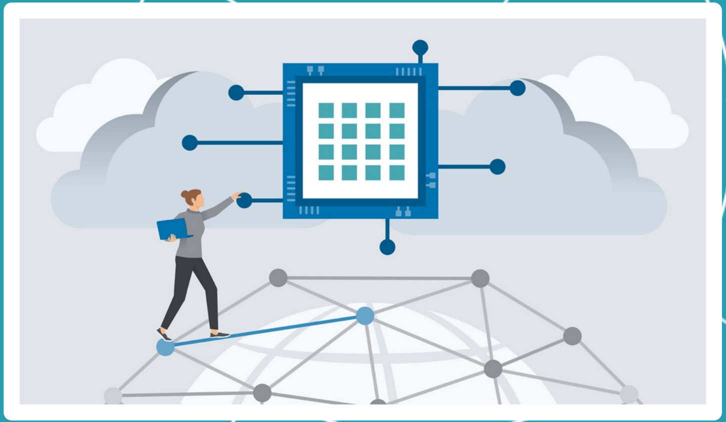
Hardware virtualization



Mathieu Bacou
mathieu.bacou@telecom-sudparis.eu

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

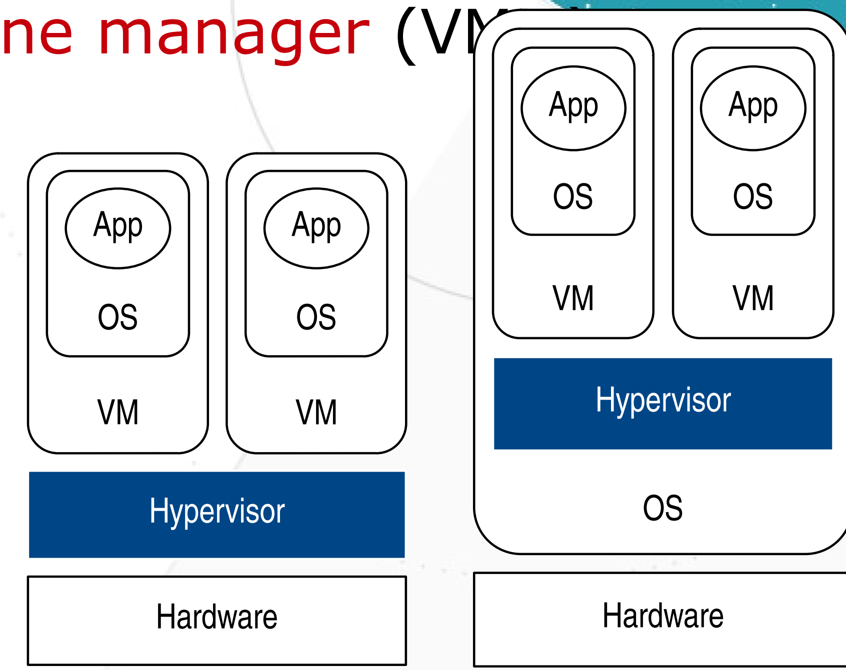
I. Hypervisor

II. Virtual machine and guest OS

III. User interface: libvirt

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 (VM
- Two types:
  - Type 1: native
    - Bare metal
    - Guest OSes are processes
  - Type 2: hosted
    - Process of a normal OS
    - Guest OSes are subprocesses



