CSIE 5310

I/O Virtualization

Prof. Shih-Wei Li

Department of Computer Science and Information Engineering
National Taiwan University

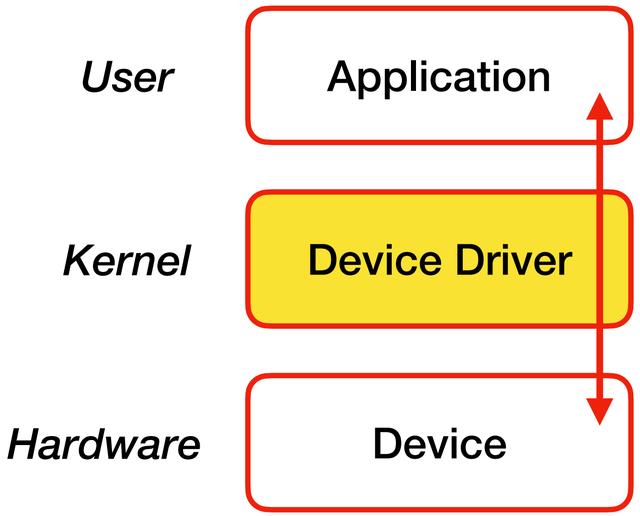
I/O Virtualization

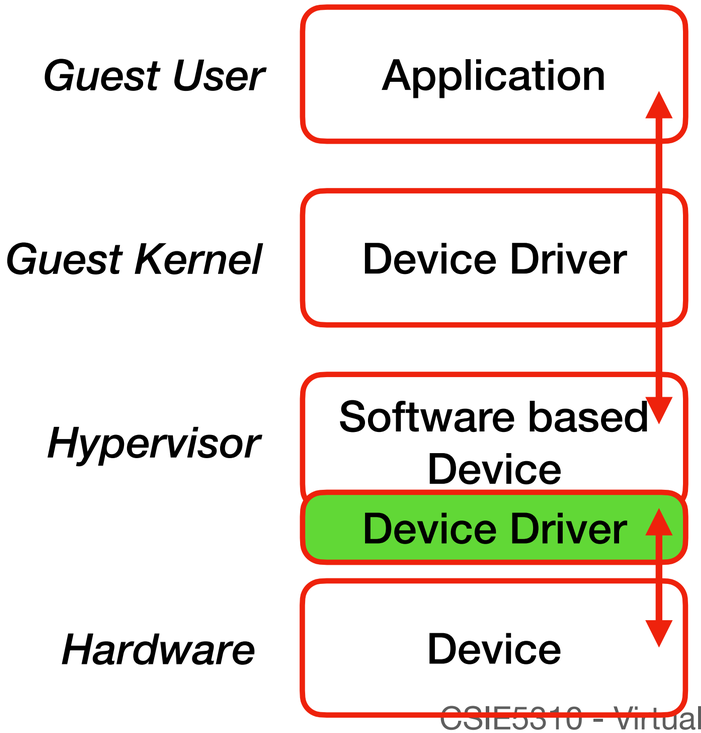
- Users interact with computers through I/O devices
  - Wide range of I/O devices needs to be virtualized
- Strategy: construct a virtual version for a given I/O device type, then virtualizing the I/O activity directed at the device
- Classic I/O virtualization is implemented via ***I/O interposition***
  - VMMs converts the I/O requests to the equivalent request for the underlying physical device to carry out such that VMs think they perform real I/Os

.

I/O support in traditional v.s. virtual systems

- Traditional I/O virtualization decouples virtual I/O activity from physical activity, so the VMMs can interpose on the I/O of its VMs





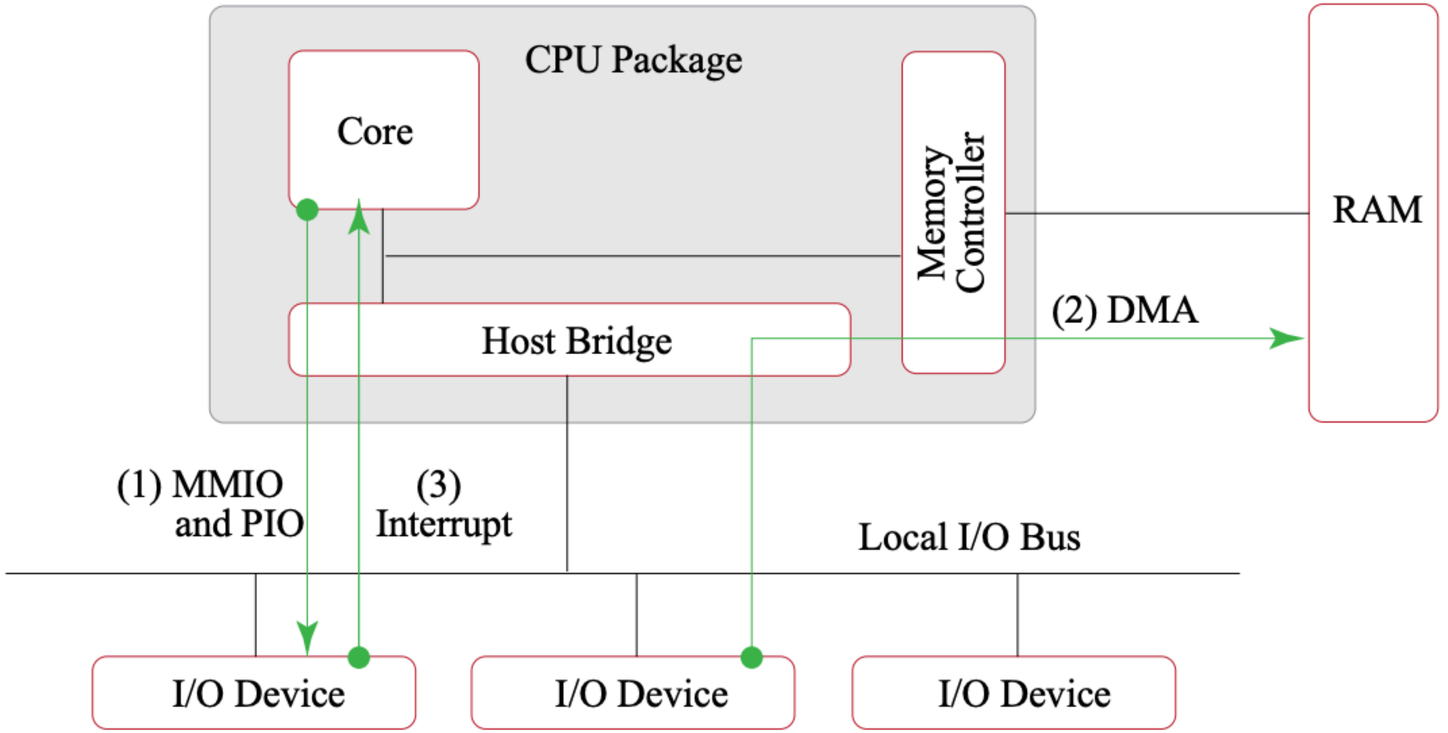
I/O Interposition

- Trap-and-emulate to interpose VM I/O activities
- Allows hypervisors to encapsulate the state of VM's I/O devices
  - Provide transparent VM portability — map virtual I/O devices to different hardware platforms with different physical I/O devices
  - Simplifies support for VM migration (from one machine to another), upgrading and maintaining host hardware resources
- Support features in virtual devices not natively provided by physical devices
  - Example: replicate disk write to recover from disk failures, storage deduplication, etc…

Agenda

- **Introduction to I/O subsystems**
- Introduction to I/O virtualization

Interactions between I/O devices, CPU, and memory



Physical I/O (1)

- The OS kernel figures out (probes) what devices are available on the machine
  - Firmware (BIOS, UEFI) exposes the description of available devices via some standard format like ACPI tables
  - Arm machines rely on vendor provided device tree (DT) files
- The OS kernel retrieves device information in the device probing
  - The hypervisor prepares the virtual device information for VMs to query

Physical I/O (2)

- An I/O device usually consists of registers, how does the software use the registers?
- Hardware provides mechanisms for CPUs to communicate with I/O devices
  - On modern Intel servers, communications CPUs and I/Os flow through the host bridges (Peripheral Component Interconnect Express (PCIe))
  - PCIe provides high throughput - posing a challenge when virtualizing it

PIO and MMIO (1)

- CPUs interact with the I/O devices via ***Port-mapped IO (PIO)*** and ***Memory-mapped IO (MMIO)*** instructions
  - CPU specifies the devices' associated addresses to use with the PIO and MMIO instructions

PIO and MMIO (2)

- PIO:
  - Addresses of PIO are called ports
    - Ports are limited to 16 bits
  - Used via the ***OUT*** and **IN** x86 instructions; not supported by Arm
    - The OUT and IN instructions write/read 1-4 bytes to/from the I/O devices
    - Example: ports 0x60-0x64 are reserved for PS/2 devices
  - Hypervisors can configure VMCS to trap the VM's PIO instructions

PIO and MMIO (3)

- MMIO:
  - Addresses of MMIO are physical memory addresses
  - CPUs access I/O devices via (regular) memory load/store instructions
  - The host bridge controller and memory controller are aware of the MMIO address association and route data accordingly

Direct Memory Access (DMA)

- The CPUs can use PIO or MMIO instructions to move data from I/O devices to memory and vice versa
  - Problem: transferring large of amount data is slow; the CPUs must be continuously and synchronously involved to perform explicit PIOs or MMIOs
- DMA's goal: avoid getting CPUs involved in data transferring
- DMA's approach: allows ***devices to read/write from/to the memory directly***
  - CPUs only initiate DMA operations; save CPU resources for running other tasks
  - DMA devices notify the CPU (via interrupts) when operations are complete

Interrupts (IRQs)

