TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Virtualization

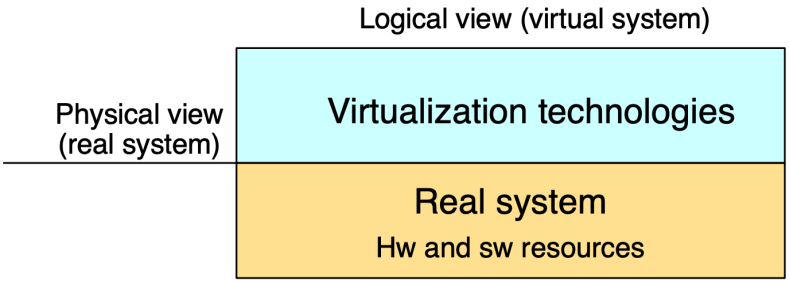
Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2022/23

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Virtualization

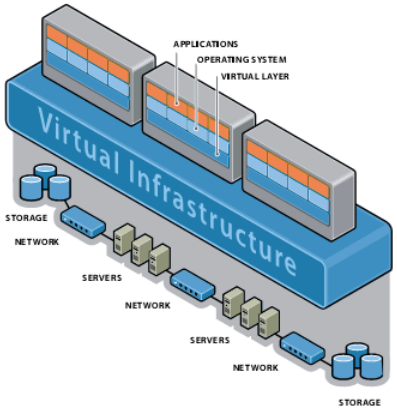
- High-level abstraction to hide details of underlying implementation
- Abstraction of computing resources
  - Logical view different from physical one



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

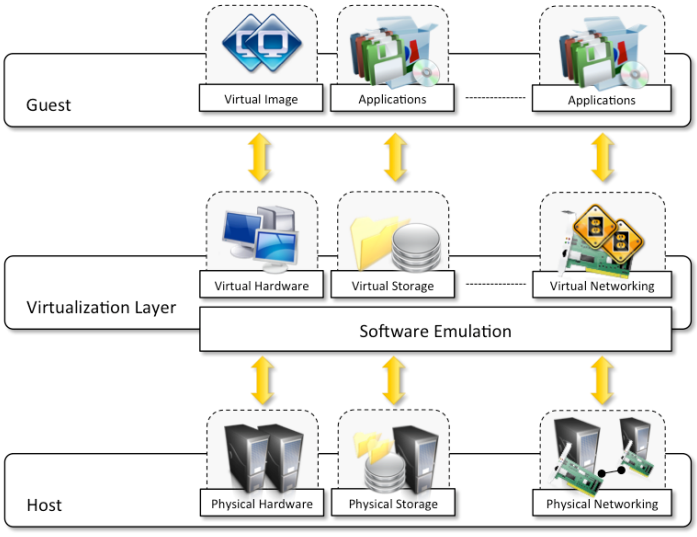
Virtualization of resources

- System (hw and sw) resources virtualization
  - Virtual machine, container, …
- Storage virtualization
  - Storage Area Network (SAN), …
- Network virtualization
  - Virtual LAN (VLAN), Virtual Private Network (VPN), …
- Data center virtualization



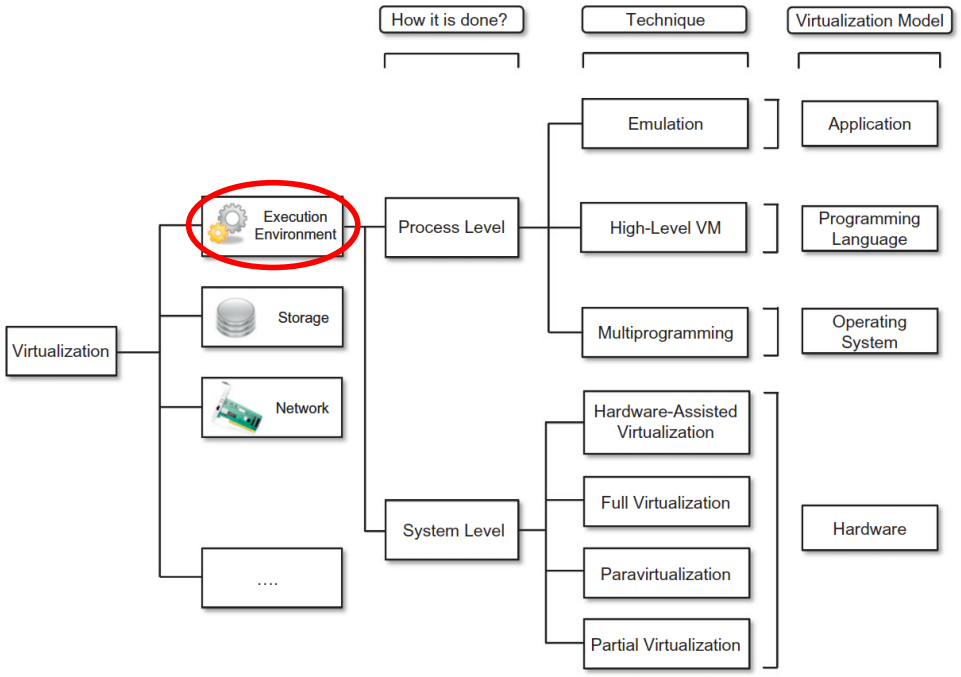
Components of virtualized environment

- 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

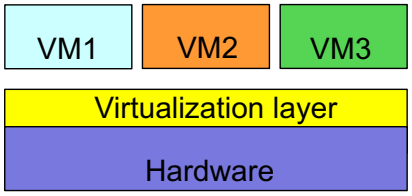
Taxonomy of virtualization techniques



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

Virtual Machine

- Virtual Machine (**VM**) allows to represent hw/sw resources of a machine differently from their reality
  - E.g., VM hw resources (CPU, network card, ...) different from physical resources of the real machine
  - E.g., sw resources (OS, …) different from sw resources of the real machine
- A single physical machine can be used to host several computing environments
  - Multiple VMs on the same physical machine



Virtualizzazione: cenni storici

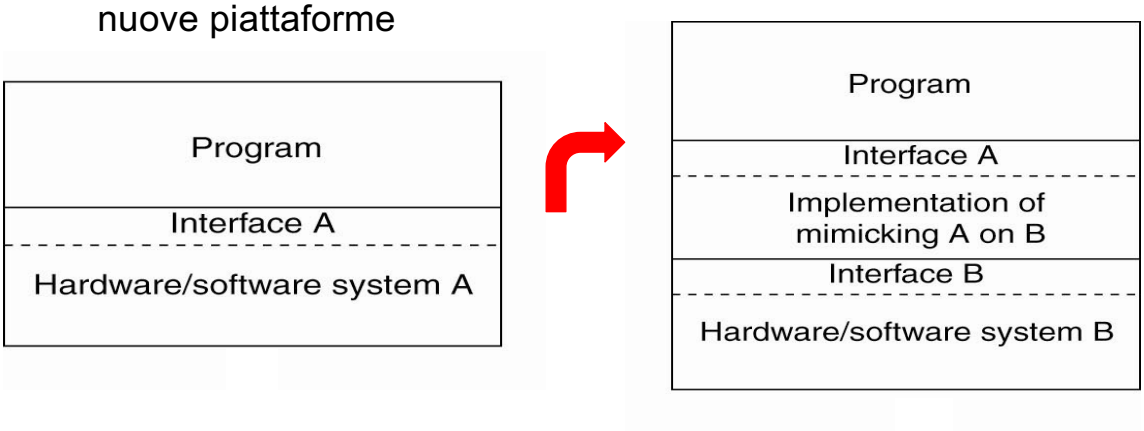
- Il concetto di VM è un'idea "vecchia", essendo stato definito negli anni '60 in un contesto centralizzato
  - Ideato per consentire al software *legacy* (esistente) di essere eseguito su mainframe molto costosi e condividere in modo trasparente le (scarse) risorse fisiche
  - Ad es. il mainframe IBM System/360-67
- Negli anni '80, con il passaggio ai PC il problema della condivisione trasparente delle risorse di calcolo viene risolto dai SO multitasking
  - L'interesse per la virtualizzazione svanisce

Virtualizzazione: cenni storici

- Alla fine degli anni '90, l'interesse rinasce per rendere meno onerosa la programmazione hw special-purpose
  - VMware fondata nel 1998
- Si acuisce il problema dei costi di gestione e di sottoutilizzo di piattaforme hw e sw eterogenee
  - L'hw cambia più velocemente del sw (middleware e applicazioni)
  - Aumenta il costo di gestione e diminuisce la portabilità
- Diventa nuovamente importante la condivisione dell'hw e delle capacità di calcolo non usate per ridurre i costi dell'infrastruttura
- E' una delle tecnologie abilitanti del Cloud computing

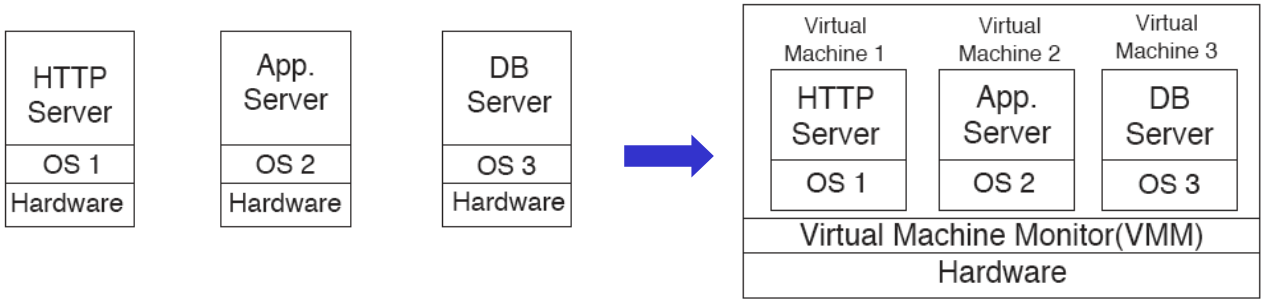
Virtualizzazione: vantaggi

- Facilita la compatibilità, portabilità, interoperabilità e migrazione di applicazioni ed ambienti
  - Indipendenza dall'hw
  - Create Once, Run Anywhere
  - VM legacy: eseguire vecchi SO o vecchie applicazioni su nuove piattaforme



Virtualizzazione: vantaggi

- Permette il *consolidamento dei server* in un data center, con vantaggi economici, gestionali ed energetici
  - Multiplexing di molteplici VM sullo stesso server
  - Obiettivo: ridurre il numero totale di server usati, utilizzandoli in modo più efficiente
  - Vantaggi:
    - Riduzione di costi, consumi energetici e spazio occupato
    - Semplificazione nella gestione, manutenzione ed upgrade dei server
    - Riduzione dei tempi di downtime, tramite live migration di VM



