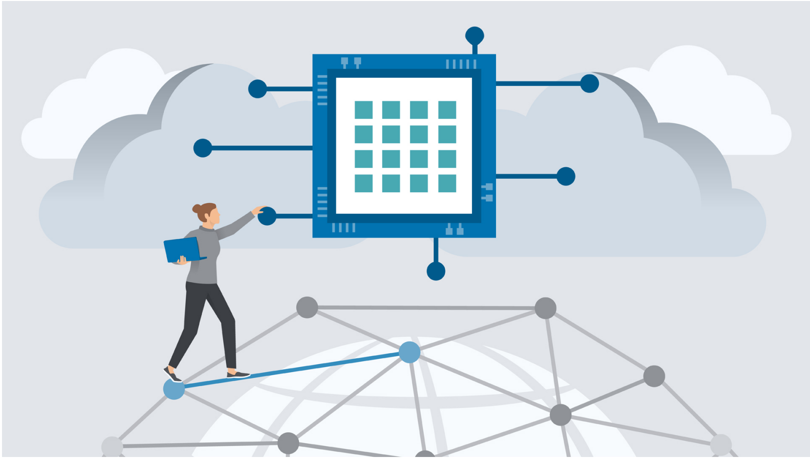
Hardware Virtualization



Mathieu Bacou
mathieu.bacou@telecom-sudparis.eu
Télécom SudParis, IMT, IP Paris, Inria

What is virtualization?

- Abstraction of **physical resources into virtual resources**
  - More complex management: sharing, access rights
  - Unified hardware access: easier development
- Many kinds:
  - Operating systems: virtual memory, threads…
    - Microsoft Windows, Linux, Mac OSX, BSDs, Android…
  - Emulators: instruction translation
  - Language virtual machines: optimized emulator
    - Java Virtual Machine (JVM), Python…
  - Containers: virtual OS
    - Docker, LXC…
  - Virtual machines: virtual hardware
    - QEMU/KVM, Xen, VMWare ESXi, VirtualBox, Microsoft Hyper-V…

What is hardware virtualization?

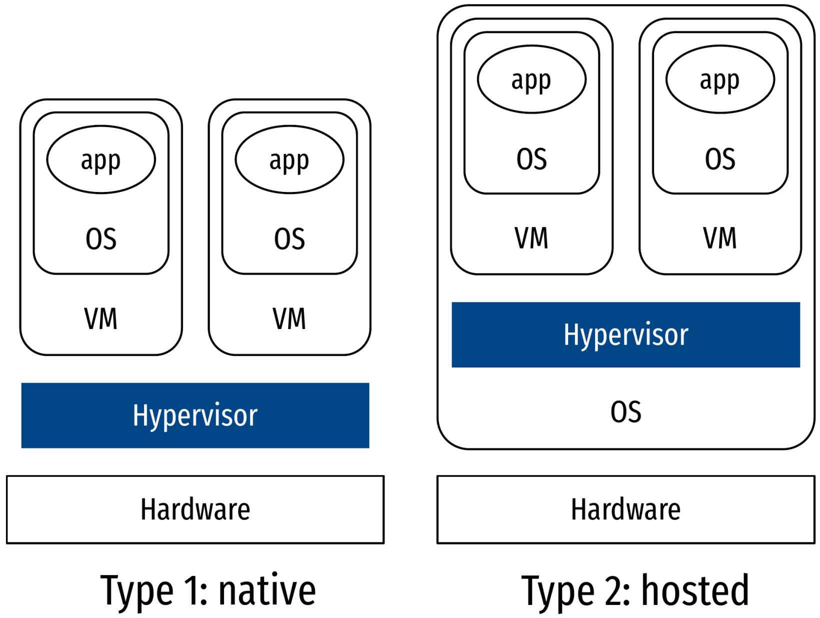
- Virtualize hardware for multiple OSes at the same time!
  - Virtual CPUs
  - Additional level of memory addressing
  - Virtual storage
  - Virtual network
  - IRQs, clocks…
- A **hypervisor** runs **guest OSes** in **virtual machines**

Actors of hardware virtualization

1. Hypervisor
2. Virtual machine and guest OS
3. User interface

Hypervisor

- A hypervisor (HV) is a special OS that runs guest OSes
  - Manages virtual machines (VM) where guest OSes are run: also called **virtual machine manager** (VMM)
- Two types:
  - **Type 1: native**
    - Bare metal
    - Guest OSes are processes
  - **Type 2: hosted**
    - Process of a normal OS
    - Guest OSes are subprocesses



Types of hypervisors.

