x86-64: I/O Virtualization without Hardware Support

Hardware and Software Support For Virtualization

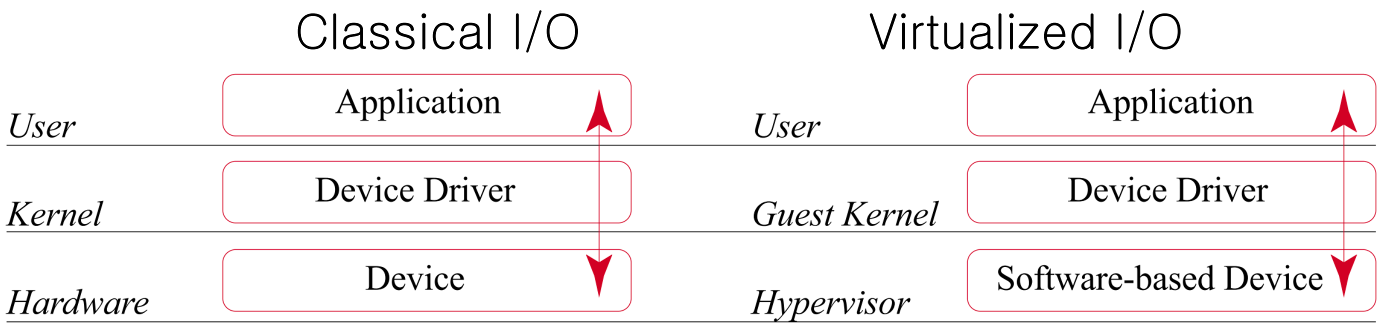
Chapter 6.1-6.3

I/O Activity

- Previous focus in define virtual machines around three key attributes:
  - Equivalence
  - Safety
  - Performance
- Mostly useful to reason in terms of CPU and MMU, but **not for I/O**
- I/O requires a new attribute: **interposition**
  - Interposing on the guest-OS I/O activity enables the hypervisor to **observe**, **control** and **manipulate,** transparently it
  - The activity can be **decoupled** from the underlying I/O devices
- Activity generated and consumed by the VM is referred as **Virtual I/O** (e.g., a disk read)
- **Hardware assistance** for such activity will be critical in production systems
  - Additive to the previously presented (assumes CPU and MMU hw assisted)

Benefits of I/O Interposition

- Host **exposes Virtual I/O devices** to the guests
  - **Traps** when guest try to access them, and **emulates** the intended behavior using real devices
  - Hypervisor encodes (software) virtual devices and interposes Virtual I/O activity
- Allows the hypervisor to **encapsulate** the entire state of the VM, simplifying VM handling
  - E.g., **Suspend** and **resume** the VM transparently
- Decoupling and encapsulation from real devices is key for **live migration**
  - Different I/O devices in different servers



