I/O in Linux Hypervisors and Virtual Machines
Lecture for the Embedded Systems Course
CSD, University of Crete (May 8 & 10, 2014)

- Manolis Marazakis (maraz@ics.forth.gr)

FORTH-ICS Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)

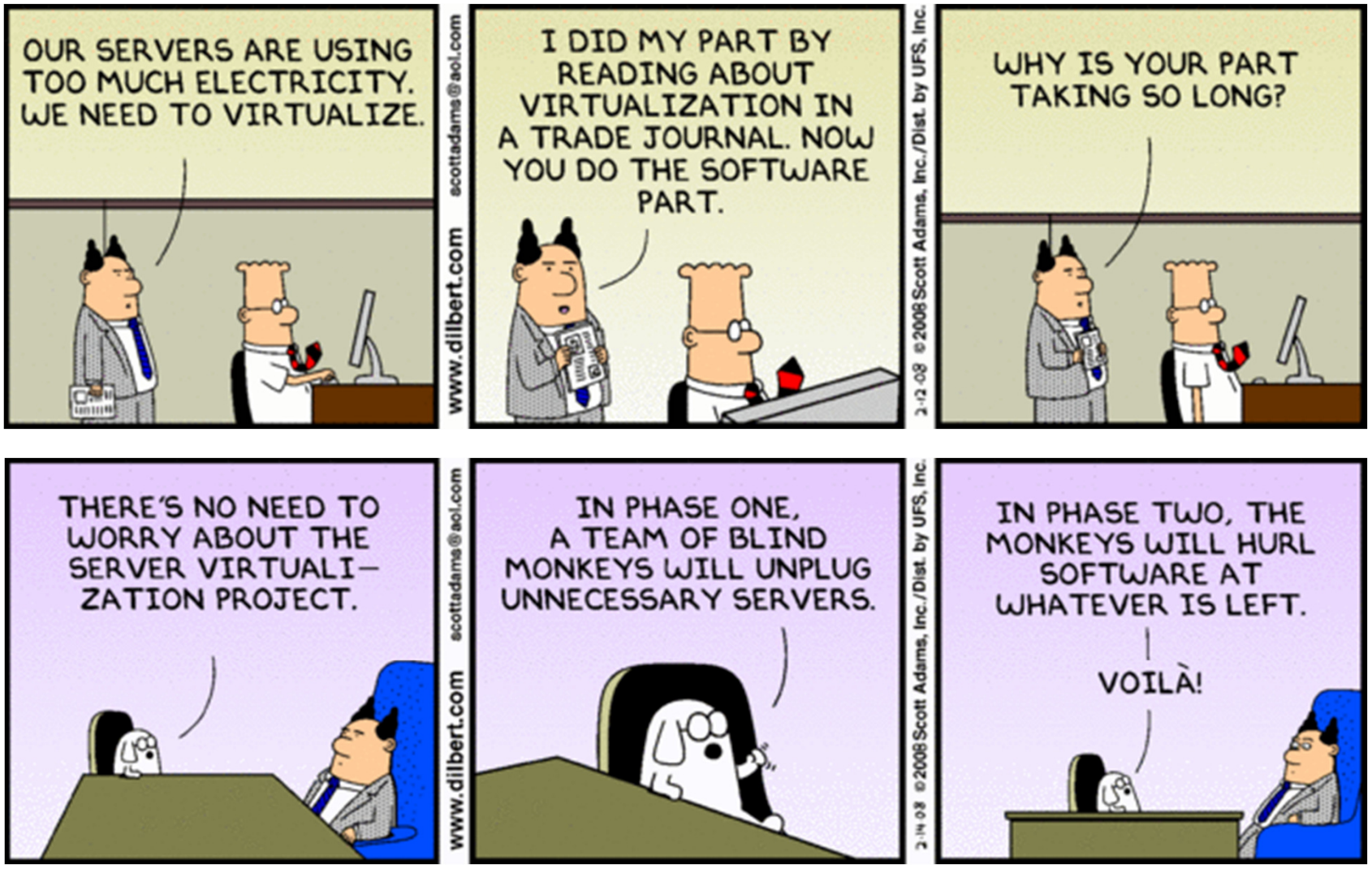
Virtualization Use-cases

- Server (workload) consolidation
- Legacy software systems
- Virtual desktop infrastructure (VDI)
- End-user virtualization (e.g. S/W testing & QA, OS research)
- Compute clouds
- Embedded (e.g. smartphones)

How does virtualization work, in detail ?

… emphasis on I/O

Virtualization is hard !



Outline

- Elements of virtualization
  - Machine emulator, Hypervisor, Transport
- Present alternative designs for device I/O path
  - Device emulation (fully virtualized)
  - Para-virtualized devices
  - Direct device assignment (pass-through access)
- Follow the I/O path of hypervisor technologies commonly used on Linux servers (x86 platform)
  - xen, kvm
- Provide a glimpse of hypervisor internals *
  - kvm: virtio
  - xen: device channels, grant tables
  - * VMware: market leader … but out-of-scope for this lecture

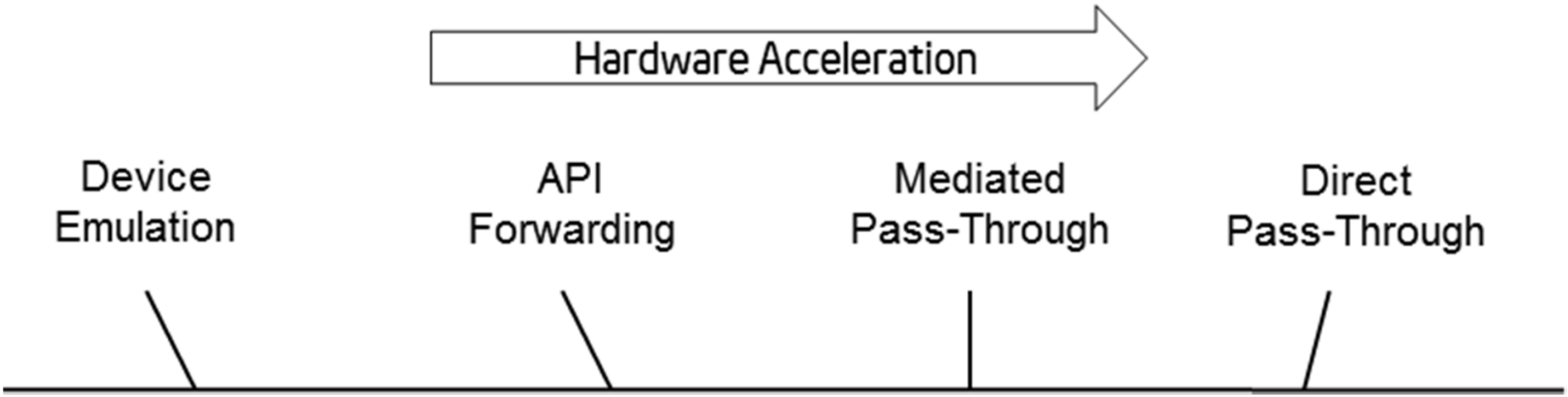
Why is I/O hard to virtualize ?

- Multiplexing/de-multiplexing for guests
- Programmed I/O (PIO), Memory-mapped I/O (MMIO)
  - PIO: privileged CPU instructions specifically for performing I/O
    - I/O devices have a separate address space from general memory
  - MMIO: the CPU instructions used to access the memory are also used for accessing devices
    - The memory-mapped I/O region is protected
- Direct Memory Access (DMA)
  - Allow certain hardware subsystems within the computer to access system memory for reading and/or writing independently of the CPU
    - Synchronous: triggered by software
    - Asynchronous: triggered by devices (e.g. NIC)
- Implementation layers:
  - system call (application to GuestOS) → trap to VMM
  - driver call (GuestOS) → paravirtualization
    - Hypercall between modified driver in GuestOS and VMM
  - I/O operation (GuestOS driver to VMM)

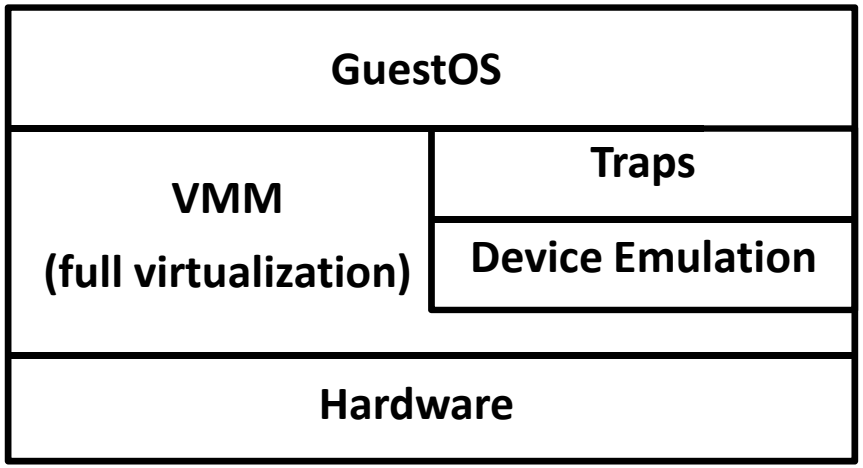
I/O device types

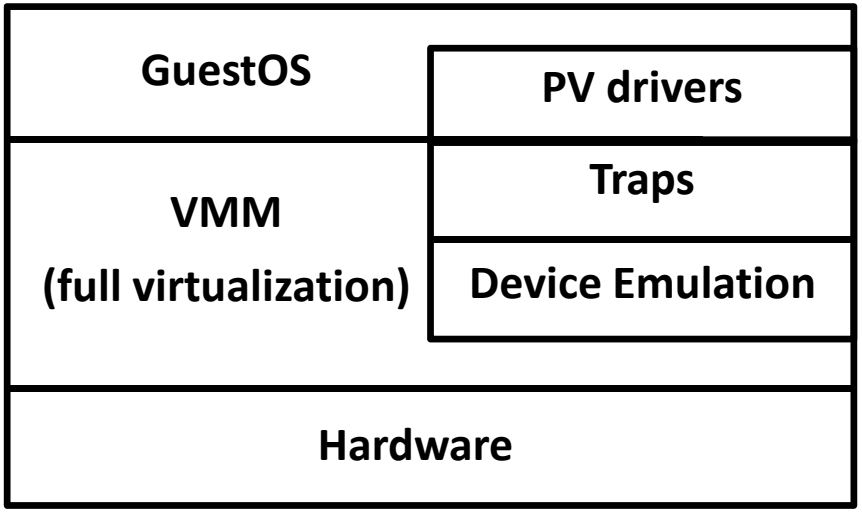
- Dedicated
  - E.g. display
- Partitioned
  - E.g. Disk
- Shared
  - E.g. NIC
- Devices that can be enumerated (PCI, PCI-Express)
  - VMM needs to emulate ‘discovery’ method, over a bus/interconnect
- Devices with hard-wired addresses (e.g. PS/2)
  - VMM should maintain status information on virtual device ports
- Emulated (e.g. experimental hardware)
  - VMM must define & emulate all H/W functionality
  - GuestOS needs to load corresponding device drivers

VM I/O acceleration



Device emulation: full virtualization vs para-virtualization





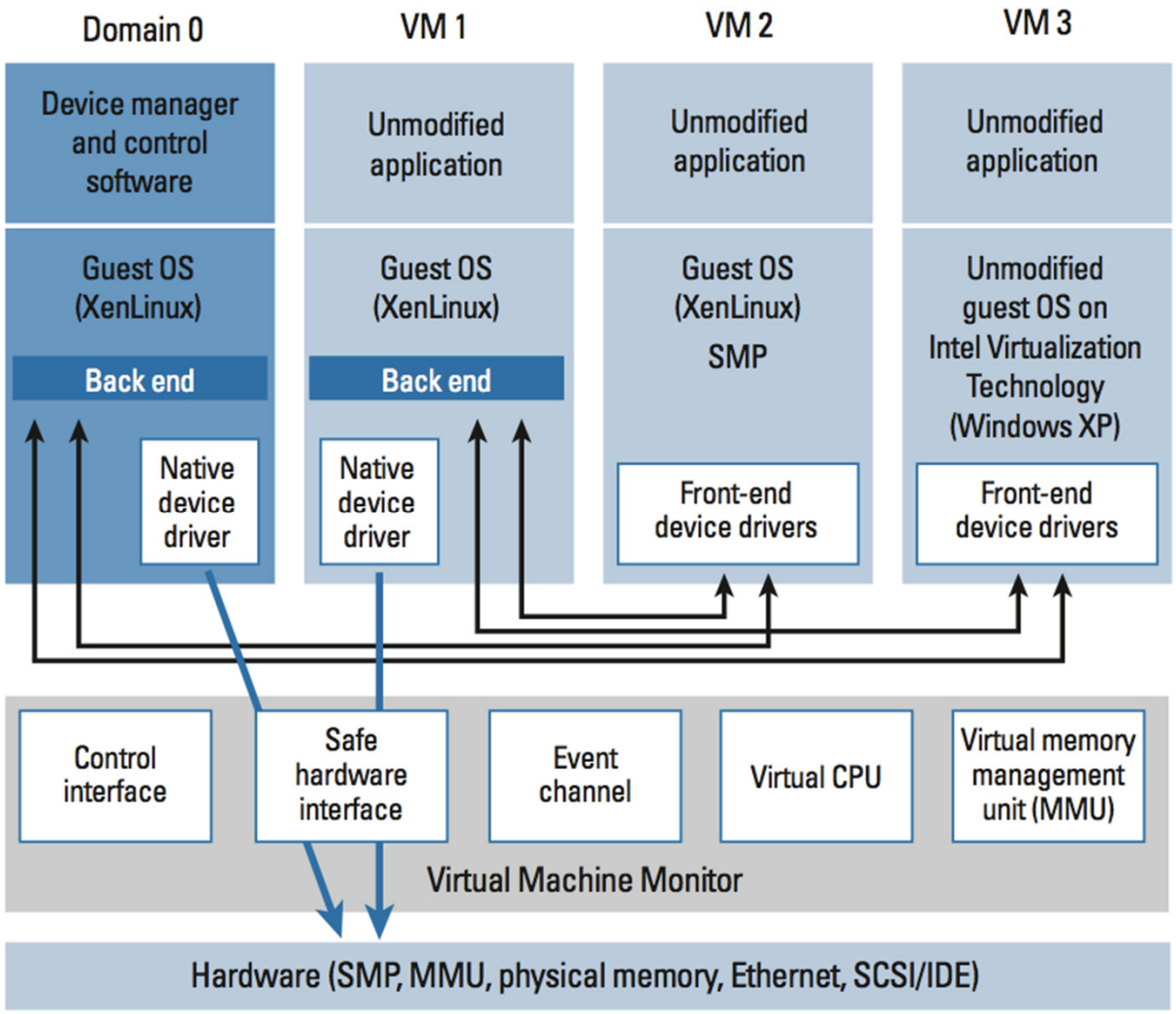
Device model (with full-system virtualization)

- VMM intercepts I/O operations from GuestOS and passes them to device model at the host
- Device model emulates I/O operation interfaces:
  - PIO, MMIO, DMA, …
- Two different implementations:
  - Part of the VMM
  - User-space standalone service

xen history

- Developed at Systems Research Group, Cambridge University (UK)
- Creators: Keir Fraser, Steven Hand, Ian Pratt, et al (2003)
- Broad scope, for both Host and Guest
- Merged in Linux mainline: 2.4.22
- Company: Xensource.com
  - Acquired by Citrix Systems (2007)

Xen VMM: Paravirtualization (PV)



OS'es

- Dedicated control domain: Dom0
- Modified Guest OS

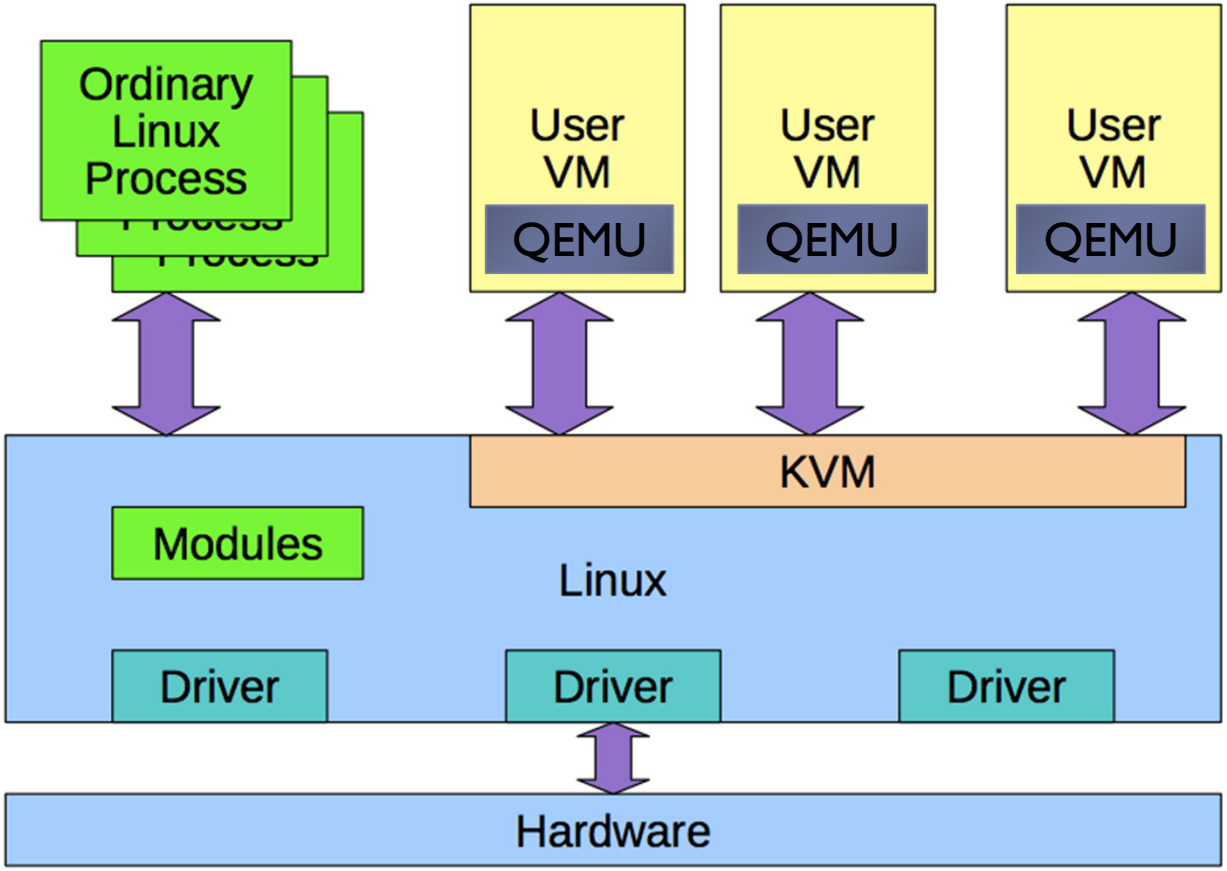
Devices

- Front-end (net-front) for Guest OS to communicate with Dom0
- I/O channel (zero copy)
- Backend (net-back) for Dom0 to communicate with underlying systems

kvm history

- Prime creator: Avi Kivity, Qumranet, circa. 2005 (IL)
  - Company acquired by RedHat (2008)
- “Narrow” focus: x86 platform, Linux host
  - Assumes Intel VT-x or AMD svm
- Merged in Linux kernel mainline: 2.6.20
  - … < 4 months after 1st announcement !