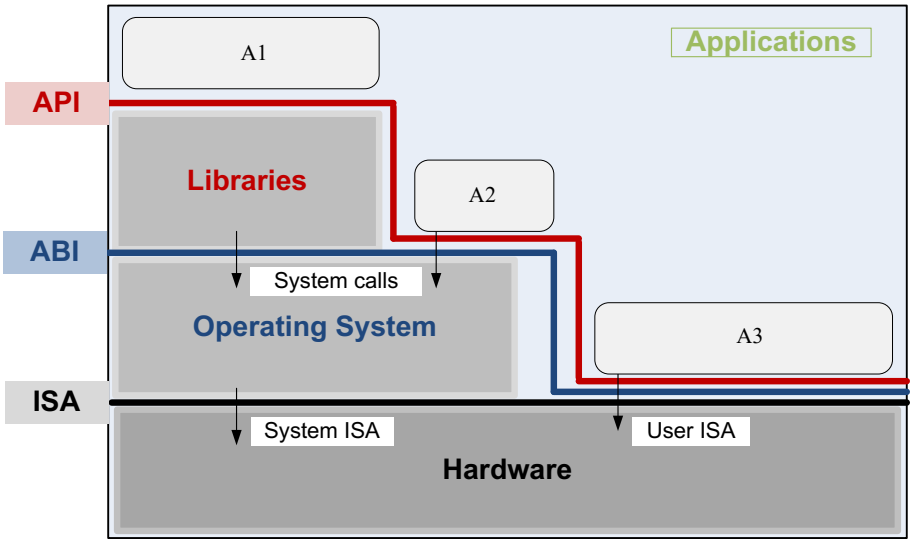
Virtualizzazione: vantaggi

- Permette di isolare componenti malfunzionanti o soggetti ad attacchi di sicurezza, incrementando affidabilità e sicurezza delle applicazioni
  - Macchine virtuali di differenti applicazioni non possono avere accesso alle rispettive risorse
  - Bug del software, crash, virus in una VM non possono danneggiare altre VM in esecuzione sulla stessa macchina fisica
- Permette di isolare le prestazioni di diverse VM
  - Tramite lo scheduling delle risorse fisiche che sono condivise tra molteplici VM in esecuzione sulla stessa macchina fisica
- Permette di bilanciare il carico sui server
  - Tramite la migrazione della VM da un server ad un altro

Uso di ambienti di esecuzione virtualizzati

- In ambito personale e didattico
  - Per eseguire simultaneamente diversi SO sulla stessa macchina
  - Per semplificare l'installazione di sw
- In ambito professionale
  - Per debugging, testing e sviluppo di applicazioni
- In ambito aziendale
  - Per consolidare l'infrastruttura del data center
  - Per garantire *business continuity*: incapsulando interi sistemi in singoli file (system image) che possono essere replicati, migrati o reinstallati su qualsiasi server

Interfaces in computer system



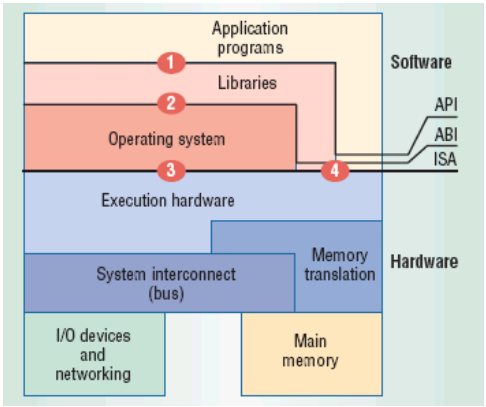
Applications:

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

Interfaces in computer system and virtualization

At which level can virtualization be implemented?

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



- Virtualization goal: mimic behavior of these interfaces

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

Levels of virtualization implementation

- 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*)

Levels of virtualization implementation

- 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

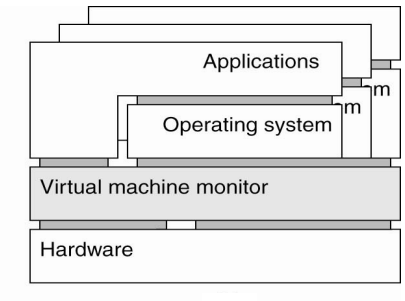
Levels of virtualization implementation

- 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

Levels of virtualization implementation

- 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>



Levels of virtualization implementation

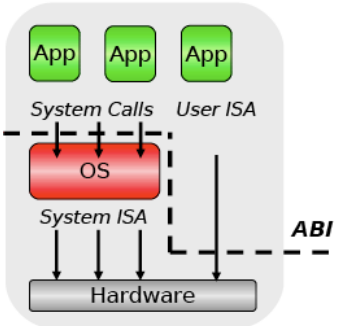
- Operating system level (aka *containers*)
  - Goal: create multiple isolated containers
  - Examples: Docker, Linux Containers

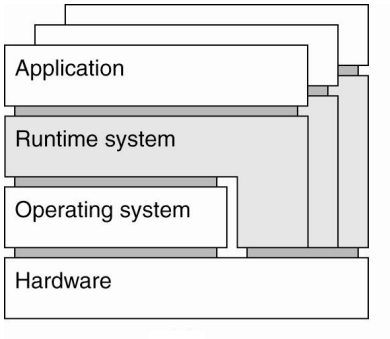
- Library level
  - Goal: create execution environment to run apps in a host environment that does not suite native apps
    - Rather than creating a VM to run full OS and apps
  - Examples:
    - Wine: runs Windows apps on top of POSIX-compliant OS by translating Windows API calls into POSIX calls on-the-fly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

Levels of virtualization implementation

- User application level (aka *process VMs*)
  - Virtual platform that executes an individual process
  - Provides virtual ABI or API environment for user applications
  - Application 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>





Levels of virtualization implementation: 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

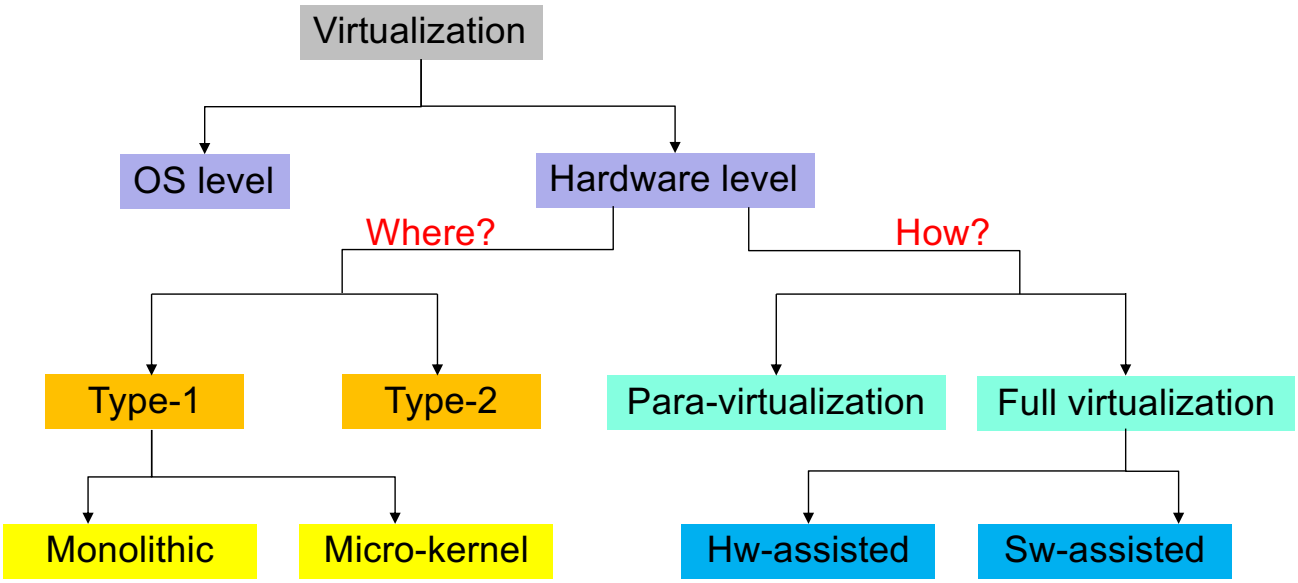
- Let's focus on **system-level virtualization** (achieved through VMM or hypervisor)

- **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 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*
     - **Para-virtualization**

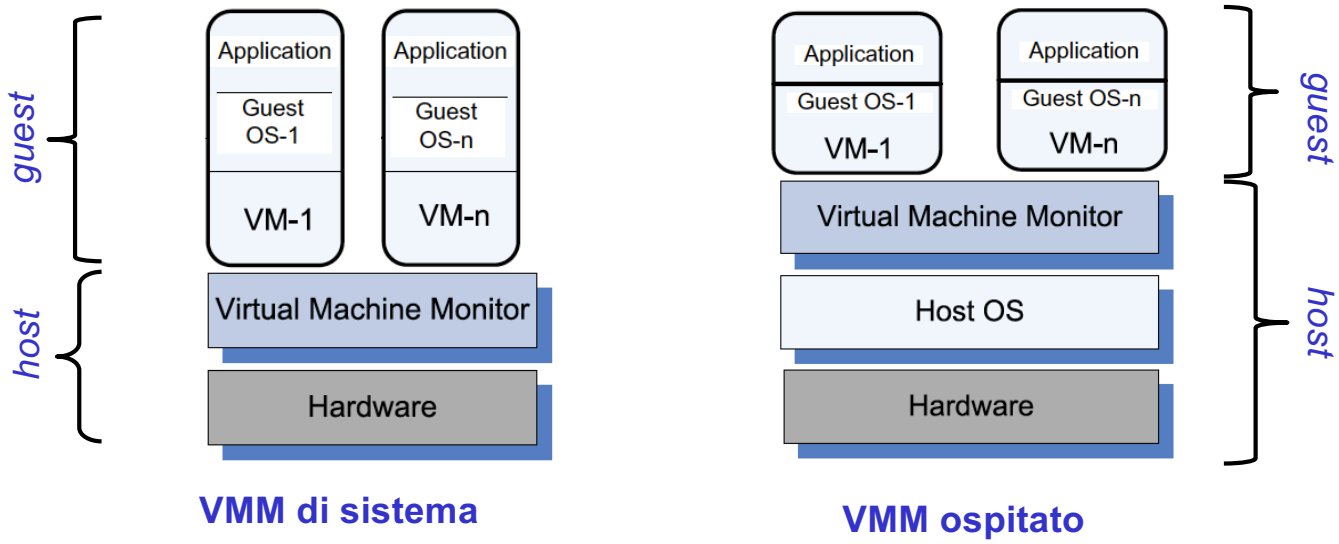
System-level virtualization: taxonomy



VMM di sistema o VMM ospitato

In quale livello dell'architettura di sistema si colloca il VMM?