Benefits of I/O Interposition

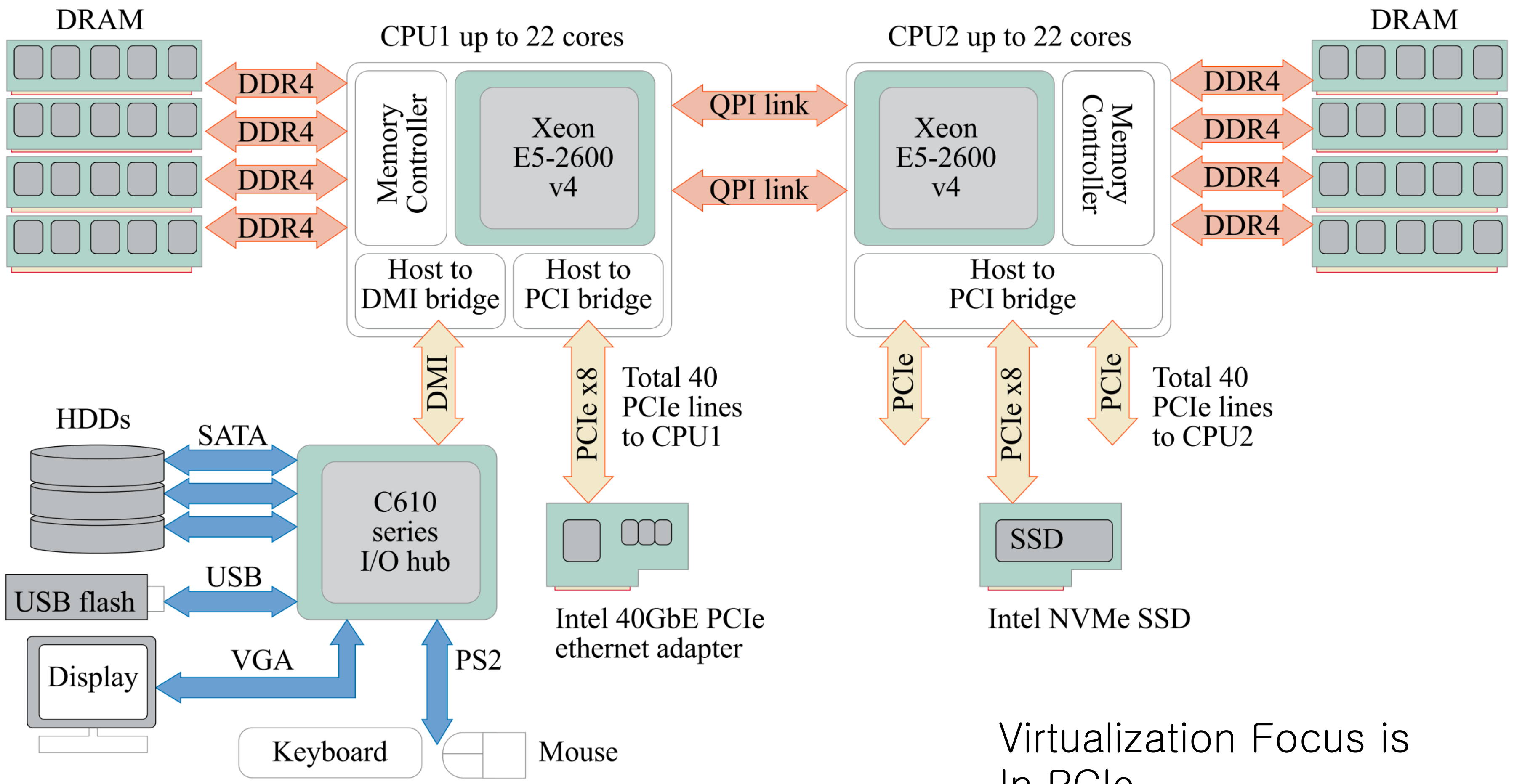
- Interposing can be exploited by the hypervisor to perform **dynamic decoupling** from software I/O devices to real I/O devices and backward
  - E.g., Upgrade a storage device online
- Completes CPU and memory to achieve true **server consolidation**
  - Reduce operation costs of the whole infrastructure
- Allows **aggregation** of real I/O resources
  - Improve performance and/or reliability
- Add new features not actually supported by the device
  - E.g., Replicated write disk to transparently recover from disk failures, compression decompression, accounting, metering, etc…
- I/O interposition **makes possible** apply optimizations to memory images of VM
  - E.g., memory overcommitting (ballooning), page migration, COW, …

Summary

- Add features to the devices
- State encapsulation

Review: Real Organization of I/O (current systems)

