Introduction to Virtualization

Virtualization and Cloud Computing - ay 2023/2024

Enrico RUSSO <enrico.russo@dibris.unige.it>
Giacomo LONGO <giacomo.longo@dibris.unige.it>

Virtualization

Virtualization

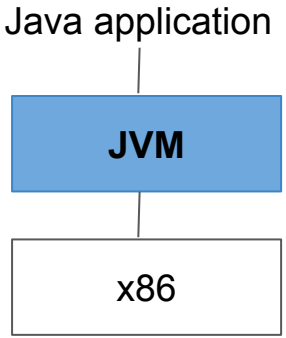
“Virtualization is the application of the **layering** principle

Virtualization (by examples)

Virtualization is not limited to any particular field of computer science.

For example Java. **JVM** is the abstraction:

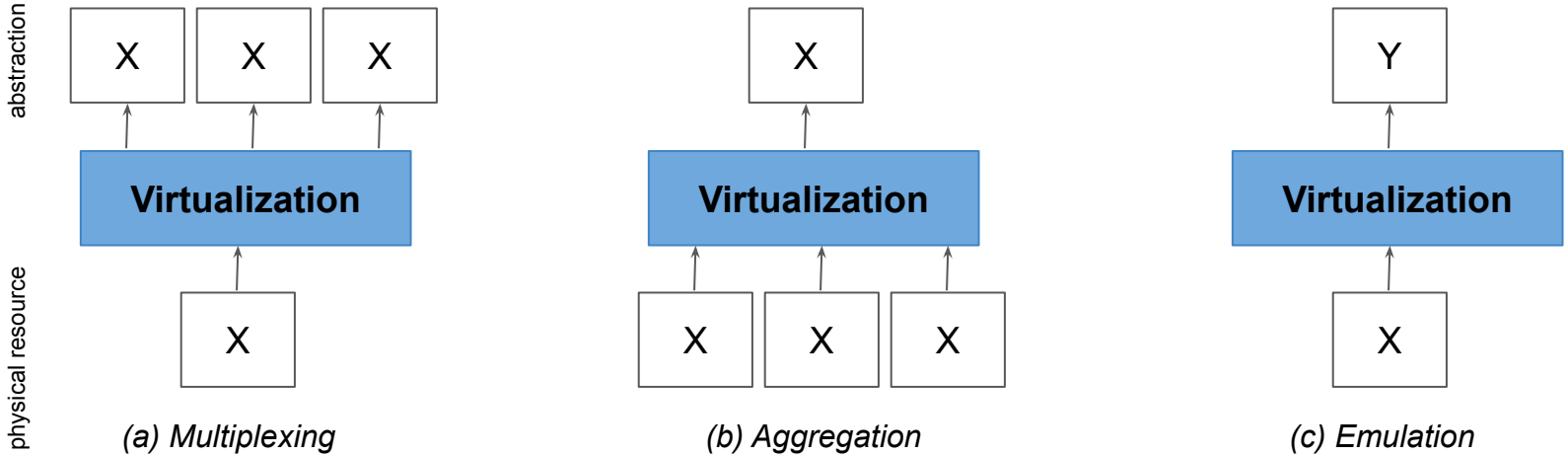
- relies on a single lower layer: the underlying computer (x86)
- exposes the bytecode-structured instruction set as the interface
- namespace: the virtual machine instance



Other examples: operating systems, Redundant Array of Independent Disks (RAID) systems, virtual memory, …

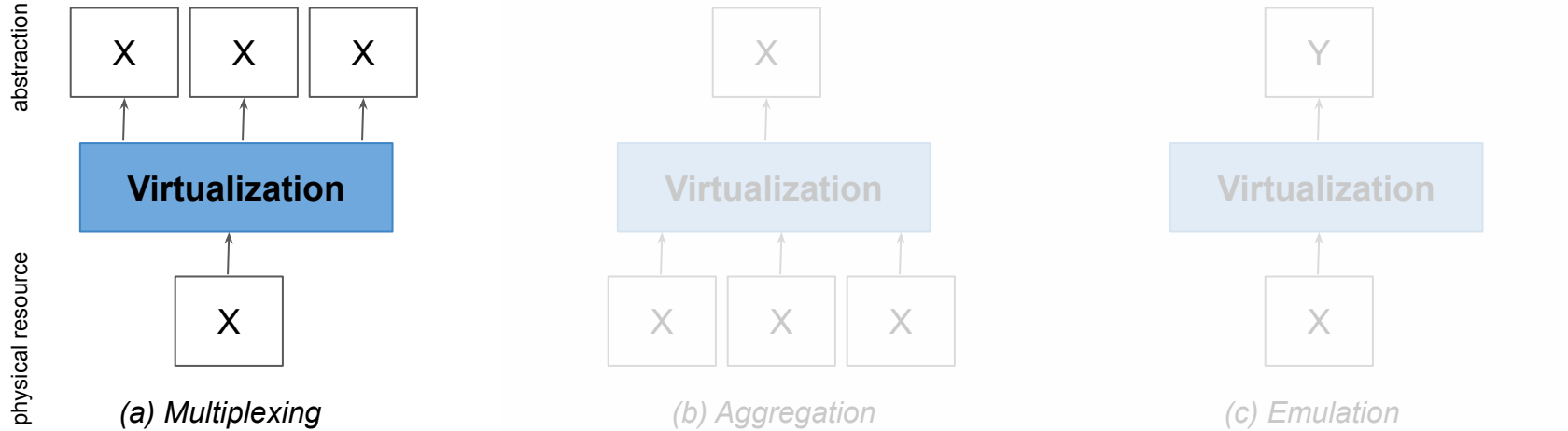
Virtualization techniques

Virtualization is always achieved by using and combining **three** simple techniques.



Virtualization techniques: multiplexing

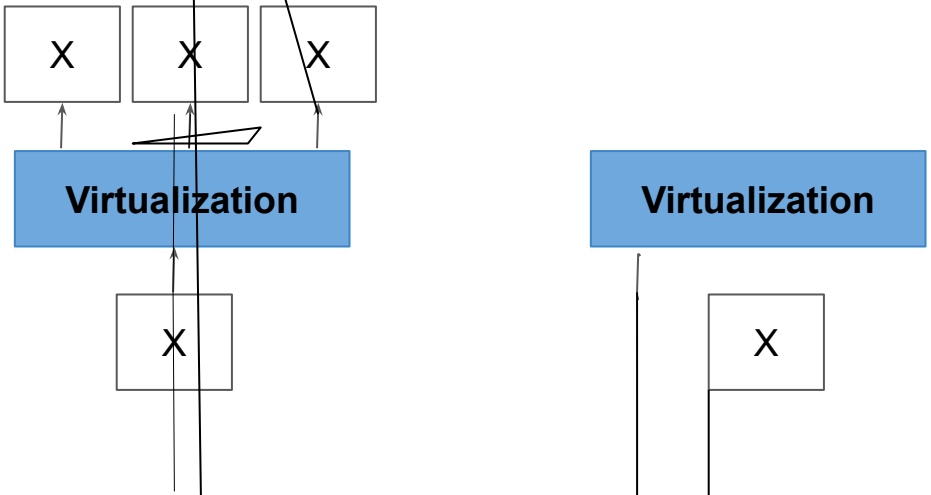
Virtualization is always achieved by using and combining **three** simple techniques.



exposes a resource among multiple virtual entities, e.g., the OS scheduler multiplexes the CPU core and hardware threads among the set of runnable processes.

