Introduction – Part 2: System-level virtual machines – Some fundamental definitions

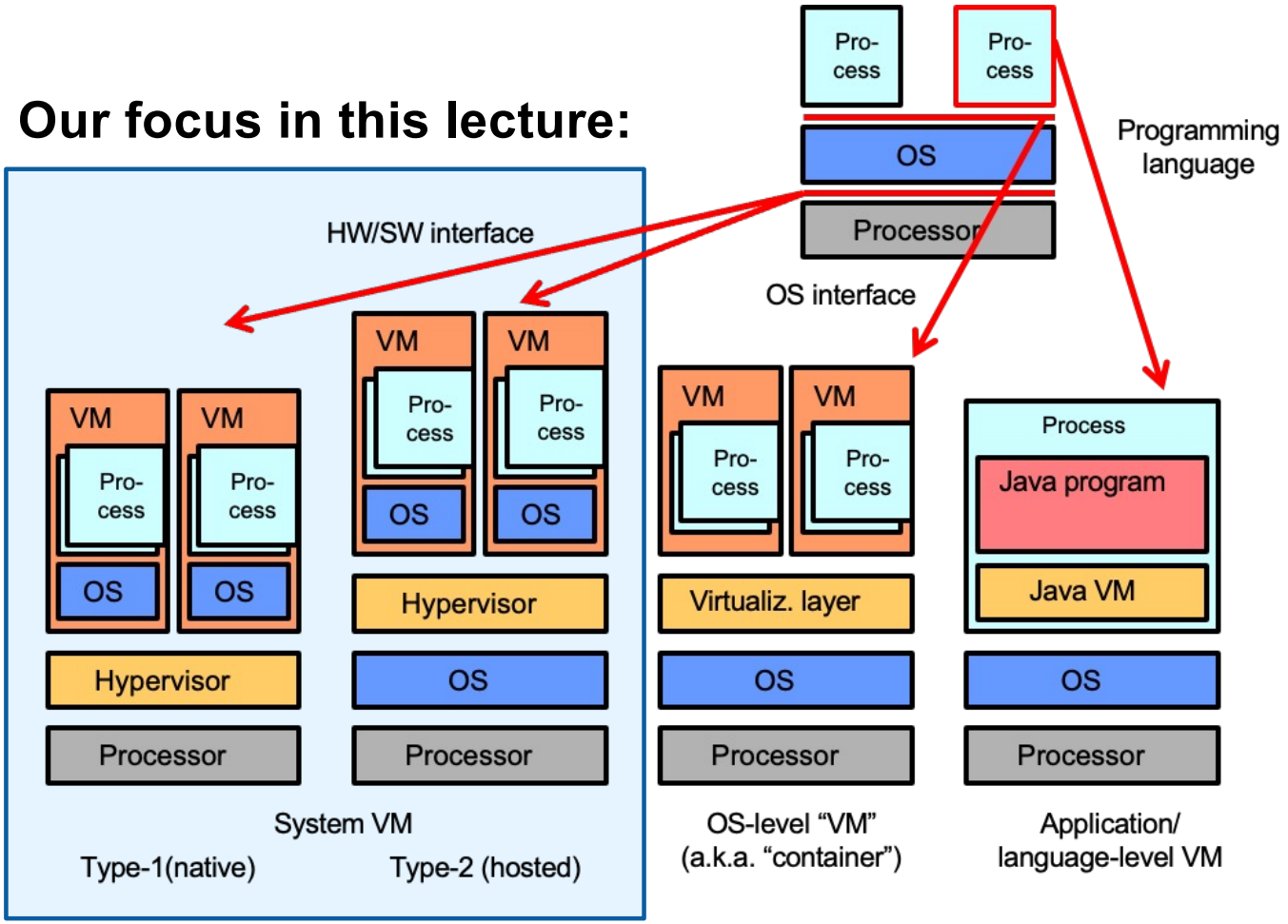
Renaud Lachaize

Univ. Grenoble Alpes

M2 MoSIG & Phelma-SEOC

October 2025

Reminder: Types of virtualization



(Figure courtesy of Gernot Heiser, UNSW Sydney)

Reminder: System-level virtual machines – Basic definition (1/2)

- *A (system-level) virtual machine is an **efficient & isolated duplicate** of a real (physical) machine*
- Often abbreviated as **"System VMs"**, **"VMs"** or **"Guest VMs"** or **"Guests"**
- Machine **resources** include CPU(s), main memory (RAM), I/O devices (disks, NICs, peripherals …)
- Goals:
  - **"Duplicate"** (a.k.a. **"fidelity"**)**:** Code running in a VM cannot distinguish between real or virtual hardware.
  - **"Isolated":** Several VMs execute concurrently on the same machine without interfering with each other (at least w.r.t. safety and security considerations).
  - **"Efficient":** VMs should execute at a speed close to that of real hardware.

