TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Virtualization

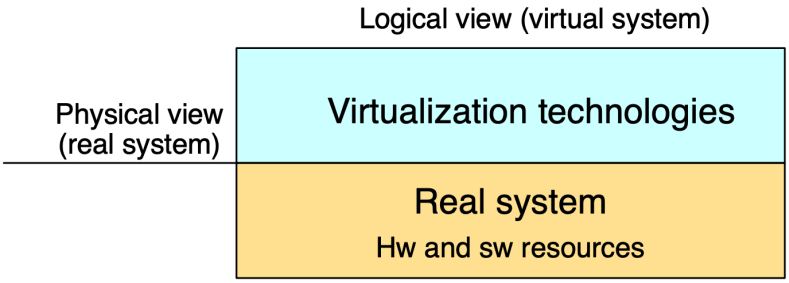
Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2023/24

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Virtualization

- High-level abstraction to hide details of underlying implementation
- Abstraction of computing resources
  - Logical view different from physical one



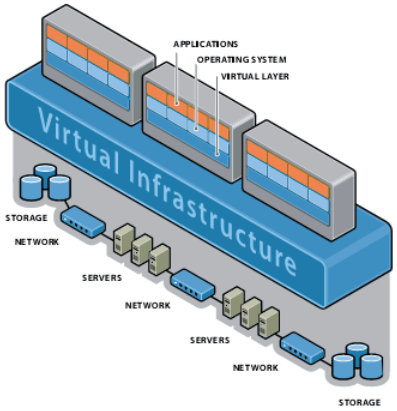
- How? Decouple user-perceived architecture and behavior of hw and sw resources from their physical realization
- Goals:
  - Agility, flexibility, performance, reliability, security, …

Virtualization of resources

- System (hw and sw) resources virtualization
  - Virtual machines, containers, unikernels, …

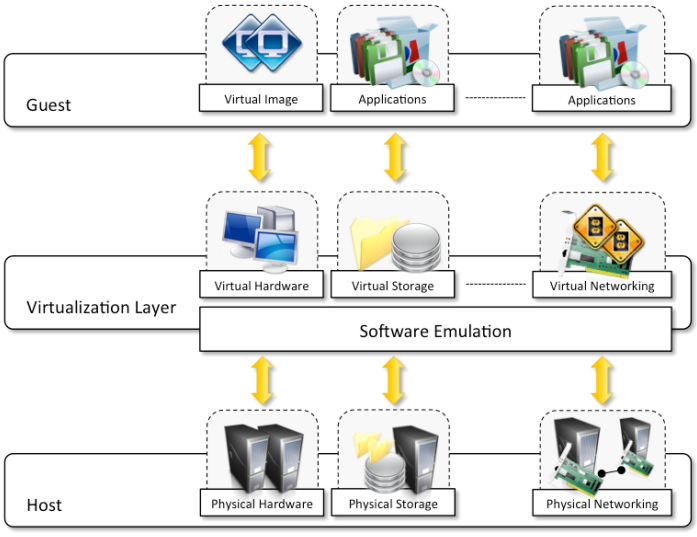
Our focus

- Storage virtualization
  - Storage Area Network (SAN), …
- Network virtualization
  - Virtual LAN (VLAN), Virtual Private Network (VPN), …
- Data center virtualization



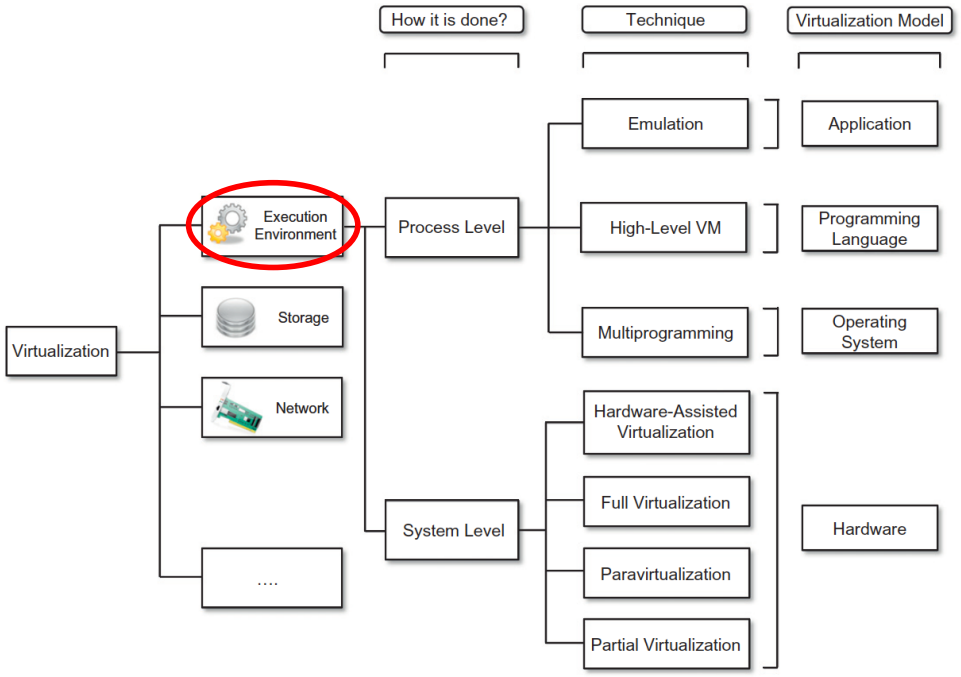
Components of virtualized environment

- 3 major components:
  - Guest
  - Host
  - Virtualization layer
- **Guest**: interacts with virtualization layer rather than with host
- **Host**: original environment where guest is supposed to be managed



- **Virtualization layer**: responsible for recreating same or different environment where guest will operate

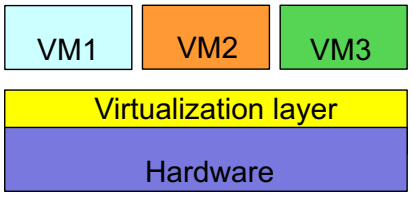
Taxonomy of virtualization techniques



- **Execution environment virtualization** is the oldest, most popular and developed area ⇒ our focus

Virtual Machine

- A **virtual machine** (**VM**) is a complete compute environment with its own isolated processing capabilities, memory, and communication channels
- Allows to represent hw/sw resources of a physical machine differently from their reality
  - E.g., VM hw resources (CPU, network card, ...) different from physical resources of the real machine
  - E.g., VM sw resources (OS, …) different from sw resources of the real machine
- A single physical machine can be used to host several VMs



Virtualization: a brief history

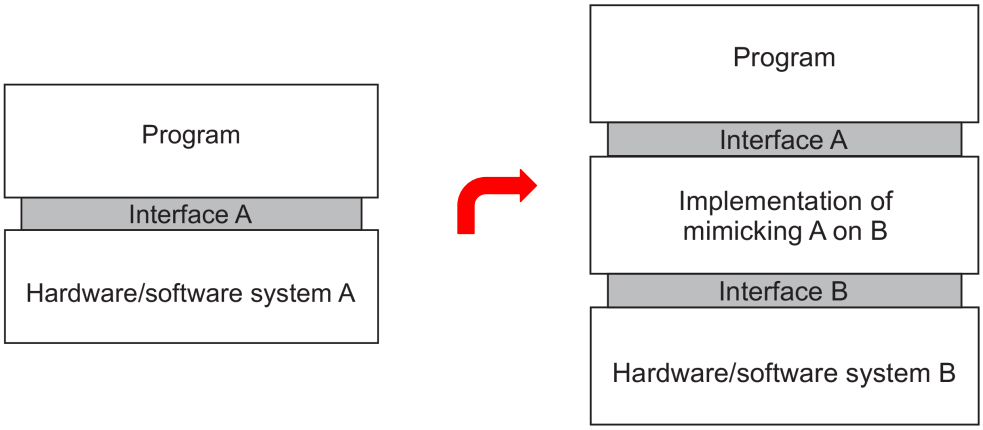
- Virtualization and VMs are an “old” idea in computer science
  - Dates back to the 1960s in a centralized context
  - Designed to allow legacy (existing) software to run on expensive mainframes and transparently share (scarce) physical resources
  - E.g., IBM System/360-67 mainframe
- In the 1980s, with the transition to PCs, the problem of transparently sharing computing resources was solved by multitasking OSs
  - Virtualization became less of an issue

Virtualization: a brief history

- At the end of the 1990s, interest in virtualization revamped to make programming special-purpose hw less burdensome
  - VMware founded in 1998
- Moreover, management costs and under-utilization of hw and sw platforms exacerbate the need for virtualization solutions
  - Hw changes faster than sw (middleware and applications)
  - Management costs increase and application portability decreases
  - Sharing underutilized computing resources becomes important again to reduce infrastructure costs
- Nowadays, virtualization is one the enabling technologies for cloud computing

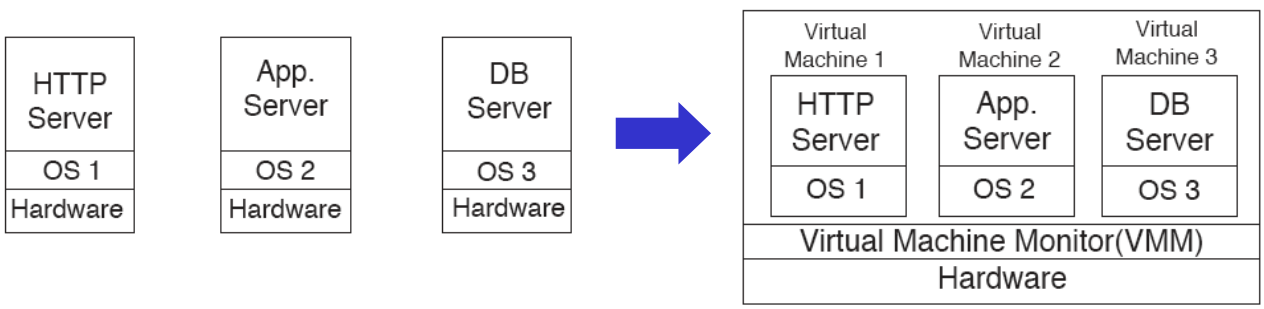
Virtualization: pros

- Facilitates compatibility, portability, interoperability and migration of applications and environments
  - Hw independence: create once, run everywhere
  - Legacy VMs: run old OSs or applications on new platforms



Virtualization: pros

- Allows server consolidation in data center, with economic, management and energy advantages
  - How? Multiplexing multiple VMs on same physical server
  - Goal: reduce number of physical servers and use them efficiently
  - ✓ Reduce costs, energy consumption and occupied space
  - ✓ Simplify server management, maintenance and upgrade
  - ✓ Reduce downtime through live migration of VMs



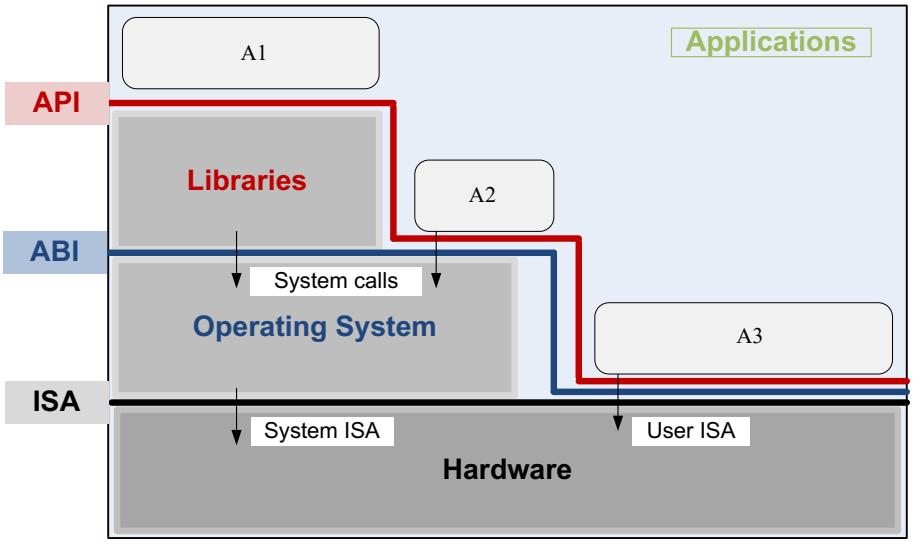
Virtualization: pros

- Allows to isolate application components that are malfunctioning or under security attacks, thus increasing applications reliability and security
  - VMs running different components cannot access each other's resources
  - Software bugs, crashes, viruses in a VM cannot harm other VMs running on the same physical machine
- Allows to isolate performance of different VMs
  - By scheduling shared physical resources among different VMs running on the same physical machine
- Allows to balance load on physical machines
  - By migrating VMs from a physical machine to another

Reasons to use virtualization

- Personal and educational
  - Run several OSs simultaneously on the same physical machine
  - Simplify sw installation
  - Develop, test and debug applications
  - Simulate a distributed environment on a single machine
- Enterprise
  - Consolidate data center infrastructure and ensure business continuity
  - Encapsulate entire systems in single files (system images) that can be replicated, migrated or reinstalled on any server
  - Enable DevOps

Interfaces in computer system



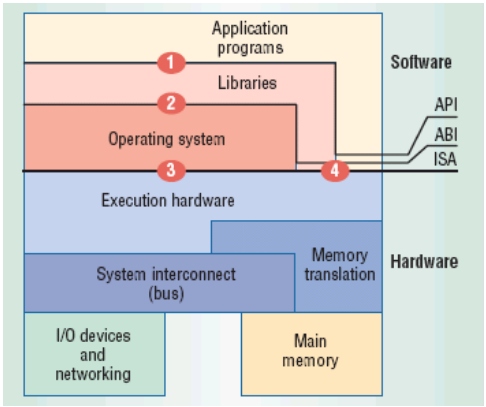
Applications:

- use library functions (A1)
- make system calls (A2)
- execute machine instructions (A3)

Interfaces in computer system and virtualization

At which level can virtualization be realized?

