# I/O in Linux Hypervisors and Virtual Machines
## Lecture for the Embedded Systems Course
## CSD, University of Crete (May 7 & 9, 2025)


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

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

# Outline

- Elements of I/O virtualization
  - Machine emulator, Hypervisor, Transport
- Alternative designs for the 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 most commonly used on Linux servers (x86 platform)
  - xen, kvm
- Provide a glimpse of hypervisor internals *
  - xen: device channels, grant tables
  - kvm: virtio
  - * VMware: market leader … but out-of-scope for this lecture

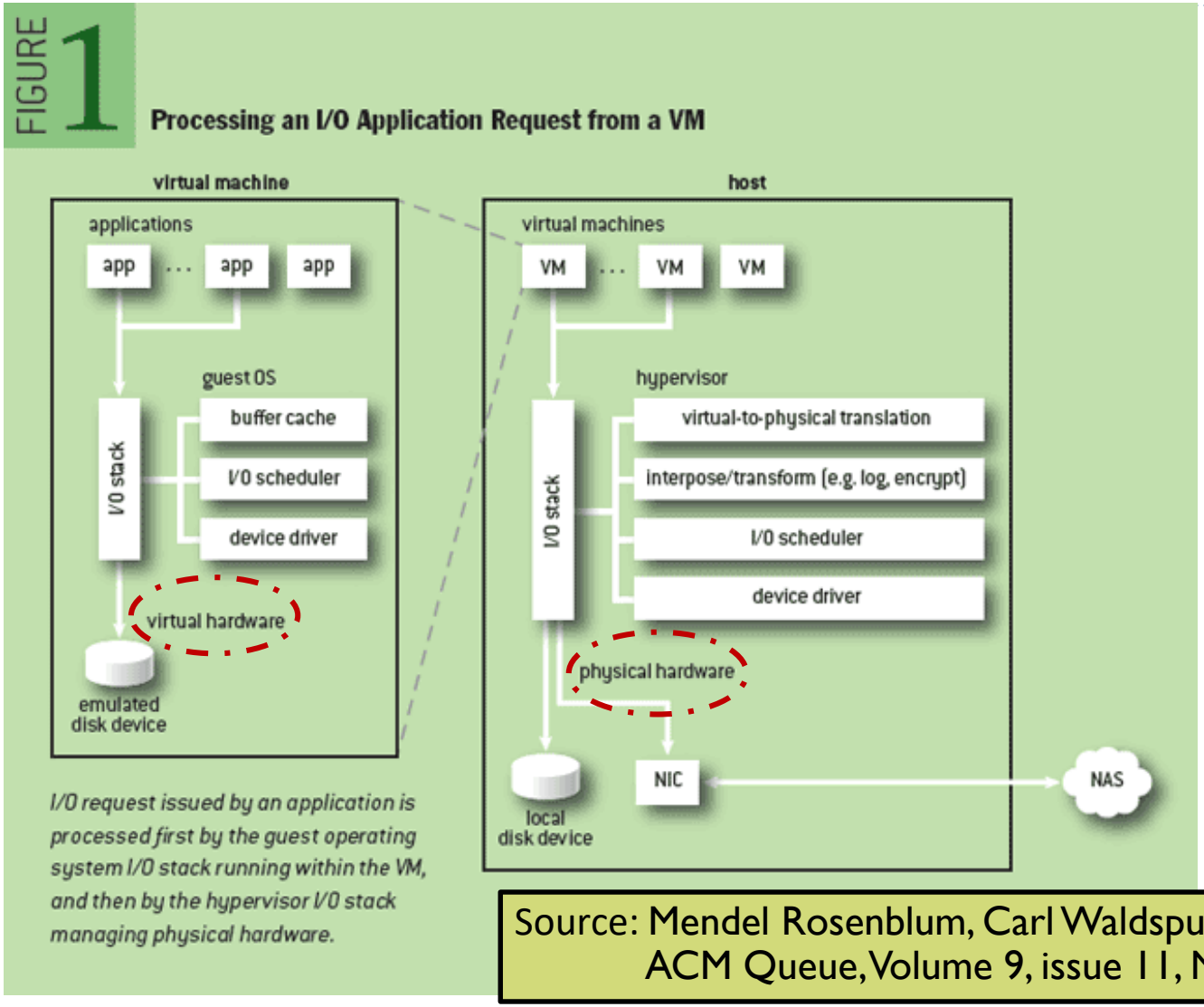
# I/O device types

- Dedicated
  - E.g. Display
- Partitionable
  - E.g. Disk
- Shared
  - E.g. NIC
- Devices that can be enumerated (PCI, PCI-Express)
  - VMM needs to emulate ‘discovery’ over a system 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

# Why is I/O hard to virtualize ?

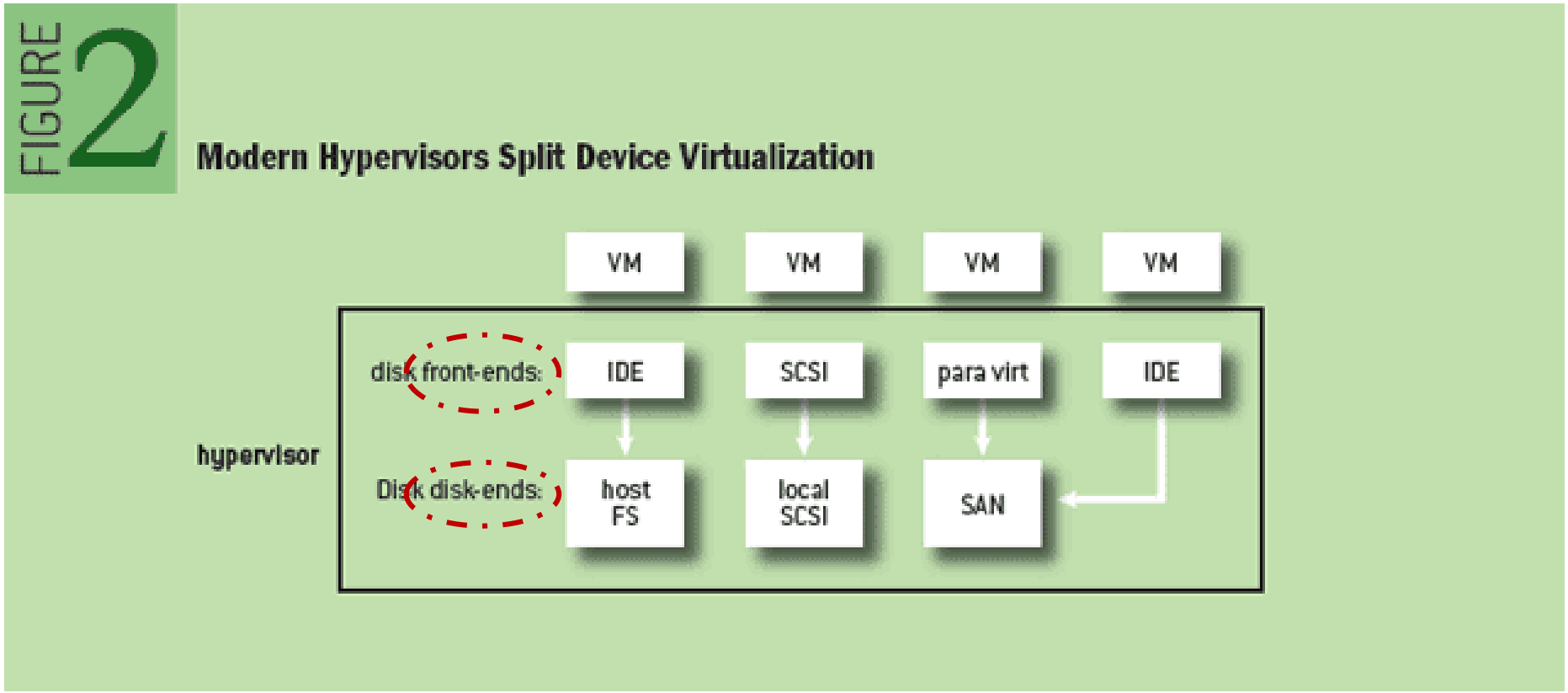
- Multiplexing/de-multiplexing for guests
  - Programmed I/O (PIO): privileged CPU instructions specifically for I/O
    - I/O devices have a separate address space from general memory
  - Memory-mapped I/O (MMIO): CPU instructions for memory access are also used for accessing devices
    - The memory-mapped I/O region is protected
- Direct Memory Access (DMA)
  - Allow 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 calls (application to GuestOS) → trap to VMM
  - driver calls (GuestOS) → paravirtualization
    - Hypercall between modified driver in GuestOS and VMM
  - I/O operations (GuestOS driver to VMM)

# Processing an I/O Request from a VM



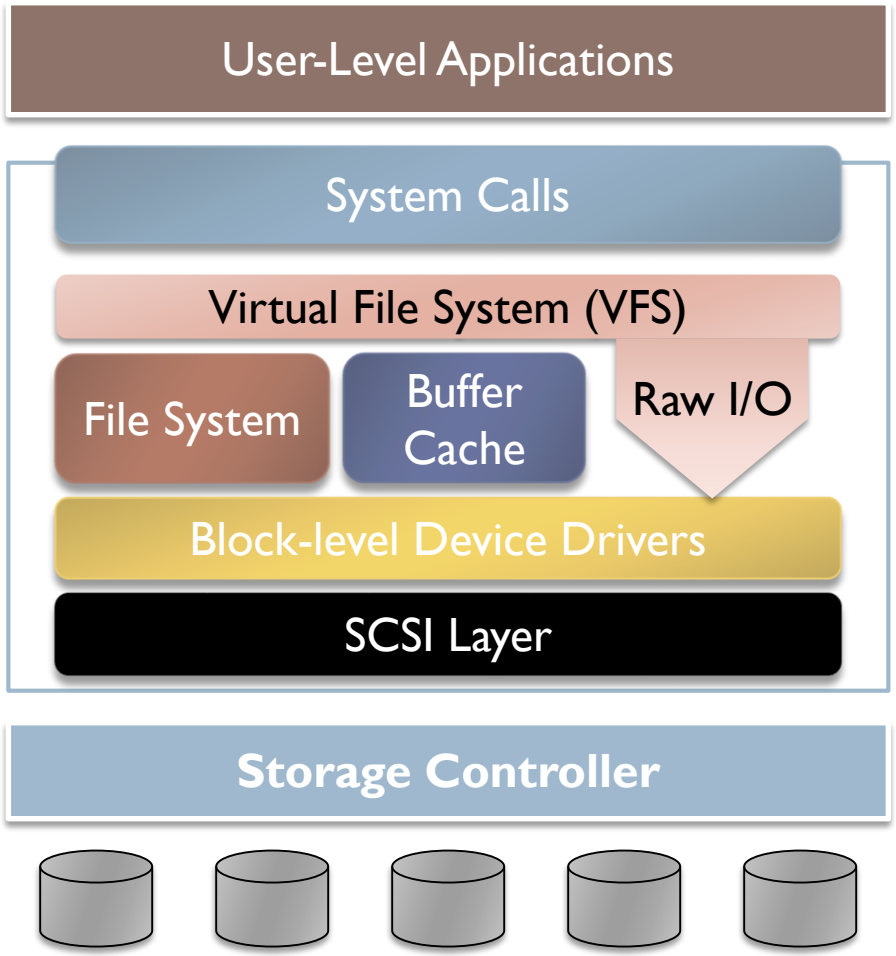

Source: Mendel Rosenblum, Carl Waldspurger: "**I/O Virtualization**" ACM Queue, Volume 9, issue 11, November 22, 2011