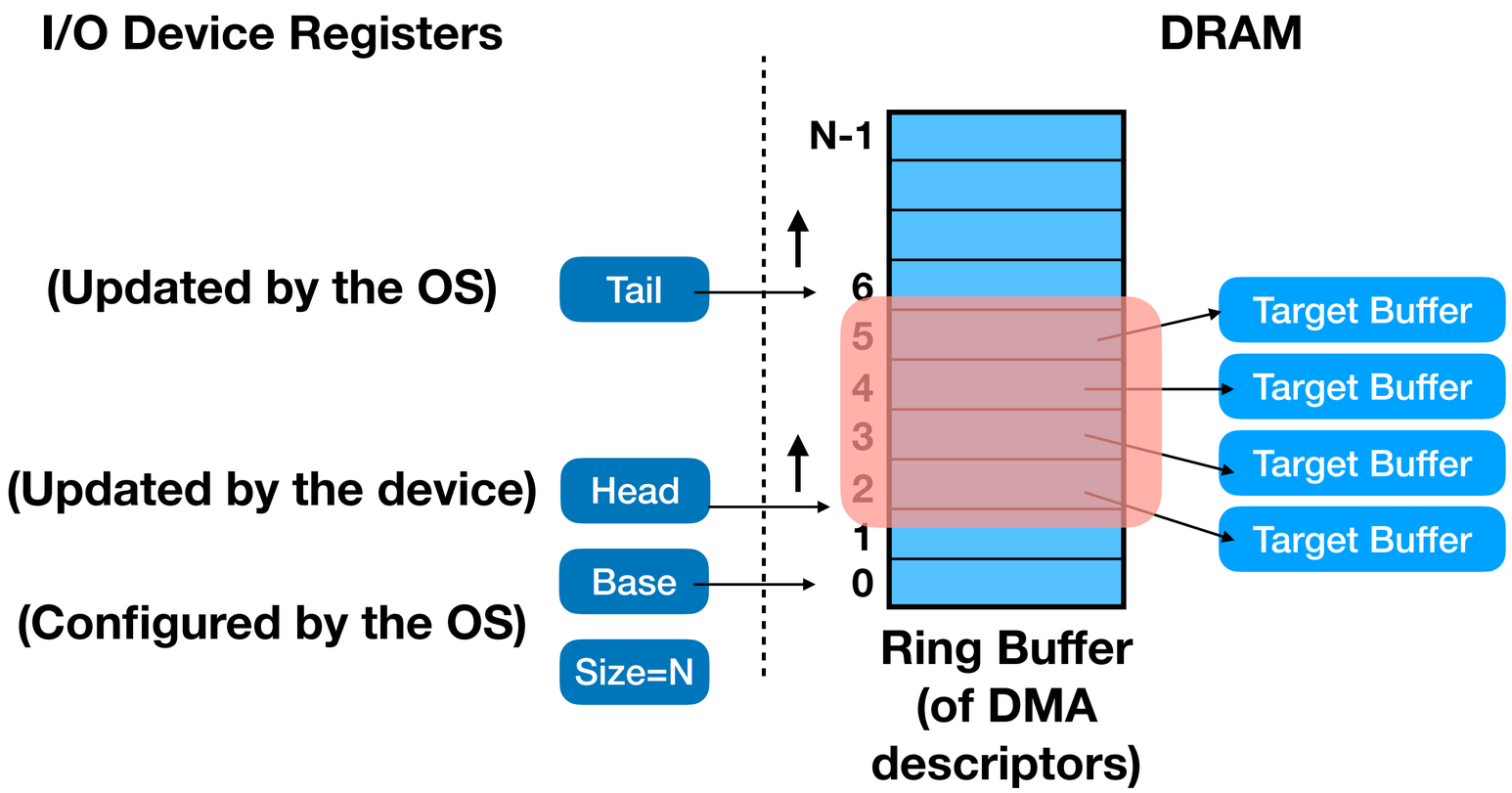
- I/O devices trigger **asynchronous events notification** at the CPU by issuing interrupts
- Each interrupt is associated with a unique number
  - On x86, each number corresponds to an entry in the Interrupt Descriptor Table (IDT); the entry contains a pointer to the respective interrupt handler
  - When an interrupt fires, the hardware invokes the associated interrupt handler from the IDT
- The interrupt handlers can be provided by the device drivers or the OS kernel
  - A CPU can trigger Inter-processor interrupts (IPIs) to notify other CPUs; why is IPI useful?
- The ***interrupt controller*** handles interrupt-related operations performed by the OS kernel or VMM
  - We will talk about that in the next lecture

Driving Devices through Ring Buffers (1)

- Commodity devices, such as PCIe SSDs and Network Controllers (NICs), deliver high throughput I/Os with 10-100 Gbits/s bandwidth

- These devices stream I/Os via one or more producer/consumer ***ring buffers:*** arrays in memory shared between the OS device driver and the associated device

Driving Devices through Ring Buffers (2)



Driving Devices through Ring Buffers (3)

- ***DMA descriptors***
  - ***T***ypically specifies the I/O status and the address and size of ***DMA target buffers***: the memory areas used by the device DMAs to write/read incoming/outgoing data

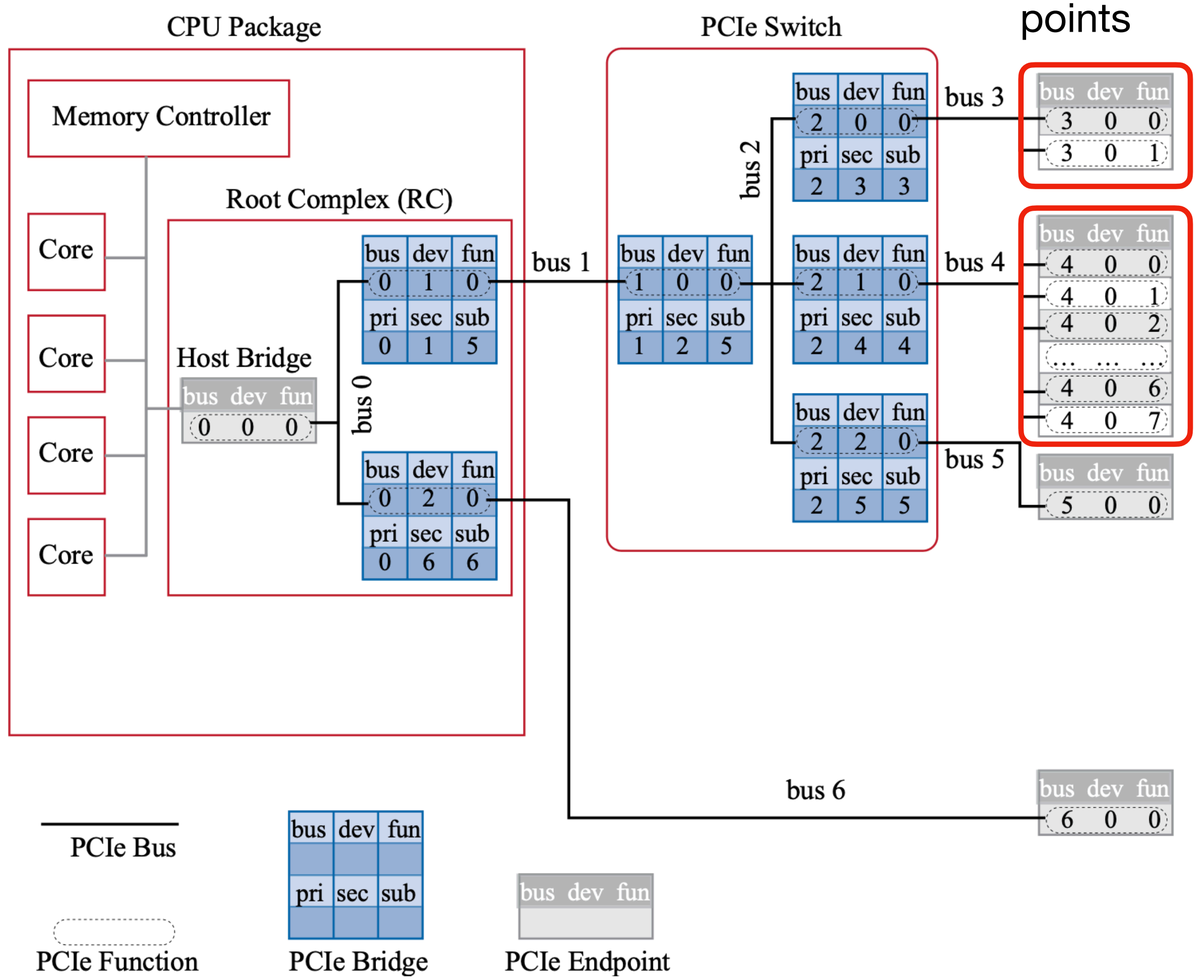
Driving Devices through Ring Buffers (4)

- The devices must know the direction of the memory operations: either (1) transmitted from memory into the device or (2) received from the device into memory
  - Disk drives specify the direction in the DMA descriptor
  - NIC employs different rings for transmit (Tx) and receive (Rx) activities; it may employ multiple Rx/Tx queues for scalability!
- Upon initialization, the OS device driver allocates the rings and configures the device with the ring size and base locations
- Devices notify the OS kernel by triggering an interrupt when data is sent/received

PCIe (1)

- PCIe is abbreviation of PCI Express: a specification of a local serial bus that is standardized by the industry (PCI-SIG)
- The topology of the PCIe fabric arranges as a tree:
  - Root: a ***host bridge*** that channels all types of PCIe traffic to/from the CPU cores/memory, including: PIO, MMIO, DMA, and interrupts
  - Edges: PCIe buses
  - Nodes: PCIe functions; functions are either bridges (connect to buses) or endpoints:
    - Endpoints correspond to individual I/O channels in physical PCIe devices: example: dual-port NIC has two endpoints