Reminder: System-level virtual machines – Basic definition (2/2)

The resources exported by a virtual machine may or may not correspond to the ones of the underlying physical hardware.

Typically, in practice, on a server/desktop machine:

- **Regarding the functional interface:**
  - The VM exports the same CPU model as the one of the physical machine (*same ISA: Instruction Set Architecture – such as Intel/AMD x86-64*).
  - Some ISA features (available for the physical machine) may be disabled for the VM.
  - The VM may or may not export the same types of I/O devices as the one of the physical machine.
- **Regarding the amount of available resources:**
  - A VM typically exports fewer resources than the total of physical resources.
  - Two main reasons:
    - Virtualization overhead
    - Concurrent execution of multiple VMs (with decent performance)

A very brief history of system VMs (1/4)

- System VM technology has been used **for a very long time** (from the **1960s** until today) **in the context of mainframe computers**, which are a typical example of **hardware/software co-design**.
  - **Famous example: IBM mainframes** (System/360, System/370, System/390, and more recently the IBM Z family).

- In terms of **hardware/software interface**, the **technical requirements for virtualization** (regarding **fidelity** + **isolation** + **efficiency**) have been clearly identified since (at least) the early 1970s.

A very brief history of system VMs (2/4)

- With the boom of the ”small machines” (PCs, workstations, small servers, and later embedded systems), **new hardware architectures** (CPUs and devices) have appeared between the 1970s and the 1990s.
  - Typical CPU examples: Intel/AMD x86, ARM, MIPS

- Because they were mostly designed for cheap and ~single-purpose computers, **most of these architectures** have **hardware/software interfaces** that **do not comply** with the **well-known requirements** for virtualization.

- Roughly speaking, **this does not mean that running system VMs on such hardware is impossible** … **but rather that:**
  - This requires complex engineering “tricks” (such as dynamic binary translation) and possibly trade-offs between the different aspects (fidelity, isolation, efficiency).
  - And even in the best cases, efficiency is limited.

A very brief history of system VMs (3/4)

- In the **mid/end-1990s**, with the “Internet boom” and the early days of Cloud computing, system-level virtualization gained **a new momentum**.

- Academic researchers at Stanford University (within the “Disco” project) developed (software-level) techniques to improve the feasibility and efficiency of system VMs on “non-virtualizable” architectures such as MIPS and Intel x86.

  - This led to the creation of the **VMware** company.

  - Many other (academic and industrial) works on efficient virtualization followed.

A very brief history of system VMs (4/4)

- Starting from the mid-2000s, **most hardware vendors released new versions of their architectures, featuring built-in support for virtualization**.
- This enabled improved performance and a simpler design of the software stack.

- Note that this hardware support for virtualization was typically (re)introduced in several steps.
- For example, in the case of Intel x86-64 (“Intel VT extensions”):
  - ~2005: support for CPU virtualization (new execution modes)
  - ~2008-2010: Support for MMU virtualization (extended/nested page tables)
  - ~2005-2012: Support for I/O & interrupt virtualization
  - (some new features still added regularly today)