# Device Front-Ends & Back-Ends

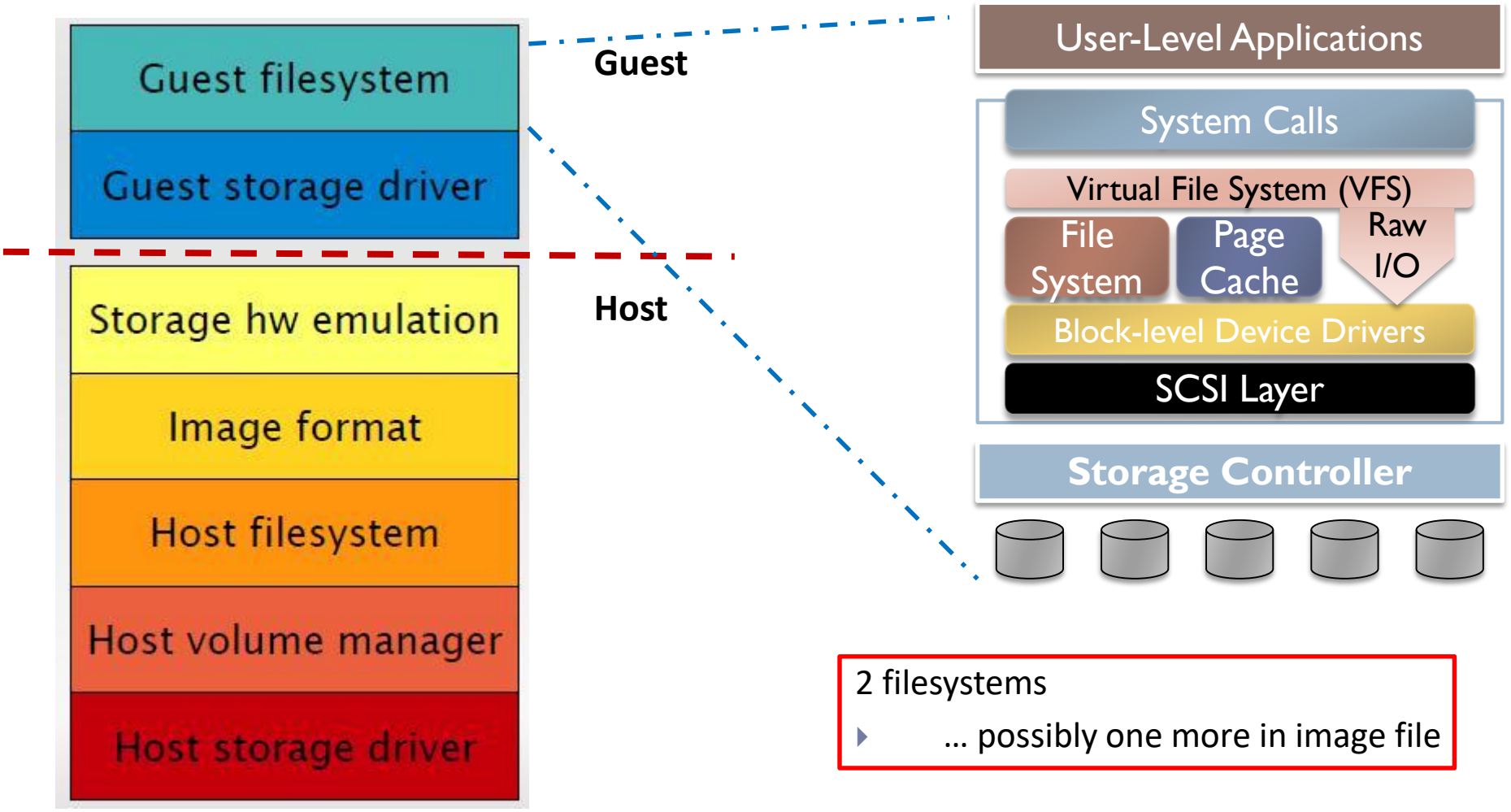



Source: Mendel Rosenblum, Carl Waldspurger: "**I/O Virtualization**" ACM Queue, Volume 9, issue 11, November 22, 2011