- Strictly related to computer system interfaces
  - Hw/sw interface (**system ISA**: primarily for system resource management, *privileged* instructions executed only by OS) [interface 3]
  - Hw/sw interface (**user-level ISA**: primarily for computation, *non-privileged* instructions executed by any program) [interface 4]
  - System calls [interface 2]
    - **ABI** (Application Binary Interface): interface 2 + interface 4
  - Library calls (**API**) [interface 1]



- Essence of virtualization: mimic behavior of these interfaces

Smith and Nair, The architecture of virtual machines, IEEE Computers, 2005

Implementation levels of virtualization

- Virtualization can be implemented at various operational levels:
  - ISA level
  - **Hardware level** (aka *system VMs*)
  - **Operating system level** (aka *containers*)
  - Library level
  - User application level (aka *process VMs*)

Our focus

Implementation levels of virtualization

- ISA level
  - Goal: emulate a given ISA by ISA of host machine
    - E.g., MIPS binary code can run on x86-based host with help of ISA emulation
  - ISA emulation can be done through *code interpretation* or *dynamic binary translation*
    - Code interpretation is slow: every source instruction is interpreted by emulator in order to execute native ISA instructions
    - Dynamic binary translation is faster: converts in blocks rather than instruction by instruction

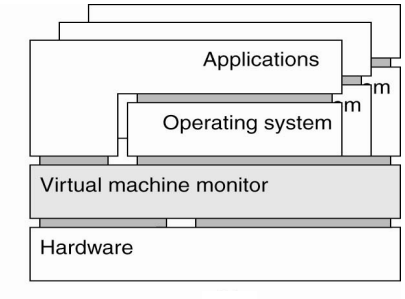
Implementation levels of virtualization

- Hardware level (aka *system VMs*)
  - Goal: virtualize host resources, such as its processors, memory, and I/O devices
  - Based on **Virtual Machine Monitor** (**VMM**), aka *hypervisor*
    - VMM handles interaction with underlying hw platform for CPU, memory, and I/O resource access

Implementation levels of virtualization

- Hardware level (aka *system VMs*)
  - Provides a complete environment in which multiple VMs can coexist
    - VMM manages hardware resources and shares them among multiple VMs and provide isolation and protection of VMs
    - When a VM performs a privileged instruction or operation that directly interacts with shared hw, VMM intercepts the instruction, checks it for correctness, and performs it
  - Examples: VMware, KVM, Xen, Parallels, VirtualBox

Multiple instances of combinations <applications, OS>



Implementation levels of virtualization

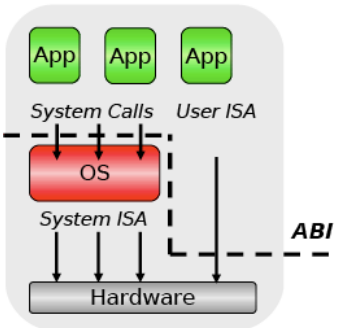
- Operating system level (aka *containers*)
  - Goal: create multiple isolated containers
  - Examples: Docker, Linux Containers, Podman

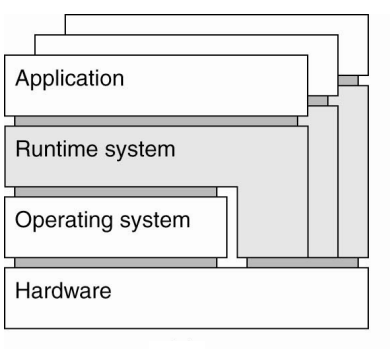
- Library level
  - Goal: create execution environment to run apps in a host environment that does not suite native apps
    - Rather than creating a VM to run full OS and apps
  - Examples:
    - Wine: runs Windows apps on top of POSIX-compliant OS by translating Windows API calls into POSIX calls on-the-fly
    - Cygwin: "Get that Linux feeling – on Windows"

Implementation levels of virtualization

- User application level (aka *process VMs*)
  - Virtual platform that executes a single process
  - Provides virtual ABI or API to user application
  - Application is compiled into intermediary, portable code (e.g., Java bytecode) and executed in runtime environment provided by process VM
  - Examples: JVM, .NET CLR

Multiple instances of combinations <application, runtime system>





Implementation levels of virtualization: summing up

- Relative merits of virtualization at different levels

Level of Virtualization	Functional Description	Example Packages	Merits, App Flexibility/ Isolation, Implementation Complexity
Instruction Set Architecture	Emulation of a guest ISA by host	Dynamo, Bird, Bochs, Crusoe	Low performance, high app flexibility, median complexity and isolation
Hardware-Level Virtualization	Virtualization on top of bare-metal hardware	XEN, VMWare, Virtual PC	High performance and complexity, median app flexibility, and good app isolation
Operating System Level	Isolated containers of user app with isolated resources	Docker Engine, Jail, FVM	Highest performance, low app flexibility and best isolation, and average complexity
Run-Time Library Level	Creating VM via run-time library through API hooks	Wine, vCUDA, WABI, LxRun	Average performance, low app flexibility and isolation, and low complexity
User Application Level	Deploy HLL VMs at user app level	JVM, .NET CLR, Panot	Low performance and app flexibility, very high complexity and app isolation

Level of Implementation	Higher Performance	Application Flexibility	Implementation Complexity	Application Isolation
Instruction Set Architecture (ISA)	X	XXXXX	XXX	XXX
Hardware-Level Virtualization	XXXXX	XXX	XXXXX	XXXX
Operating System Level	XXXXX	XX	XXX	XX
Run-Time Library Support	XXX	XX	XX	XX
User Application Level	XX	XX	XXXXX	XXXXX

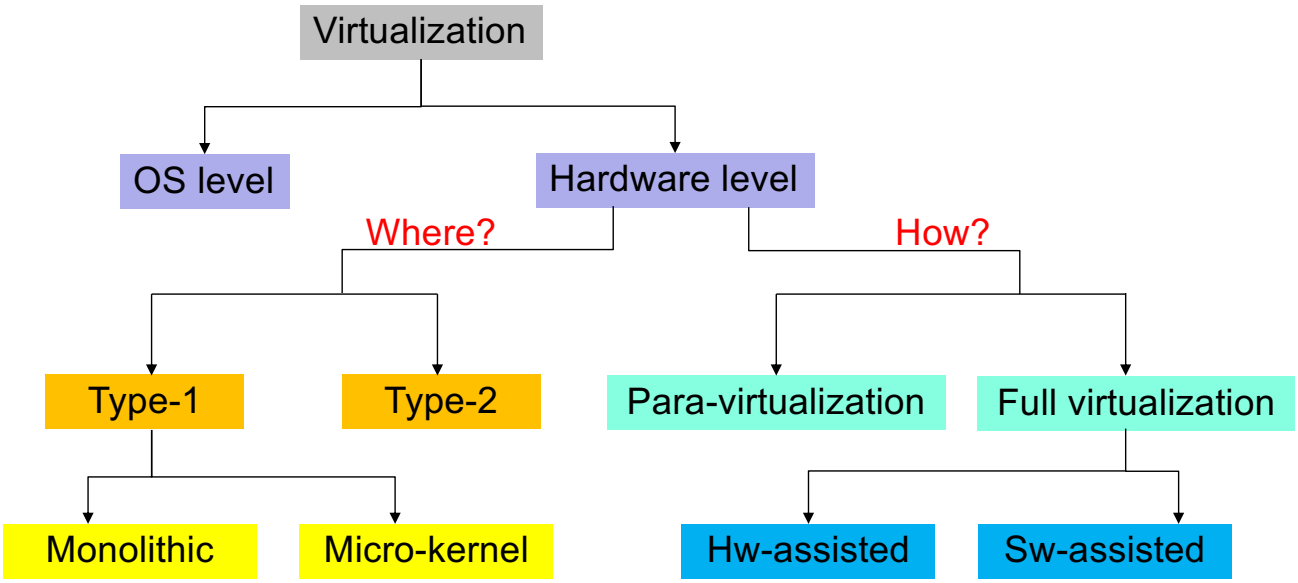
System-level virtualization: terminology

- Let's focus on **system-level virtualization** (achieved through VMM or hypervisor)
- **Host**: base platform on top of which VMs are executed; made of:
  - Physical machine
  - *Possible* host OS
  - VMM
- **Guest**: everything inside a single VM
  - Guest OS and applications executed inside the VM

System-level virtualization: taxonomy

- Let's classify system-level virtualization solutions according to:
  1. Where to deploy VMM
     - **System VMM** (aka *type-1*, *native* or *bare-metal* hypervisor)
     - **Hosted VMM** (aka *type-2* hypervisor)
  2. How to virtualize instruction execution
     - **Full virtualization**
       - *Software-assisted*
       - *Hardware-assisted*
     - **Paravirtualization**

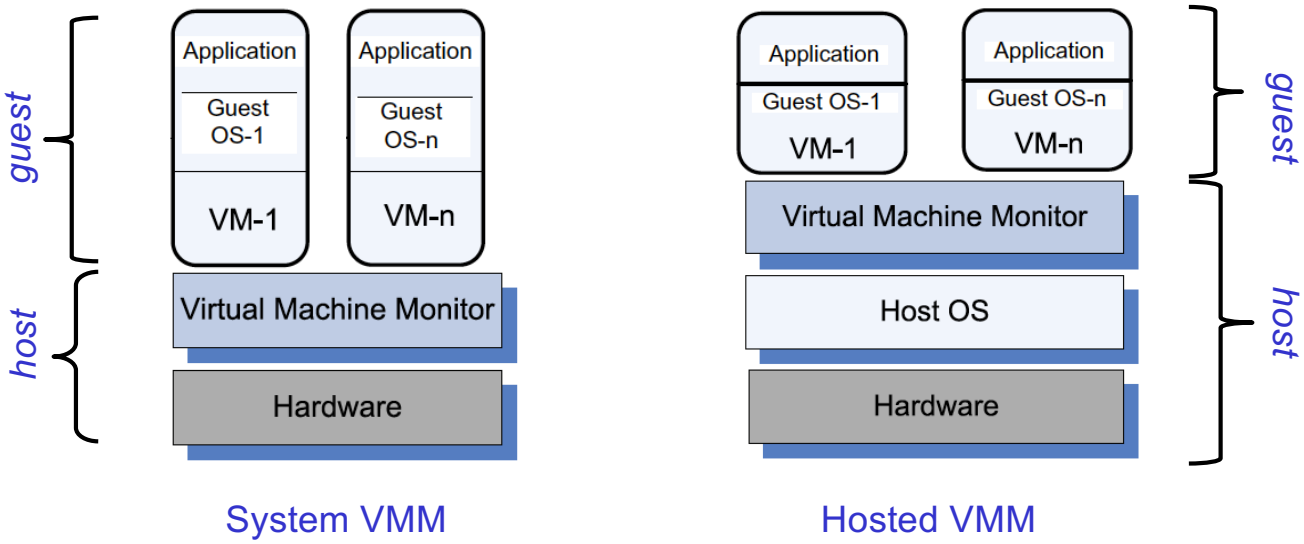
System-level virtualization: taxonomy



System vs. hosted VMM

In which level of the system architecture is VMM deployed?

- Directly on hardware: **system (or native) VMM**
- On top of host OS: **hosted VMM**



System VMM Hosted VMM

System vs. hosted VMM

- **System VMM** (*type-1*): runs directly on hw, offers virtualization features integrated into a simplified OS
  - VMM can have *microkernel* (only basic functions, no device drivers) or *monolithic* architecture
  - Examples: Xen, KVM, VMware ESXi, Hyper-V
- **Hosted VMM** (*type-2*): runs on top of host OS, accesses hw resources via host OS system calls
  - Interacts with host OS via ABI and emulates virtual hw ISA for guest OS
  - ✓ Can use host OS to manage devices and use low-level services (e.g., resource scheduling)
  - ✓ No need to change guest OS
  - ✗ Performance degradation with respect to system VMM
  - Examples: Bochs, Parallels Desktop, VirtualBox

Full virtualization vs paravirtualization

How to manage the interaction between VMs and VMM in order to access to physical resources, i.e., how to manage the execution of privileged instructions that require direct access to hardware or other privileged resources?

- **Full virtualization**
- **Paravirtualization**

- For a comparison of platform virtualization software
  en.wikipedia.org/wiki/Comparison_of_platform_virtualization_software

Full virtualization vs paravirtualization

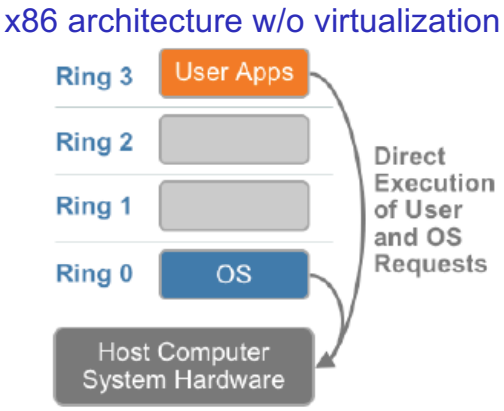
- Full virtualization
  - VMM exposes to each VM simulated hw interfaces that are *functionally identical* to those of the underlying physical machine
  - VMM intercepts requests for privileged access to the hardware (e.g., I/O instructions) and emulates the expected behavior
  - Examples: KVM, VMware ESXi, Microsoft Hyper-V
- Paravirtualization
  - The VMM exposes to each VM simulated hw interfaces that are *functionally similar* (but not identical) to those of the underlying physical machine
  - Hardware is not emulated, but a minimal software layer (Virtual Hardware API) is created to ensure VM management and their isolation
  - Examples: Xen, Oracle VM, PikeOS

Full virtualization vs paravirtualization

- Full virtualization pros and cons
  - ✓ Run unmodified guest OS
  - ✓ Complete isolation between VM instances: security, ease of emulating different architectures
  - ✗ VMM is more complex
  - ✗ Require collaboration with processor to make virtualization more efficient: why?