Virtualization techniques: aggregation

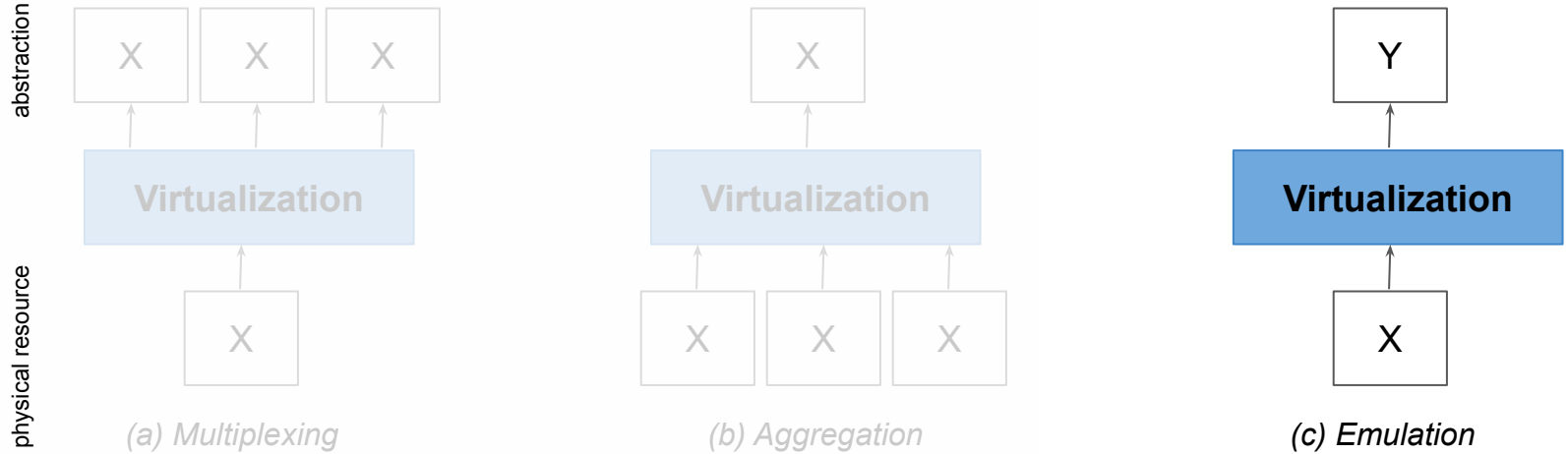
Virtualization is always achieved by using and combining **three** simple techniques.



takes multiple physical resources and makes them appear as a single abstraction, e.g., a RAID controller aggregates multiple disks into a single volume.

Virtualization techniques: emulation

Virtualization is always achieved by using and combining **three** simple techniques.



relies on **a level of indirection in software** to expose a virtual resource or device that corresponds to a **physical device**, even if it is **not present** in the current computer system. For example, cross-architectural emulators run one processor architecture on another (X=x86, Y=ARM).

Virtual Machines

Virtual machine

"A Virtual Machine (VM) is a complete compute environment with its own **isolated** processing capabilities, memory, and communication channels"

This definition applies to a range of distinct, incompatible abstractions.

Lightweight VM

relies on a combination of **hardware and software isolation** mechanisms to ensure that **applications running directly on the processor** (e.g., as native x86 code) are **securely isolated** from **other sandboxes** and the **underlying operating system**.

An example of Lightweight VMs are **containers**.

System Level VM

virtually provides the hardware of a computer and can run

- a standard Operating System (guest OS) and its applications
- in full isolation from the other virtual machines and the rest of the environment.

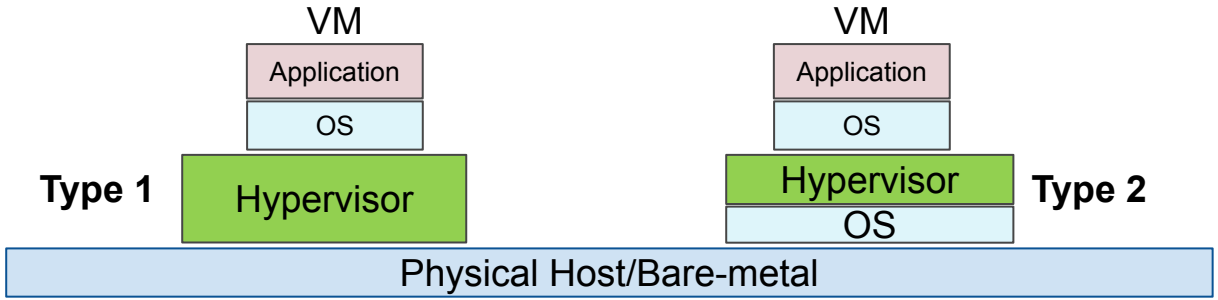
System Level VM: hypervisor

a special form of system software that runs virtual machines with the goal of **minimizing execution overheads**.

Hypervisors incorporate a combination of

- **multiplexing**: allocates and schedules the physical resources appropriately among the VMs
- **emulation**: provides a certain piece of hardware that it presents to the guest VM, regardless of what the actual physical hardware is (mainly ensures **compatibility** and **portability**)

Hypervisor types



- **Type 1** (or **bare metal**): acts like a **lightweight operating system** and runs directly on the host's hardware
- **Type 2** (or **hosted**) runs as a **software layer** on an operating system, like other computer programs.

Popek and Goldberg

Popek and Goldberg

Popek and Goldberg formalized the relationship between a VM and hypervisor (which they call VMM) in 1974[1].

"A virtual machine is taken to be an **efficient**, **isolated duplicate** of the real machine.
We explain these notions through the idea of a virtual machine monitor (VMM). As a piece of software, a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially **identical with the original machine**; second, programs running in this environment
show at worst only **minor decreases in speed**; and last, the VMM is in **complete control of system resources**"

[1]Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. Communications of the ACM, 17(7):412–421, 1974. DOI: 10.1145/361011.361073

Popek and Goldberg: VMM

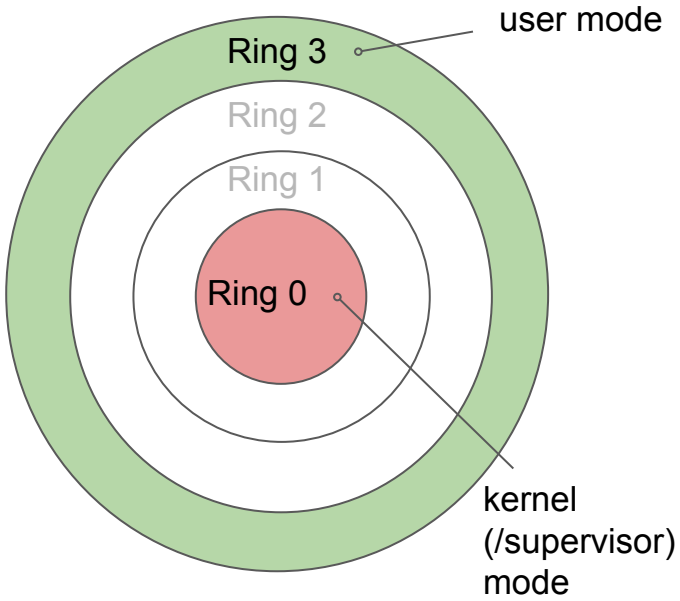
Popek and Goldberg's definition is **consistent** with the **broader definition** of virtualization: VMM applies the layering principle to the computer with three specific criteria of

- **equivalence**: *duplication* ensures that the exposed resource (i.e., the virtual machine) is equivalent with the underlying computer system
- **safety**: *isolation* requires that virtual machines are isolated from each other as well as from the hypervisor (enforced modularity)
- **performance**: *efficiency* needs the virtual system to show, at worst, a minor decrease in speed.