Software layer(s) for system-level virtualization

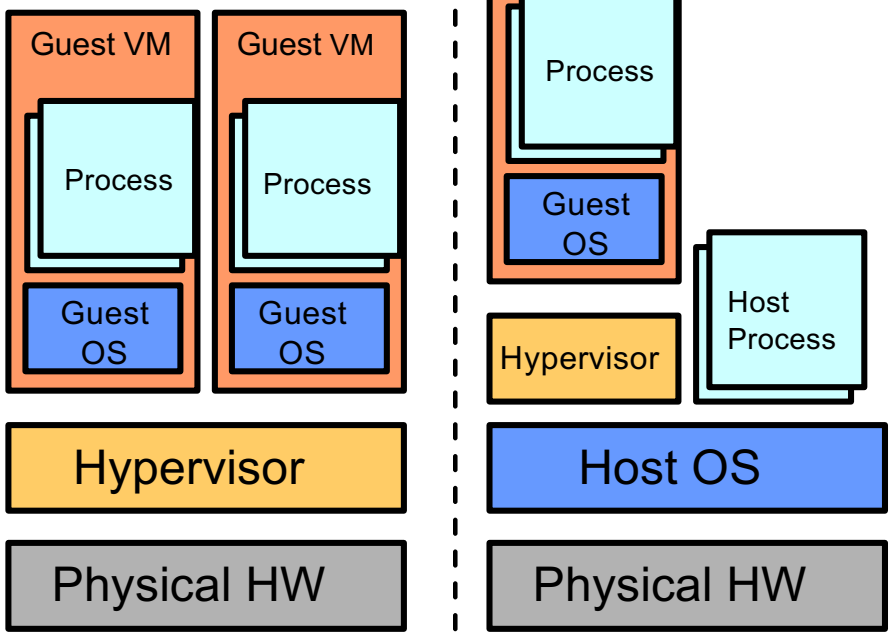
- Even with hardware-assisted virtualization, software support remains necessary for managing certain aspects of the execution of a system VM.
- The software is charge of this virtualization is not necessarily structured in a monolithic way. There are often multiple sub-components involved, even when considering only the elementary aspects needed for the execution of a VM.
- The software layers in charge of supporting (system-level) virtual machines are often called a **"Hypervisor"** and/or a **"Virtual Machine Monitor (VMM)"**.
- **Important warning:**
  - Unfortunately, **there is no standardized definition of the vocabulary**.
  - Quite often, "hypervisor" and "VMM" are used interchangeably but this is not always the case.
  - Sometimes, one of these two expressions refers to the complete virtualization software, while the other one refers to a specific portion/sub-component.
  - Sometimes, these expressions refers to distinct, non-overlapping (and cooperating) sub-components.

Types of hypervisors (1/5)

- In terms of software architecture, there are two main design approaches for the base layers of a virtualization stack.
- Here, we mainly focus on the virtualization of CPU and memory resources (i.e., for the moment we do not consider the virtualization of I/O devices).

- **Type-1 (a.k.a. "native") hypervisor:**
  - The hypervisor **runs directly on the bare physical machine**.
  - **Examples:** Xen, Microsoft Hyper-V, VMware ESX/vSphere
- **Type-2 (a.k.a. "hosted") hypervisor:**
  - The hypervisor **runs on an extended host**, which manages the physical machine.
  - Examples: Linux KVM, VirtualBox, VMware Workstation, VMware Fusion, Parallels

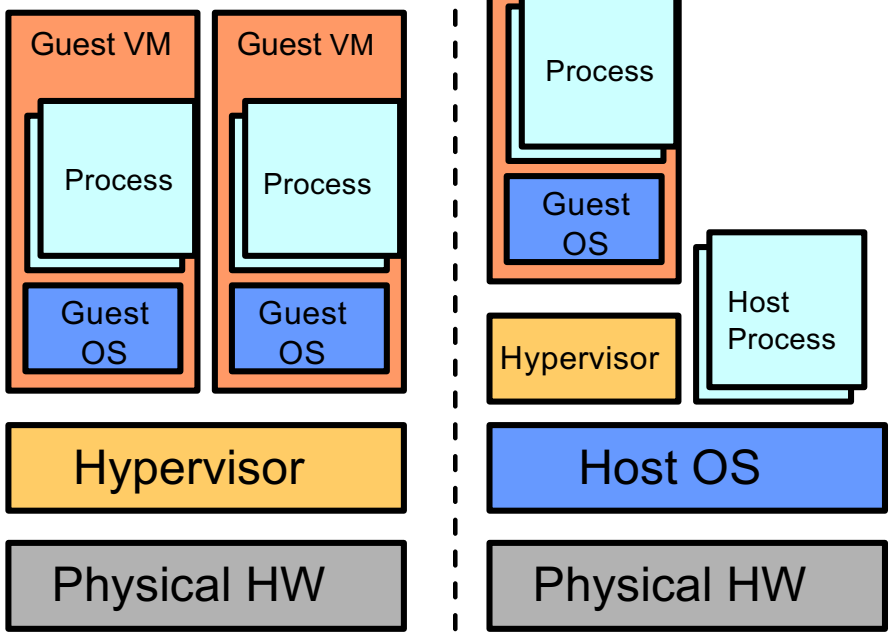
Types of hypervisors (2/5)



In any case:

- Each guest VM has it own (guest) OS instance.
- Different VMs may have similar or different guest OSes.
- The expression “host software” corresponds to the layer(s) below the guests.