Issues to address for system-level virtualization

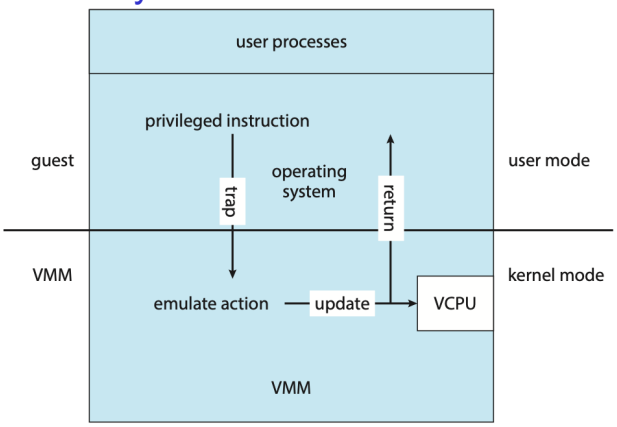
- Non-virtualized processor architecture operates according to at least 2 protection levels (*rings*): supervisor and user
  - Ring 0: most privileged (unrestricted access to system resources)
  - Ring 3: least privileged



- With virtualization
  - VMM operates in supervisor mode (ring 0)
  - Guest OS and applications (i.e., VM) operate in user mode (guest OS in ring 1 or 3)
  - **Ring deprivileging** problem: guest OS operates in a ring which is not its own ⇒ cannot execute privileged instructions (e.g., `lidt` in x86 to load interrupt descriptor table)
  - **Ring compression** problem: since applications and guest OS run at the same level, guest OS space must remain protected

How to address ring deprivileging

- **Trap-and-emulate**
  - When guest OS attempts to execute a privileged instruction (which can run only in supervisor mode), an exception (trap) must be notified to VMM and control must be transferred to it; VMM performs a safety check on the requested operation, executes ("emulates") its behavior and returns the result to guest OS
  - Instead non-privileged instructions (all?) run by guest OS do not trap and are directly executed



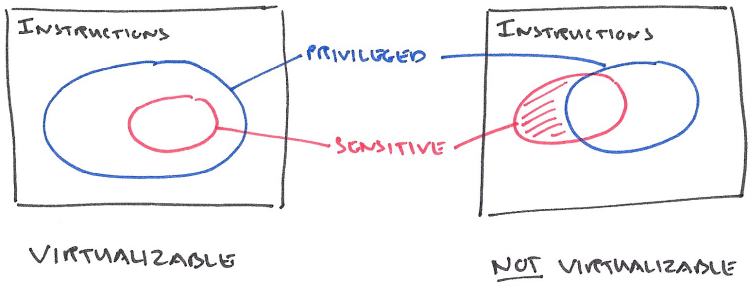
Popek and Goldberg virtualization requirements

- Popek and Goldberg (1974) defined a set of conditions sufficient for a computer architecture to support system virtualization *efficiently*
- They classified ISA instructions into 3 groups:
  1. *Privileged instructions*: do not trap when the processor is in supervisor mode, but trap when in user mode
     - Privileged state: determines resource allocation (privilege mode, addressing context, exception vectors, ...)
  2. *Sensitive instructions*: change underlying resources (e.g., do I/O or change page tables) or observe information that indicates current privilege level (thus exposing that guest OS does not run on bare metal); can be
     - Control sensitive: change privileged state
     - Behavior sensitive: expose privileged state
  3. *Innocuous instructions*: not sensitive

Popek and Goldberg virtualization requirements

- Necessary condition: *For any conventional computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions*

- Problem: condition is not always satisfied
  - There may be sensitive but non-privileged instructions that are executed in user mode without causing a trap to OS



blog.acolyer.org/2016/02/19/formal-requirements-for-virtualizable-third-generation-architectures

Condition for virtualization

- Common architectures are non-virtualizable according to Popek and Goldberg's condition
  - x86: many instructions are non-virtualizable, because are sensitive but non-privileged
    - E.g., `pushf` (push flags) is non-privileged
  - MIPS: mostly virtualizable, but...
    - Kernel registers `$k0`, `$k1` (needed to save/restore state) are user-accessible
  - ARM: mostly virtualizable, but
    - Some instructions are undefined in user-mode

Condition for virtualization

- From Popek and Popek and Goldberg's condition:
  - Privileged and sensitive but non-privileged instructions that are executed in user mode must be virtualized
- Issue:
  - Privileged instructions result in traps: ok
  - Sensitive but non-privileged do not result in traps, how can we virtualize them?
- *1st solution:* trap-and-emulate
  - Privileged and non-privileged sensitive instructions trap and divert control to VMM
  - Seems easy but … how to implement it?
- *2nd solution*: paravirtualization
  - Modify guest OS, by either preventing non-privileged sensitive instructions or making them non-sensitive (i.e., changing the context)

Full virtualization: solutions

- How to realize the trap mechanism?
  - At hardware level if processor supports virtualization

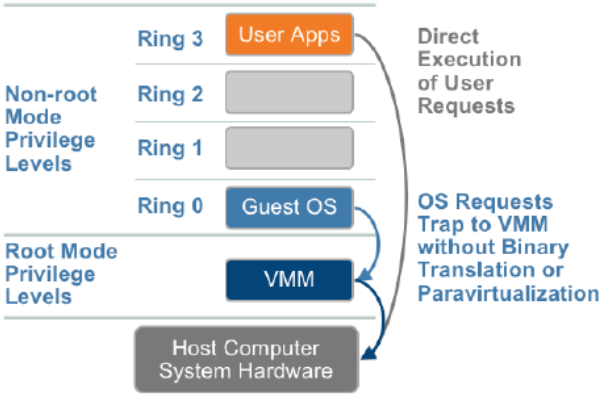
    👉 **hardware-assisted CPU virtualization**
  - At software level if processor does not support virtualization

    👉 **fast binary translation**
    - The elder solution

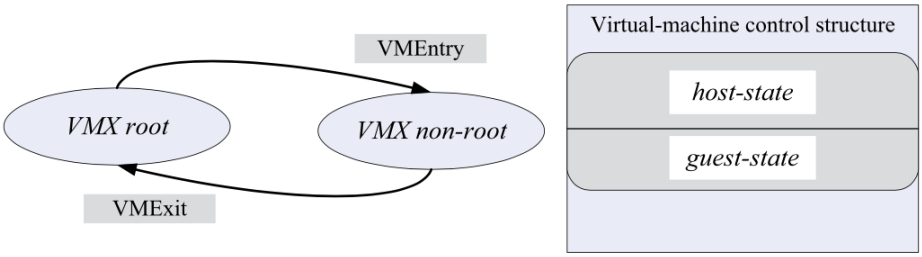
Hardware-assisted CPU virtualization

- Hardware-assisted CPU virtualization (Intel VT-x and AMD-V) provides two new CPU operating modes (root mode and non-root mode), each supporting all 4 x86 protection rings
  - VMM runs in root mode (Root-Ring 0), while guest OSs run in guest mode in their original privilege levels (Non-Root Ring 0): no longer ring deprivileging and ring compression issues
  - VMM can control guest execution through VM control data structures in memory

x86 architecture with full virtualization and *hardware-assisted CPU virtualization*



Hardware-assisted CPU virtualization: VT-x

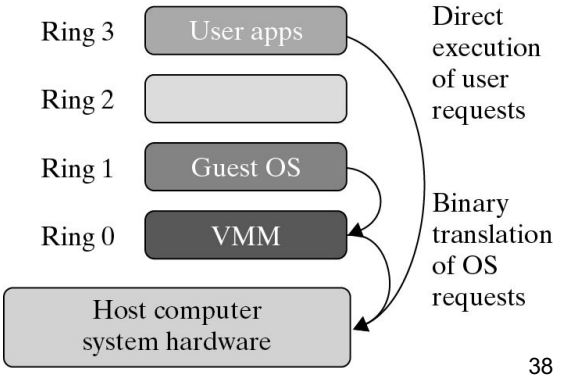


- VMX root: intended for hypervisor operations (like x86 without VT-x)
- *VMX non-root:* intended to support VMs
- When executing *VMEntry* operation, processor state is loaded from *guest-state* of VM scheduled to run, then control is transferred from hypervisor to VM
- *VMExit* saves processor state in *guest-state* area of running VM; it loads processor state from *host-state*, then transfers control to hypervisor

Fast binary translation

- VMM trap mechanism for privileged instructions is offered by processors with hardware support for virtualization
  - How to achieve full virtualization without hw support?
- Fast binary translation: VMM scans code before its execution to replace blocks containing privileged instructions with functionally equivalent blocks containing instructions for notifying exception to VMM
  - Translated blocks are directly executed on hw and stored in a cache for future reuse
  - ✗ Higher complexity and lower performance wrt to hw-assisted virtualization

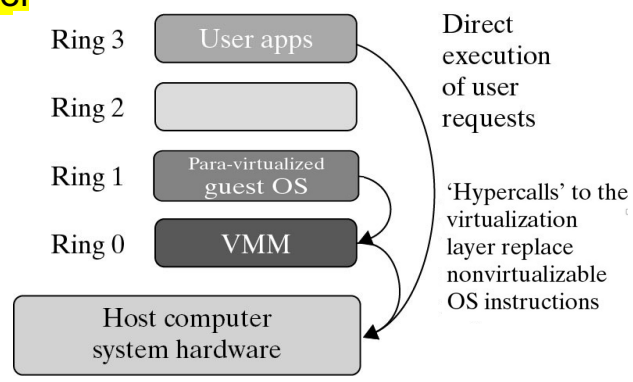
x86 architecture with full virtualization and fast binary translation



Paravirtualization

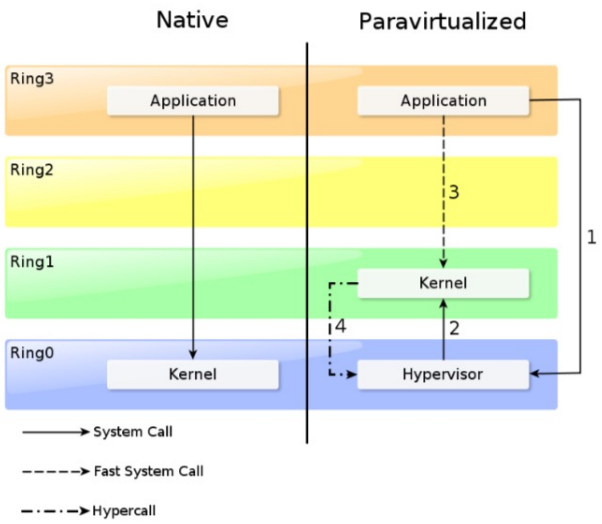
- Non-transparent virtualization solution
  - Guest OS kernel must be modified to let it invoke the virtual API exposed by VMM
- Non-virtualizable instructions are replaced by *hypercalls* that communicate directly with hypervisor
  - Hypercall: *software trap* from guest OS to hypervisor, just as syscall is software trap from app to kernel

hypercall : hypervisor = syscall : kernel



x86 architecture with *paravirtualization*

Paravirtualization: hypercall execution



- When application running in VM issues a guest OS system call, through the hypercall the control flow jumps to hypervisor, which then passes control back to guest OS

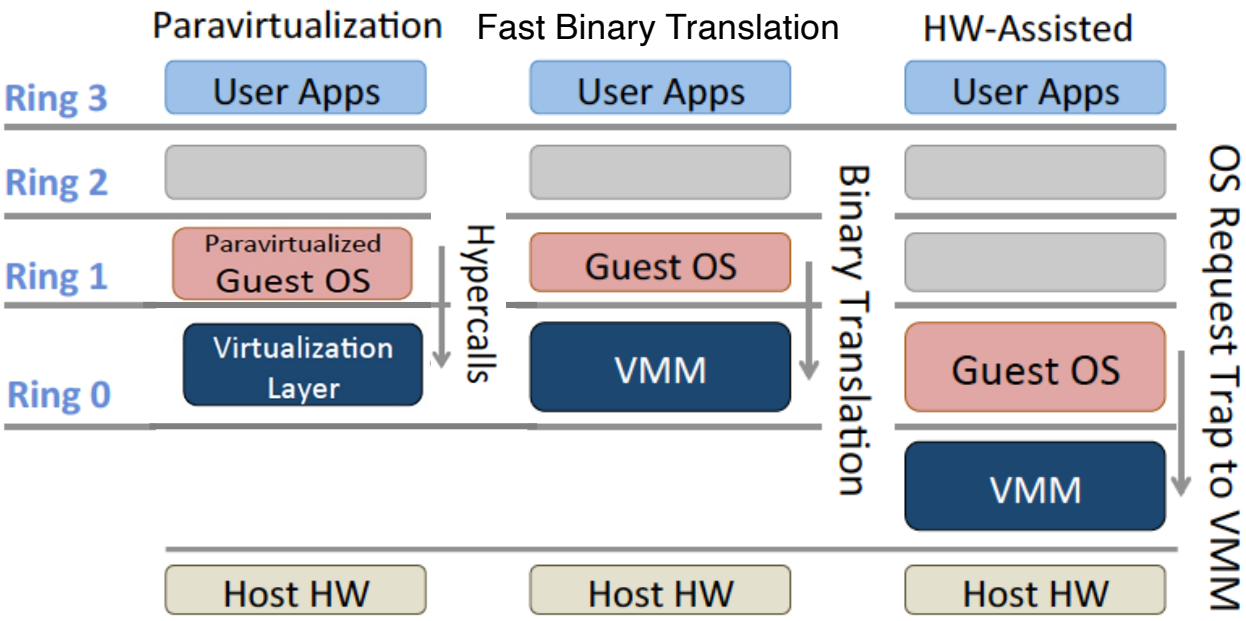
Source: “The Definitive Guide to XEN hypervisor”

Paravirtualization: pros & cons

- Pros (vs full virtualization):
  - ✓ Relatively easier and more practical implementation
  - ✓ Less overhead wrt fast binary translation
  - ✓ Does not require virtualization extensions from host CPU as hw-assisted virtualization does

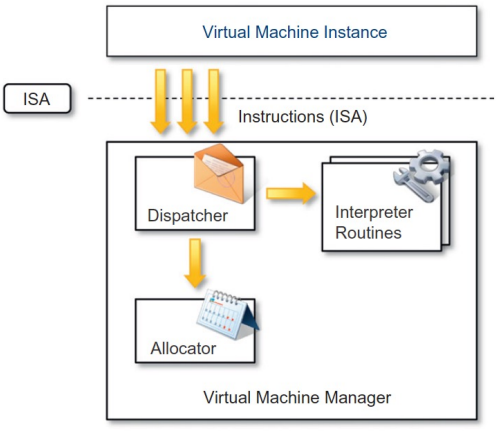
- Cons (vs full virtualization):
  - ✗ Requires source code availability of OS to modify guest OS and make it paravirtualized
  - ✗ Cost of maintaining paravirtualized OSs
    - Paravirtualized OS cannot run directly on hardware