kvm VMM: tightly integrated in the Linux kernel



- Hypervisor: Kernel module
- Guest OS: User-space process (QEMU)
- Requires H/W virtualization extensions

xen vs kvm

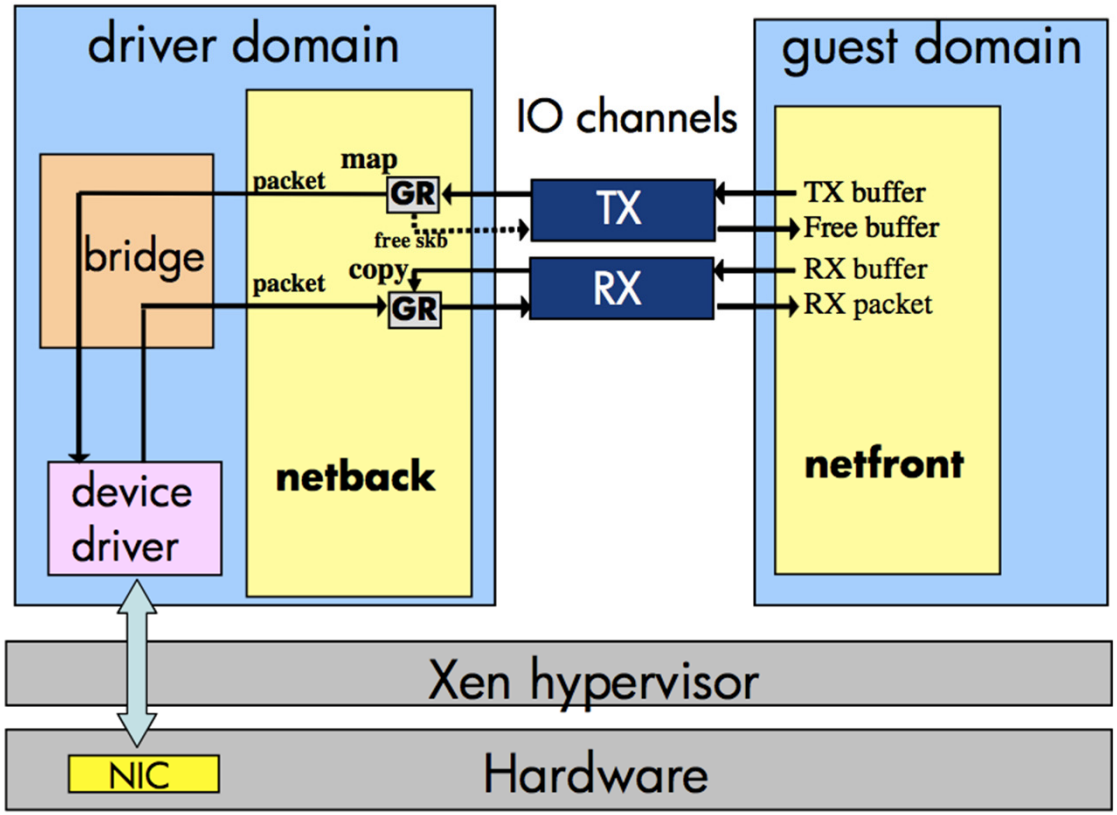
Xen

- Strong support for para-virtualization with modified host-OS
  → Near-native performance for I/Os
- Separate code based for DOM0 and device drivers
- Security model:
  Rely on DOM0
- Maintainability
  Hard to catch up all versions of possible guests due to PV

KVM

- Requires H/W virtualization extension – Intel VT, AMD Pacifica (AMD-V)
- Limited support for para-virtualization
- Code-base integrated into Linux kernel source tree
- Security model:
  Rely on Commodity/Casual Linux systems
- Maintainability
  Easy – Integrated well into infrastructure, code-base

I/O virtualization in xen

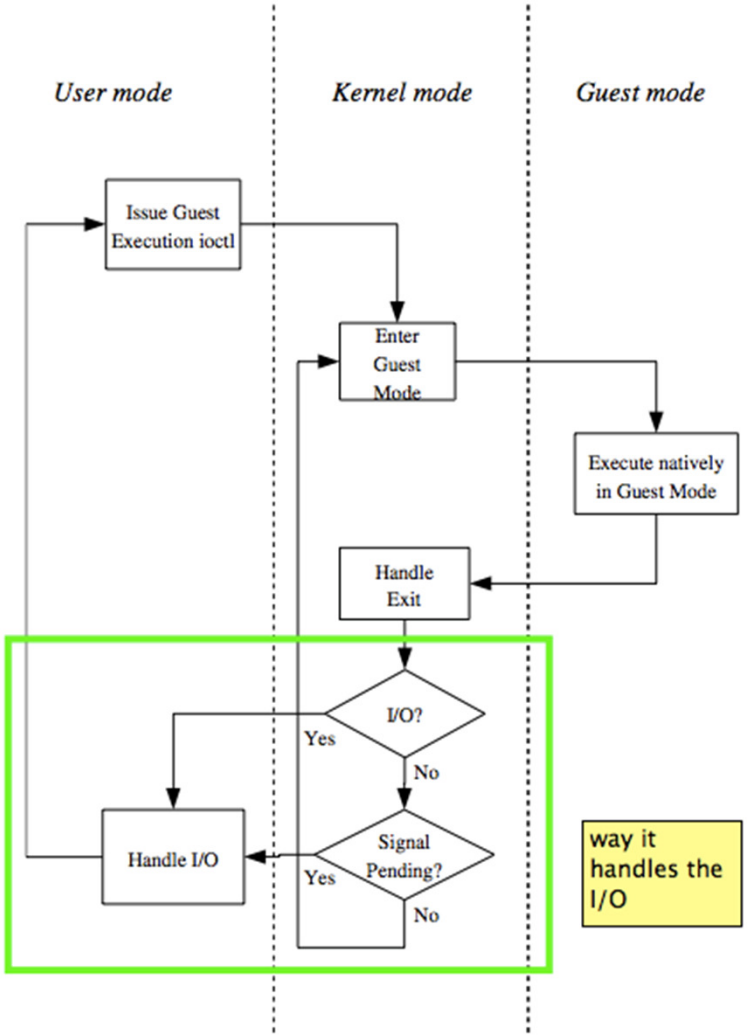


- Bridge in driver domain: multiplex/de-multiplex network I/Os from guests
- I/O Channel
  - Zero-copy transfer with Grant-copy
  - Enable driver domain to access I/O buffers in guest memory

- Xen network I/O extension schemes
  - Multiple RX queues, SR-IOV …

Source: "Bridging the gap between software and hardware techniques for i/o virtualization" Usenix Annual Technical conference, 2008

I/O virtualization in kvm

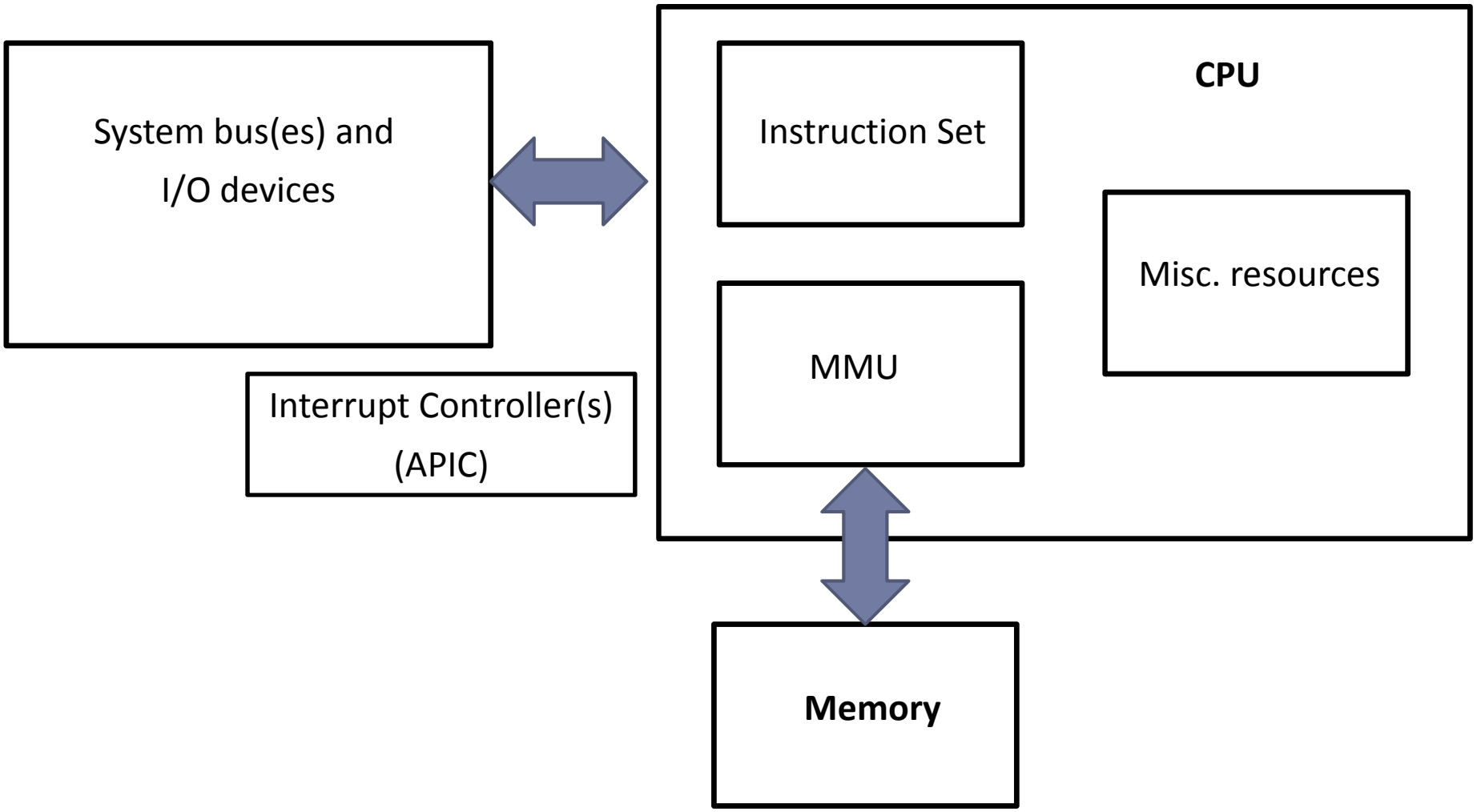


- Native KVM I/O model
- PIO: Trap
- MMIO: The machine emulator executes the faulting instruction
- Slow due to mode-switch!
- Extensions to support PV
  - VirtIO: An API for Virtual I/O aims to support many hypervisors of all type

Virtualization components

- Machine emulation
  - CPU, Memory, I/O
  - Hardware-assisted
- Hypervisor
  - Hyper-call interface
  - Page Mapper
  - I/O
  - Interrupts
  - Scheduler
- Transport
  - Messaging, and bulk-mode

Emulated Platform



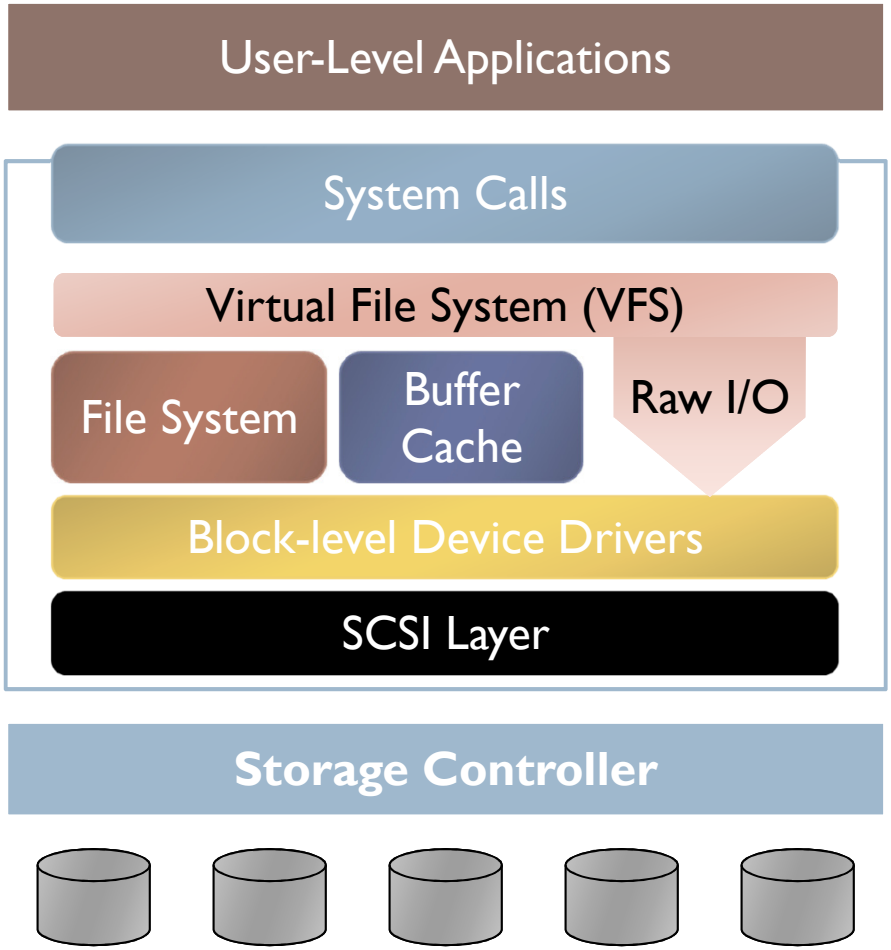
QEMU machine emulator

- Creator: Fabrice Bellard (circa. 2006)
- Machine emulator using a dynamic translator
  - Run-time conversion of target CPU instructions to the host instruction set
  - Translation cache
- Emulated Machine := { CPU emulator, Emulated Devices, Generic Devices } + “machine description”
  - Link between emulated devices & underlying host devices
  - Alternative storage back-ends – e.g. POSIX/AIO (→ thread pool)
  - Caching modes for devices:
    - cache=none → O_DIRECT
    - cache=writeback →buffered I/O
    - Cache=writethrough → buffered I/O, O_SYNC

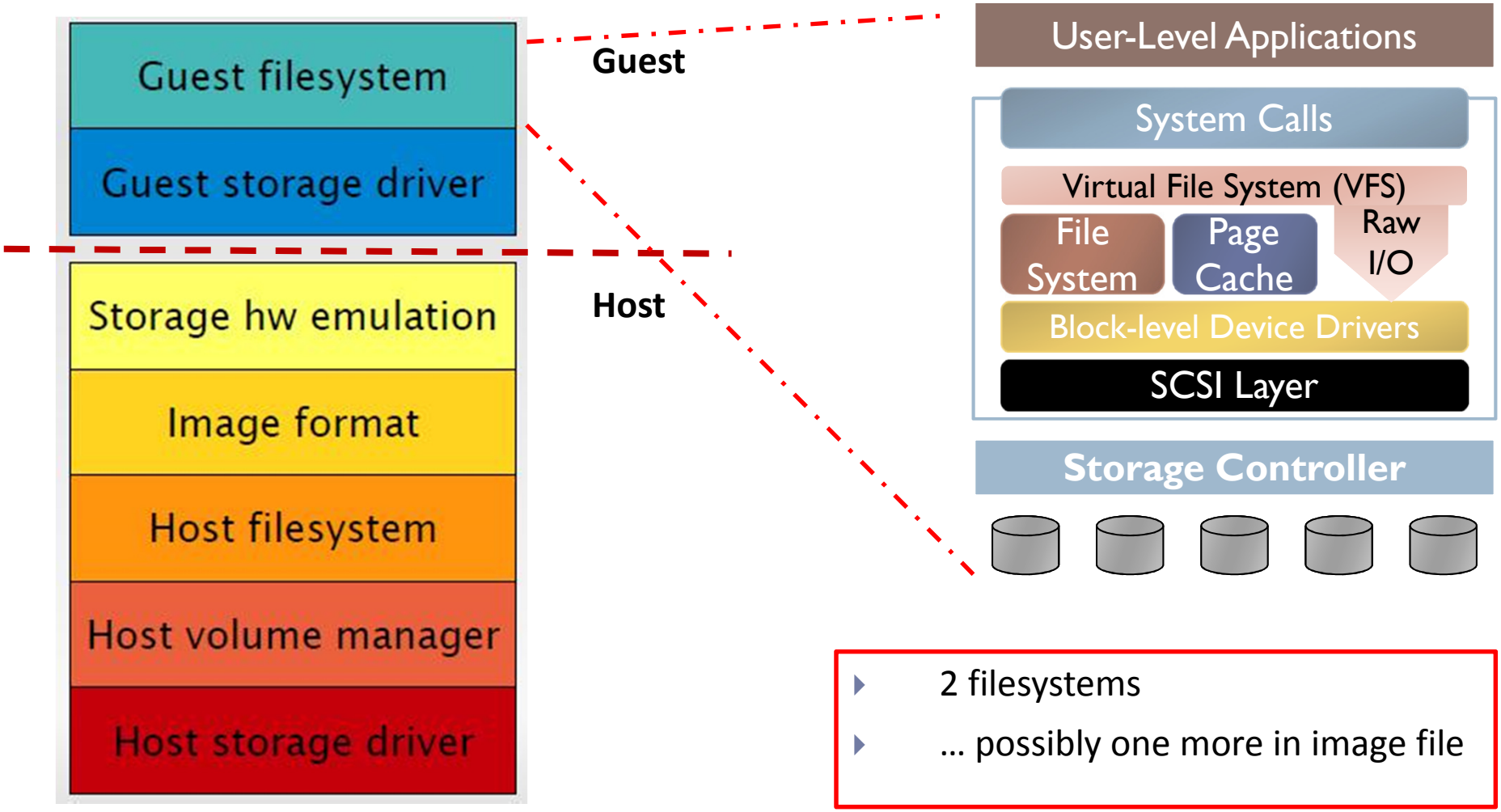
Virtual disks

- Exported by host
- Physical device/partition
- … or logical device
- … or file
  - Raw-format
  - Image format (e.g. : .qcow2, .vmdk, “raw” .img)
    - Features: compression, encryption, copy-on-write snapshots