Popek and Goldberg Theorem: preliminaries

CPU Privilege Rings are structural layers that **limit interaction** between installed applications on a computer and core processes

- lower number = higher privilege
- certain CPU instructions only available in lower rings
- Windows/Linux use Ring 0 for running kernel and Ring 3 for running user programs
- interrupts change to Ring 0



Popek and Goldberg Theorem: preliminaries

The instruction set of a CPU can be categorized into **three** classes:

- **privileged** instructions can only be executed in supervisor mode and cause a trap when attempted from user mode
- **sensitive** instructions update the system state (control-sensitive), or its semantics depend on the actual values set in the system state (behavior-sensitive)
- **innocuous** instructions are unprivileged and non-sensitive.

Popek and Goldberg Theorem

“For any conventional third-generation computer, a VMM may be constructed if the set of **sensitive instructions** for that computer is a **subset** of the set of **privileged instructions**”

The theorem holds when all control-sensitive and behavior-sensitive instructions are also privileged, i.e.,

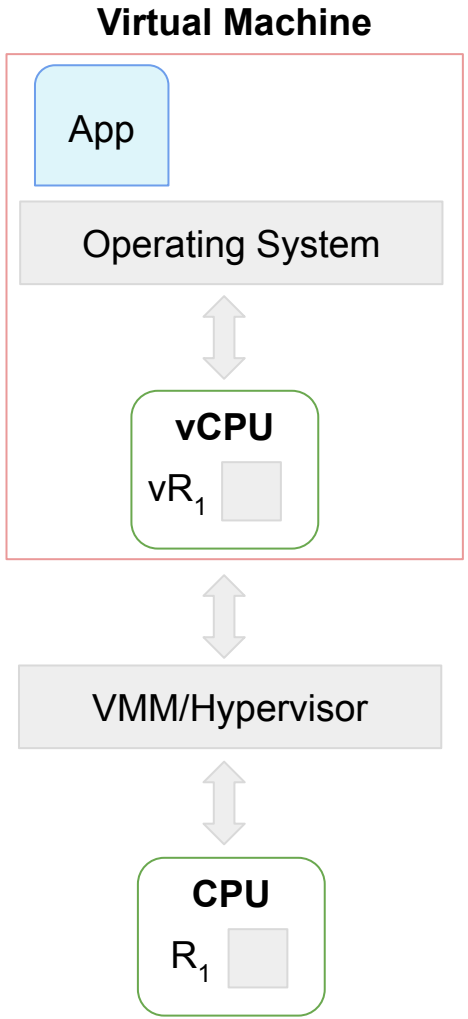
$$\{\text{control-sensitive}\} \cup \{\text{behavior-sensitive}\} \subseteq \{\text{privileged}\}:$$

Trap and emulate (T/E)

A CPU with an instruction set respecting the Popek/Goldberg theorem allows a VMM to use a **trap-and-emulate** strategy (and provides VMs that are "*efficient and isolated duplicates of the real machine*")

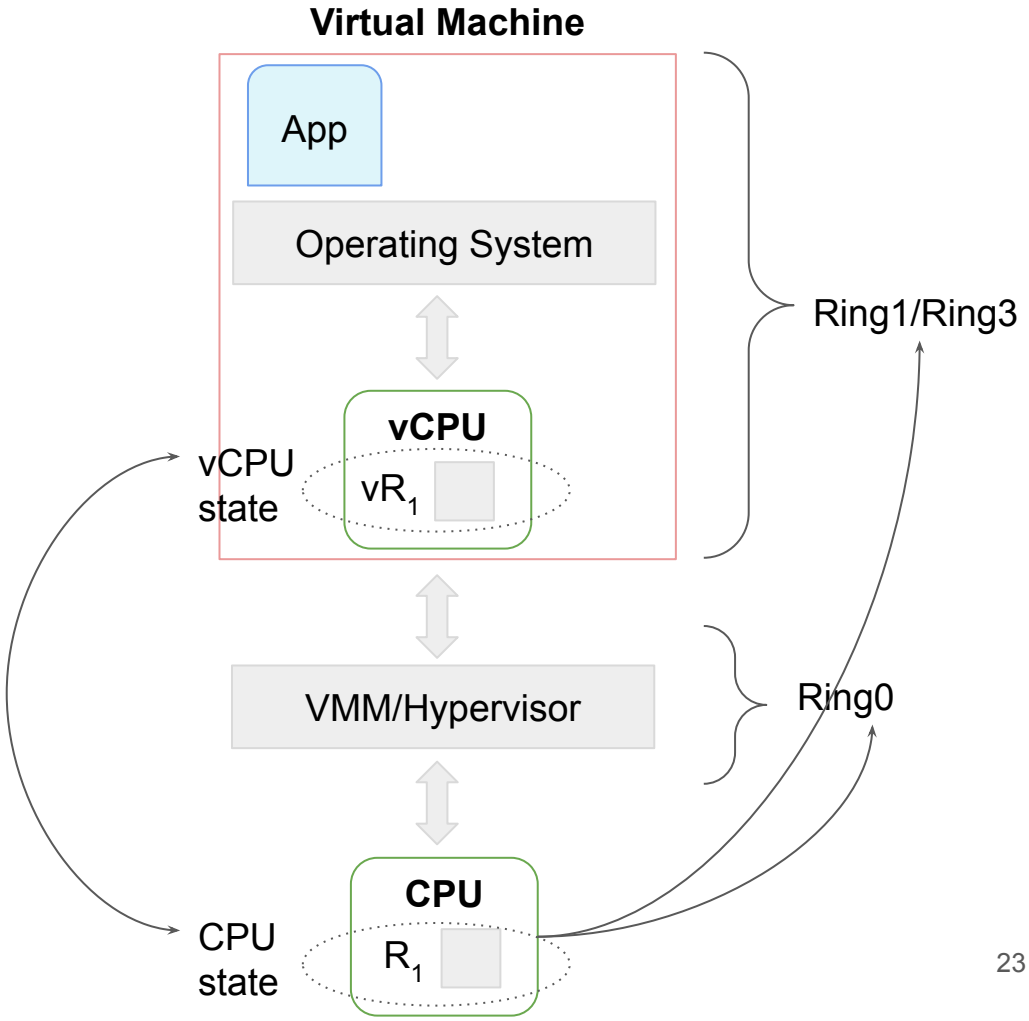
- **VMM** is executed in **privileged mode** (Ring 0) while **guest OSes** and **applications** run in **user mode** (Ring 1 and 3, respectively)
- all instructions executed by virtual CPUs are **executed by the actual CPU** in non-privileged mode (**Direct Execution**)
- when the virtual CPU tries to execute a **privileged instruction**, it triggers a **trap**
- the exception is managed by the **VMM** that **emulates the effects** of the privileged instruction on the virtual CPU (and return the control to the VM)

T/E (simplified)



