- Service announcement(s)
- Introduction
  - Virtual machine
  - Why a virtual machine
  - Terminology
  - Terminology and theorems
  - Hardware vs software virtualization
- Virtualization
  - Software based virtualization
    - Introduction
    - Trap and emulate
      - Shadow page tables
    - Intel processors
      - Problems with pure trap and emulate
  - Hardware-based virtualisation
    - Goals
    - Rationale
    - Example
      - intel VT-x
        - New instructions
        - Extended page tables
    - KVM
      - Introduction
      - Memory overcommitment
      - Balooning
      - Kernel same-page merging
      - Qemu
        - Emulating devices normally
        - Emulating devices with VIRTIO
    - Xen Paravirtualization
      - Introduction
      - The Xen hypervisor
- Containerization
  - Introduction
  - Namespaces

POLITECNICO
MILANO 1863

System virtualisation

Advanced Operating Systems

Vittorio Zaccaria | Politecnico di Milano | '25/26

Scan below QR code for course website

Service announcement(s)

Seminar:

- Thursday, 27 November 2025 at 18:00. **"Secure boot and secure storage"** seminar from Security Pattern
- Seminar will be held online in my Webex room
- Participation in the seminars is automatically verified through the institutional Webex account.
- Attendance at each seminar is awarded 0.5 points.
- The evaluation points earned during seminars and, if applicable, lectures do not contribute to passing; they are only added if the scores for Part A and Part B are both sufficient.

Strike:

- Friday November 28th there will be a National General Strike
- I suggest moving our lecture **entirely online** in my Webex room

POLITECNICO
MILANO 1863

Introduction

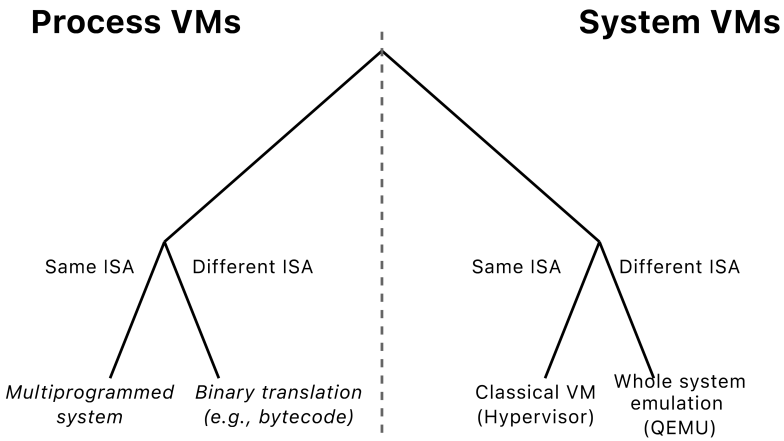
Virtual machine

Definition (Popek and Goldberg, 1974):

- An **efficient, isolated duplicate of the real system** that can run a commodity OS (system-level virtualisation).
- Based on a **virtual machine monitor** that **relies solely on direct execution**

The definition boils down to the following requirements

- **Fidelity**: equivalence of behavior with the real machine
- **Safety**: the virtual machine cannot override the VMM's control of virtualized resources
- **Efficiency**: programs should "show at worst only minor decreases in speed"



Why a virtual machine

- Consolidate and partition hardware
  - Use one physical machine at 100% instead of two at 50% (consolidation)
  - Fewer bigger machines
- React to variable workloads
  - Reduced hardware and administration costs for data-centers
  - Horizontal scalability
- Standardized infrastructure
  - Simplifies software distribution for complex environments
  - Isolated network and storage
- Security sandboxing, fault tolerance (checkpointing and rollback)

Terminology

Host system: the OS where virtual machines run

Guest system: the OS that runs on top of the virtual machine

Virtual machine monitor (VMM) or **Hypervisor**:

- Software program that translates/mediates access to physical resources such as interrupts or sensitive processor state
- Ensures isolation

Type 1 Hypervisor: Also called *native* hypervisor; runs on bare metal without any OS abstraction

Type 2 Hypervisor: Runs in the context of another OS (think about KVM or VirtualBox).

Terminology and theorems

Instruction types: Unprivileged and privileged. The latter are those that trap in user mode.

Virtualization idea: Run privileged instructions in a de-privileged mode.

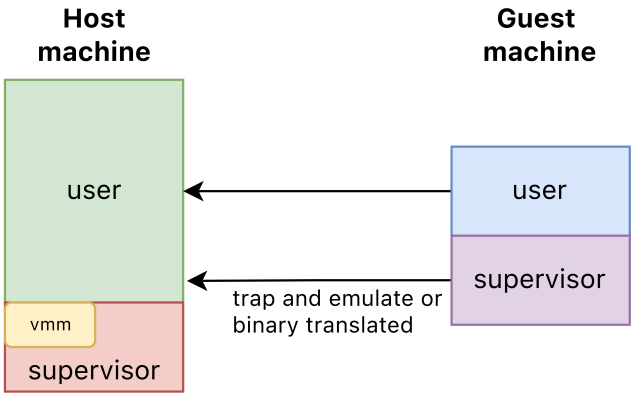
Instruction sensitivity An instruction is virtualization-sensitive if it is:

- Control sensitive: it modifies directly the machine state (e.g., enabling or disabling interrupts, modifying the interrupt vector table)
- Behavior sensitive: instructions that behave differently when used in either user or supervisor mode. It might affect **fidelity**.

