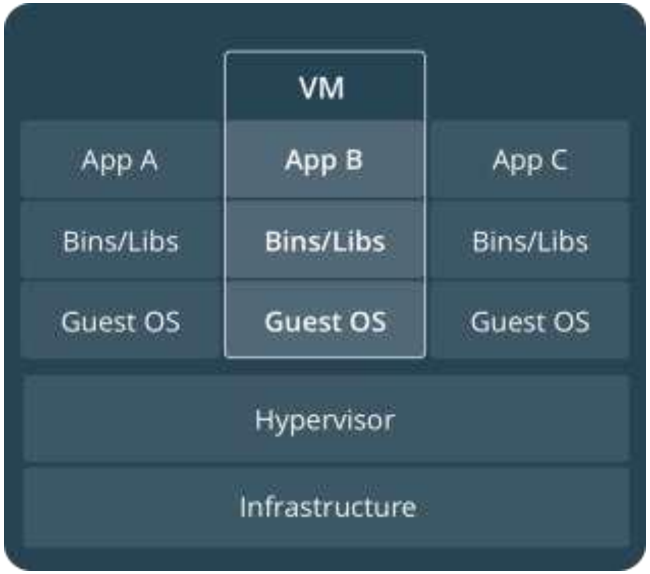
Containers

- Container is an isolated execution environment created to package
  - code, libraries, package manager, app, data
- Outside the container is everything that sysadmin requires
  - logging, remote access, system config., network config., monitoring
- Linux containers
  - Run everywhere regardless of kernel version, host distribution
  - Only container and host architecture need to match
  - Run anything that can run on kernel, if packaged in container
- Simply, it is a lightweight VM
  - own process space, network interface, root privileges, file system

Comparing Containers and VMs



Containers are an app level construct



VMs are an infrastructure level construct to turn one machine into many servers

Containers

- Containers share base OS, but have different libraries, utilities, root filesystem, view of processes, etc.
- VMs have different copies of the entire OS and all components
- Linux Containers are built with two constructs
  - Namespaces: a way to provide isolated view of a certain global resource to a set of processes, for e.g., root file system.
    - Mount, PID, Network namespaces
  - Cgroups: a way to set resource limits on a group of processes
- The two constructs allow isolation of processes into a bubble and set resource limits

Containers and Orchestration

- Container implementations like LXC and Docker use the constructs to build container abstraction
  - Docker is optimized for a single application, while LXC is a general container
- Frameworks like Kubernetes help in lifecycle management and autoscaling of containers across hosts

Docker

- Docker is a platform for developing, shipping & running application using container-based virtualization technology
- Allows you to separate applications from infrastructure to deliver software quickly
- An implementation of the container idea, a package format, resource isolation, an ecosystem for execution
- Allows quick provisioning using copy-on-write constructs
- Allows to create and share images across the ecosystem

Docker Architecture

- Docker uses client-server architecture
- Docker client: primary way for users to interact, issues commands to docker daemon (through CLI)
- Docker daemon: listens to requests and manages objects
- Docker objects: images, containers, networks, volumes etc.
- Docker hub: public registry to store docker images
- Docker image: a read-only template with instructions for creating a Docker container.
- Docker container: a runnable instance of image, defined also by configuration

Docker image

- Docker image is a binary with all the requirements to run a container, as well as metadata describing its needs and capabilities
- Images are read only containers used to create containers
- Image contains software you wish to run
- Images are stored in hub

Docker file

- Docker file
  - text document with all the commands a user could call on the CLI to assemble an image
- Docker file is used for automation of work by specifying the steps we need on an image

- Each instruction in a Dockerfile creates a layer in the image.
- When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt

Docker Engine

- Docker Engine is the program that enables containers to build, shipped and executed
- A client-server application with
  - Docker daemon (*dockerd*) taking requests from docker client through API
  - Command line interface client *docker*
- Docker Engine uses Linux kernel namespace & control groups

How does Docker work?

- You can build Docker images that hold your applications
- You can create Docker containers from those Docker images to run your applications
- You can share those Docker images via Docker Hub or your own registry
  - official repos are available at https://hub.docker.com/explore/

Container Orchestration

- Automation of the operational efforts required to run containerized workloads and services
- Functions required for container lifecycle management
  - provisioning, deployment, scaling, networking, load balancing etc.
- Docker Swarm is Docker’s own orchestration tool
- Amazon Elastic Container Service (ECS)
  - Easily deploy, manage, and scale containerized applications
- Kubernetes is open source container orchestration engine for automating deployment, scaling, and management of containerized application

End of Virtualization