# I/O stack: Host side



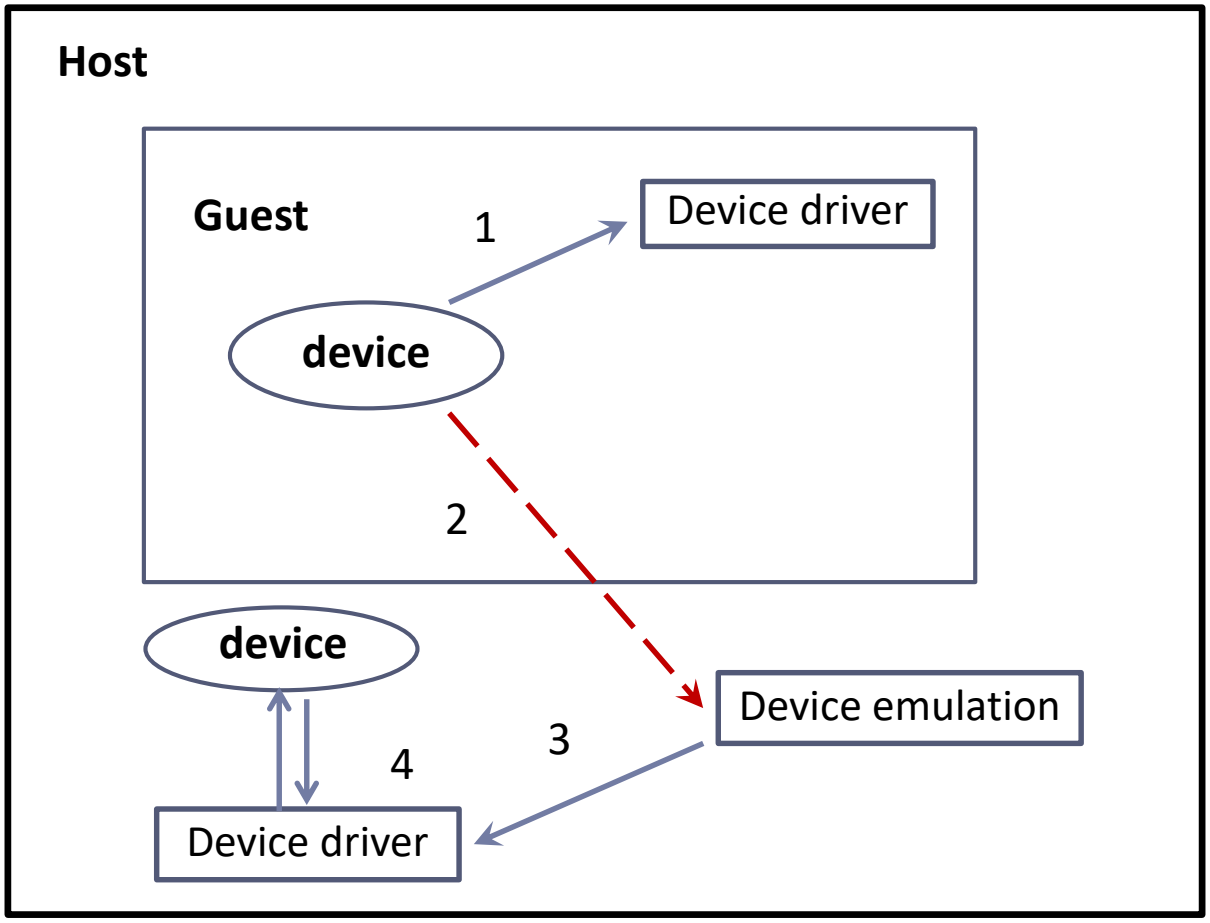
# I/O stack: Host + Guest




2 filesystems

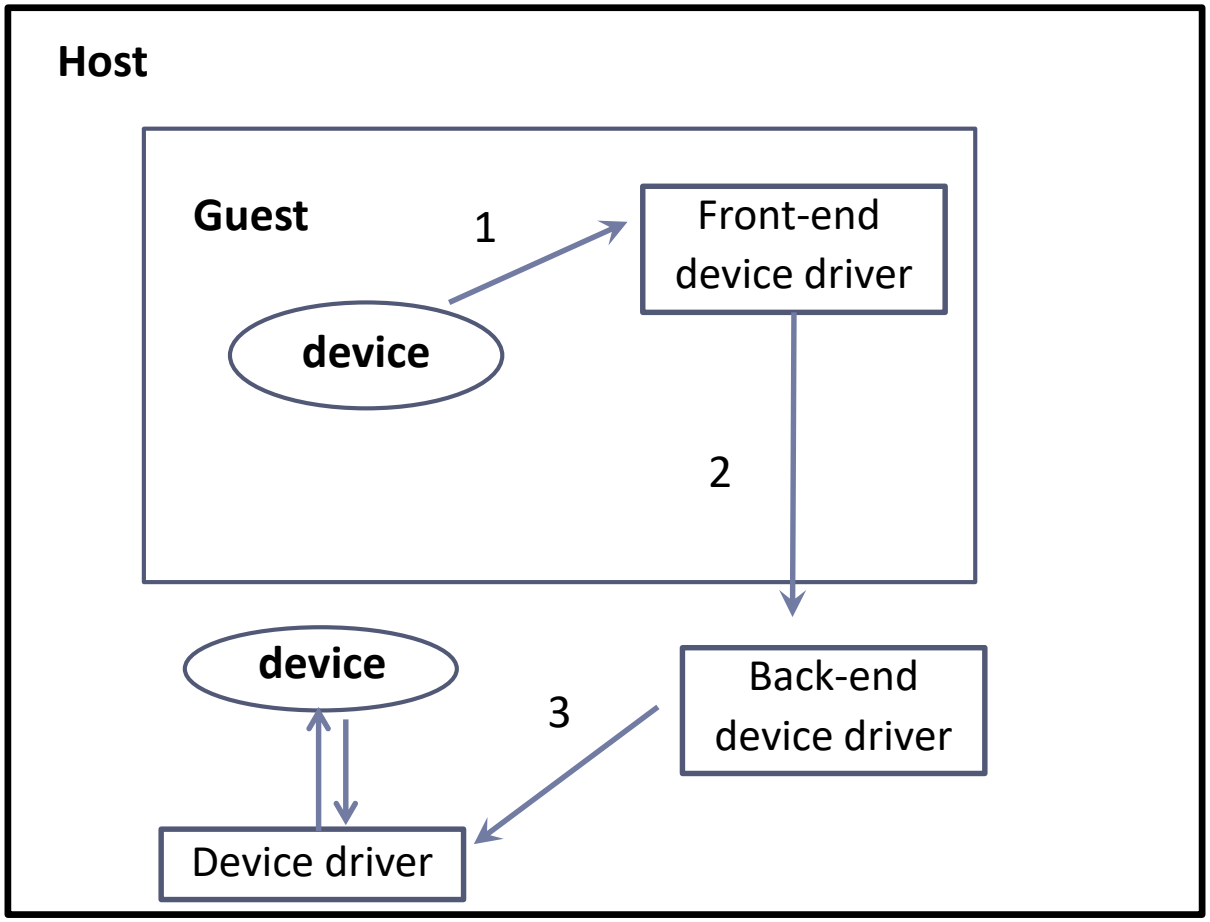
- … possibly one more in image file

# Device Emulation



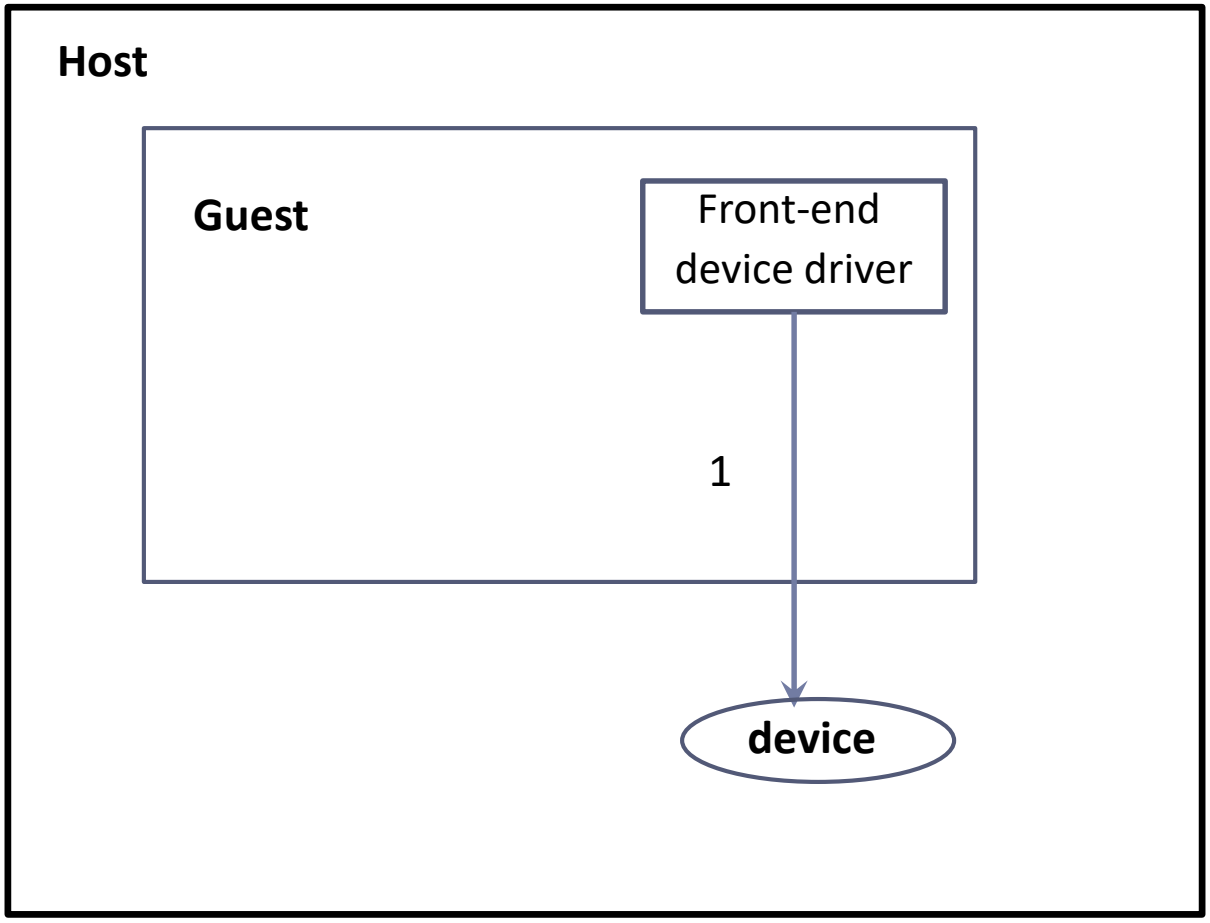

- Hypervisor emulates devices → traps I/O accesses (PIO, MMIO)
- Emulation of DMA and interrupts

# Para-virtualized drivers




- Hypervisor-specific virtual device drivers (front-end) in Guest OS
- Involvement of Hypervisor (for Back-end)

# Direct device assignment

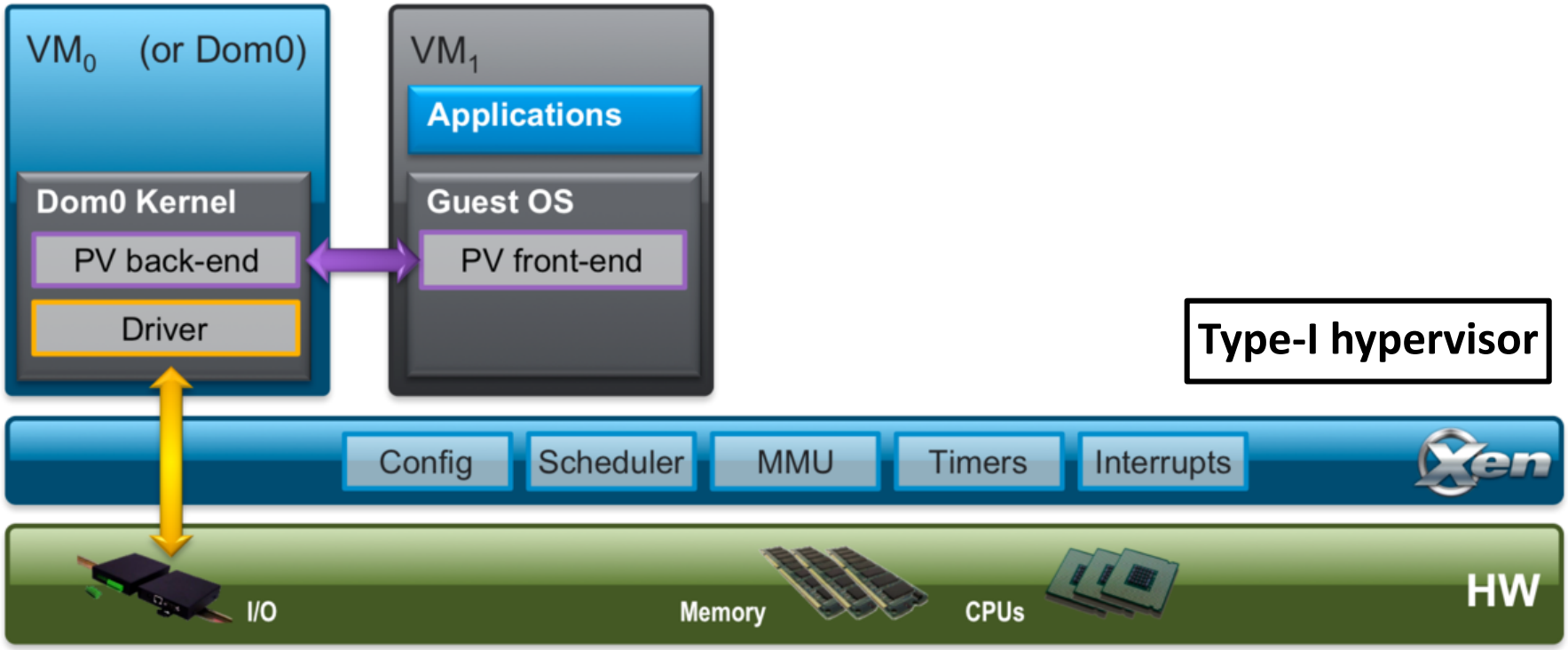



- Bypass Hypervisor to directly access I/O devices
- Security & safety concerns
  - IO-MMU for address translation & isolation (DMA restrictions)
  - SR-IOV for shared access

# xen history

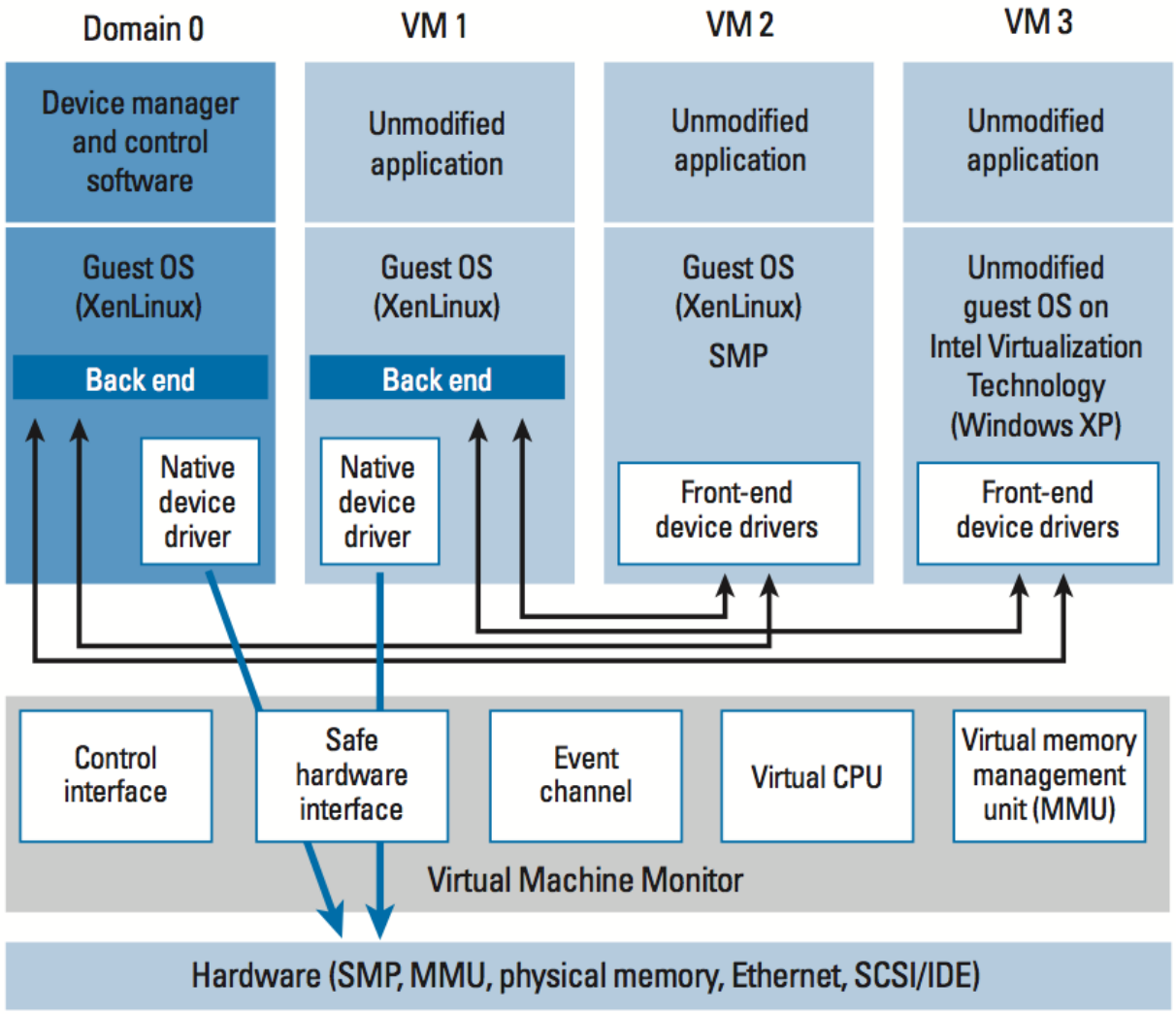
- Developed at Systems Research Group, Cambridge University (UK)
- Creators: Keir Fraser, Steven Hand, Ian Pratt, et al (2003)
- Broad scope, for both Host and Guest
- Merged in Linux mainline:  2.4.22
- Company: Xensource.com
  - Acquired by Citrix Systems (2007)

# Xen VMM: Paravirtualization (PV) 1/2




source: https://wiki.xenproject.org/wiki/Paravirtualization_(PV)