I/O stack: Host

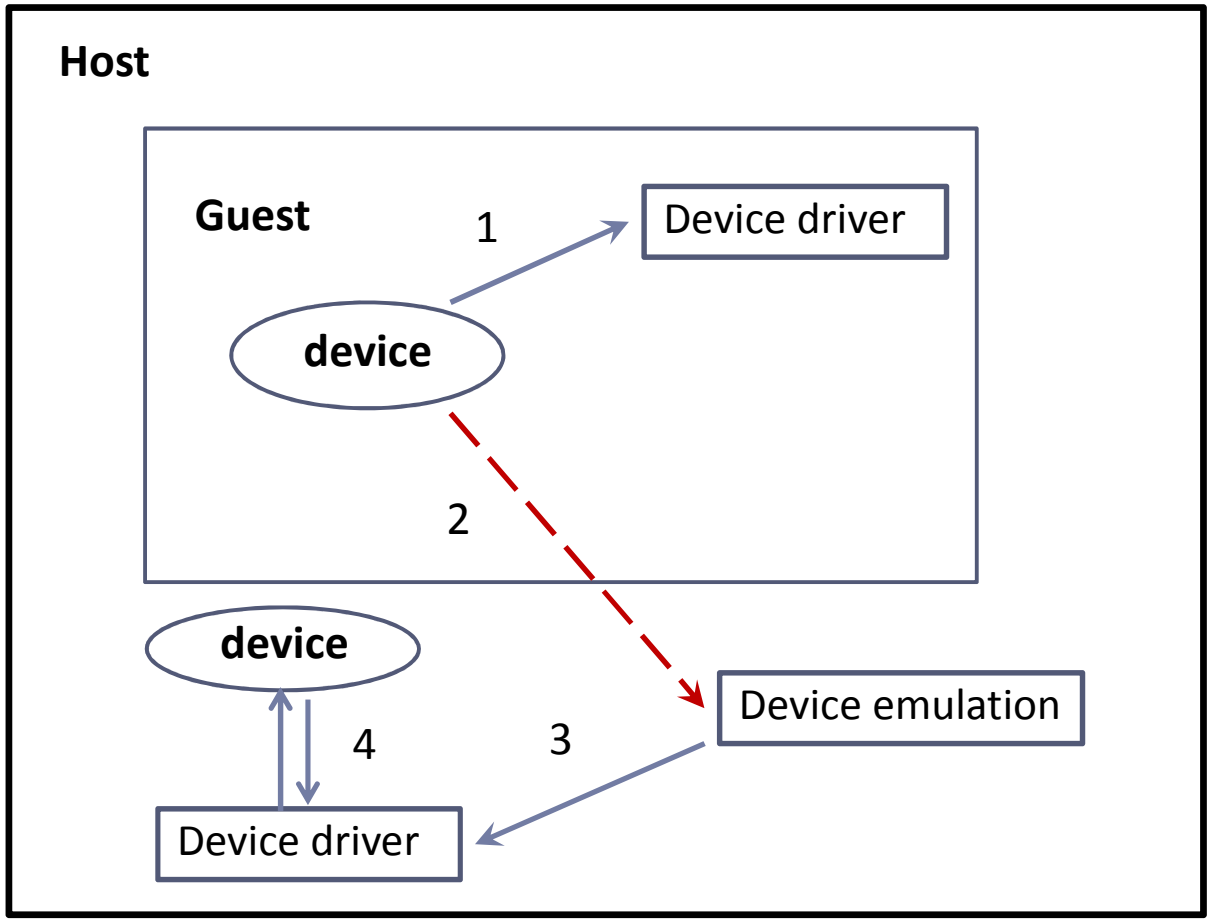


I/O stack: Host + Guest



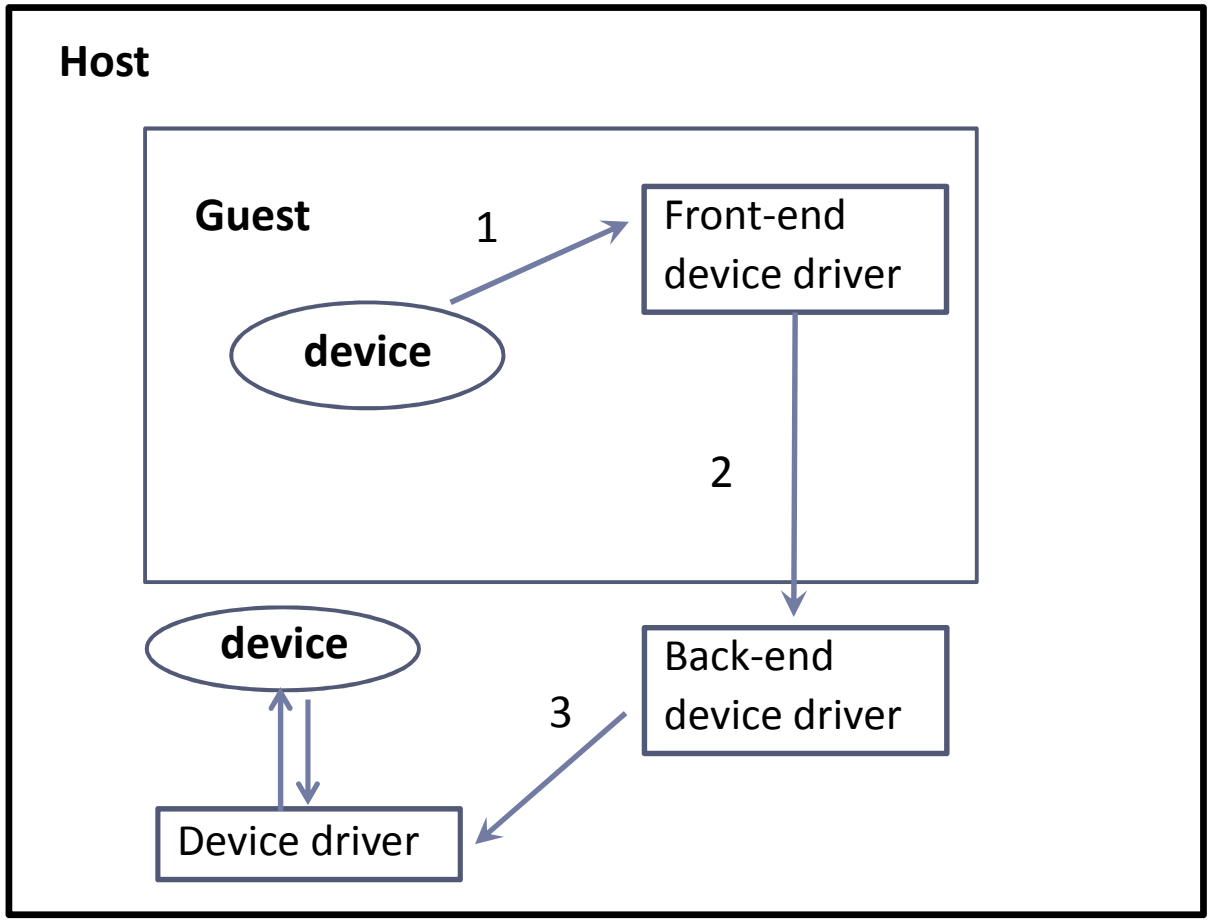
- 2 filesystems
- … possibly one more in image file

Device Emulation



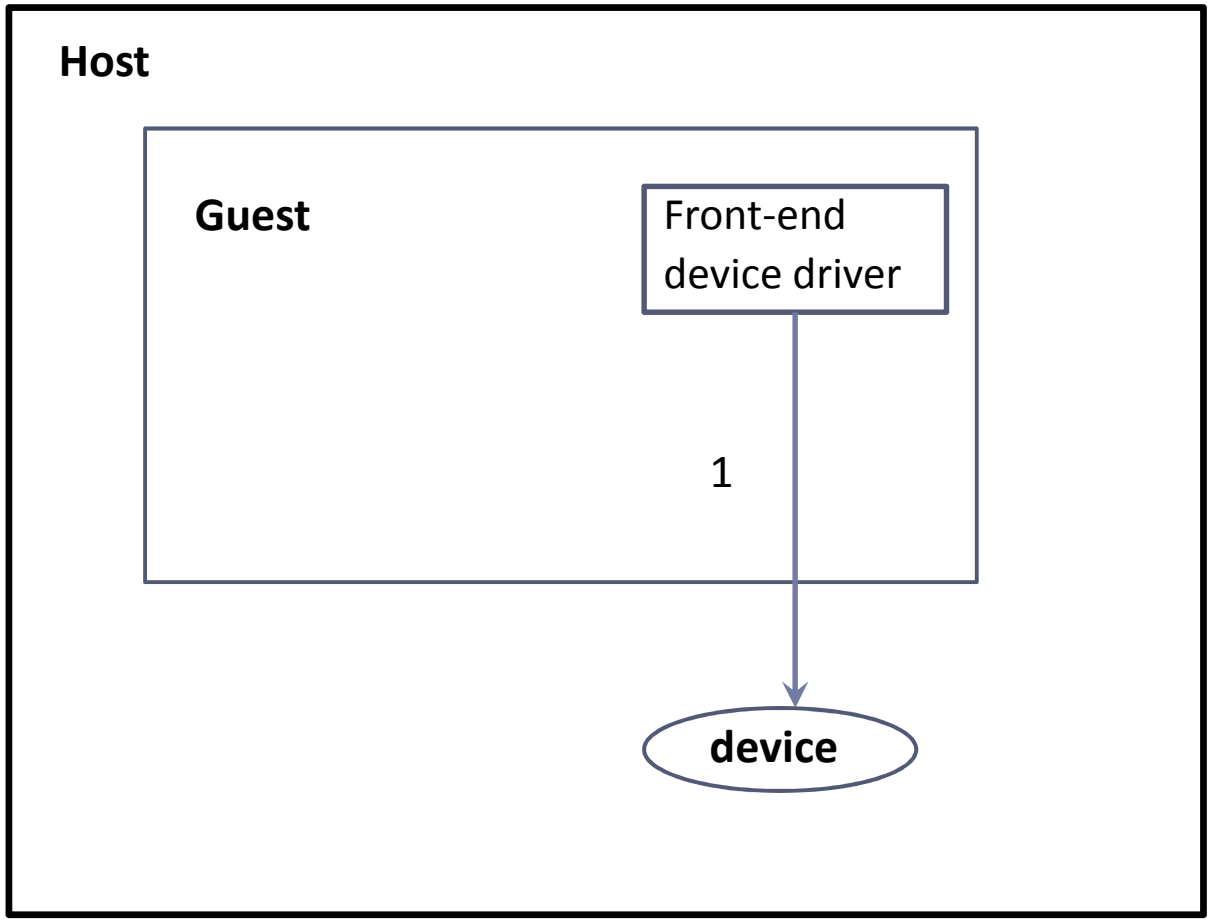
- Hypervisor emulates real devices → traps I/O accesses (PIO, MMIO)
- Emulation of DMA and interrupts

Para-virtualized drivers



- Hypervisor-specific virtual drivers in Guest
- Involvement of Hypervisor (Back-end)

Direct device assignment

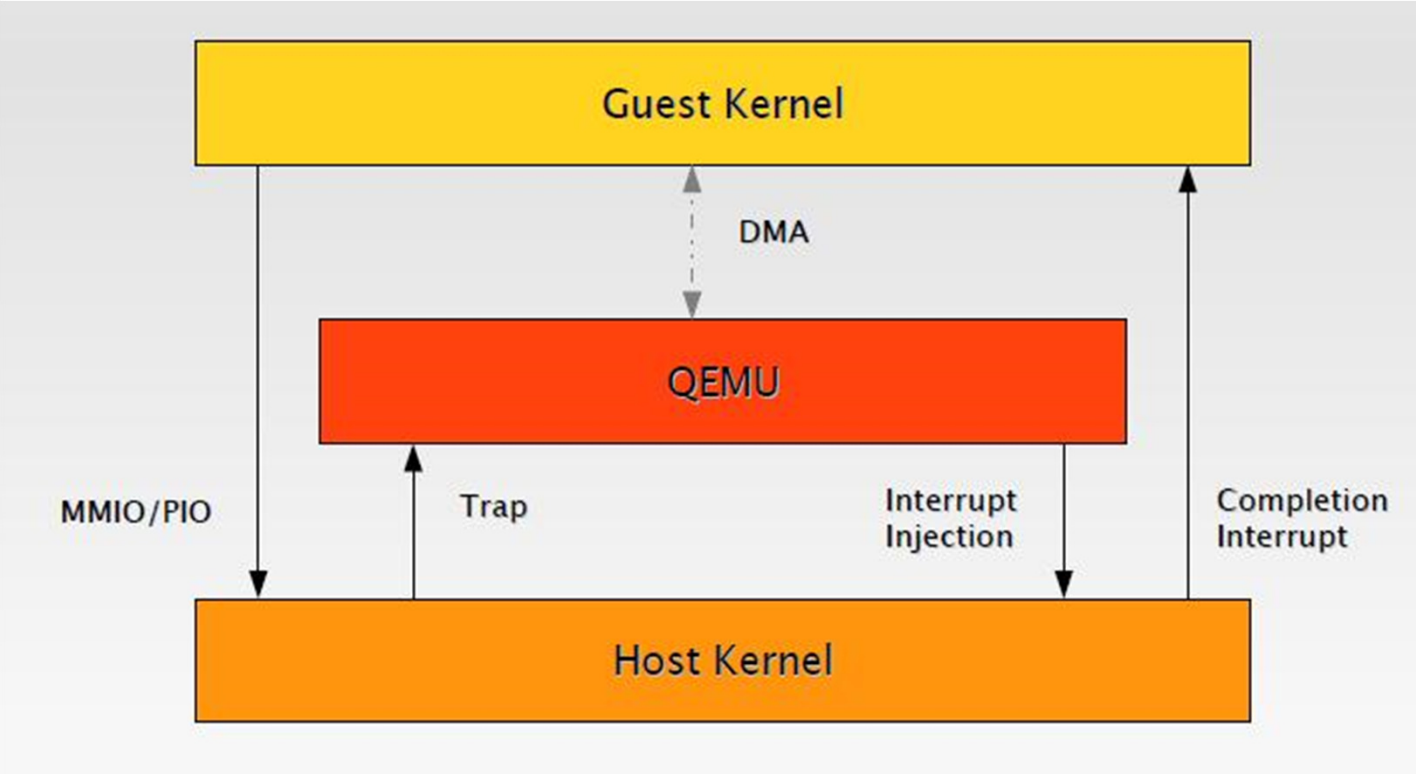


- Bypass of the Hypervisor to directly access devices
- Security & safety concerns !
- IOMMU for address translation & isolation (DMA restrictions)
- SR-IOV for shared access

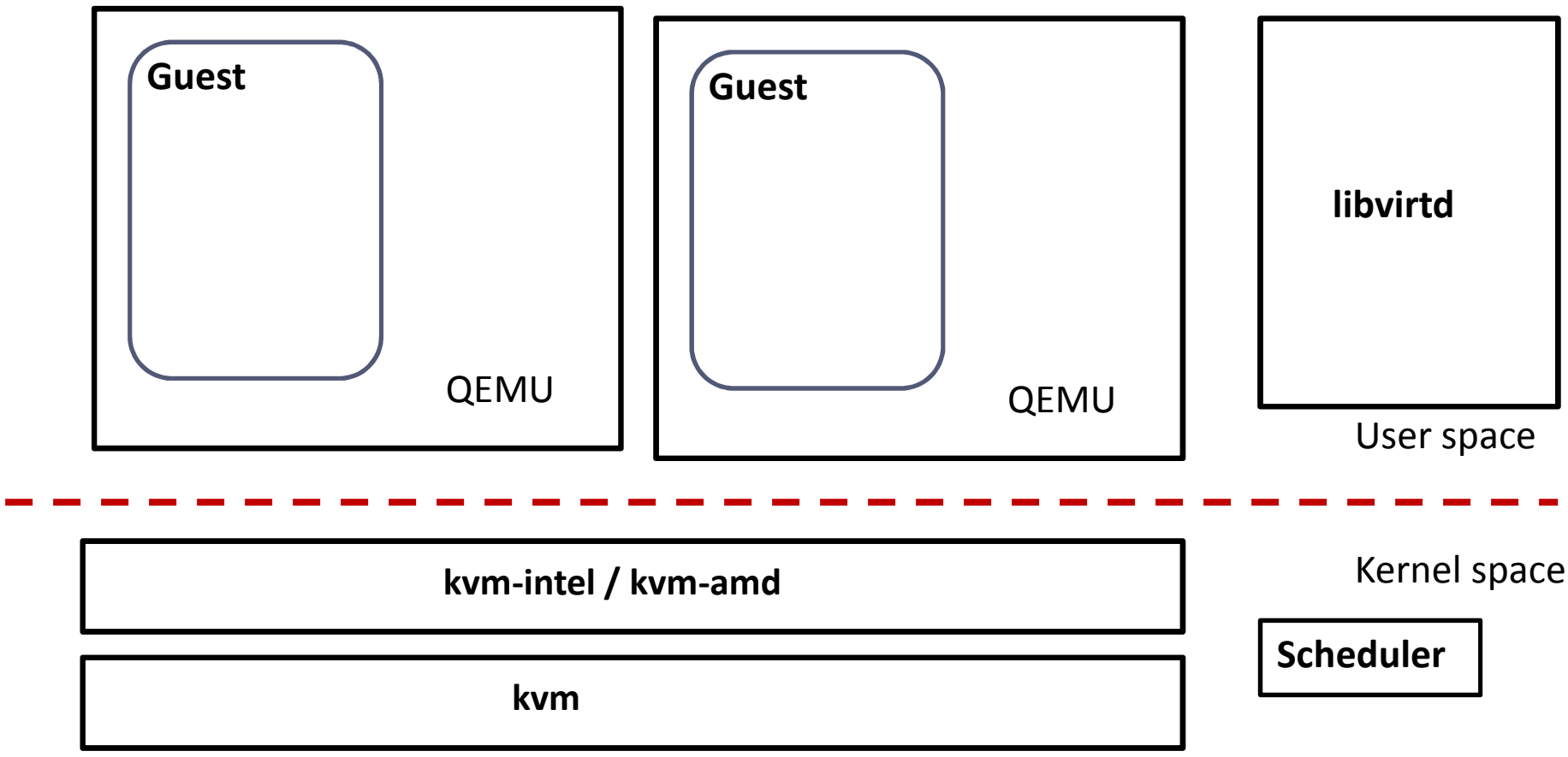
kvm guest initialization (command-line)

```
qemu-system-x86_64 -L /usr/local/kvm/share/qemu
-hda /mnt/scalusMar2013/01/scalusvm.img
-drive
file=/mnt/scalusMar2013/01/datavol.img,if=virtio,index=0
,cache=writethrough
-net nic -net user,hostfwd=tcp::12301-:22
-nographic
-m 4096 -smp 2
```