Two difference devices, each with multiple end points



From: Figure 6.5 in Hardware and Software Support for Virtualization

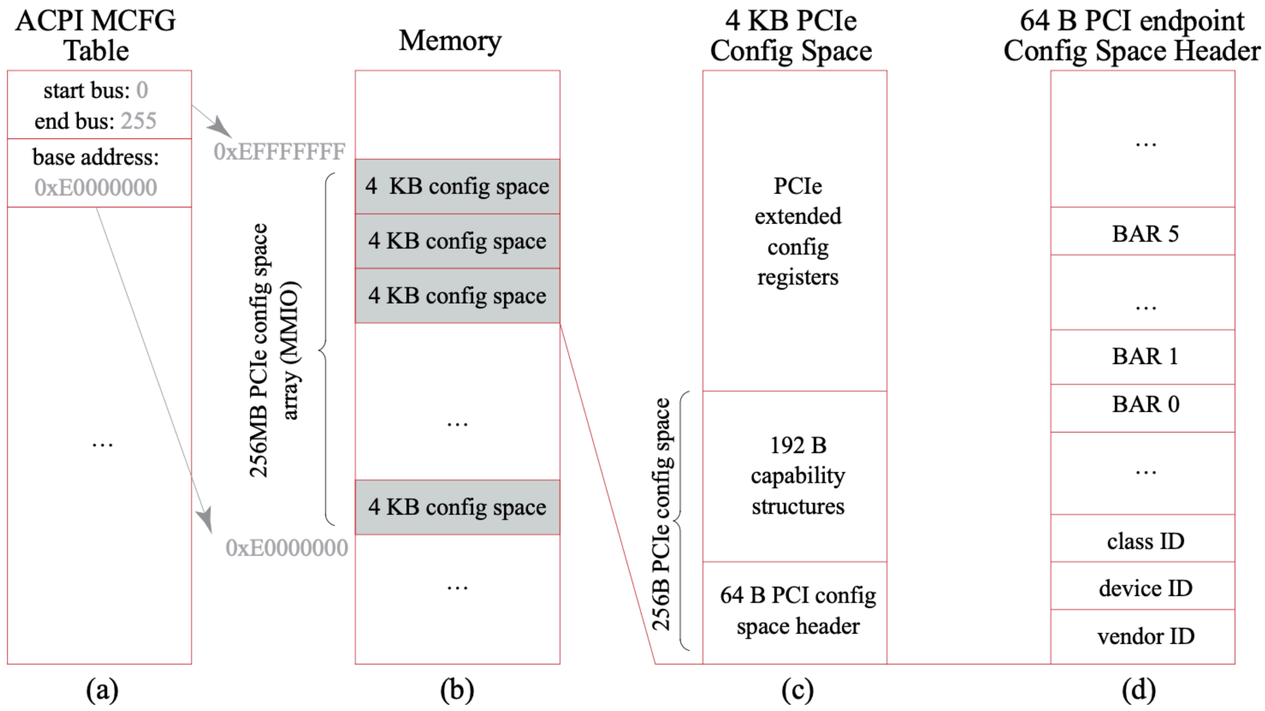
PCIe (2)

- The OS finds the ***size*** and ***address*** of the ***PCIe configuration space array*** (includes device information) from the MCFG (Memory-Mapped Configuration) ACPI (Advanced Configuration and Power Interface) table

- A (4KB) configuration space contains information of a PCIe node: a PCI space that constitutes a ***config space header*** (in (d)) that specifies device attributes:

  - Functional class of the device (network, storage, etc.), vendor ID, device ID

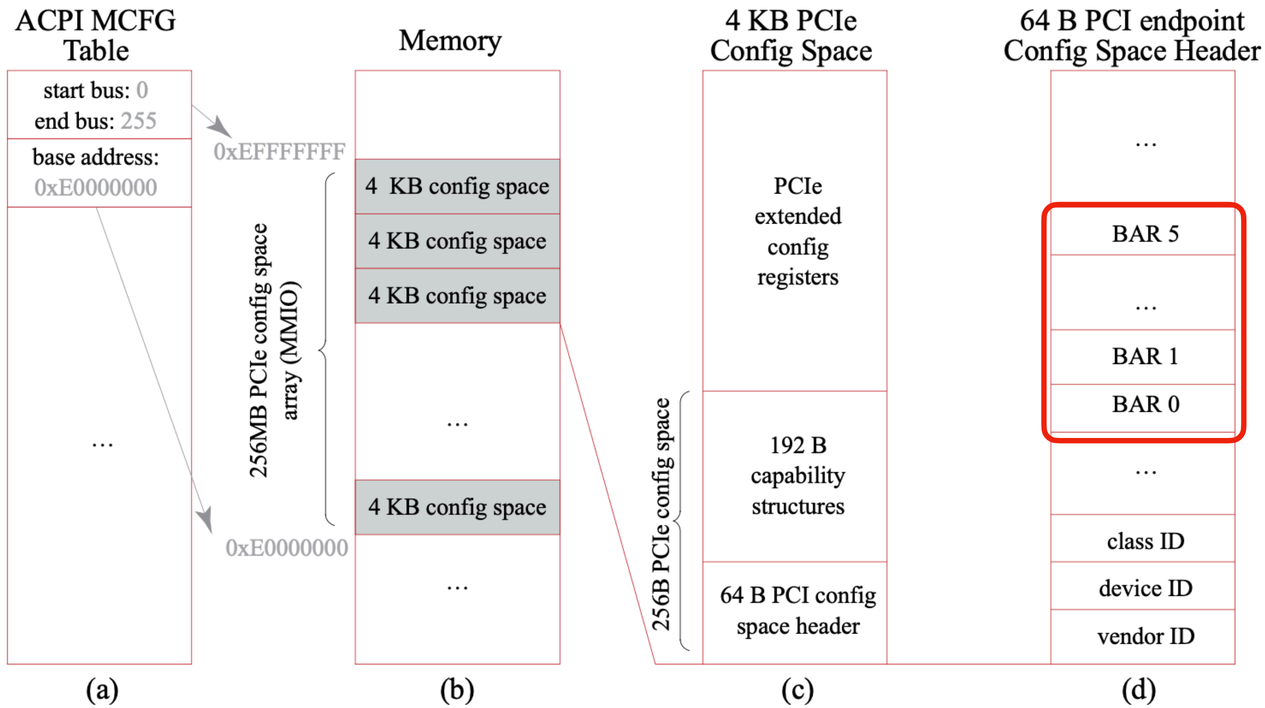
  - The device attributes allow the OS kernel to identify and initialize the device



From: Figure 6.6 in Hardware and Software Support for Virtualization

PCIe (3)

- Each PCIe endpoint can have up to six ***Base Address Registers (BARs)***:
  - Each BAR publicizes the MMIO addresses (of the device registers) to be used by the OS kernel; example: the head/tail register for the ring buffer
- PCIe supports ***Message Signaled Interrupts (MSIs)***
  - Allows a device to fire MSI interrupts to the LAPIC (local advanced programmable interrupt controller)



From: Figure 6.6 in Hardware and Software Support for Virtualization

Agenda

- Introduction to I/O subsystems
- **Introduction to I/O virtualization**

Objectives

- The guest OS believes that it exclusively controls all the physical I/O devices
  - The VM discovers, initializes, and drives I/O devices (potentially reusing the native device drivers)
  - Hypervisors must manage physical I/Os for themselves and VMs to enable sharing
    - Provide fake I/O devices for their hosted VMs

Different ways for Virtualizing I/O

- Includes 3 types:
  - I/O Emulation
  - I/O Paravirtualization (or paravirtualization)
  - Direct Device Assignment (device passthrough)
- The hypervisor support depends on hardware availability
  - Device passthrough requires special hardware support

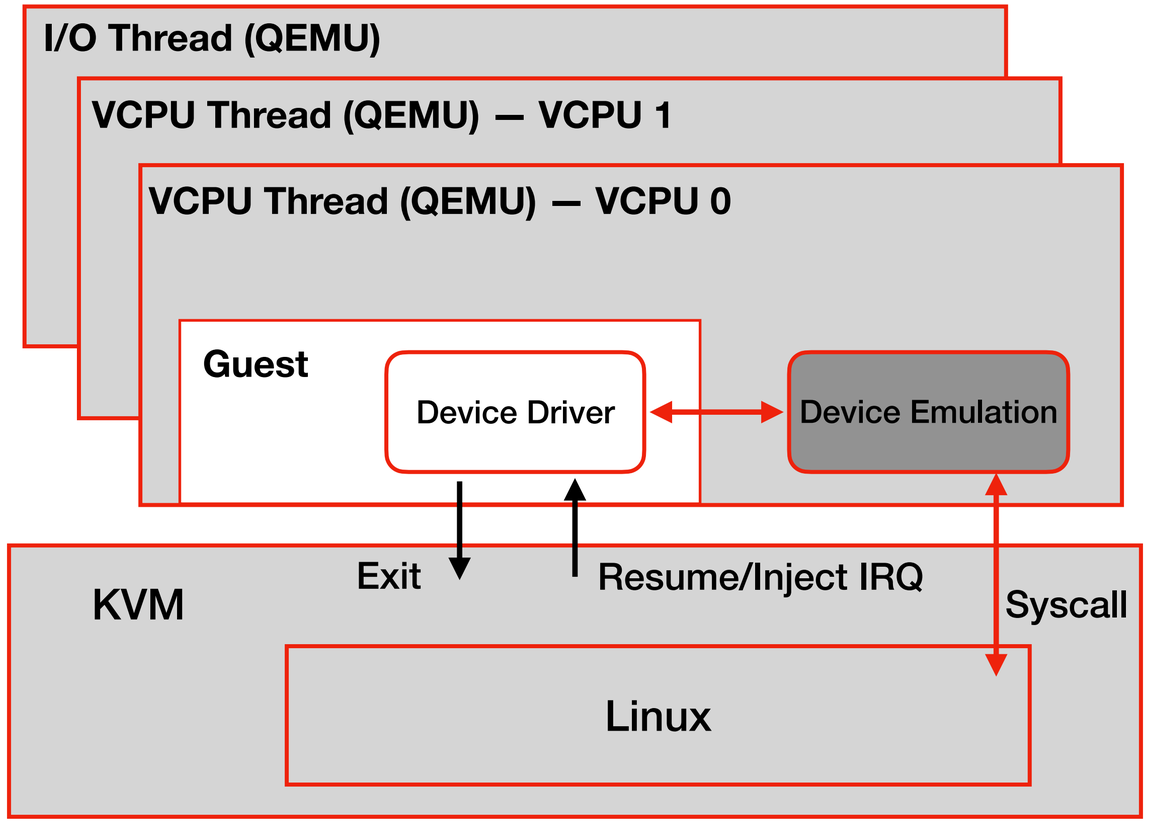
Agenda

- Introduction to I/O subsystems
- Introduction to I/O virtualization
  - **I/O Emulation**
  - I/O Paravirtualization
  - Direct Device Assignment