# Xen VMM: Paravirtualization (PV) 2/2




OS'es

- Dedicated control domain: Dom0
- Modified Guest OS

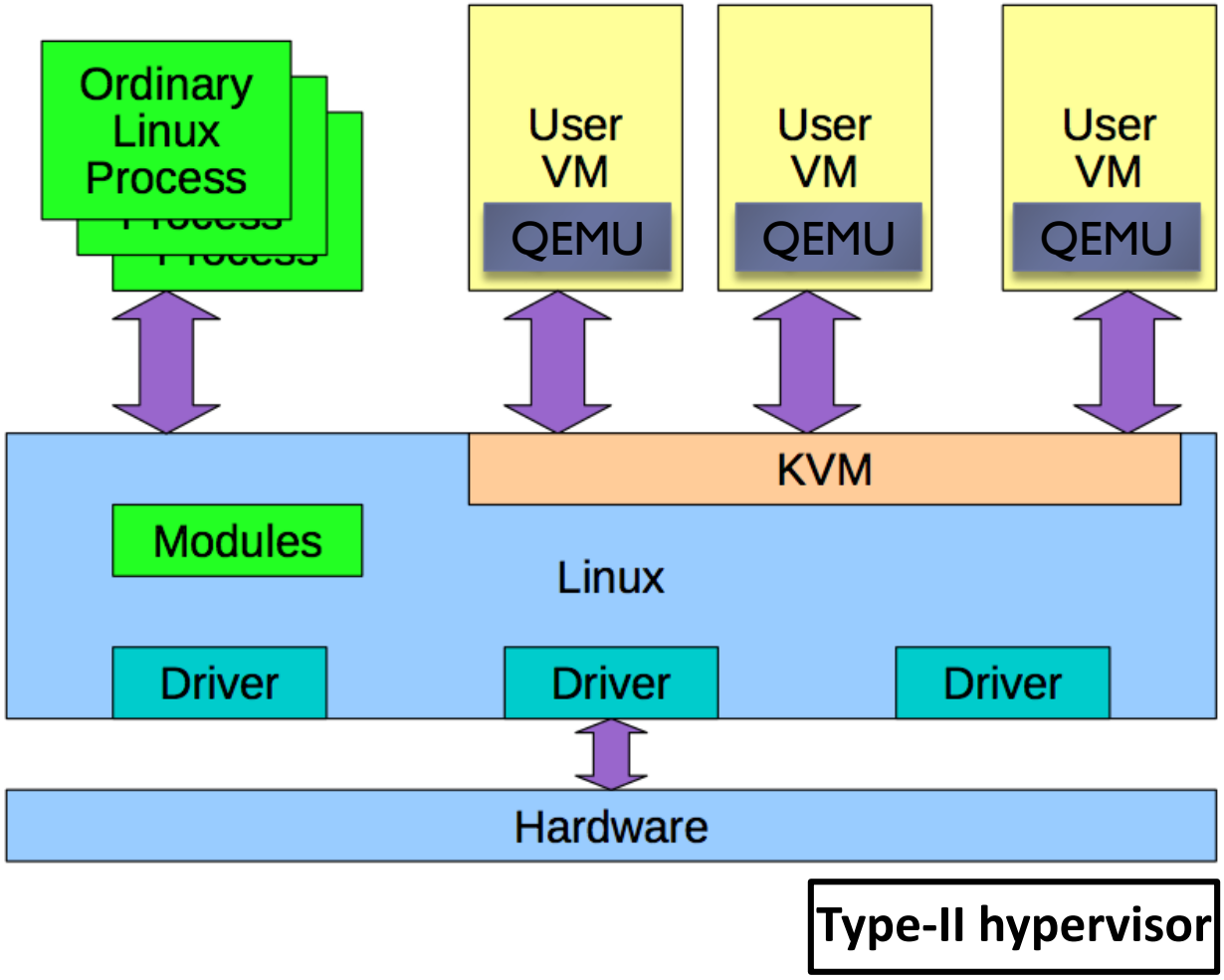
Devices

- 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

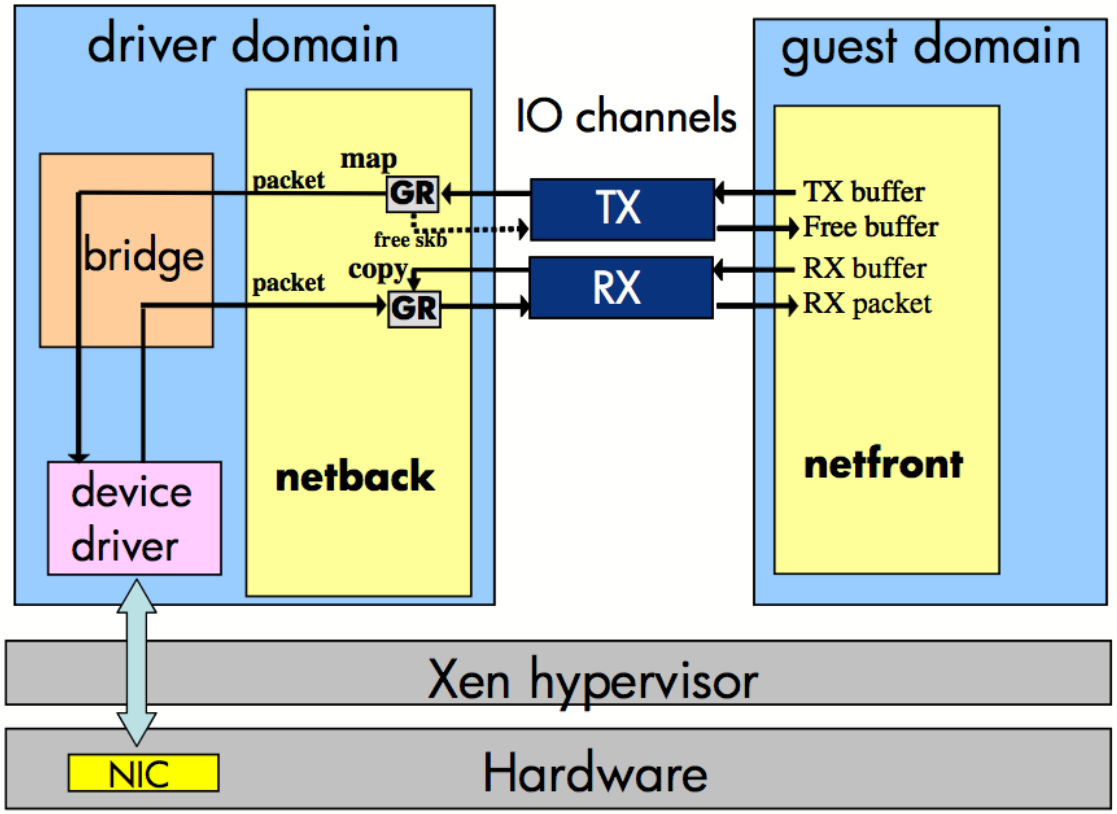
## Xen

- Strong support for para-virtualization with modified host-OS
  → Near-native performance for I/O devices
- Separate code based for DOM0 and device drivers
- Security model:
  Rely on DOM0
- Maintainability:
  Hard to keep up with 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 (via virtio)
- 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

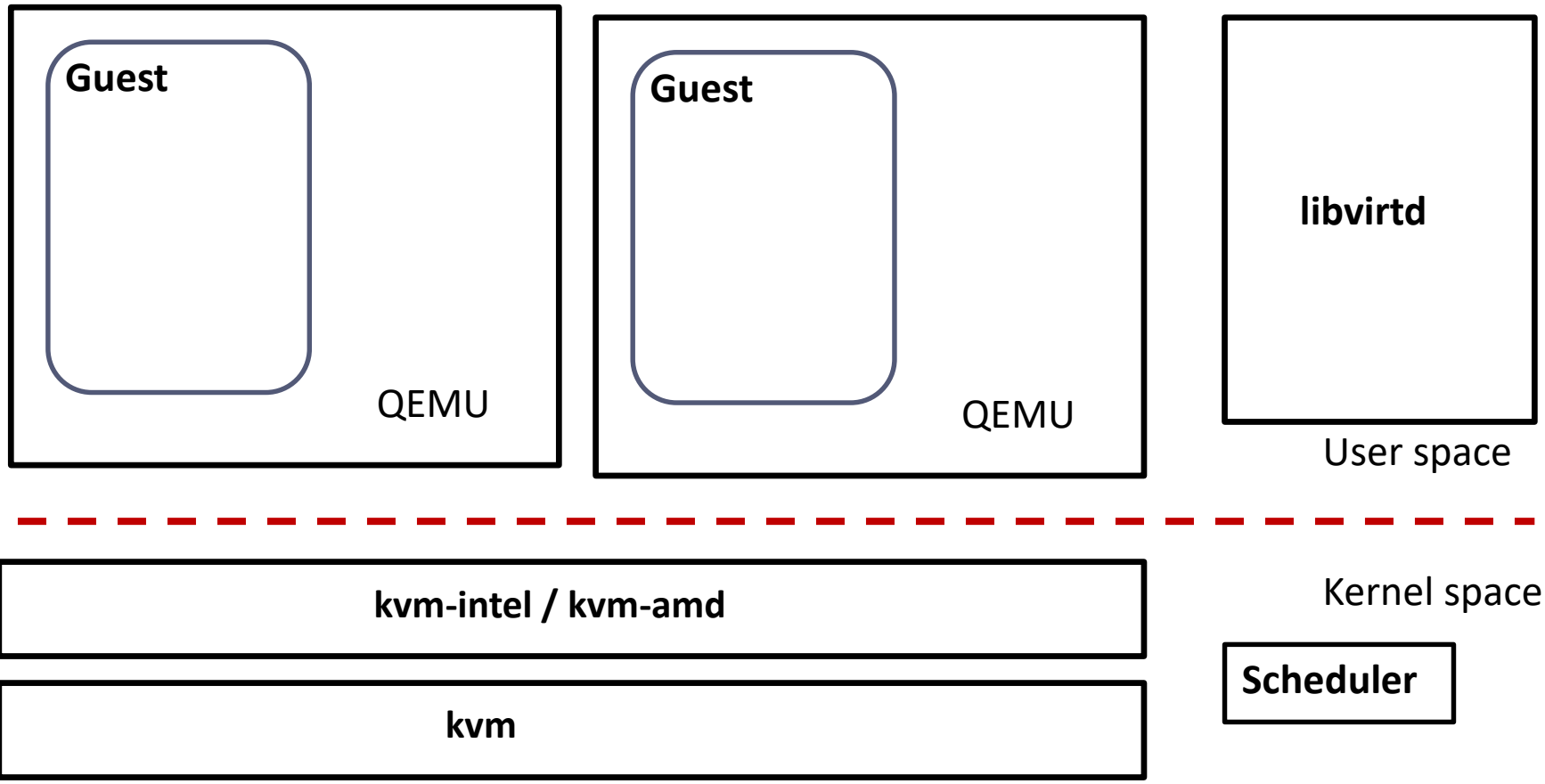
# Xen device channels

- 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)