Virtualized I/O flow



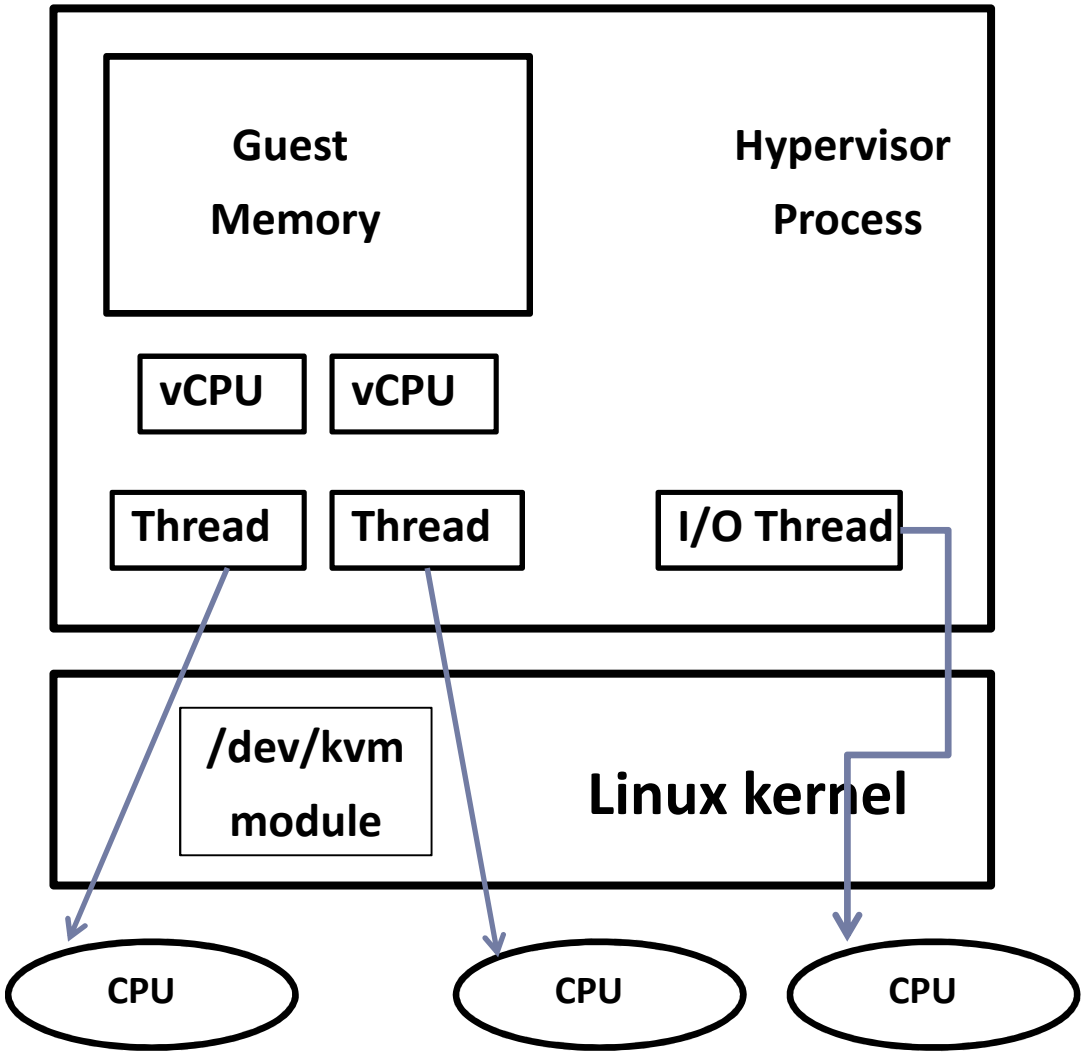
kvm run-time environment



kvm execution model

- Processes create virtual machines
  - A process together with /dev/kvm is in essence the Hypervisor
- VMs contain memory, virtual CPUs, and (in-kernel) devices
- Guest ("physical") memory is part of the (virtual) address space of the creating process
  - Virtual MMU+TLB, and APIC/IO-APIC (in-kernel)
  - Machine instruction interpreter (in-kernel)
- vCPUs run in process context (i.e. as threads)
  - To the host, the process that started the guest _is_ the guest!

Run-time view of a kvm guest

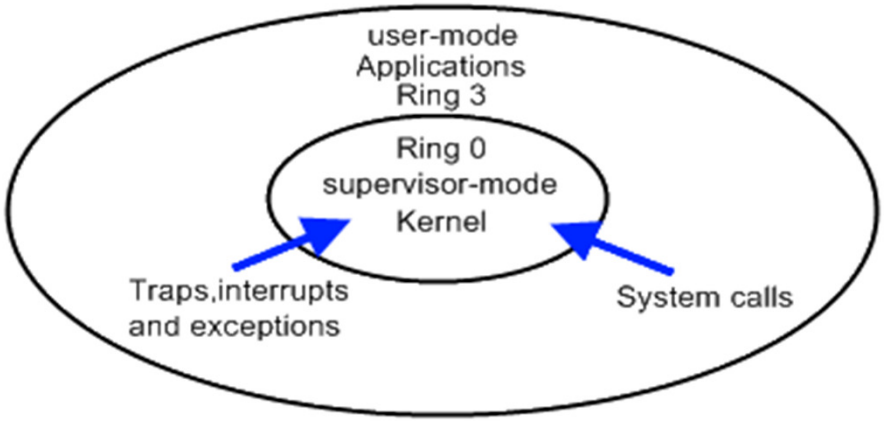


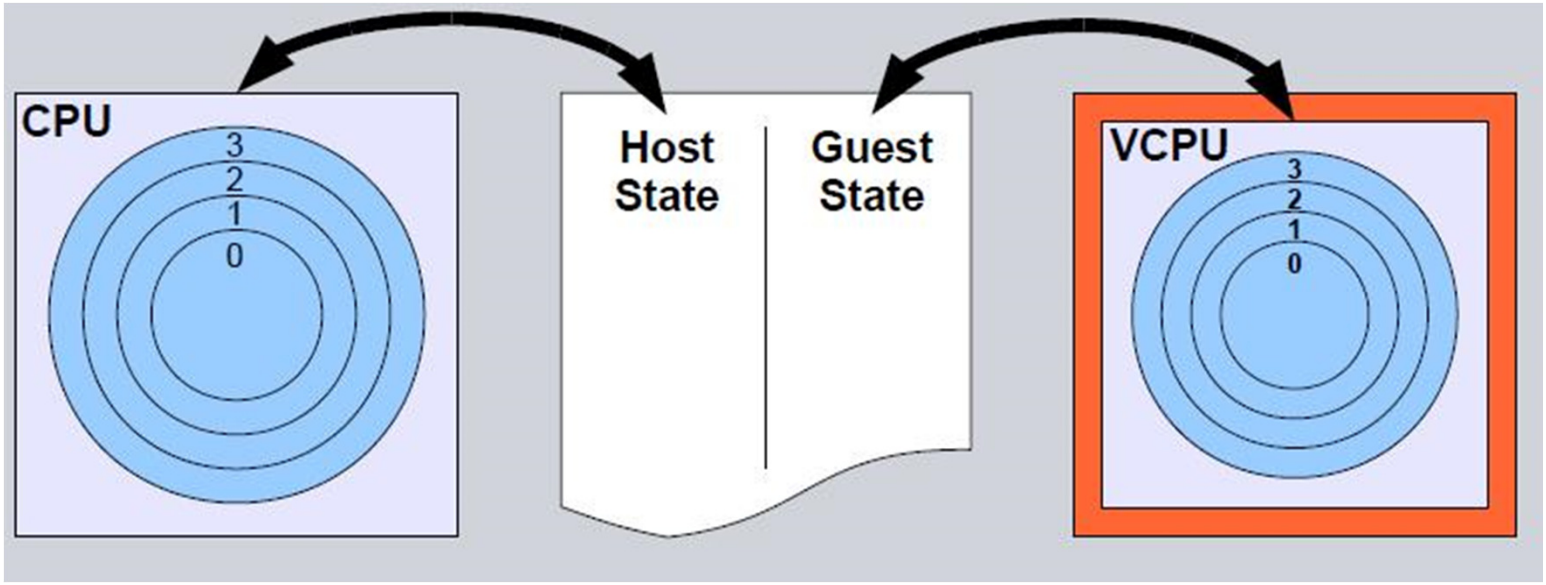
- Guest – Host switch via scheduler
- Use of Linux subsystems: scheduler, memory management, …
- Re-use of user-space tools
  - VM images
  - Network configuration

Virtualization of (x86) instruction set

- Trap-and-Emulate
- Alternatives:
  - Binary translation (e.g. early VMware), with “code substitution”
  - CPU para-virtualization (e.g. Xen), with “hyper-calls”
  - Hardware-assisted virtualization
    - Objective: reduce need for binary translation and emulation
    - Host + Guest state, per core
    - “World switch”
    - Intel VT-x, AMD SVM
    - Intel EPT (extended page tables), AMD NPT (nested page tables)
      - Add a level of translation for guest physical memory
      - R/W/X bits → can generate faults for physical memory accesses
    - vCPU-tagged address spaces → avoid TLB flushes

Virtualized CPU



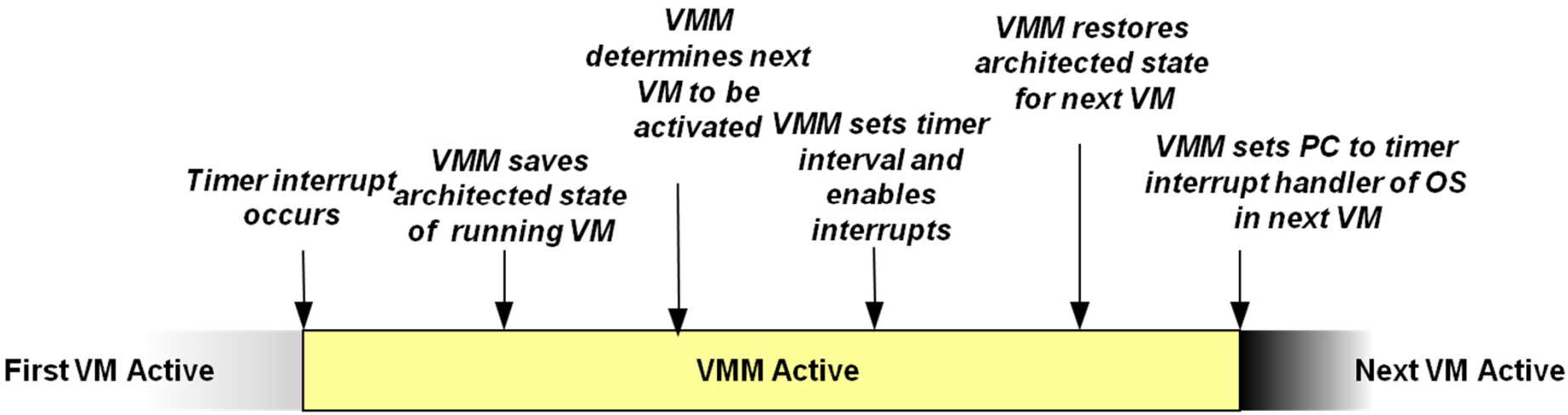


Trap-and-Emulate

- VMM and Guest OS :
  - System Call (ring 3)
    - CPU will trap to interrupt handler vector of VMM (ring 0).
    - VMM jump back into guest OS (ring 1).
  - Hardware Interrupt
    - Hardware makes CPU trap to interrupt handler of VMM.
    - VMM jump to corresponding interrupt handler of guest OS.
  - Privileged Instruction
    - Running privilege instructions in guest OS will trap to VMM for instruction emulation.
    - After emulation, VMM jumps back to guest OS.

Switching between VMs

- Timer Interrupt in running VM.
- Context switch to VMM.
- VMM saves state of running VM.
- VMM determines next VM to execute.
- VMM sets timer interrupt.
- VMM restores state of next VM.
- VMM sets PC to timer interrupt handler of next VM.
- Next VM active.



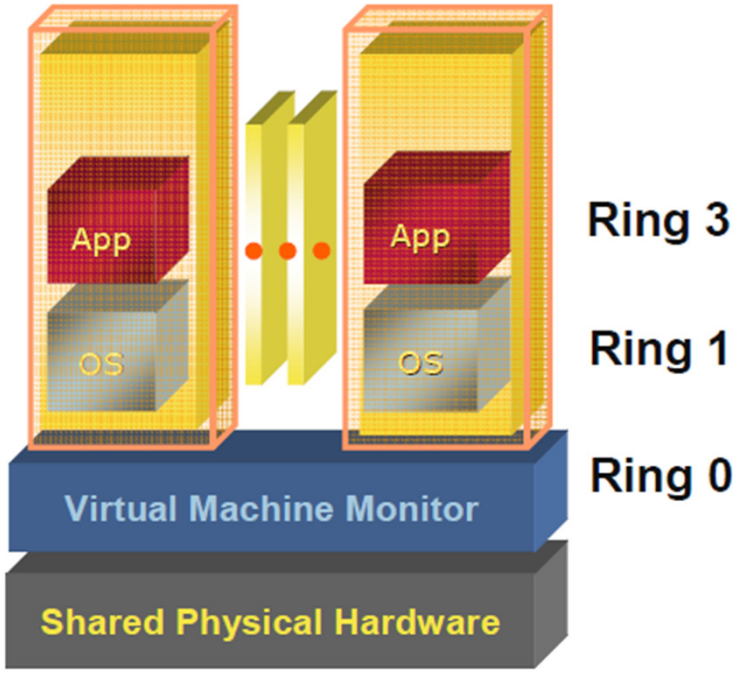
VM state management

- VMM will hold the system states of all VMs in memory
- When VMM context-switches from one VM to another:
  - Write the register values back to memory
  - Copy the register values of next guest OS to CPU registers.

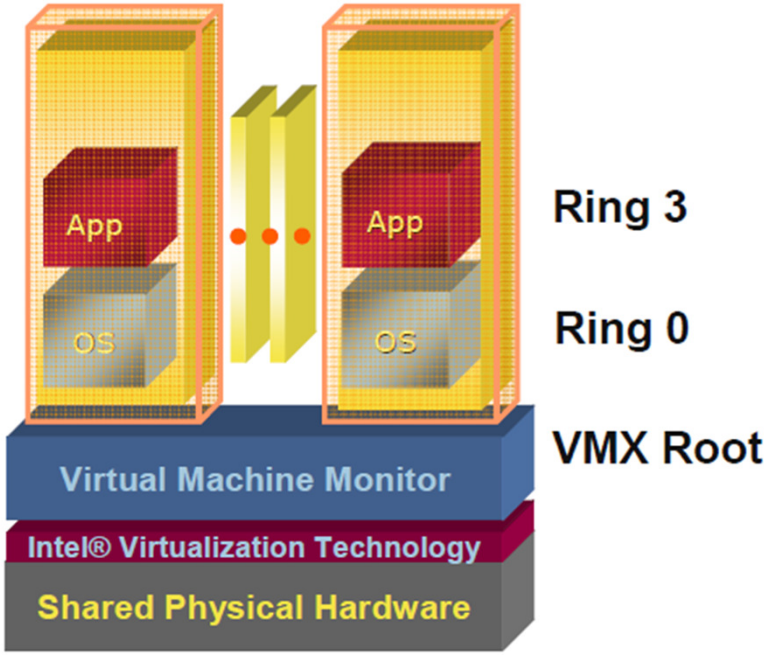
Intel VT-x

- VMX Root Operation (Root Mode)
  - All instruction behaviors in this mode are no different to traditional ones.
  - All legacy software can run in this mode correctly.
  - VMM should run in this mode and control all system resources.
- VMX Non-Root Operation (Non-Root Mode)
  - All sensitive instruction behaviors in this mode are redefined.
  - The sensitive instructions will trap to Root Mode.
  - Guest OS should run in this mode and be fully virtualized through typical "*trap and emulation model*".