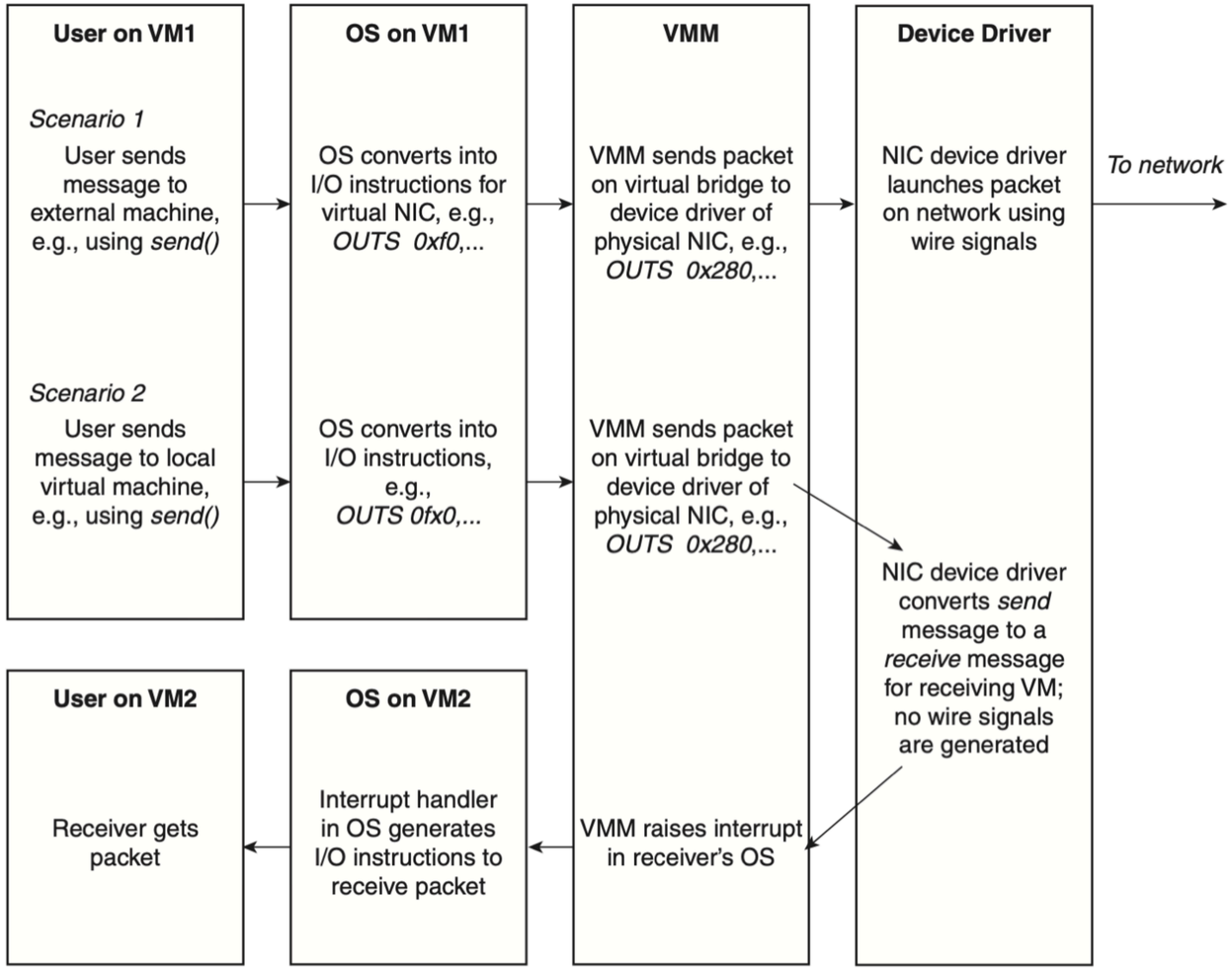
I/O Emulation

- Trap VMs' accesses to I/O devices via *MMIO and PIO* so the hypervisor can intercept
  - Adopting trap and emulate (I/O interposition)
- How do hypervisors interpose PIOs or MMIOs?
  - On x86, hypervisors can configure the VMCS to trap PIO instructions
  - Hypervisors unmap MMIO addresses in NPT or SPT to trap MMIOs
- To support I/O emulation:
  - Hypervisors respond to the VM's MMIOs and PIOs by reflecting the I/O operations on the virtual I/O device
  - Inject virtual interrupts to VMs for the virtual devices

I/O Emulation in KVM



I/O Emulation: Virtualizing the Network Interface



I/O Emulation: Virtualizing the e1000 NIC

- Guest OS can directly use the e1000 driver
- QEMU includes the e1000 emulation (file hw/net/e1000.c in QEMU)

```
00:03.0 Ethernet controller: Intel 82540EM Gigabit Ethernet Controller
        Flags: bus master, fast devsel, latency 0, IRQ 11
        Memory at febc0000 (32-bit, non-prefetchable) [size=128K]
        I/O ports at c000 [size=64]
        Expansion ROM at feb40000 [disabled] [size=256K]
        Kernel driver in use: e1000
```

Agenda

- Introduction to I/O subsystems
- Introduction to I/O virtualization
  - I/O Emulation
  - **I/O Paravirtualization**
  - Direct Device Assignment

I/O Paravirtualization (1)

- I/O emulation-based virtualization is easy to implement but incurs performance overhead
- Problem: Physical devices were not designed originally for virtualization
  - (1) Sending/receiving a single Ethernet frame via e1000 involves multiple register accesses
    - Multiple VM exits (traps) per frame are required in an I/O emulation setup!
  - (2) Mismatch of legacy hardware interface with virtualization interface
    - Hypervisors configure trap-and-emulate on devices I/O accesses based on page granularity
    - I/O registers are usually co-located within the same page; causing unnecessary traps:
      - Example: e1000's registers are packed; cannot avoid trapping STATUS register accesses because ICR (has to clear up upon read) is co-located in the same page

I/O Paravirtualization (2)

- Design ***paravirtual devices*** with virtualization in mind to address the limitations of I/O emulation; the VMs and the hypervisors agree upon a device specification
  - Goal: minimize the number of VM exits to achieve significant performance improvement
- Require special ***"paravirtual" device drivers*** that are compatible with the hypervisor
  - The drivers do not work for a real hardware

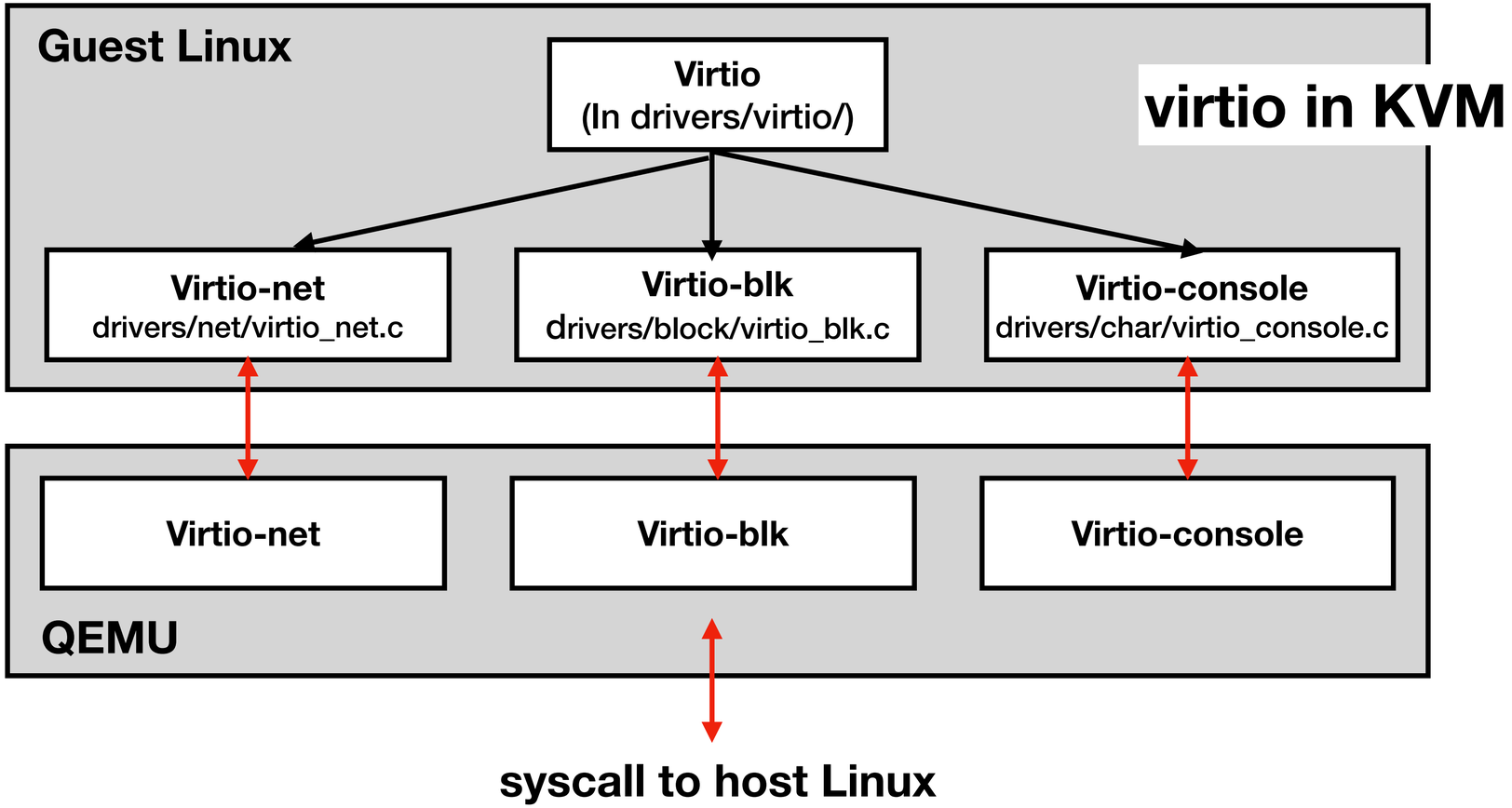
Virtio (1)