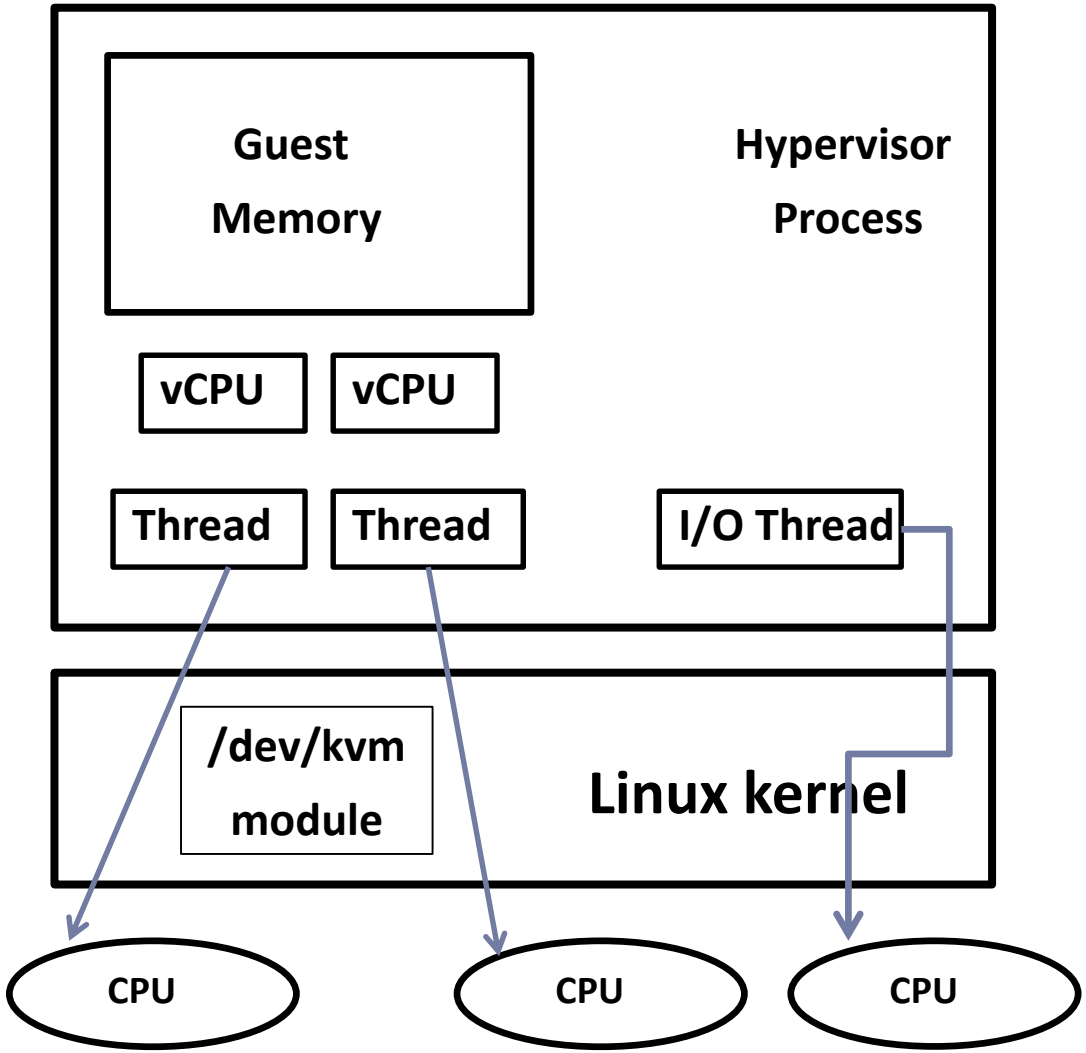
# Xen grant tables

- Share & Transfer pages between domains
  - a software implementation of certain IOMMU functionality
- 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 …
- 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

# kvm run-time environment



# 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

# 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!

kvm API demonstration:

- https://lwn.net/Articles/658511/
- https://github.com/soulxu/kvmsample

# kvm API sample

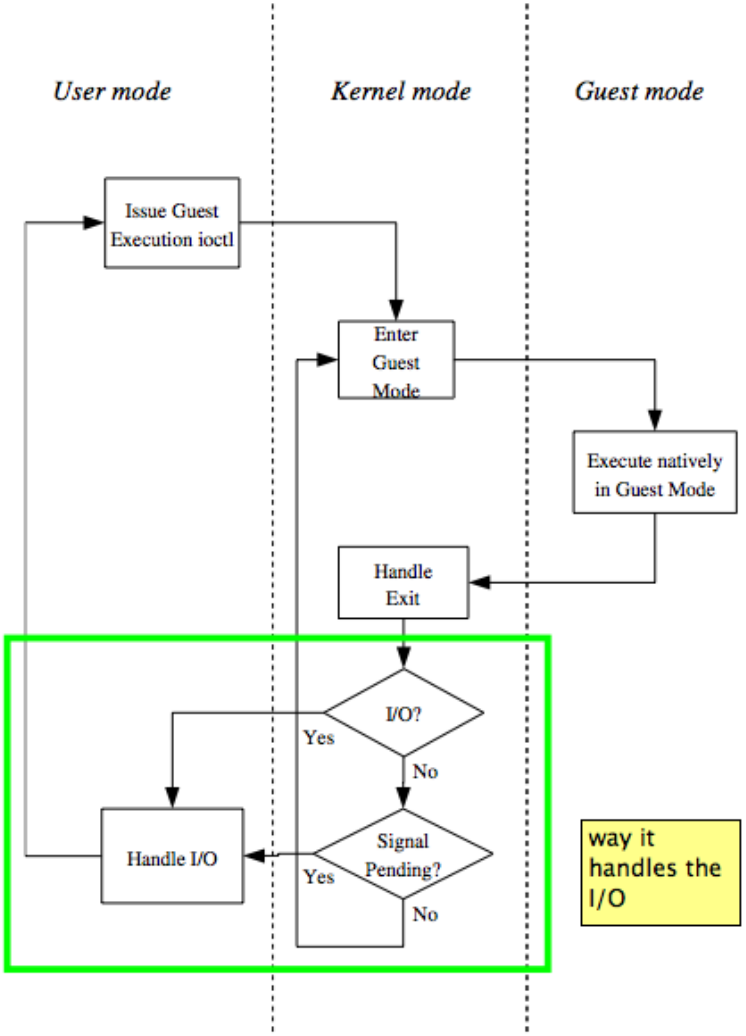
```
#include <linux/kvm.h>
struct kvm {
  int dev_fd;
  int vm_fd;
  __u64 ram_size;
  __u64 ram_start;
  int kvm_version;
  struct kvm_userspace_memory_region mem;
  struct vcpu *vcpus;
  int vcpu_number;
};
struct vcpu {
   int vcpu_id;
   int vcpu_fd;
   pthread_t vcpu_thread;
   struct kvm_run *kvm_run;
   int kvm_run_mmap_size;
   struct kvm_regs regs;
   struct kvm_sregs sregs;
   void *(*vcpu_thread_func)(void *);
};
```

```
struct kvm *kvm = malloc(sizeof(struct kvm));
kvm->dev_fd = open(KVM_DEVICE, O_RDWR);
kvm->vm_fd = ioctl(kvm->dev_fd, KVM_CREATE_VM, 0);
kvm->ram_size = ram_size;
kvm->ram_start =  (__u64)mmap(NULL, kvm->ram_size,
PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE, -1, 0);

kvm->mem.slot = 0;
kvm->mem.guest_phys_addr = 0;
kvm->mem.memory_size = kvm->ram_size;
kvm->mem.userspace_addr = kvm->ram_start;
ret = ioctl(kvm->vm_fd, KVM_SET_USER_MEMORY_REGION, &(kvm->mem));

struct vcpu *vcpu = malloc(sizeof(struct vcpu));
vcpu->vcpu_id = 0;
vcpu->vcpu_fd = ioctl(kvm->vm_fd, KVM_CREATE_VCPU, vcpu->vcpu_id);
vcpu->kvm_run_mmap_size = ioctl(kvm->dev_fd, KVM_GET_VCPU_MMAP_SIZE, 0);
vcpu->kvm_run = mmap(NULL, vcpu->kvm_run_mmap_size, PROT_READ |
PROT_WRITE, MAP_SHARED, vcpu->vcpu_fd, 0);
pthread_create(&(kvm->vcpus->vcpu_thread), (const pthread_attr_t *)NULL, kvm->vcpus[i].vcpu_thread_func, kvm);
pthread_join(kvm->vcpus->vcpu_thread, NULL);
```