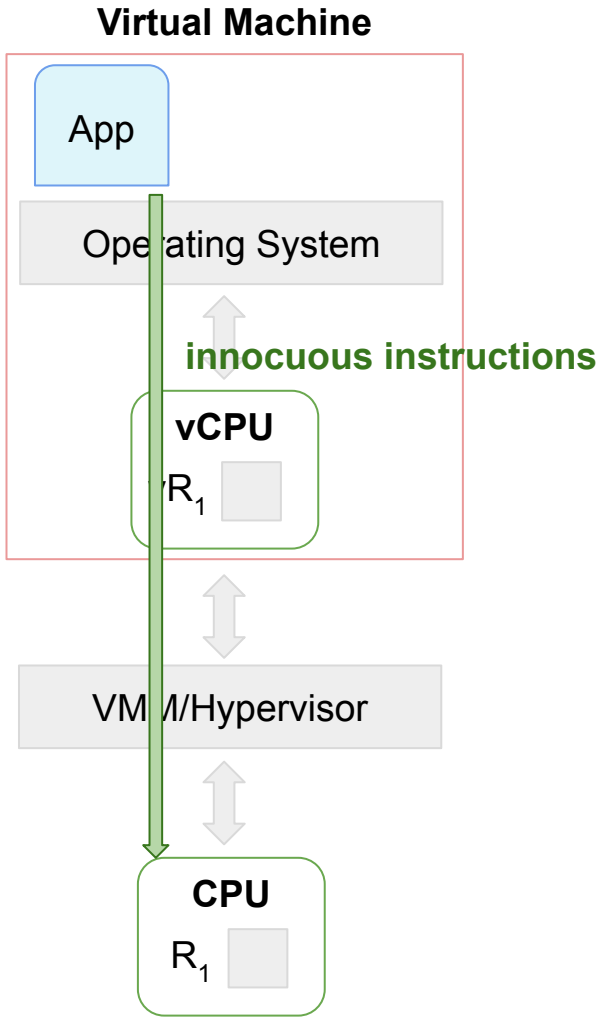
T/E (simplified)

Need to be kept coherent and synchronized before/during/after the VM runs



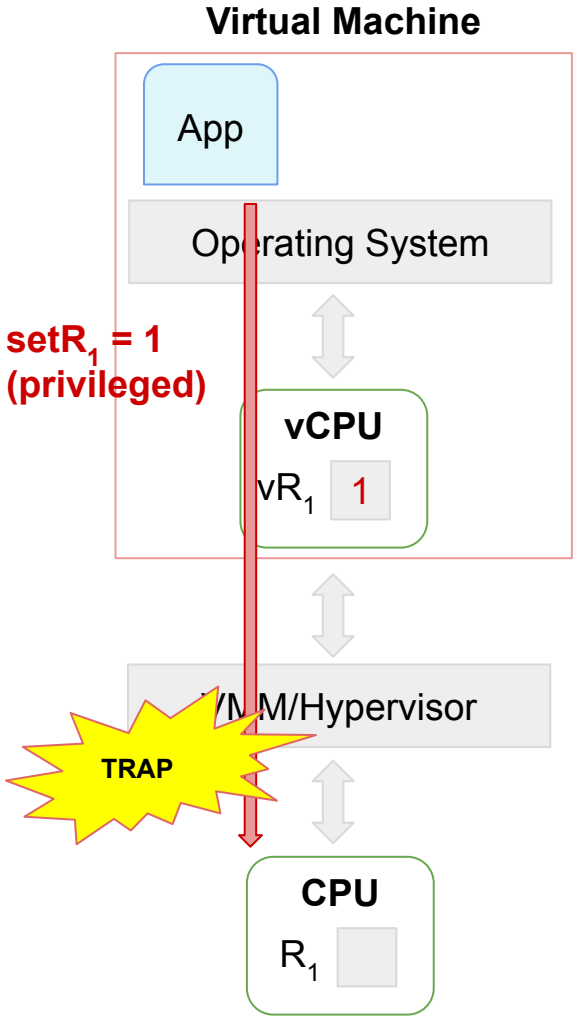
T/E (simplified)

- the application inside the VM runs at **Ring 3** like any other process running directly on the processor
- unprivileged and non-sensitive instructions are **executed directly** on the **real CPU**



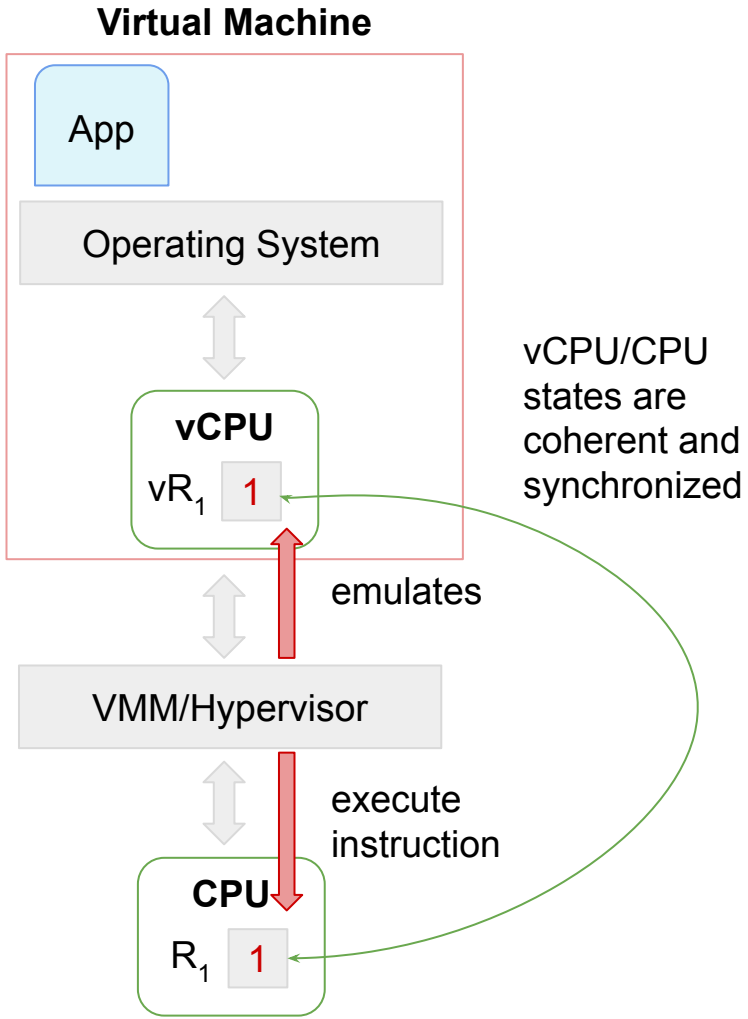
T/E (simplified)

- the application executes a **privileged instruction** to write 1 on the R_1 register
- the execution fires a TRAP that is caught by the VMM (it runs in Ring 0)



T/E (simplified)

- the VMM executes the privileged instruction
- the VMM emulates the effects of the privileged instructions