- The de-facto paravirtual I/O framework for KVM/QEMU
  - Offer a common guest-host interface and communication mechanism
  - Supports various paravirtual devices: network (virtio-net), block (virtio-block), char (virtio-console), etc.
- Virtio devices are exported to VMs like other physical/emulated devices

```
00:02.0 VGA compatible controller: Device 1234:1111 (rev 02)
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet
00:04.0 Ethernet controller: Red Hat, Inc Virtio network device
00:05.0 SCSI storage controller: Red Hat, Inc Virtio block device
```

lspci in VM

Virtio (2)


Guest Linux
Virtio
(In drivers/virtio/)
virtio in KVM
Virtio-net
drivers/net/virtio_net.c
Virtio-blk
drivers/block/virtio_blk.c
Virtio-console
drivers/char/virtio_console.c
Virtio-net
Virtio-blk
Virtio-console
QEMU
syscall to host Linux

Virtio (3)

- Incorporate a ring buffer data structure called **virtqueue**
  - VMs post descriptors/buffers to virtqueue; the VMM consumes them
  - VMs' normal virtqueue operations do not exit to the hypervisor
    - Unlike VMs' MMIO/PIO accesses to emulated devices that cause a lot of exits
    - VMs can intentionally trigger exits: make the "virtqueue_kick" hypercall to hypervisor to explicitly request service; the hypercalls are usually batched to reduce exit overheads

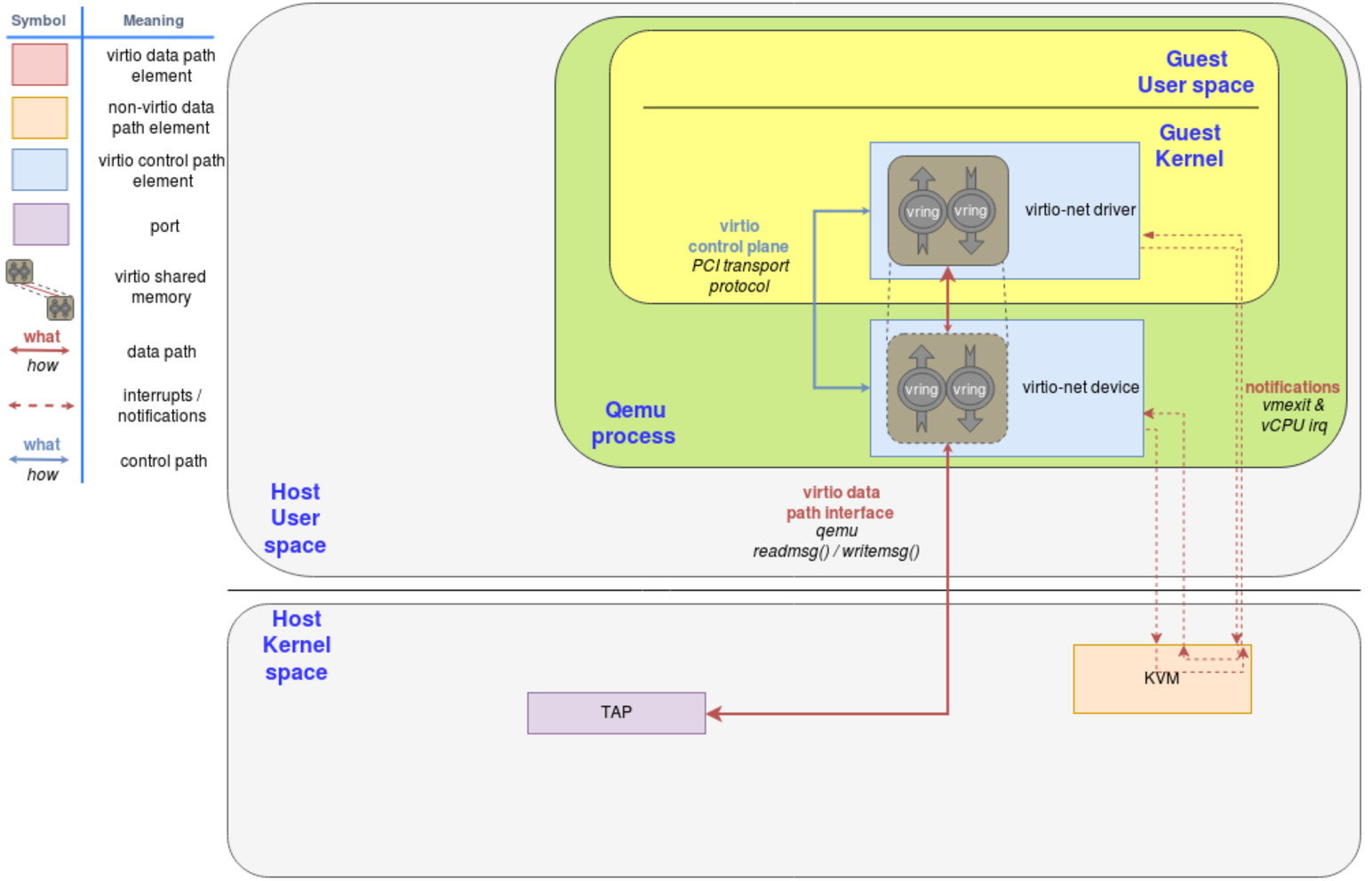
Virtio (4)

- To further reduce interrupts and exits, virtqueue is associated with two modes of execution: ***NO_INTERRUPT*** and ***NO_NOTIFY***
  - *NO_INTERRUPT mode*: the VM could turn the mode on; if on, the hypervisor stops delivering virtual interrupts associated with the virtio device until it is turned off
    - virtio_net uses this mode in the Tx (transmission) virtqueue to avoid receiving virtual interrupts until a transmission finishes
  - *NO_NOTIFY mode*: the hypervisor could turn the mode on to disable notifications from the V
    - virtio_net routinely uses this mode to efficiently handle bursty TCP traffic

Virtio (5)

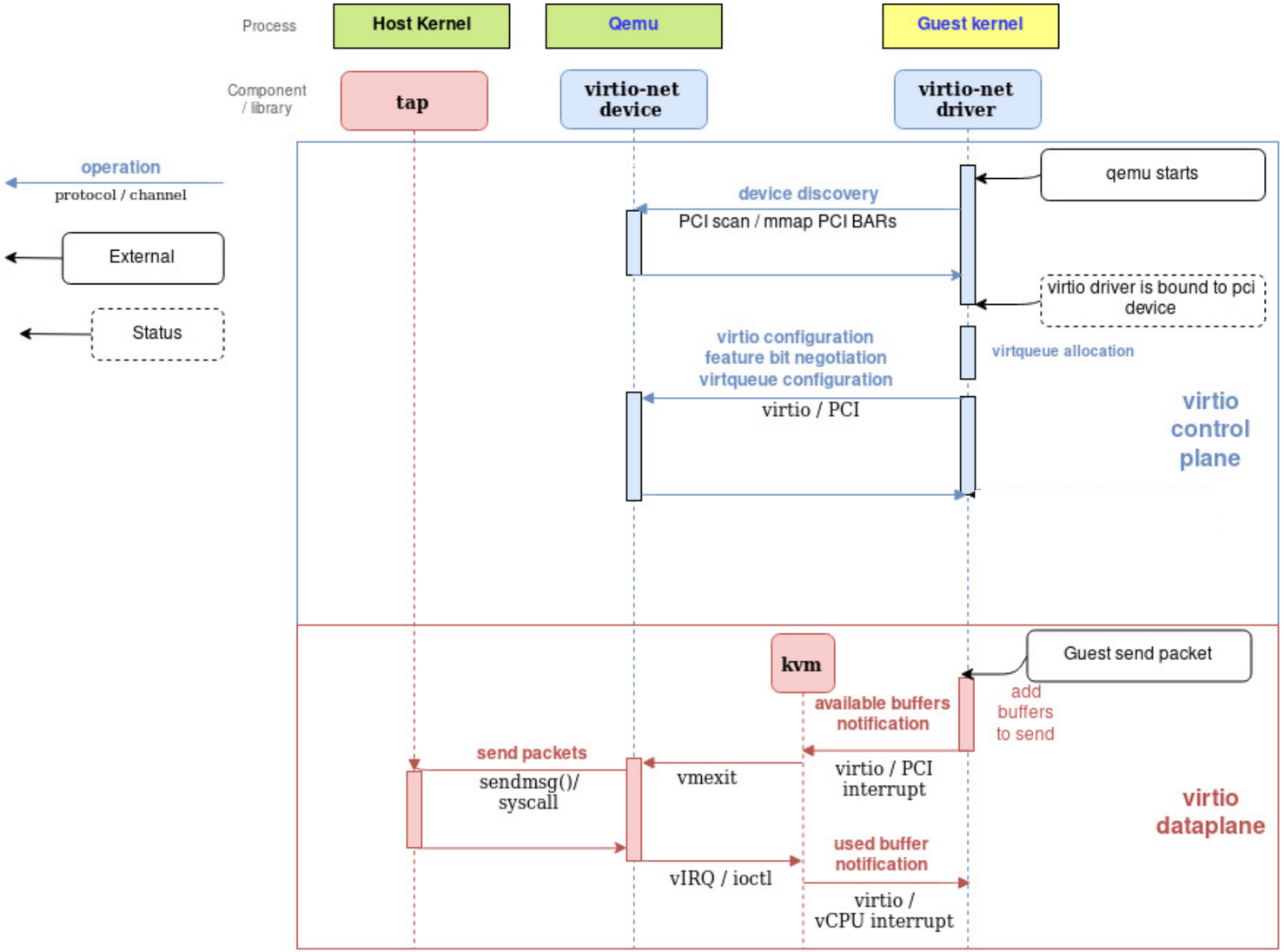
- From https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net: “The virtio drivers must be able to allocate memory regions that both the hypervisor and the devices can access for reading and writing, i.e., via memory sharing terminology”