Summing up different approaches



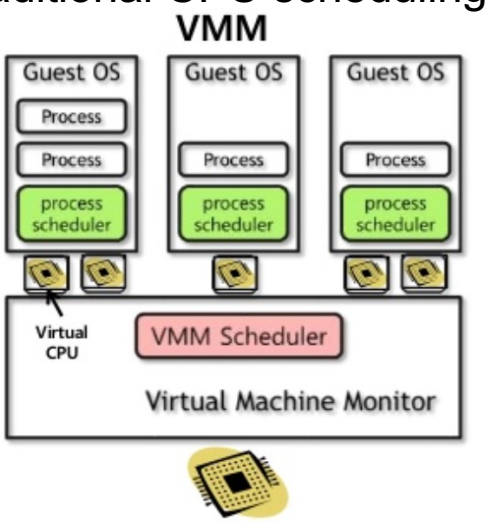
VMM reference architecture

- 3 main modules
  - *Dispatcher*: VMM entry point that reroutes privileged instructions issued by VMs to one of the other two modules
  - *Allocator* (or *scheduler*): decides about the system resources to be provided to VM
  - *Interpreter*: executes a proper routine when VM executes a privileged instruction



VMM reference architecture: scheduler

- VMM scheduler: additional scheduling layer with respect to traditional CPU scheduling

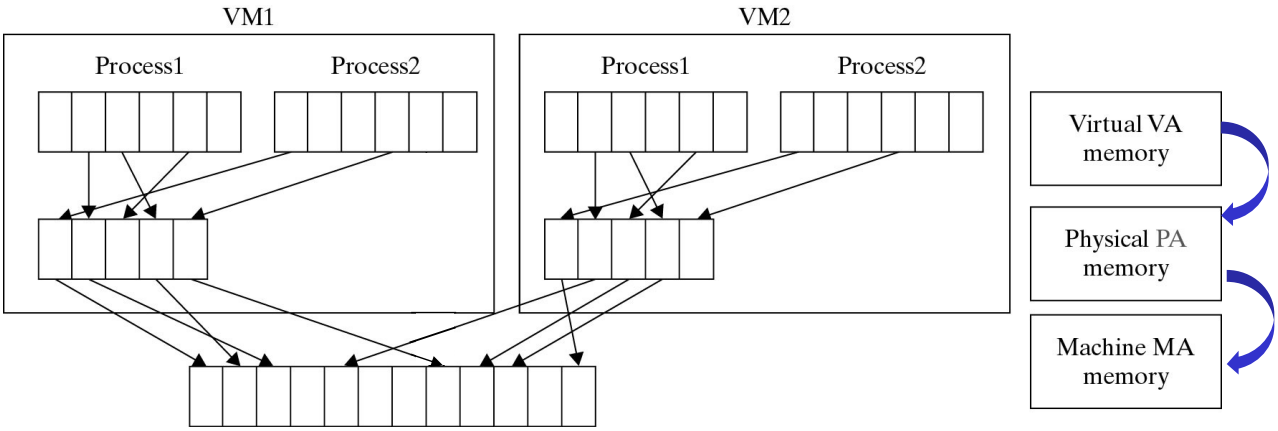


- How to schedule virtual CPUs on physical CPUs?

Memory virtualization

- In a *non-virtualized* environment
  - One-level memory mapping: from virtual memory to physical memory provided by page tables
  - MMU and TLB hardware components to optimize virtual memory performance
- In a *virtualized* environment
  - All VMs share the same machine memory and VMM needs to partition it among VMs
  - **Two-level memory mapping**: from guest virtual memory to guest physical memory to host physical memory
- Some terms
  - *Guest virtual memory*: memory visible to apps; continuous virtual address space presented by guest OS to apps
  - *Guest physical memory*: memory visible to guest OS
  - *Host/machine physical memory*: actual hw memory visible to VMM

Two-level memory mapping



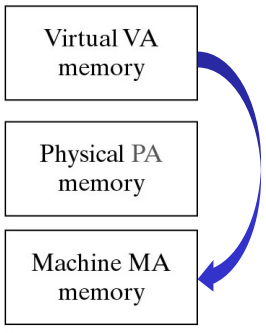
- Going from guest virtual memory to host physical memory requires two-level memory mapping

  GVA (guest virtual address) ➔ GPA (guest physical address) ➔ HMA (host machine address)

- Guest physical address ≠ host machine address: why?
  - Hints: many VMs; what does guest OS expect about its memory?

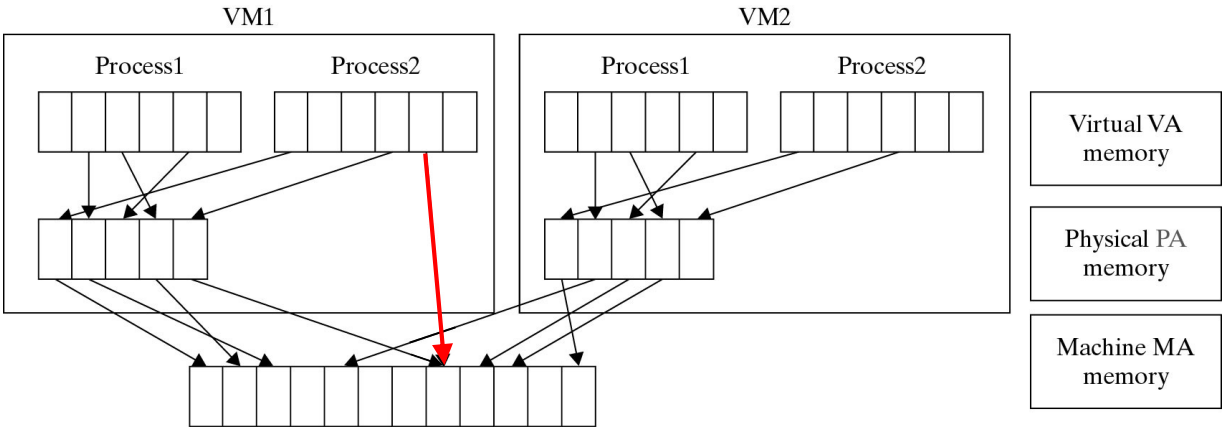
Shadow page tables

- To avoid unbearable performance drop due to extra memory mapping, VMM maintains shadow page tables (**SPTs**) and uses them to accelerate address mapping
  - So to achieve direct mapping from GVA to HPA



- SPT directly maps GVA to HPA
  - Guest OS creates and manages page tables (PTs) for its virtual address space without modification
    - But these PTs are not used by MMU hardware
  - VMM creates and manages PTs that map virtual pages directly to machine pages
    - These VMM PTs are the shadow page tables and are loaded into MMU
  - VMM needs to keep SPTs consistent with changes made by each guest OS to its PTs

Memory mapping with SPTs



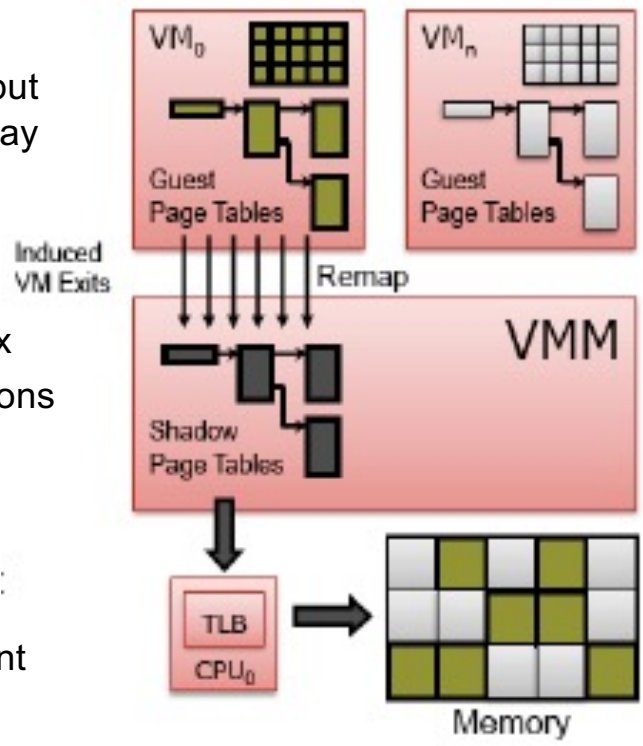
- VMM uses TLB hardware to map virtual memory directly to machine memory to avoid the two levels of translation on every access (red arrow)

Shadow page tables consistency

- When guest OS changes its PTs, VMM needs to update SPTs to enable a direct lookup
- How?
  - VMM maps guest OS PTs as read only
  - When guest OS writes to PTs, trap to VMM
  - VMM applies write to SPT and guest OS PT, then returns
  - Aka memory tracing
  - Adds overhead

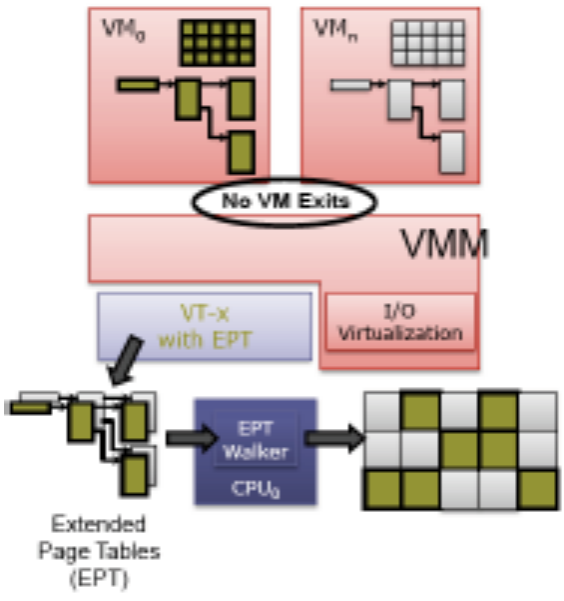
Challenges in memory virtualization with SPT

- Address translation
  - Guest OS expects contiguous, zero-based physical memory, but underlying machine memory may be non contiguous: VMM must preserve this illusion
- Page table shadowing
  - SPT implementation is complex
  - VMM intercepts paging operations and constructs copy of PTs
- Overheads
  - SPTs consume significant host memory
  - SPTs need to be kept consistent with guest PTs
  - VM exits add to execution time



Hw support for memory virtualization

- SPT is a software-managed solution: let's consider a more efficient hardware solution
- Second Level Address Translation (SLAT) is the hardware-assisted solution for memory virtualization (Intel EPT and AMD RVI) to translate GVA into HPA
- Using SLAT significant performance gain with respect to SPT: around 50% for MMU intensive benchmarks



Case study: Xen

- The most notable example of paravirtualization

  www.xenproject.org (initially developed at Cambridge Univ.)

  – Open-source type-1 (system VMM) hypervisor with microkernel design
  – Offers to guest OS a virtual interface (hypercall API) to whom guest OS must refer to access machine physical resources
  – Supports both **paravirtualization** (PV) and hardware-assisted virtualization (HVM)
    - With paravirtualization Xen requires PV-enabled guest OSs and PV drivers (part of Linux kernel and other OSs)
      – OSs ported to Xen: Linux, NetBSD, FreeBSD
    - With HVM also unmodified guest OSs (e.g., Windows)
  – Foundation for commercial virtualization products (e.g., XenServer, Oracle VM)
  – Also embedded Xen distros
  – Powers IaaS providers (e.g., Alibaba, Amazon, Rackspace)

Xen: pros and cons

✓ Thin hypervisor model
  – 300K lines of code on x86, 65K on Arm
  – Small footprint and interface (around 1MB in size)
  – Scalable: up to 4,095 host CPUs with 16Tb of RAM
  – More robust and secure than other hypervisors, see youtu.be/sjQnAIJji4k
  – But still vulnerable to attacks xenbits.xen.org/xsa

✓ Continuously improved

✓ Flexibility in management
  – Tuning for performance

✓ Low overhead (within 2%) with respect to bare metal machine without virtualization

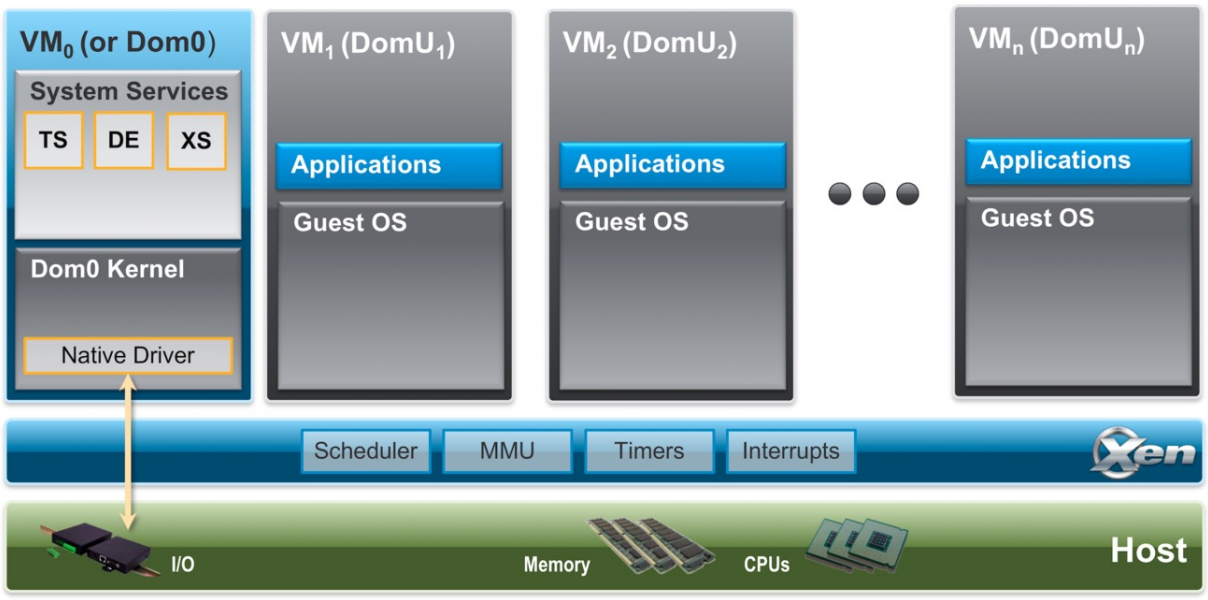
✓ Supports VM live migration

✗ I/O performance still remains challenging

Xen architecture

- Goal of Cambridge Univ. group who designed Xen (late 1990s, first release in 2003)
  - Design hypervisor capable of scaling to ~100 VMs running applications without any modifications to ABI
- Microkernel design
- What can be paravirtualized?
  - Privileged instructions
    - Privileged instructions issued by guest OS are replaced with hypercalls
  - Page tables (memory access)
  - Disk access and I/O devices
  - Interrupts and timers