Type 1: native   Type 2: hosted

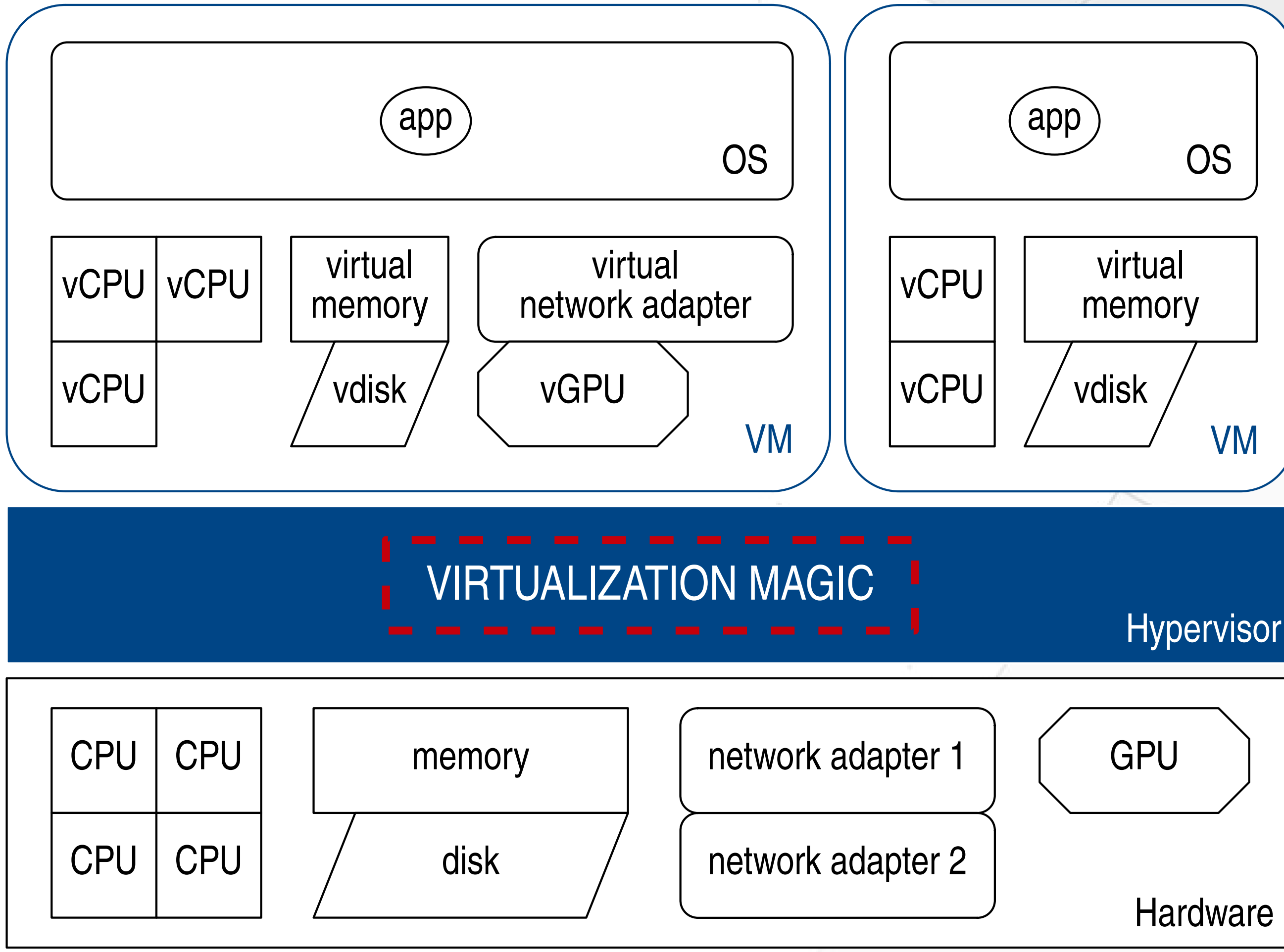
Hypervisors in the cloud

- 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!
- States: 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



User-friendly interface: libvirt

- Common and stable layer to manage VMs
  - Works also with other hypervisors
  - Also to manage storage and network
- Used by user front-ends: virsh, virtmanager...
  - Clients to libvirtd daemon

Commands and concepts

- Interactive shell: `virsh`
  - Help: `virsh help`
- Create a storage pool (a collection of VM images):

```
virsh pool-define-as mypool dir - - - - /path/to/pool/images
virsh pool-build
virsh pool-start
```

- Create a volume (a VM image):

```
virsh vol-create-as mypool myvolume 10GiB --format qcow2
```

Abstraction of VM images to manage them across the cloud (migration, replication...)

- Create a domain (a VM specification) using `virt-install` and a given install method (here, CD-ROM):

```
virt-install --name myubuntudomain --os-type=linux –os-variant=ubuntu20.04 \
  --memory 4096 --vcpus=4,maxvcpus=8 --network bridge=virtbr0 \
  --virt-type kvm --cpu host --disk 10,format=qcow2 --cdrom ubuntu.iso
```

  - Will also create the volume (so you can skip `vol-create-as`)
- Start a domain (run a VM): `virsh start myubuntudomain`
- Interactively jump into the domain: `virt-viewer --connect qemu:///session myubuntudomain`
- Shutdown a domain (terminate a VM): `virsh shutdown myubuntudomain` (force terminate with `virsh destroy`)
- Manually edit XML configuration of a domain: `virsh edit myubuntudomain`
- And more for network, checkpointing, device configuration...

Virtualization in the cloud

I. Life-cycle
II. Scalability
III. Resource management
IV. 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

Service scalability

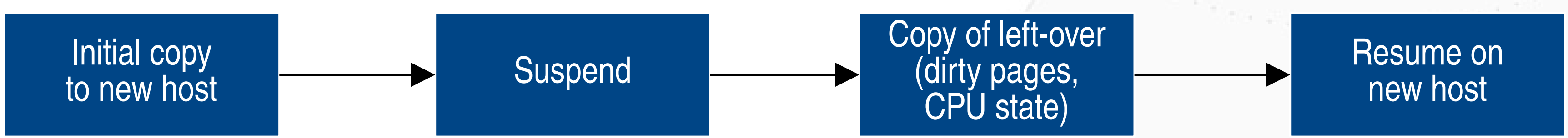
- Horizontal: add VMs
  - Under load spikes, replicate the service
    - Kill useless replicates after burst
  - Load balance between replications
- Automatic scaling

- Vertical: enlarge VMs
  - All hardware is virtual: dynamic addition of vCPUs or memory
- Hard to implement: how to unmap unused memory from the guest OS?
- Also: shutdown and replace with stronger VM
  - Keep the same image!