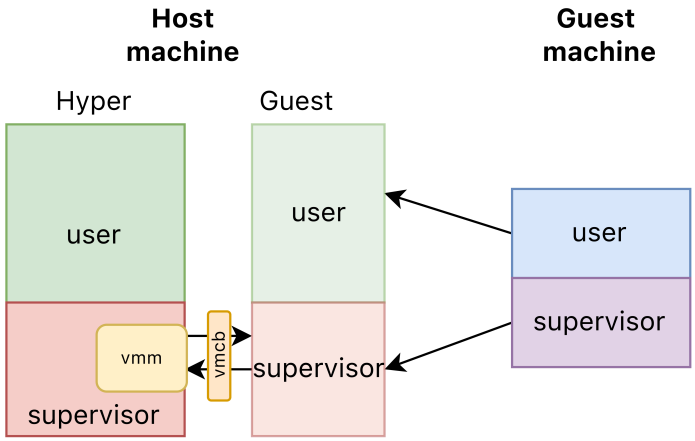
Theorem (Popek and Goldberg) For any conventional computer, a virtual machine monitor may be built if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

Hardware vs software virtualization

Software based virtualization



Hardware based virtualization



Virtualization

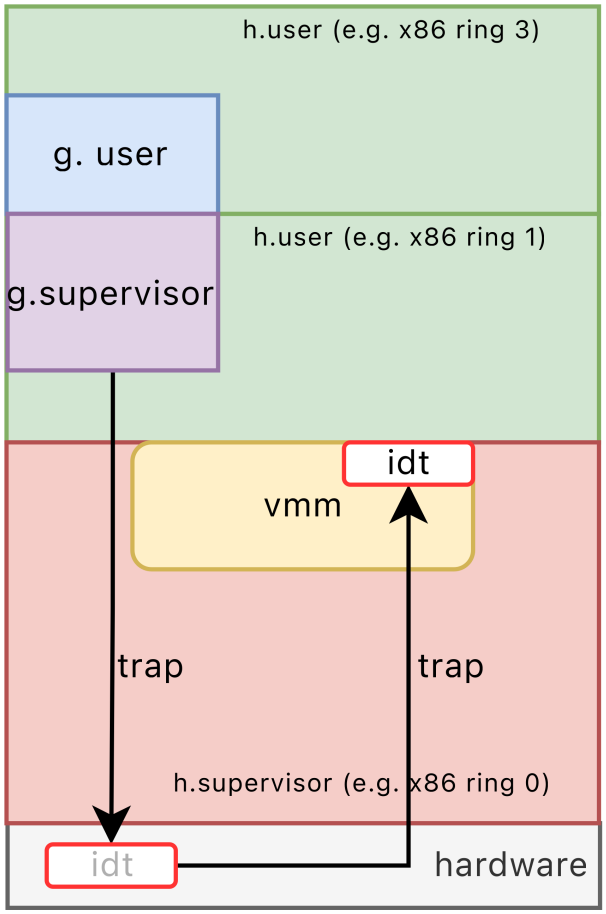
POLITECNICO
MILANO 1863

Software based virtualization

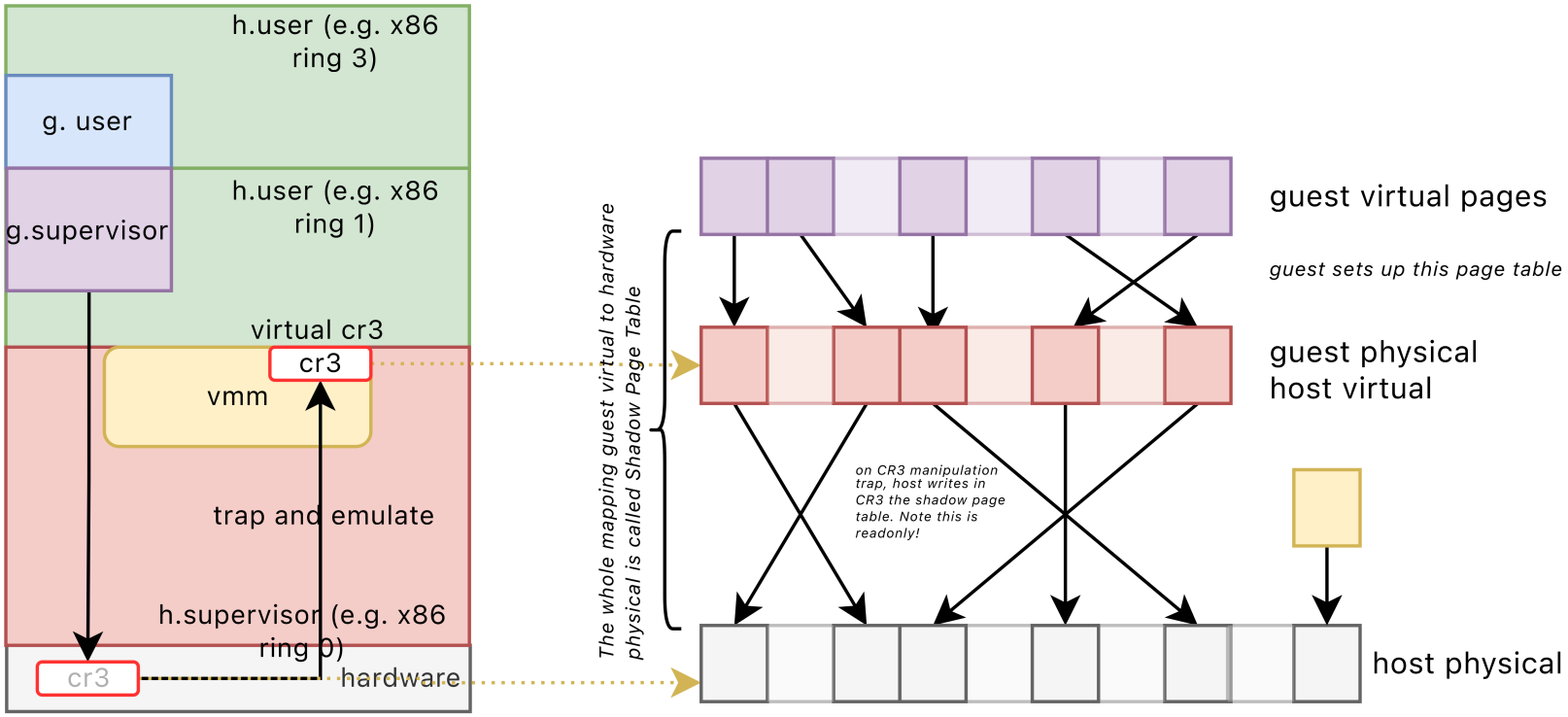
Introduction

Software-based virtualization means **deprivileging**:

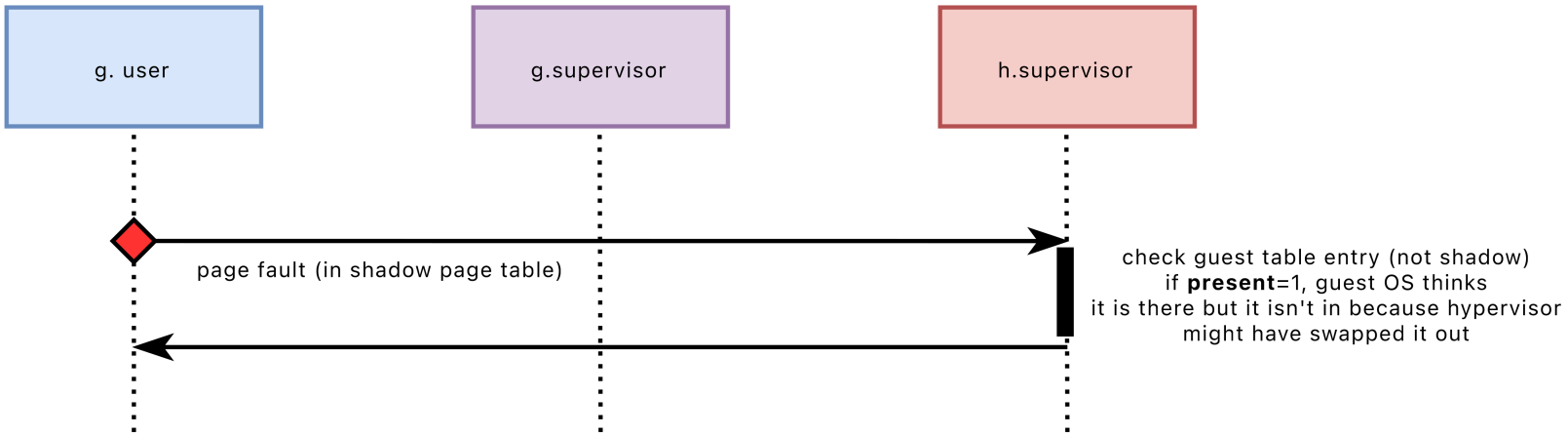
- `g.supervisor` is translated into `h.user`
- privileged instruction or memory access produces an interceptable trap
- `h.supervisor` installs its own structures nstead of those dictated by `g.supervisor`.



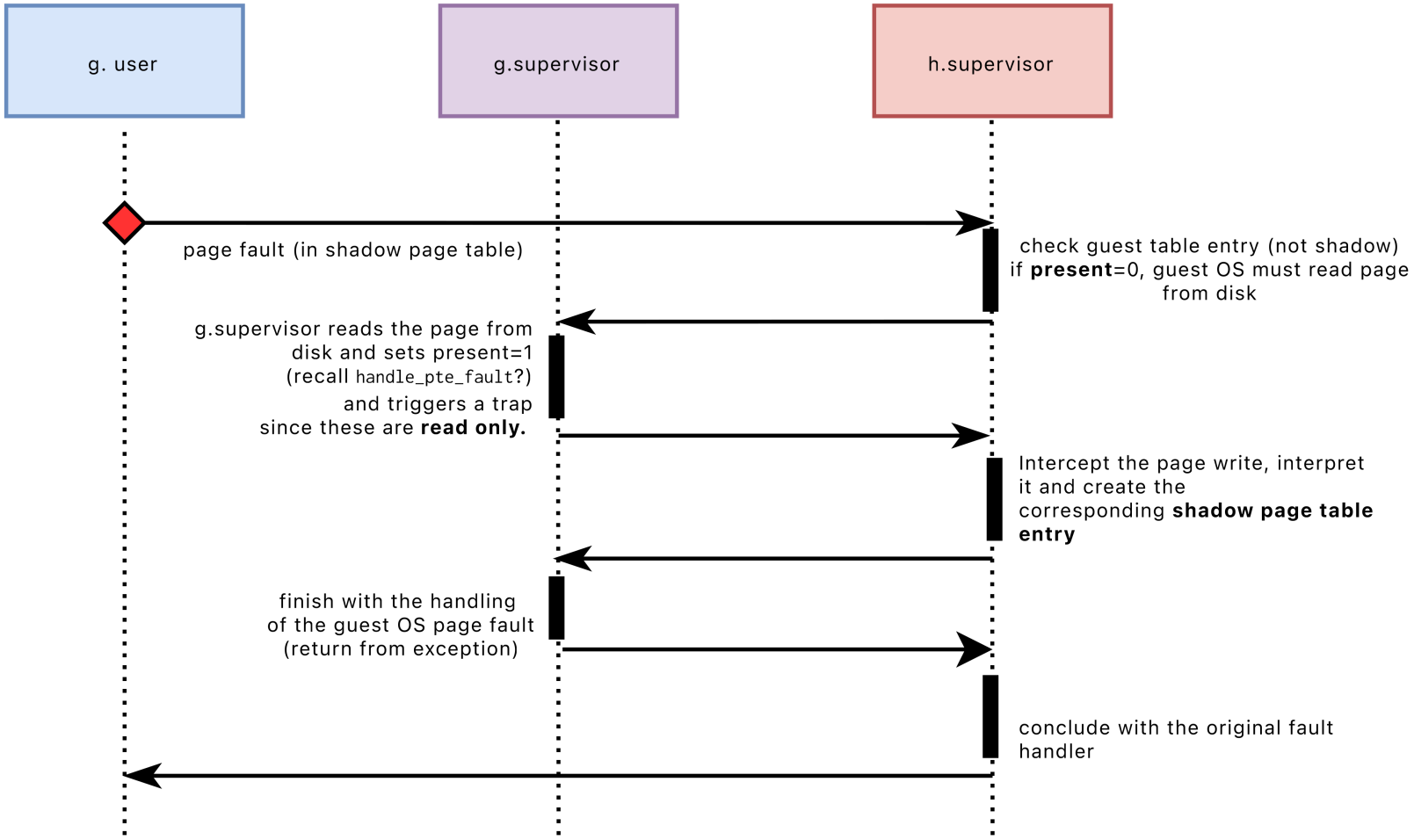
Trap and emulate » Shadow page tables



Trap and emulate » Shadow page tables



Trap and emulate » Shadow page tables



Software based virtualization » Intel processors

Problems with pure trap and emulate

Virtualization sensitive instructions Originally, intel had unprivileged virtualisation sensitive instructions:

- Instructions manipulating the interrupt flags (`pushf` , `popf` ..) → either creating confusion in `g.supervisor` or `h.supervisor` cannot track the state of interrupts correctly.
- Reading and writing segment descriptors and registers (`pop seg` , `push seg` , `mov seg` , `sgdt`) → `g.supervisor` can see that it has been deprivileged by reading the Current Privilege Level and/or VMM state.

