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

System-Level Virtualization

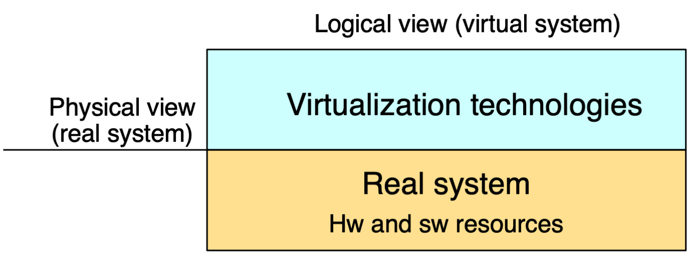
Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Virtualization

- Provides a high-level abstraction that hided the details of the underlying implementation
  - Hardware and software resources
- Offers a logical view of computing resources that differs from the physical ones



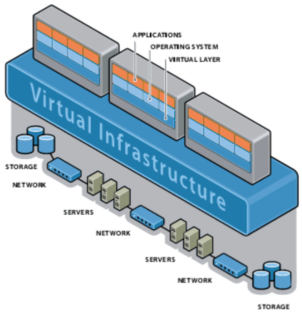
- How? Decouples user-perceived architecture and behavior of hw and sw resources from their physical implementation
- Goals:
  - Portability, efficiency, reliability, security, …

Virtualization of resources

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

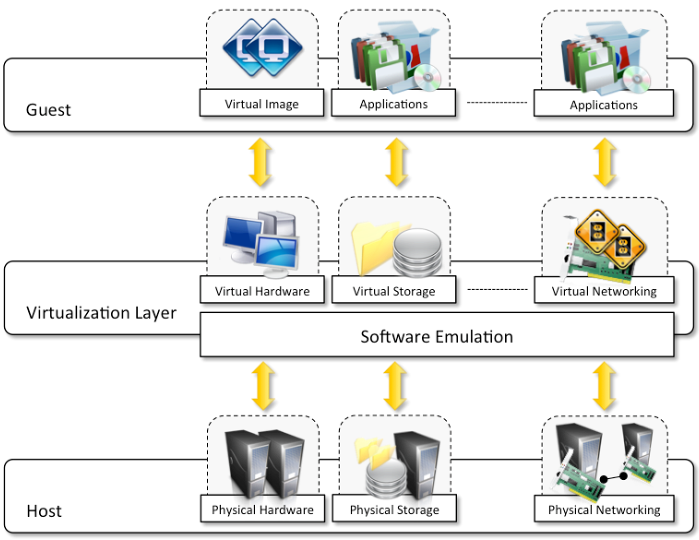
Our focus

- Storage virtualization
  - Abstracts storage resources
  - Storage Area Network (SAN), virtual disks, …
- Network virtualization
  - Virtual LAN (VLAN), Virtual Private Network (VPN), …
- Data center virtualization
  - Software-Defined Data Center (SDDC)

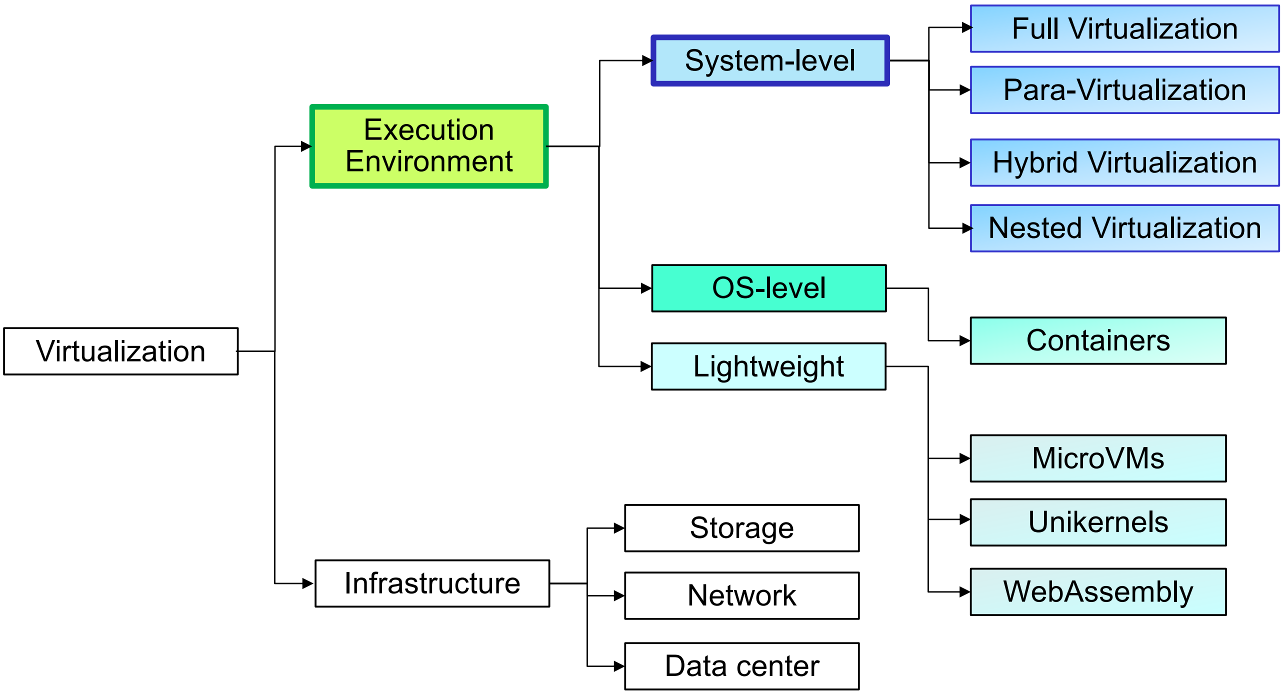


Major components of virtualization

- **Guest**:
  - The virtualized environment
  - Interacts with the virtualization layer, not directly with host
- **Host**:
  - The physical system
  - Provides the original hw and sw resources
- **Virtualization layer**
  - Responsible for creating the virtual environment
  - Manages resource allocation, isolation, and interaction between guest and host



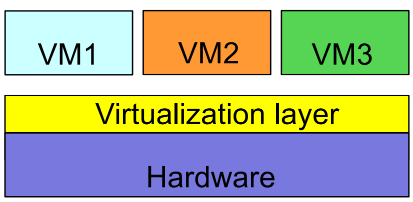
Taxonomy of virtualization techniques



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

Virtual Machine

- **Virtual Machine** (**VM**): a complete compute environment with its own isolated processing capabilities, memory, and communication channels
- Represents hw/sw resources of a physical machine differently from their physical reality
  - E.g., VM hw resources (CPU, network card, ...) can differ from physical machine's hw resources
  - E.g., VM sw resources (OS, applications) can differ from physical machine's sw resources
- A single physical machine can host multiple independent VMs