- Direttamente sull'hardware: **VMM di sistema**
- Sopra il SO host: **VMM ospitato**



VMM di sistema o VMM ospitato

- **VMM di sistema** (type-1): eseguito direttamente sull'hw, offre funzionalità di virtualizzazione integrate in un SO semplificato
  - L'hypervisor può avere un'architettura a *microkernel* (solo funzioni di base, no device driver) o *monolitica*
  - Esempi: Xen, KVM, VMware ESX, Hyper-V
- **VMM ospitato** (type-2): eseguito sul SO host, accede alle risorse hw tramite le chiamate di sistema del SO host
  - Interagisce con il SO host tramite l'ABI ed emula l'ISA di hw virtuale per i SO guest
  - Vantaggio: può usare il SO host per gestire le periferiche ed utilizzare servizi di basso livello (es. scheduling delle risorse)
  - Vantaggio: non occorre modificare il SO guest
  - Svantaggio: degrado delle prestazioni rispetto a VMM di sistema
  - Esempi: Bochs, Parallels Desktop, VirtualBox

Virtualizzazione completa o paravirtualizzazione

Quale modalità di dialogo tra la VM ed il VMM per l'accesso alle risorse fisiche, ovvero come gestire l'esecuzione di istruzioni privilegiate?

- **Virtualizzazione completa**
- **Paravirtualizzazione**

Confronto qualitativo di diverse soluzioni di virtualizzazione
https://en.wikipedia.org/wiki/Comparison_of_platform_virtualization_software

Virtualizzazione completa o paravirtualizzazione

- Virtualizzazione completa (full)
  - Il VMM espone ad ogni VM interfacce hw simulate *funzionalmente identiche* a quelle della sottostante macchina fisica
  - Il VMM *intercetta* le richieste di accesso privilegiato all'hw (ad es. istruzioni di I/O) e ne *emula* il comportamento atteso
  - Esempi: KVM, VMware ESXi, Microsoft Hyper-V
- Paravirtualizzazione
  - Il VMM espone ad ogni VM interfacce hw simulate *funzionalmente simili* (ma non identiche) a quelle della sottostante macchina fisica
  - Non viene emulato l'hw, ma viene creato uno strato minimale di sw (Virtual Hardware API) per assicurare la gestione delle VM ed il loro isolamento
  - Esempi: Xen, Oracle VM (basato su Xen), PikeOS

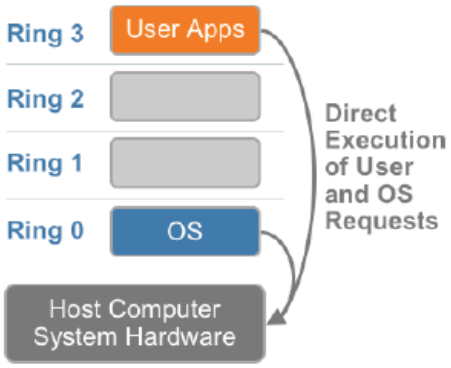
Virtualizzazione completa: pro e contro

- Vantaggi
  - Non occorre modificare il SO guest
  - Isolamento completo tra le istanze di VM: sicurezza, facilità di emulare diverse architetture

- Svantaggi
  - VMM più complesso
  - Necessaria la collaborazione del processore per implementazione efficace: perché?

Problemi per realizzare la virtualizzazione di sistema

- L'architettura non virtualizzata del processore opera secondo almeno 2 livelli (ring) di protezione: *supervisor* e *user*
  - Ring 0: privilegi massimi
  - Ring 3: privilegi minimi

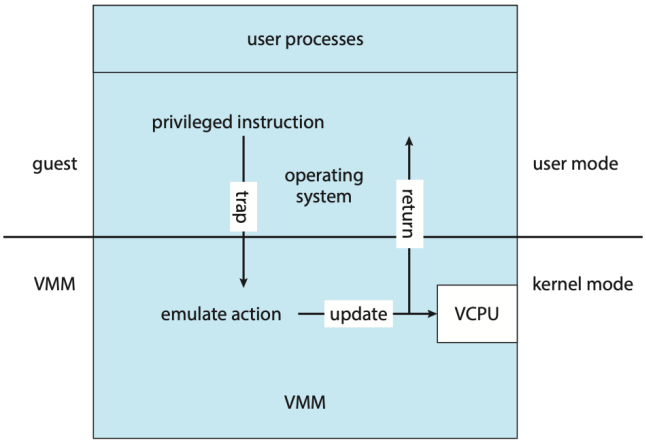


Architettura x86 *senza virtualizzazione*

- Con la virtualizzazione:
  - VMM opera in supervisor mode (ring 0)
  - SO guest e applicazioni (quindi la VM) operano in user mode (ring 3 o ring 1 per SO guest)
  - Problema del ring deprivileging: il SO guest opera in un ring che non gli è proprio ⇒ non può eseguire istruzioni privilegiate (e.g., `lidt` in x86, load interrupt descriptor table)
  - Problema del ring compression: poiché applicazioni e SO guest eseguono allo stesso livello, occorre proteggere lo spazio del SO

Virtualizzazione completa: soluzioni

- Come risolvere il ring deprivileging?
  - **Trap-and-emulate**: quando il SO guest tenta di eseguire un'istruzione privilegiata, occorre notificare un'eccezione (trap) al VMM e trasferirgli il controllo; il VMM controlla la correttezza dell'operazione richiesta e ne esegue ("emula") il comportamento
  - Le istruzioni non privilegiate eseguite dal SO guest sono invece eseguite direttamente



Popek and Goldberg virtualization requirements

- Popek and Goldberg (1974) defined a set of conditions sufficient for a computer architecture to support system virtualization *efficiently*
- *Privileged instructions*: cause a trap if executed in user mode
  - Privileged state: determines resource allocation (privilege mode, addressing context, exception vectors, ...)
- *Sensitive instructions*: change underlying resources (e.g., doing I/O or changing page tables) or observe information that indicates current privilege level (thus exposing that guest OS does not run on bare metal)
  - Can be control- or behavior-sensitive
    - control sensitive: changes privileged state
    - behavior sensitive: exposes privileged state
- *Innocuous instructions*: not sensitive

Popek and Goldberg virtualization requirements

- **Theorem**: “For any conventional third-generation computer, an effective VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.”
- In other words... trap-and-emulate: it is sufficient that all the instructions that could affect the correct functioning of VMM (sensitive instructions) always trap and pass control to VMM
- Seems easy but…

Popek and Goldberg virtualization requirements

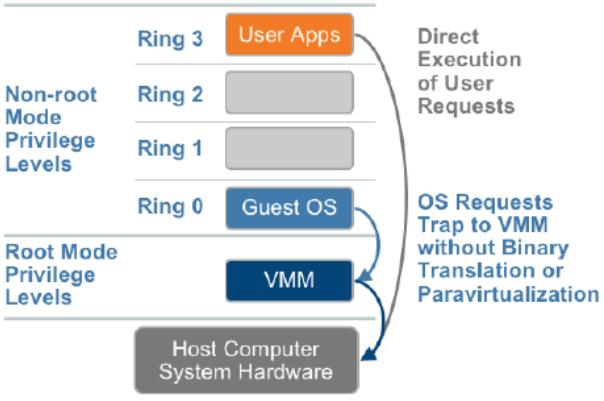
- Implementation of trap-and-emulate is challenging: common architectures are non-virtualizable according to Popek and Goldberg’s theorem
  - x86: many instructions are non-virtualizable, because are *sensitive but un-privileged*
    - E.g., `pushf` (push flags) is not privileged
  - MIPS: mostly virtualizable, but...
    - Kernel registers `$k0`, `$k1` (needed to save/restore state) are user-accessible
  - ARM: mostly virtualizable, but
    - Some instructions are undefined in user-mode

Virtualizzazione completa: soluzioni

- Come realizzare il meccanismo di trap?
  - A livello hardware se il processore fornisce supporto alla virtualizzazione ⇒ **hardware-assisted CPU virtualization**
  - A livello software se il processore non fornisce supporto alla virtualizzazione ⇒ **fast binary translation**

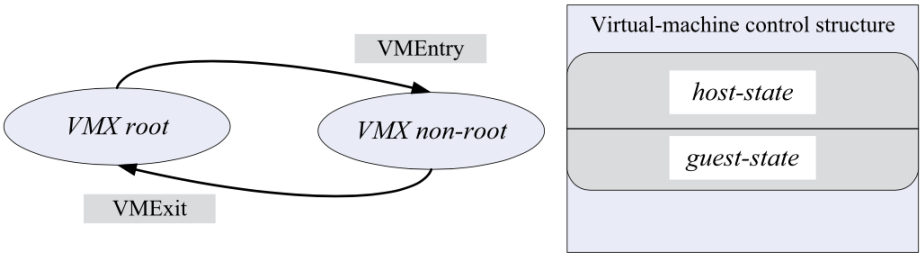
Hardware-assisted CPU virtualization

- 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 problems
  - VMM can control guest execution through VM control data structures in memory



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

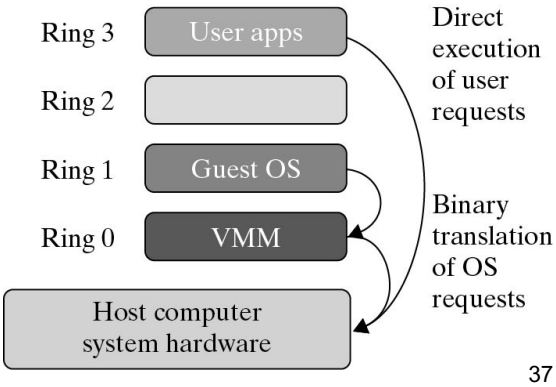
Hardware-assisted CPU virtualization: VT-x



- 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

- Il meccanismo di trap al VMM per le istruzioni privilegiate è offerto solo dai processori con supporto hardware per la virtualizzazione (Intel VT-x e AMD-V)
  - IA-32 non lo è: come realizzare la virtualizzazione completa in mancanza del supporto hw?
- Fast binary translation: il VMM scansiona il codice prima della sua esecuzione per sostituire blocchi contenenti istruzioni privilegiate con blocchi funzionalmente equivalenti e contenenti istruzioni per la notifica di eccezioni al VMM
  - I blocchi tradotti sono eseguiti direttamente sull'hw e conservati in una cache per eventuali riusi futuri
  - Maggiore complessità del VMM e minori prestazioni

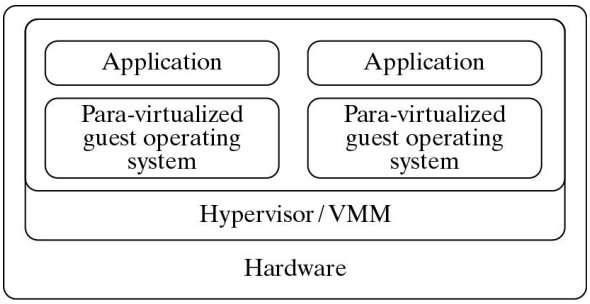


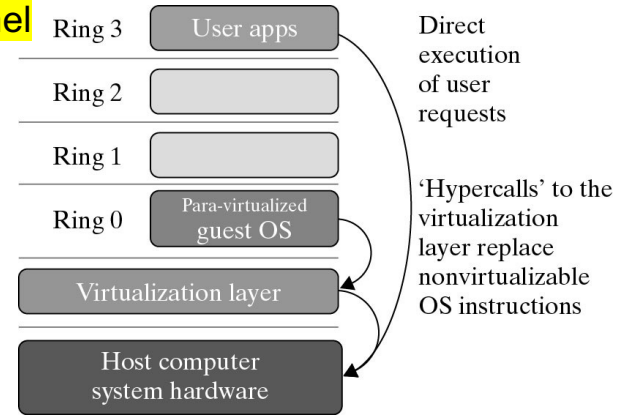
Architettura x86 con virtualizzazione completa e *fast binary translation*