Types of hypervisors (3/5)



Warning:

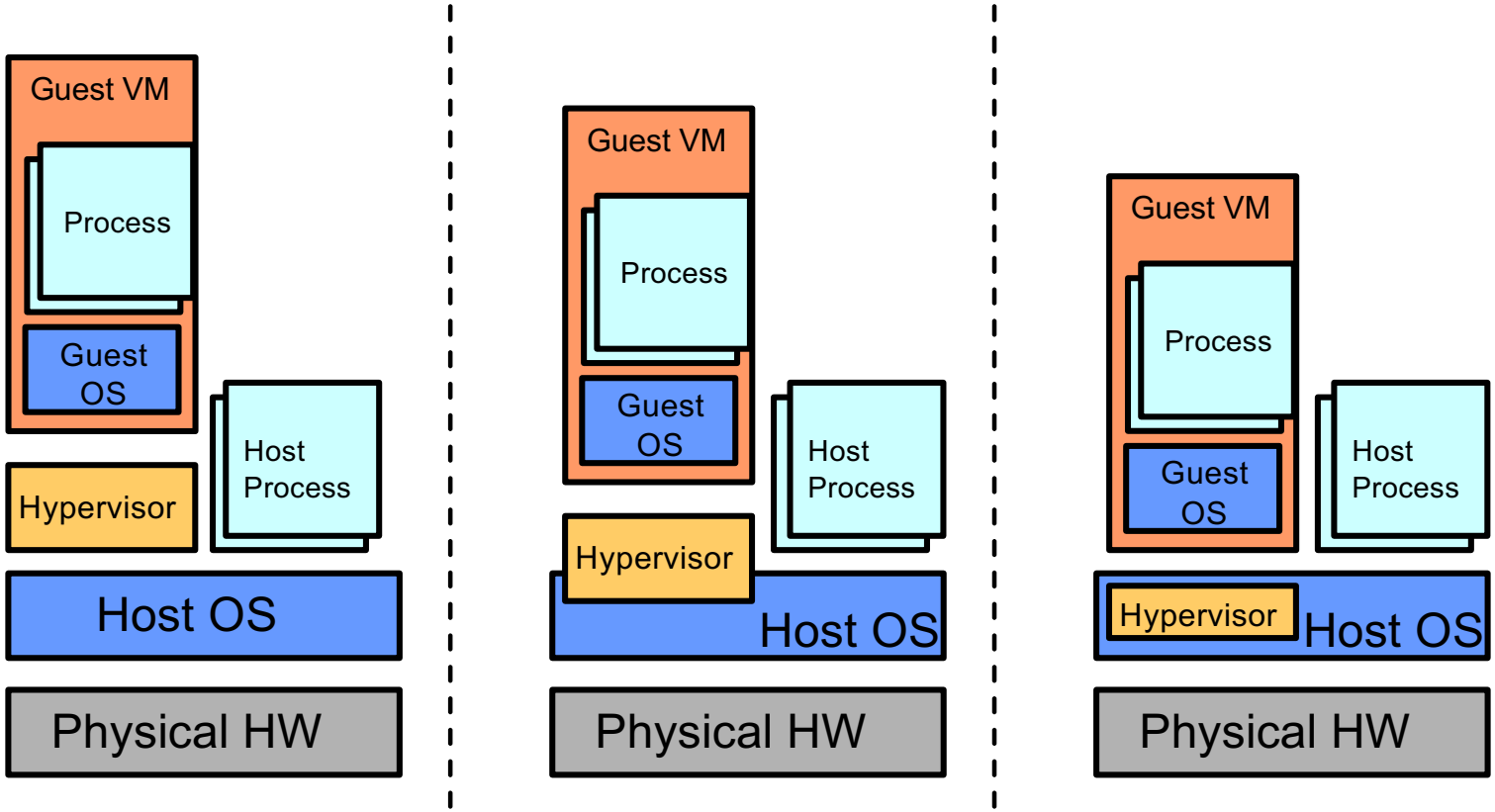
- These two categories have been introduced in the early 1970s.
- However, over the years, the definitions have been used and interpreted in various/loose ways, leading to **some confusion and disagreements** (even today, even between experts).