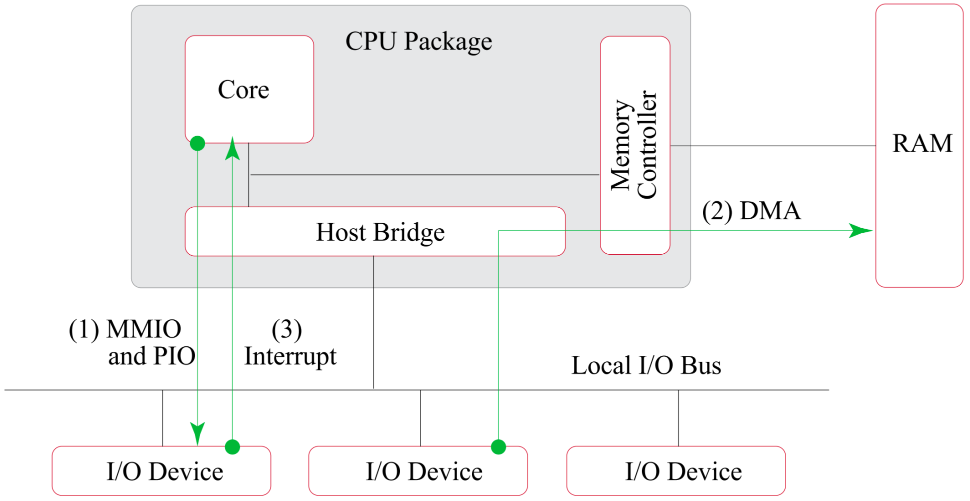
- Two-socket server with Xeon E5-2600 v4



Virtualization Focus is In PCIe

Discovering and Interaction with I/O devices

- On start-up, OS should guess what devices are available
  - **Far from trivial**. Involves firmware (either BIOS or UEFI) code execution
  - Firmware provides to the OS the available devices in a standard way such as **ACPI** (Advanced Configuration and Power Interface) **MCFG Table**
  - Queried by OS
- Devices can be interacted from CPU following two approaches
  - **Port-Mapped IO** (PIO): devices are **separated** from memory physical address space accessed via specific instructions (IN and OUT)
    - E.g., `0x0060-0x0064` are used by PS/2 keyboard devices
    - `0xF01-0xF30` IDE devices
  - **Memory-Mapped IO (MMIO)**: **registers** in the devices are mapped into **physical address space**.
    - Memory controllers and host bridge controller are aware of the addresses (to route them accordingly)

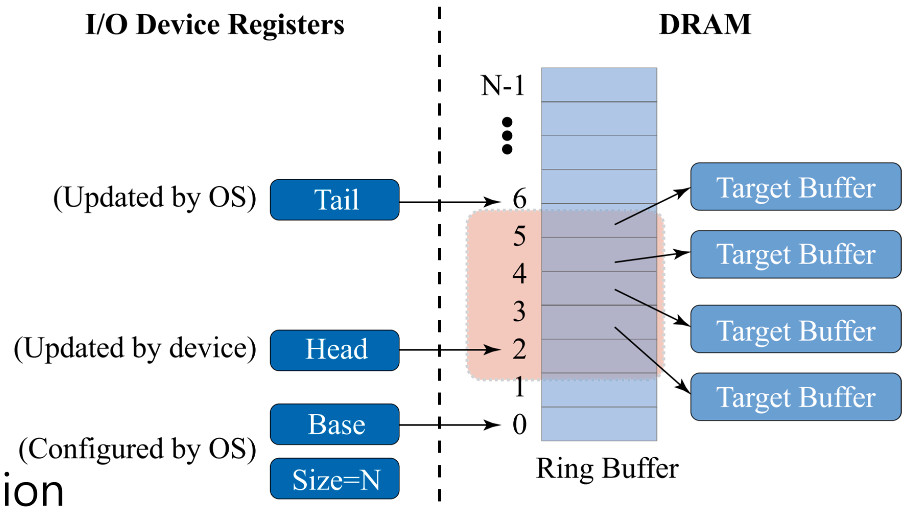


CPU, Memory Interaction

- **Direct Memory Access** (DMA)
  - Rely on specific hardware to move large chunks of data from/to memory
- **Interrupts**
  - Hardware can asynchronously send **events** notification to CPU
  - Each interruption has a number assigned (*interrupt vector)*. Used in the Interrupt Description Table (IDT), tracked by `%idtr` register per core
- Local Advanced Programable Interrupt Controller (**LAPIC**) (per core)
  - Handles OS interrupt related operations; e.g., Enable or disable interruptions, notify the device interrupt attention
  - `IRR` (interrupt request register) in LAPIC 256-bit RO register with **pending** interrupts
  - `ISR` (interrupt service register) in LAPIC RW that marks the interrupts **currently** served
  - `EOI` signals the end of service by the OS (**clears** the highest prio. bit in `ISR`)
  - Recent LAPIC (**2xLAPIC**) versions registers are accessed via **Model-specific Registers** (`MSR`) [Registers defined in the x86 ISA, per model, to monitor or control hw]. **xLAPIC was using MMIO**