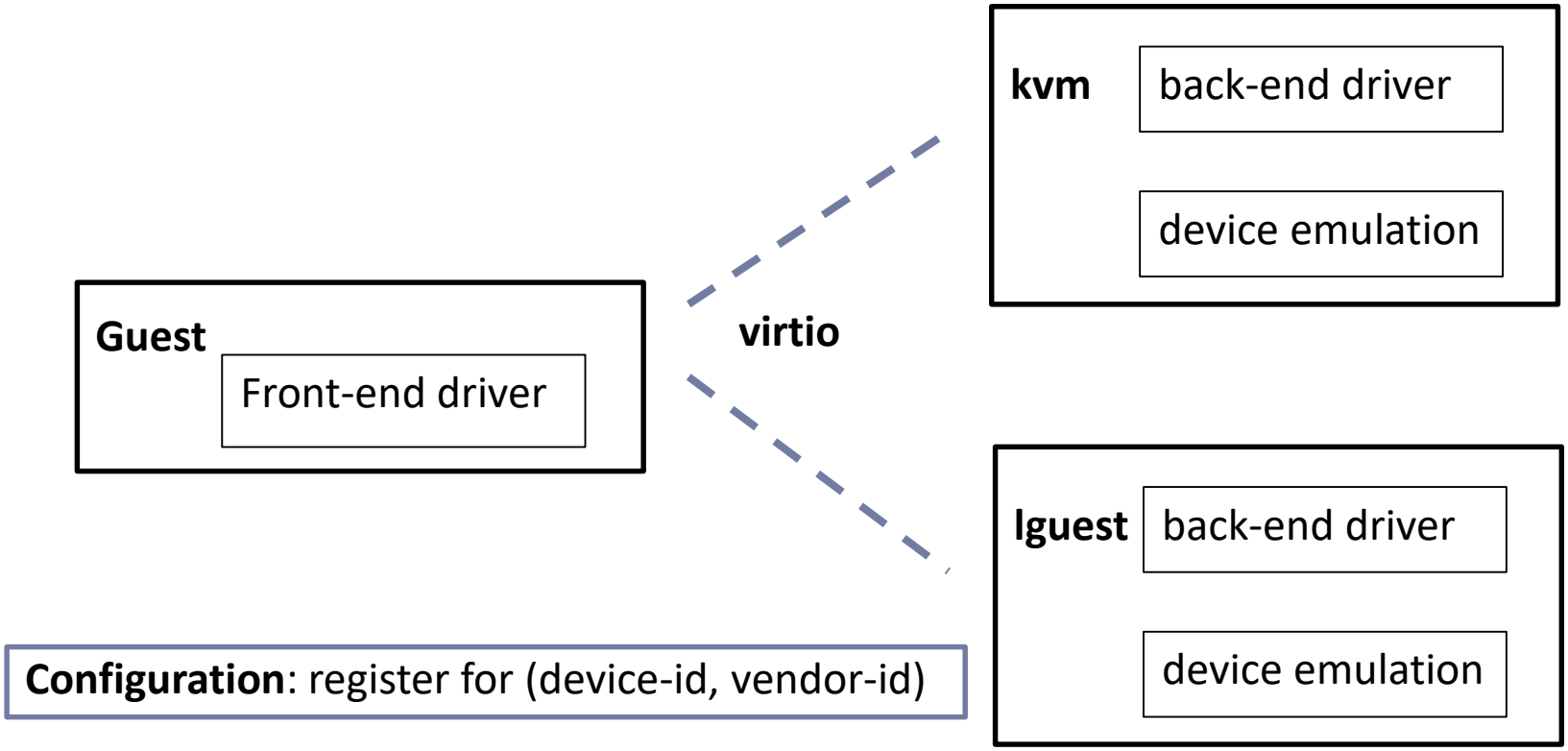
# I/O virtualization in kvm



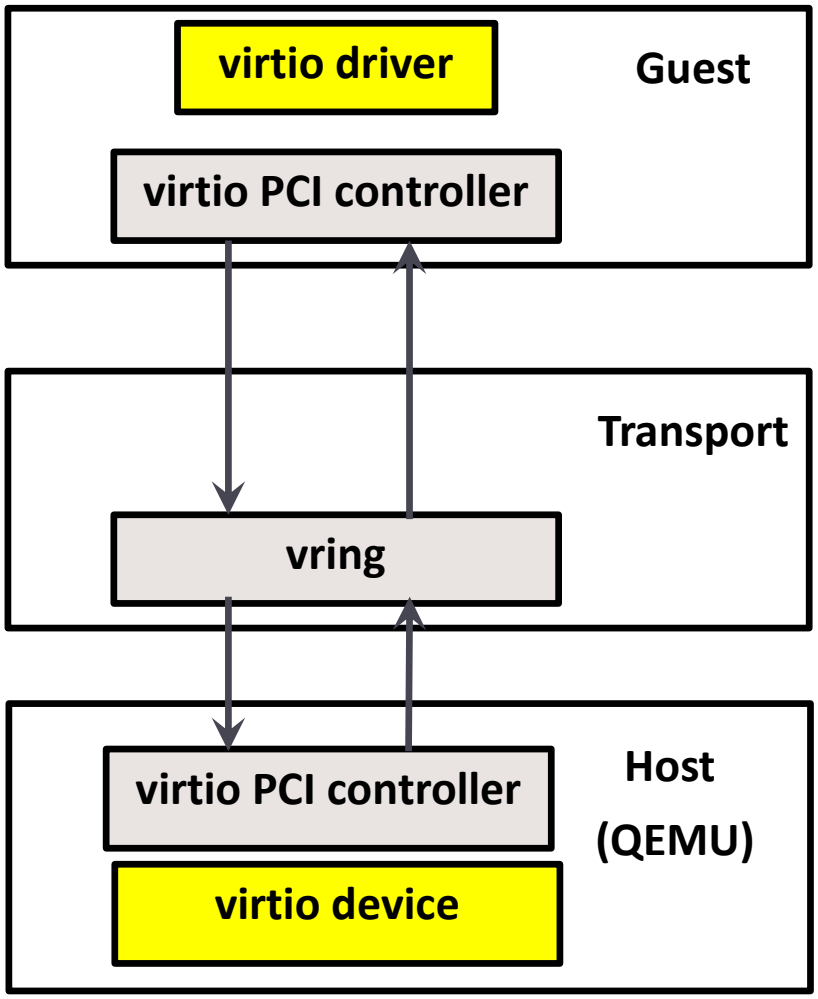

- Native KVM I/O model
- PIO: Trap
- MMIO: The machine emulator executes the faulting instruction
- Slow due to mode-switch!
- Extensions to support PV
  - VirtIO: An API for Virtual I/O aims to support many hypervisors (of all types)

# 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

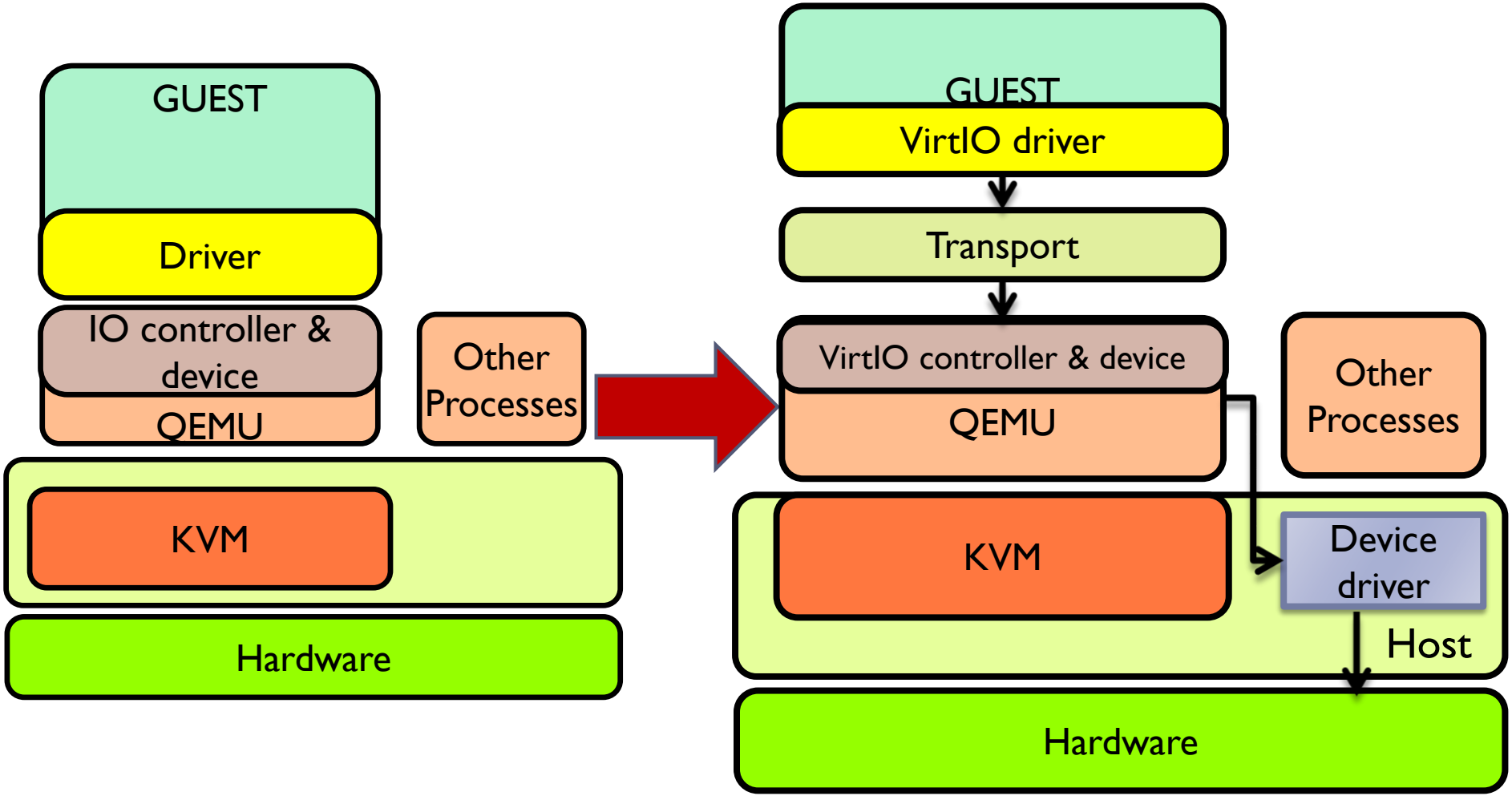
A device in QEMU.
Accepts I/O requests from front-end.
Perform I/O operation via physical device.