Intel VT-x



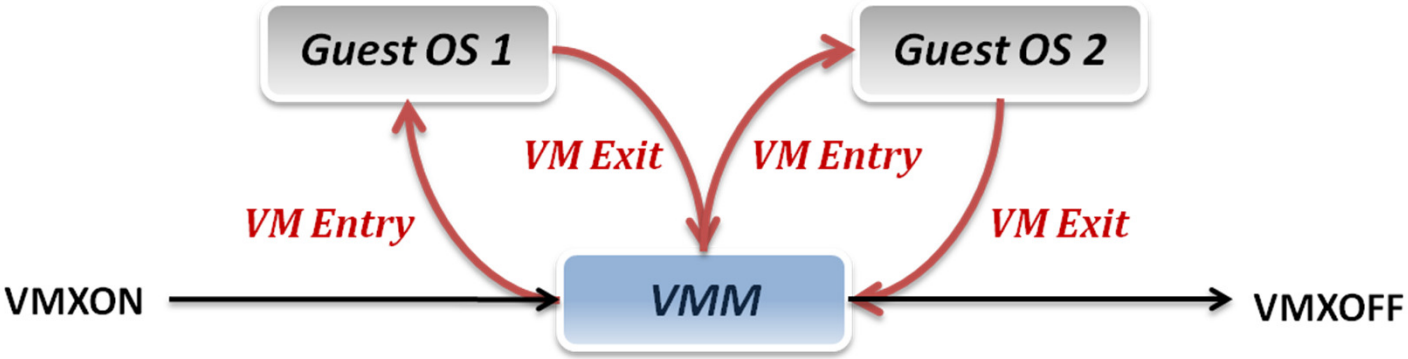
- VMM de-privileges the guest OS into Ring 1, and takes up Ring 0
- OS un-aware it is not running in traditional ring 0 privilege
- Requires compute intensive SW translation to mitigate



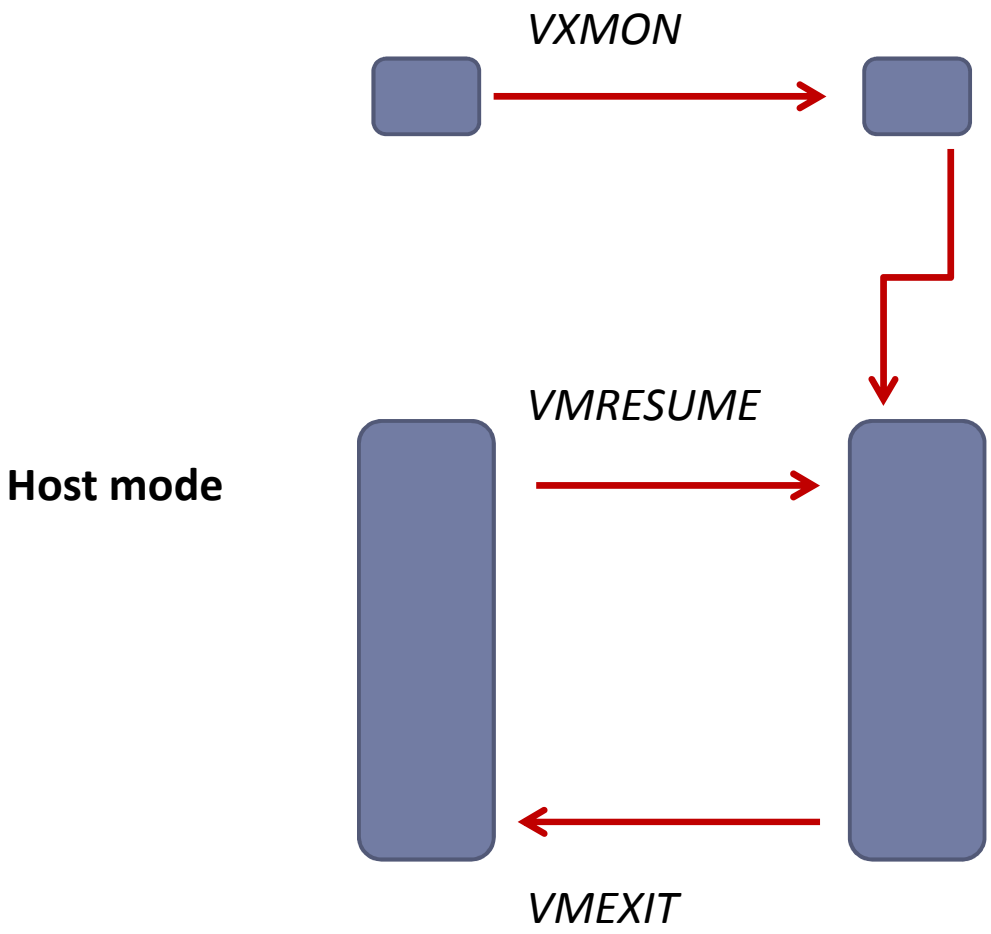
- VMM has its own privileged level where it executes
- No need to de-privilege the guest OS
- OSes run directly on the hardware

Context Switch

- VMM switch different virtual machines with Intel VT-x :
  - VMXON/VMXOFF
    - These two instructions are used to turn on/off CPU Root Mode.
  - VM Entry
    - This is usually caused by the execution of **`VMLAUNCH/VMRESUME`** instructions, which will switch CPU mode from Root Mode to Non-Root Mode.
  - VM Exit
    - This may be caused by many reasons, such as hardware interrupts or sensitive instruction executions.
    - Switch CPU mode from Non-Root Mode to Root Mode.



Intel vmx



Expensive transitions:

- VM entry/exit
- Mode switch (world switch)

Some “exits” are heavier than others

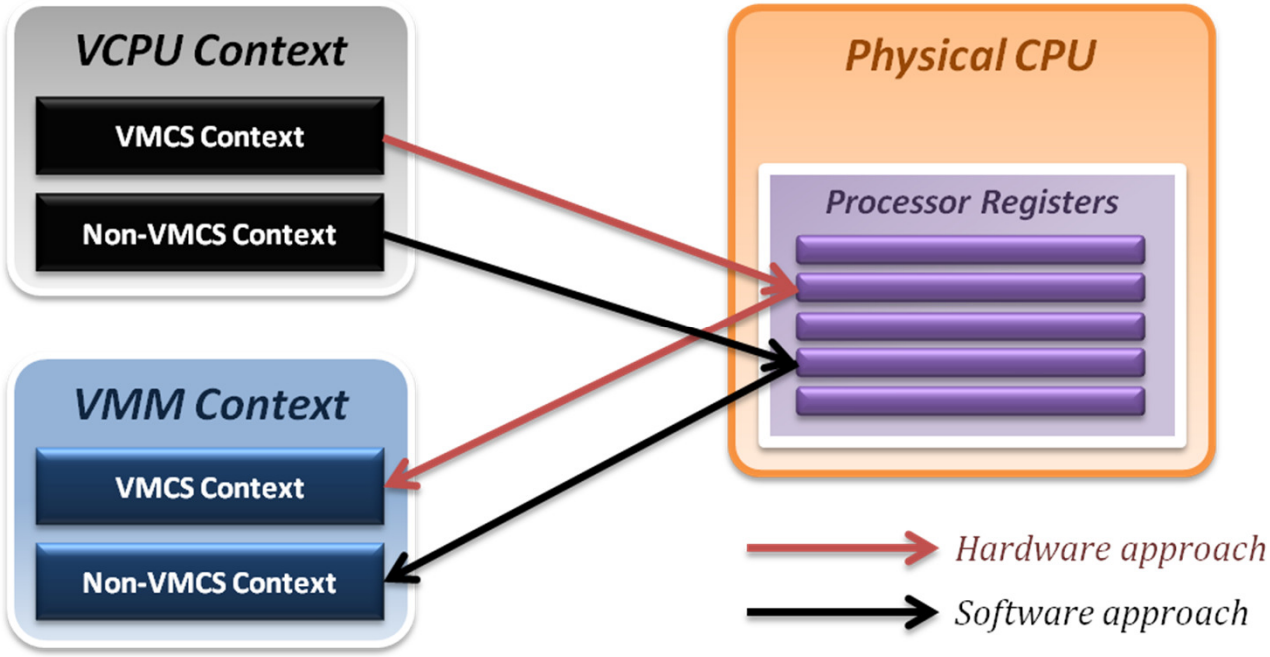
VMCS: *Virtual Machine Control Structure*

- State Area
  - Store host OS system state when VM-Entry.
  - Store guest OS system state when VM-Exit.
- Control Area
  - Control instruction behaviors in Non-Root Mode.
  - Control VM-Entry and VM-Exit process.
- Exit Information
  - Provide the VM-Exit reason and some hardware information.

- When VM Entry or VM Exit occur, CPU will automatically read/write corresponding information into VMCS.

System state management

- Binding virtual machine to virtual CPU
  - VCPU (Virtual CPU) contains two parts:
    - VMCS: maintains virtual system states, which is approached by hardware.
    - Non-VMCS: maintains other non-essential system information, which is approach by software.
  - VMM needs to handle Non-VMCS part.



EPT: Extended page Table

- Instead of walking along with only one page table hierarchy, EPT implements one additional page table hierarchy:
  - One page table is maintained by guest OS, which is used to generate guest physical address.
  - Another page table is maintained by VMM, which is used to map guest physical address to host physical address.

- For each memory access operation, the EPT MMU will directly get guest physical address from guest page table, and then get host physical address by the VMM mapping table automatically.

Cost of VM exits

[source: Landau, et al, WIOV 2011]

Exit Type	Number of Exits	Cycle Cost/Exit
External interrupt	8961	363,000
I/O instruction	10042	85,000
APIC access	691249	18,000
EPT violation	645	12,000

- netperf client run on 1 GbE, with para-virtualized NIC
- Total run: ~7.1×10^{10} cycles vs ~5.2×10^{10} cycles for bare-metal
- **35% slow-down** due to the guest and hypervisor sharing the same core

kvm components

- "hypervisor" module (/dev/kvm character device)
  - … heavily relies on Linux kernel subsystems
  - ioctl() API calls for requests
  - 1 file descriptor per "resource":
    - System: VM creation, capabilities
    - VM: CPU initialization, memory management, interrupts
    - Virtual CPU (vCPU): access to execution state
- Platform emulator (qemu)
- Misc. modules & tools:
  - KSM (memory deduplication for VM images)
  - Libvirt.org tools (virsh, virt-manager)

kvm Hypervisor API

- ioctl() system calls to /dev/kvm
  - Create new VM
  - Provision memory to VM
  - Read/write vCPU registers
  - Inject interrupt into vCPU
  - “Run” a vCPU

Qemu main control loop

```
fd = open("/dev/kvm", O_RDWR);
ioctl(fd, KVM_CREATE_VM, …);
ioctl(fd, KVM_CREATE_VCPU, …);
for(;;) {
  ioctl(fd, KVM_RUN, …);
  switch(exit_reason) {
    case EXIT_REASON_IO_INSTRUCTION: … break;
    case EXIT_REASON_TASK_SWITCH: … break;
    case EXIT_REASON_PENDING_INTERRUPT: … break;
     …
  }
}
```

VM creation with kvm

```
int fd_kvm = open("/dev/kvm", O_RDWR);
int fd_vm = ioctl(fd_kvm, KVM_CREATE_VM, 0);

ioctl(fd_vm, KVM_SET_TSS_ADDR, 0xfffffffffffd000);
ioctl(fd_vm, KVM_CREATE_IRQCHIP, 0);
```

Adding physical memory to a VM with kvm

```
void *addr = mmap(NULL, 10 * MB, PROT_READ | PROT_WRITE,
MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
struct kvm_userspace_memory_region region = {
    .slot = 0,
    .flags = 0, // Can be Read Only
    .guest_phys_addr = 0x100000,
    .memory_size = 10 * MB,
    .userspace_addr = (__u64)addr
};
ioctl(fd_vm, KVM_SET_MEMORY_REGION, &region);
```

vCPU initialization with kvm

```
int fd_vcpu = ioctl(fd_vm, KVM_CREATE_VCPU, 0);
struct kvm_regs regs;

ioctl(fd_vcpu, KVM_GET_REGS, &regs);
regs.rflags = 0x02;
regs.rip = 0x0100f000;
ioctl(fd_vcpu, KVM_SET_REGS, &regs);
```

Running a vCPU in kvm

```
int kvm_run_size = ioctl(fd_kvm, KVM_GET_VCPU_MMAP_SIZE, 0);
// access to the arguments of ioctl(KVM_RUN)
struct kvm_run *run_state =
        mmap(NULL, kvm_run_size, PROT_READ | PROT_WRITE,
        MAP_PRIVATE, fd_vcpu, 0);
for (;;) {
        int res = ioctl(fd_vcpu, KVM_RUN, 0);
        switch (run_state->exit_reason) {
            // use run_state to gather informations about the exit
        }
}
```

Programmed I/O (PIO) in kvm

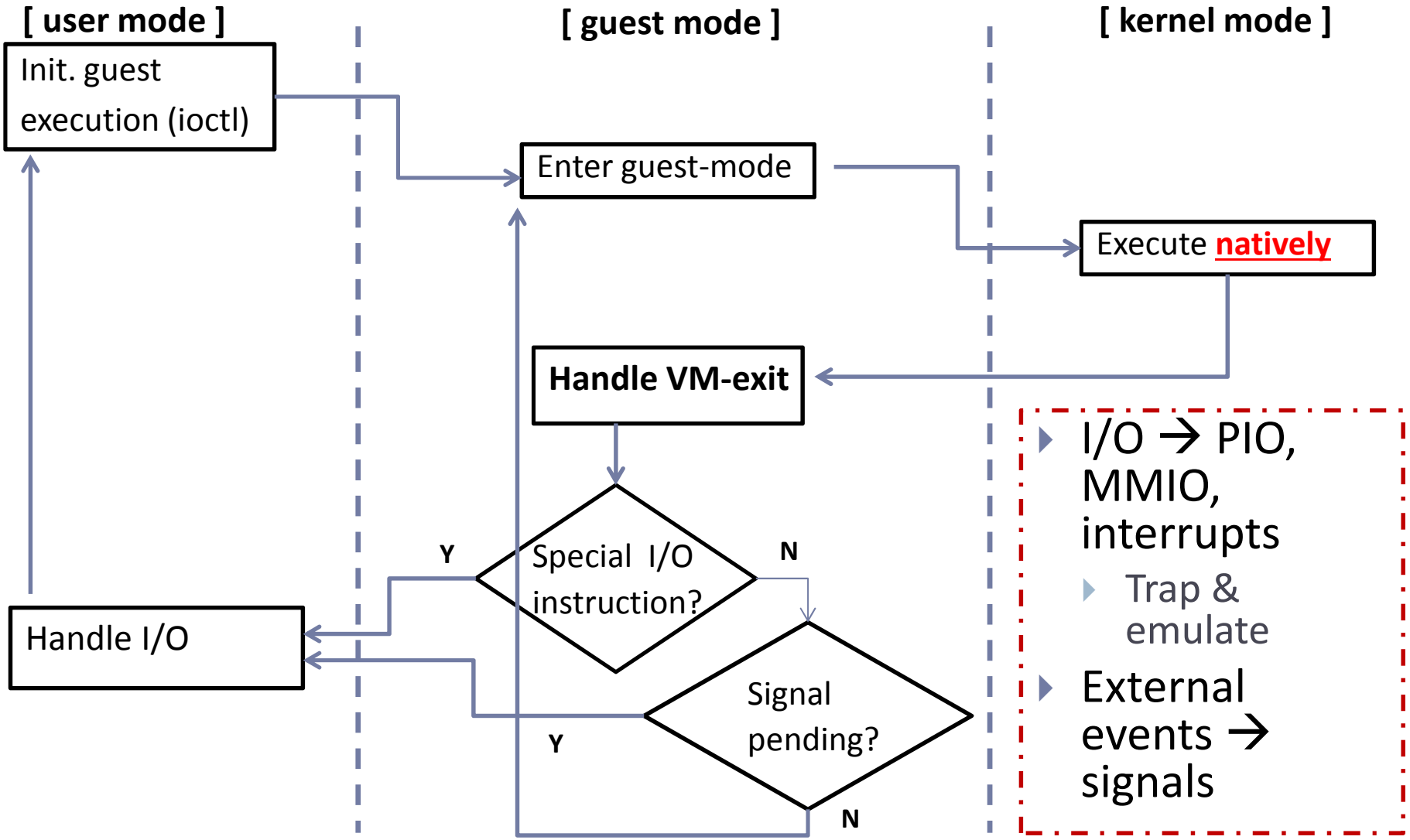
```
int kvm_run_size = ioctl(fd_kvm, KVM_GET_VCPU_MMAP_SIZE, 0);
// access to the arguments of ioctl(KVM_RUN)
struct kvm_run *run_state =
  mmap(NULL, kvm_run_size, PROT_READ | PROT_WRITE,
    MAP_PRIVATE, fd_vcpu, 0);
for (;;) {
        int res = ioctl(fd_vcpu, KVM_RUN, 0);
        switch (run_state->exit_reason) {
                // use run_state to gather informations about the exit
        }
}
```