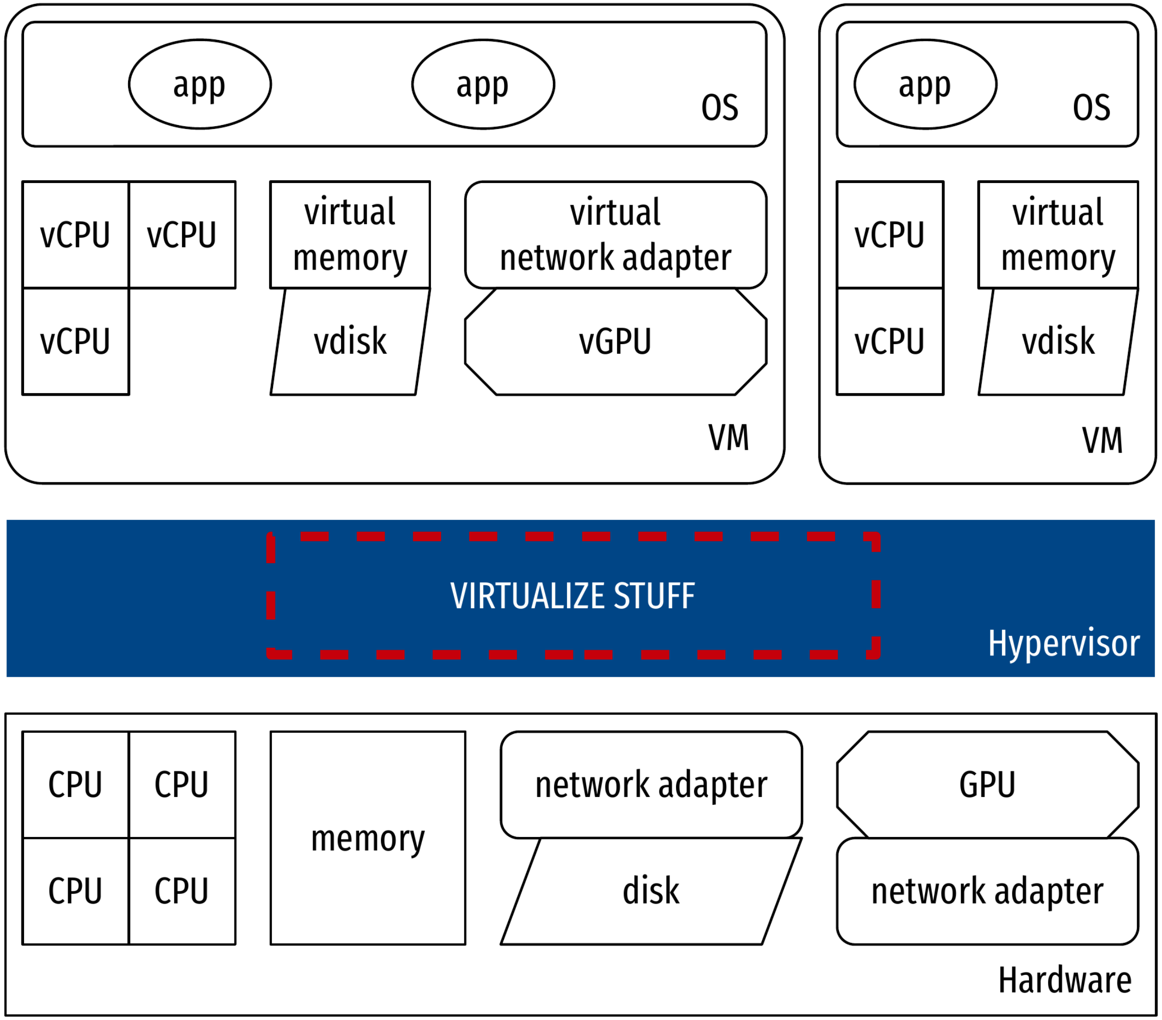
Hypervisors in the cloud

- Type 1 (Xen, KVM…):
  - Optimized for **maximum resource virtualization**
    - Bare metal
  - Low performance overhead
    - Only one (big) task: run guest OSes
  - More secure
    - Isolation of guest OSes at lower level
- Type 2 (VirtualBox, QEMU/KVM…):
  - Easier to install and use
- For these reasons, **the cloud relies on type 1** rather than type 2
  - *But also operating system-level virtualization (next chapter)*

Virtual machine

- Cohesive ensemble of virtualized resources that represent a complete machine
  - Hardware is virtualized: a **guest OS** is still needed!
- Status: running, suspended, shut down
- When running:
  - **State of virtual hardware**
    - Memory, I/O queues, processor registers and flags…
    - “Easy” checkpointing with snapshots
- When stopped:
  - A **disk image**
    - Files of guest OS
    - Easy replication by copying disk image

Virtual machine: the stack



Stack of a virtual machine.

User-interface

- Use hypervisor’s features to let a user manage VMs and related resources
  - Examples: VirtualBox, QEMU’s CLI, virsh, virtmanager…
- GUIs, TUIs
  - Graphical display emulation for desktop environments in VMs, etc.

Demo: QEMU/KVM

- Creation and usage of a QEMU/KVM VM:
  - Run a guest OS in a VM booting from “CD-ROM”
  - Run the installed OS booting from overlayed disk image in a fully-featured VM
  - Run the same OS in a weaker VM
- QEMU is a bit hard to use: prefer libvirt for VM management and configuration

QEMU logo.

KVM logo.

libvirt logo.

User-friendly interface: libvirt

- Common and stable layer to manage VMs
  - Works with many hypervisors
  - Also manages storage and network
- Used by user front-ends: virsh, virtmanager…
  - **Clients** to libvirtd daemon

Commands and concepts

- Interactive shell: `virsh`
  - Help: `virsh help`
- Managing VM images:
  - Manage **storage pools** (collections of VM images):
    - `virsh pool-` commands family
  - Manage **volumes** (VM images) in storage pools
    - `virsh vol-` commands family
  - **Abstraction of VM images** to manage them across the cloud
    - *Useful for migration, replication, etc.*
- Managing **domains** (specifications of VM guests)
  - High-level command to install guests: `virt-install`
  - Manually edit a defined domain: `virsh edit`
- Administrating domains:
  - Start: `virsh start`
  - End: `virsh destroy`
    - **Force-stops the domain** (think “pulling the plug”!)
    - `virsh shutdown` to demand shutdown gracefully as from (virtual) hardware
- Accessing domains:
  - Get a TTY console: `virsh console`
  - Connect to display: `virt-viewer`

Virtualization in the cloud

1. Life-cycle
2. Scalability
3. Resource management
4. Security and reliability

Life-cycle of VMs in the cloud

- Easy **deployment**: one VM image, multiple VMs–services
- Easy **administration**: all software, no hardware
- Seen as a resource unit in the cloud
  - **Accounting** based on VM size and uptime

Zurich (europe-west6) Par mois Par heure

Type de machine	Processeurs virtuels	Mémoire	Prix (USD)	Prix des VM préemptives (USD)
n1-standard-1	1	3.75GB	$0.0665	$0.01400
n1-standard-2	2	7.5GB	$0.1329	$0.02800
n1-standard-4	4	15GB	$0.2658	$0.05600
n1-standard-8	8	30GB	$0.5317	$0.11210
n1-standard-16	16	60GB	$1.0634	$0.22420
n1-standard-32	32	120GB	$2.1268	$0.44840
n1-standard-64	64	240GB	$4.2535	$0.89670
n1-standard-96	96	360GB	$6.3803	$1.3451

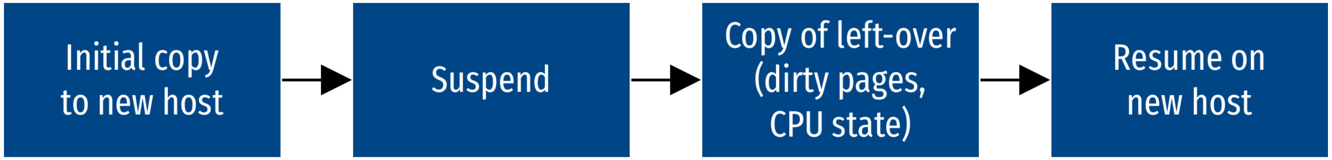
Excerpt of Google Cloud Platform pricing for generic VMs of the Compute Engine (Nov. 2020).