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 used: too harmful when it collapses
- Migration: VMs are loosely attached to hosts, so move them around
  - Optimize resource usage on physical hosts
  - Optimize datacenter usage by powering only needed hosts

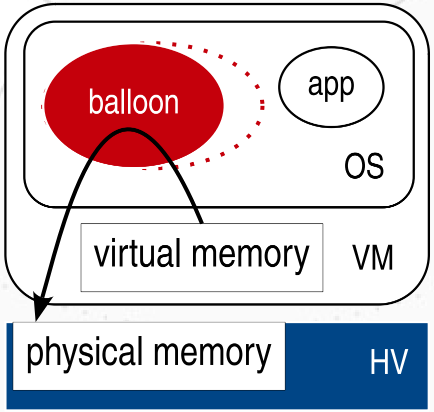
consolidation



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

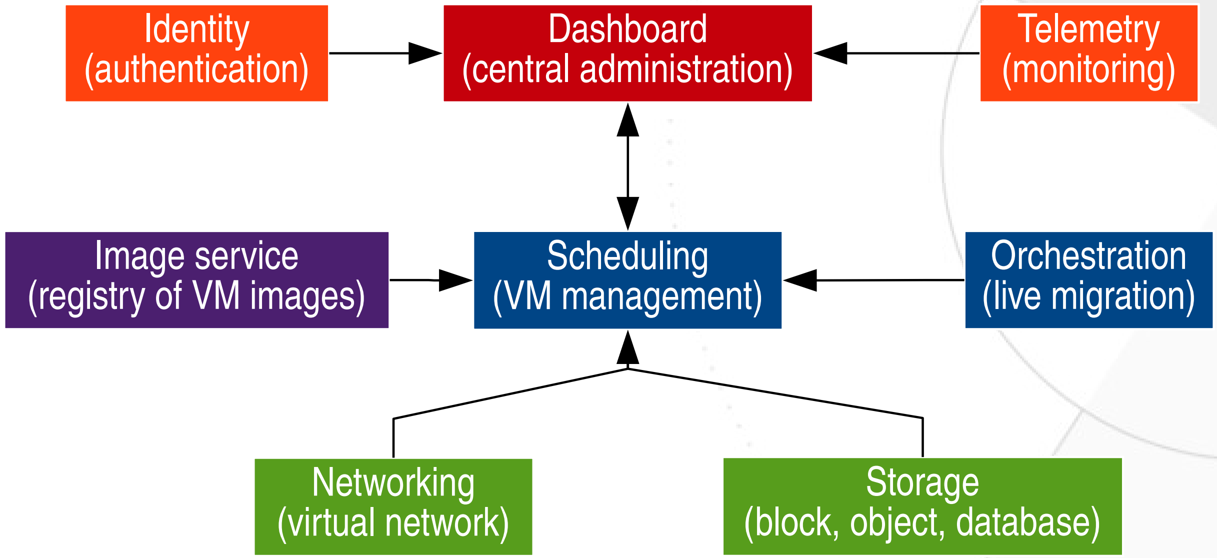


Paravirtualized ballooning

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



Cloud infrastructure example:
OpenStack (simplified view)

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: libvirt for VM management and configuration