Memory-mapped I/O (MMIO) in kvm

Exit reason : KVM_EXIT_MMIO

```
struct {
        __u64 phys_addr;
        __u8 data[8];
        __u32 len;
        __u8 is_write;
} mmio;
```

kvm guest execution flow

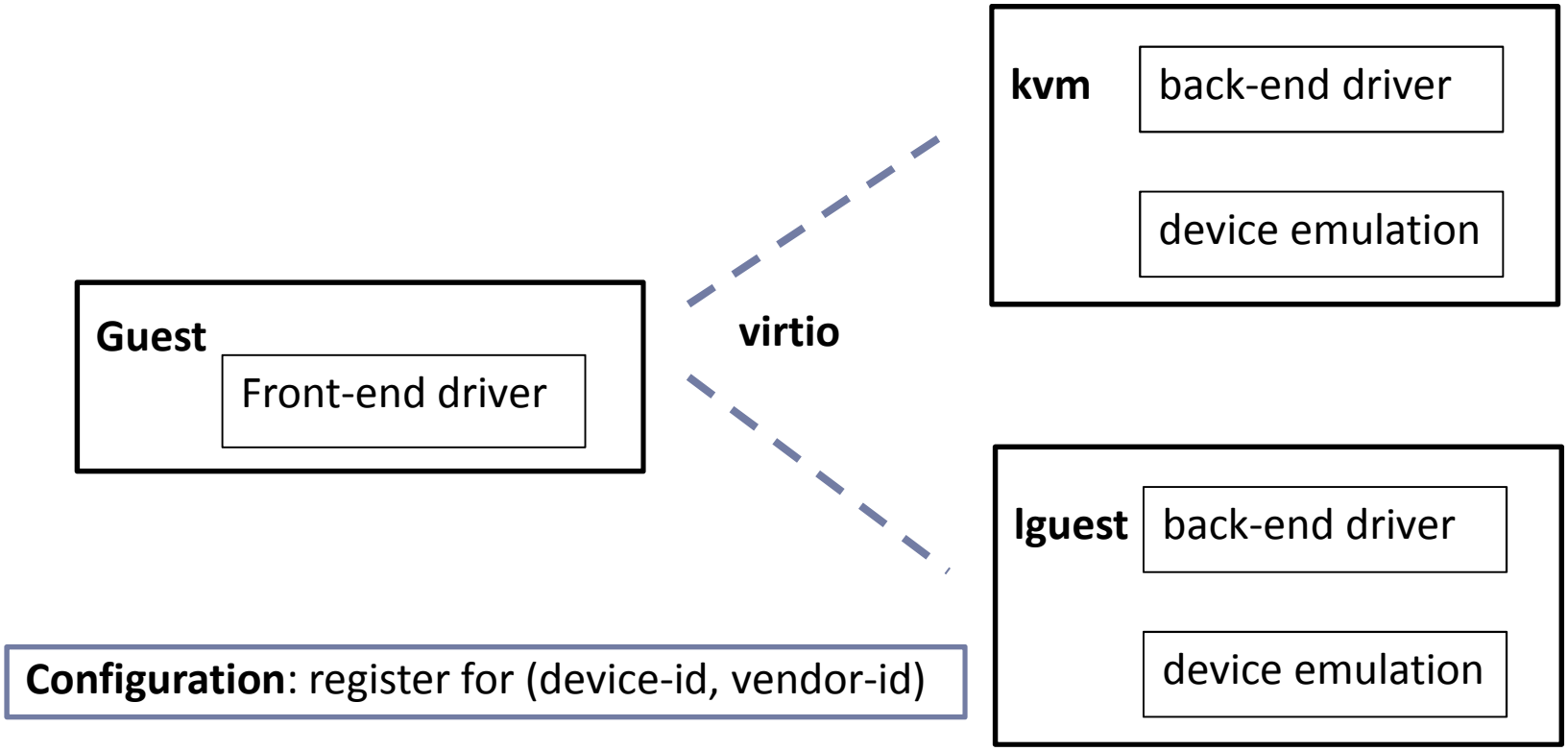


Device I/O in kvm

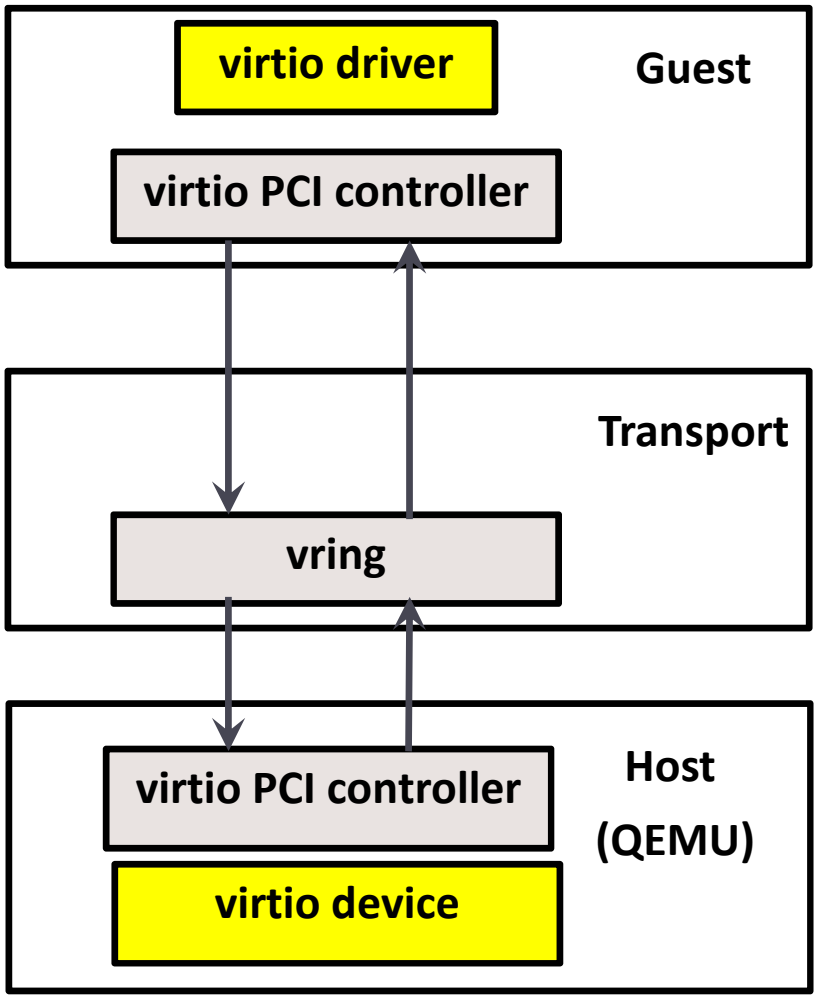
- Configuration via MMIO/PIO
- eventfd for events between host/guest
  - irqfd : host → guest
  - ioeventfd : guest → host
- virtio: abstraction for virtualized devices
  - Device types: PCI, MMIO
  - Configuration
  - Queues

virtio

- A family of drivers which can be adapted for various hypervisors, by porting a shim layer
- Related: VMware tools, Xen para-virtualized drivers
- Explicit separation of Drivers, Transport, Configuration



virtio architecture



Front-end

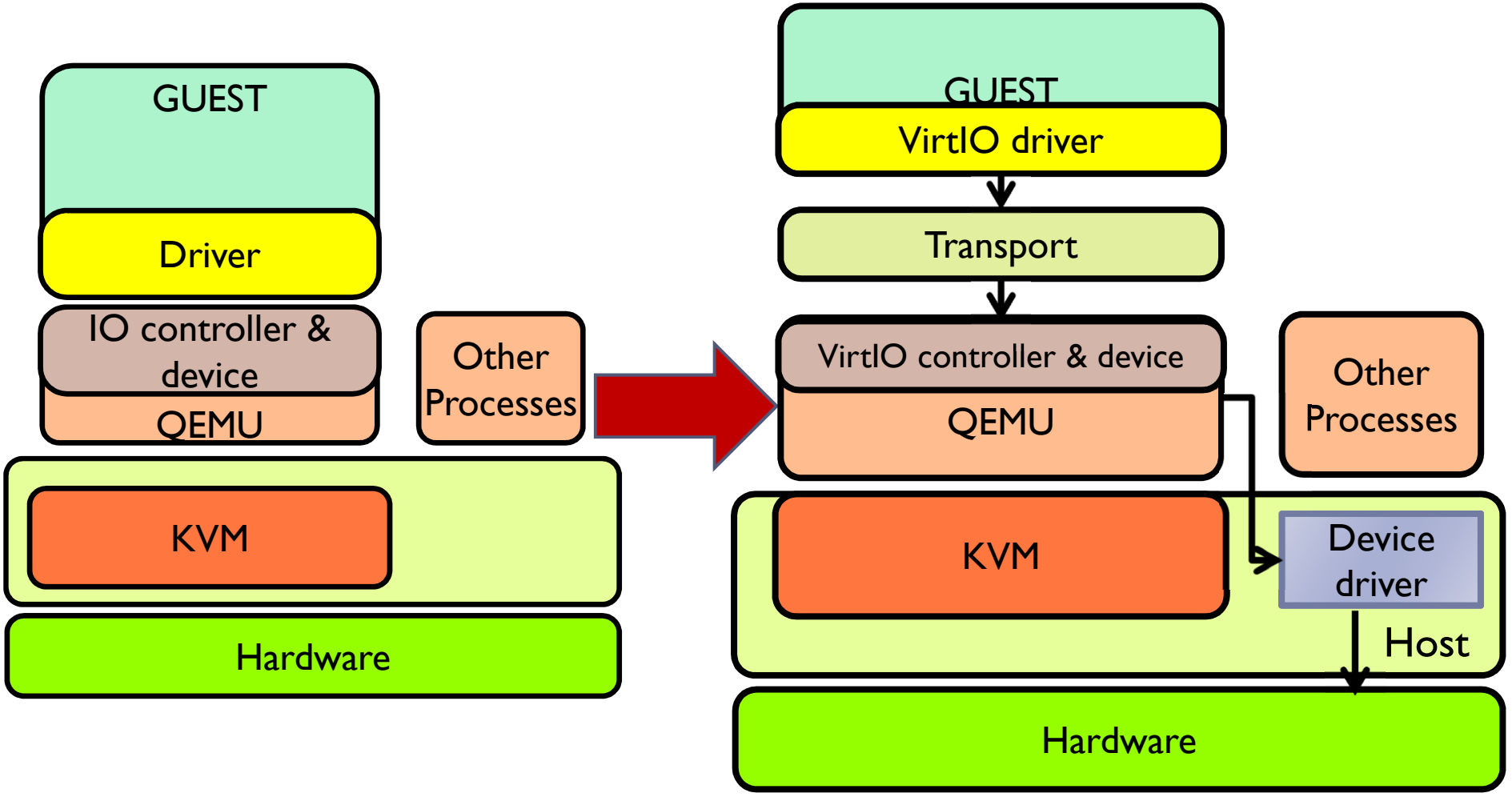
A kernel module in guest OS.
Accepts I/O requests from user process.
Transfer I/O requests to back-end.

Back-end

A device in QEMU.
Accepts I/O requests from front-end.
Perform I/O operation via physical device.

- **Virtqueues (per device)**
- **Vring (per virtqueue)**
- **Queue requests**

kvm with virtio



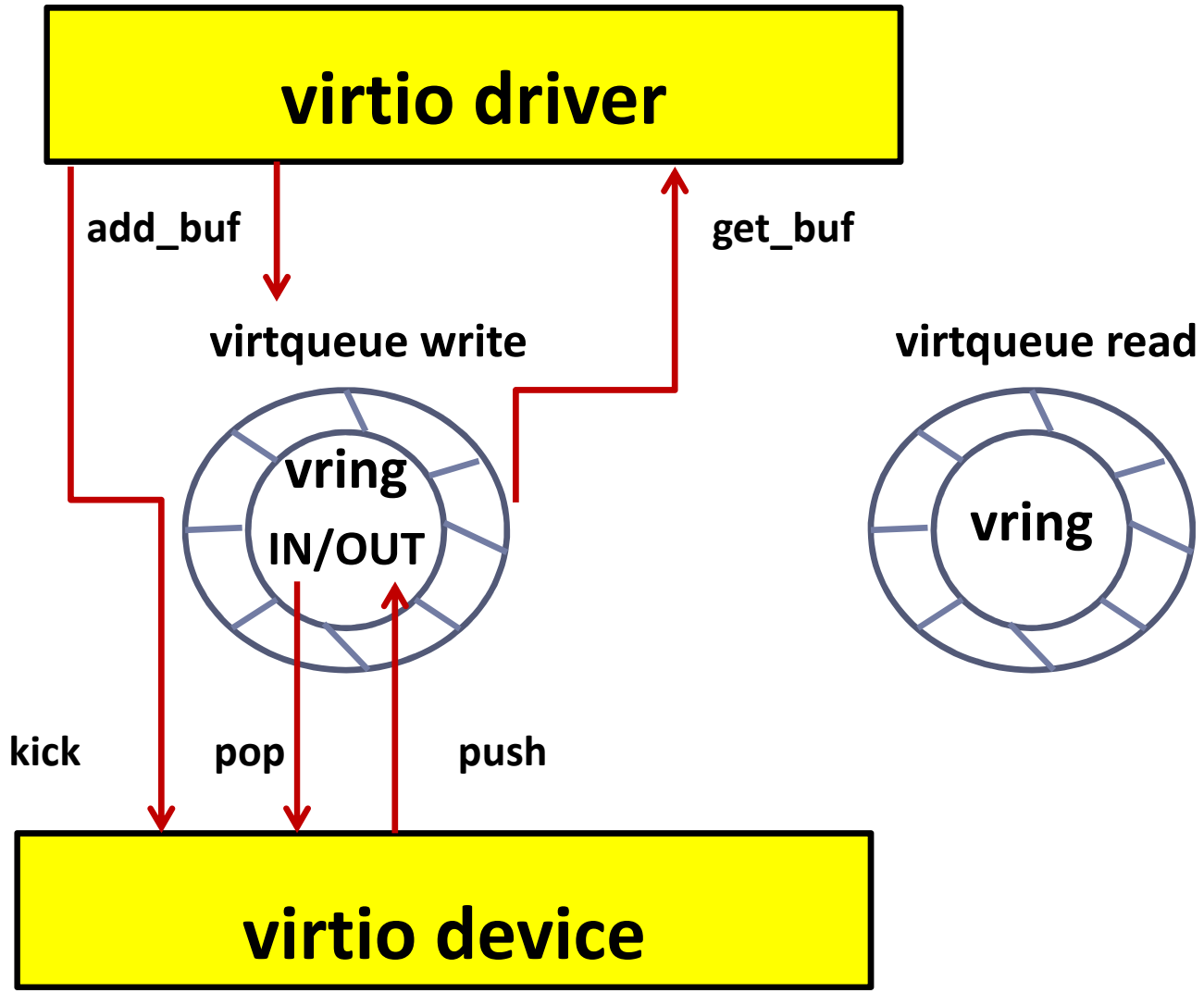
[kvm with 'default' device emulation]

[kvm with 'virtio' device handling]

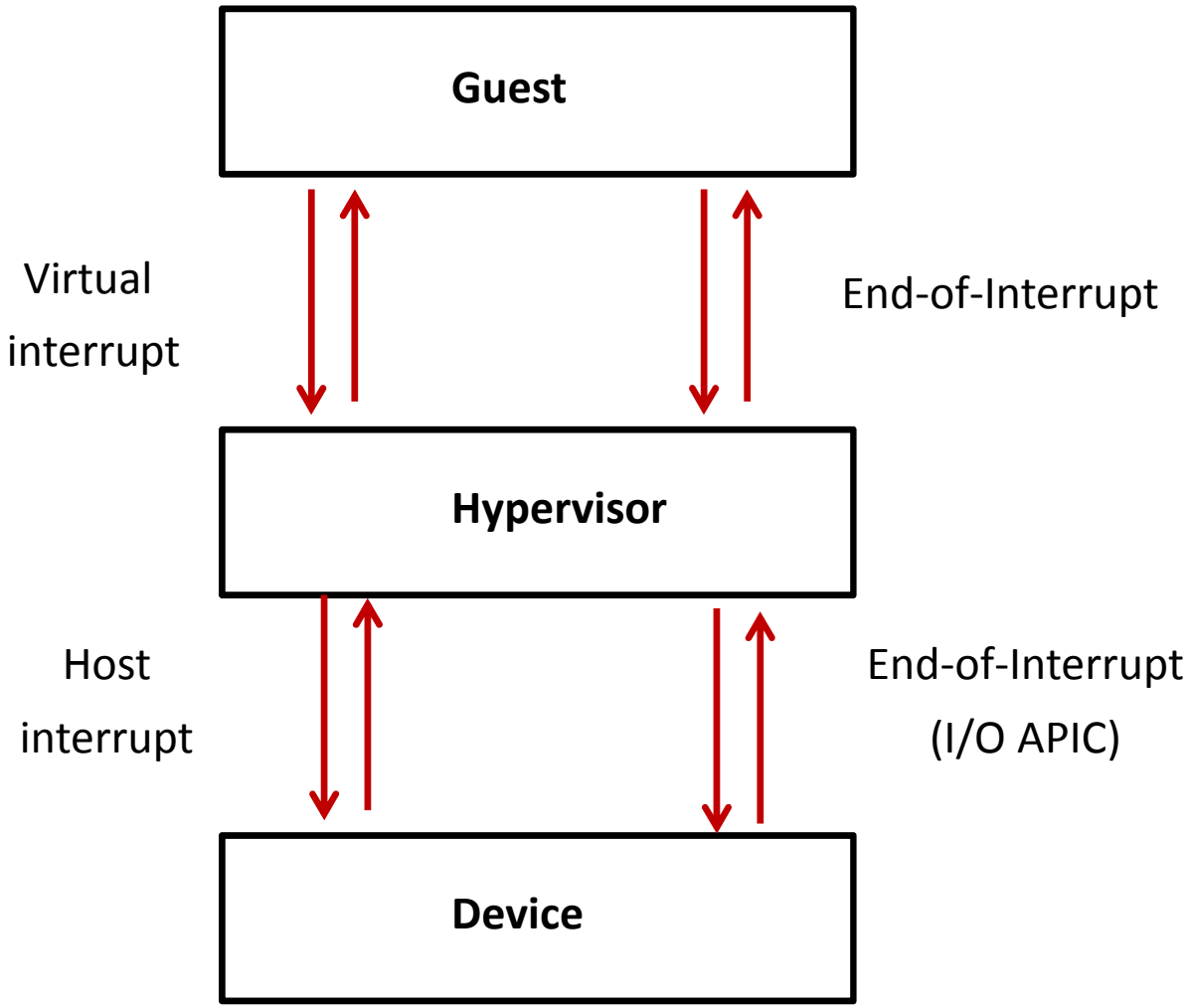
vring & virtqueue

- vring: transport implementation (ring-buffer)
  - shared (memory-mapped) between Guest and QEMU
    - Reduce the number of MMIOs
  - published & used buffers
  - descriptors
- virtqueue API:
  - add_buf: expose buffer to other end
  - get_buf: get next used buffer
  - kick: (after add_buf) notify QEMU to handle buffer
  - disable_cb, enable_cb: disable/enable callbacks
- “buffer” := scatter/gather list → (address, length) pairs
- QEMU: virtqueue_pop, virtqueue_push
- virtio-blk: 1 queue
- virtio-net: 2 queues