Virtualization: a brief history

- An "old" idea in computer science
  - Originates in the 1960s in centralized computing
  - Goals: let legacy software run on expensive mainframes and transparently share scarce physical resources
  - Example: IBM System/360-67 mainframe

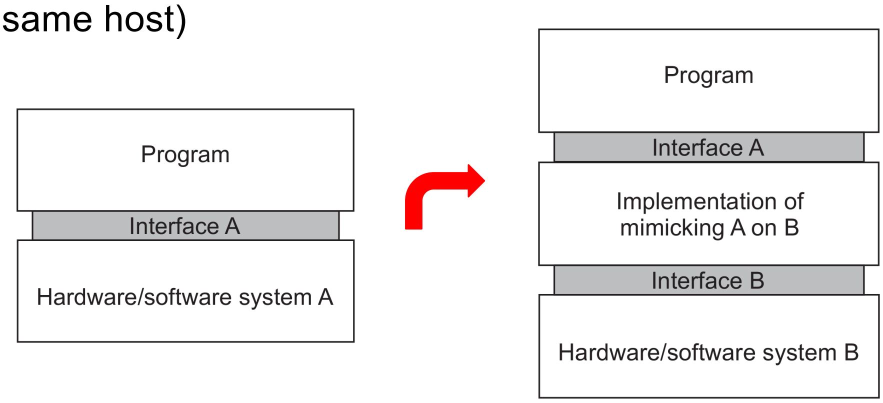
- Shift in the 1980s with the rise of PCs
  - OS multitasking to transparently share computing resources
  - Virtualization became less relevant for a while

Virtualization: a brief history

- Late 1990s: virtualization interest returns
  - Needed to simplify programming on special-purpose hardware
  - VMware founded in 1998
- Why the comeback?
  - Hardware evolves faster than software → higher management costs
  - Poor application portability
  - Significant under-utilization of hardware resources
  - Need to share resources efficiently and reduce infrastructure costs
- Today, virtualization is a core technology in cloud computing

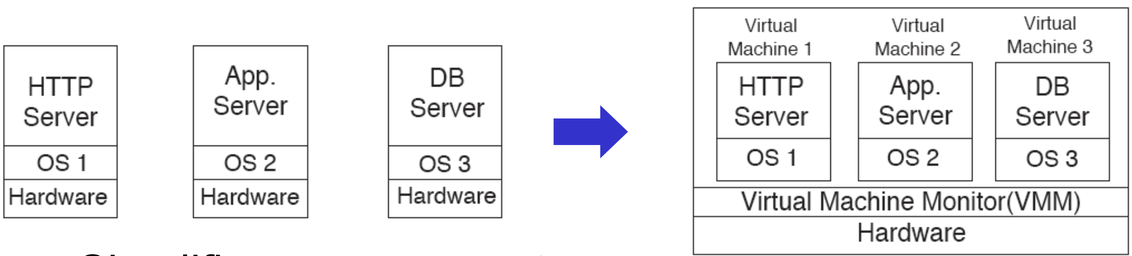
Virtualization: benefits

- Improves portability, compatibility, interoperability, and migration
  - Hardware independence: create once, run everywhere
  - Legacy support: run old OSes/apps on modern hardware
  - Support heterogeneous environments (different OSes on same host)



Virtualization: benefits

- Enables server consolidation and efficiency
  - VMs share one physical machine (multiplexing)
  - Reduces number of servers → lower cost, energy, and space usage
  - Better utilization of otherwise idle hardware



- Simplifies management
  - Easier maintenance and upgrades
  - Rapid provisioning: quickly create, clone, or destroy VMs
  - Snapshots and rollback simplify testing and recovery
  - Per-VM monitoring and resource accounting (important for cloud billing)

Virtualization: benefits

- Improves reliability and security
  - Strong isolation: faults or attacks in one VM do not affect others
  - Malware, bugs, and crashes remain contained within the VM
  - Micro-segmentation and secure virtual networks
  - Disaster recovery: VM images can be replicated and restored elsewhere

- Provides performance isolation
  - Controlled scheduling of shared resources ensures predictable performance
  - Reduces noisy-neighbor effects

Virtualization: benefits

- Improves load balancing and high availability
  - Improves load distribution across physical hosts
  - Live migration enables moving VMs seamlessly across physical hosts and reduces downtime
  - Enhances availability in cloud and data-center environments

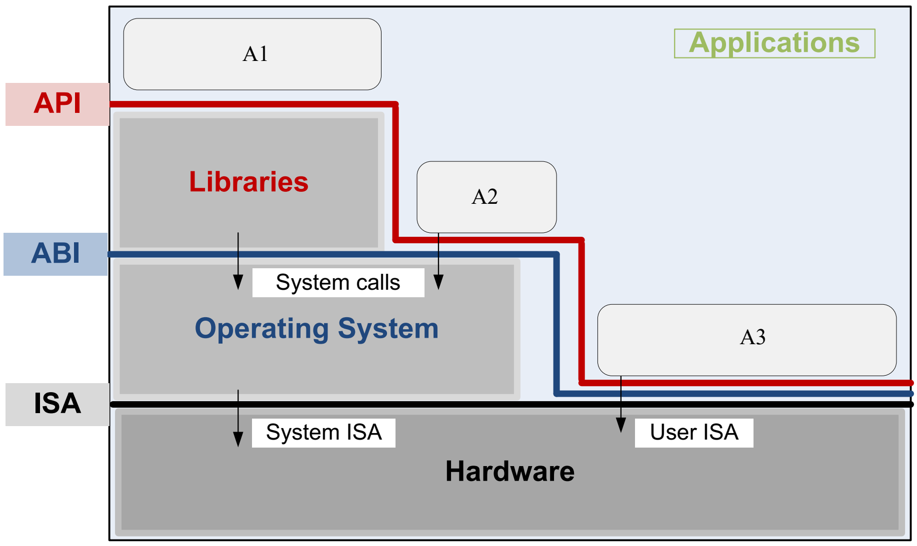
- Better development and testing
  - Safe sandbox environments
  - Easy creation of testbeds and multi-VM setups
  - Useful for CI/CD pipelines and experimentation