Types of hypervisors (4/5)

- In both designs (type 1 and type 2), the hypervisor is in charge of creating the virtual machines and handling the virtualization-related events (i.e., traps and/or hypercalls regarding hardware emulation – as explained later).
- **In a type-1 setup, the hypervisor is in charge of CPU scheduling and (real) hardware resource allocation.** Thus, the code of a type-1 hypervisor also includes code that is not strictly focused on virtualization.
- In a type-2 setup, the scheduling and resource management aspects are handled by the host OS.
- **The distinction between type-1 and type-2 does not make specific assumptions about the execution mode (hardware privileges) in which the hypervisor is running.**
- **It is possible to have a type-2 design in which the hypervisor runs with full hardware privileges.** Therefore, a type-2 design can be efficient. Example: Linux KVM.

Types of hypervisors (5/5)

In the case of a type-2 approach, various levels of integration (between the hypervisor and the host OS) are possible:



The Popek & Goldberg theorem (1/4)

- A seminal publication, providing **a formalization of the requirements** for (efficiently) **virtualizable architectures**.

- Research article published in **1974**:
  - *"Formal requirements for virtualizable third-generation architectures"*, Communications of the ACM. Volume 17, Issue 7. July 1974.
  - Authors: Gearld Popek & Robert Goldberg, computer scientists at UCLA.

The Popek & Goldberg theorem (2/4)

- **Some definitions regarding CPU instructions:**
  - **Sensitive instructions:** Instructions whose behavior (i.e., side effects) differ according to the current execution mode (user mode or supervisor mode).
  - **Privileged instructions:** Instructions that cause a trap (i.e., an exception) if executed in user mode.
- **Main idea of the theorem:**
  - A machine is (efficiently) virtualizable **only if the sensitive instructions are a subset of the privileged instructions**.
  - In other words, whenever you try to do something forbidden in user mode, the hardware should trap.
  - Also, just attempting to read sensitive state in user mode should trigger a trap.
  - If the above conditions are fulfilled, a so-called **"trap-and-emulate" approach** becomes viable: simply run guest code natively (in unprivileged mode) and handle traps to emulate sensitive instructions.

The Popek & Goldberg theorem (3/4)

- **Case study: Why is the (legacy) Intel x86 architecture not compliant with the requirements of the theorem?**
  - **Example 1:** the POPF instruction
    - Replaces the flags register, impacting the bit that enables/disables interrupts
    - When executed in user mode, this instruction has simply no effect on the interrupt bit … but does not trap
  - **Example 2:** Reading the code-segment selector
    - When executed in user mode, this instruction does not trap and reveals the current execution mode
  - (And also many other issues …)

The Popek & Goldberg theorem (4/4)

- **A performance warning about "trap and emulate":**
  - **Simply fulfilling the requirements of the P&G theorem does not necessarily imply that the performance overhead of virtualization is low/negligible.**
  - Indeed, on most CPU architectures, like other causes of execution mode switches, **traps have a significant impact on performance**.

- Therefore, most virtualization systems use **one or several techniques** to **reduce the occurrences of traps**.
  - In other words: Reduce the frequency of intervention of the virtualization code. Try to increase the ratio of instructions that remain within the context of the guest.
  - **Approach 1 – Paravirtualization:** Modify guest code to introduce a higher-level interface ("hypercalls") between the guest and the virtualization layers.
  - **Approach 2 – Hardware-assisted CPU virtualization:** Introduce additional execution mode(s) for the virtualization layer. Thus, many traps are only between guest user code (guest applications/libraries) and guest kernel code.

Paravirtualization

- Name coined around 2002 (in the context of the Denali and Xen projects) but the concept has existed since the early 1970s.
- **Main ideas:**
  - Modify the guest code (usually, only the guest kernel) to make it aware that it runs in a virtualized context.
  - Introduce higher-level guest-host interface enabling explicit “hypercalls”.
- **Main benefits:**
  - More efficient interactions between guest and host
  - Simplified implementation of the virtualization system
- **Different scopes are possible**
  - Paravirtualized devices (modifications only via installation of new device drivers)
  - More general kernel modifications (e.g., for memory management)
- This approach can be used with or without hardware support for virtualization.