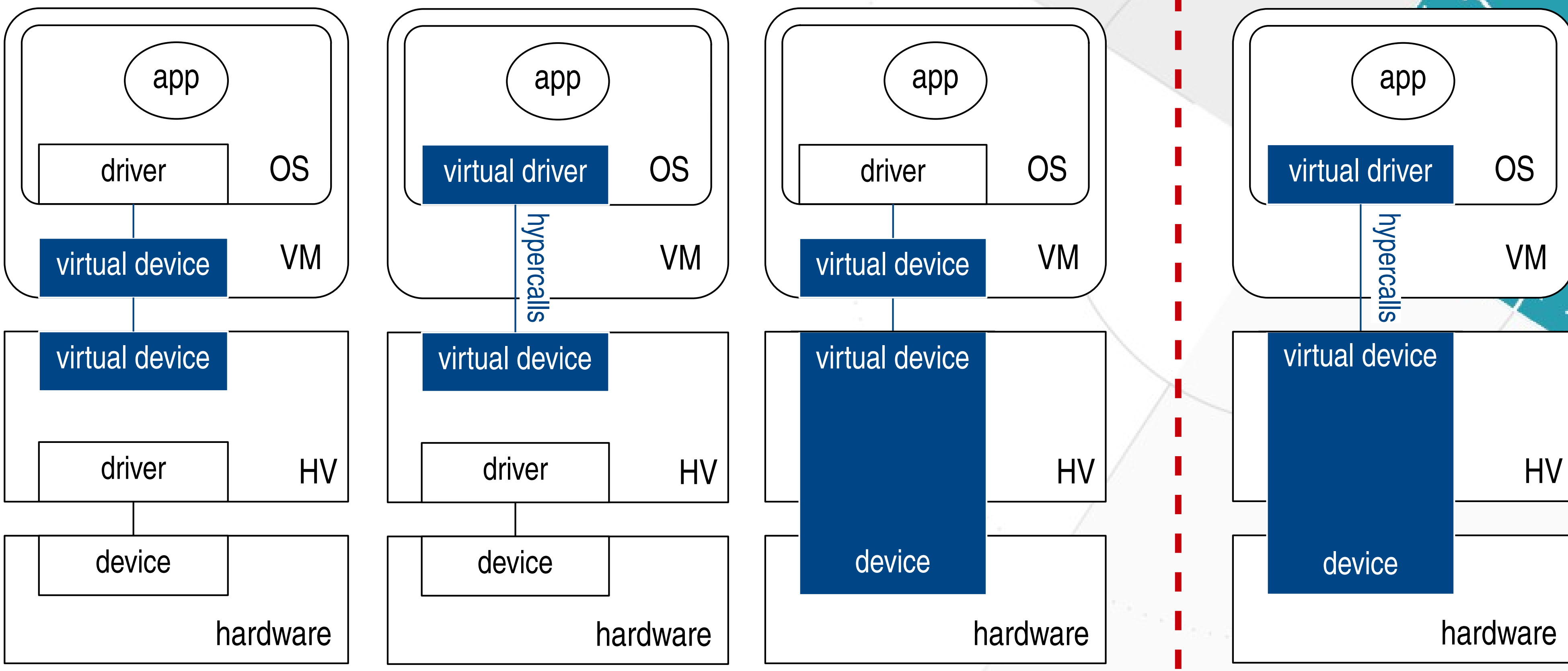
Internals of an hypervisor

I. Modes of virtualization
II. Architectural overview of QEMU/KVM
III. Virtualization of CPUs
IV. Virtualization of memory
V. Virtualization of I/O and devices

Modes of virtualization

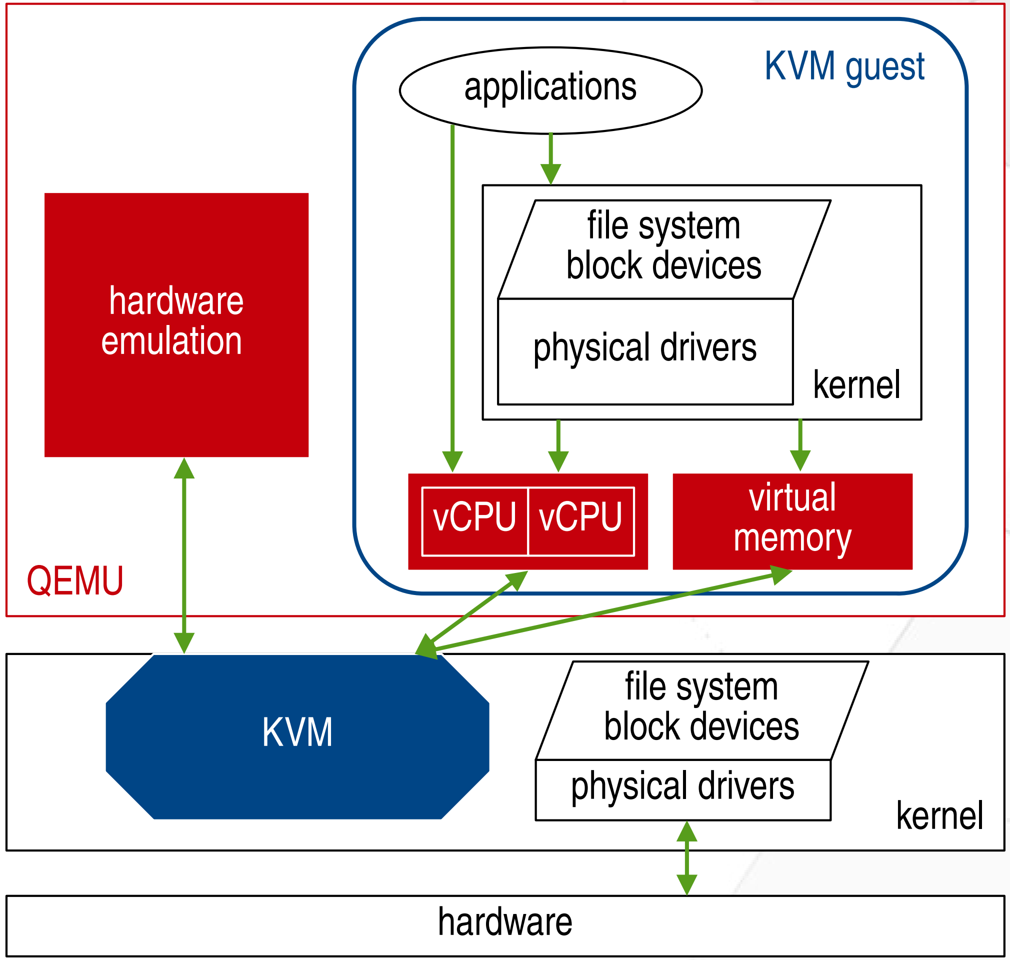
- Three modes to virtualize a guest OS:
  1) Full virtualization: total simulation of virtual hardware
     - Unmodified guest OS
     - Binary translation
  2) Paravirtualization: virtualization interface between guest OS and HV
     - Paravirtualized guest OS: deep changes, paravirtualized drivers
     - 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