Problems with pure trap and emulate

Excessive faulting On x86-32, `sysenter` and `sysexit` are used for each system call but trap into the VMM any time they are executed by the guest OS.

Several operation modes (real mode, protected mode, v8086 mode, and system management mode) mean higher complexity.

Indeed, before the introduction of VMware, engineers from Intel Corporation were convinced their processors could not be virtualized in any practical sense.

Hardware-based virtualisation

Goals

Main goals

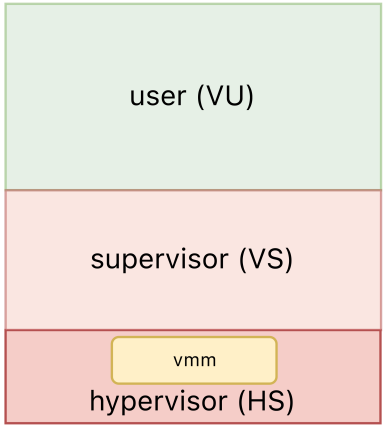
- Avoid the problems of deprivileging, i.e., g.supervisor works as a in a traditional supervisor (no ring aliasing), by adding new modes
- Allow the state of the guest can be explicitly and comprehensively saved and resumed and which is used for some shadow structure

Secondary goals:

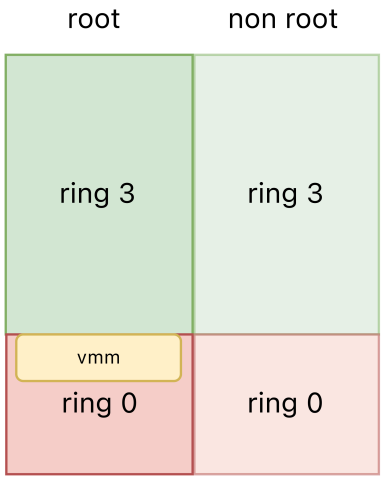
- On x86 allow finally to obey Obey Popek-Goldberg requirements
- Improve performance
  - Reduce # of traps from system calls and interrupts
  - Avoid shadow paging overhead

Rationale

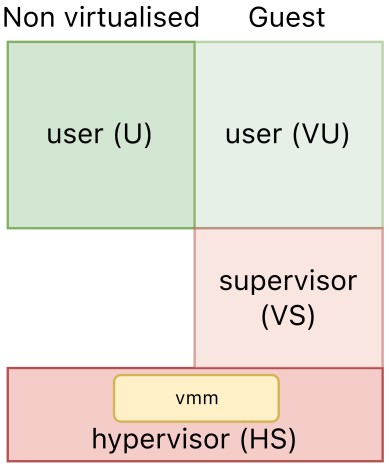
PowerPC/ARM



X86 - VT-x



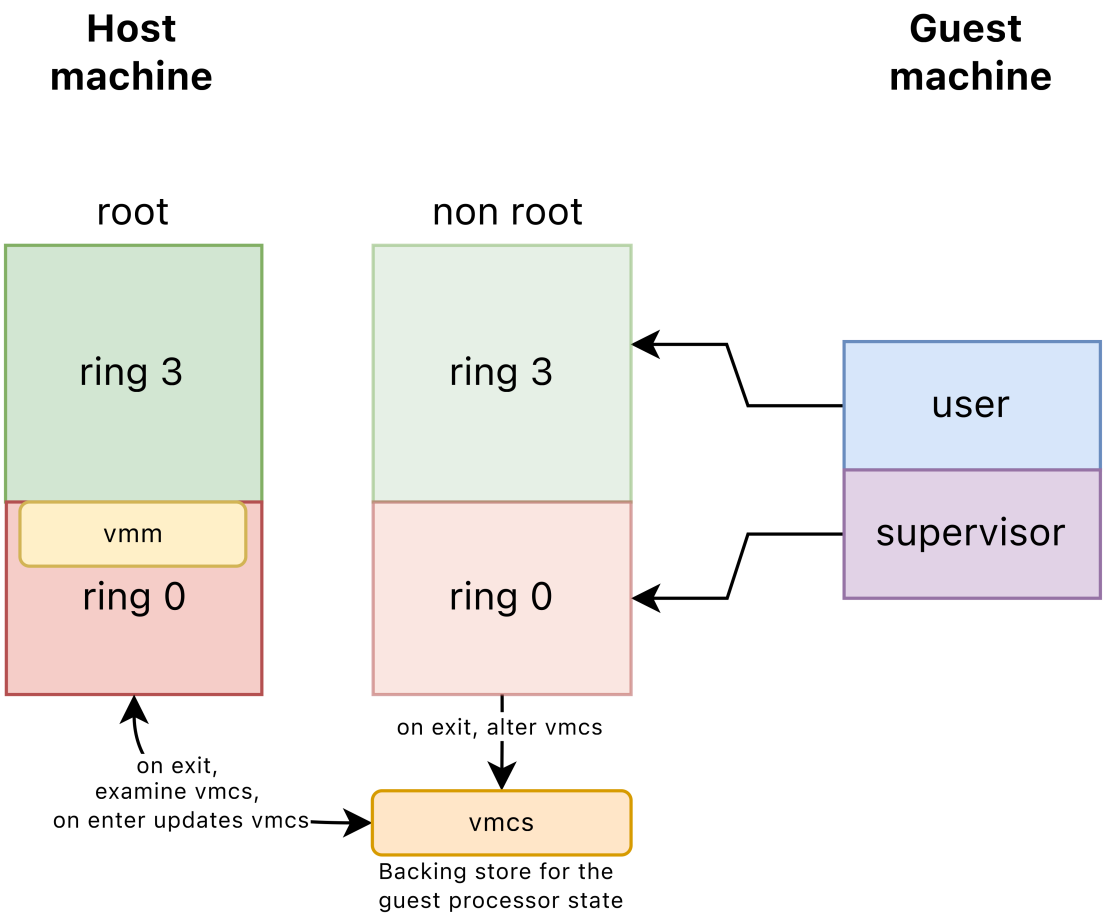
RISCV



Alternative implementations:

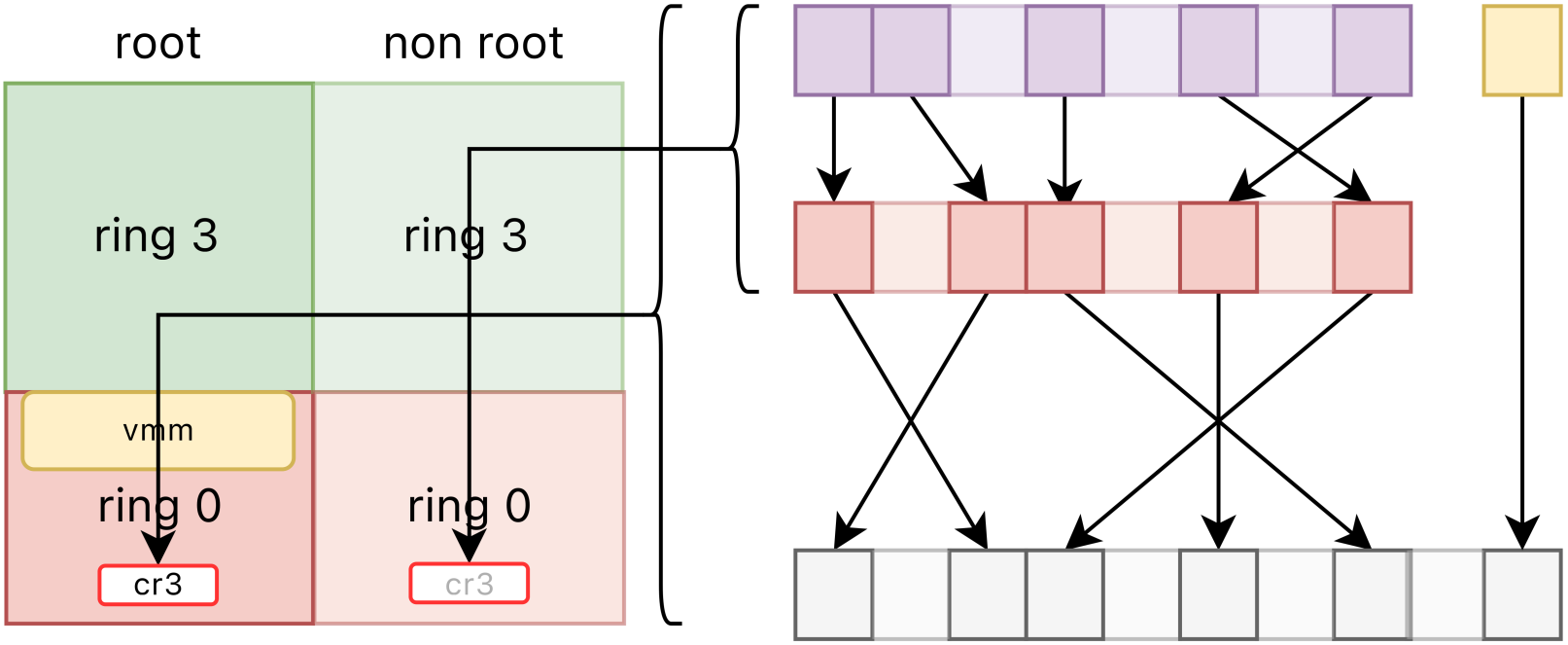
- A third mode in addition to user/supervisor (PowerPC, ARM)
- A second dimension for virtual machines vs hypervisors (x86, s390, RISC-V)