virtio_net on QEMU



From: https://www.redhat.com/rhdc/managed-files/2019-09-12-virtio-networking-fig1.png

virtio_net on QEMU: send buffer flow diagram



From: https://www.redhat.com/rhdc/managed-files/2019-09-12-virtio-networking-fig2.png.jpg

Virtio Performance

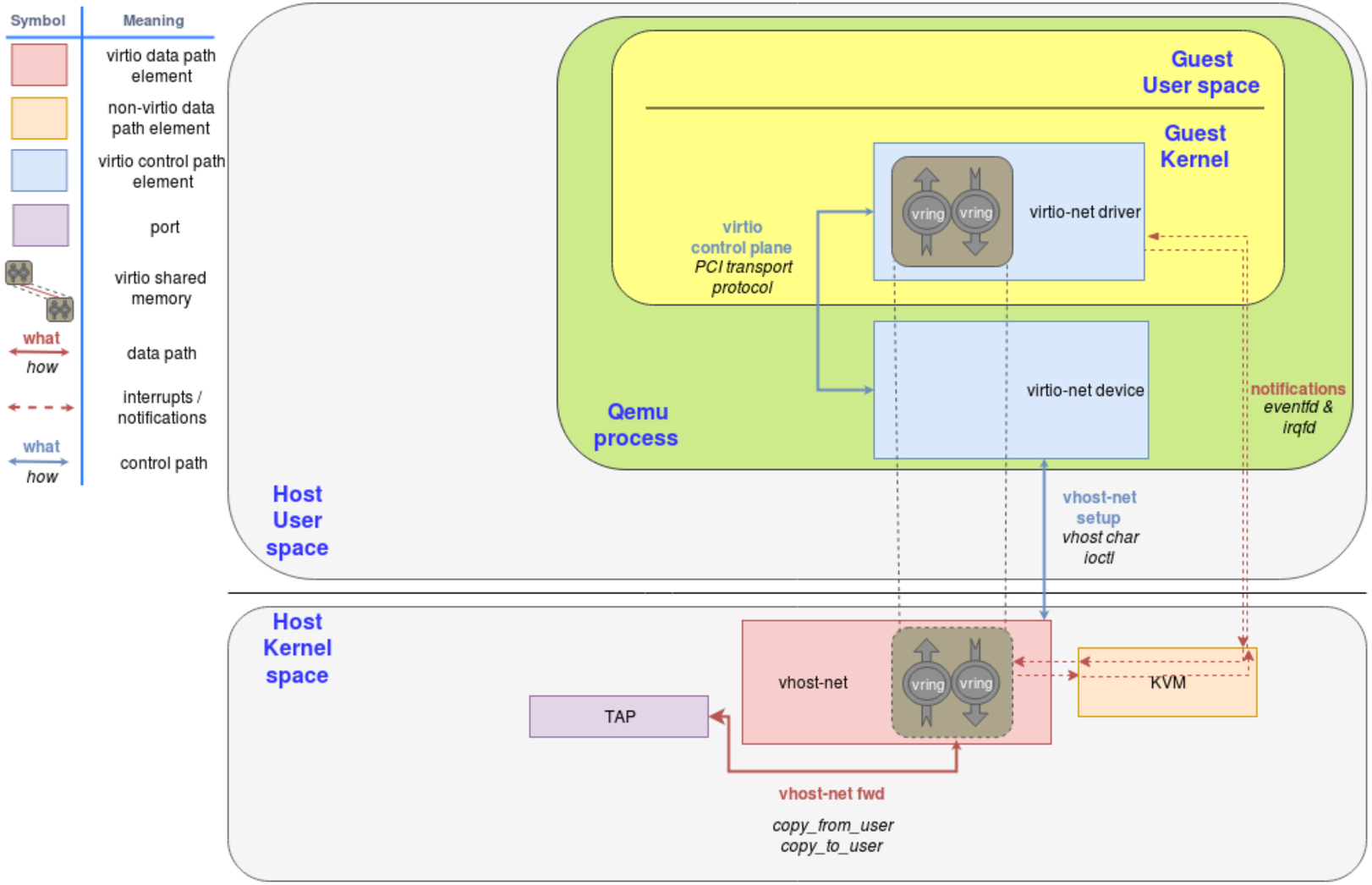
Table 6.2: Netperf TCP stream running in a Linux 3.13 VM on top of Linux/KVM (same version) and QEMU 2.2, equipped with e1000 or virtio-net NICs, on a Dell PowerEdge R610 host with a 2.40GHz Xeon E5620 CPU

	Metric	e1000	Virtio-net	Ratio
Guest	throughput (Mbps)	239	5,230	22x
	exits per second	33,783	1,126	1/30x
	interrupts per second	3,667	257	1/14x

Virtio Optimization: vhost (1)

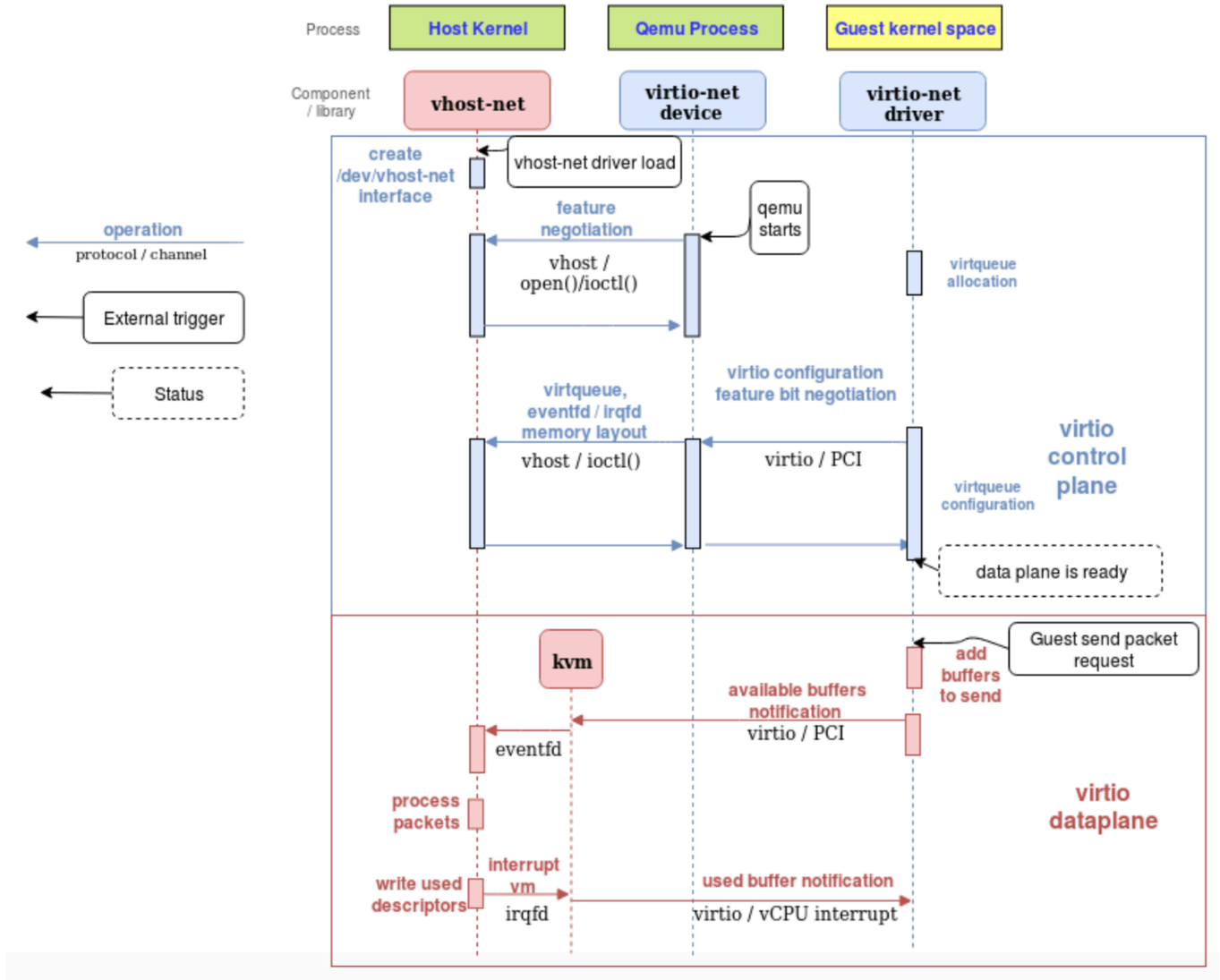
- High throughput requirement for modern NICs
- KVM/QEMU supports **vhost-net** for optimizing virtio-net performance
  - vhost-net is a kernel driver that implements efficient data-plane for virtio-net
- Move data-plane to the Linux kernel to process packets in kernel space
  - Reduce context switch overhead significantly
- Control planes still stays in QEMU
  - Mostly used during device initialization