Virtualization modes of guest OS



Full virtualization | Paravirtualization | Hardware-assisted virtualization | Hardware-assisted paravirtualization

Architectural overview

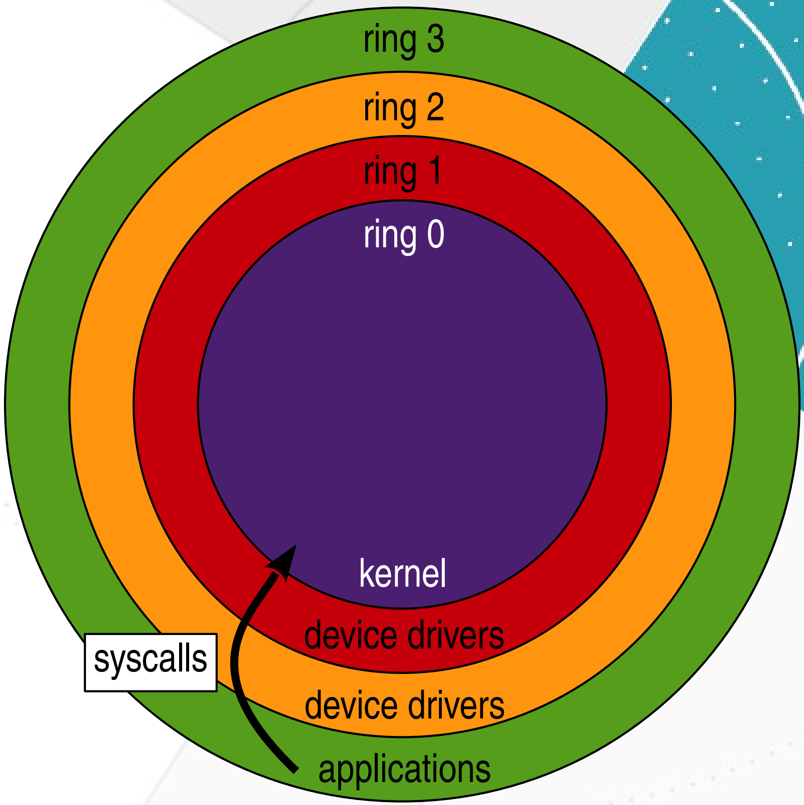


Virtualization of CPUs

- 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

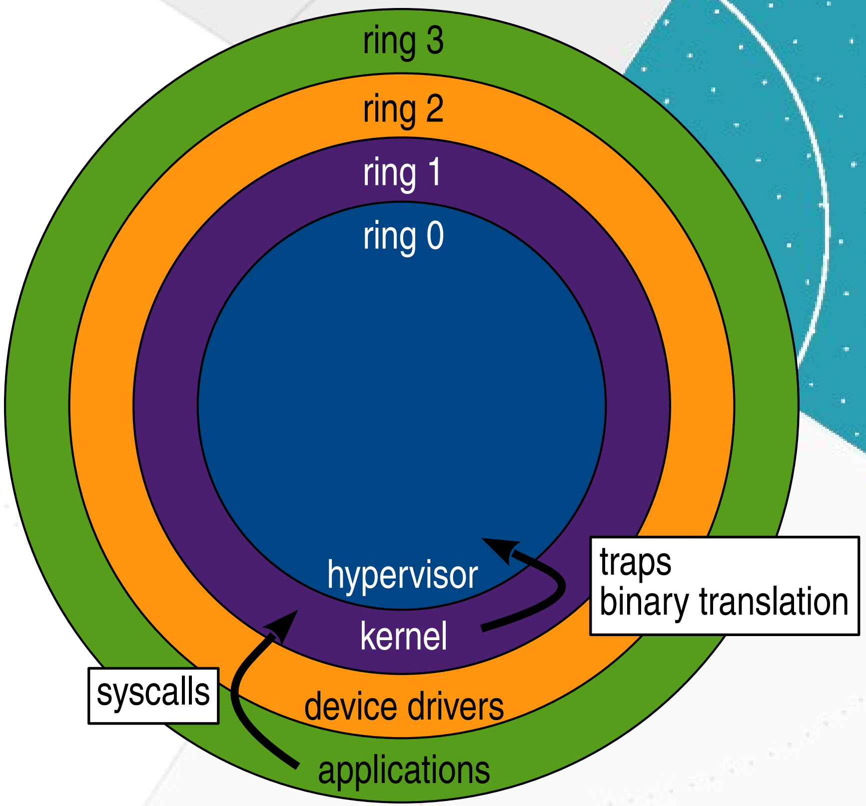
- General protection mechanism
- Userspace in ring 3
  - Use hardware by asking the kernel through syscalls
- Kernel in ring 0
  - Full, exclusive control over the hardware
- Other rings generally unused