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

# 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

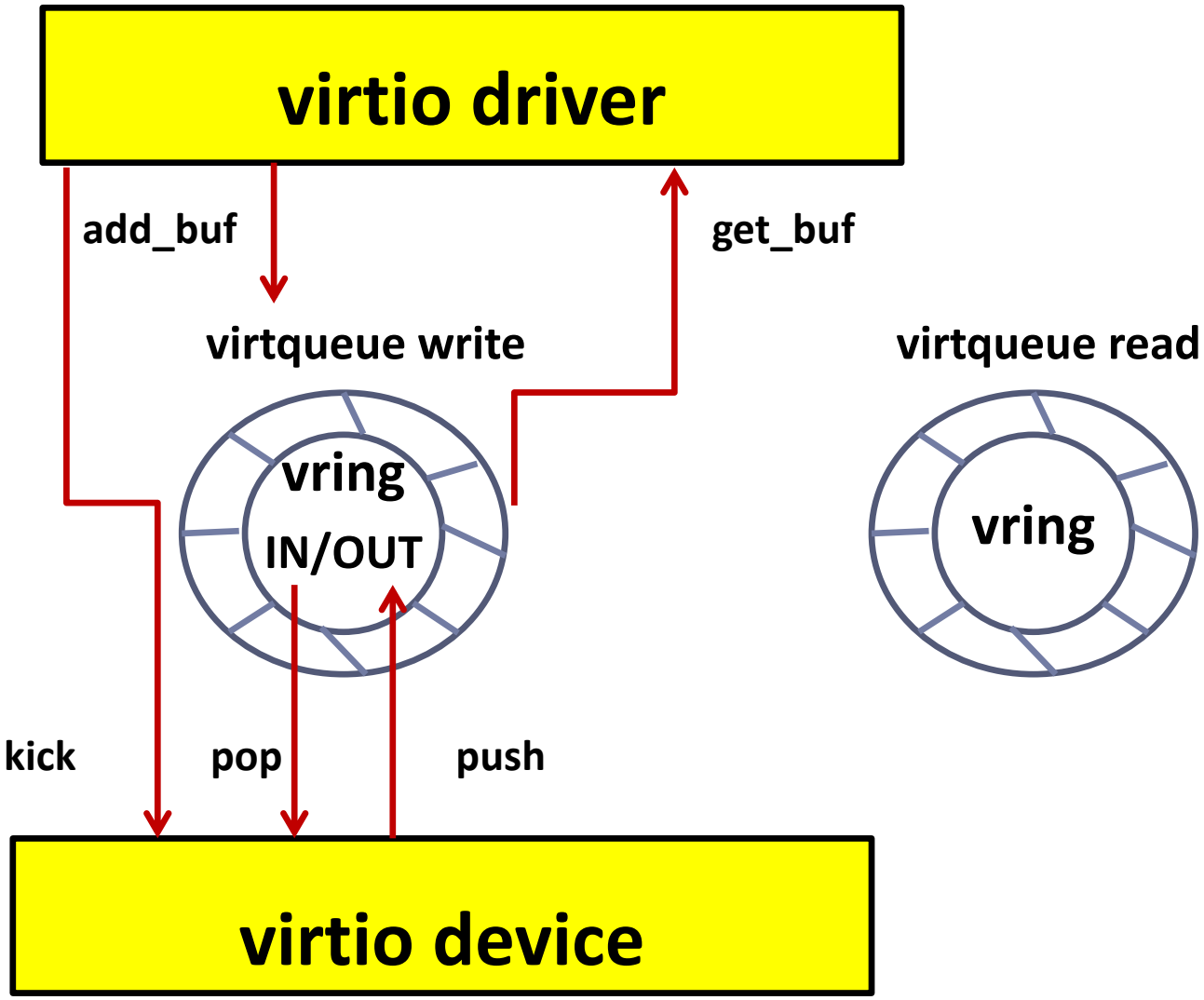
# kvm with virtio



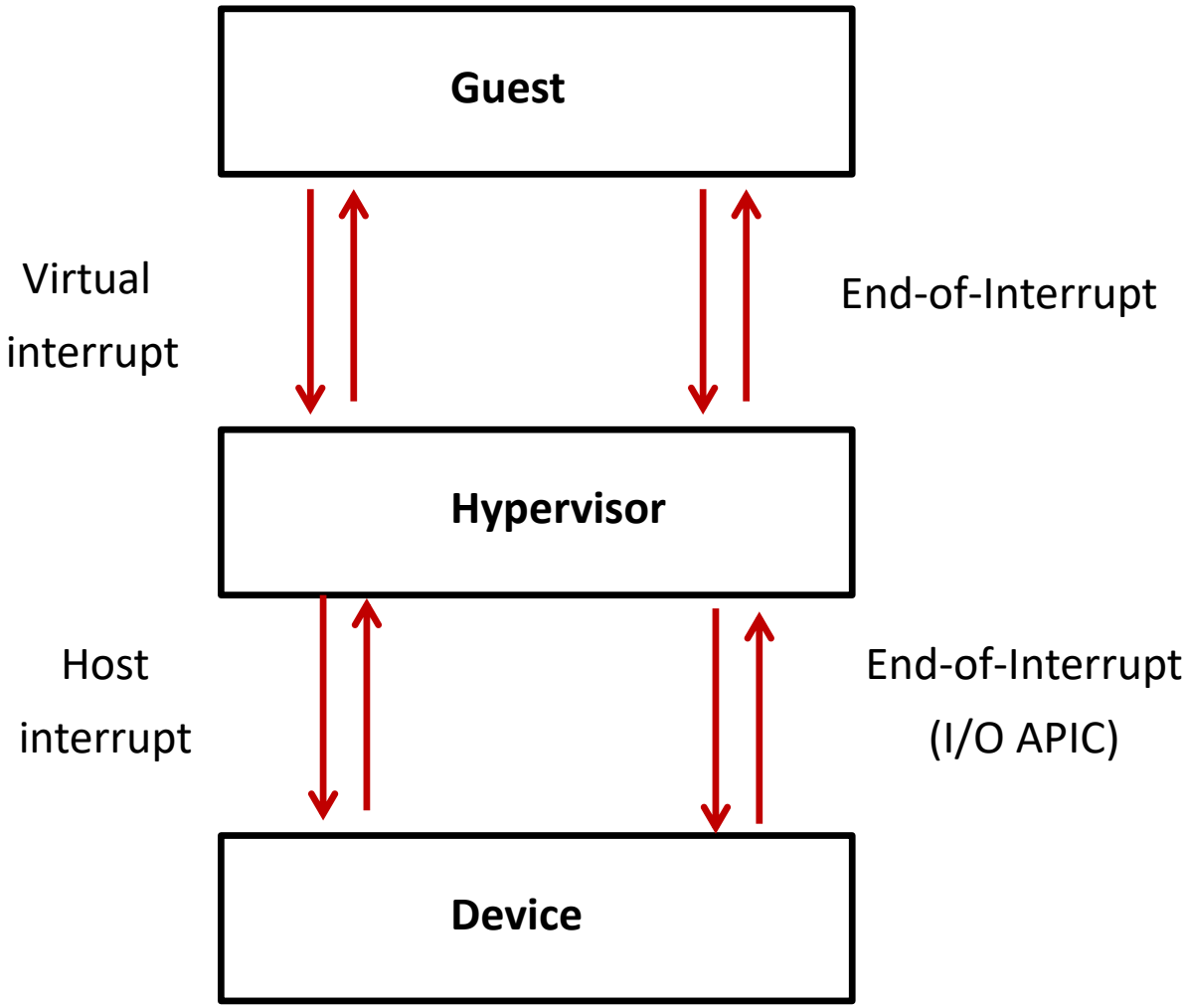

[ kvm with 'default' device emulation ]

[ kvm with 'virtio' device handling ]

# virtio processing flow



# Virtual interrupts

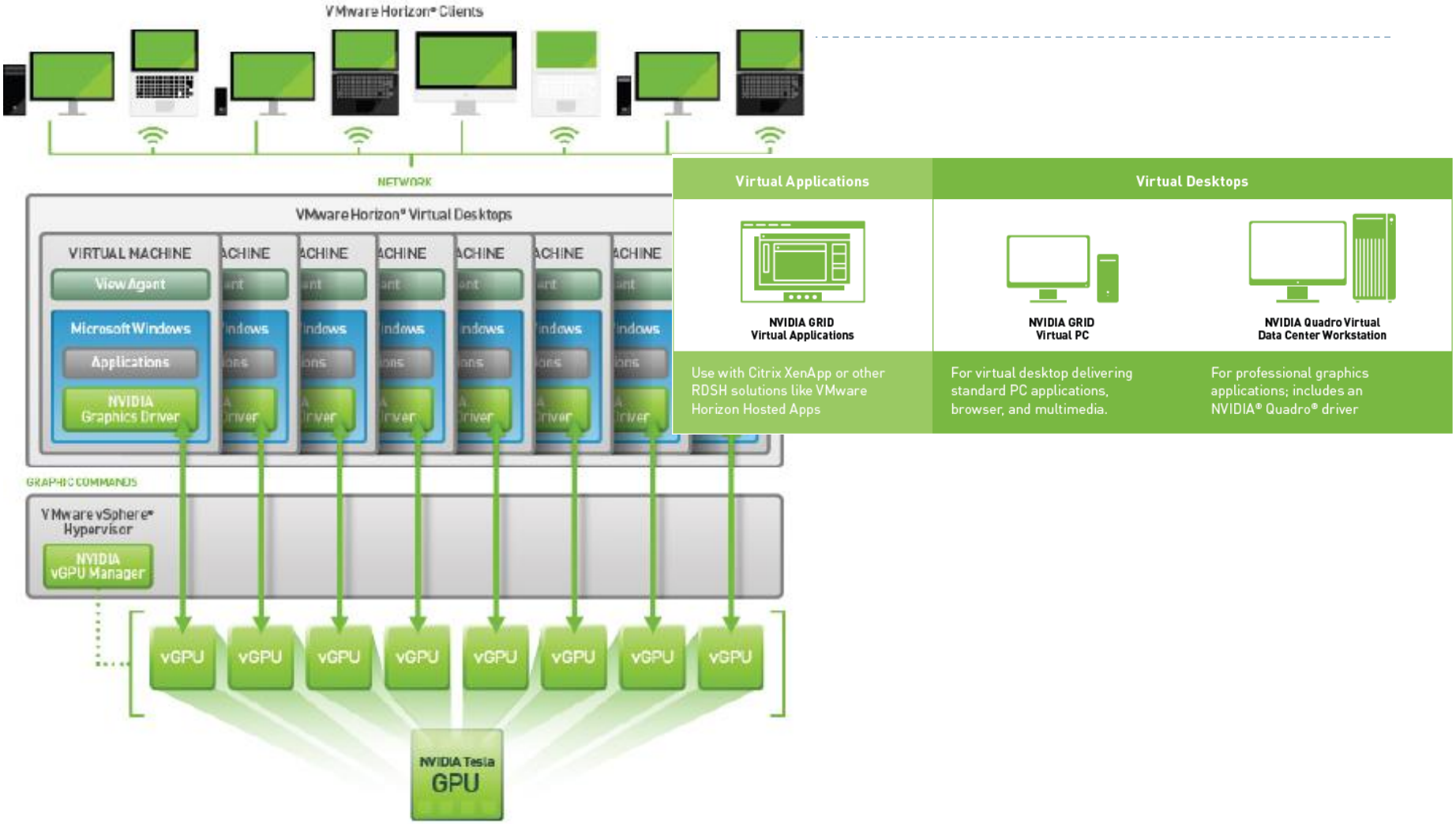



**At least 2 exits**:
- Delivery
- Completion signal

# Sources

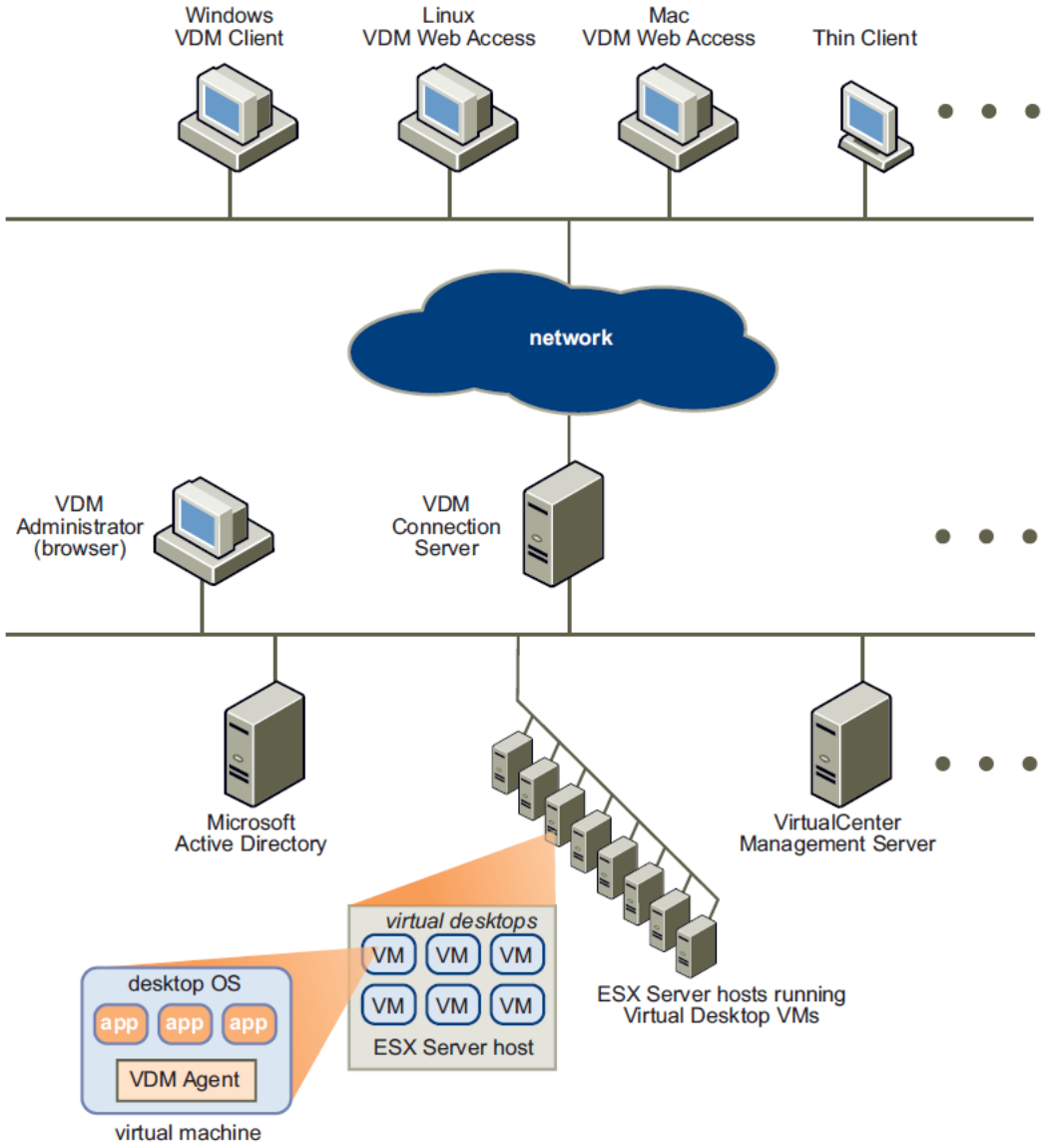

- Mendel Rosenblum, Carl Waldspurger: "**I/O Virtualization**"
  ACM Queue, Volume 9, issue 11, November 22, 2011
  URL: http://queue.acm.org/detail.cfm?id=2071256
- Avi Kivity, et al: kvm: The Linux Virtual Machine Monitor, Proceedings of the Linux Symposium, 2007
  - http://www.linux-kvm.com/sites/default/files/kivity-Reprint.pdf
  - http://kerneltrap.org/node/8088
- 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 Xen-Linux: Architecture, Implementation, and Optimizations,** OLS 2006
- **Xen: finishing the job,** lwn.net - 2009