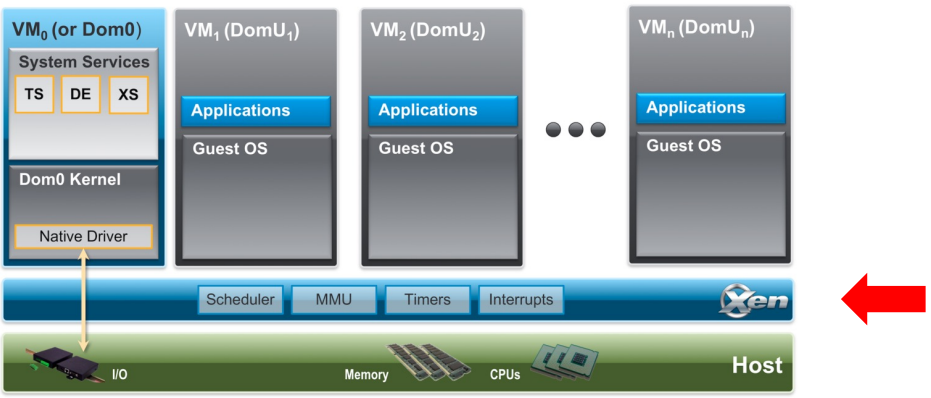
Xen architecture



wiki.xenproject.org/wiki/Xen_Project_Software_Overview

Xen architecture: hypervisor

- In charge of scheduling, memory management, interrupt and device control
- Per-domain and per-vCPU info management



Xen architecture: domains

- 2 kinds of domains: control domain that starts and manages all the others unprivileged domains
- Guest domains: DomU (unprivileged)
  - Represent VM instances, each running its OS and apps
  - Run on virtual CPUs (vCPUs)
  - Totally isolated from hw (i.e., no privilege to access hw or I/O functionality)
- Dom0 (control domain): specialized VM having special privileges that is, capability to access hw directly, handles all access to system's I/O functions and interacts with the other VMs
  - Mandatory, initial domain started by Xen on boot
  - Contains drivers for all devices and systems services: Device Emulation (DS), XenStore/XenBus (XS), and Toolstack (TS)

Dom0 components: XenStore and Toolstack

- XenStore: information storage space shared between domains managed by *xenstored* daemon
  - Stores configuration and status information
  - Implemented as hierarchical key-value storage
    - When values are changed in the store, a watch function notifies listeners (e.g., drivers) of changes of the key they have subscribed to
  - Communicates with guest VMs via shared memory using Dom0 privileges
- Toolstack: allows a user to manage VM lifecycle (create, shutdown, pause, migrate) and configuration
  - To create a new VM, a user provides a configuration file describing memory and CPU allocation and device configurations
  - Toolstack parses this file and writes this information in XenStore
  - Takes advantage of Dom0 privileges to map guest memory, to load kernel and virtual BIOS and to set up initial communication channels with XenStore and with virtual console when a new VM is created

CPU scheduler in Xen

- Hypervisor scheduler decides, among all the virtual CPUs (vCPUs) of the various VMs, which ones should execute on the physical CPUs (pCPUs)
  - Further scheduling level with respect to those provided by OS (scheduling of processes and scheduling of user-level threads within processes)
- Xen allows to choose among different CPU schedulers
  - **Credit** scheduler is the default
- Scheduling algorithm goals:
  - Make sure that domains get fair share of CPU
    - *Proportional share* algorithm: allocates pCPU in proportion to the number of shares (weights) assigned to vCPUs
  - Keep the CPU busy
    - *Work-conserving* algorithm: does not allow pCPU to be idle when there is work to be done
  - Schedule with low latency

Credit scheduler

- Proportional fair share and work-conserving scheduler
- Each domain (including Domain0) is assigned a *weight* and a *cap* (tunable parameters)
  - Weight: relative pCPU allocation per domain (default 256)
  - Cap: maximum amount of CPU a domain can use
    - cap = 0 (default): vCPU can receive any extra CPU (i.e., work-conserving)
    - cap ≠ 0: limits amount of pCPU that vCPU receives (i.e., non work-conserving); expressed as % of pCPU (e.g., 100 = 1 pCPU, 50 = 0.5 pCPU)
- The scheduler transforms the weight into a *credit* allocation for each vCPU
  - The credit value represents the pCPU share that the domain is expected to have
  - As a vCPU runs, it consumes credit
  - If its credit value is negative, the domain is in *OVER* priority; otherwise, in *UNDER* priority wiki.xenproject.org/wiki/Credit_Scheduler

Credit scheduler: algorithm

- For each pCPU, the scheduler maintains a queue of vCPUs, with all the vCPUs in UNDER priority first, followed by vCPUs in OVER priority
  - Round-robin ordering within UNDER and OVER priorities
  - Scheduler picks the vCPU at the head of the queue
  - Selected vCPU receives 30 ms time slice before being preempted to run another vCPU
  - VCPUs in OVER priority cannot be scheduled unless there is no UNDER VCPUs in the queue
- The scheduler load balances vCPUs across pCPUs on SMP (symmetric multi-processor) host
  - Before a pCPU goes idle, it considers other pCPUs in order to find any UNDER credit vCPU: no pCPU is idle when there is runnable work in the system

Performance comparison of hypervisors

- Developments in virtualization techniques and CPU architectures have reduced the performance cost of virtualization but still some overhead
  - Especially when multiple VMs compete for hw resources
- We consider two performance comparison studies
  - Papers available on course site
  - "Old" studies but overall message is still valid
- Take-home message
  - No one-size-fits-all solution exists
  - Different hypervisors show different performance characteristics for varying workloads

Performance comparison of hypervisors

A component-based performance comparison of four hypervisors (IM 2013)

- Microsoft Hyper-V, KVM, VMware vSphere and Xen, all with *hardware-assisted virtualization*
- Analyzed components: CPU, memory, disk I/O and network I/O

- Results
  - Performance depends on type of virtualized hw resource, but no single hypervisor always outperforms the others
    - vSphere performs the best, but the others perform respectably
    - CPU and memory: lowest levels of overhead
    - I/O and network: Xen overhead for small disk operations
- Takeaway: consider application type because different hypervisors may be best suited for different workloads

Performance comparison of hypervisors

Performance overhead among three hypervisors: an experimental study using Hadoop benchmarks (BigData 2013)

- Use Hadoop MapReduce apps to evaluate and compare the performance impact of three hypervisors
  - Commercial one (undisclosed), Xen, and KVM
- Results
  - For CPU-intensive benchmarks, negligible performance difference among hypervisors
  - For I/O-intensive benchmarks significant performance variations
    - Commercial hypervisor best at disk writing, KVM best for disk reading
    - Xen best when combination of disk reading and writing with CPU-intensive computation

VM portability

- **VM image**: a single file for each VM which contains a bootable OS, data files, and applications
- Virtual machine images come in different formats
- How to import and export VM images and avoid vendor lock-in?
- Open Virtualization Format (OVF)
  - Open industry standard (ISO 17203) for packaging and distributing VMs
    - Virtual-platform agnostic
  - Image stored in .ova file (Open Virtual Appliance)
  - VM configuration specified in XML format within a .ovx file
  - Supported by many virtualization products including Hyper-V, VMware, VirtualBox, XenServer

VM resizing and migration

- Useful techniques to deploy and manage large-scale virtualized environments
  - **Dynamic resizing** for vertical scaling (scale up, scale down) of VMs
  - **Live migration** of VMs
    - Move VM between different physical machines (or even data centers) without stopping it

VM dynamic resizing

- Fine-grain mechanism with respect to migrating or rebooting VMs
  - Example: app running on a VM consumes a lot of resources, thus VM starts running out of RAM and CPU
  - Solution: dynamically resize VM (aka *warm* resizing)

✓ More cost-effective and faster than VM reboot

✗ Not supported by all virtualization products and guest OSs

- What can be resized *without* stopping and rebooting the VM?
  - Number of virtual CPUs
  - Memory

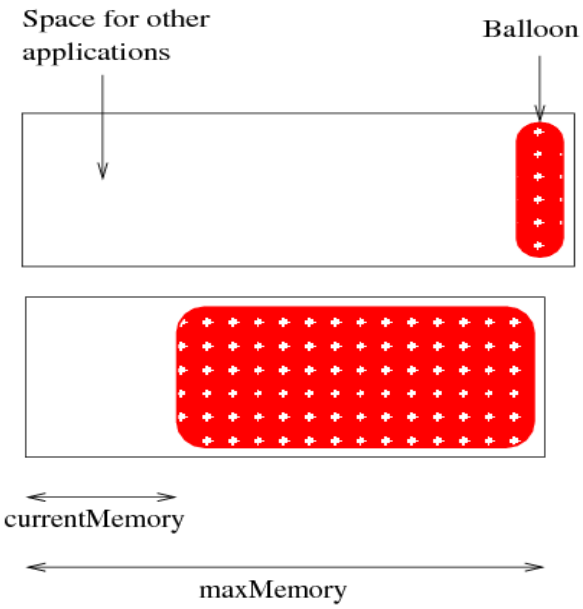
VM dynamic resizing: CPU

- Add or remove virtual CPUs (without turning off VM)
- Linux supports CPU hot-plug/hot-unplug
  www.kernel.org/doc/html/latest/core-api/cpu_hotplug.html
  - Uses information in virtual file system `sysfs` (processor info is in `/sys/devices/system/cpu`)
  - `/sys/devices/system/cpu/cpuX` for cpuX (X = 0, 1, 2, …)
  - To turn on cpu #5:
    ```
    echo 1 > /sys/devices/system/cpu/cpu5/online
    ```
  - To turn off cpu #5:
    ```
    echo 0 > /sys/devices/system/cpu/cpu5/online
    ```
- VM CPU resizing can be managed using `virsh`
  - `virsh`: command line tool to configure and manage virtual machines, available with some hypervisors (KVM, Xen)
  - E.g., set the number of vCPUs while VM is running (cannot exceed max. number of vCPUs)

```
virsh setvcpus <vm_name> <vcpu_count> --current
```


VM dynamic resizing: memory

- Based on *memory ballooning*
  - Mechanism used by hypervisors (e.g., KVM, Xen and VMware) to pass memory back and forth between hypervisor and guest OSs
  - In KVM: *virtio_balloon* driver
- When balloon deflates: more memory for the VM
  - Anyway, VM memory size cannot exceed maxMemory
- When balloon inflates
  - Swap memory pages to disk



VM migration

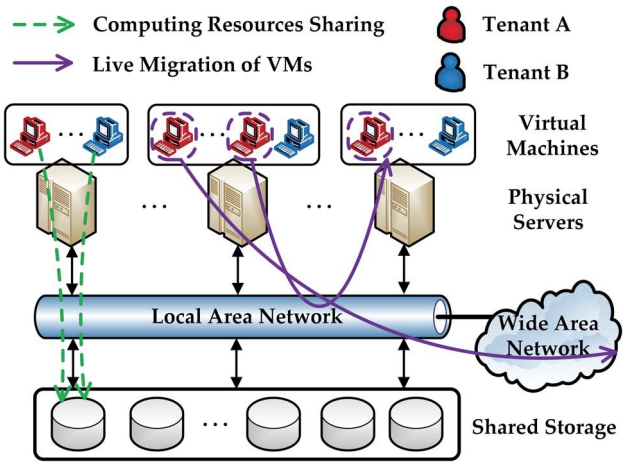
- Pros
  - Useful in clusters and virtual data centers to:
    - ✓ Consolidate infrastructure
    - ✓ Add failover flexibility
    - ✓ Balance load
- Cons
  - ✓ Requires VMM support
  - ✓ Migration overhead is non-negligible
  - ✓ WAN migration is scarcely supported

VM migration

- Approaches to migrate VM instances between physical machines:
  - Stop and copy: shutdown source VM and transfer VM image to destination host, but downtime can be too long
    - VM image can be large and network bandwidth limited

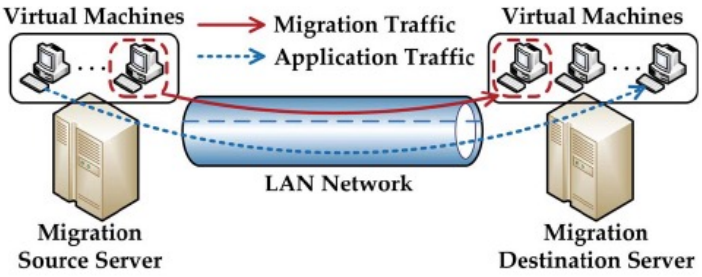
Our focus

  - **Live migration**: source VM is running during migration
    - Largely used by Google: > 1M migrations per month



VM live migration

- Preliminary steps before starting VM live migration
  - *Setup* phase: determine source host, destination host and VM to migrate (goals of load balancing, energy efficiency, server consolidation)
- What to migrate? Memory, storage, network connections
- How? In a transparent way wrt applications running inside the VM
  - But migration transparency is hard to achieve, live migration still causes application downtime: how to limit it?