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

CPU rings: full virtualization

- Guest userspace in ring 3
  - Use hardware by asking the kernel through syscalls
- Kernel in ring 1, unmodified
- Hypervisor in ring 0
  - Full, exclusive control over the hardware
  - Trap requests from guest kernel



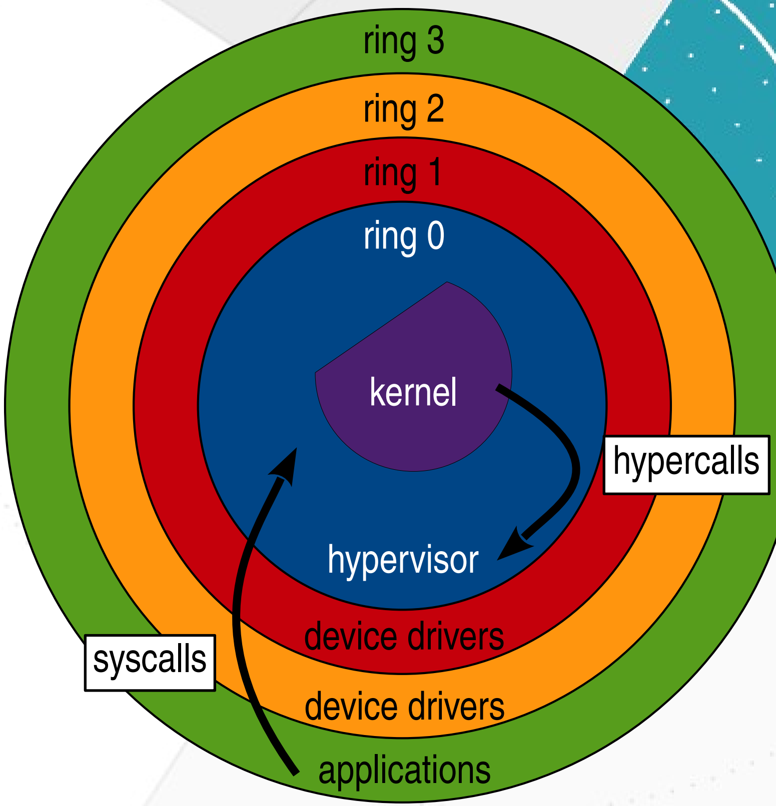
Privilege rings with hypervisor: full virtualization

Full virtualization

- Install traps to intercept privileged instructions
  - Other instructions are directly executed as normal
- On trap: simulate the instruction with shadowing
  - Security checks to maintain isolation between guest VMs
  - Binary translate instruction before privileged execution
- Upside: unmodified guest OS
- Downside: huge performance impact
  - You can check it by running a QEMU VM without KVM!

CPU rings: paravirtualization

- Guest userspace in ring 3
  - Use hardware by asking the kernel through syscalls
- Kernel in ring 0, modified for paravirtualization
  - Use hardware by asking the hypervisor through hypercalls
- Hypervisor in ring 0
  - Full, exclusive control over the hardware



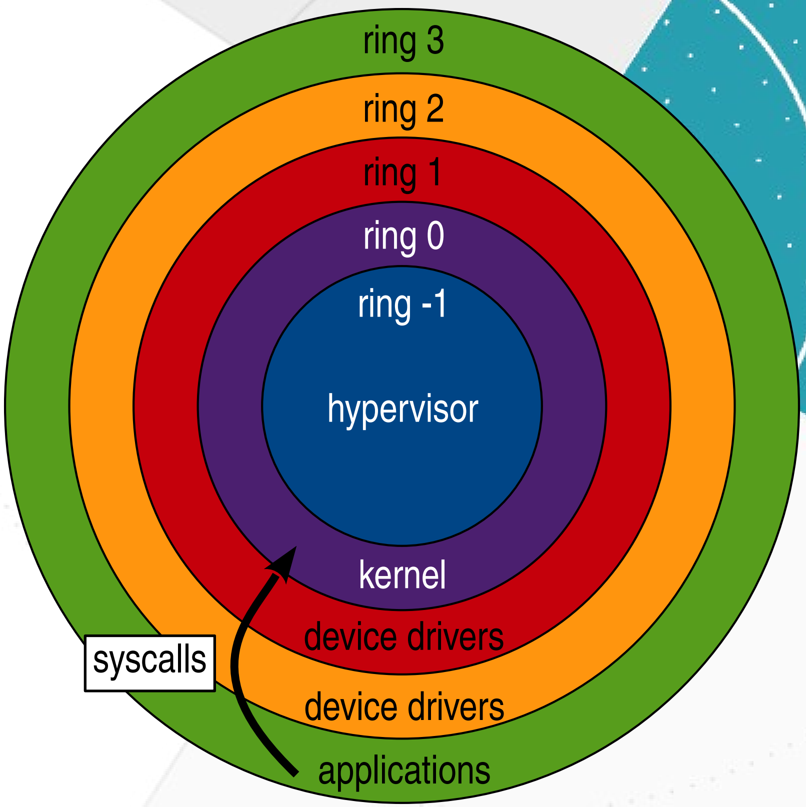
Privilege rings with hypervisor: paravirtualization