Driving Devices Through Ring Buffers

- Devices can exceed 10-100Gb/s. How do it?
- Producer/consumer ring buffers
  - Memory **shared** between the **device driver** and the **physical device**
  - Entries in the ring called **DMA descriptors→** where should done the operation and bit to help driver and device to synchronize
- Devices will use one of such rings to perform each operation
  - E.g., NIC uses at least one Tx and Rx rings (per physical cable).
  - Several rings to promote **scalability** (multiple CPU)
- Rings are initialized by OS driver initialization
  - Head and tail pointers accessed via MMIO
  - Devices and drivers should consider empty/full rings
- **Interrupt coalescending** handles high-throughput scenarios



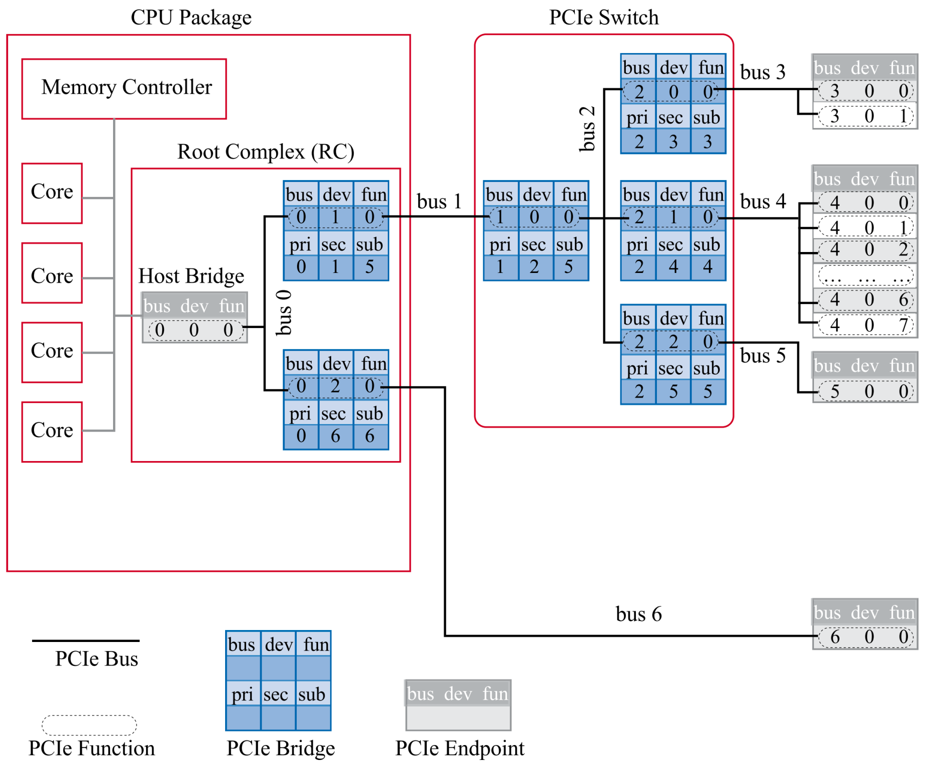
Tx Ring

Example

- OS wants to transmit two packets after a period of inactivity
- Tx head == tail == DMA descriptor k , i.e., is empty
- OS driver set k and k+1 pointing to the packets to send
- Turn bit “production” and update tail to (k+2)%N
- NIC processes sequentially from head *k* and *k+1*
- Asynchronously inform the OS driver via coalesced interrupts (setting ISR register bits) that the operation has been served, clearing the descriptors k and k+1

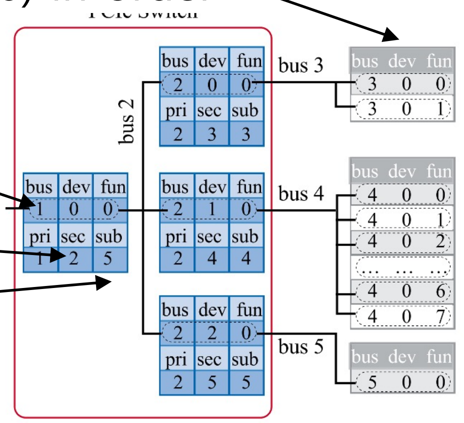
PCIe (Peripheral Component Interconnect Express)