CPU virtualization (1/2)

- **How to virtualize the CPU?**
  - We will provide a very quick overview.
  - **To simplify the discussion**, we consider a setup with a type-1 hypervisor, and a single guest VM, with a single CPU, running a single application.
- **Case 1: Without hardware support (but assuming trap & emulate)**
  - **All the code of the guest VM runs in user mode** (guest applications and guest kernel).
  - **The hypervisor code runs in supervisor mode.**
  - In essence, the hypervisor configures the hardware so that the guest application and the guest kernel are running as distinct “processes” (without privileges).
  - Upon each syscall or exception (e.g., page fault) triggered by the guest application, the hypervisor catches the hardware trap, forwards the event to the guest kernel, and schedules it.
  - Whenever the guest kernel executes a sensitive instruction (or issues an explicit hypercall), this triggers a trap to the hypervisor, which emulates the instruction, and then transfers back control to the guest kernel).
  - When a guest kernel syscall/exception handler completes, this triggers a trap to the hypervisor, which transfers control back to the guest application.

CPU virtualization (2/2)

- **Case 2: With hardware support ("HVM": Hardware-assisted virtual machines)**
  - Roughly speaking, **we now have distinct execution modes** for the guest application (**user mode**), the guest kernel, (**supervisor mode**) and the hypervisor (**hypervisor mode**).
  - Transitions between the guest application and the guest kernel take place "as usual" (i.e., like in a non-virtualized system), upon syscalls or exceptions.
  - Whenever the guest kernel executes a sensitive instruction (or issues an explicit hypercall), this triggers a trap to the hypervisor, which emulates the instruction, and then transfers back control to the guest kernel).
  - Thus, in some/many cases (but not all) system calls and exceptions triggered by guest applications do not require the hypervisor intervention.

  - Note: At the hardware level, mode switches within the guest (between user mode and supervisor mode) have typically a lower performance overhead than the transitions between the guest and the hypervisor mode.
  - The transitions between the guest context and the hypervisor context are often named **"VM exits"** (from guest to hypervisor) and **"VM entries"** (from hypervisor to guest).

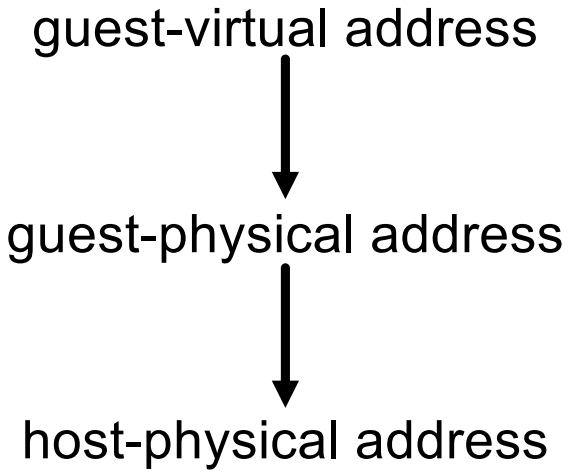
Memory virtualization (1/4)

- **Traditional machines (i.e., without hypervisors) already virtualize the main memory**
  - The **hardware MMU maps virtual addresses to physical addresses**.
  - On most CPU architectures, the translation between virtual and physical addresses is performed via **paging**, i.e., at the granularity of (fixed-sized) pages using a per-process data structure (**"page table"** or **"PT"**) configured by the kernel.
  - The total virtual memory capacity exposed to each process is typically larger than the total physical memory capacity. This is possible thanks to multiple techniques used by the OS kernel, including disk swapping.

- **How to virtualize the memory for a system VM?**
  - We will provide a very quick overview.
  - **To simplify the discussion**, we consider a setup with a type-1 hypervisor, and a single guest VM, with a single CPU, running a single application. Also, we do not consider disk swapping.