Scalability

- **Horizontal**: add VMs
  - Under load spikes, replicate the service
    - Kill useless replicates after burst
  - **Load balance** between replications
  - Often with automatic scaling
- **Vertical**: enlarge VMs
  - All hardware is virtual: dynamic addition of vCPUs or memory
- Hard to implement: how to reclaim unused memory from the guest OS when downscaling?
- Also: shutdown and replace with stronger VM
  - Keep the same image!
  - Reconfigure applications

Resource management

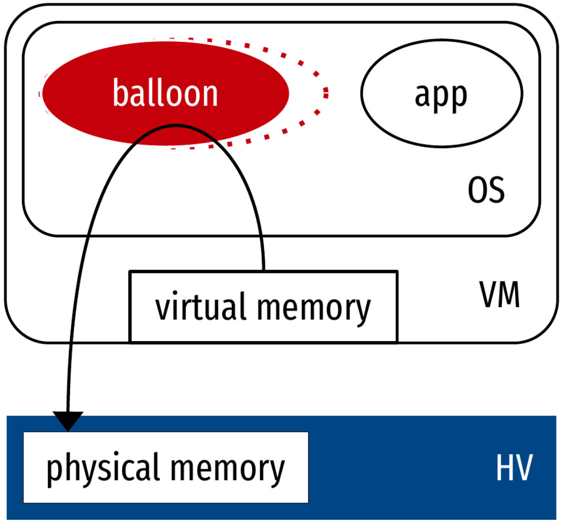
- Fit N VMs on M physical hosts
  - Many resources to take into consideration: memory, CPU, disk, network…
  - Hard optimization problem with many dimensions
- **Overcommitment**: resources are virtual, so give out more than physically available
  - Rarely, or very cautiously used: too harmful when it collapses
- **Migration**: VMs are loosely attached to hosts, so move them around
  - Migration allows **consolidation**
    - Optimize resource usage on physical hosts
    - Optimize datacenter usage by powering only needed hosts



Seamless live migration of a VM.

Resource management: memory

- Hard to manage: spatial sharing
  - You can't get more memory!
  - Different from CPU: time sharing, you can simply wait
- **Overcommitment**: resources are virtual, so give more memory than physically available
  - Rarely used: too harmful when it collapses because the system thrashes, swapping pages
- **Ballooning**: reclaim memory from guests
  1. Inflate: ask for memory pages
  2. Give the pages back to the HV
  - Paravirtualized mechanism
  - Rarely used: too hard to estimate balloon size
    - Too hard to estimate working set size
    - Too big makes the VM swap, destroys performance

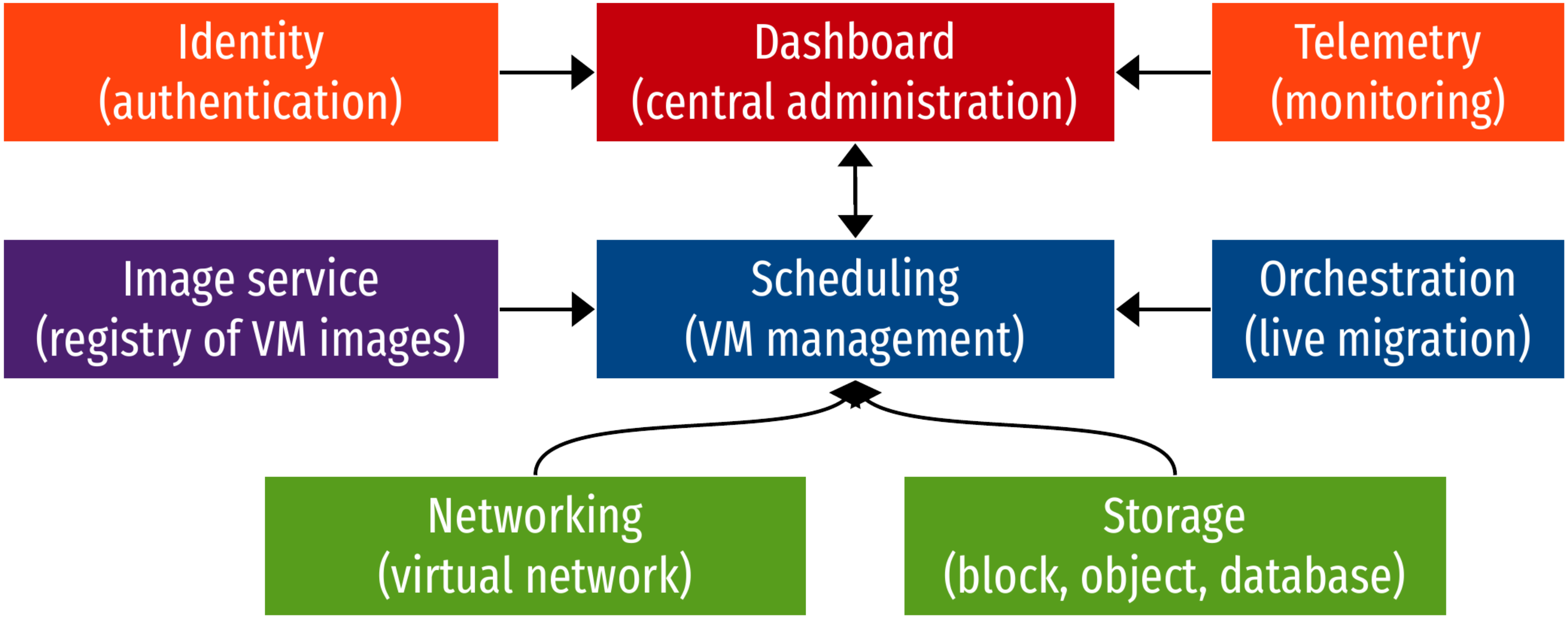


Paravirtualized ballooning.

Security and reliability

- **Isolation** between VMs
  - Different guest OSes, virtual hardware
  - Access policies enforced by the hypervisor
- Automatic **checkpointing** and resuming
  - Automatic failure handling
  - Redundancy

Cloud infrastructure: overview



Example of cloud infrastructure: OpenStack (simplified)