Reasons to use virtualization

- Personal and educational
  - Learn and experiment with multiple OSes
  - Develop, test, and debug applications in isolated environments
  - Simulate distributed systems on a single machine

- Enterprise and professional
  - Support DevOps workflows and continuous integration
  - Encapsulate entire systems in VM images for replication, migration, or redeployment
  - Ensure business continuity and disaster recovery readiness
  - Run legacy applications or OSes on modern platforms

Interfaces in computer system



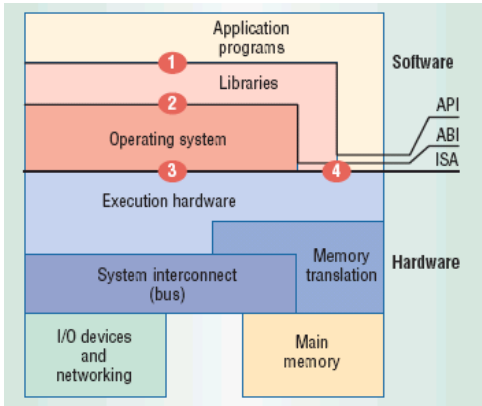
Applications:

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

Interfaces in computer system and virtualization

Levels where virtualization can be realized

- Key idea: virtualization is strictly related to computer system interfaces → mimics their behavior
  - Hw/sw interface (**system ISA**): manages system resources, privileged instructions executed only by OS [interface 3]
  - Hw/sw interface (**user-level ISA**): focused on computation, *unprivileged* instructions executed by any program [interface 4]
  - System calls: controlled access to OS services from user applications [interface 2]
  - **ABI** (Application Binary Interface): ensures binary compatibility [interfaces 2 + 4]
  - Library calls (**API**): high-level interface for application software to use system services [interface 1]



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

Implementation levels of virtualization

- Virtualization can be implemented at various operational levels
  - Primary involved interface is indicated within [ ]
- ISA level [interface 4]
- **Hardware level** (aka *system VMs*) [interface 3]
- **Operating system level** (aka *containers*) [interface 2]

Our main focus

- Library level [interface 1]
- User application level (aka *process VMs*) [interfaces 4 + 1]

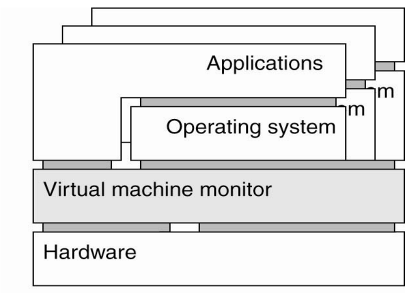
Implementation levels of virtualization

- ISA level
  - Goal: emulate a given ISA (unprivileged instructions) on the host machine for portability
  - Methods
    - Code interpretation: interpret each source instruction to host ISA instructions
    - Dynamic binary translation: convert code in blocks for faster execution
  - Example: run MIPS binary code on x86 host

Implementation levels of virtualization

- Hardware level (aka *system VMs*)
  - Goal: virtualize host resources (CPUs, memory, I/O) via **Virtual Machine Monitor** (**VMM**), aka *hypervisor*
    - VMM manages hw resources and shares them among multiple VMs
    - VMM intercepts privileged instructions and enforces isolation
  - Provides complete environment where multiple VMs coexist
  - Examples: VMware, KVM, Xen, Parallels, VirtualBox
  - Examples: microVMs
    - Lightweight VMs with minimal OS (e.g., Firecracker, Kata Containers)

Multiple instances of combinations <applications, OS>



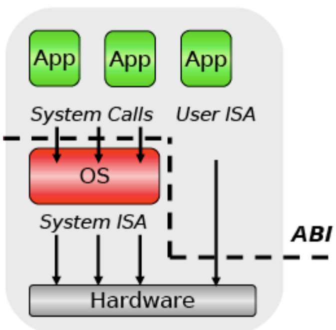
Implementation levels of virtualization

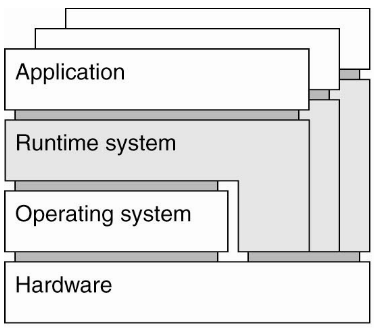
- Operating system level (aka *containers*)
  - Goal: create multiple isolated containers sharing the OS kernel
  - Containers rely on OS kernel to provide system calls
  - Examples: Docker, LXC, Podman
- Library level
  - Goal: run apps on host OS that does not natively support them
    - Translate high-level API calls from the application and translate them to host OS API rather than emulating full OS
  - Examples:
    - Wine: Windows apps on top of Linux https://www.winehq.org
    - Cygwin: “Get that Linux feeling – on Windows” https://cygwin.com

Implementation levels of virtualization

- User application level (aka *process VMs*)
  - Virtual platform for a single process
  - Provides virtual ABI/API to user app
  - Executes portable intermediate code (e.g., Java bytecode)
  - Examples: JVM, .NET CLR, WebAssembly runtimes