Paravirtualization

- Modify guest OS for paravirtualization
  - Use an API provided by the hypervisor: hypercalls
  - Otherwise run in ring 0 as usual
- Upside: very good performance
- Downside: work to paravirtualize guest OS
- Extends to paravirtualized devices and drivers
  - Front-end driver in guest OS, back-end driver in HV
  - In QEMU/KVM: virtio drivers

CPU rings: hardware-assisted

- Guest userspace in ring 3
  - Use hardware by asking the kernel through syscalls
- Kernel in ring 0, unmodified
- Hypervisor in ring -1
  - Full, exclusive control over the hardware



Privilege rings with hypervisor: hardware-assisted virtualization

Hardware-assisted virtualization

- Hypervisor in new, over-privileged ring -1
  - Guest OS in expected ring 0, with ring 3 for userspace
- On privileged operations from the guest, transition from VM context to HV context
  - Similar to the OS handling events and exceptions from its userspace with processor help
  - The hard work is moved down to the processor (Intel VT-x, AMD-V)
- Upside: unmodified guest, very good performance
- Downside: none
- Extends to memory: Extended Page Tables
- Extends to devices: IOMMU, IRQ virtualization...

Code speaks: KVM virtualization

Pseudo-code of a vCPU thread

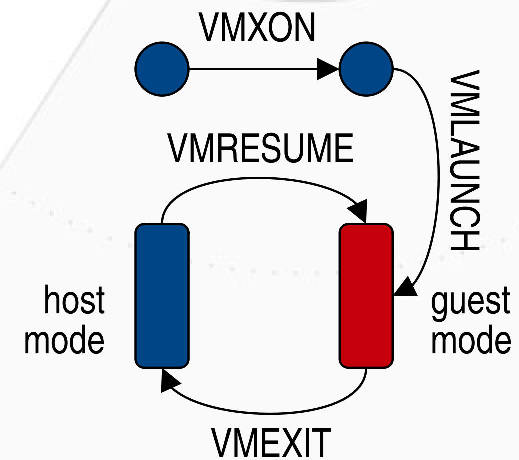
```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
for (;;) {
     ioctl(KVM_RUN)
     switch (exit_reason) {
     case KVM_EXIT_IO:  /* ... */
     case KVM_EXIT_HLT: /* ... */
     }
}
```

Jump into guest code with VM{LAUNCH,RESUME} until VMEXIT (hypercall, etc.)

Handle virtualization: I/O, VM halt...

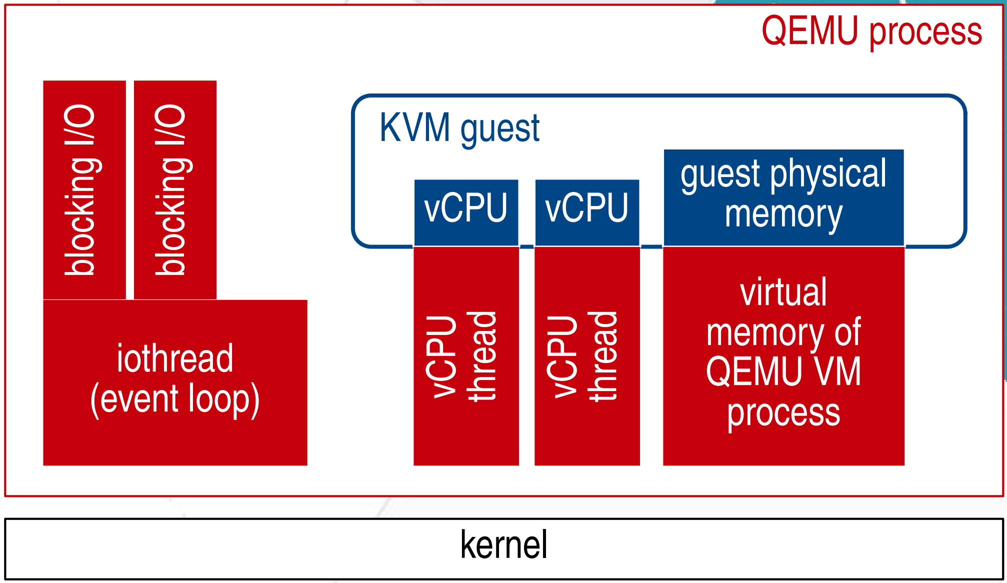
- KVM relies on structures managed by the CPU: Virtual Machine Control Structure (VMCS, Intel)
  - Stores vCPU context (registers, flags, etc.), reason for switching to HV context...
- KVM `ioctls` use special CPU instructions (Intel set)