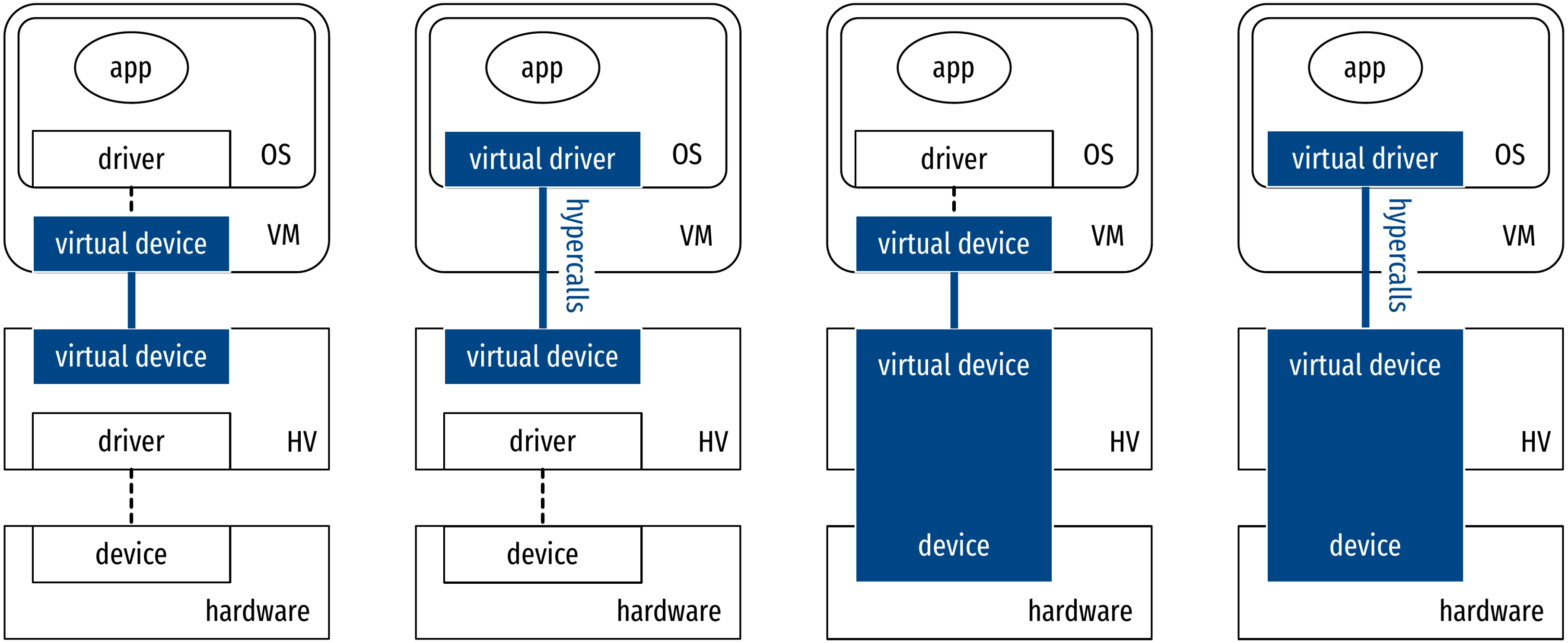
Internals of an hypervisor

1. Modes of virtualization
2. Architectural overview of QEMU/KVM
3. Virtualization of CPUs
4. Virtualization of memory
5. Virtualization of I/O and devices

Modes of virtualization

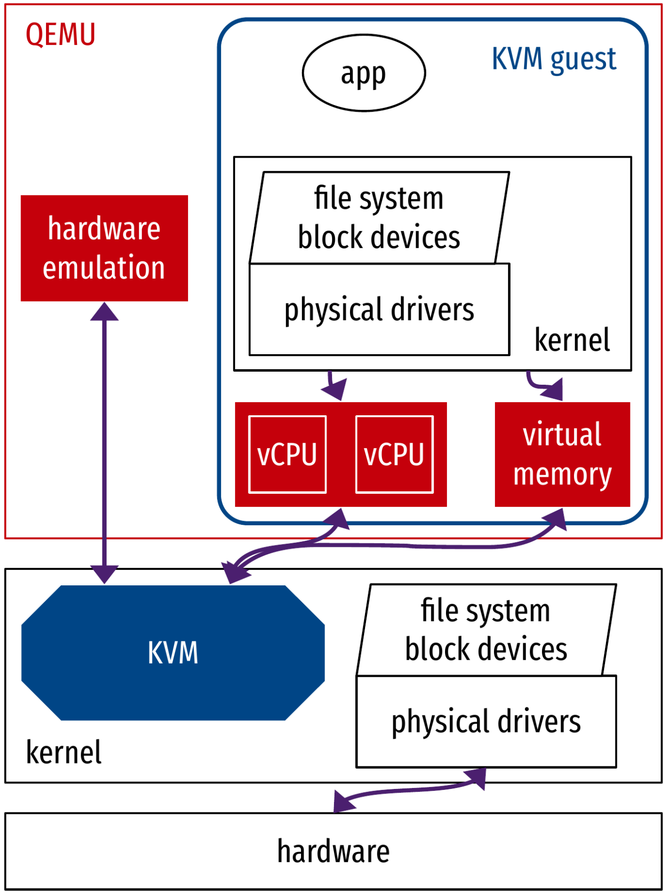
- Three modes to virtualize a guest OS:
  1. **Full virtualization**: total simulation of virtual hardware
     - Unmodified guest OS
     - Binary translation
  2. **Paravirtualization**: virtualization interface between guest OS and HV
     - Paravirtualized guest OS: deep changes, paravirtualized drivers
     - Software optimizations of guest OS * HV interaction: hypercalls
  3. **Hardware-assisted virtualization**: the physical hardware helps executing virtualized OS operations
     - Unmodified guest OS
     - Hardware support for virtualized execution (Intel VT-x, AMD-V…)
- Orthogonal to HV types

Modes of virtualization of a guest OS



Full virtualization. Paravirtualization. Hardware-assisted virt. Hardware-assisted virt.

Architectural overview of QEMU with KVM



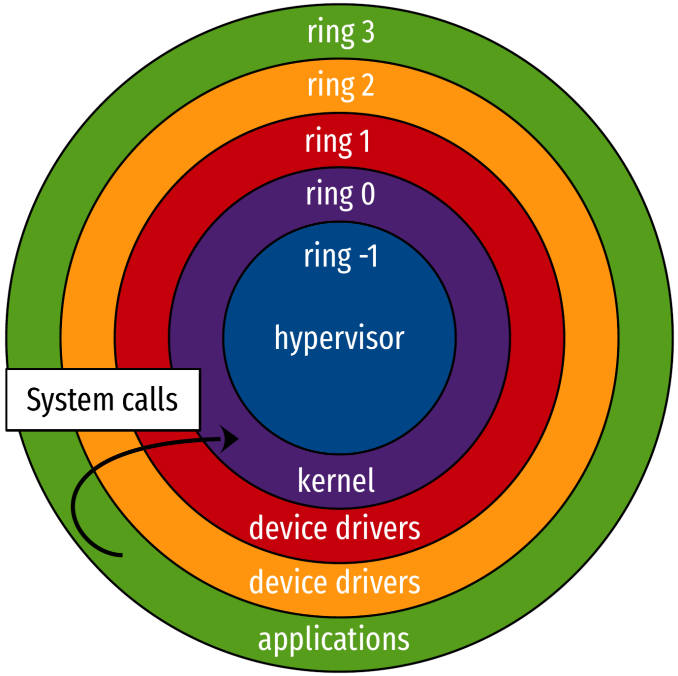
Architecture of QEMU when using KVM.