Multiple instances of combinations <application, runtime system>





System-level virtualization: terminology

- Let's focus on **system-level virtualization** via VMM/hypervisor

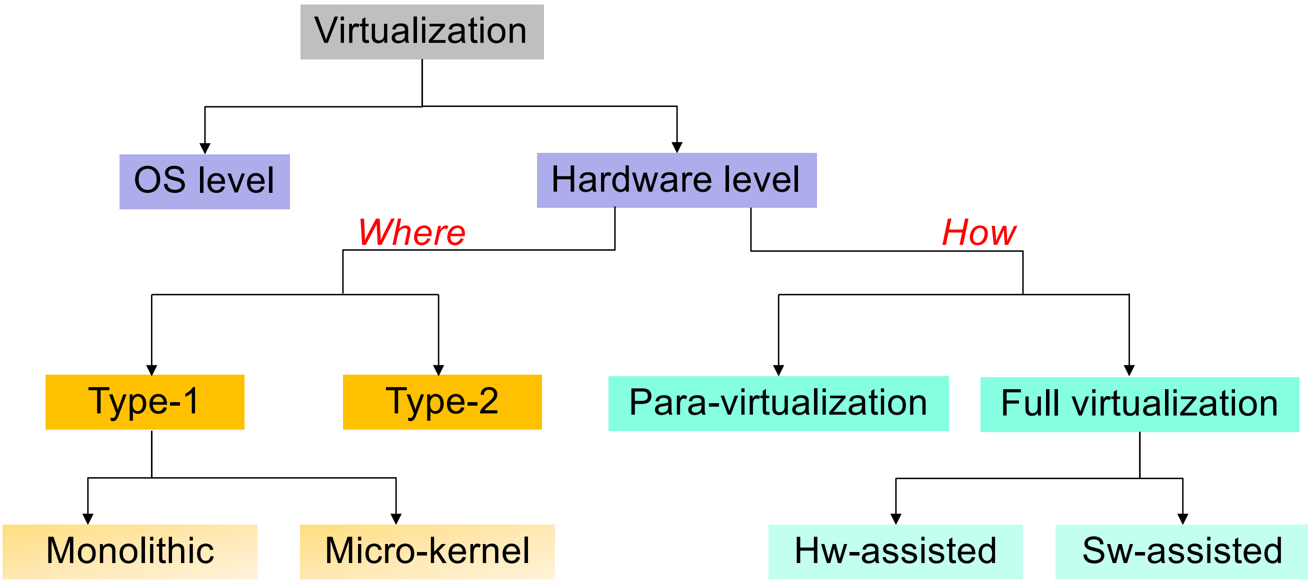
- **Host**: base platform on top of which VMs run; consisting of:
  - Physical machine
  - *Optional* host OS
  - VMM/hypervisor
- **Guest**: everything inside the VM
  - Guest OS
  - Applications executed inside the VM

System-level virtualization: taxonomy

- System-level virtualization classification
  1. Where VMM is deployed
     - **System VMM**
     - **Hosted VMM**

  2. How to virtualize instruction execution
     - **Full virtualization**
       - *Software-assisted*
       - *Hardware-assisted*
     - **Paravirtualization**

System-level virtualization: taxonomy

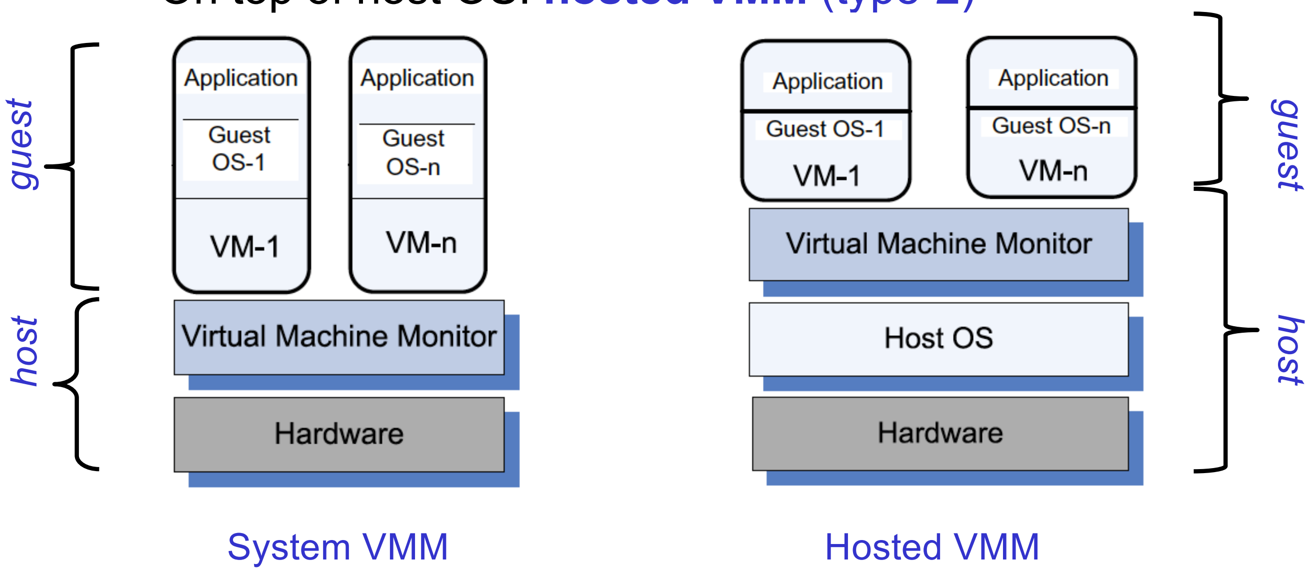


Comparison of platform virtualization software
https://en.wikipedia.org/wiki/Comparison_of_platform_virtualization_software

System vs hosted VMM

VMM deployment level in system architecture

- Directly on hardware: **system VMM** (type-1 / native / bare-metal)
- On top of host OS: **hosted VMM** (type-2)



System vs hosted VMM

- **System VMM** (*type-1*): runs directly on hw and provides virtualization features integrated into a minimal OS
  - Can use either *microkernel* (only basic functions, no device drivers) or *monolithic* (integrated drivers) architecture design
  - Examples: Xen, VMware ESXi, Xen, Microsoft Hyper-V, KVM, Nutanix AHV
- **Hosted VMM** (*type-2*): runs on top of host OS and uses host OS drivers and system calls
  - Interacts with the host OS via the ABI and presents virtualized hw (emulated or hardware-assisted) to the guest OSes
  - ✓ Leverages host OS for device management and low-level services (e.g., resource scheduling)
  - ✓ No guest OS modifications required
  - ✗ Lower performance with respect to system VMM due to extra sw layer
  - Examples: VirtualBox, Parallels Desktop

Privileged instructions

- Instructions that can only be executed by OS kernel (or hypervisor)
- They access or control critical system resources such as:
  - CPU modes (switching between user and kernel)
  - Memory management (page tables, MMU)
  - I/O devices (disk, network, peripherals)
  - Interrupt handling
- Purpose: prevent user programs from directly manipulating hw or compromise system stability
- Examples:
  - Changing the CPU mode (user → kernel)
  - Modifying control registers (x86 `CR3` for page tables)
  - Direct access to I/O ports

Full virtualization vs paravirtualization

How to manage execution of privileged instructions in VMs
- **Full virtualization**
- **Paravirtualization**

- Full virtualization
  - VMM exposes to each VM simulated hw interfaces *identical* to the underlying physical machine
  - VMM intercepts attempts to perform privileged instructions (e.g., I/O, TLB update) and emulates their behavior
  - Examples: KVM, VMware ESXi, Microsoft Hyper-V
- Paravirtualization
  - VMM exposes to each VM simulated hw interfaces that are *similar* but not identical to the underlying physical machine
  - Hardware is not fully emulated; a minimal software layer (Virtual Hardware API) manages VMs and ensures isolation
  - Examples: Xen, Oracle VM, PikeOS