Paravirtualization

- Non-transparent virtualization solution
  - Guest OS kernel must be modified to let it invoke the virtual API exposed by hypervisor
- 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

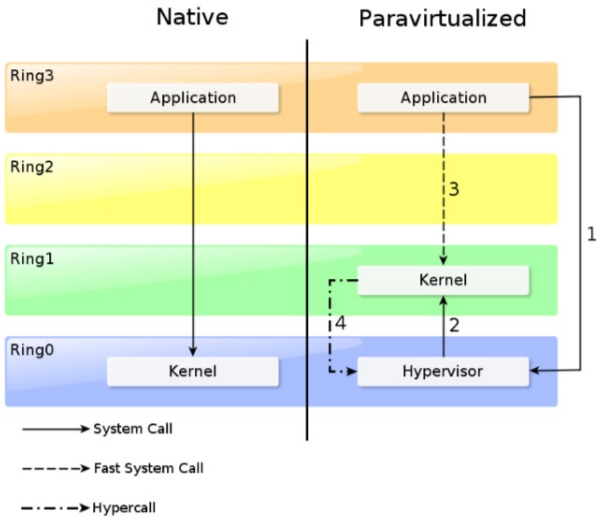
hypercall : hypervisor = syscall : kernel





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

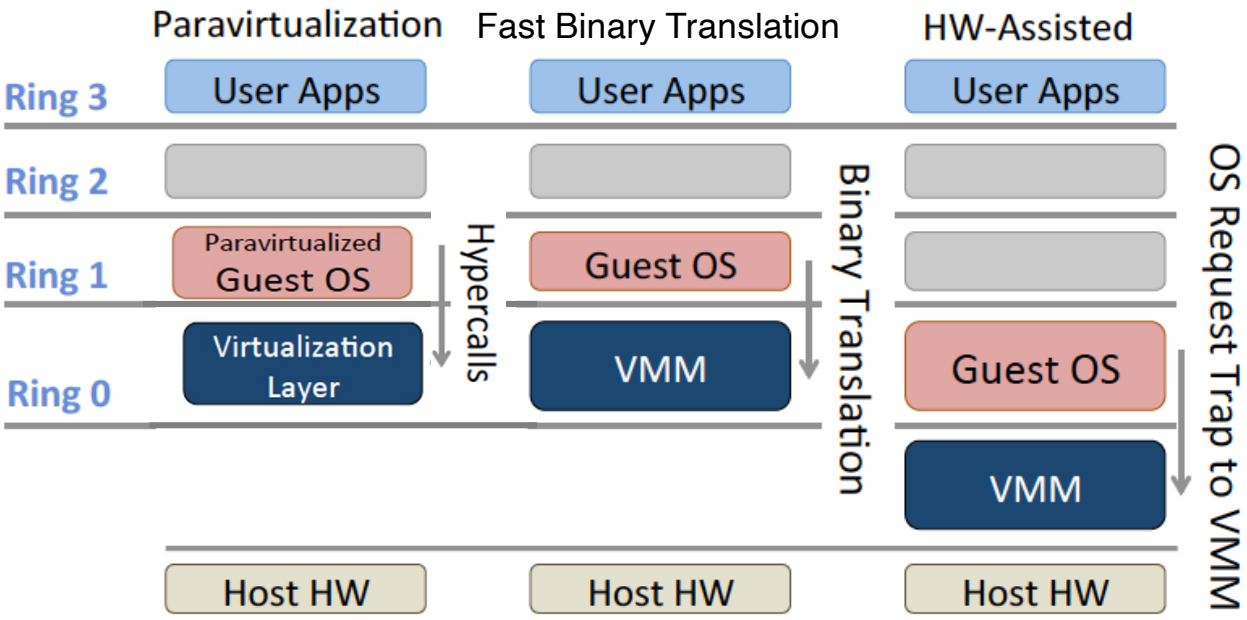
Source: "The Definitive Guide to XEN hypervisor"

Paravirtualization: pros & cons

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

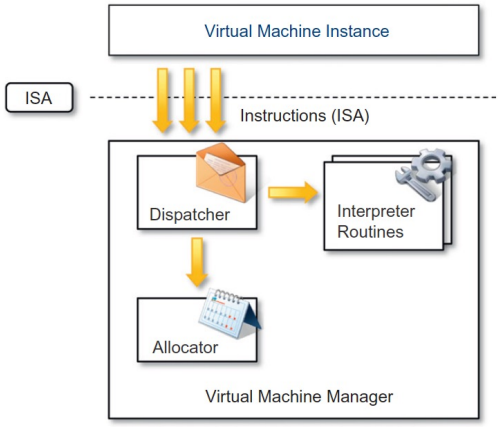
- Cons:
  - Requires source code of OSs to be available
    - OSs that cannot be ported (e.g., Windows) can use ad-hoc device drivers that remap the execution of critical instructions to the virtual API exposed by the VMM
  - Cost of maintaining paravirtualized OSs
    - Paravirtualized OS cannot longer run on hardware directly

Summing up different approaches



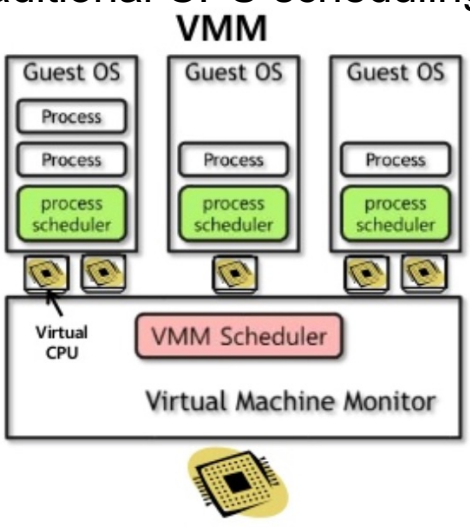
VMM reference architecture

- 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

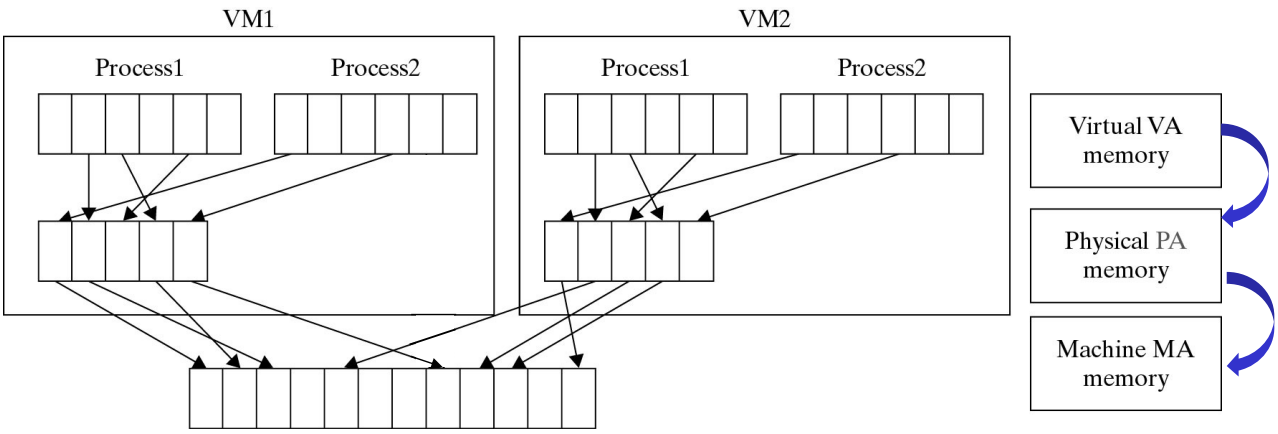


- How to schedule virtual CPUs on physical CPUs?

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

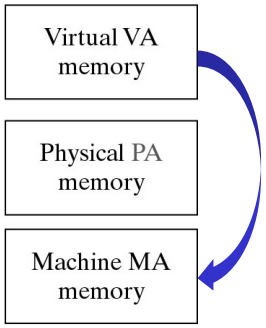


- Going from guest virtual memory to host physical memory requires two-level memory mapping

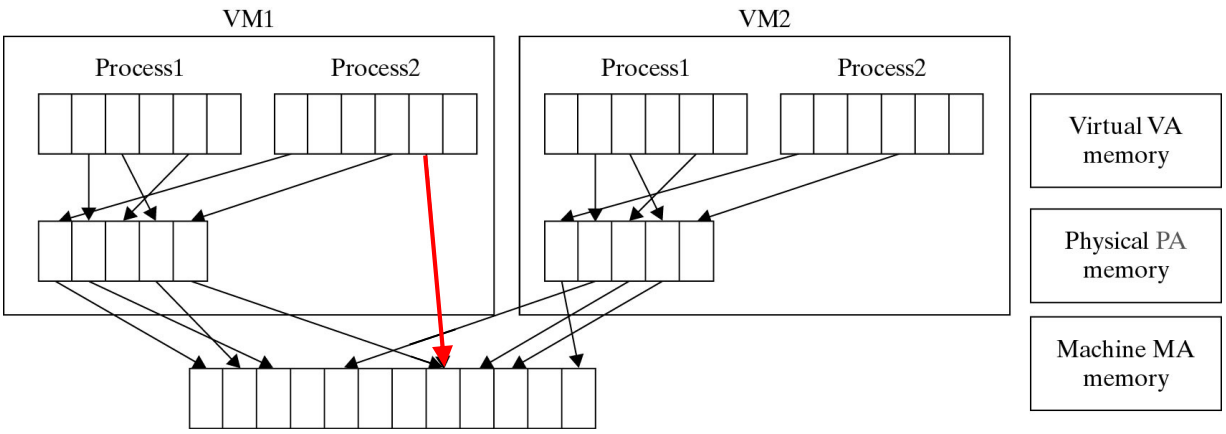
  GVA (guest virtual address) ➔ GPA (guest physical address) ➔ HMA (host machine address)
- Guest physical address ≠ host machine address: why?
  - Hints: many VMs; what does guest OS expect about its memory?