P/G Theorem violations

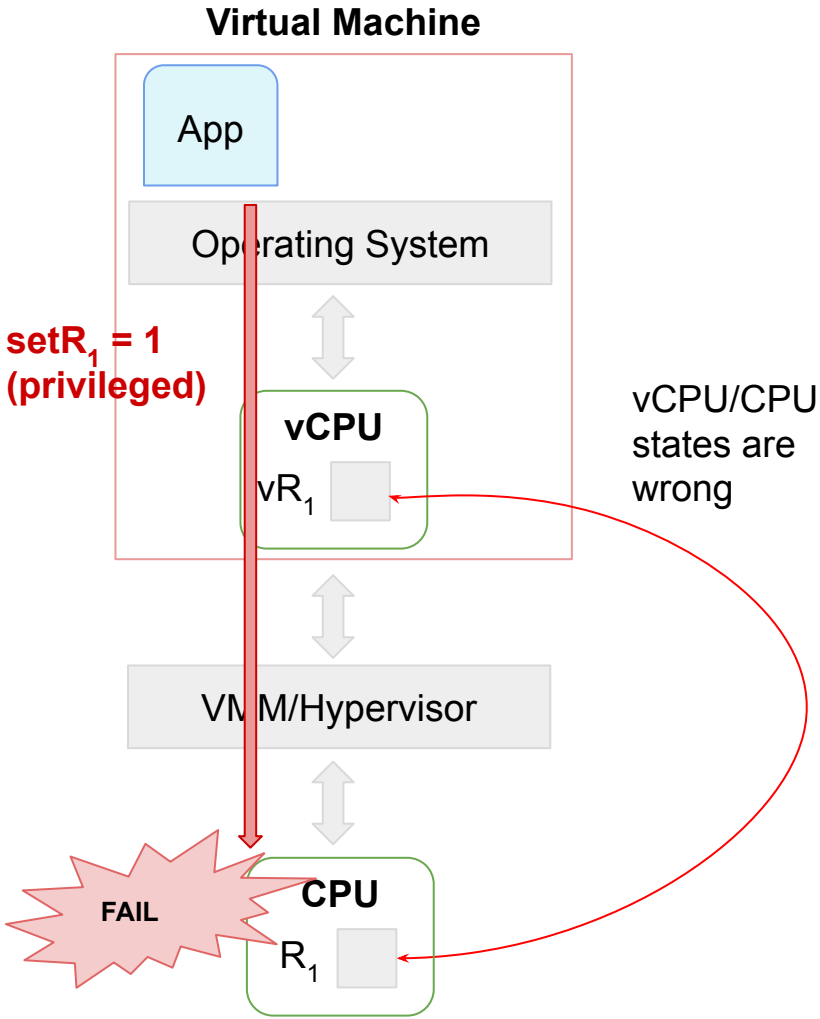
For **decades**, no widely-deployed architectures have complied with the requirements of Popek/Goldberg theorem. For example, **x86-32** has **17** sensitive instructions that violate such requirements.

Example. `popf` pops 16 bits from the top of the stack to the `%eflags` register

- bit 9 of `%eflags` masks interrupts (i.e., enables/disables interrupts)
- `popf` is sensitive but not privileged. What happens if guest OS (Ring 1) runs `popf` to `%eflags`?
- In Ring 0, `popf` can set bit 9, but the CPU **silently ignores** setting from `popf` of system flags (bit 9) when running in Ring 1

T/E (simplified)

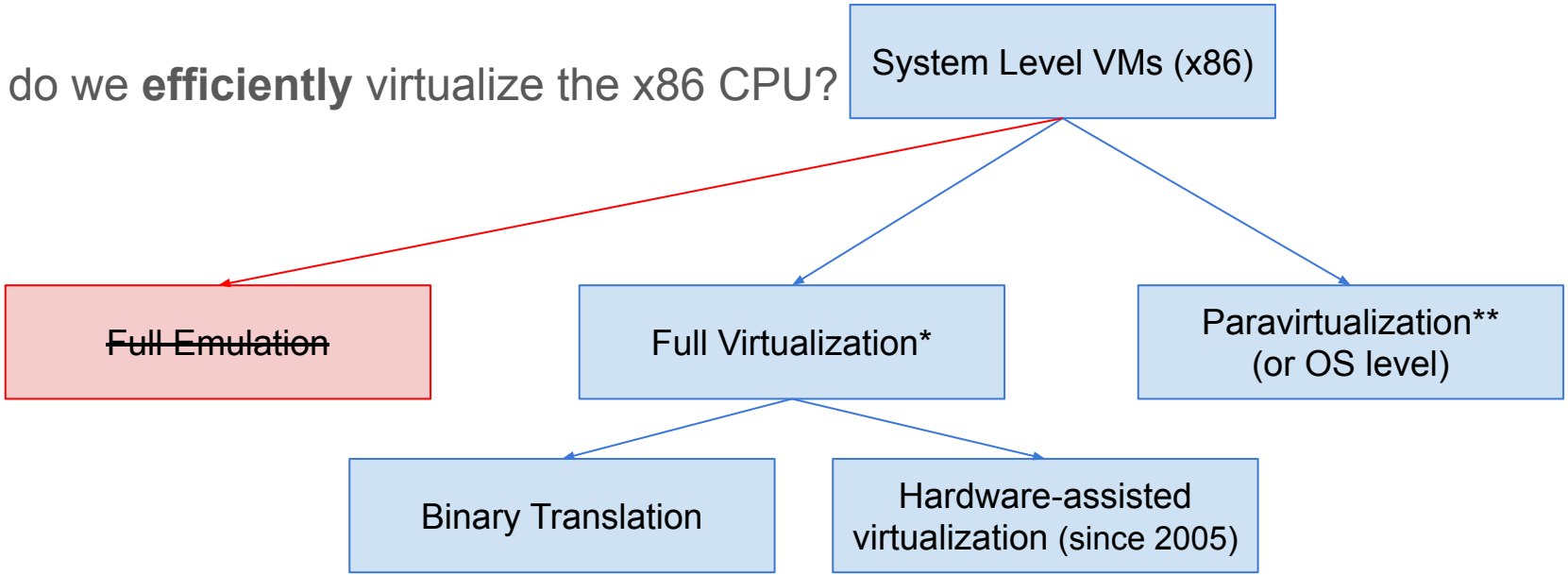
- let's consider `popf` == setR_1 and $R_1 \subset$ `%eflags`
- the instruction is executed on the real CPU but it fails silently
- the VMM can't catch the request to execute the instruction in a privileged mode
- CPU and vCPU continue the execution of the next instructions in a wrong state



Virtualizing the Unvirtualizable

Virtualizing the Unvirtualizable

How do we **efficiently** virtualize the x86 CPU?