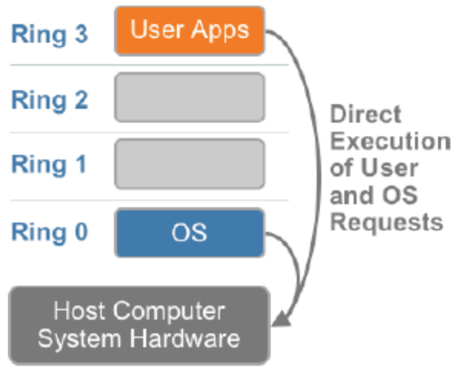
Full virtualization vs paravirtualization

- Full virtualization: pros
  - ✓ Runs unmodified guest OSs
  - ✓ Strong isolation between VMs: improves security and allows emulating different architectures independently
- Full virtualization: cons
  - ✗ VMM is more complex
    - It must intercept and emulates all privileged instructions
  - ✗ Requires processor support to make virtualization more efficient: why?

Issues to address for system-level virtualization

- Non-virtualized processor architectures use at least 2 protection rings: supervisor and user
  – Ring 0: most privileged (unrestricted access to system resources)
  – Ring 3: least privileged

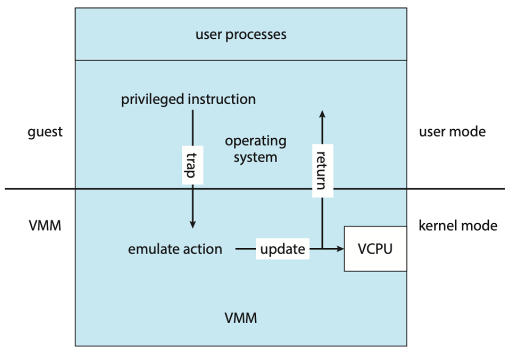
x86 architecture w/o virtualization



- With virtualization
  – VMM operates in supervisor mode (ring 0)
  – Guest OS and applications run in less privileged rings (guest OS in ring 1 or 3)
  – **Ring deprivileging** problem: guest OS executes in a ring that is not its own → cannot execute privileged instructions directly
  – **Ring compression** problem: since both guest OS and applications run at the same level, guest OS must enforce protection

Addressing ring deprivileging

- **Trap-and-emulate**
  – When the guest OS executes a privileged instruction (e.g., LIDT), an exception (trap) is raised and control is transferred to the VMM; the VMM performs a safety check on the requested operation, executes ("emulates") its behavior, and returns the result to guest OS
  – Non-privileged instructions, instead, run by guest OS do not trap and are executed directly



- Issue: on some architectures, certain instructions do not trap, requiring more complex solutions

Popek and Goldberg virtualization requirements

- Popek and Goldberg requirements (1974) define when an architecture can support *efficient* virtualization
- Conditions:

Equivalence

1. A program running under the hypervisor should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.

Resource control

2. The hypervisor should be in complete control of the virtualized resources.

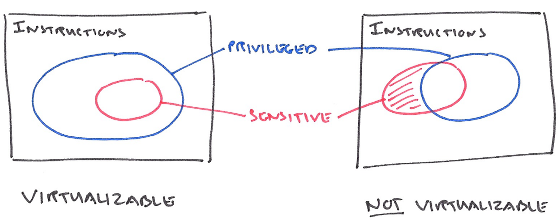
Efficiency

3. A statistically significant fraction of machine instructions must be executed without the intervention of the hypervisor.

Formal requirements for virtualizable third generation architectures, 1974
https://dl.acm.org/doi/pdf/10.1145/361011.361073

Instruction types and virtualization theorem

- ISA instruction classification
  - Privileged instructions: execute only in supervisor mode; trap in user mode
  - Sensitive instructions: affect or reveal CPU/system state
    - Control-sensitive: modify CPU configuration/state (e.g., interrupt/paging tables)
    - Behavior-sensitive: reveal CPU state
  - Innocuous instructions: neither sensitive nor privileged
- **Popek and Goldberg theorem**: *a VMM can be constructed if all sensitive instructions are privileged*



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

Problem and implications

- Problem: some architectures have sensitive non-privileged instructions, e.g.:
  - x86: many sensitive non-privileged, such as:
    - `PUSHF`, which modified the flags register
    - access or modification of `CR3` register, used for memory paging and virtual memory management
  - MIPS: mostly virtualizable, but `$k0`, `$k1` registers are accessible in user mode
  - ARM: mostly virtualizable, but some instructions undefined in user mode
- Must virtualize:
  - Privileged instructions → trap to VMM and emulate
  - Sensitive non-privileged instructions → must also be be virtualized

Solutions for virtualizing sensitive instructions

- Trap-and-emulate
  - Privileged instructions trap → VMM emulates instruction behavior
  - Non-privileged instructions run directly on CPU
  - Limitation: some sensitive non-privileged instructions do not trap on certain CPUs → which solutions?

1. **Hardware-assisted virtualization** (hw level)
   - VMM relies on hardware to intercept privileged and sensitive instructions automatically
   - All sensitive instructions trap automatically to VMM
2. **Fast binary translation** (sw level)
   - VMM rewrites sensitive instructions on the fly
3. **Paravirtualization**
   - Modify guest OS to avoid or neutralize sensitive non-privileged instructions

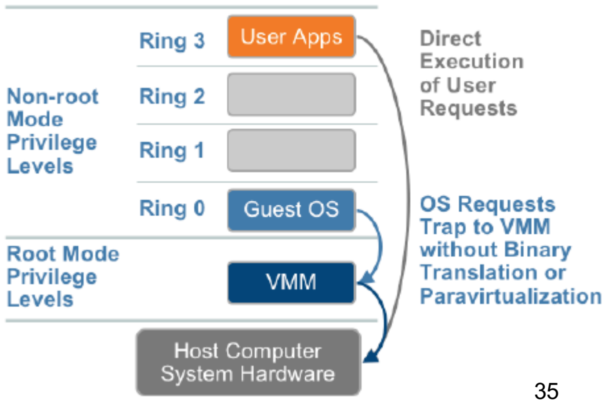
Hardware-assisted CPU virtualization

- Idea: introduce a new privilege level for the hypervisor
  - Hypervisor privilege > OS/kernel privilege
  - All sensitive instruction trap to hypervisor level
- Provides two new CPU operating modes
  - **Root mode** for the VMM
  - **Non-root mode** for guest OSs
- Each mode supports all 4 x86 protection rings