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



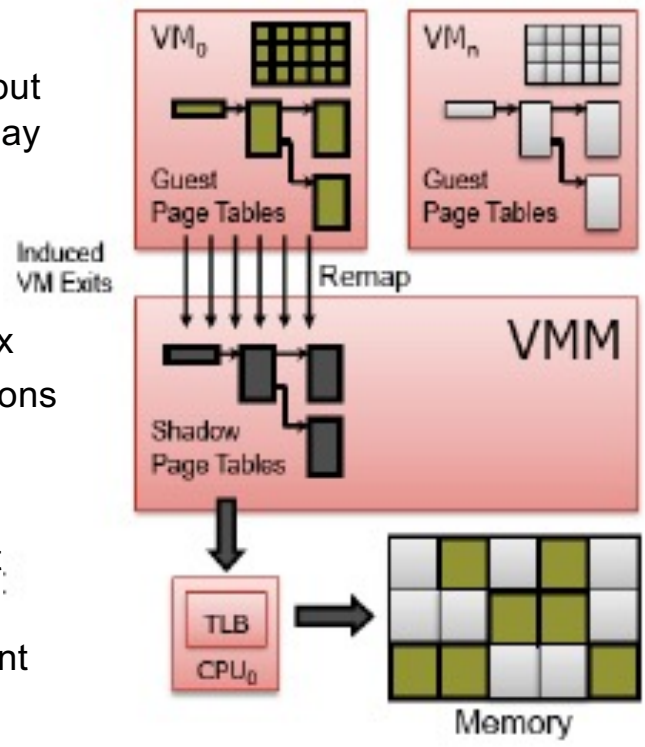
- VMM uses TLB hardware to map virtual memory directly to machine memory to avoid the two levels of translation on every access (red arrow)

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

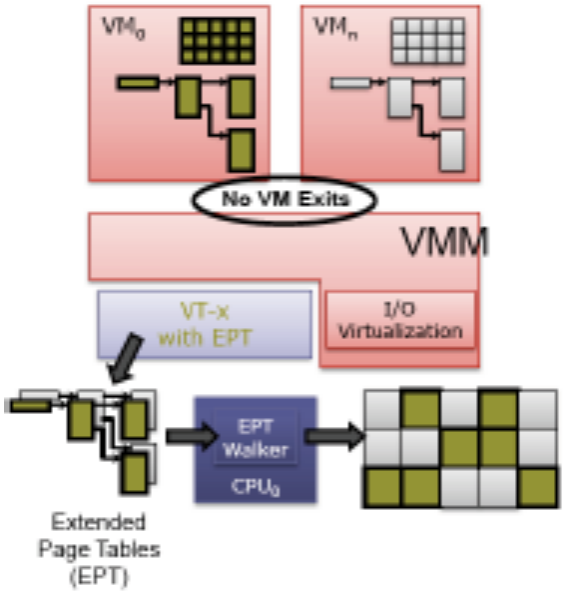
Challenges in memory virtualization with SPT

- 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 (developed at University of Cambridge)
  - 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 and OpenSolaris
    - With HVM also unmodified guest OSs (e.g., Windows)
  - Foundation for commercial virtualization products (e.g., Oracle VM and Qubes OS)
  - Powers IaaS providers (Alibaba, Amazon, IBM, Rackspace, …)
    - In 2017 Amazon began a shift to KVM for new EC2 instance types

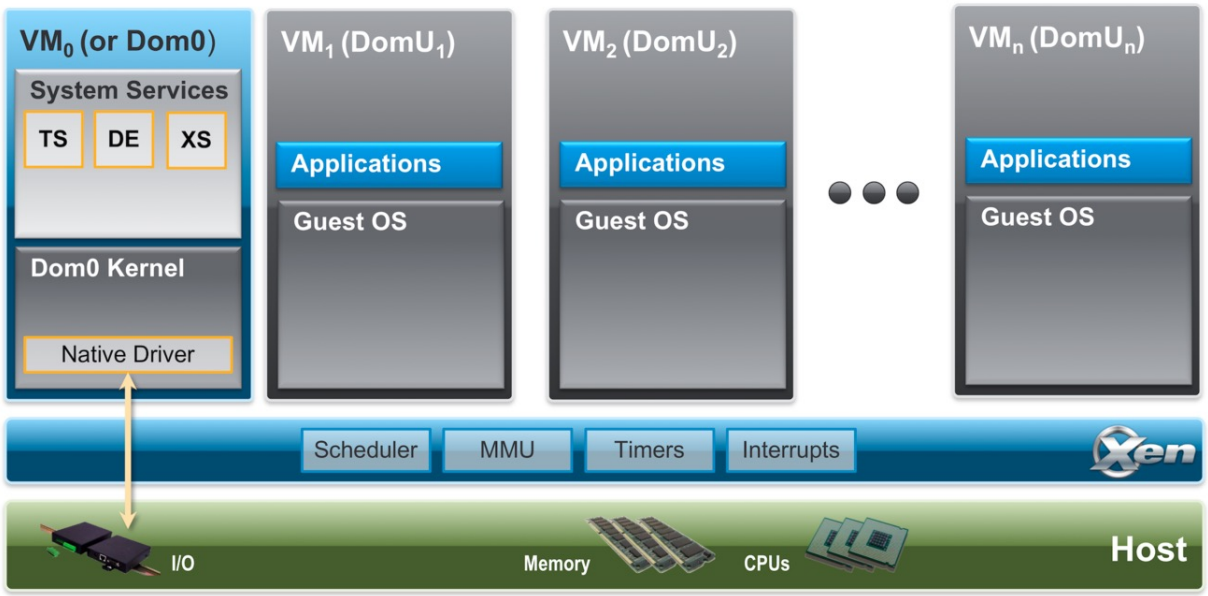
Xen: pros and cons

- Pros
  - 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 https://youtu.be/sjQnAIJji4k
    - But still vulnerable to attacks https://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
- Cons
  - I/O performance still remains challenging

Xen architecture

- Goal of Cambridge group who designed Xen (late 1990s, first release in 2003)
  - Design VMM 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

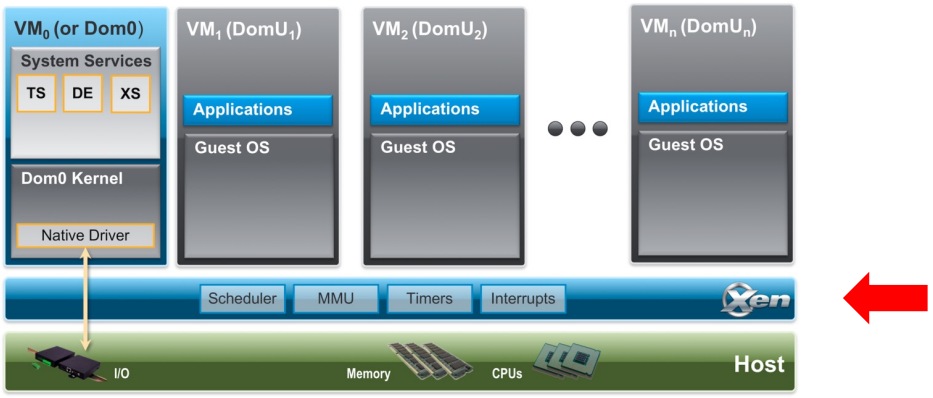
Xen architecture



https://wiki.xen.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 allocations 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 schedulers 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 one in Xen
- 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 is assigned a *weight* and optionally a *cap* (tunable parameters)
  - Weight: relative CPU 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 CPU that vCPU receives (e.g., 100 = 1 pCPU, 50 = 0.5 pCPU)
  - The scheduler transforms the weight into a *credit* allocation for each vCPU; as a vCPU runs, it consumes credits
- For each pCPU, the scheduler maintains a queue of vCPUs, with all the under-credit vCPUs first, followed by over-credit vCPUs; the scheduler picks the first vCPU in the queue
- Automatically load balances vCPUs across pCPUs on SMP host
  - Before a pCPU goes idle, it will consider other pCPUs in order to find any runnable vCPU; this approach guarantees that no pCPU idles when there is runnable work in the system wiki.xen.org/wiki/Credit_Scheduler

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 on the course site
  - "Old" studies but overall message 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* settings
- 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 (Citrix, Hyper-V, VMware, VirtualBox, …)

VM resizing and migration

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

VM dynamic resizing

- Fine-grain mechanism with respect to VM migrating or rebooting
  - Example: app running on a VM starts consuming a lot of resources and VM starts running out of RAM and CPU
  - Solution: dynamically resize the VM (aka warm resize)
- Pros: more cost-effective and faster than VM reboot
- Cons: 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

- To add or remove virtual CPUs (without VM turning off)
- Linux-based systems support CPU hot-plug/hot-unplug
  https://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:

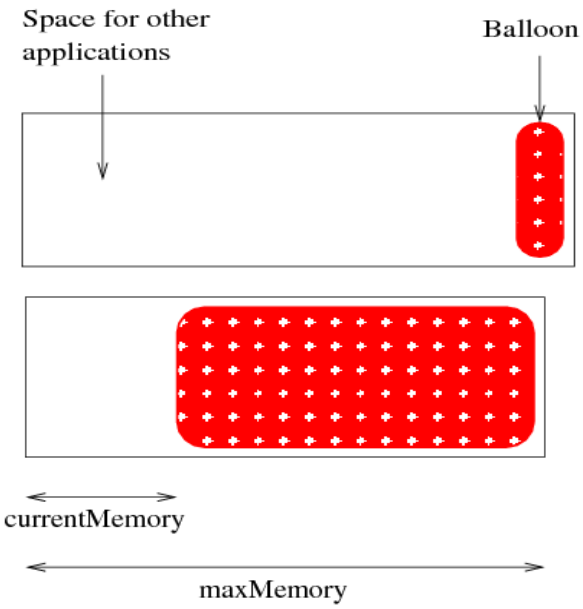
    ```
    echo 1 > /sys/devices/system/cpu/cpu5/online
    ```
  - To turn off cpu #5:

    ```
    echo 0 > /sys/devices/system/cpu/cpu5/online
    ```
- Can be controlled using virsh
  - Command line tool to configure and manage virtual machines, available with some hypervisors (KVM, Xen)
  - To set the number of vCPUs while VM is running (cannot exceed the maximum number of vCPUs)

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


VM dynamic resizing: memory

- Based on *memory ballooning*
  - Mechanism used by many 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



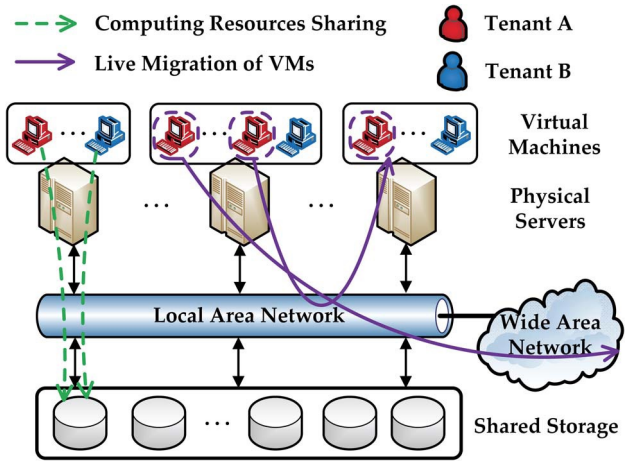
Migrazione di VM

- Vantaggi della migrazione
  - Utile in cluster e data center virtuali per:
    - Consolidare l'infrastruttura
    - Avere flessibilità nel failover
    - Bilanciare il carico
- Svantaggi e problemi
  - Supporto da parte del VMM
  - Overhead di migrazione non trascurabile
  - Migrazione in ambito WAN non banale