Virtualization of CPUs

- Problems: the guest OS has expectations
    1. **Unlimited control** over the hardware
        - But now it's the hypervisor!
    2. **Exclusive control** over the hardware
        - But now there are many OSes to share with!
- Effects:
    - Changes in protection rings to de-privilege guest OS
    - VM context switching to share hardware among guests

CPU protection rings

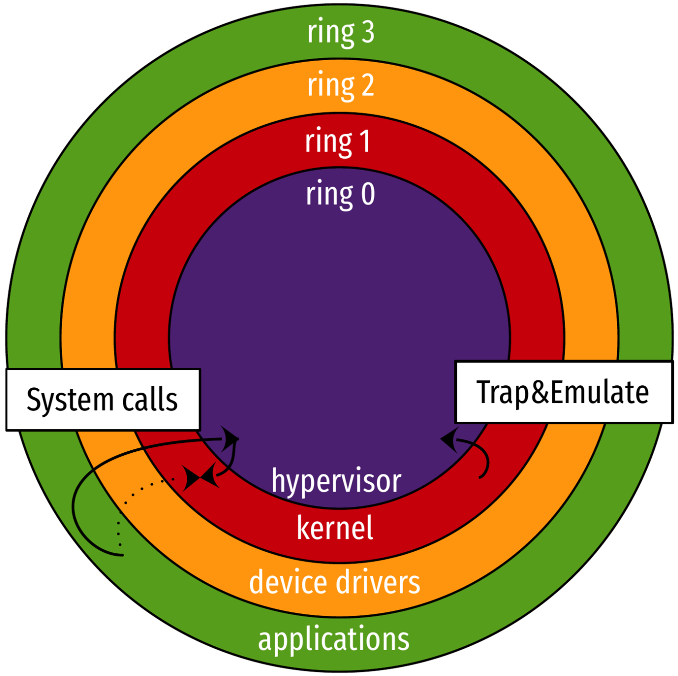
- General protection mechanism
- Userspace in ring 3
    - Use hardware by asking the kernel through **system calls** (syscalls)
- Kernel in ring 0
    - Full, exclusive control over the hardware
- Other rings generally unused



Privilege rings for x86 (numbered from highest privilege to lower).

CPU protection rings: full virtualization (1/2)

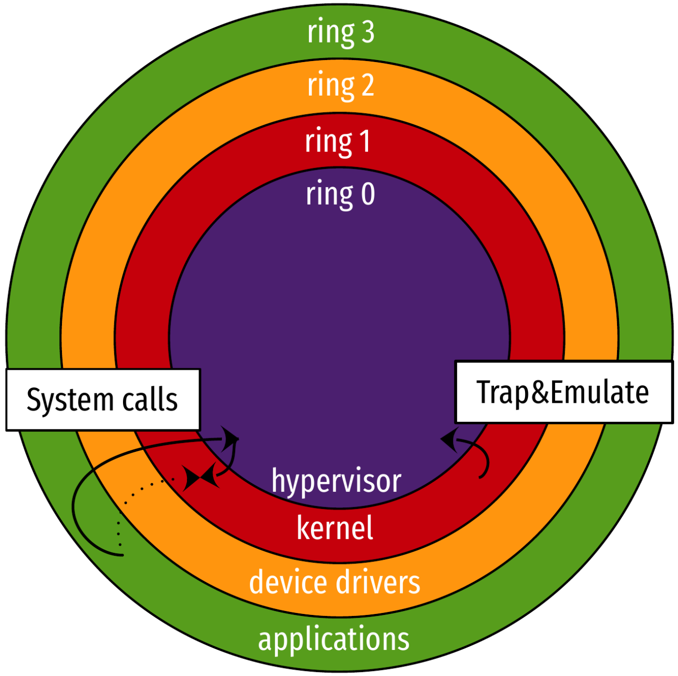
- Guest userspace in ring 3
  - Use hardware by asking the kernel through **system calls** (syscalls)
  - Implementation of syscalls uses interrupts, which control is privileged: the hypervisor redirects syscalls to the kernel in ring 1
- Kernel in ring 1, **unmodified**
  - Privileged operations are caught by the hypervisor
- **Hypervisor** in ring 0
  - Full, exclusive control over the hardware



Privilege rings with a hypervisor: full virtualization.

CPU protection rings: full virtualization (2/2)

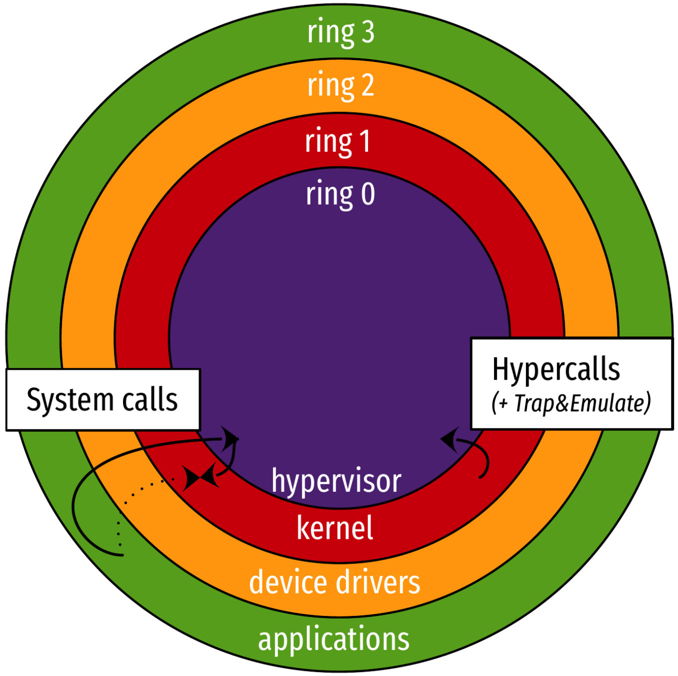
- The hypervisor implements **trap and emulate** workflow
  1. Privileged operations from the guest OS in ring 1 trigger General Protection Faults
  2. Hardware calls into ring 0 (i.e., hypervisor) to handle them
  3. Hypervisor emulates guest OS operations (**shadowing**)
- Upside: unmodified guest OS
- Downside: huge performance impact
  - This is visible when running a VM with QEMU, but without KVM!



Privilege rings with a hypervisor: full virtualization.