Hardware-assisted CPU virtualization

- VMM runs in Root-Ring 0, while guest OSs run in guest mode in Non-Root Ring 0 at their original privilege levels
  - Eliminates ring deprivileging and ring compression issues
- When a guest OS executes a privileged or sensitive instruction in Non-Root mode, the CPU automatically stops guest execution and switches control to the VMM, which runs in Root mode (VM-exit)
  - VMM can control guest execution through VM control data structures in memory
- Implemented in modern CPUs (e.g., Intel VT-x and AMD-V, since 2005)

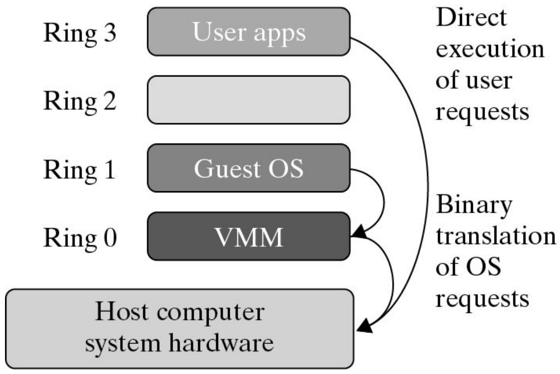
x86 architecture with with full virtualization and *hardware-assisted CPU virtualization*



Fast binary translation

- Software-based and the oldest solution
- VMM scans code before execution
- Replaces blocks containing privileged instructions with functionally equivalent blocks that notify the VMM when executed
  - Translated blocks run directly on hw and are cached for future reuse
- ✗ Higher complexity and lower performance compared to hw-assisted virtualization

x86 architecture with full virtualization and fast binary translation



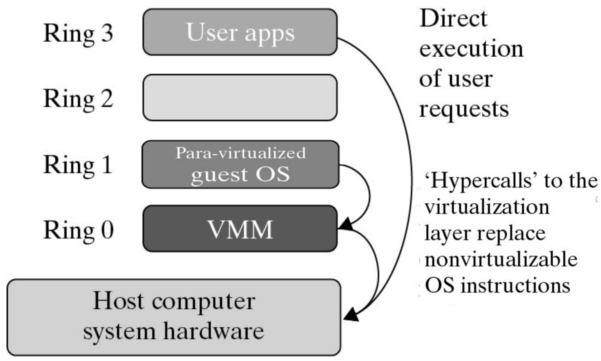
Paravirtualization

- Non-transparent virtualization
  - Guest OS kernel must be modified to use the virtual API exposed by the VMM
- Non-virtualizable instructions are replaced with *hypercalls* that communicate directly with the hypervisor
- Hypercall: *software trap* from the guest OS to the hypervisor

hypercall : hypervisor = syscall : kernel

- Hypercall execution
  - Control jumps from guest OS to hypervisor
  - Hypervisor performs the requested operation
  - After handling, control returns to guest OS kernel, which then resumes the application

x86 architecture with *paravirtualization*

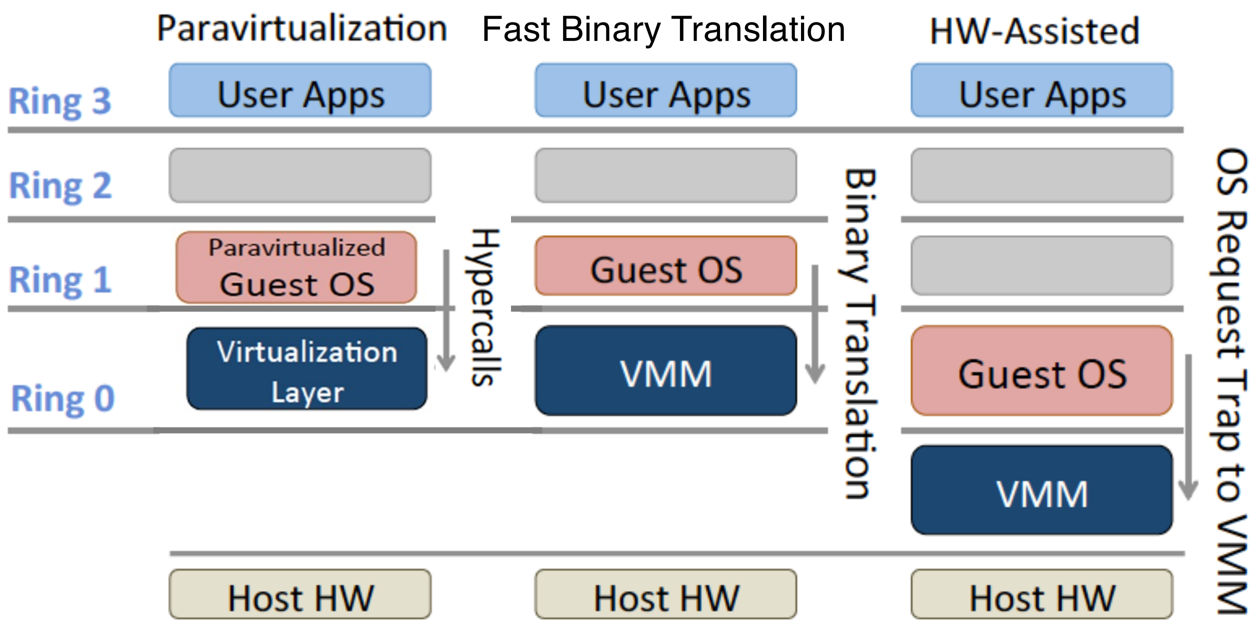


Paravirtualization

- Pros:
  - ✓ Easier and more practical to implement than full virtualization on non-virtualizable hardware
  - ✓ Lower runtime overhead compared to fast binary translation
  - ✓ Does not require virtualization extensions (unlike hw-assisted virtualization)

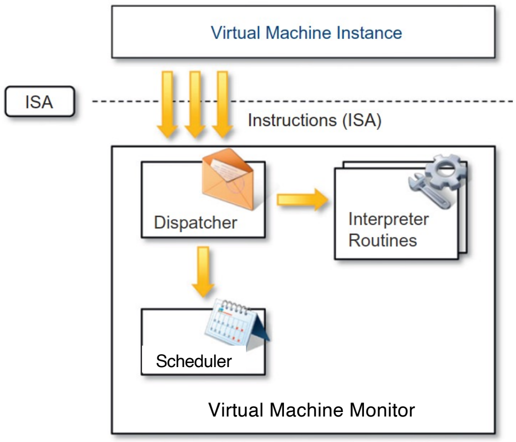
- Cons:
  - ✗ Requires source code access to modify the guest OS and make it paravirtualized
  - ✗ Maintenance overhead: each paravirtualized OS version must be maintained separately
    - Paravirtualized OS cannot run directly on hardware