Migrazione di VM

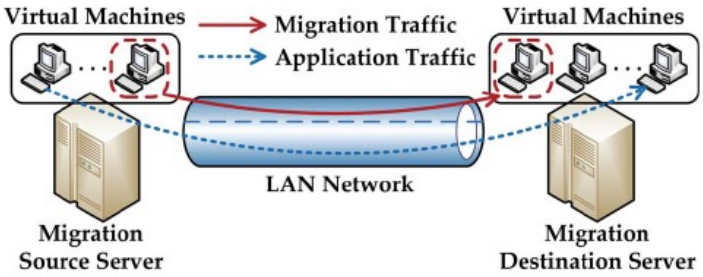
- Approcci per migrare istanze di macchine virtuali tra macchine fisiche:
  - **Stop and copy**: si spegne la VM sorgente e si trasferisce l'immagine della VM sull'host di destinazione, ma il downtime può essere troppo lungo
    - L'immagine della VM può essere grande e la banda di rete limitata
  - **Live migration**: la VM sorgente è in funzione durante la migrazione

Live migration largamente usata da Google: più di 1M di migrazioni al mese



Migrazione live di VM

- Prima di avviare la migrazione live
  - Fase di *setup*: si seleziona l'host di destinazione (ad es. con obiettivo di load balancing, energy efficiency, oppure server consolidation)
- Cosa migrare? Memoria, storage e connessioni di rete
- Come? In modo trasparente alle applicazioni in esecuzione sulla VM
  - Costo della migrazione live: vi è comunque un downtime dell'applicazione



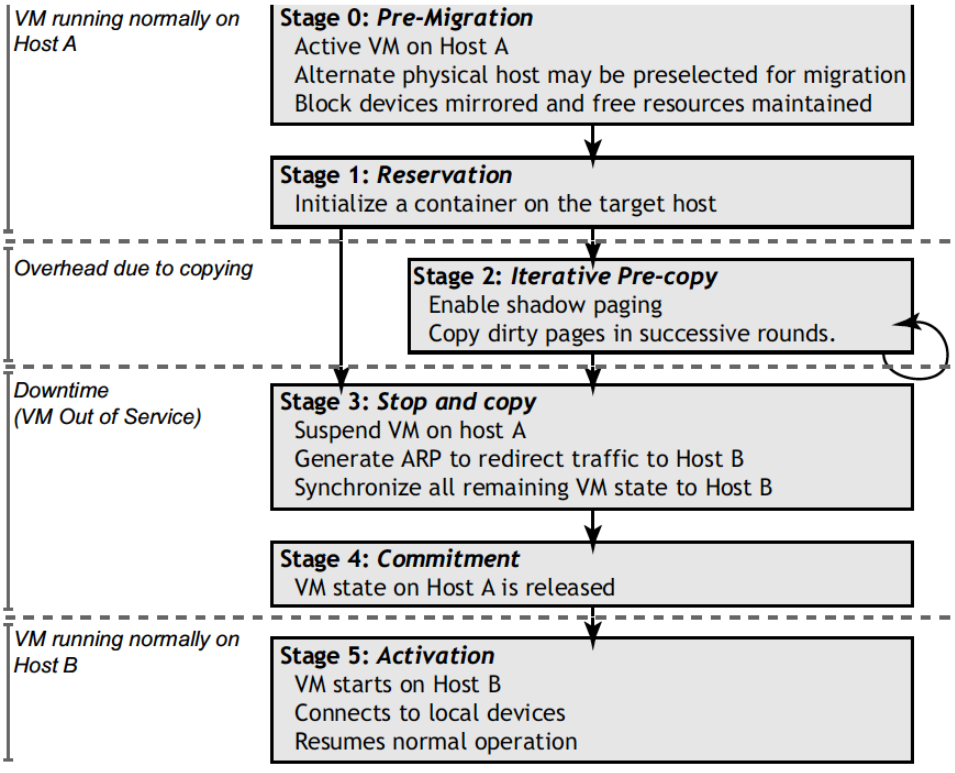
Migrazione live di VM: storage e rete

- Per migrare lo storage:
  - Usare storage condiviso da host sorgente e destinazione
    - SAN (Storage Area Network) o più economico NAS (Network Attached Server) o file system distribuito (e.g., NFS, GlusterFS o CEPH
  - In assenza di storage condiviso: il VMM sorgente salva tutti i dati della VM sorgente in un file di immagine, che viene trasferito sull'host di destinazione
- Per migrare le connessioni di rete:
  - La VM sorgente ha un indirizzo IP virtuale (eventualmente anche MAC virtuale)
    - Il VMM conosce il mapping tra IP virtuale e VM
  - Se host sorgente e destinazione sono su stessa sottorete IP, non occorre fare forwarding su host sorgente
    - Invio di risposta ARP non richiesta da parte dell'host destinazione per avvisare che l'indirizzo IP è stato spostato in una nuova locazione ed aggiornare quindi le tabelle ARP

Migrazione live di VM: memoria

- Per migrare la memoria (inclusi registri della CPU e stato dei device driver):
  1. Fase di *pre-copy*: il VMM copia in modo *iterativo* le pagine da VM sorgente a VM di destinazione *mentre* la VM sorgente è in esecuzione
     - All'iterazione *n* copiate le pagine modificate durante iterazione *n*-1
  2. Fase di *stop-and-copy*: la VM sorgente viene fermata e vengono copiate pagine *dirty*, stato della CPU e dei device
     - Tempo di downtime: da qualche msec a qualche sec, in funzione di dimensione della memoria, tipo di app e banda di rete
  3. Fasi di *commitment* e *reactivation*: la VM di destinazione carica lo stato e riprende l'esecuzione; la VM sorgente viene rimossa (ed eventualmente spento l'host sorgente)
- Noto come approccio **pre-copy**
  - La memoria è copiata *prima* che l'esecuzione della VM riprenda a destinazione
  - Soluzione comune (es. KVM, VMWare, Xen, Google CE)

VM live migration: overall process



C. 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

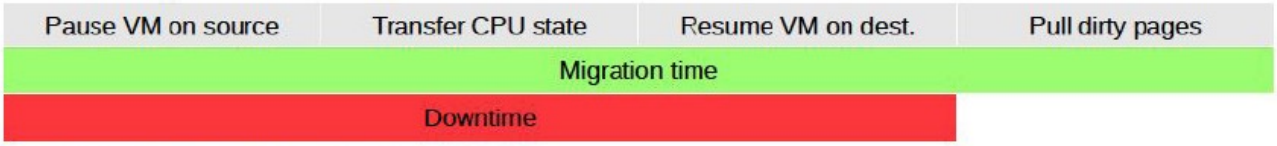
VM live migration: alternatives for memory

- **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 the destination, so to reduce app performance degradation after the 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

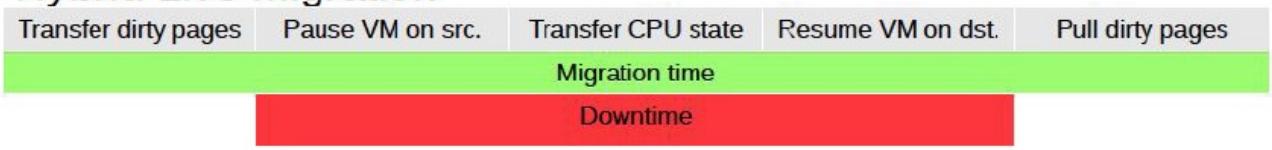
Approaches for migrating memory

- Pre-copy Live Migration

- Post-copy Live Migration



- Hybrid Live Migration



Courtesy of C.Vojtech, http://bit.ly/2h7wSWB

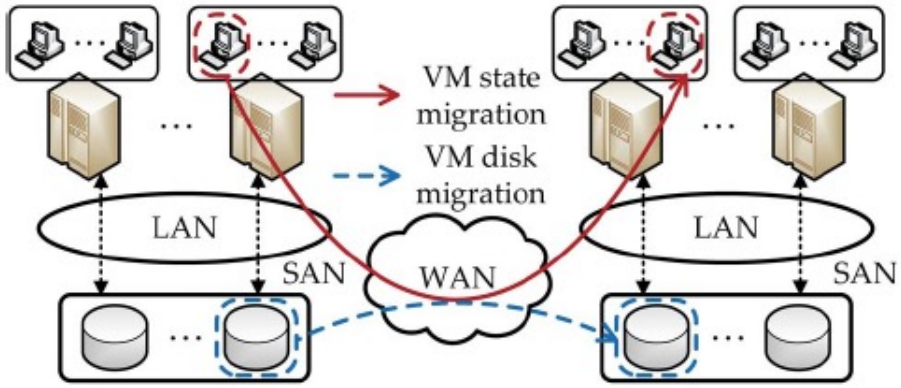
Live VM migration and hypervisors

- Live VM migration is supported by open-source and commercial hypervisors
  - E.g., KVM, Hyper-V, Xen, VirtualBox
- Can be controlled 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 environments

- How to achieve live migration of VMs 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 environments: storage

- Approaches to migrate storage in WAN
  - **Shared storage**
    - Cons: 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
    - Cons: does not work if source crashes
  - **Pre-copy/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 the write rate at the source is too fast, use write throttling to slow down the VM so that migration can proceed

VM migration in WAN environments: network

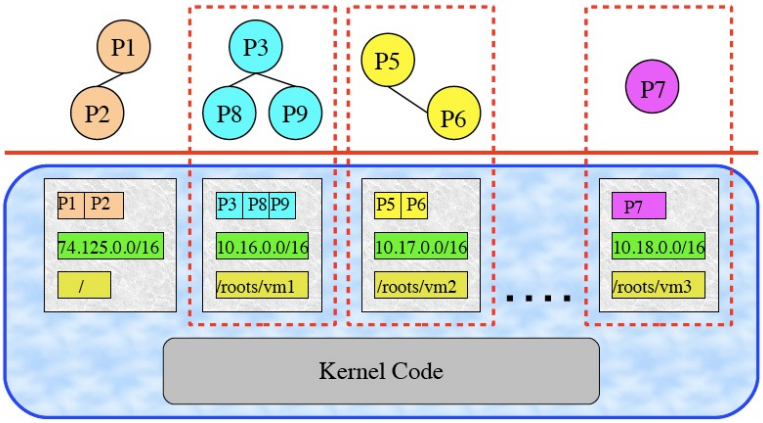
- Approaches to migrate network connections in WAN
  - **IP tunneling**
    - Set up an IP tunnel between the old IP address at source VM and the new IP address at destination VM
    - Use tunnel to forward all packets that arrive at source VM for the old IP address
    - Once migration has completed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