CPU protection rings: paravirtualization (1/2)

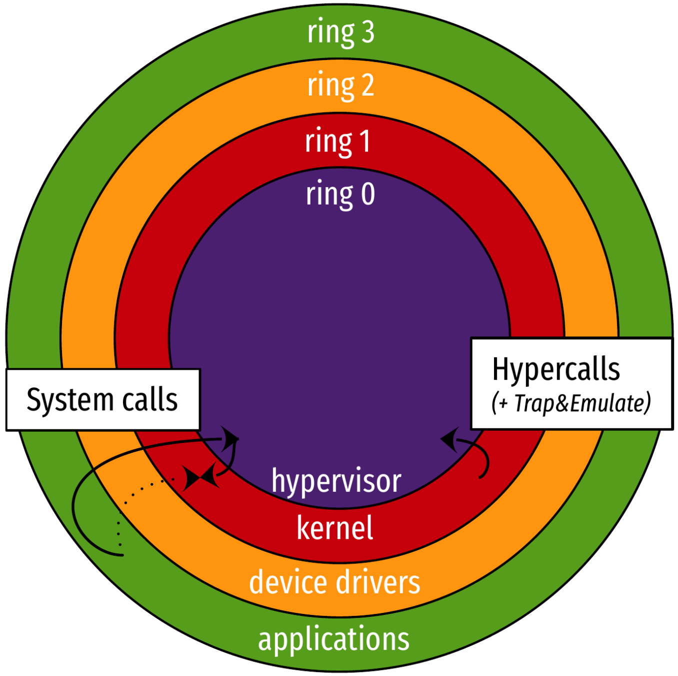
- Guest userspace in ring 3
  - Use hardware by asking the kernel through **system calls** (syscalls)
  - Implementation of syscalls uses interrupts, which control is privileged: the hypervisor redirects syscalls to the kernel in ring 1
- Kernel in ring 1, **modified for paravirtualization**
  - Unmodified privileged operations are caught by the hypervisor
  - Modified privilege operations are implemented by requesting the hypervisor via **hypercalls**
- **Hypervisor** in ring 0
  - Full, exclusive control over the hardware



Privilege rings with a hypervisor: paravirtualization.

CPU protection rings: paravirtualization (2/2)

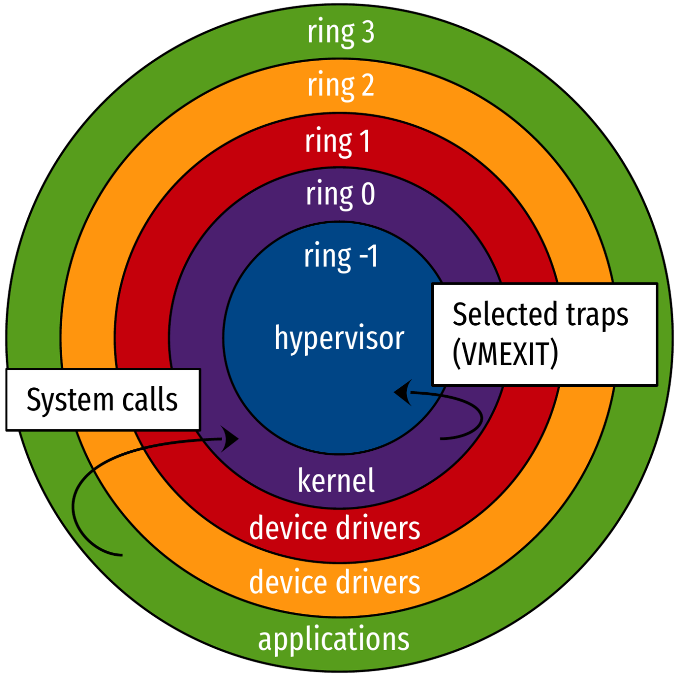
- The hypervisor offers an API (hypercalls) for the guest OS to ask for privileged operations without trap and emulation
- Upside: very good performance
- Downside: work to paravirtualize the guest OS
- Extends to **paravirtualized devices**:
  - Implementations tailored for virtual environments
  - Two sides: a front-end driver in the guest OS, and a back-end driver in the hypervisor
  - In QEMU/KVM: `virtio` drivers



Privilege rings with a hypervisor: paravirtualization.

CPU protection rings: hardware-assisted virtualization (1/2)

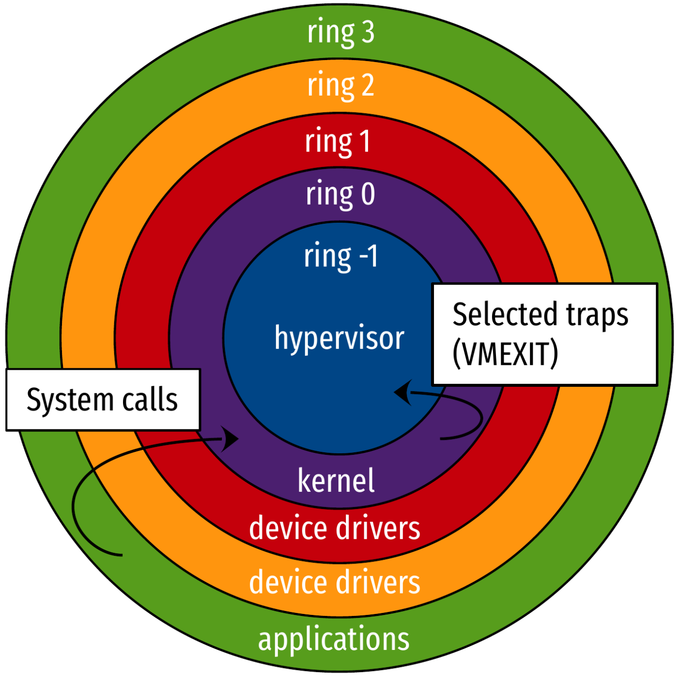
- Guest userspace in ring 3
  - Use hardware by asking the kernel through **system calls** (syscalls)
- Kernel in ring 0, **unmodified**
  - Most privileged operations are actually executed by the hardware, in a safe way
  - Some may be selected for trapping by the hypervisor
    - They trigger a **VMEXIT** to pass control
- **Hypervisor** in ring "-1"
  - Full, exclusive control over the hardware
  - Not an actual ring, but conceptually similar



Privilege rings with a hypervisor: hardware-assisted virtualization.