Virtio Optimization: vhost (2)



From: https://www.redhat.com/rhdc/managed-files/2019-09-12-virtio-networking-fig3.png

Virtio Optimization: vhost (3)



From: https://www.redhat.com/rhdc/managed-files/2019-09-12-virtio-networking-fig4.png

Agenda

- Introduction to I/O subsystems
- Introduction to I/O virtualization
  - I/O Emulation
  - I/O Paravirtualization
  - **Direct Device Assignment**

Virtual I/O with hardware support (1)

- Use software approach to disallow VMs to directly access physical I/O devices but the interposing incurs performance overhead
- Hypervisors support ***direct device assignment*** (or device passthrough):
  - Assign physical devices directly to VMs if forgoing security
  - VMs can use the drivers for the hardware device directly
  - Much smaller performance overhead than other approaches since the virtualization layer is removed
- Problems:
  - Security: a malicious user can control the device to DMA hypervisor's or other VMs' memory
  - Scalability: physical hardware devices are limited in quantity

Virtual I/O with hardware support (2)

- Hardware provides support for direct device assignment
  - IO Memory Management Unit (***IOMMU***) helps with security
  - Single-Root I/O (***SRIOV***) helps with scalability
- The features make direct device assignment a viable approach

Issues with Virtual Machine DMA

- Issues with DMA using VM owned devices:
  - The device can access arbitrary memory locations; What if a malicious device driver controls the DMA device?
  - The guest owned device cannot DMA buffers allocated guest physical addresses (gPA); DMA only work on host physical addresses (hPAs)

IOMMU (1)

- Major chip vendors introduced IOMMU to address the issues in VM DMA:
  - Intel VT-d (technology for direct I/O), AMD I/O virtualization technology, and Arm's System Memory Management Unit (SMMU)
- IOMMUs consist of two main components:
  - DMA mapping engine (***DMAR***) and interrupt remapping engine (***IR***)
  - DMAR allows DMAs to be carried out with IO virtual addresses (***IOVAs***), which the IOMMU translates into host physical addresses
  - IR translates interrupt vectors fired by the devices based on an interrupt translation table (maps guest IRQ number to hardware's) configured by the hypervisor; preventing the VM from issuing arbitrary interrupts to the hypervisor

IOMMU (2)

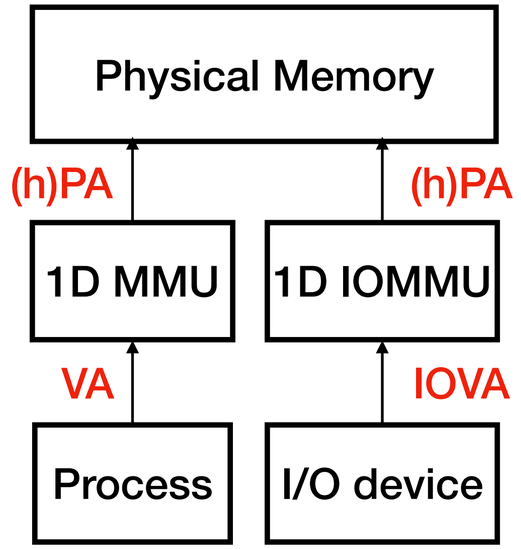
- DMAR works for both a bare-metal and VM setup:
  - Why for bare-metal?
- The IOMMU walks the page table similarly to the MMU to translate IO virtual addresses (***IOVAs***) to physical address (hPA)
  - Leverage the translation for memory access control
  - Checks if the mapping is valid, or the access permissions mismatch
- IOMMU caches page table translations using an IOTLB
- IOMMU can generate page faults; do not generally want this to happen, why?

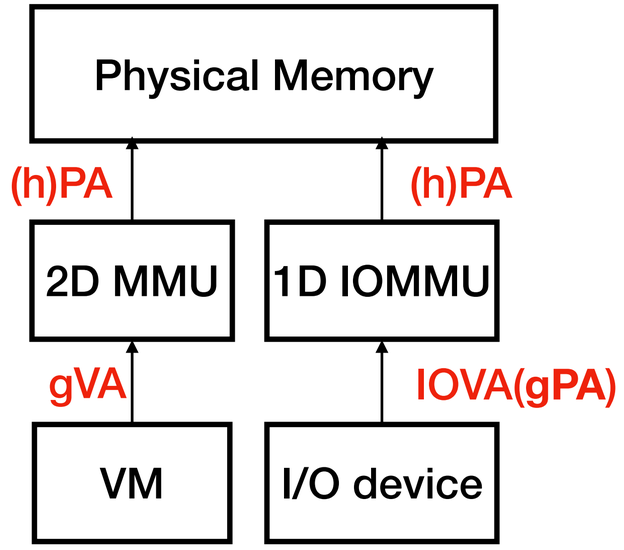
IOMMU (3)

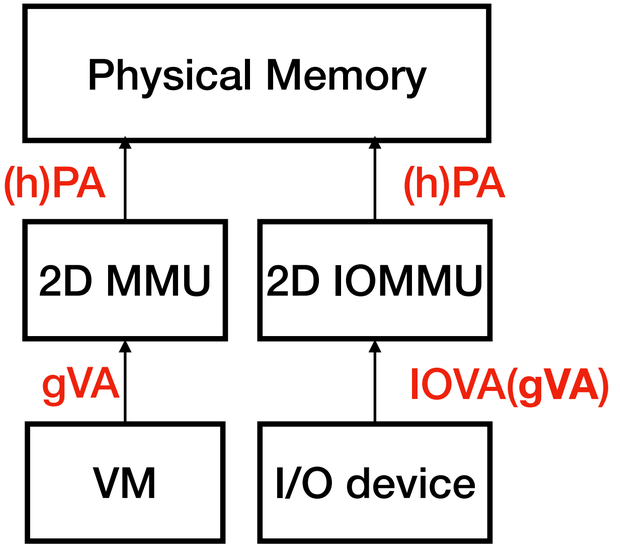
- VMs use guest physical addresses for DMA operations
- VMMs control how IOVAs are translated to physical addresses (hPA)
  - VMMs use IOVAs identical to the guest's gPAs: Pin the entire range of VM gPA to physical memory to avoid page faults

IOMMU (4)

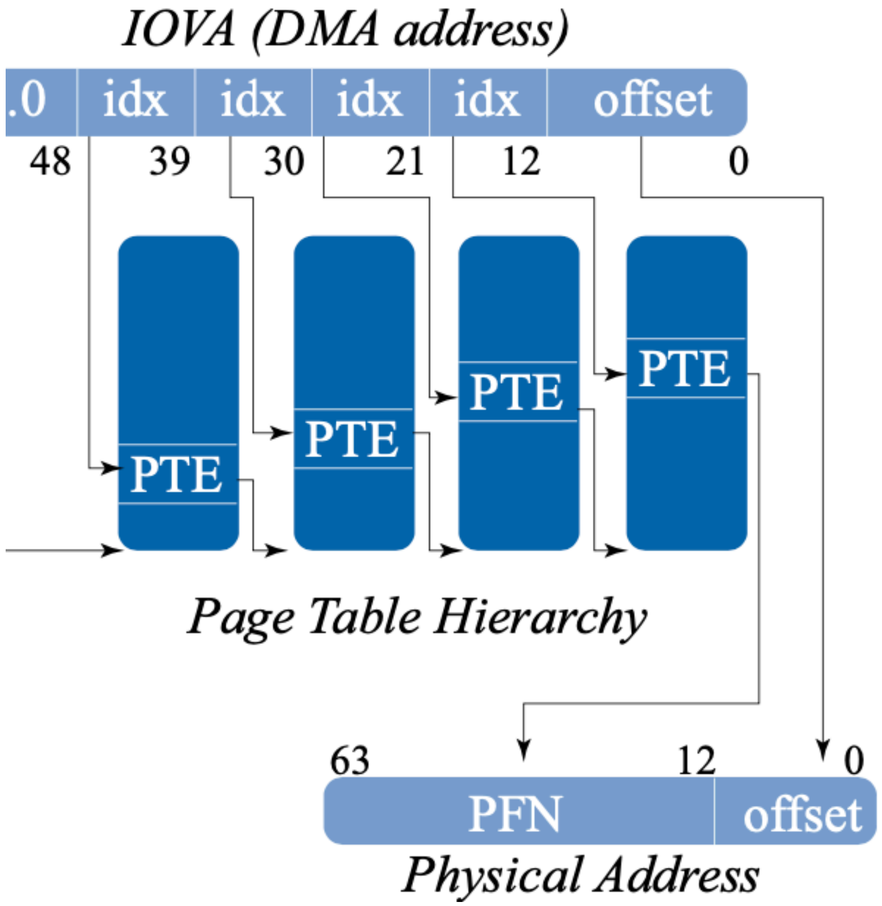
- IOMMU supports 1 or 2 dimensional page table walk
  - The former is used for kernel/hypervisor/VM protection; the latter is for guest OS to protect its DMA







IOMMU for VT-d



SRIOV (1)