# NVIDIA Virtual GPU (vGPU) + VMware Horizon

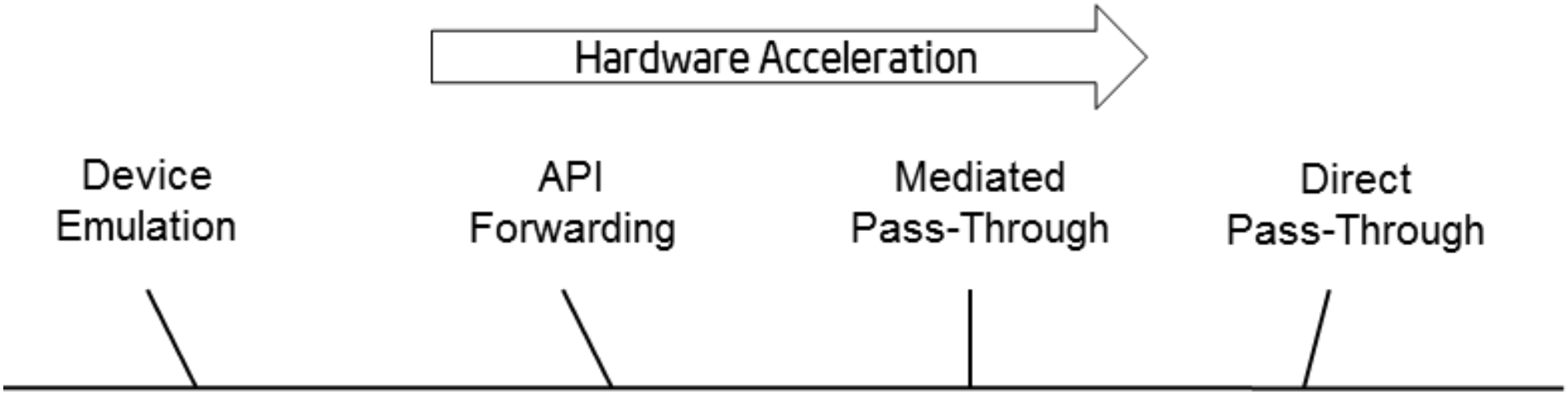



source: https://www.nvidia.com/en-eu/data-center/virtual-gpu-technology/

# VMware Virtual Desktop Infrastructure (VDI)



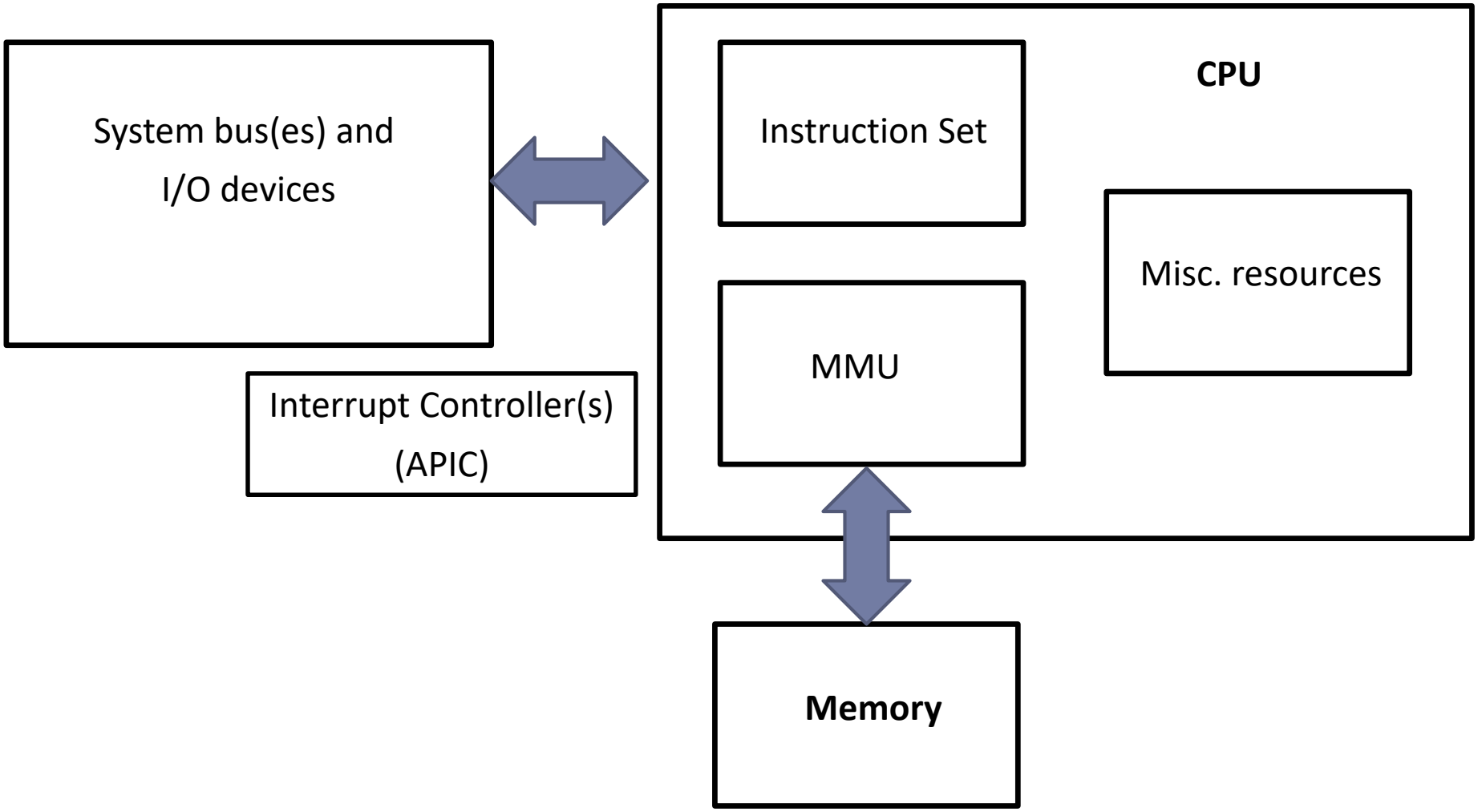
# VM I/O acceleration



# QEMU machine emulator

- Creator: Fabrice Bellard (circa. 2006)
- Machine emulator using a dynamic binary translator
  - Run-time conversion of target CPU instructions to host ISA
  - 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

# Emulated Platform



# 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

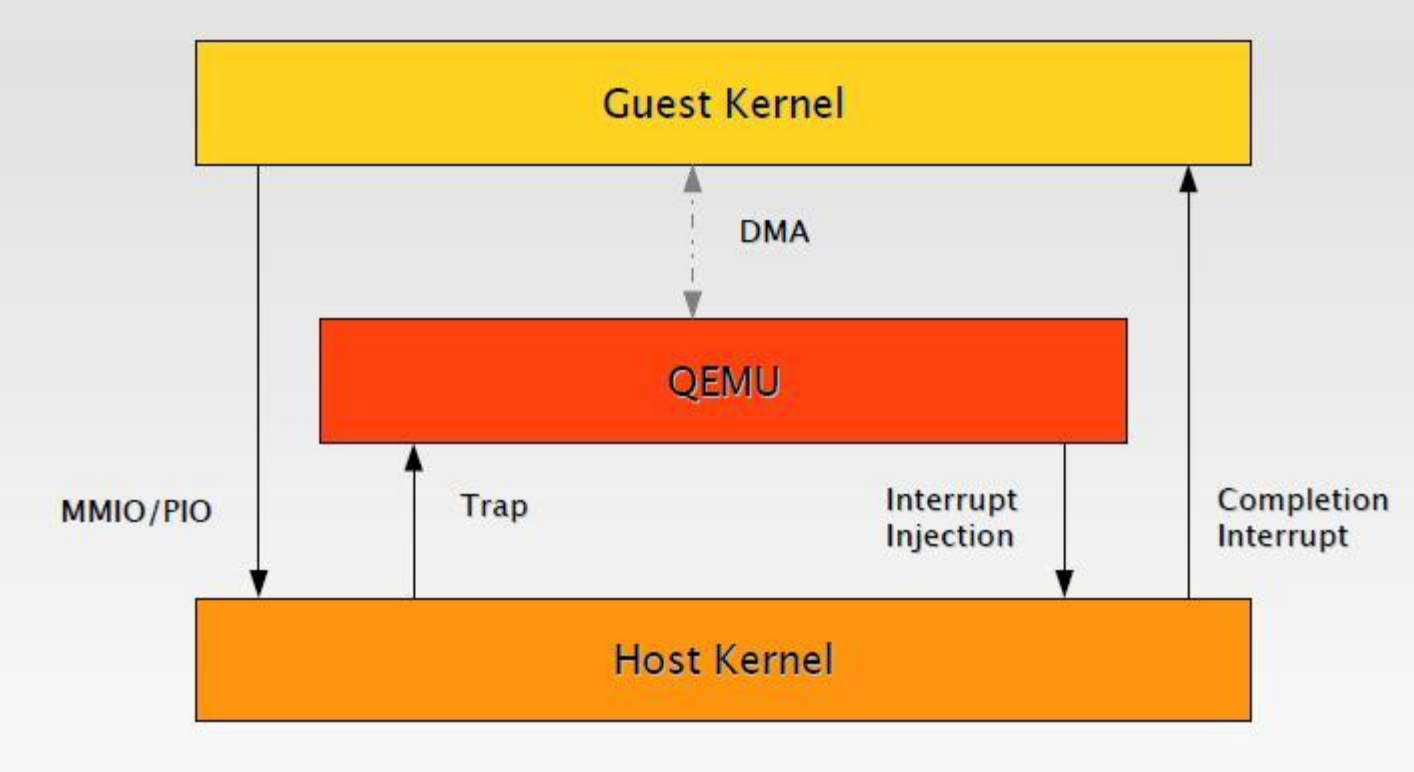
# Virtual disks

- Exported by host
- Physical device/partition
- … or logical device
- … or file (image) (e.g. : .qcow2, .vmdk, “raw” .img)
  - file format that describes containers for virtual hard disk drives
  - features: compression, encryption, copy-on-write snapshots

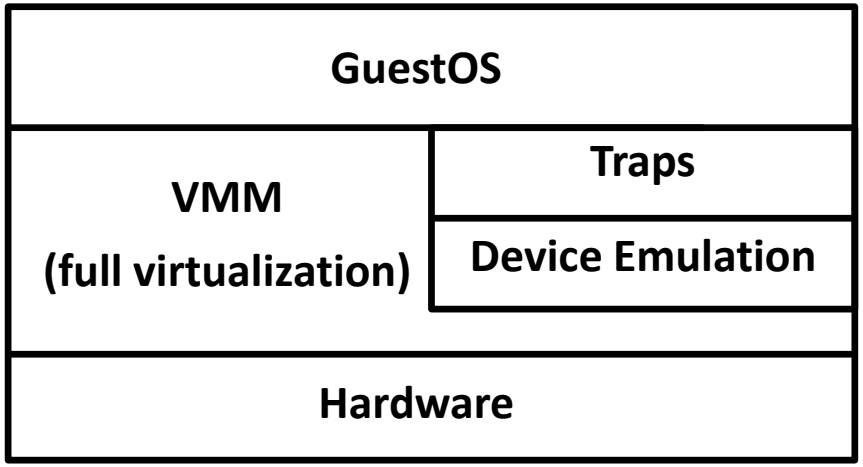
# Device model (with full-system virtualization)

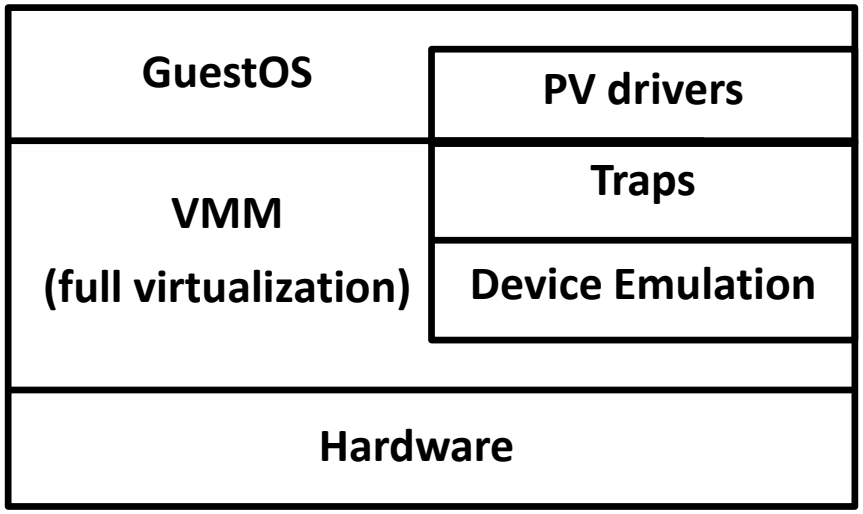
- VMM intercepts I/O operations from GuestOS and passes them to device model at the host
- Device model emulates I/O operation interfaces:
  - PIO, MMIO, DMA, …
- Two different implementations:
  - Part of the VMM
  - User-space standalone service

# Virtualized I/O flow



# Device emulation: full virtualization vs para-virtualization



# 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
```

# 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

# IO-MMU 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 IO-MMU can emulate multiple IO-MMU'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

# I/O path via virtio