CPU protection rings: hardware-assisted virtualization (1/2)

- Support from the hardware allows selected traps to be taken by the hypervisor
- Upside: very good performance with unmodified guest
- Downside: none (hardware upgrades, but it’s now widely available)
- Extends to **memory**: Second Level Address Translation (SLAT)
  - Intel: Extended Page Table (EPT)
  - AMD: Nested Page Table
- Extends to devices: IOMMU, virtualization of interrupts…



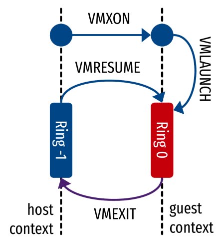
Privilege rings with a hypervisor: hardware-assisted virtualization.

Hardware-assisted virtualization with KVM

```
open("/dev/kvm");
ioctl(KVM_CREATE_VM);
ioctl(KVM_CREATE_VCPU);
for (;;) {
  // Jump into guest code with VMLAUNCH/VMRESUME until next VMEXIT (hypercall, etc.)
  exit_reason = ioctl(KVM_RUN);
  switch (exit_reason) {
  case KVM_EXIT_IO:     // Handle VM I/O
  case KVM_EXIT_HLT:    // Handle VM halting
  // ...
  }
}
```

Pseudo-code of a vCPU thread.

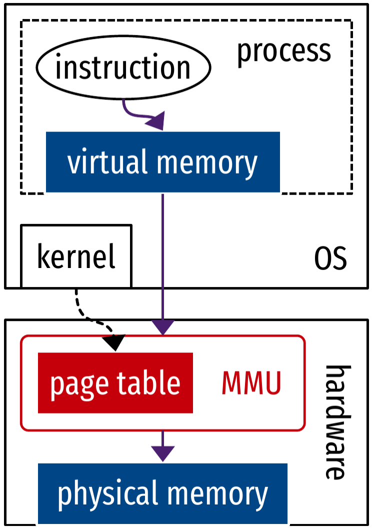
- KVM relies on **structures managed by the CPU**: Virtual Machine Control Structure (VMCS, Intel)
  - Stores vCPU context: registers, flags, etc.)
  - Includes reason for switching to hypervisor context…
- KVM `ioctl`s use special CPU instructions (examples are from Intel's set)



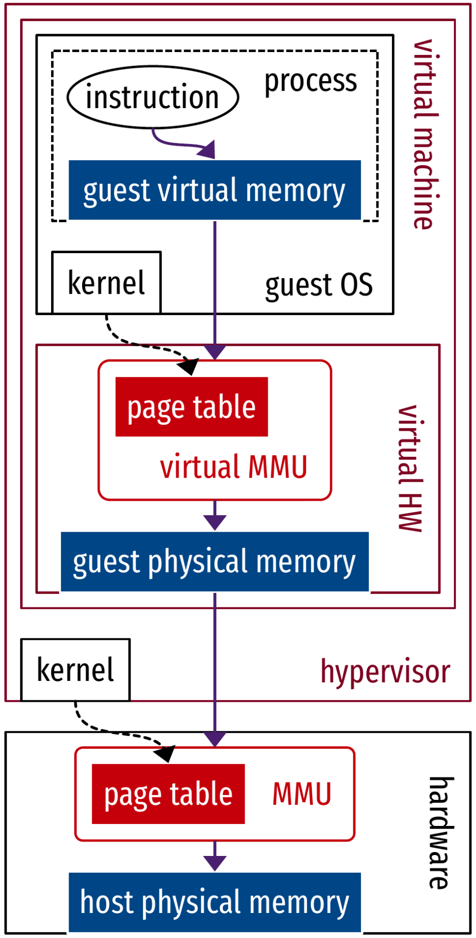
Codeflow of KVM with Intel VT-x.

Virtualization of memory

- Problem of translating memory addresses
  - How to implement a “virtual MMU”?



Native case: virtual memory of a process.



Virtualized case: guest memory vs. host memory.

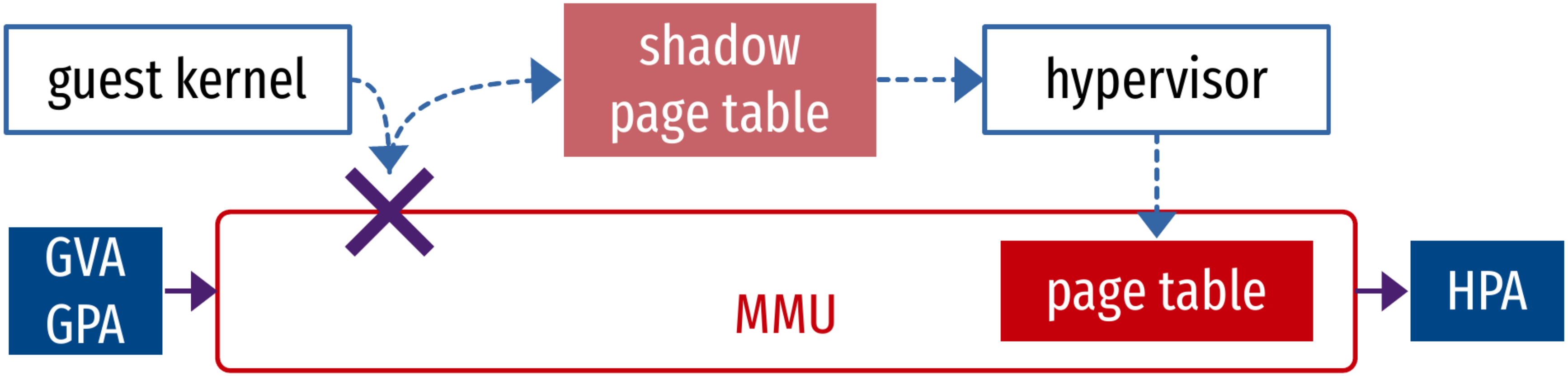
Virtualized memory translation

Translation of virtual addresses in a virtualized environment.

- The physical MMU is already used for the hypervisor page table
- Add a level of memory address translation:
  - Software solution: **shadow page table**
  - Hardware-assisted solution: **Second Level Address Translation** (SLAT)