Example » intel VT-x » New instructions

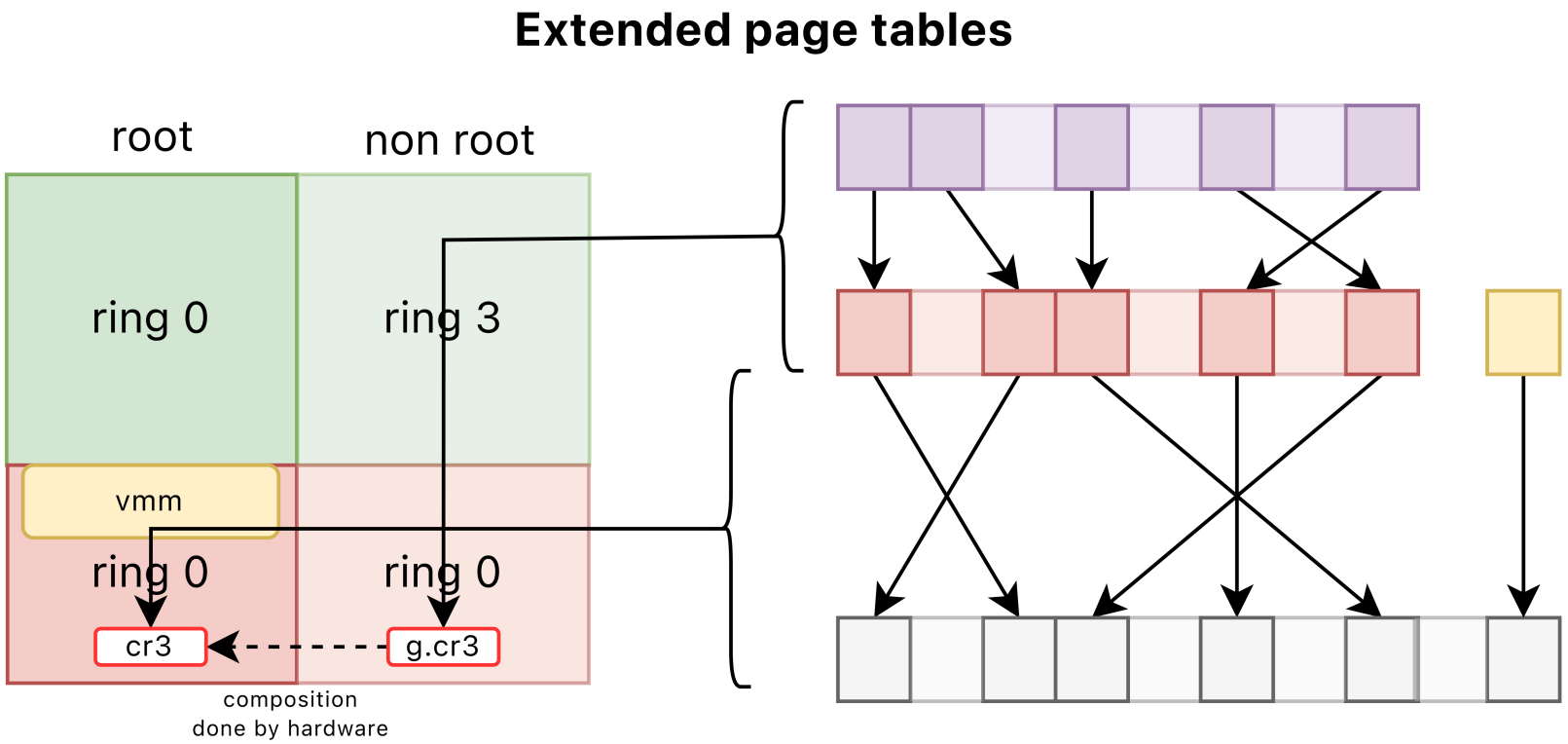


Example » intel VT-x » Extended page tables

Traditional shadow page tables



Example » intel VT-x » Extended page tables



Hardware-based virtualisation » KVM

Introduction

- KVM (Kernel-based Virtual Machine) is an open source virtualization technology built into the Linux kernel that allows Linux to function as a hypervisor.

- KVM converts Linux into a type 1 hypervisor, with Linux providing operating system components like memory management, scheduling, I/O, etc. that are needed to run VMs. VMs run like threads.

- To use KVM, you need a version of Linux from 2007 or later that supports hardware virtualization on an x86 processor.

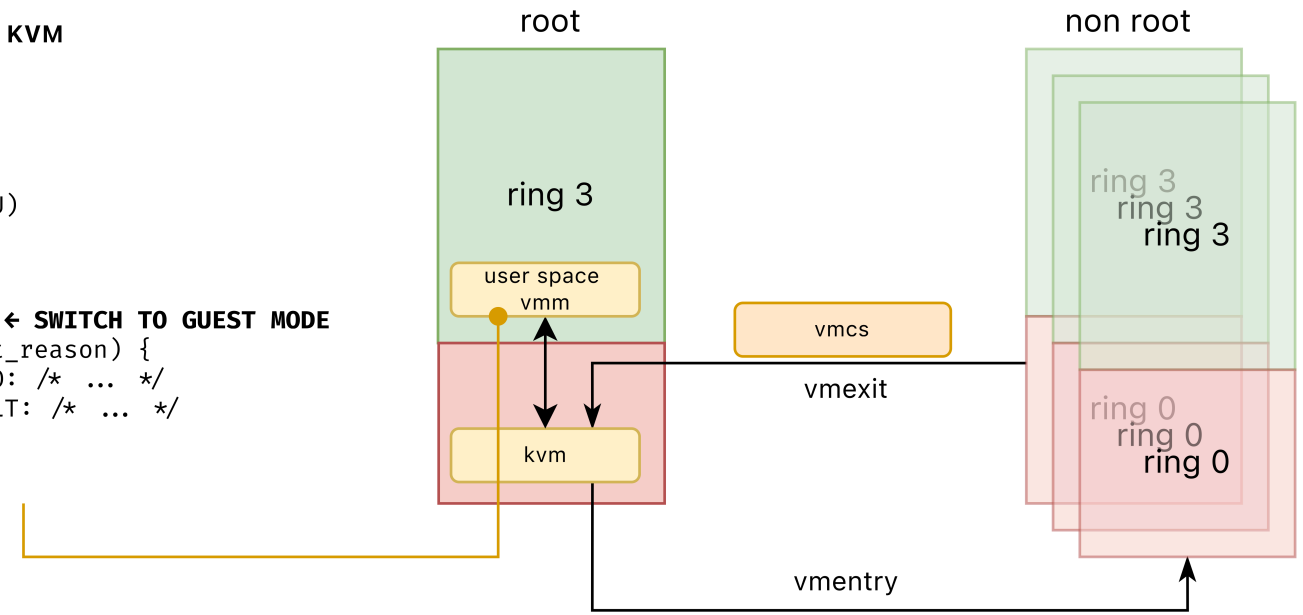
Introduction

Very minimal VMM using KVM

```
// HOST MODE
open("/dev/kvm")

ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)


for (;;) {
  ioctl(KVM_RUN); // ← SWITCH TO GUEST MODE
  switch (vmcs → exit_reason) {
     case KVM_EXIT_IO: /* ... */
     case KVM_EXIT_HLT: /* ... */
  }
}
```



Memory overcommitment

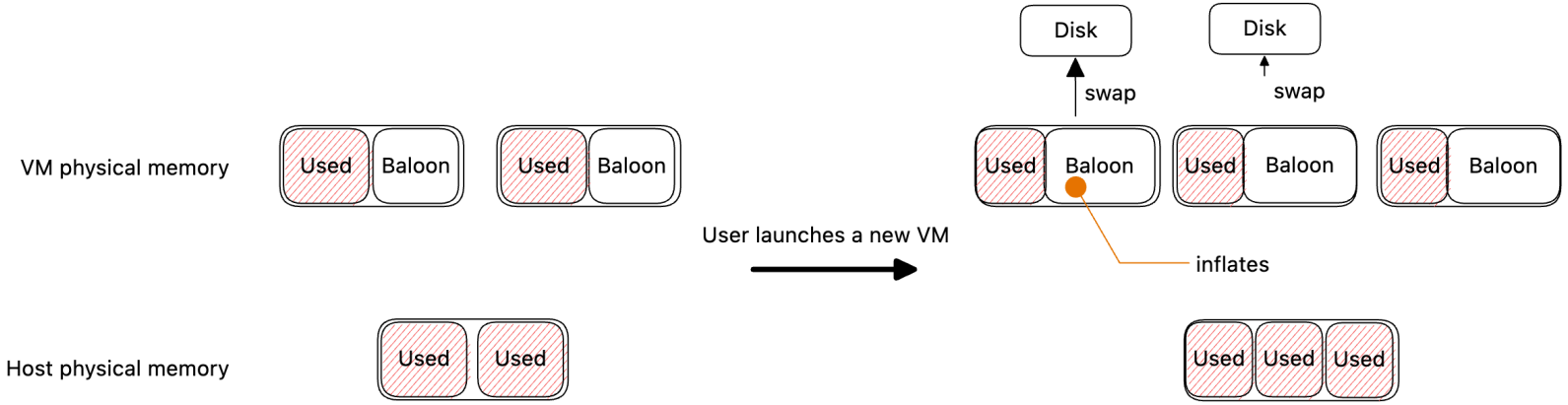
- Most of VMs do not use 100% memory available all the time, so we could allocate VMs memory more than the total physical memory that hosts have.