Summing up different approaches



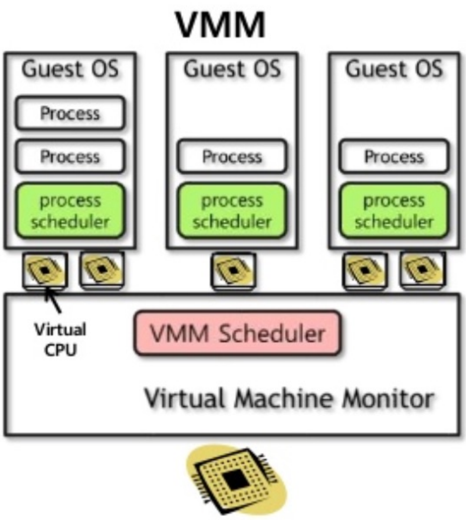
VMM reference architecture

- Main modules
  - *Dispatcher*: VMM entry point that reroutes privileged instructions issued by VMs to the appropriate module
  - *Scheduler*: invoked by the dispatcher when a VM requests system resources; decides resource allocation among multiple VMs
  - *Interpreter*: executes the appropriate routine for each privileged instruction



VMM CPU scheduler

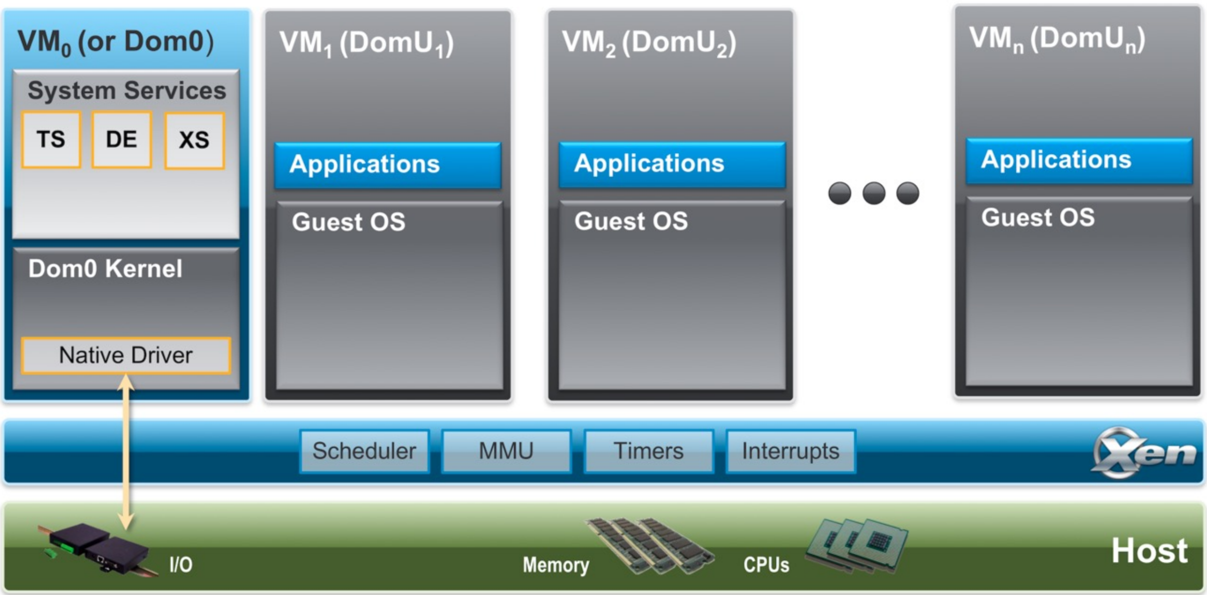
- Guest OS sees virtual CPUs
- Physical CPUs on host machine are multiplexed among VMs
- How to schedule virtual CPUs on physical CPUs?



Case study: Xen

- What Xen is https://www.xenproject.org
  - Open-source type-1 (system VMM) hypervisor with microkernel design
  - Originated at Cambridge University in the late 1990s
  - Foundation for XenServer, Oracle VM, and used in cloud and embedded systems (e.g., AWS, Alibaba, ARM)
  - Supports paravirtualization (PV), hardware-assisted virtualization (HVM) and PV-HVM hybrids
- Key ideas for paravirtualization (no hw support yet)
  - Guest OS interacts with the hypervisor via a hypercall API
  - Paravirtualization replaces:
    - Privileged instructions → hypercalls
    - Page table updates
    - I/O device access
    - Interrupt & timer operations
  - Requires PV-enabled OSs and drivers (e.g., Linux)

Xen: architecture



https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview

Xen: architecture

- Thin micro-hypervisor
  - 300K LoC on x86, 65K on ARM
- Dom0 (privileged control domain)
  - Owns device drivers and multiplexes I/O
  - Runs management services
    - XenStore (XS): shared key–value config store
    - Toolstack (TS): VM lifecycle management (create, shutdown, pause, migrate) and configuration
    - Device Emulation (DE)
  - Mandatory, started on boot
- DomU (unprivileged guest domains / VMs)
  - Run PV or HVM guest OSs
- Hypervisor manages: vCPU scheduling, memory, interrupts

Xen and paravirtualization

- In many modern deployments, Xen no longer uses paravirtualization to virtualize the CPU
  - HVM guests rely on HVM
  - PV guests are still used when hardware lacks virtualization extensions
- However, paravirtualization remains important for I/O
  - Rather than inefficiently emulate hardware I/O devices, Xen defines virtual-only devices and provides drives for them

https://wiki.xenproject.org/wiki/Understanding_the_Virtualization_Spectrum

- Summarizing Xen impact
  - First to popularize paravirtualization
  - Influenced modern cloud hypervisors (AWS, Google)
  - Clear example of a micro-hypervisor
    - Reduces attack surface, increases stability and security, and keeps the hypervisor small and lightweight
  - Still used in security-sensitive and embedded environments