virtio processing flow

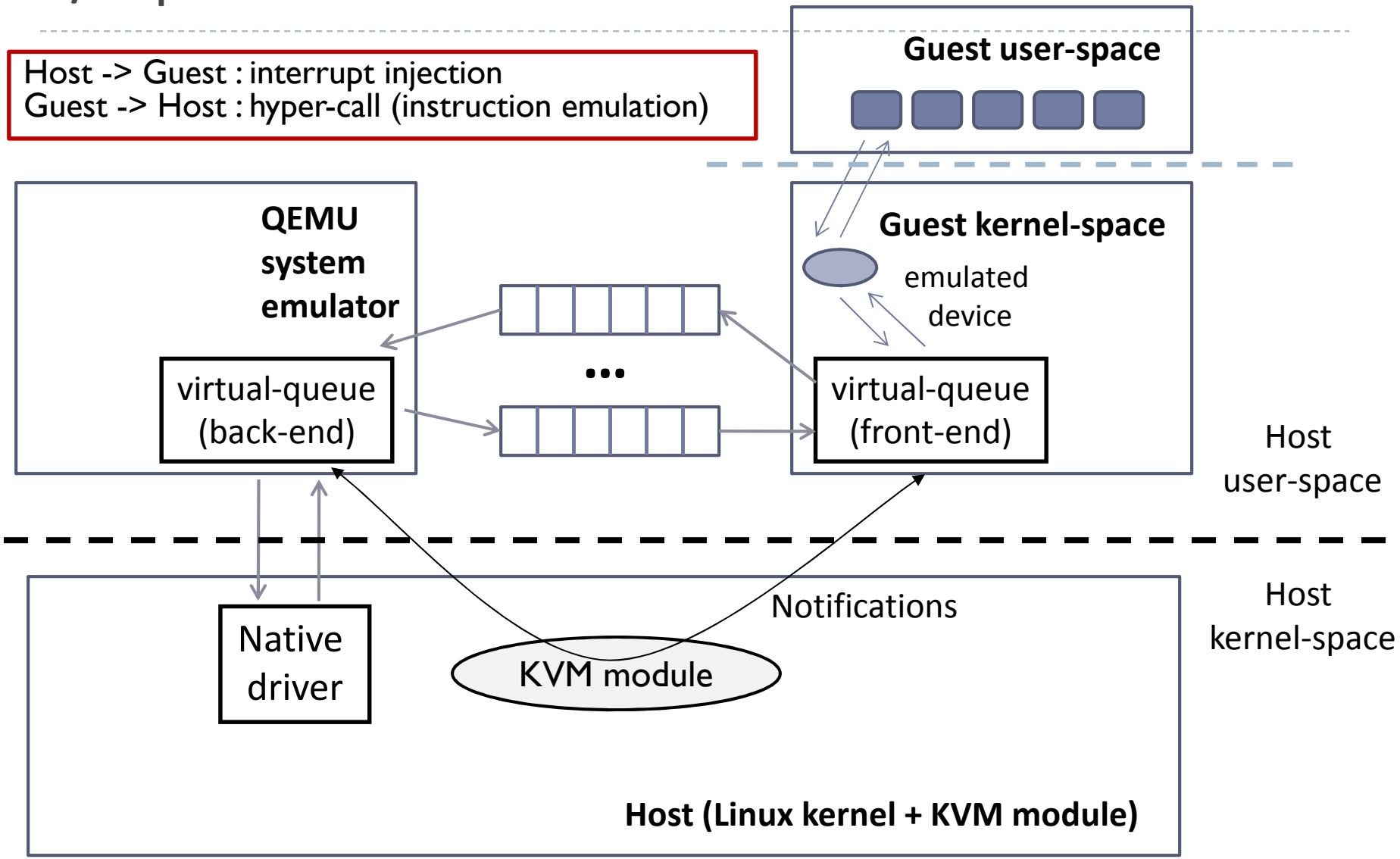


Virtual interrupts



At least 2 exits:
- Delivery
- Completion signal

I/O path via virtio

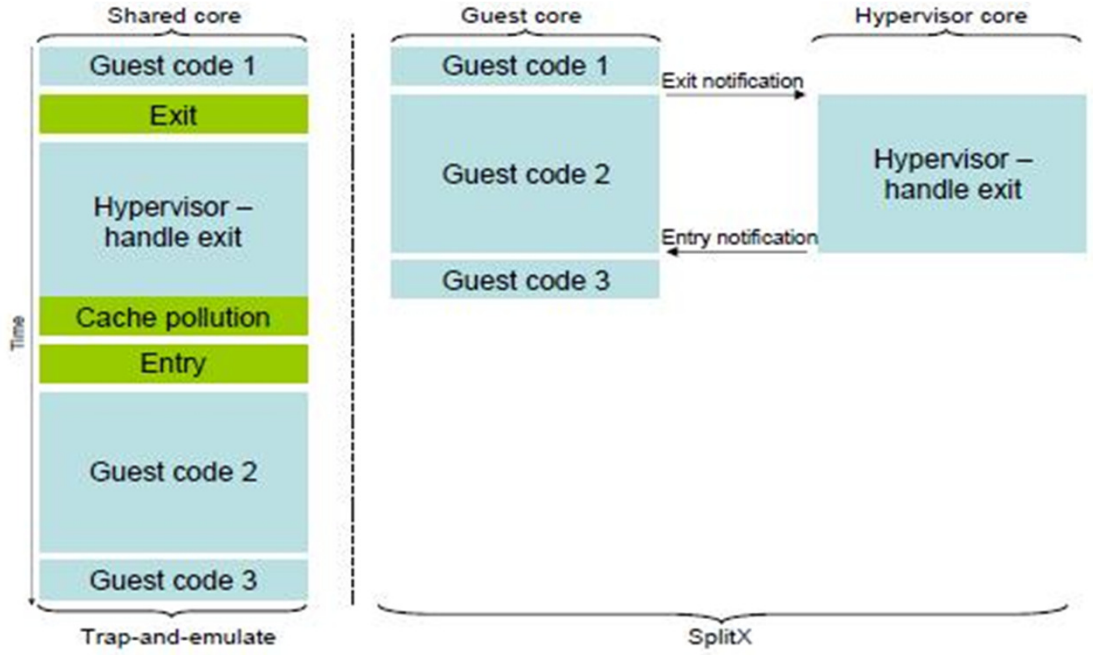


vhost-net

- Guest networking based on in-kernel assists
  - virtio + ioeventfd + irqfd
- Avoid “heavy” VM exits, and reduce packet copying

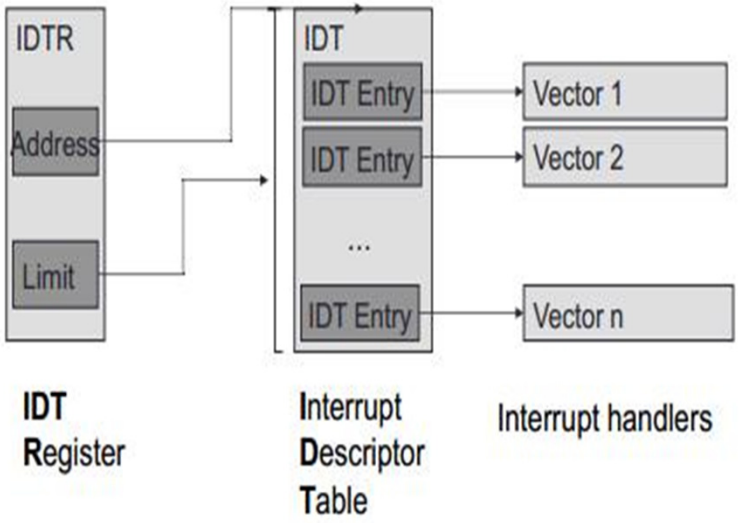
Acceleration of VMs (SplitX)

- Cost of an VM exit → 3 components
  - Direct (CPU "World switch")
  - Synchronous (due to exit processing in hypervisor)
  - Indirect (slowdown from having 2 contexts on same core → cache pollution)

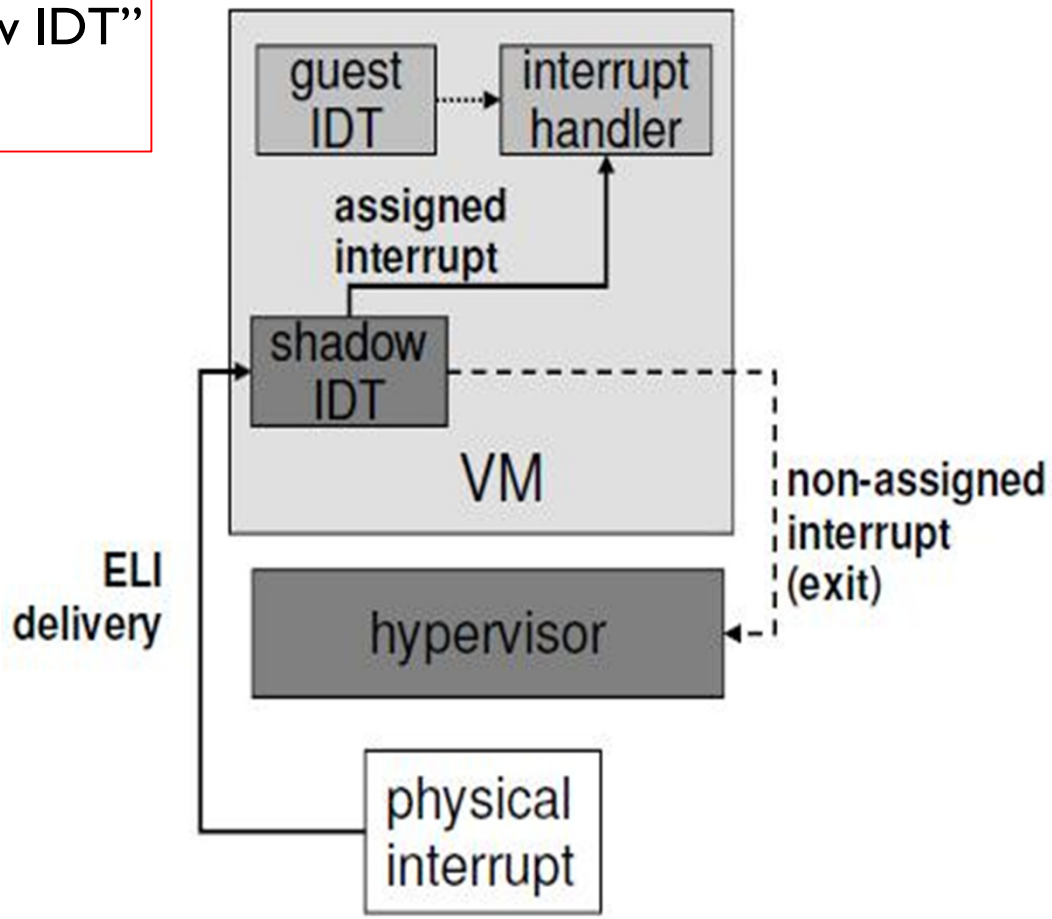


Acceleration of VMs (ELI)

Instead of running the guest with its own IDT, run the guest with "shadow IDT" … prepared by the host



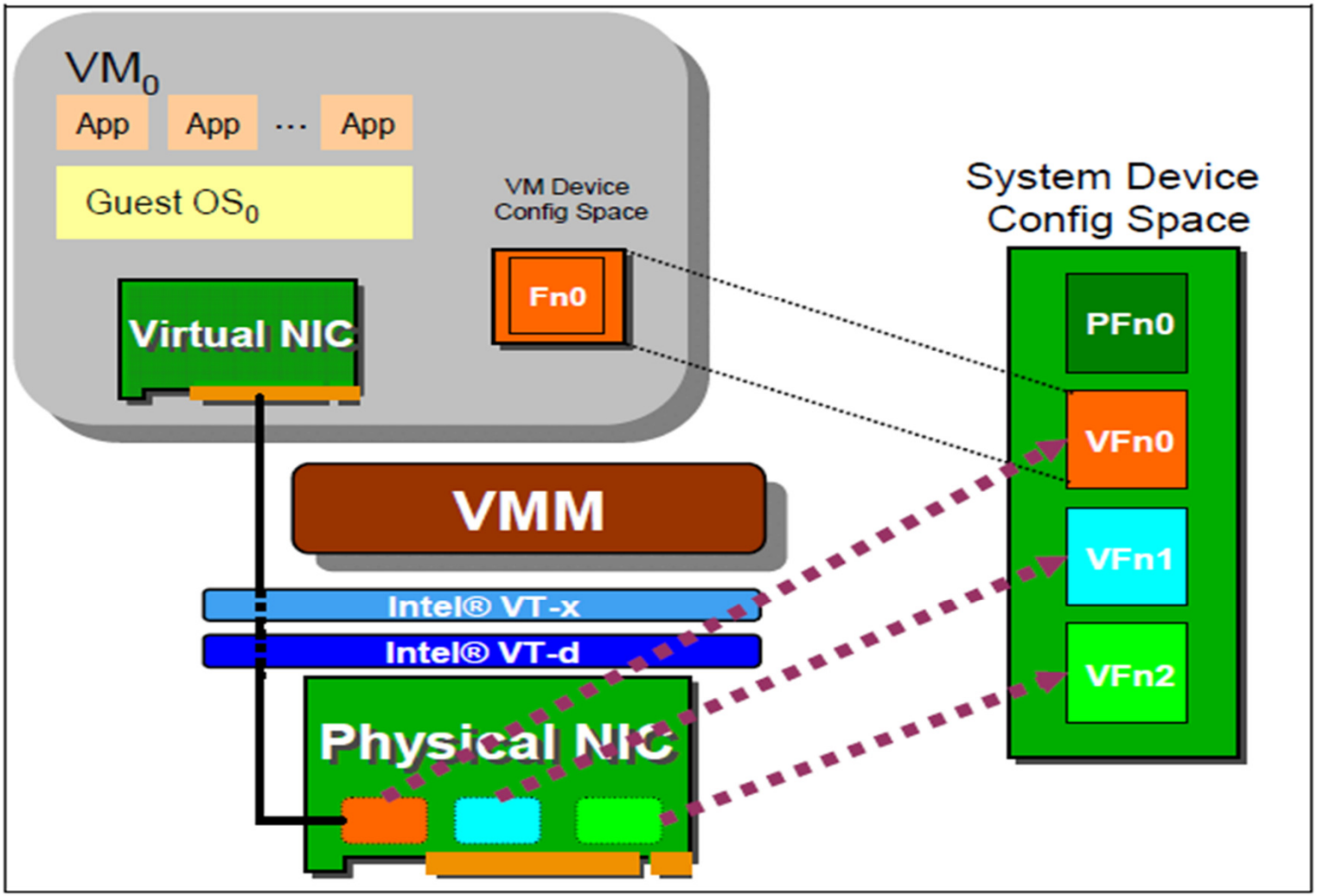
- I/O devices raise interrupts
- CPU temporarily stops the currently executing code
- CPU jumps to a pre-specified interrupt handler



SR-IOV (1/2)

- PCI-SIG Single Root I/O Virtualization and Sharing spec.
  - Physical Functions (PF)
    - Full configuration & management capability
  - Virtual Functions (VF)
    - “lightweight” functions: contain data-movement resources, with a reduced set of configuration resources
- An SR-IOV-capable device can be configured to appear (to the VMM) in the PCI configuration space as multiple functions
- The VMM assigns VF(s) to a VM by mapping the configuration space of the VF(s) to the configuration space presented to the VM.

SR-IOV : NIC example

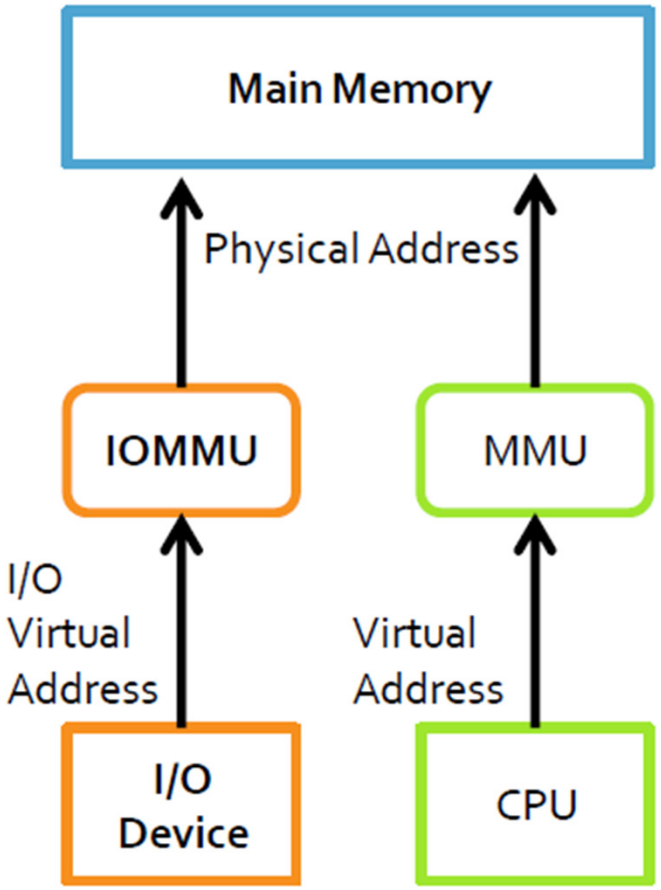


IOMMU

- Faster I/O via Pass-Through Devices
  - Exclusively used devices can be directly exposed to guest VM, without introducing device virtualization code
  - However, erroneous/malicious DMA operations are capable of corrupting/attacking memory spaces
  - Options:
    - Remapping of DMA operations by hardware (e.g. Intel VT-d)
    - Virtualizable devices (e.g. PCI-Express SR-IOV)