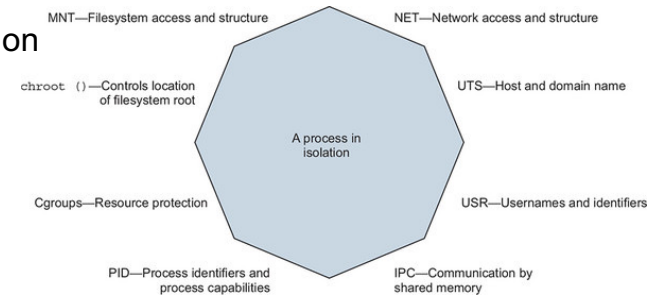
OS-level virtualization

- 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

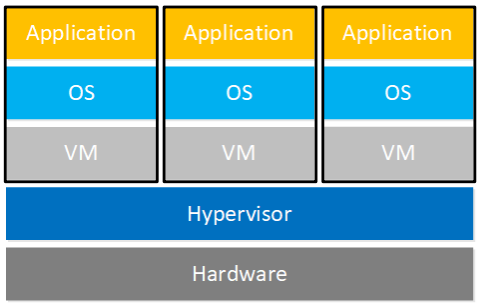
- mnt: isolates mount points
- pid: isolates the PID space, so that each process only sees itself and its children (PID 1, 2, 3, …)
- network: allows each container to have its dedicated network stack: its own private routing table, set of IP addresses, socket listing, firewall, and other network-related resources
- user: isolates user and group IDs, e.g., allowing a non-root user on the host to be mapped with the root user within the container
- uts (Unix timesharing): provides dedicated host and domain names
- ipc: provides dedicated shared memory for IPC, e.g., different Posix message queues

Mechanisms: cgroups

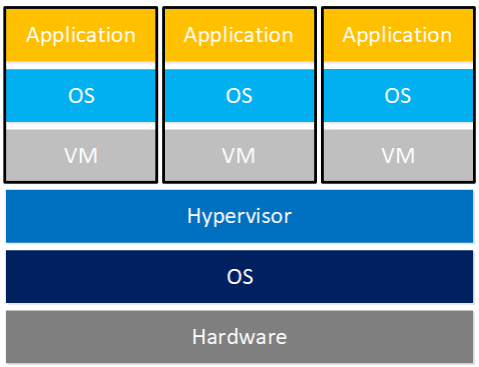
- cgroups: control groups
- Allows to limit, measure and isolate the use of hw resources (CPU, memory, block I/O, network) of a set of processes
- Low-level filesystem interface similar to `sysfs` and `procfs`
  - Default location in `/sys/fs/cgroup`

OS-level virtualization: pros

- VMM-based (type 1) vs container-based virtualization

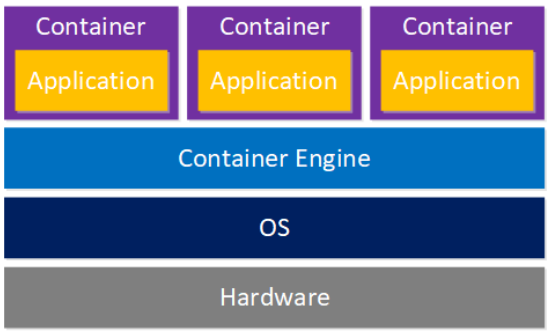


Type 1



Type 2

In a nutshell: lightweight vs. heavyweight



OS-level virtualization: pros

With respect to VMM-based virtualization

- 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

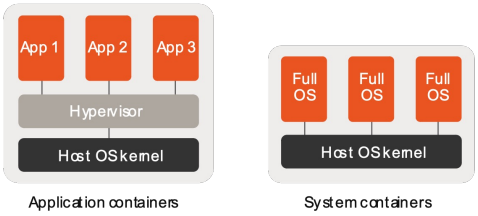
OS-level virtualization: cons

With respect to VMM-based virtualization

- 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
- Greater 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 *application container* engine
- FreeBSD Jail
- LXC (LinuX Containers)
  - Supported by mainline Linux kernel
  - Provides *system containers* (full OS image), while Docker provides *application containers*
  - LXD: system container manager built on top of LXC
- Podman
  - Supports Open Container Initiative (OCI) containers
  - Docker compatible CLI
- OpenVZ and Virtuozzo Containers

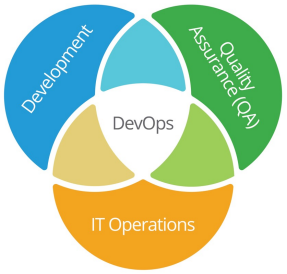


OS-level virtualization: only Linux?

- Windows and OS X support container-based virtualization
  - See Docker Desktop
- Alternative: install a VM with Linux as guest OS and then install a container-based virtualization product inside VM
  - ✗ Performance loss

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

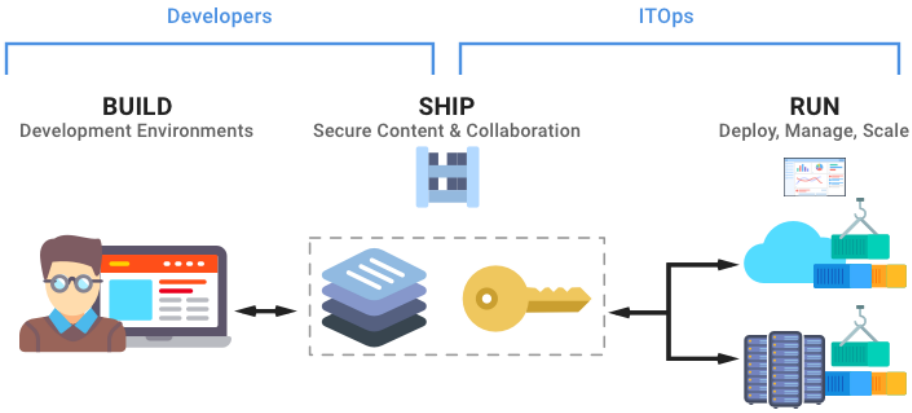


DevOps and CI/CD

- CI/CD = Continuous Integration and Continuous Delivery/Deployment
  - Continuous integration: sw development practice that merges work of all developers working on the 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

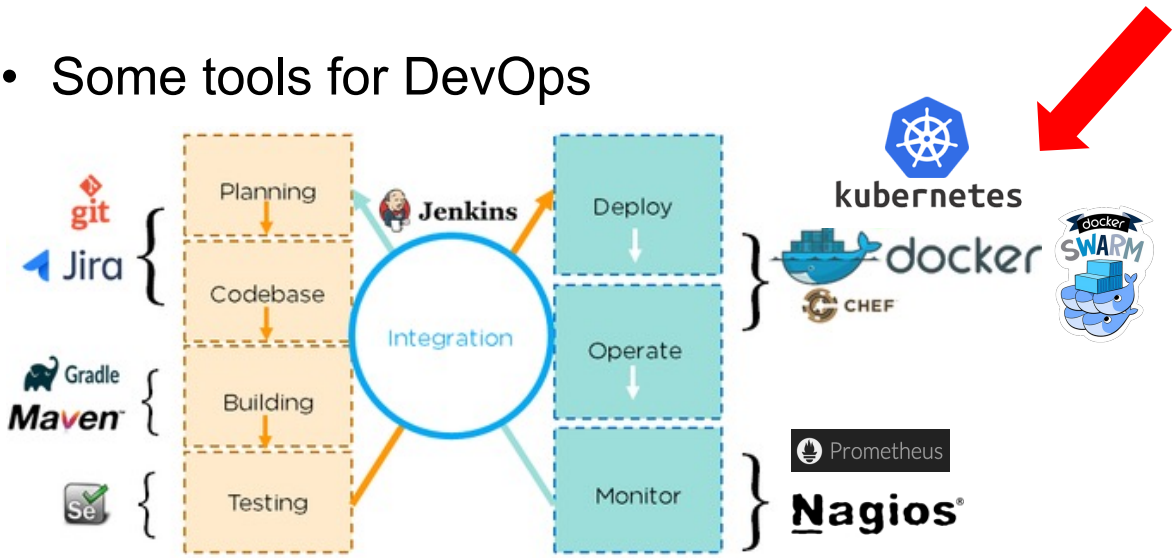
Containers and DevOps

- Containers are now 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 and DevOps

- Some tools for DevOps



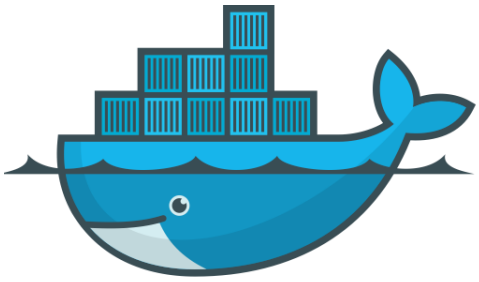
Containers, microservices, and serverless

- 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

http://www.ce.uniroma2.it/courses/sdcc2223/slides/Docker.pdf



Container resizing

- As for VMs, we can resize and migrate containers

- Resizing (CPU, memory, I/O) changes container limits
- Dynamic (i.e., on running container without stopping it)? Depends on container engine and underlying OS
- Resizing with Docker

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

  – Some example

  `$ docker update --cpu-shares 512 containerID`

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

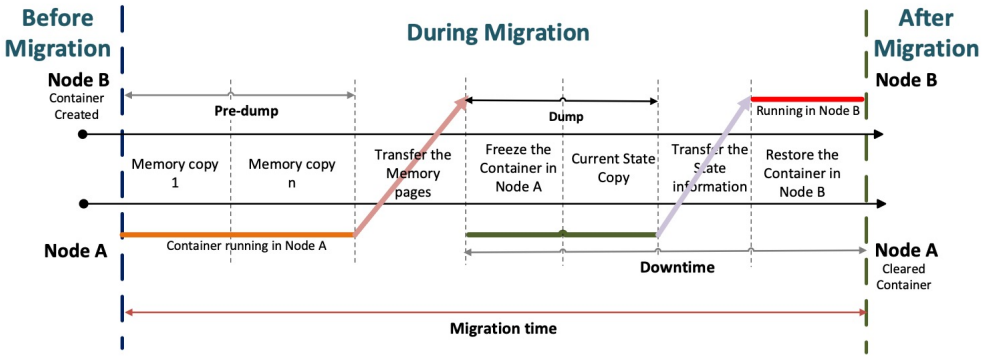
  – Low-level solution: `cgroups` limits can be changed on the fly

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 apps: 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

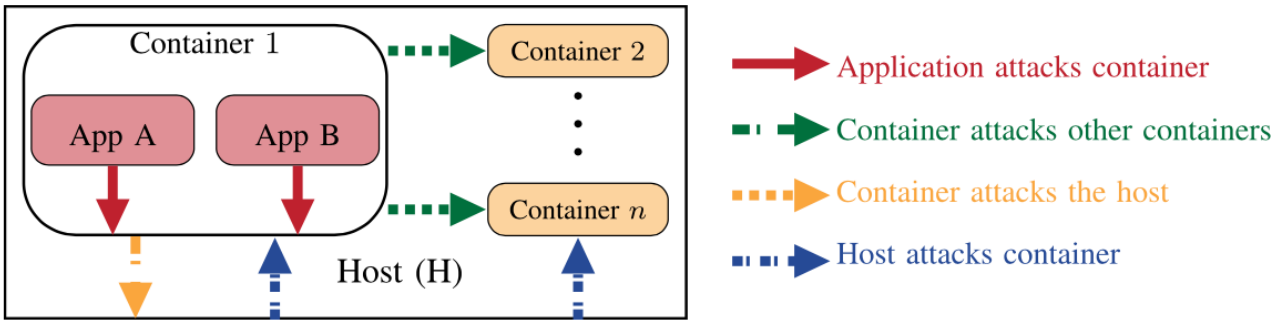
Live migration of containers

- Use CRIU tool to support live migration (in Docker and other container engines) through checkpointing and restoration technique
  - 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 (experimental)



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

Containers in the Cloud

- Containers and container development platforms as first-class Cloud services
- **Container-as-a-Service** (CaaS)
  - Amazon Elastic Container Service (ECS)
    - Two launch modes: EC2 and Fargate (run containers without having to provision or manage EC2 instances)
  - Azure Container
  - Google Cloud Run

Container orchestration

- 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)
  - Amazon Elastic Container Service
  - Google Kubernetes Engine
  - Marathon

Fully managed Cloud services

Hypervisors and containers in Cloud

- Which virtualization technology for IaaS providers?
  - Pros of hypervisor-based virtualization: greater security, isolation, and flexibility (different OSs on same PM)
  - Container-based virtualization pros: smaller-size deployment and thus larger density, reduced startup and shutdown times
- Some question
  - Containers inside VMs or on top of bare metal?
  - Will containers replace VMs in Cloud offering?