> If your dedicated server has 15GB memory, without memory overcommit, you can create 15 VPS of 1GB each. In this scenario if all the VMs are only utilizing 200MB of 1 GB assigned, then all the remaining 800MB from 15 VMs will not be utilized and left in the server. Instead of wasting 800 $\times$ 15 = ~11GB, you can create 11 new VMs with that remaining space.

- As KVM VMs are processes, Linux virtual memory allows by default to have each process consume all of its address space, so you can do overcommitment by default.

Ballooning

- Ballooning is a way to enforce that some of your VMs actually stay with their usage of physical memory under a certain threshold.
- This is achieved by a driver that reserves a certain amount of physical pages within the VM, forcing swapping. Which memory pages are given back is the decision of the guest operating system (OS).



Kernel same-page merging

> AKA page deduplication

- Goal is to share memory pages that have identical contents between multiple processes or virtualized guests.
- Scans for physical pages that have identical content, and identifies the virtual pages that are mapped to those physical pages.
  - It leaves one page unchanged,
  - Re-maps each duplicate page to point to the same physical page
  - Releases the extra physical pages for re-use.
  - It also marks both virtual pages as copy on write
- KSM was originally developed for use with KVM but it can be useful to any application which generates many instances of the same data.
- Comparing pages can consume lot of computing power. Be careful when using this.

Qemu

Writing a VMM, even with KVM, is not an easy feat. You'd have to handle all the interaction with the hardware, i.e.:

Emulated devices: devices that do not happen to be attached to your host: a virtual disk, a virtual network card etc.

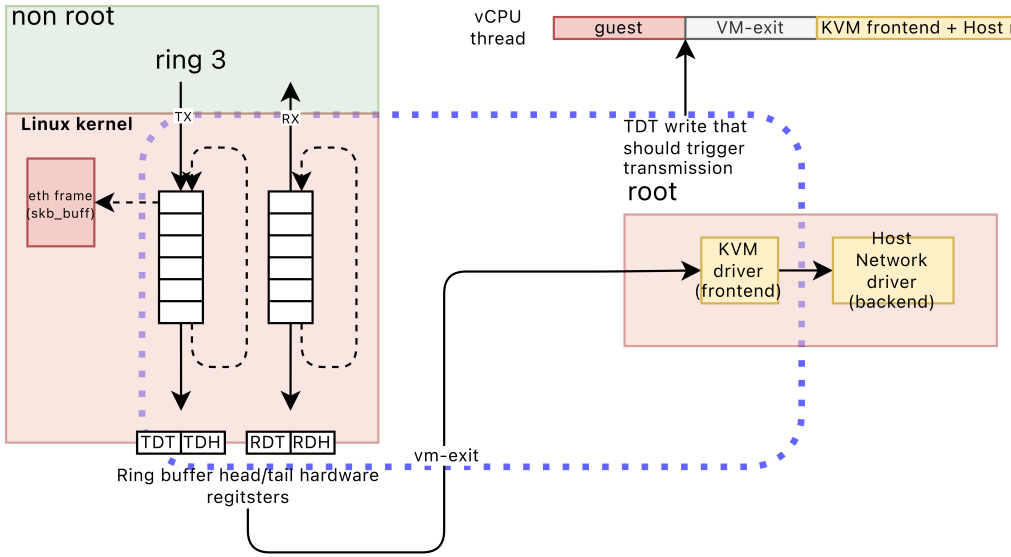
Hardware passthrough: actual hardware attached to your host that you must somehow expose to the virtual machine, i.e., a physical GPU or a physical network card

QEMU can be used as a VMM based on kvm if you use the `--enable-kvm` switch.

QEMU provides a solid ecosystem of existing "emulated" devices and can provide fast access to them thanks to the VIRTIO interface.

Qemu » Emulating devices normally

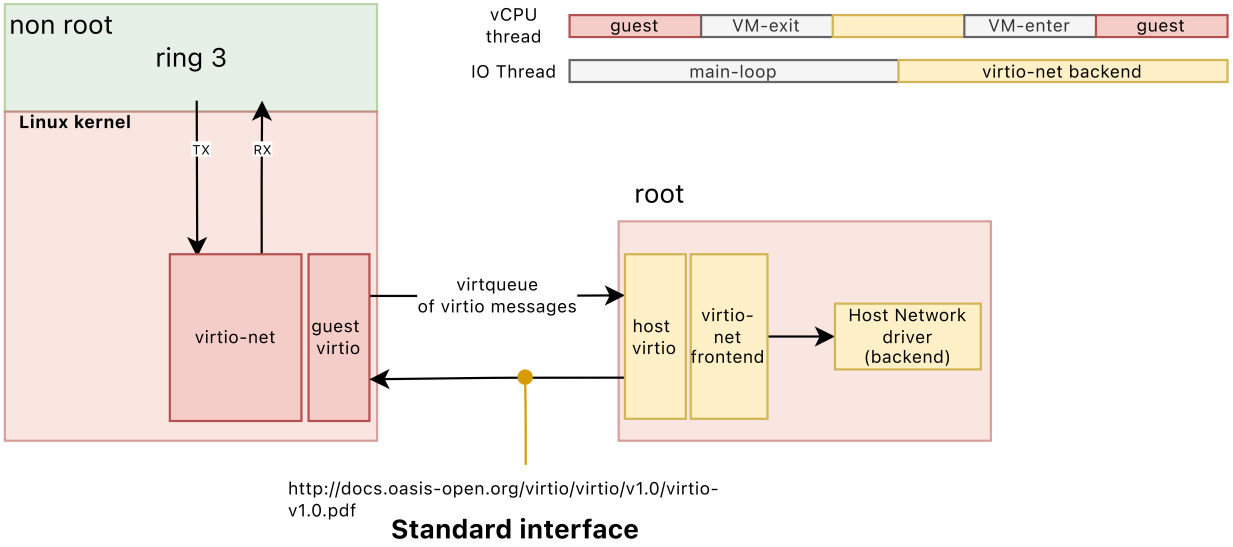
Emulation of the network interface controller