Memory virtualization (2/4)

- **Some vocabulary** about **addresses**
  - In the context of a **guest VM**, the guest kernel configures the (virtual) MMU to map **"guest-virtual" memory** addresses to **"guest-physical" memory** addresses.
  - The **hypervisor** must provide the configuration for mapping **"guest-physical"** (a.k.a. **"host-virtual"**) **memory** addresses to **"host-physical"** (a.k.a. **"real"**) **memory** addresses.



Memory virtualization (3/4)

- **Case 1: Without hardware support for memory virtualization (but assuming trap & emulate) – Using "shadow" page tables**
  - Note: **This approach is applicable to hypervisors with or without hardware support for CPU virtualization** (with some differences in the implementation details). Here, we just describe the basic, general idea.
  - Every attempt to modify the MMU performed by the guest kernel triggers a trap to the hypervisor.
  - The guest kernel has the illusion that it controls the real MMU.
  - The hypervisor understands and keeps track of the MMU configuration that the guest kernel wants to set up (page table of the current guest application, hereafter named **Guest-PT**).
  - The hypervisor maintains its own page table for each guest VM (hereafter named **Hyp-PT**).
  - In the real MMU (controlled by the hypervisor), there is no hardware support for performing multi-level address translations (because the MMU is not aware of the notion of "guest VM").
  - Therefore, the hypervisor cannot configure the MMU to directly apply the page table configuration provided by the guest kernel.
  - Instead, the hypervisor builds a new PT (hereafter named **Shadow-PT**) to configure the real MMU. **The Shadow-PT merges the information of the Guest-PT and Hyp-PT in order to directly provide the mappings between "guest-virtual" and "host-physical" addresses.**

Memory virtualization (4/4)

- **Case 2: With hardware support for memory virtualization – Using "extended" page tables**
  - Note: **This approach is only applicable to hypervisors with hardware support for CPU virtualization.**
  - In this approach, **the MMU hardware is modified/extended to introduce two levels of translation**, i.e., two kinds of page tables.
  - **The guest kernel has direct control over the real MMU to configure the Guest-PT**, i.e., page table mappings for translations between the guest-virtual addresses and the guest-physical addresses (like in a traditional system). **No hypervisor intervention/tracking is required** for this Guest-PT.
  - The hypervisor controls the real MMU to set up the Hyp-PT, i.e., page table mappings for translations between the guest-physical addresses and the host-physical addresses
  - Unlike the previous approach, **there is no need for a Shadow-PT**.
  - Whenever necessary, **the MMU hardware automatically performs the 2-level PT "walk"** (in Guest-PT, then Hyp-PT) in order to translate a guest-virtual address into
  - In this design, the Hyp-PT is often called **"Extended PT (EPT)"** or **"Nested PT (NPT)"**.

Nested virtualization (1/2)

- **What is it ?**
  - Consists in running a guest VM “within” another guest VM.
  - In other words, allows running a guest VM (“L2” layer) managed by a guest hypervisor (“L1” layer), itself managed by a host hypervisor (“L0”) running on the physical machine.
  - The L0 and L1 hypervisors may or may not be the same

- Actually **useful in various realistic scenarios**, such as:
  - Development, testing, and debugging of hypervisors
  - Deployment of a custom virtualization stack (controlled by the tenant) on top of a public IaaS platform (controlled by the cloud provider)

Nested virtualization (2/2)

- **Possible, but with some caveats**
  - Most of the current hardware platforms provide limited or no built-in support for nesting.
  - Nesting introduces **significant overheads** but, for some setups, careful optimizations can mitigate the performance degradation (~within 10% of single-level virtualization).
  - Generally requires support from the L0 hypervisor (e.g., for emulation of the hardware virtualization features).
  - Co-design of L0 and L1 hypervisors can provide some performance improvements.
- Currently supported by some virtualization stacks

References & Acknowledgments

This course is partially based on the following resources:

- **For an overview of the topic:**
  - Andrew Tanenbaum & Herbert Bos. **Modern Operating Systems**. 4th edition (2014) or above. Pearson Education. **See chapter about "virtualization and the Cloud".**

- **Main textbook about system virtual machines:**
  - Edouard Bugnion, Jason Nieh, Dan Tsafrir. **Hardware and software support for virtualization.** Morgan & Claypool. 2017.

- Lecture by Gernot Heiser (UNSW Sidney, 2025) on Virtualization Principles. https://cgi.cse.unsw.edu.au/~cs9242/25/lectures/03b-vms.pdf