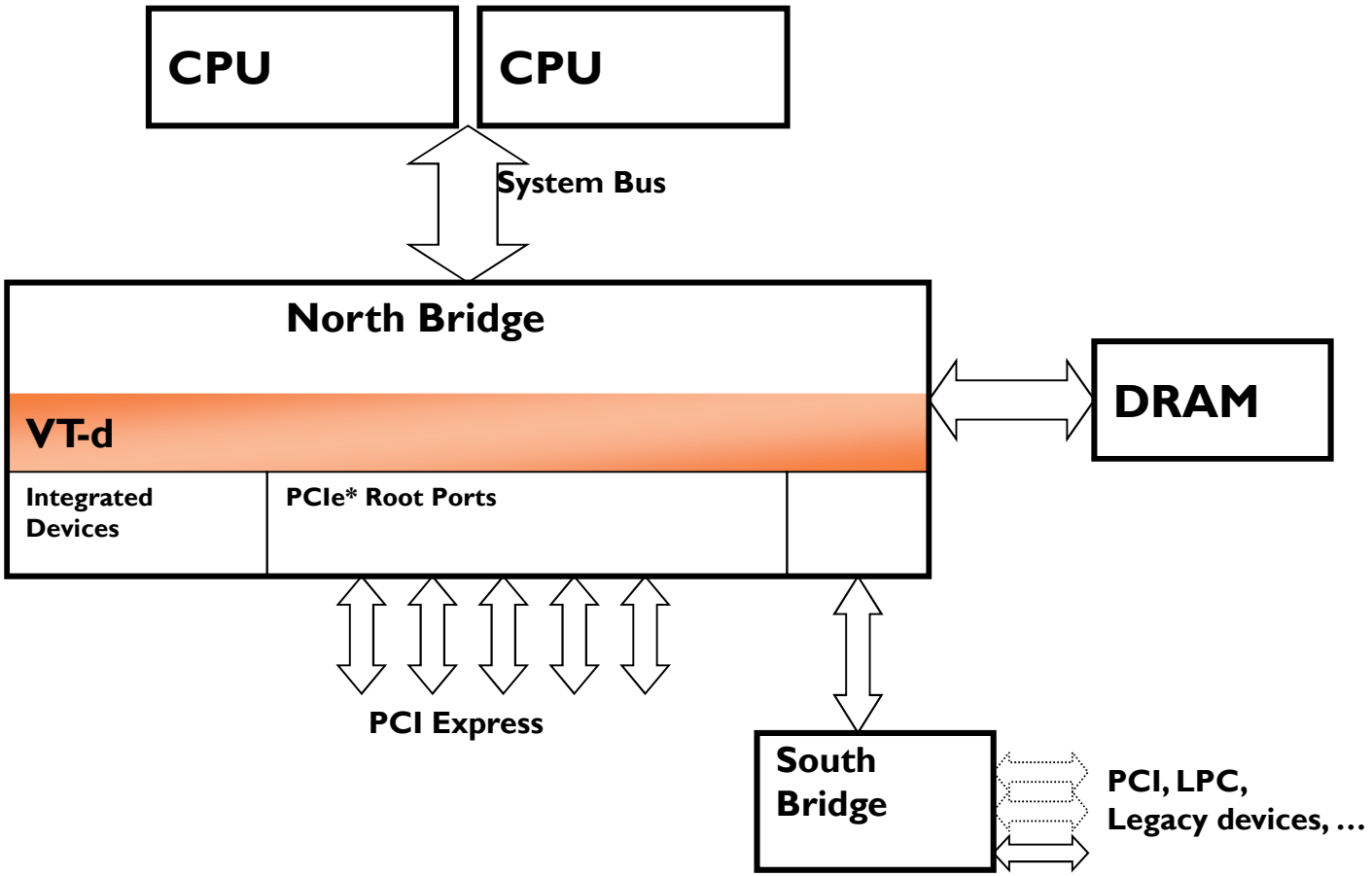
- I/O MMU
  - allow a guest OS running under a VMM to have direct control of a device
  - fine-grain control of device access to system memory
  - transparent to device drivers
  - software: swiotlb (bounce buffers), xen grant tables
  - hardware: AMD GART, IBM Calgary, AMD Pacifica

IOMMU concept

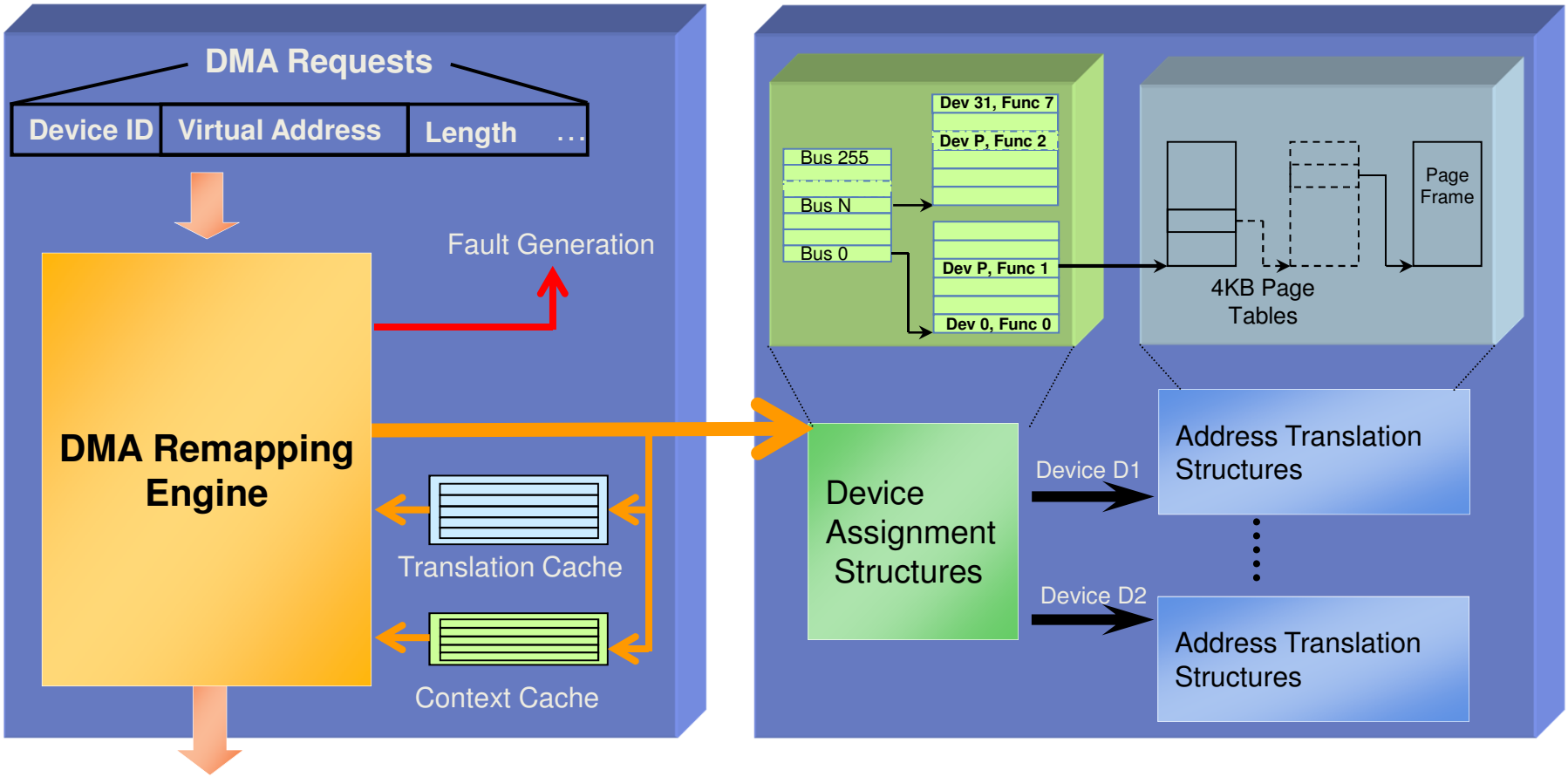


- Indirection between addresses used for DMA and physical addresses
  - Similar MMU, for address translation in the device context
- Devices can only access memory addresses "present" in their protection domain
- I/O virtualization through direct device assignment

Intel VT-d


CPU
CPU
System Bus
North Bridge
VT-d
Integrated Devices
PCIe* Root Ports
DRAM
PCI Express
South Bridge
PCI, LPC,
Legacy devices, …

Intel VT-d

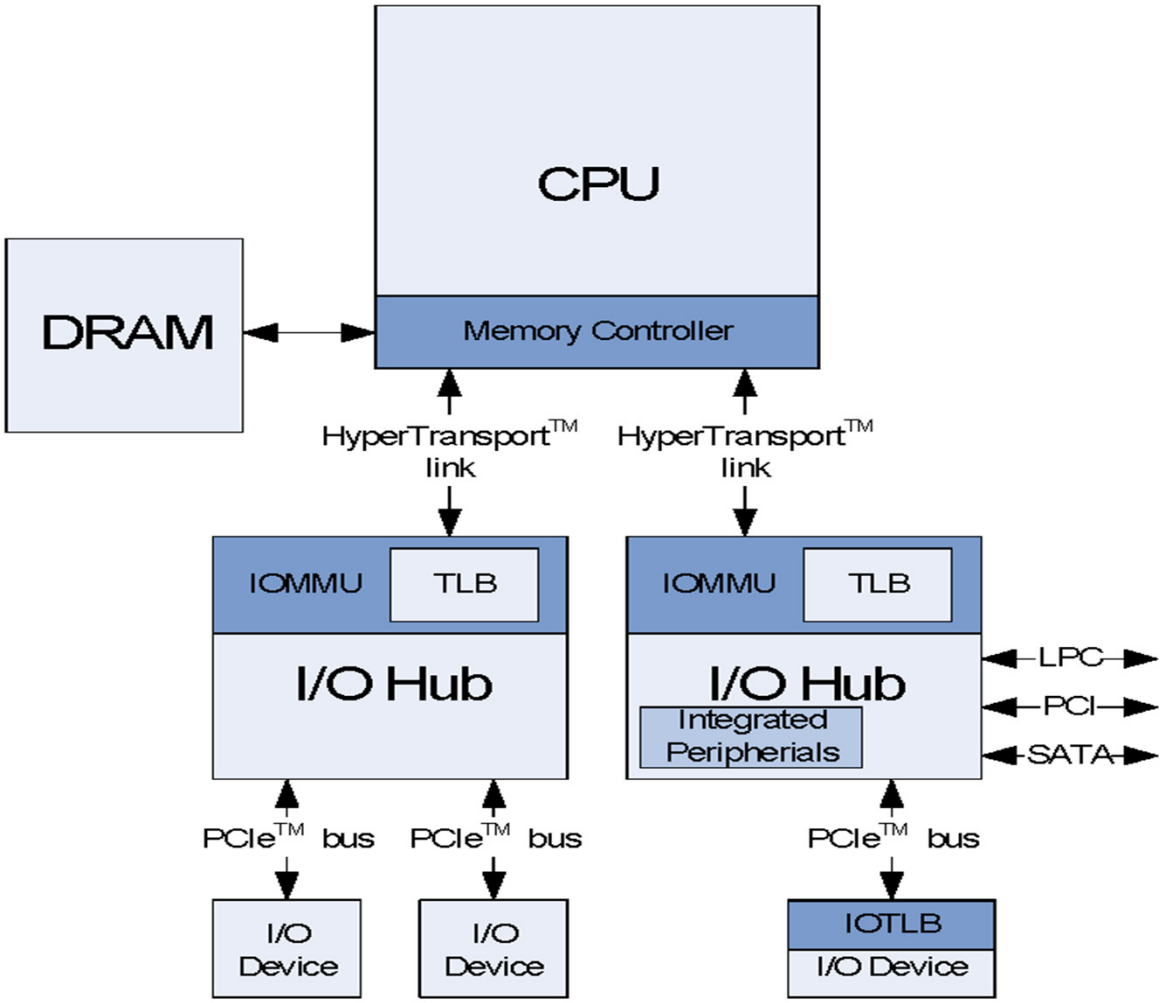


Memory Access with System Physical Address

Memory-resident Partitioning And Translation Structures

AMD's IOMMU architecture

[source: AMD IOMMU specification (revision 2)]



Xen device channels

- Asynchronous shared-memory transport
- Event ring (for interrupts)
- Xen “peer domains”
  - Inter-guest communication
  - Mapping one guest’s buffers to another
  - Grant tables for “DMA” (bulk transfers)
- Xen dom0 (privileged domain) can access all devices
  - Exports subset to other domains
  - Runs back-end of device drivers (e.g. net, block)

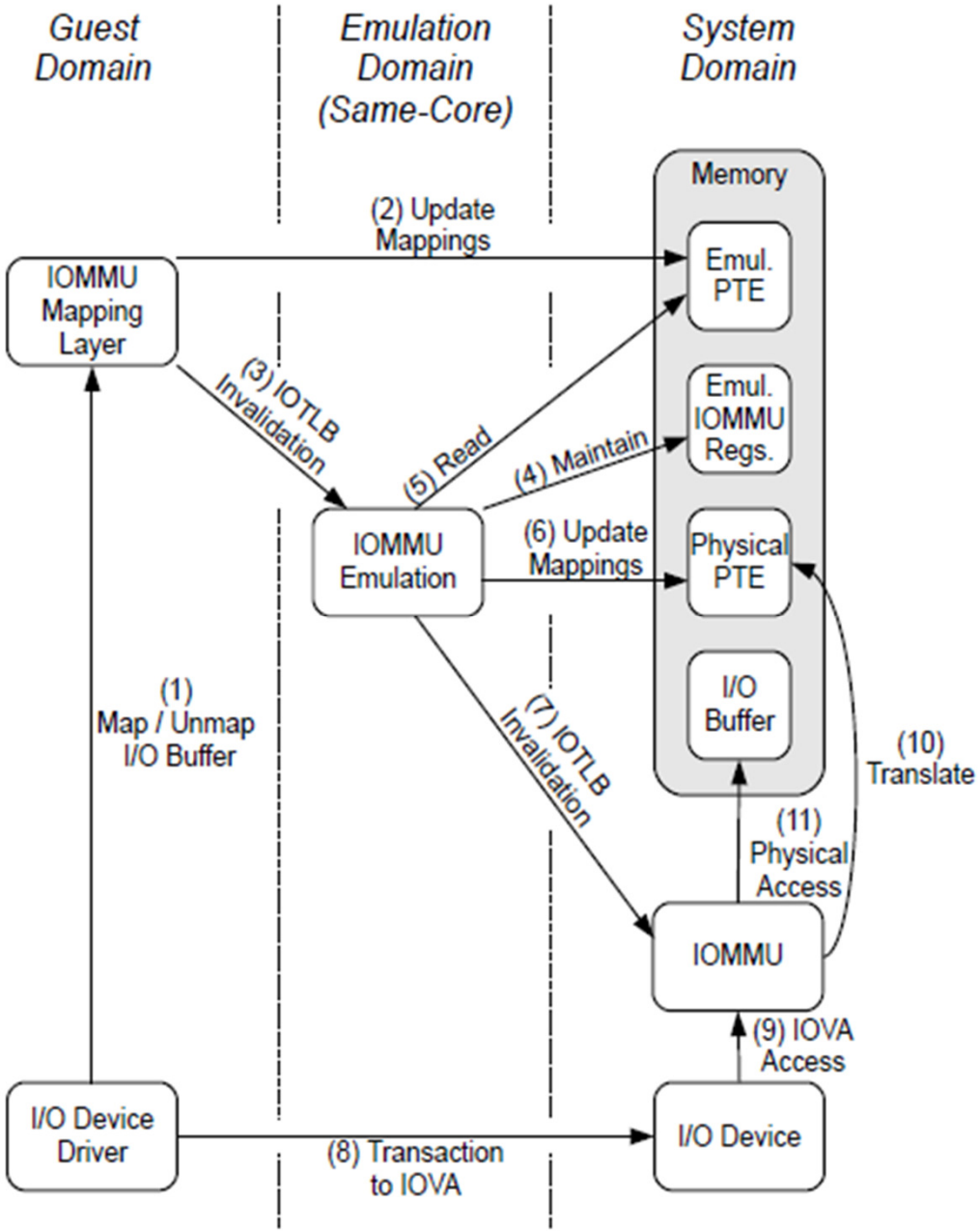
Xen grant tables

- Share & Transfer pages between domains
  - a software implementation of certain IOMMU functionality
- Shared pages:
  - Driver in local domain “advertises” buffer → notify hypervisor that this page can be access by other domains
  - Use case: block drivers → receive their data synchronously, i.e. know which domain requested data to be transferred via DMA
- Transferred pages:
  - Driver in local domain “advertises” buffer → notify hypervisor
  - Driver then transfers page to remote domain _and_ takes a free page from a producer/consumer ring (“page-flip”)
  - Use case: network drivers → receive their data asynchronously, i.e. may not know origin domain (need to inspect network packet before actual transfer between domains)
  - With RDMA NICs, we can transfer (DMA) directly into domains …

IOMMU Emulation

- Shortcomings of device assignment for unmodified guests:
  - Requires pinning all of the guest's pages, thereby disallowing memory over-commitment
  - exposes the guest's memory to buggy device drivers
- A single physical IOMMU can emulate multiple IOMMU's (for multiple guests)
- Why?
  - Allow memory over-commitment (pin/unpin during map/unmap of I/O buffers)
  - Intra-guest protection, redirection of DMA transactions
  - … without compromising inter-guest protection

IO Emulation



Sources (1)

- www.linux-kvm.org
- www.xen.org
- Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, **Xen and the Art of Virtualization**, SOSP'03
- **Xen (v.3.0. for x86) Interface Manual**
  - http://pdub.net/proj/usenix08boston/xen_drive/resources/developer_manuals/interface.pdf
- Jun Nakajima, Asit Mallick, Ian Pratt, Keir Fraser, **X86-64 XenLinux: Architecture, Implementation, and**
- **Optimizations,** OLS 2006
- **Xen: finishing the job, lwn.net - 2009**
- Avi Kivity, et al: **kvm: The Linux Virtual Machine Monitor**, Proceedings of the Linux Symposium, 2007
  - http://www.linux-kvm.com/sites/default/files/kivity-Reprint.pdf
  - http://kerneltrap.org/node/8088

Sources (2)

- Muli Ben-Yehuda, Eran Borovik, Michael Factor, Eran Rom, Avishay Traeger, Ben-Ami Yassour
  Adding Advanced Storage Controller Functionality via Low-Overhead Virtualization, *FAST '12*
- Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, Dan Tsafrir **ELI: Bare-Metal Performance for I/O Virtualization**, *ASPLOS '12*
- Alex Landau, Muli Ben-Yehuda, Abel Gordon
  SplitX: Split Guest/Hypervisor Execution on Multi-Core, *WIOV '11*
- Abel Gordon, Muli Ben-Yehuda, Dennis Filimonov, Maor Dahan,
  VAMOS: Virtualization Aware Middleware, *WIOV '11*

Sources (3)

- Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, Ben-Ami Yassour
  The Turtles Project: Design and Implementation of Nested Virtualization, *OSDI '10*
- Ben-Ami Yassour, Muli Ben-Yehuda, Orit Wasserman
  On the DMA Mapping Problem in Direct Device Assignment, *SYSTOR '10*
- Alex Landau, David Hadas, Muli Ben-Yehuda,
  Plugging the Hypervisor Abstraction Leaks Caused by Virtual Networking, *SYSTOR '10*
- Nadav Amit, Muli Ben-Yehuda, Dan Tsafir, Assaf Schuster,
  vIOMMU: Efficient IOMMU Emulation, *USENIX ATC 2011*