Intel e1000 family, 82540EM model is commonly emulated by hypervisors (VMWare, VirtualBox, QEMU)

Qemu » Emulating devices with VIRTIO

Emulation of the network interface controller (paravirtualised)



Hardware-based virtualisation » Xen Paravirtualization

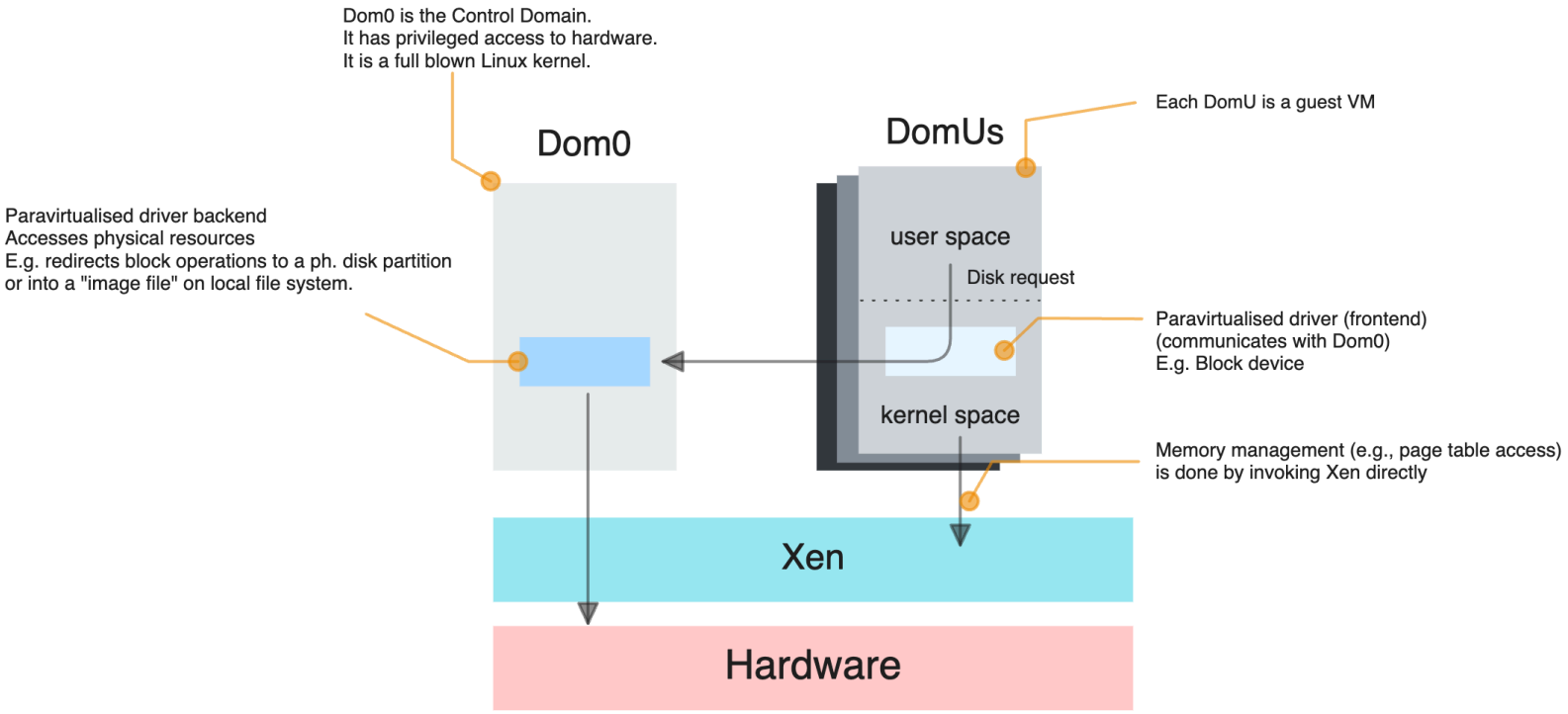
Introduction

Main idea What if we could change the target guest and replace the difficult-to-virtualize instructions with something else?

How to do it

- Rewriting all the target software from scratch is out of the question
- However, porting a well designed kernel to a new architecture is a matter of replacing the system-specific routines and recompiling the rest.
  - E.g., instead of accessing the MMU the paravirtual kernel could call the VMM for write access to page tables
- This technique was first introduced in the XEN hypervisor but nowadays it is mostly used within drivers in the guest;

The Xen hypervisor



Xen is a Type I hypervisor and requires guest OSes (User domains - DomU) to be modified to ① execute in ring 1 and ② substitute sensitive instructions with calls to the Xen kernel (*hypercalls*); **user space code runs unmodified**. A special privileged guest (called Dom0) runs Linux with all its drivers and allows Xen to manage device access from DomUs.

Containerization

Introduction

- Containers are a way to **isolate a set of processes** and make them think that they are the only ones running on the machine.
- The machine they see may feature only a subset of the resources actually available on the entire machine (e.g., less memory, less disk space, less CPUs, less network bandwidth).
- Containers **are not virtual machines**
  - Processes running inside a container are **normal processes running on the host kernel**.
  - There is **no guest kernel** running inside the container
  - You cannot run an arbitrary operating system in a container, since the kernel is shared with the host (Linux, in our case).

A part from cgroups, the mechanism that allows containers to work on Linux is **namespaces**.

Namespaces

A namespace changes and reduces the set of resources that processes belonging to it can access/manipulate. The PID is one of these resources.