VM live migration: storage

- Let's first focus on VM migration within a cluster environment

- To migrate storage
  - Can use network-accessible and shared storage system
    - SAN (Storage Area Network) or cheaper NAS (Network Attached Server) or distributed file system (e.g., NFS, GlusterFS, CEPH)
  - Without shared storage: source VMM stores all source VM data in an image file, which is transferred to destination host

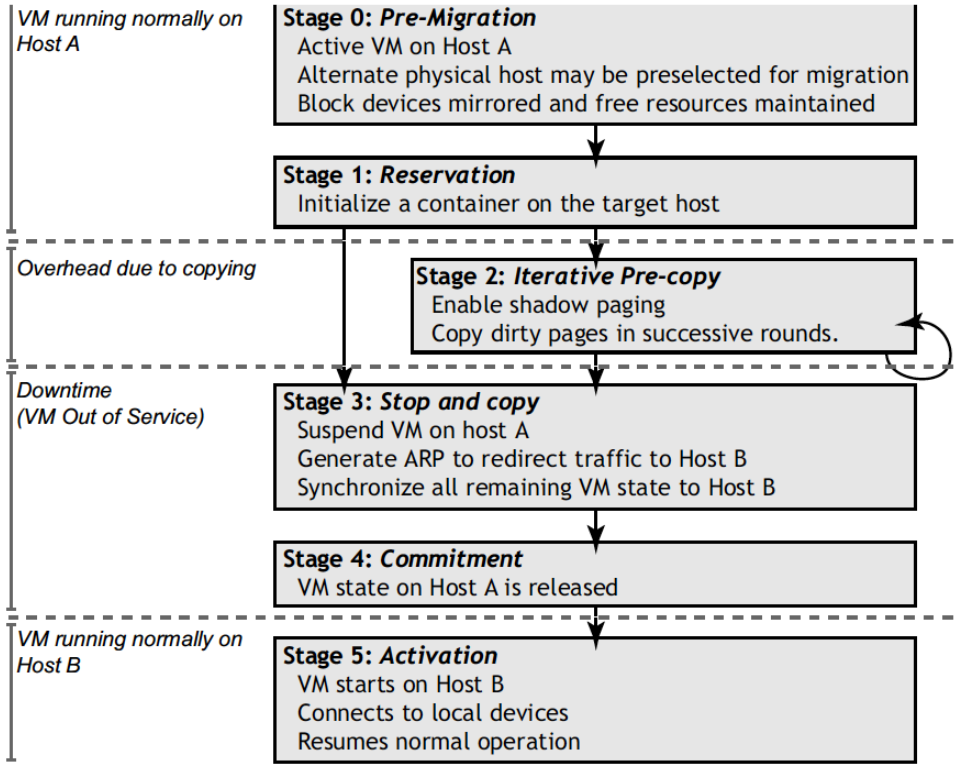
VM live migration: network

- To migrate network connections
  - Source VM has its own virtual IP address, which can be distinct from IP address of source host; can also have its own distinct virtual MAC address
    - VMM maintains a mapping of virtual IP and MAC addresses to their corresponding VMs
  - If source and destination hosts are connected to a single switched LAN, an unsolicited ARP reply from source host is provided, advertising that the IP has moved to a new location
    - A few in-flight packets might be lost
  - Alternatively, use forwarding mechanisms on source host

VM live migration: memory

- To migrate memory (including CPU and device state):
  1. Pre-copy phase: VMM copies in an iterative way the memory pages from source VM to destination VM *while* source VM is running
     - During iteration *n* those pages dirtied during iteration *n*-1 are copied
  2. Stop-and-copy phase: source VM is suspended and the last dirty pages are copied, as well as CPU and device drivers states; VM applications do not run
     - Downtime: from some msec to sec, depending on memory size, application memory workload and network bandwidth
  3. Commitment and reactivation phases: destination VM is activated and recovers application execution; source VM is removed (and source host may be turned off)
- Known as **pre-copy** approach
  - Memory is copied *before* VM execution resumes at destination
  - Popular solution (e.g, KVM, VMWare, Xen, Google CE)

VM live migration: overall process



Clark et al., Live Migration of Virtual Machines, NSDI 2005

VM live migration: alternatives for memory

- Pre-copy cannot migrate in a transparent manner memory-intensive apps
  - E.g., for write-intensive memory app, pre-copy is unable to transfer memory faster than memory is dirtied by running app
- Two alternative approaches
  - Post-copy
  - Hybrid
- **Post-copy**
  - CPU and device state are transferred immediately to destination host followed by transfer of execution control to destination host
  - Memory is fetched on-demand if needed by the running VM on the destination host (*pull* approach)
  - ✓ Reduces downtime and total migration time
  - ✗ Incurs app degradation due to page faults which must be resolved over the network

VM live migration: alternatives for memory

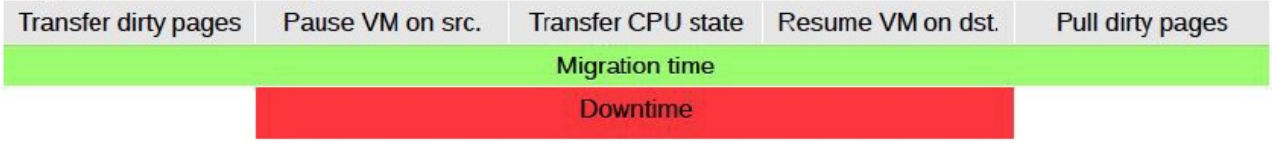
- **Hybrid**
  - Special case of post-copy migration: post-copy preceded by a bounded pre-copy stage
  - Idea: transfer a subset of the most frequently accessed memory pages before VM execution is switched to destination, so to reduce app performance degradation due to memory transfer after VM is resumed
  - ✓ Pre-copy stage reduces the number of future network-bound page faults as a large portion of VM memory is already pre-copied
- No standard implementation of post-copy and hybrid approaches in current hypervisors

VM live migration: alternatives for memory

- Summary of approaches to migrate memory
- Pre-copy Live Migration

- Post-copy Live Migration

- Hybrid Live Migration



VM live migration and hypervisors

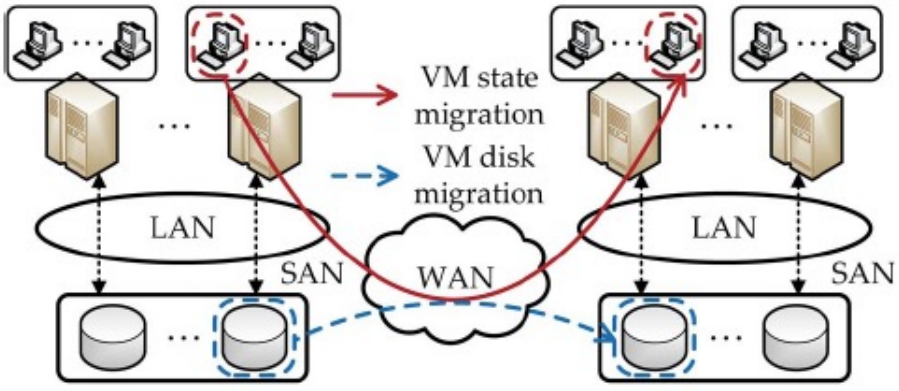
- VM live migration is supported by open-source and commercial hypervisors
  - E.g., KVM, Hyper-V, Xen, VirtualBox

- Can be managed using `virsh` with different options

```
$> virsh migrate --live [--undefinesource] \
[--copy-storage-all] [--copy-storage-inc] domain desturi
$> virsh migrate-setmaxdowntime domain downtime
$> virsh migrate-setspeed domain bandwidth
$> virsh migrate-getspeed domain
```


VM migration in WAN

- How to achieve VM live migration across multiple geo-distributed data centers?
  - Key challenge: maintain network connectivity and preserve open connections during and after migration
  - Limited support in open-source and commercial hypervisors



VM migration in WAN: storage

- Approaches to migrate storage in WAN
  - Shared storage
    - ✗ Storage access time can be too slow
  - On-demand fetching
    - Transfer only some blocks to destination and then fetch remaining blocks from source only when requested
    - ✗ Does not work if source crashes
  - Pre-copy plus write throttling
    - Pre-copy VM disk image to destination whilst VM continues to run, keep track of write operations on source (delta) and then apply delta on destination
    - If write rate at source is too fast, use write throttling to slow down the VM so that migration can proceed

VM migration in WAN environments: network

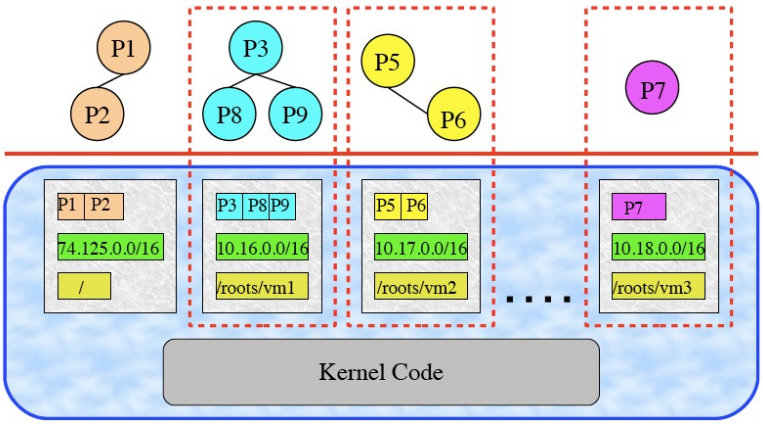
- Approaches to migrate network connections in WAN
  - IP tunneling
    - Set up an IP tunnel between old IP address at source VM and new IP address at destination VM
    - Use tunnel to forward all packets that arrive at source VM for old IP address
    - Once migration has completed and VM can respond at its new location, update the DNS entry with new IP address
    - Tear down the tunnel when no connections remain that use the old IP address
    - ✗ Does not work if source VM crashes
  - Virtual Private Network (VPN)
    - Use MPLS VPN to create the abstraction of a private network and address space shared by multiple data centers
  - Software-Defined Networking (SDN)
    - Change control plane, no need to change IP address!

OS-level virtualization

- So far system-level virtualization
- Let's now consider **operating system (OS) level virtualization** (or ***container-based virtualization***)
- Allows to run multiple isolated (*sandboxed*) user-space instances on top of a single OS
  - Such instances are called:
    - **containers**
    - jails
    - zones
    - virtual environments

OS-level virtualization

- OS kernel allows the existence of multiple isolated user-space instances, called containers
- Each container has:
  - Its own set of processes, file systems, users, network interfaces with IP addresses, routing tables, firewall rules, …
- Containers share the same OS kernel (e.g., Linux)

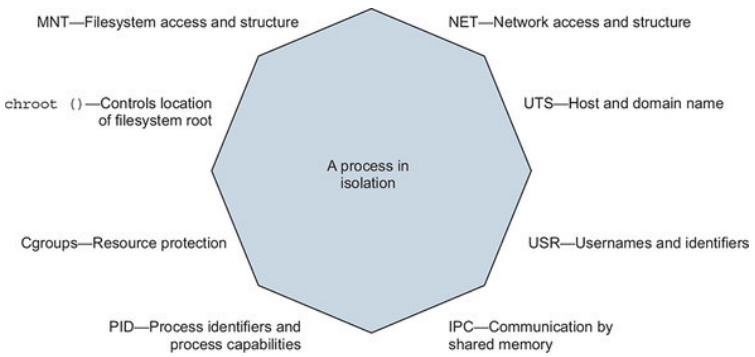


OS-level virtualization: mechanisms

- Which kernel mechanisms to manage containers?
  - Need to isolate processes from each other in terms of sw and hw (CPU, memory, …) resources
- Main mechanisms offered by Unix-like OS kernel
  - chroot (change root directory)
    - Allows to change the apparent root folder for the current running process and its children
  - cgroups (Linux-specific)
    - Manage resources for groups of processes
  - namespaces (Linux-specific)
    - Per-process resource isolation

Mechanisms: namespaces

- Feature of Linux kernel that allows to isolate what a set of processes can see in the operating environment (processes, ports, files, ...)
- Kernel resources are partitioned so that one set of processes sees one set of resources, while another set of processes sees a different set of resources
- Different types of namespaces



Mechanisms: namespaces

- mnt: isolates mount points seen by a container
  - Virtually partitions the file system: processes running in separate mount namespaces cannot access files outside of their mount point
- pid: isolates PID space, so that each process only sees itself and its children (PID 1, 2, 3, …)
- network: allows each container to have its dedicated network stack
  - Its own private routing table, set of IP addresses, socket listing, firewall, and other network-related resources
- user: isolates user and group IDs
  - E.g., allows a non-root user on host to be mapped with root user within container, without having actual root access to host

Mechanisms: namespaces

- uts (Unix timesharing): provides dedicated host and domain names
  - Allows processes to think they are running on differently named servers
- ipc: provides dedicated shared memory for IPC, e.g., different Posix message queues

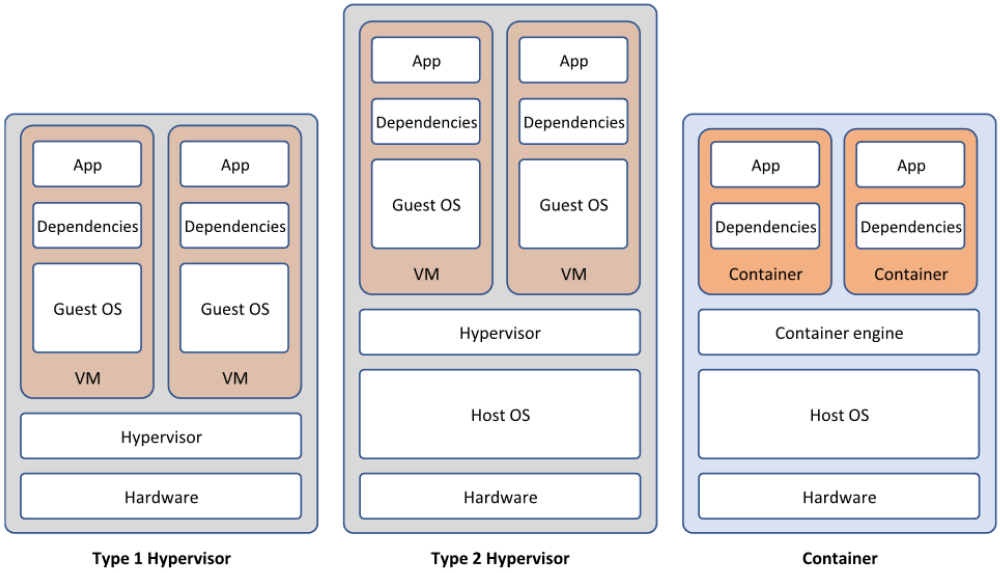
Mechanisms: cgroups

- cgroups: control groups
- Allows to limit, measure and isolate the use of hw resources (CPU, memory, block I/O, network) of a set of processes
- Low-level filesystem interface similar to `sysfs` and `procfs`
  - By default mounted on `/sys/fs/cgroup/` directory

- Mechanisms in a nutshell:
  - namespaces implements *information isolation*: what a container can see
  - cgroups implements *resource isolation*: how much resources a container can use

OS-level virtualization: pros

- VMM-based vs container-based virtualization



In a nutshell: lightweight vs. heavyweight

OS-level virtualization: pros

With respect to VMM-based virtualization (type-1)

- ✓ Minimal performance degradation
  - Apps invoke system calls directly, without VMM indirection
- ✓ Minimum startup and shutdown times
  - Seconds (even msec) per container, minutes per VM
- ✓ High density
  - Hundreds of containers on a single physical machine (PM)
- ✓ Smaller image (footprint)
  - Does not include OS kernel
- ✓ Ability to share memory pages among multiple containers running on same PM
- ✓ Increased portability and interoperability
- ✓ Containerized apps independent of execution environment

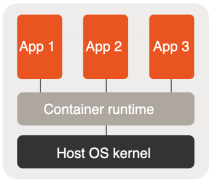
OS-level virtualization: cons

With respect to VMM-based virtualization (type-1)

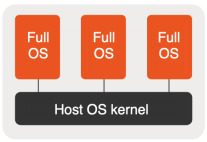
- ✗ Less flexibility
  - Cannot run different OS kernels simultaneously on same PM
- ✗ Only native applications for supported OS kernel
  - E.g., native app for Linux
- ✗ Less isolation and higher performance interference on shared system resources
  - Process-level isolation
- ✗ Higher risk of vulnerability and more threats
  - Vulnerability in OS kernel can compromise entire system
  - Since containers share OS kernel, a single compromised container could comprise host OS and other containers

OS-level virtualization: some products

- **Docker**
  - The most popular container engine
  - Provides *application containers*
  - Supports Open Container Initiative (OCI), a set of standards for containers



Application containers



System containers

- LXC (LinuX Containers)
  - Supported by mainline Linux kernel
  - Provides *system containers* (full OS image)
- Podman
  - Supports OCI
  - Docker compatible CLI
- FreeBSD Jail
- OpenVZ (for Linux)
- Virtuozzo Containers

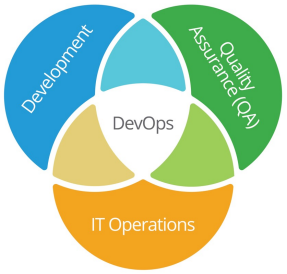
OS-level virtualization: only Linux?