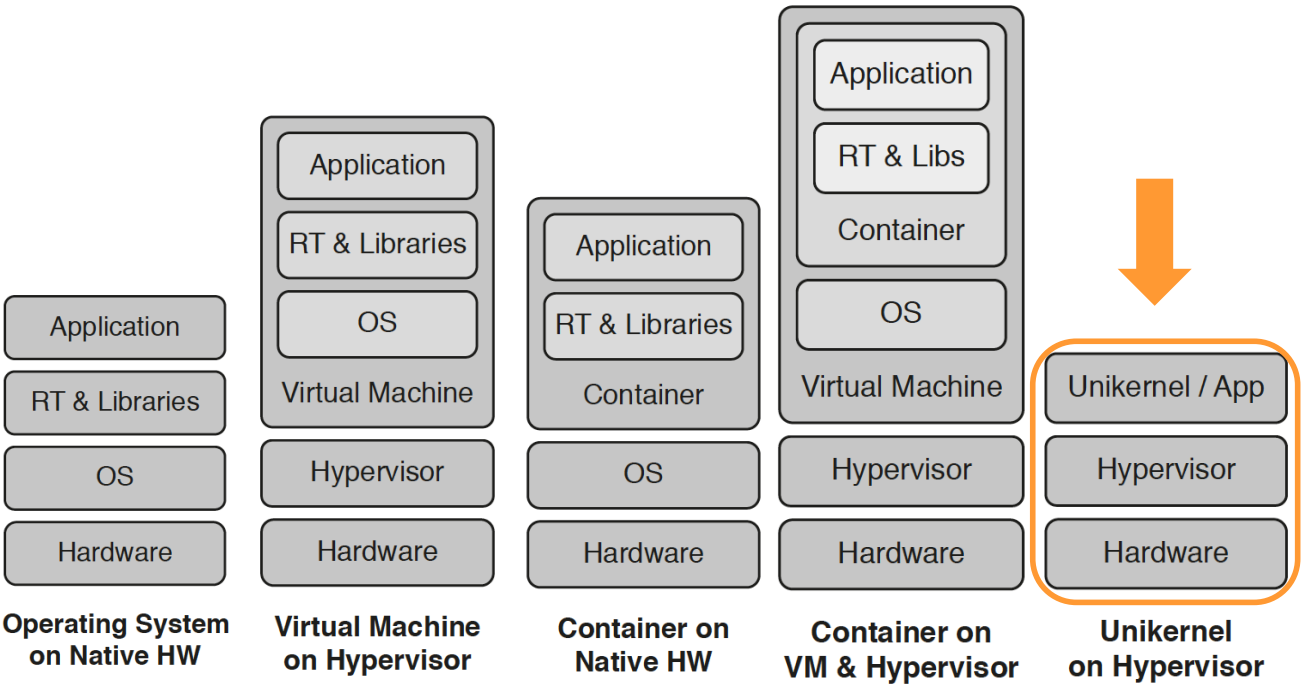
Hypervisors and containers in Cloud

- Recent virtualization trend: combine security and isolation provided by hypervisors with speed and flexibility of containers
- Firecracker: open source, tiny VMM by Amazon for creating and managing secure and efficient containers and serverless functions
  - Based on KVM but with minimalist design (excludes all non-essential functionality: no BIOS, no PCI, etc.)
  - Runs app in microVM: < 125 ms startup time and < 5 MB memory footprint
  - Written in Rust

Agache et al., Firecracker: Lightweight Virtualization for Serverless Applications, NSDI 2020

New lightweight virtualization approaches

- Deployment strategies examined so far



New lightweight approaches to virtualization

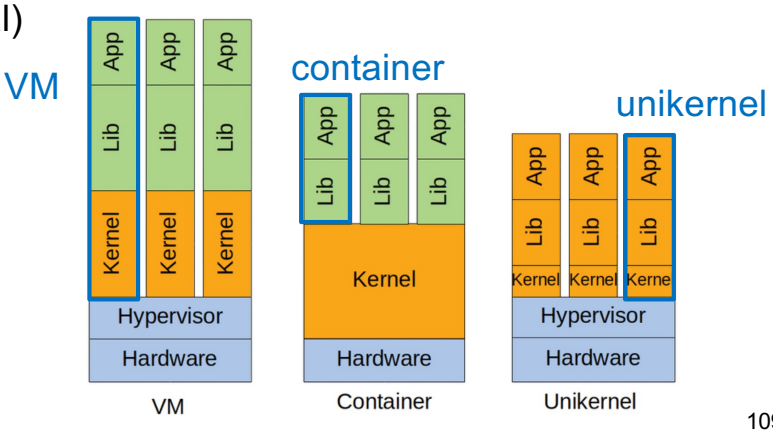
- Microservices, serverless computing, IoT and edge/fog computing demand for low-overhead (or lightweight) virtualization techniques
  - OS-level virtualization is not enough
  - How to have tiny, one-shot virtualized environments that run with great density and self-scale their resource needs?
  - How to improve security?
- **Lightweight OSs** and **unikernels**
  - Idea: avoid OS overhead and reduce attack surface
  - OS overhead: services and tools coming with common OSs (shells, editors, core utils, and package managers) are not required
  - Attack surface: images contain only the code that is strictly necessary for app to run, resulting in minimal attack surface

Lightweight operating systems

- Minimal, container-focused OSs, typically with a monolithic kernel architecture
  - 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
  - Designed for large-scale deployments, mostly targeting enterprises, with focus on automation, ease of application deployment, security, and scalability
  - Also installed on bare metal

Unikernels

- **Unikernel**
  - Specialized, single-address-space machine image constructed by using 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 (or bare metal)



Unikernels: pros and cons

- Pros:
  - Lightweight and small (minimal memory footprint)
  - Fast (no context switching)
  - Secure (reduced attack surface)
  - Fast boot (measured in ms)
- Cons:
  - Significant engineering effort in order to port apps to unikernel
  - Limited debugging tools
  - Single language runtime

    See https://www.youtube.com/watch?v=oHcHTFleNtg

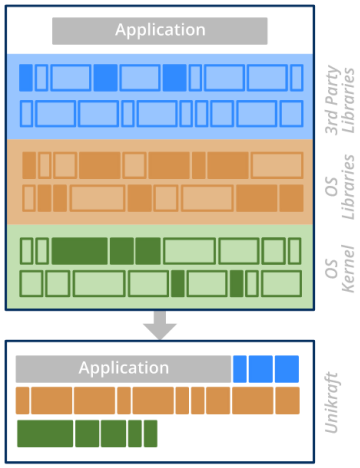
- Good news: cons almost solved with recent products

Unikernels: products

- Some unikernel products (and supported programming language):
  - MirageOS (OCaml)
  - OSv (C++, Go, Python and Java, …)
  - Unikraft
- OSv
  - Unikernel designed to run single unmodified Linux application securely as microVM on top of hypervisor (e.g., KVM, Xen, Firecracker)
  - Goal: isolation benefits of hypervisors without overhead of guest OS
  - To run app on OSv, you need to build an image by fusing OSv kernel and app files together

Unikernels: products

- Early unikernel frameworks required to write apps from scratch
- 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

Performance of virtualization approaches

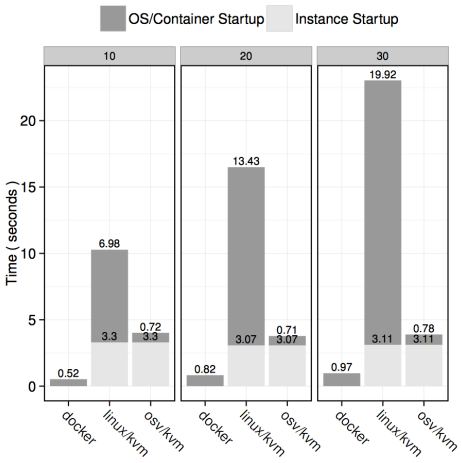
- Performance studies compare hypervisor vs. lightweight virtualization
- Overall result: overhead introduced by containers is almost negligible
  - Fast instantiation time
  - Small per-instance memory footprint
  - High density
- … but paid in terms of security

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	<50 msec	<50 MB	~8 MB	Yes	No

Source: Consolidate IoT edge computing with lightweight virtualization

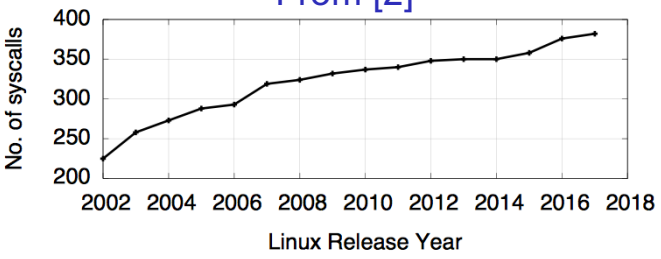
Performance of virtualization approaches

From [1]

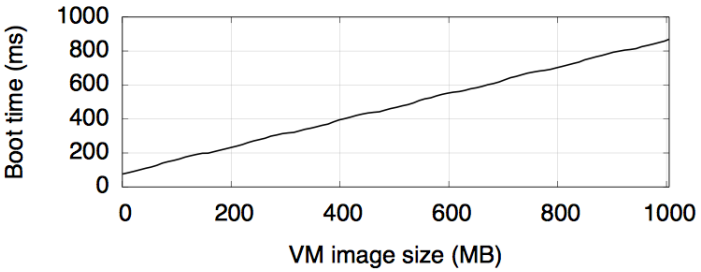


- Startup time for 10, 20 and 30 instances (includes overhead of overall provisioning time caused by OpenStack)

From [2]



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



- VM boot times grow linearly with VM size

[1] Time provisioning evaluation of KVM, Docker and Unikernels in a cloud platform, CCGrid'16

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