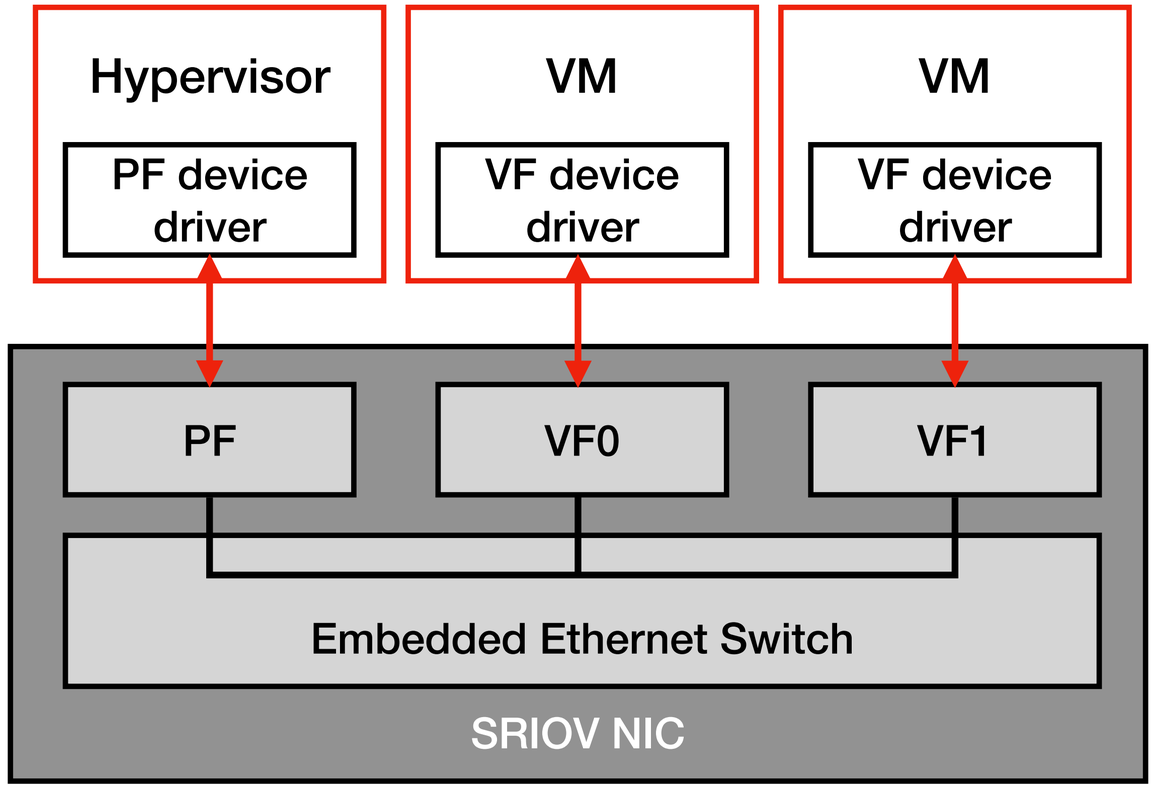
- Problems:
  - A hardware only includes a limited number of physical devices
  - Economically infeasible to purchase a physical device for each VM
- Extends PCIe to support devices that can “self-virtualize”
  - Goal: to present multiple instances of a single hardware device to software; each is assigned to a different VM and can be used directly

SRIOV (2)

- The SRIOV device multiplexes itself at the hardware level:
  - Defined to have at least one **Physical Function (PF)** and multiple **Virtual Functions (VFs)**
  - A PF is a standard PCIe function (i.e. has a standard configuration space)
  - Host software (hypervisor or OS kernel) can allocate, deallocate, or configure VFs
  - A VF implements a subset of the standard PCIe function with a limited configuration space
    - Example: a VF cannot deallocate other VFs and has no power management capabilities (shared with its respective PF)

SRIOV (3)

PF and VF drivers are identical!



SRIOV (4)

- A VF assigned to a VM can initiate DMAs
  - Hypervisors remain uninvolved in the I/O path; the device is passthrough to the VM
- Implementations SRIOV NICs from Intel and Mellanox can enable up to 512 VFs

```
06:00.0 Ethernet controller: Intel Ethernet Controller 10-Gigabit X540-AT2
        Subsystem: Intel Corporation Ethernet 10G 2P X540-t Adapter
        Flags: bus master, fast devsel, latency 0
        Memory at 91c00000 (64-bit, prefetchable) [size=2M]
        Memory at 91e04000 (64-bit, prefetchable) [size=16K]
        Expansion ROM at 91e80000 [disabled] [size=512K]
        Capabilities: [70] MSI-X: Enable+ Count=64 Masked-
        Capabilities: [a0] Express Endpoint, MSI 00
        Capabilities: [150] Alternative Routing-ID Interpretation (ARI)
        Capabilities: [160] Single Root I/O Virtualization (SR-IOV)
                Total VFs: 64, Number of VFs: 0
                VF offset: 128, stride: 2, Device ID: 1515
                Supported Page Size: 00000553, System Page Size: 00000001
                Region 0: Memory at 92300000 (64-bit, non-prefetchable)
                Region 3: Memory at 92400000 (64-bit, non-prefetchable)
        Capabilities: [1d0] Access Control Services
```

VM Performance: TCP Stream

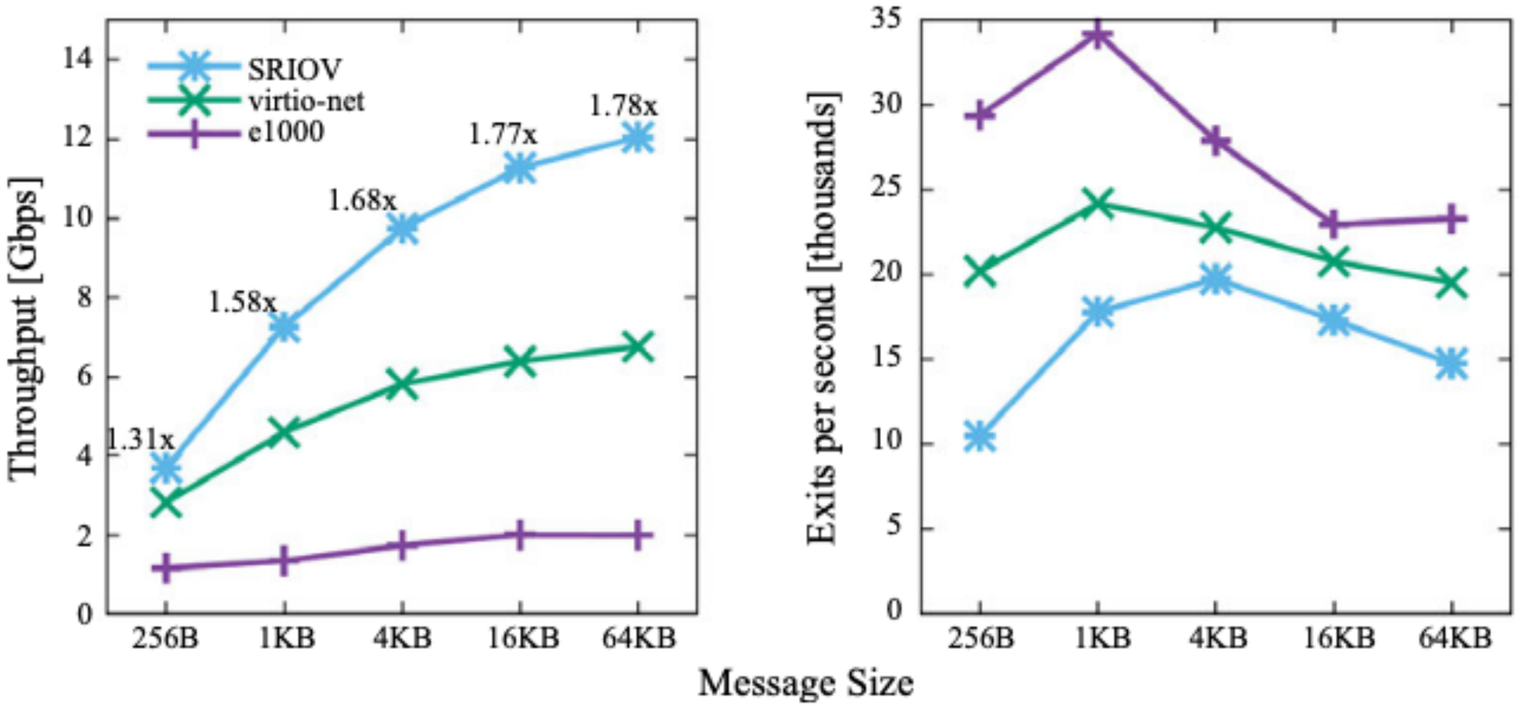


Figure 6.21: Netperf TCP stream running in a single-VCPU Linux 4.4.2 VM served by Linux/KVM (same version) and QEMU 2.5 on a Dell PowerEdge R420 host equipped with a 1.9 GHz Xeon E5-2420 CPU and a 40 Gbps SRIOV-capable Mellanox ConnectX-3 NIC. Networking of the VM is done through e1000, virtio-net (with vhost-net and MACVTAP), and a VF of the ConnectX-3. The destination is a remote R420, which does not employ virtualization. The numbers show the ratio of virtio-net to SRIOV.

Summary: Virtual I/O with hardware support

- IOMMU and SRIOV allow safe and scalable direct device assignment
- Goal: to make I/O virtualization performance comparable to bare-metal:
  - Interrupt handling is still an issue; why?

Comparison of three virtual I/O models

Virtual I/O Model	Emulation	Paravirtualization	Device Assignment
No VM-specific software	Yes	No	Yes
Interposition	Yes	Yes	No
Performance	Worse	Better	Best