Codeflow of KVM with Intel VT-x



VM context switching

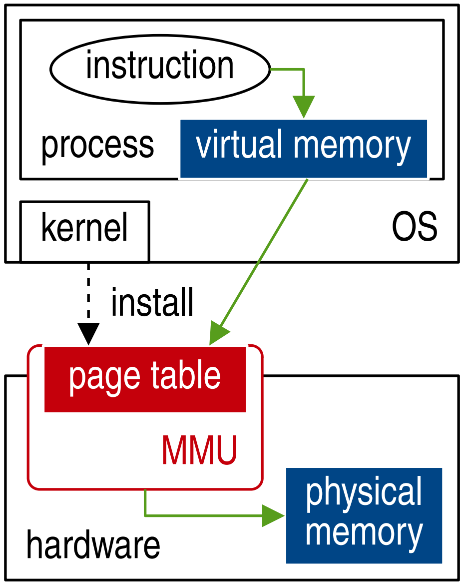
- Schedule VMs
- Switch world: CPU registers, IRQs, memory maps...
- QEMU is in Linux userland: normal scheduler switching threads
- Threading model: one per vCPU + event loop for I/O
  - Sub-threads for blocking I/O
  - Global mutex around QEMU



QEMU threading model

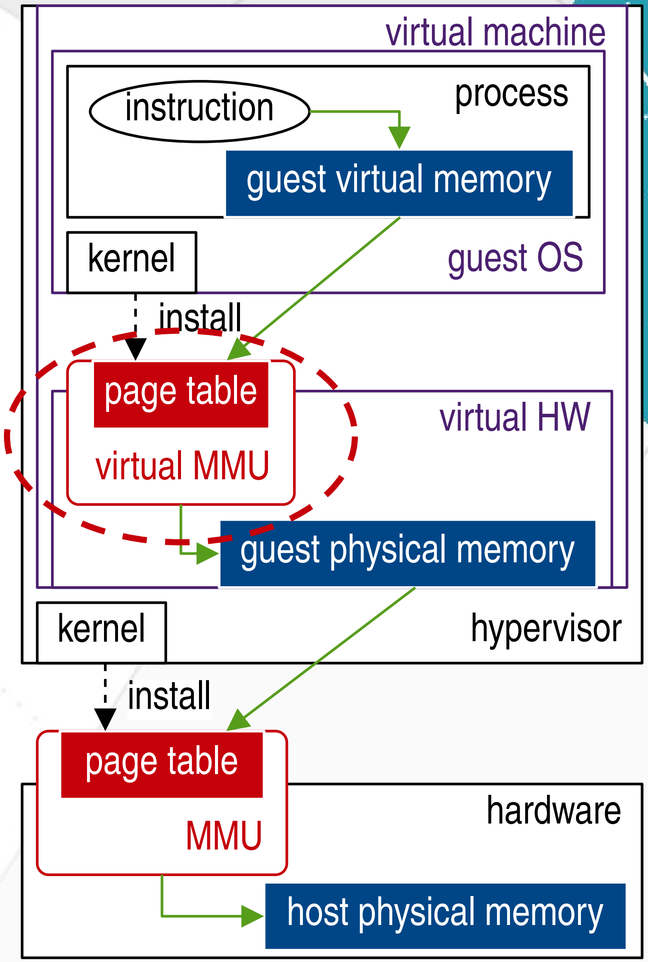
Virtualization of memory

- Problem of translating memory addresses



Native case: virtual memory of a process

How to implement this virtual MMU?



Virtualized case: guest memory vs. host memory

Virtualized memory translation

Translation of virtual addresses in virtualized environment

- The physical MMU is already used for the HV page table
- Add a level of memory address translation: shadow page table
  - Maintain a shadow page table in the hypervisor
    - Trap changes to the page table made by the guest OS
    - Map host physical memory to guest physical memory
    - Apply changes to the shadow page table
  - Very inefficient, in the critical path

HW-assisted memory translation

- With KVM and hardware assistance: Second Level Address Translation (SLAT)
  - Intel: Extended Page Table (EPT)
- “Nest” the host page table into the guest table
- The physical MMU handles the whole translation from guest virtual addr. to host physical addr.

Virtualization of I/O and devices

1) Traps and emulation
   - Using physical drivers in the HV
   - 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)
   - (QEMU/KVM) vhost: emulate devices in kernel to use kernel-only optimizations
3) Hardware assistance:
   - IOMMU: MMU to manage Direct Memory Access (DMA) of guests to devices
     - 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: creates 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