*__Full Virtualization__ can be used to run any __unmodified__ OSs transparently

**__Paravirtualization__ requires a __modified__ version of OSes

Dynamic Binary Translation

Employed by VMWare engineers in first x86 virtualization solution (released in 1999)

- although **Direct Execution** (DE) could not be used to virtualize the entire x86 architecture, it could actually be employed **most of the time**, in particular to run applications
- otherwise **Dynamic Binary Translation** (DBT): an efficient form of emulation
- known CPU states decide whether to use DE or DBT

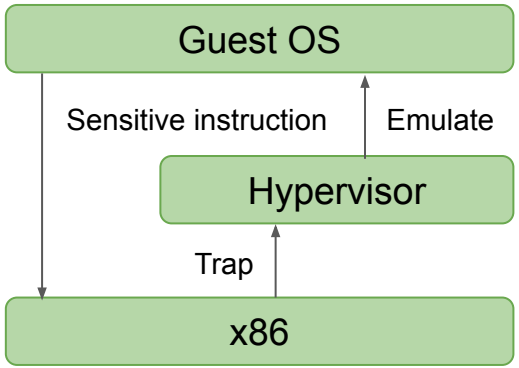
Dynamic Binary Translation

It works at runtime and takes as the input binary x86 code. Rather than interpreting the instructions one by one, DBT **compiles a group of instructions** (basic blocks) into a fragment of **executable code**

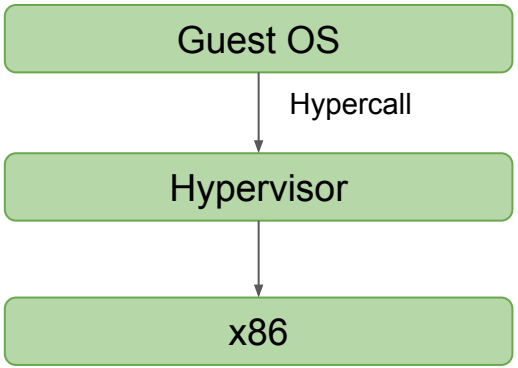
- before compiling it optimizes the code and makes it **suitable for handling sensitive instructions** ⇒ the **processor executes natively** each basic block
- the code is stored in a large **Translation Cache** (TC) ⇒ blocks are likely to appear many times and are ready for the direct execution
- blocks are chained ⇒ it allows for direct jumps between compiled fragments

Paravirtualization

- enlighten the Guest OS that it is running in a VM
- the Guest OS Kernel must be modified to invoke virtual APIs (hypercalls) exposed by the VMM, instead of sensitive instructions



Trap and Emulate



Paravirtualization

Paravirtualization: pros and cons

- is **highly efficient**
- requires that guest operating systems be modified, a possible issue with **closed-source/proprietary OSs**
- **maintenance cost** of the paravirtualized OSs
- each **VMM** has its **own hypercalls**. Some attempts to create open interfaces exist, e.g., Paravirt-ops (Linux community, IBM, Red Hat, VMware, and XenSource)

Minimal/non-intrusive paravirtualization

Guest tools and optimized virtual device drivers are solutions that resembles paravirtualization techniques without requiring OS kernel modification.

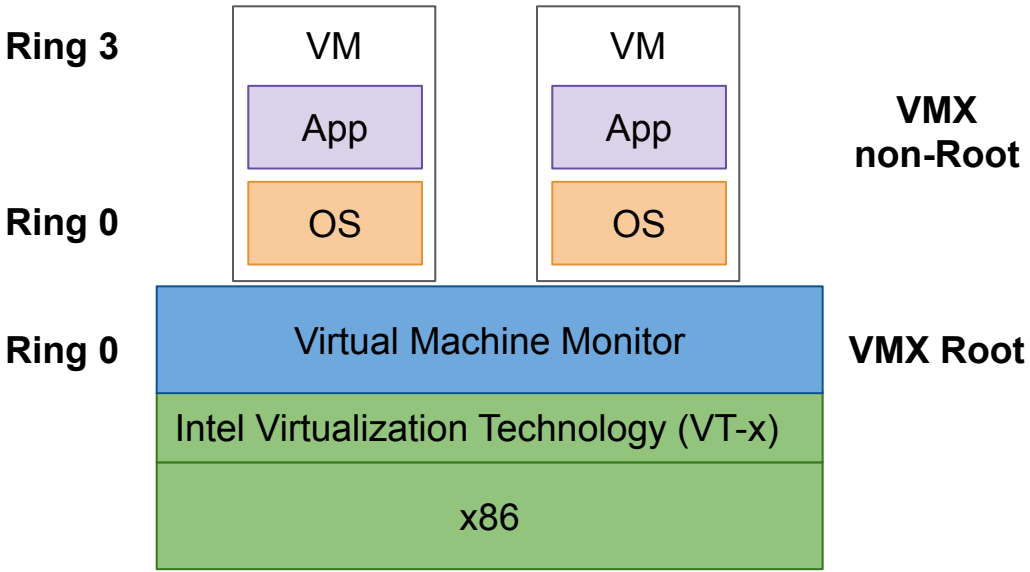
For example

- The VMware tools service provides a backdoor to the VMM Hypervisor used for services such as time synchronization, logging and guest shutdown
- Vmxnet is a paravirtualized I/O device driver that shares data structures with the hypervisor (improve throughput and reduce CPU utilization)

Hardware-assisted virtualization

- **Intel VT-x/AMD SVM** represent the extensions to traditional CPUs to support hardware-assisted virtualization (introduced in 2005)
- Recent hypervisors can only run on processors with these extensions (in 2018 VMware announced that vSphere 6.5 was the final release that supports binary translation)
- Intel introduces Virtual Machine Extension (VMX) with
  - **new privilege modes**: root/non root (guest) mode, each supporting all four x86 protection rings
  - **new instructions** (`VMXON`, `VMXOFF`, …)
  - **new data structure**: Virtual Machine Control Structure (VMCS)

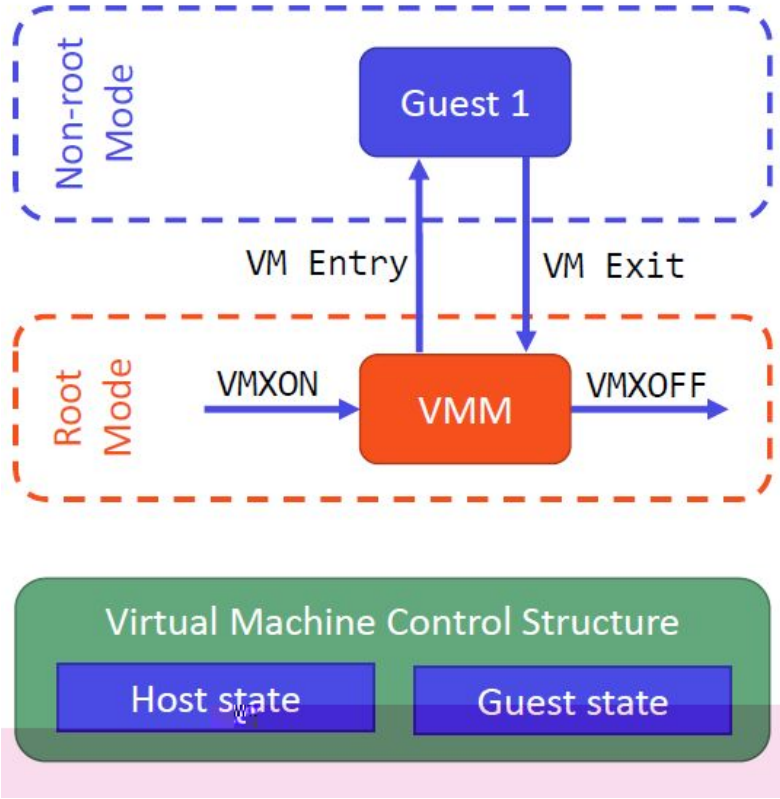
Virtual Machine Extension (VMX)



- VMM executes in VMX root-mode
- Guest OS deprivileging eliminated
- Guest OS runs directly on hardware

Virtual Machine Extension (VMX)

- VMM enters VMX operation with `VMXON`
- VMM can perform VM Entries (e.g., `VMLAUNCH`, `VMRESUME`)
- Upon `VM Entry`, the processor state is loaded from Guest state in VMCS, control is transferred from VMM to VM
- Upon `VM Exit`, the processor state is saved in Guest state in VMCS, processor state from Host state is restored, control returns to the VMM