Virtualized memory translation: shadow page table

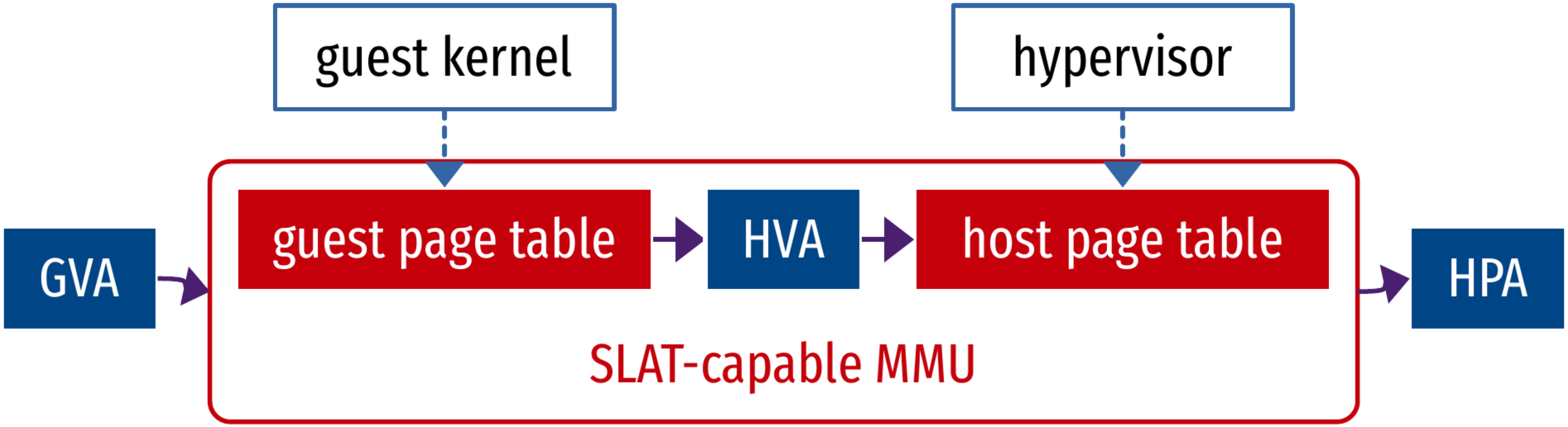
- Maintain a shadow page table (GVA to HPA) in the hypervisor
  - This is the one installed in the MMU
  - The guest OS's page table (GVA to GPA) is unused
- **Trap changes** to the page table made by the guest OS
  - Every write is recalculated and stored in the shadow page table
- Pros: **1-dimension page walk** *(see SLAT next)*
- Cons:
  - Very inefficient because of **traps** (full virtualization) on the critical path of memory management operations
  - Complex implementation



Shadow page table.

Virtualized memory translation: Second Level Address Translation (SLAT) (1/2)

1. Guest OS writes to its page table as natively, installs it in the MMU (GVA to GPA)
2. Hypervisor manages a second level page table, also installs it in the MMU (HVA to HPA)
3. The SLAT-capable MMU understands GPA as **Host Virtual Address** (HVA)



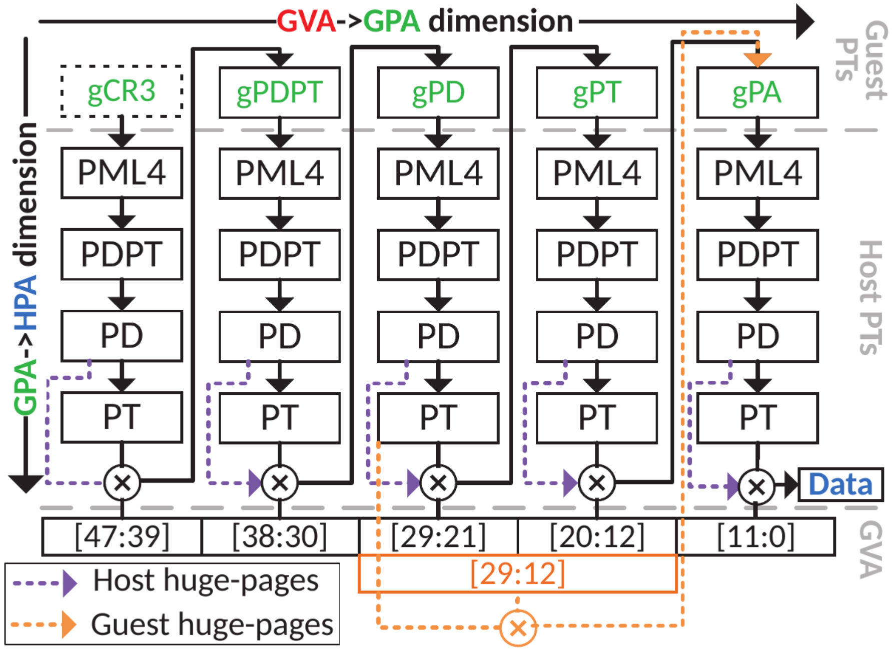
Second Level Address Translation (Nested Page Table).

Virtualized memory translation: Second Level Address Translation (SLAT) (2/2)

- Pros:
  - Efficient memory management operations for an unmodified guest OS
  - Almost no implementation work
- Cons: 2-dimension page walk, **6 times slower address translation in the worst case**
- Still the better solution:
  - Avoids complex implementation of shadow paging
  - Shadow paging is very slow anyway because of traps on memory management operations
  - Most performance overhead is compensated by:
    - The **Translation Look aside Buffer** (TLB) caches most translations
    - Huge pages avoid one level of translation

Virtualized memory translation: Second Level Address Translation (SLAT): Problem of 2-D page walk

- Given the native case: 1-D, 4 levels of page table → 4 memory accesses
- Virtualized case with SLAT: 4 levels of GVA to GPA, times 4 levels of GPA to HPA → **24 memory accesses**
  - 4 levels of guest page table, addressed as GPA
  - → 4 memory accesses to translate the GPA of 1 level to HPA, plus 1 access to actually read the level = (4+1) × 4 = 20 accesses to walk the page table
  - The walk gives a GPA -> 4 more memory accesses to translate to HPA



2-D memory address translation.
From Bergman et al. *Translation Pass-Through for Near-Native Paging Performance in VMs.* In USENIX ATC 2023.

Virtualization of I/O and devices

1. Traps and emulation
    - Guest OS uses drivers for real hardware
    - Hypervisor traps driver operations and emulate them on its own drivers
    - Bad performance
2. Paravirtualization (virtio)
    - **Front-end driver** in the guest OS, **back-end driver** in the hypervisor
    - Optimized interfaces between guest and HV (for I/O: network, block device)
3. Hardware assistance:
    - **IOMMU**: MMU to manage Direct Memory Access (DMA) of guests to devices
        - Handle HPA to GPA translation
        - Passthrough of physical functions
    - **Single Root Input Output Virtualization** (SR-IOV): virtualizable devices
        - Physical devices shared by exposing virtual functions

Hardware virtualization

- Virtualization is about **abstracting resources**
  - Hardware virtualization: create virtual machines with a hypervisor to run a guest OS
    - Full, para-, hardware-assisted virtualization
  - Example: QEMU/KVM, libvirt
- Virtualization is the cloud's cornerstone
  - Resource sharing, scalability and service delivery
- Virtualization of the hardware: CPU, memory, devices
  - A matter of collaboration between guest OS, HV and HW