- Windows and OS X also support container-based virtualization
  - E.g., Docker Desktop

- Alternative: install a VM with Linux as guest OS and then install a container-based virtualization product inside VM
  - ✗ Performance loss because of nested virtualization

Containers, DevOps and CI/CD

- Containers help in the shift to DevOps and CI/CD (Continuous Integration and Continuous Deployment)
- DevOps = Development and Operations
  - Development methodology with a set of practices aimed at bridging the gap between Development and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment

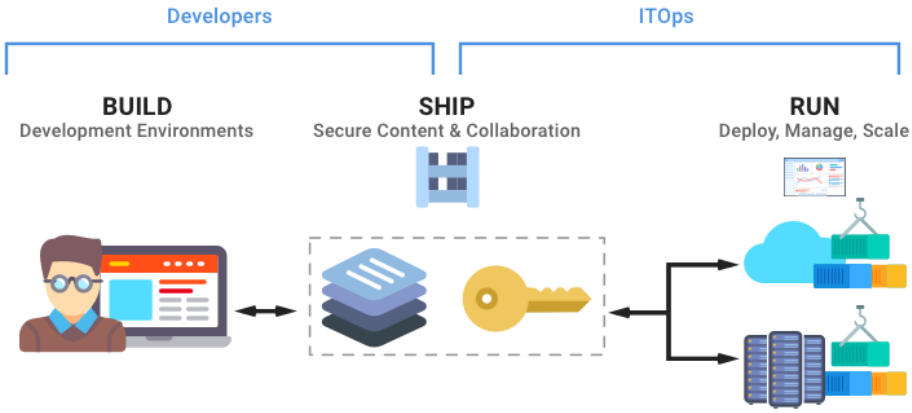


Containers, DevOps and CI/CD

- CI/CD = Continuous Integration and Continuous Delivery/Deployment
  - Continuous integration: sw development practice that merges work of all developers working on same project
  - Continuous delivery: ensures reliable and frequent releases
- In DevOps culture, the two practices are combined to enable teams to ship software releases effectively, reliably, and frequently

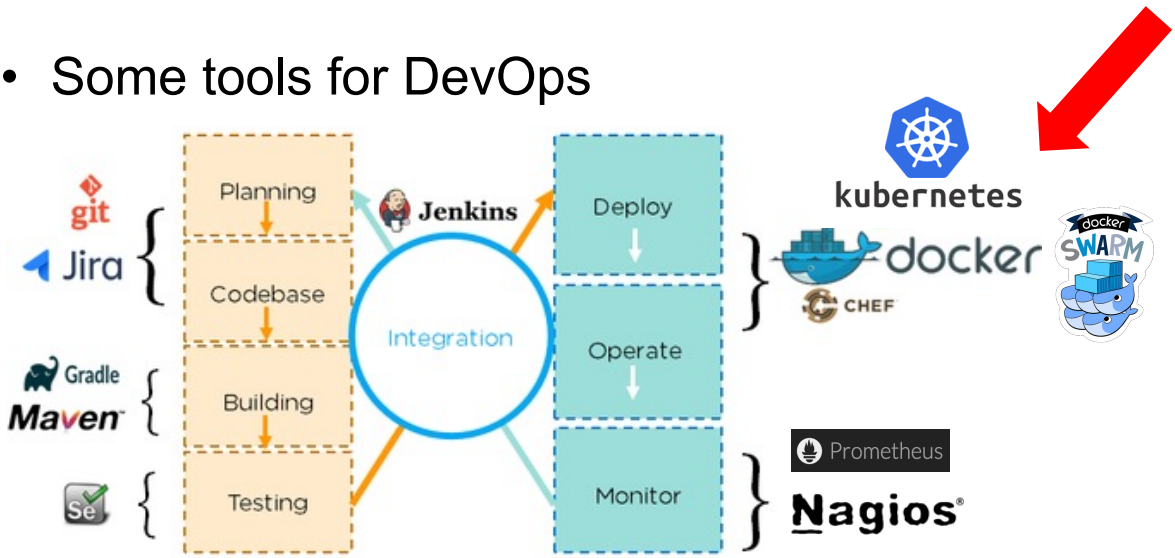
Containers, DevOps and CI/CD

- Containers are become a standard to build, package, share, and deploy apps and all their dependencies
  - Containers (more than VMs) allow developers to build code collaboratively by sharing images while simplifying deployment to different environments without further configuration



Containers, DevOps and CI/CD

- Some tools for DevOps



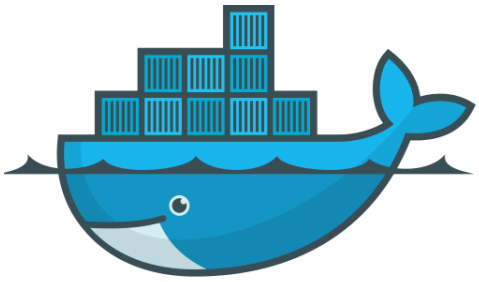
Containers, microservices, and serverless

- Using containers
  - App and all its dependencies into single package that can run almost anywhere
  - Using fewer resources than traditional VMs
- Containers are a key enabling technology for microservices and serverless computing
  - Wrap microservices and functions in containers

Docker

- Let's go into Docker details

www.ce.uniroma2.it/courses/sdcc2324/slides/Docker.pdf



Container resizing

- As for VMs, we can resize and migrate containers
- Resizing (CPU, memory, I/O) changes dynamically container limits
  - Not supported for Windows

```
$docker update [OPTIONS] CONTAINER [CONTAINER...]
```

  - Some example

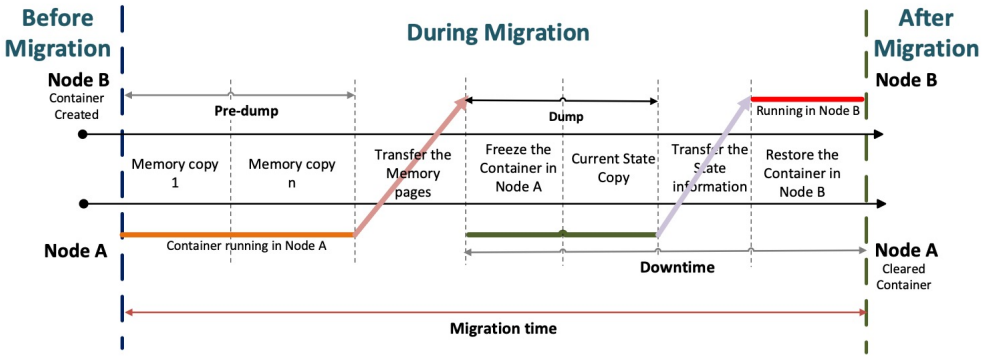
```
$ docker update --cpu-shares 512 containerID
$ docker update --cpu-shares 512 -m 300M containerID
```


Live migration of containers

- As for VM migration, we need to:
  - Save state
  - Transfer state
  - Restore from state
- State saving, transferring and restoring happen with frozen app: migration downtime
  - Use memory pre-copy or memory post-copy
- No native support in container engines, requires additional tool
- We also need to migrate container image (and volumes) and network connections

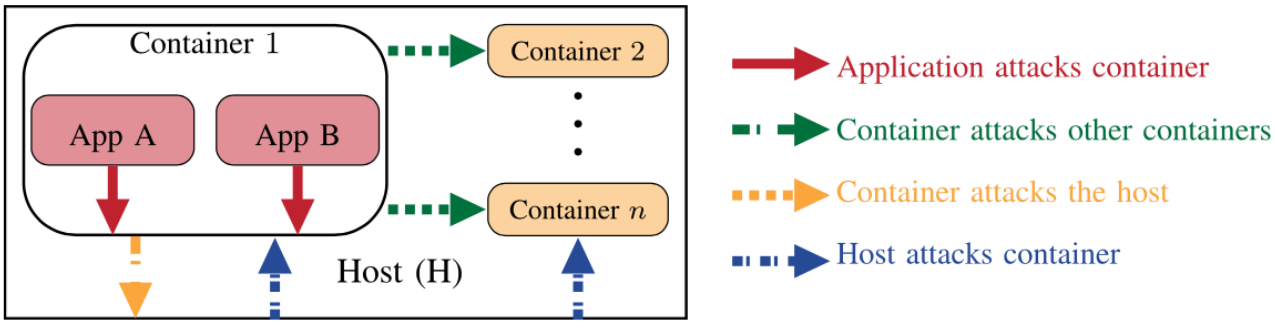
Live migration of containers

- Use CRIU tool to support live migration (in Docker and other container engines) through checkpointing and restoration
  - During checkpoint, CRIU freezes running container at source host and collects information about its CPU state, memory content, and process tree
  - Collected information is passed on to destination host, and container is resumed
  - How to with Docker



Container security

- Where attacks come from in a containerized environment?



- Example of attack: *container escape and privilege escalation*
  - Attacker can leverage containerized app's vulnerabilities to breach its isolation boundary, gaining access to host system's resources
  - Once attacker accesses host, it can escalate its privilege to access other containers or run harmful code on host

Container orchestration

- Sw platforms for managing the deployment of multi-container packaged applications in large-scale clusters
- Allow to configure, provision, deploy, monitor, and dynamically control containerized apps
  - Used to integrate and manage *containers at scale*
- Examples
  - **Docker Swarm** (see Docker slides)
  - **Kubernetes** (next lesson)
  - Marathon
  - Amazon Elastic Container Service
  - Google Kubernetes Engine

Fully managed Cloud services

Containers in Cloud

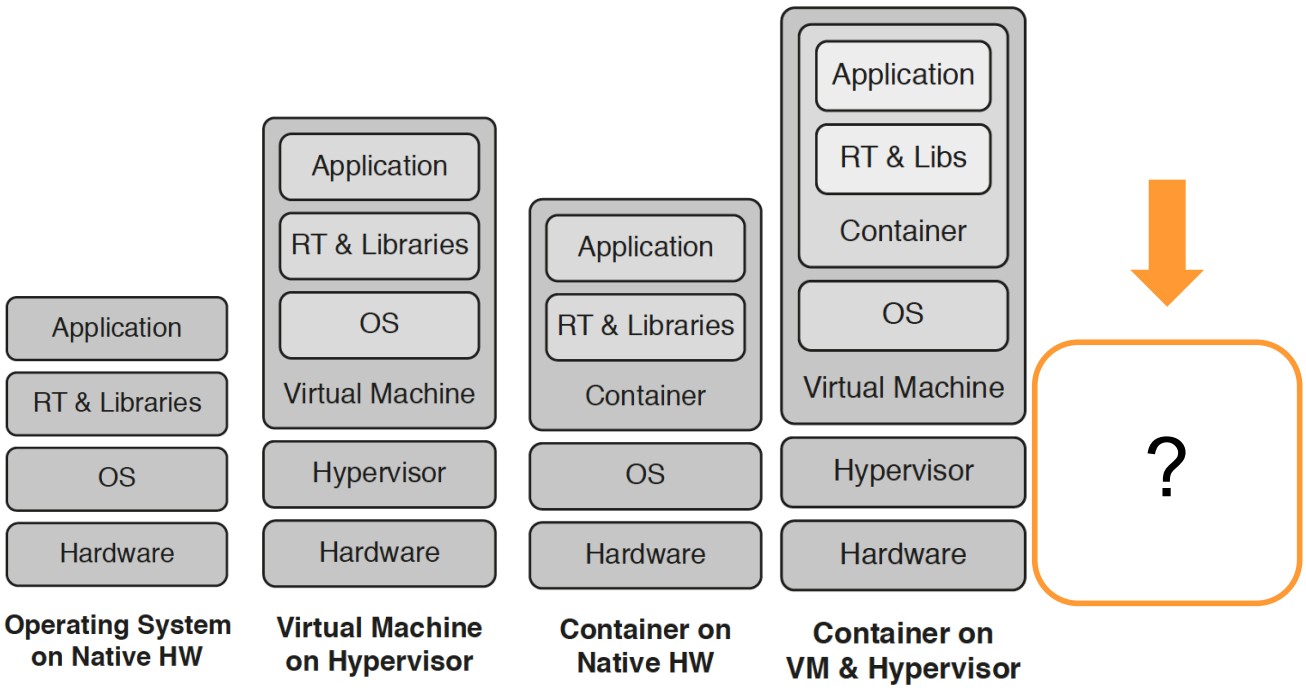
- Containers and container platforms as first-class Cloud services
- Container-as-a-Service (CaaS)
  - Amazon Elastic Container Service
    - Multiple deployments, including EC2, AWS Local Zones, Fargate
  - Azure Container
  - Google Cloud Run

Hypervisors and containers in Cloud

- Which virtualization technology for IaaS providers?
  - ✓ Hypervisor-based virtualization: greater security, isolation, and flexibility (different OSs on same PM)
  - ✓ Container-based virtualization: smaller-size deployment and thus larger density, reduced startup and shutdown times
- Some questions
  - Containers on VMs or on top of bare metal?
  - Are containers replacing VMs?