Xen: Credit scheduler

- VMM scheduler selects which virtual CPUs (vCPUs) of all VMs run on the physical CPUs (pCPUs)
  - Adds a scheduling layer below the guest OS scheduler
- Default Xen scheduler: **Credit** scheduler
  https://wiki.xenproject.org/wiki/Credit_Scheduler
- Goals:
  - Proportional fair share: each domain receives a fair share of pCPU proportional to its weight
  - Work-conserving: no pCPU stays idle if runnable vCPUs exist
  - Low latency: fast switching among runnable vCPUs
- Key parameters:
  - Weight: relative share of CPU time (default = 256)
  - Cap: optional limit on maximum CPU share
    - cap = 0 → vCPU can use extra CPU time (work-conserving)
    - cap > 0 → upper bound (% of one pCPU)

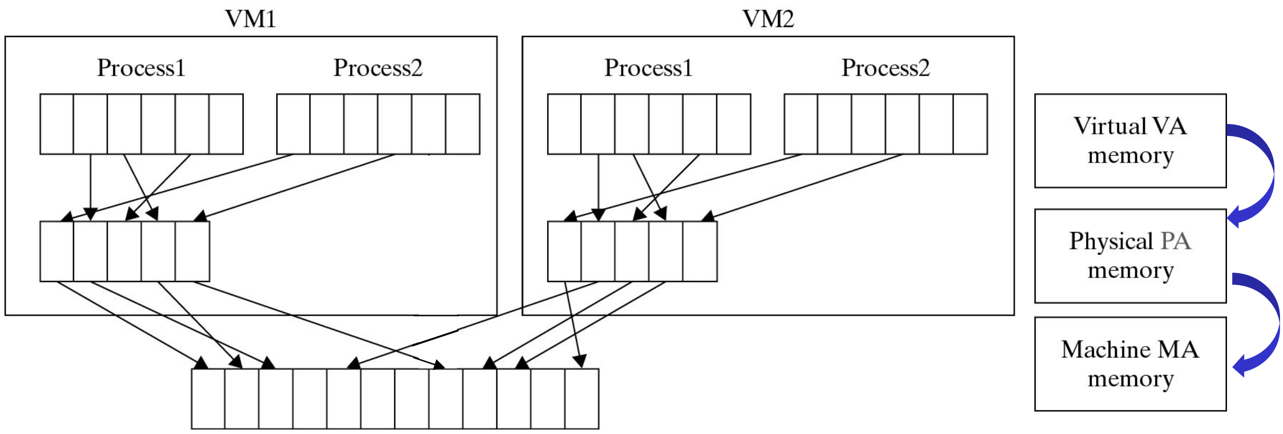
Xen: Credit scheduler

- Each vCPU receives credits proportional to its weight
- Running consumes credits → vCPU becomes:
  - UNDER (credits ≥ 0)
  - OVER (credits < 0)
- Each pCPU keeps a runqueue of vCPUs
  - UNDER vCPUs first, then OVER
  - OVER vCPUs run only if no UNDER exist
  - Round-robin within each group
  - Time slice: 30 ms
- Placement
  - vCPU is inserted into its home pCPU runqueue when runnable
  - May be migrated to another pCPU by load balancing
- Load balancing
  - Before going idle, a pCPU checks others for runnable UNDER vCPUs; ensures no pCPU is idle while work exists

Memory virtualization

- *Non-virtualized* environment
  - Single-level memory mapping: virtual address (VA) → physical address (PA) via page tables
  - MMU and TLB accelerate translations
- *Virtualized* environment
  - Multiple VMs share machine memory → VMM must partition it
  - Requires **two-level mapping:** guest virtual address (**GVA**) → guest physical address (**GPA**) → host physical address (**HPA**)
- Terminology
  - Guest virtual memory: visible to apps
  - Guest physical memory: visible to guest OS
  - Host physical memory: actual machine memory visible to VMM

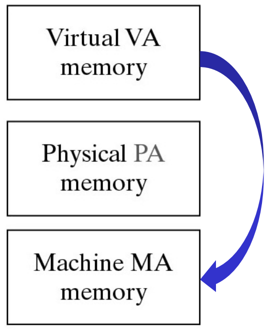
Two-level memory mapping



- **GVA → GPA → HPA**
- GPA ≠ HPA: why?
  - Because each guest OS assumes it owns *contiguous, zero-based* physical memory
  - VMM must preserve this illusion

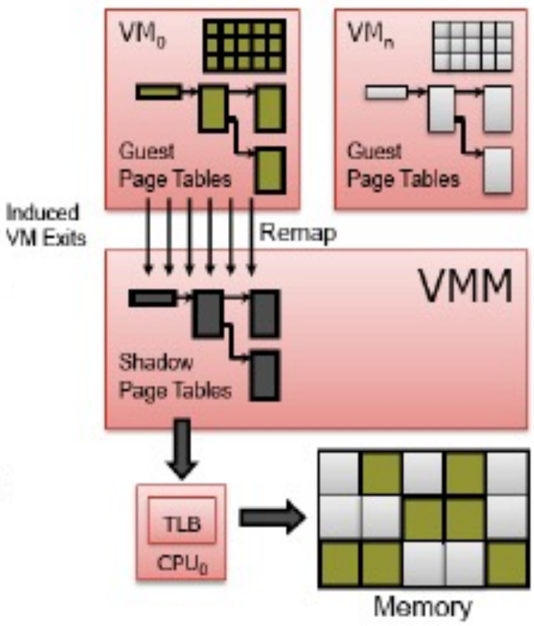
Shadow page tables (SPT)

- Goal: avoid performing two translations on every access by giving hardware a direct **GVA → HPA** mapping
- Key mechanisms:
  - VMM maintains shadow page tables (**SPTs**) **which** directly maps GVA to HPA and uses them to accelerate
  - Guest OS builds its own page tables normally, but MMU uses SPTs created by the VMM
  - VMM keeps SPTs consistent with guest page tables
  - Guest page tables are mapped read-only
  - Any guest update → trap to VMM, which updates both guest PT and SPT ("memory tracing")



SPT drawbacks

- High overhead: frequent traps and VM exits
  - Every guest PT write → VM exit → high overhead
  - A VM exit is similar to a very expensive context switch
- Large memory footprint for maintaining SPTs
- Complex software logic
  - VMM intercepts paging operations and constructs copy of PTs



Hw support for memory virtualization

- More efficient hardware solution: Second Level Address Translation (SLAT)
- Hardware performs the second translation (GPA → HPA)
- Removes need for SPT maintenance
- Dramatically reduces traps and VM exits
- Provides significant speedup (~50%) for memory-intensive workloads
- Examples: Intel EPT, AMD RV