(image by `luigiliberolucio.starace@unina.it`)

Virtualization: Memory

Virtualization: memory (preliminaries)

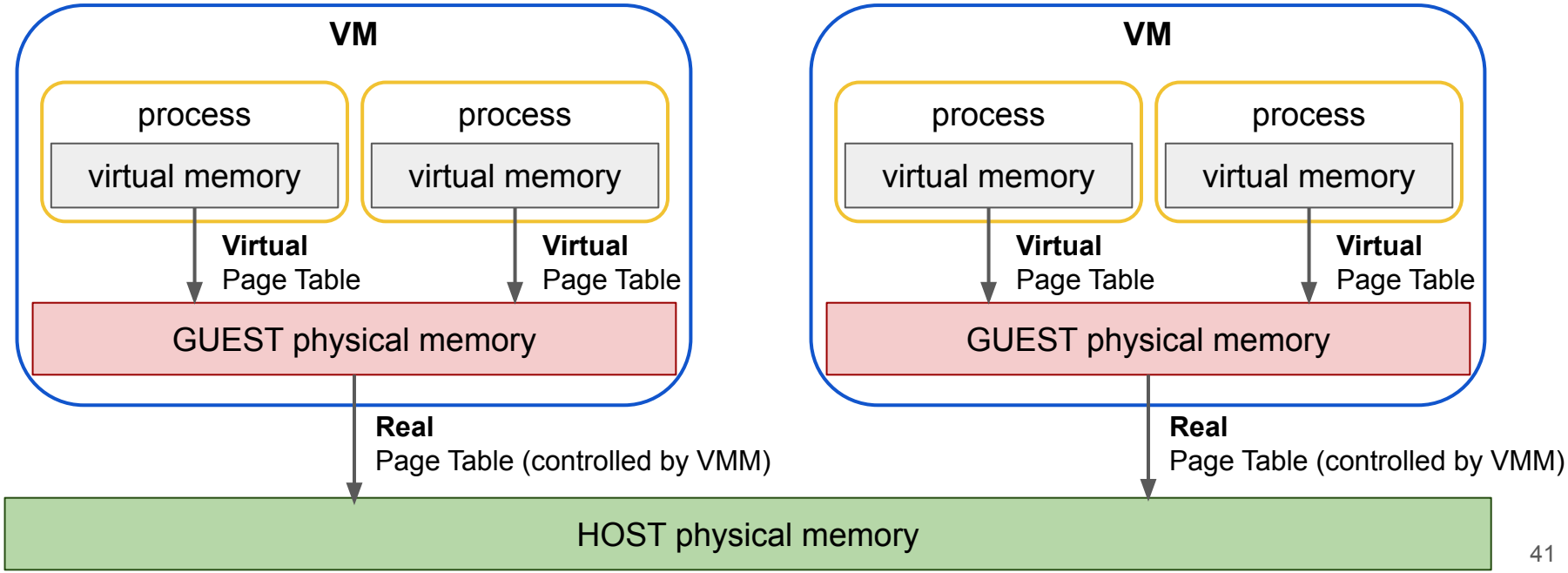
Operating systems **already leverage virtualization** for physical memory management. In particular:

- the **physical** address space is the **system** RAM; **virtual** memory is the memory that is controlled by an **OS**
- virtual and physical are both split in **fixed-length blocks**, i.e., **pages** and **frames**, respectively
- a dedicated CPU unit (the **MMU**) **corresponds** the **virtual** address to the proper **physical** address & vice versa
- MMU keeps correspondences in the physical RAM (**Page Table**); it uses a **Translation-Lookaside Buffer** (**TLB**) to **cache** recent memory translations of virtual to physical and **reduce the time** taken to access physical memory[1]

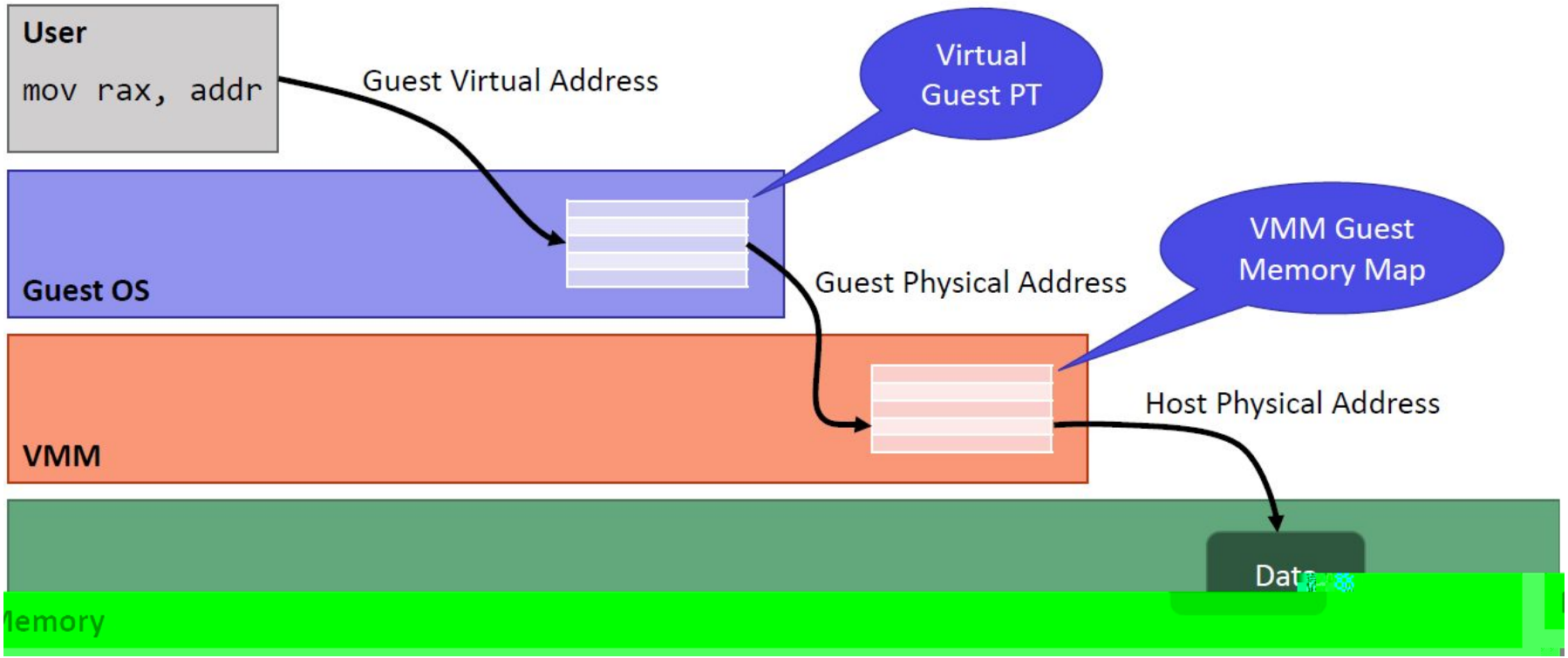
[1]Typically access times for a TLB is ~10 ns where main memory access times are around 100 ns

Virtualization: memory

Hypervisors introduce a **further level of indirection** that they have to manage to keep VMs efficient.