New lightweight virtualization approaches

- Deployment strategies examined so far

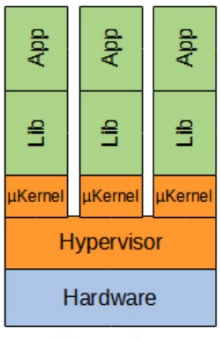


New lightweight approaches to virtualization

- Microservices, serverless computing, edge/fog computing, compute continuum demand for low-overhead (or lightweight) virtualization techniques, even lighter than containers
  - Additional requirement: improve security
- **MicroVM**, **lightweight OSs** and **unikernels**
  - Overall idea: reduce OS overhead and attack surface
  - OS overhead: services and tools coming with common OSs (shells, editors, core utils, and package managers) are not needed
  - Attack surface: images contain only the code that is strictly necessary for app to run, thus resulting in minimal attack surface

MicroVM runtimes

- Tiny, specialized VMMs that run lightweight VMs (microVMs)
- Goal: reduce memory footprint and improve security of virtualization layer
- Firecracker: VMM purpose-built by Amazon for creating and managing secure, efficient and multi-tenant microVMs
  - Why? To enable AWS Lambda and AWS Fargate
  - Based on KVM but with minimalist design (exclude unnecessary devices and guest functionality)
  - Open source, written in Rust
  - Runs app in microVM: < 125 ms startup time and <5 MB memory footprint
  - Scales to thousands of multi-tenant microVMs
  - Supported OS guests inside microVM: Linux and OSv



Firecracker microVM

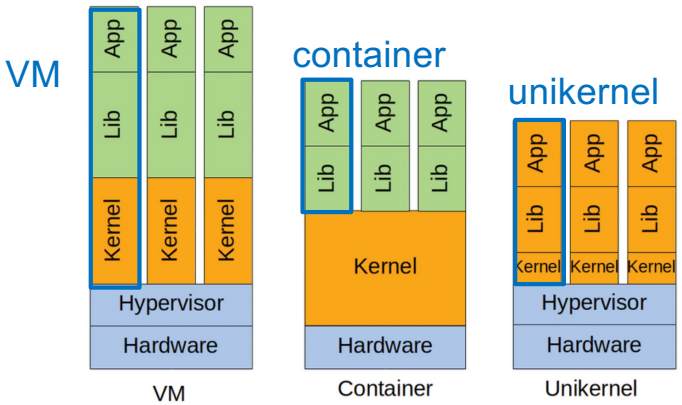
Agache et al., Firecracker: Lightweight virtualization for serverless applications, NSDI 2020

Lightweight operating systems

- Minimal, container-focused OSs, typically with monolithic kernel architecture
  - Special-purpose OSs to run containerized apps
  - E.g., Fedora CoreOS, Rancher OS
- Fedora CoreOS
  - Minimal, monolithic and compact Linux distribution
    - Only minimal functionalities required for deploying apps inside containers, together with built-in mechanisms for service discovery, container management and configuration sharing
  - Goal: provide the best container host to run containerized workloads securely and at scale
  - Can be installed on hypervisor or bare metal
  - Fast bootstrap and small memory footprint: ~5 ms on Firecracker using 11 MB of memory
  - Includes Docker and podman by default

Unikernels

- Specialized, small, lightweight, single-address-space operating system with kernel included as library within application (aka library OS)
  - Sort of very lightweight VM specialized to single app: executable directly into kernel, resulting in monolithic process that runs entirely in kernel mode
  - Built by compiling high-level language directly into specialized machine image that runs directly on hypervisor
  - Goal: isolation benefits of hypervisor without overhead of guest OS



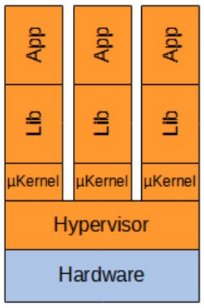
Unikernels: pros and cons

- Pros from specialized = high performance
  - ✓ Lightweight and small (minimal memory footprint)
  - ✓ Fast app execution (no context switching)
  - ✓ Fast boot (measured in ms)
  - ✓ Secure (reduced attack surface)
- Cons:
  - ✗ Engineering effort in order to port app to unikernel
  - ✗ Limited debugging tools
  - ✗ Single language runtime
  - ✗ Early unikernel frameworks required to write app from scratch

    See www.youtube.com/watch?v=oHcHTFleNtg
- Good news: cons almost solved with recent unikernel frameworks

Unikernels: frameworks

- Frameworks (and programming language):
  - MirageOS (OCaml)
  - OSv (C, C++, Go, Python, Java, Rust, …)
  - Unikraft
  - Unikernel Linux: integrate unikernel optimization techniques in Linux (others are clean slate)



Osv unikernel

- OSv
  - Unikernel designed to run single unmodified Linux application securely as microVM on top of hypervisor (e.g., KVM, Xen, VMWare, Virtualbox, Firecracker)
  - Linux binary compatible unikernel
  - To run app on OSv, one needs to build an image by fusing OSv kernel and app files together (Capstan tool)
  - Open-source and fast
    - Can boot in ~5 ms on Firecracker using 11 MB of memory

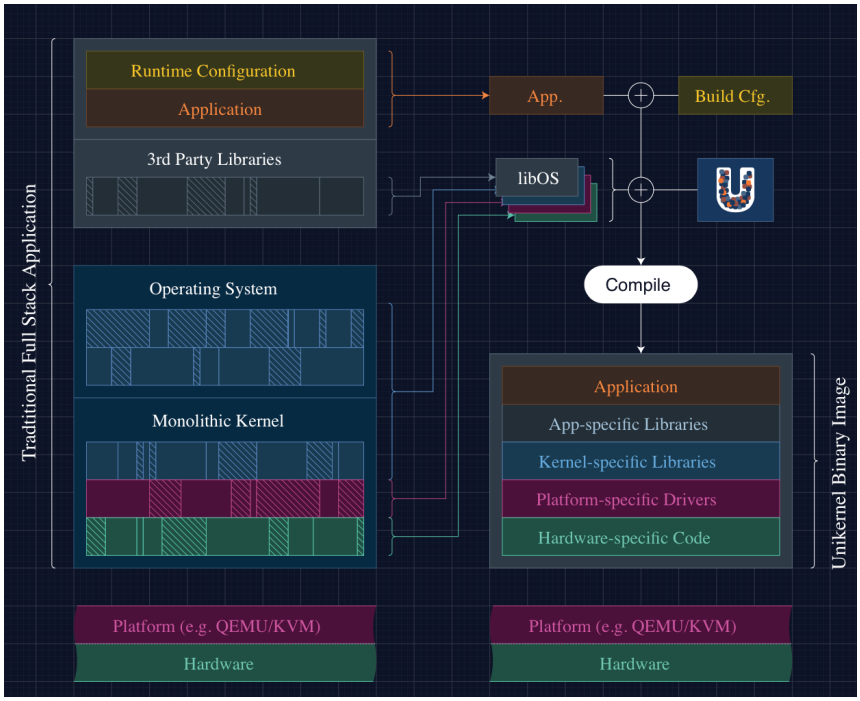
Unikernels: frameworks

- Unikraft
  - Fast, secure and open-source Unikernel Development Kit
  - Goal: build unikernels easily, quickly and without time-consuming expert work
  - Supports multiple hypervisors (e.g., Xen and KVM) and CPU architectures
  - Ability to run wide range of apps (even complex: Redis, Nginx, Memcached) and languages
  - POSIX compliant
  - Written in C

Kuenzer et al., Unikraft: fast, specialized unikernels the easy way, EuroSys 2021

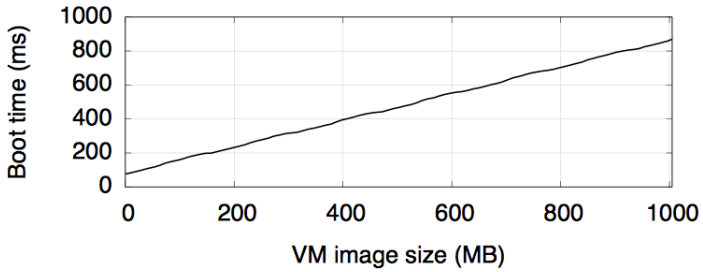
Unikernels: frameworks

- Unikraft

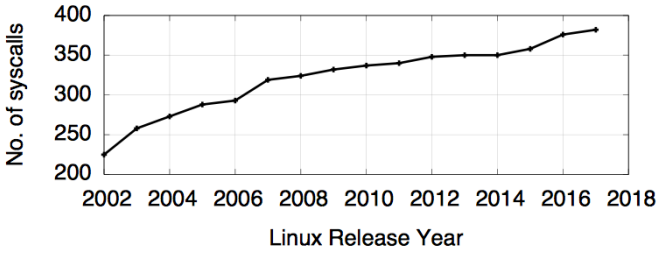


Performance of virtualization approaches

- VM boot times grow linearly with VM size



- Difficulties in securing containers due to growth of Linux syscall API



My VM is lighter (and safer) than your container, SOSP'17

Performance of virtualization approaches

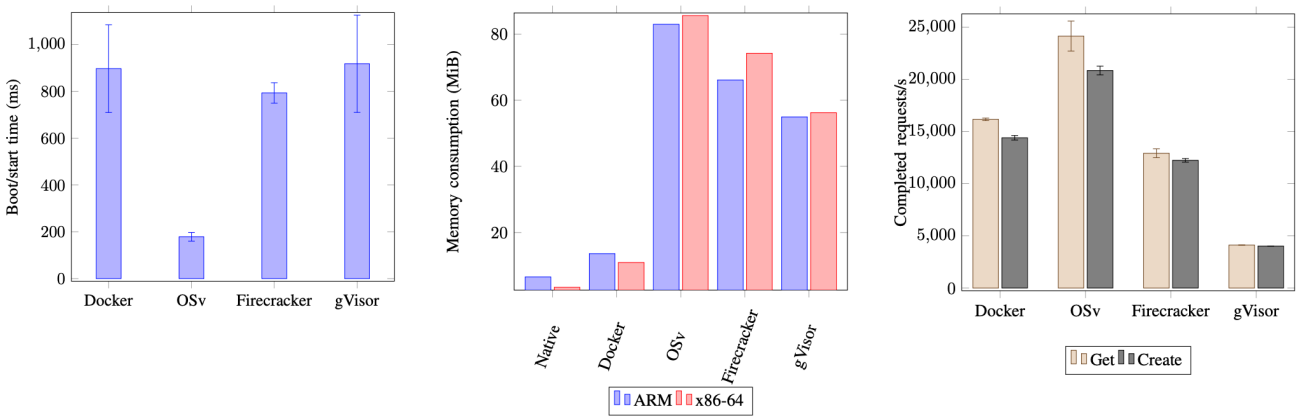
- Performance studies compare hypervisor vs. lightweight virtualization
- Overall result: overhead introduced by containers is almost negligible
  - Fast instantiation time
  - Small per-instance memory footprint
  - High density
- … but paid in terms of security

Virtualization	Boot time	Image size	Memory footprint	Programming language dependance	Live migration support
VM	~5/10 sec	~1 GB	~100 MB	No	Yes
Container	~0.8/1 sec	~50 MB	~5 MB	No	Non-native
Unikernel	<10 msec	<20 MB	~10 MB	Partially	No

From: Consolidate IoT edge computing with lightweight virtualization, 2018

Performance of virtualization approaches

- Lightweight virtualization methods are needed for edge computing and compute continuum
- Overall result: no clearly winning solution so far, each one has its own strengths and weaknesses



Source: A functional and performance benchmark of lightweight virtualization platforms for edge computing, EDGE 2022

Even newer: WebAssembly

- WebAssembly (Wasm): portable, binary instruction format for a stack-based VM
  - Enables memory-safe, sandboxed execution
  - Designed as a portable compilation target for programming languages, enabling deployment on web for client and server apps
    - Born as alternative to execute JavaScript code in browsers
  - Allows to write in a variety of languages, compile them to Wasm, and execute them in a Wasm runtime
  - Efficiency: executes at near-native speed
  - Security: Wasm uses software-based fault isolation techniques to sandbox the executing module
    - Wasm interacts with the host system via the WebAssembly System Interface (WASI)
    - Wasm module cannot directly perform an OS system call due to sandboxing, but imports equivalent WASI functions instead

WebAssembly: an example

- Factorial function written in C and its corresponding WebAssembly code after compilation
  - In .wat text format (human-readable textual representation of Wasm) and in .wasm binary format

C source code

```
int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

WebAssembly .wat text format

```
(func (param i64) (result i64)
  local.get 0
  i64.eqz
  if (result i64)
      i64.const 1
  else
      local.get 0
      local.get 0
      i64.const 1
      i64.sub
      call 0
      i64.mul
  end)
```

WebAssembly .wasm binary format

```
00 61 73 6D 01 00 00 00
01 06 01 60 01 7E 01 7E
03 02 01 00
0A 17 01
15 00
20 00
50
04 7E
42 01
05
20 00
20 00
42 01
7D
10 00
7E
0B
0B
```


References

- Section 3.2 of van Steen & Tanenbaum book
- Smith and Nair, The architecture of virtual machines, IEEE Computer, 2005
- Bugnion et al., Hardware and software support for virtualization, 2017
- Hwang et al., Virtual machines and virtualization of clusters and data centers, 2011
- Agache et al., Firecracker: Lightweight virtualization for serverless applications, 2020
- Kuenzer et al., Unikraft: fast, specialized unikernels the easy way, 2021
- Morabito et al., Consolidate IoT edge computing with lightweight virtualization, 2018