Virtualization: memory



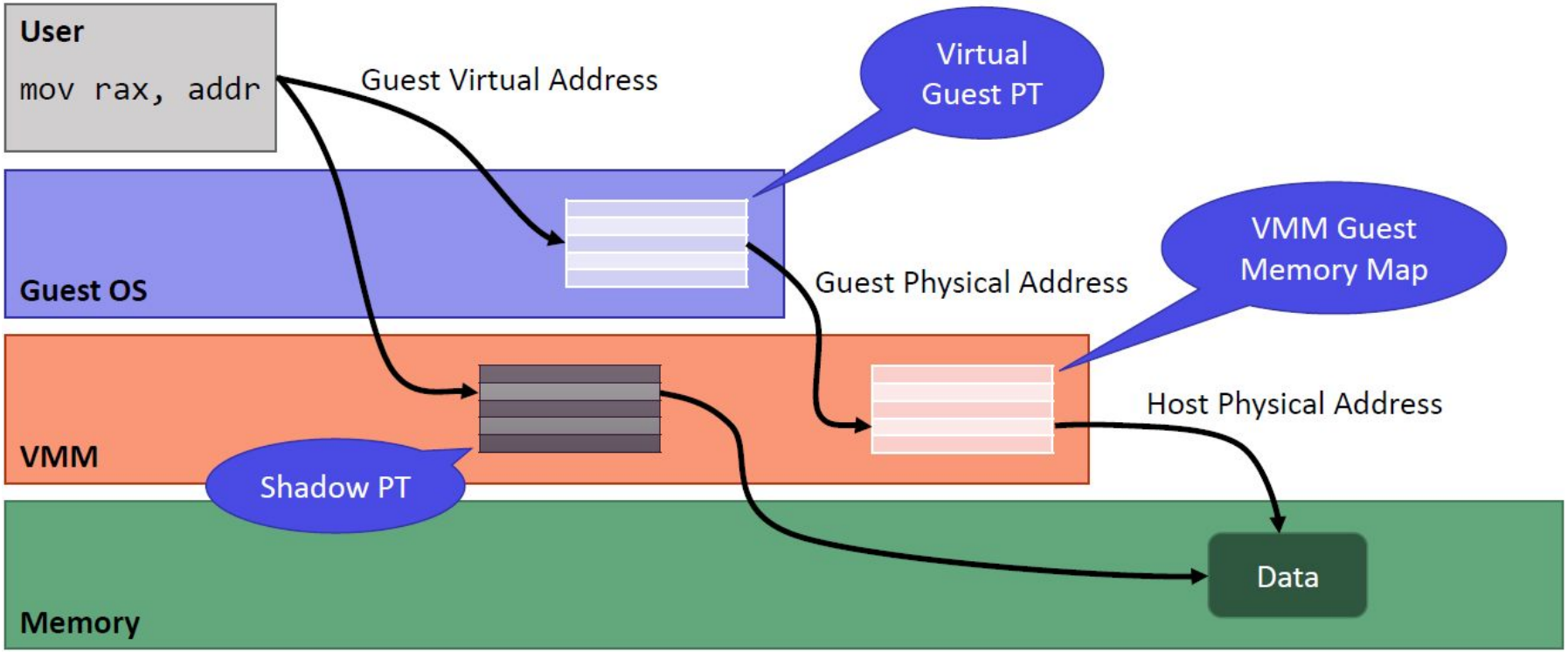
(image by luigiliberolucio.starace@unina.it)

Shadow Page Tables

To avoid a decrease of performance due to the complexity of double memory mapping, VMMs maintain a **Shadow Page Tables** (**SPT**).

- SPT maps **Guest Virtual Addresses** to **Host Physical Addresses**
- Guest OSs create and manage PTs for their processes as usual, but these **PTs are not used** in the MMU anymore
- VMM creates and manages **SPT**s, mapping virtual pages to real machine pages. These are loaded in the Guest MMU

Shadow Page Tables



(image by luigiliberolucio.starace@unina.it)

Building Shadow Page Tables

VMM requires to maintain consistent SPTs. To this aim

- the Guest OS can continue to modify its PT at any time but **hardware frames** containing **Guest PTs** are **marked as read only**
- if Guest updates its PT an **exception** (a "*hypervisor-induced*" page fault[1]) is generated
- VMM **catches** the exception, examines the faulting write, and **updates** the SPT

Unfortunately, **page faults are extra expensive** in virtualized environments, because they lead to `VM exits` ⇒ the hypervisor regains control, and this involves saving and restoring lots of state, and may tens of thousands of cycles.

[1]a "conventional" page fault involves recovering pages actually swapped out of RAM

Second Level Address Translation (SLAT)

- **SPTs** are a **purely software** managed solution
- **SLAT** is a **hardware assisted** solution for memory virtualization, aka Nested Page Tables (AMD) or Extended Page Tables (Intel)
- SLAT simply **extend PTs** so that they can have several layers of mapping: guest virtual addresses, guest physical addresses, and host physical addresses
- No need to maintain SPTs and no need for VM exits
- Switching virtual machines changes the mapping the same way an OS does when switching processes

Memory virtualization and VMM tricks

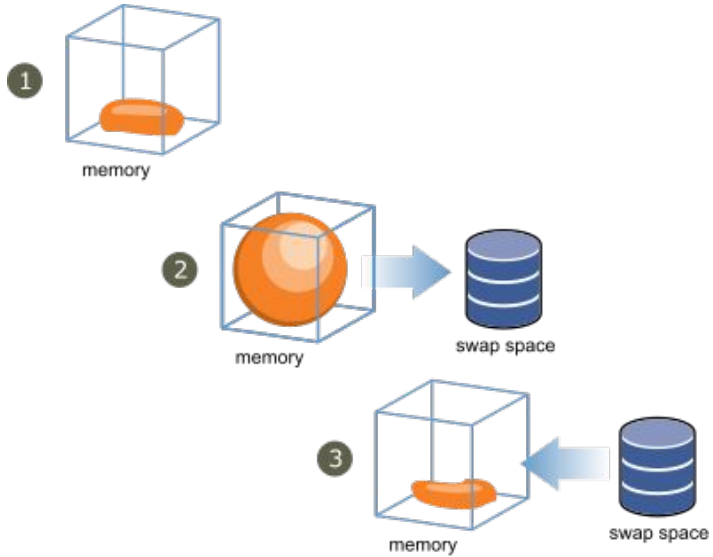
The VMM can play tricks with virtual memory just like an OS can

- **Paging**
  - a VMM can page parts of a guest to disk
  - it allows the VMM to allocate more virtual memory to VMs than there is actual physical memory (**memory overcommitment**)
- **Shared pages**
  - a VMM can share read-only pages between guests, e.g., the Linux kernel (**memory deduplication**)

Memory Ballooning

Reclaiming memory (and paging out guest memory) is **complex** because a VMM has no clue which pages should be kept in memory. **Ballooning** is a (relaxed) paravirtualization technique to reclaim memory **directly from Guest OSes** leveraging guest agents.

1. A small balloon module is loaded into the guest OS. It can **communicate** with the **VMM** via a **private channel**
2. When the VMM wants to **reclaim** memory, it instructs the driver to **inflate** by allocating pinned physical pages within the VM. Inflating the balloon increases memory pressure in the guest OS, causing it to invoke its own native memory management algorithms
3. **Deflating** the balloon **frees up** memory for general use within the guest OS.



(image from VMware-VSphere docs)