- Similar to a **lossless network** infrastructure (transaction, data, physical layer)
  - Handles packets with defined routing, flow control, error detection, retransmission and QoS
- Arranged as a **Hierarchy**
  - Root of the tree in CPU (usually)
  - End points are functions e.g., Dual-port NIC
  - A bus connect to 1-32 lanes PCIe (v3)
  - **Peak bandwidth** per lane 985MB/s (v3), 1.9GB/s (v4), 3.98GB/s (v5)



PCIe

- **Node Enumeration**
  - Each node (function) and edge (bus) is uniquely identified in the PCI graph as 16-bit N `bus:device:function(BDF)` (8, 5 and 3 bits)
- **Edge Enumeration**
  - Buses are numbered from 0 upward (up to 256) in order
  - Each G bridge is denoted by
    - **Primary bus**: upstream that G connects
    - **Secondary** bus: first downstream bus
    - **Subordinate**: last downstream bus



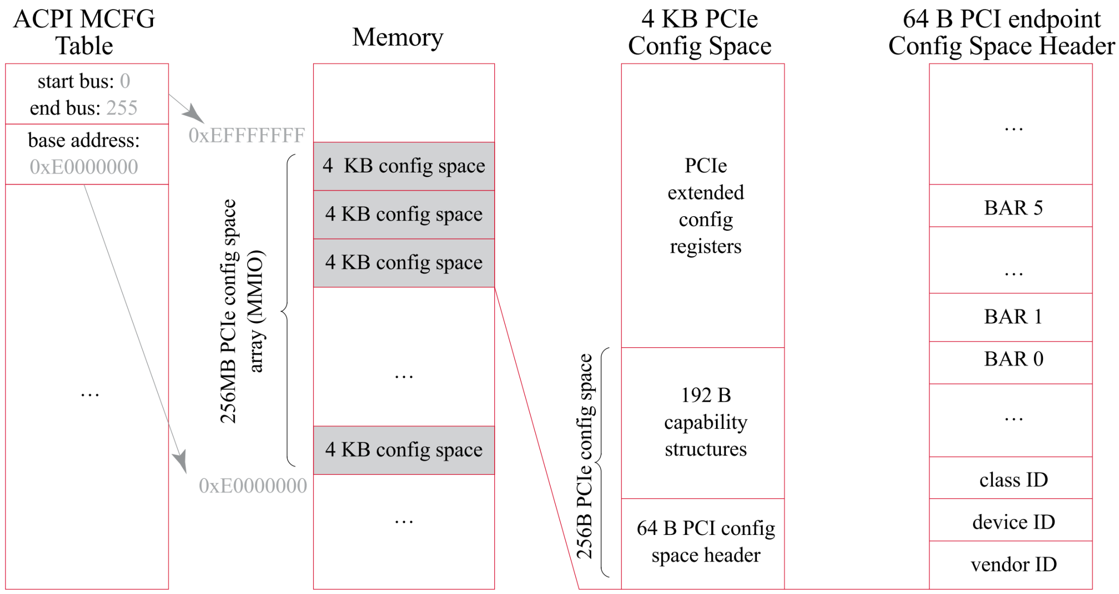
- Edge Enumeration + Node Enumeration **describes the hierarchy** (and allows to route packets to the right consumer)

PCIe Configuration Space

- BIOS/UEFI makes available ACPI tables to OS. OS can access then via MMIO
  - `dmesg | grep MMCONFIG`
- MCFG tables provides the address to each configuration space in the PCIe
  - An entry for each valid BDF (2^64) with 4KB of config = 256MB
- Config Space 3 parts
  - First 256B valid **PCI** configuration space (backwards compatibility)
  - Capability structure describes functional aspects of the device class (e.g. Network, storage,), Vendor ID, component ID, etc…
    - Used for the OS to use the right driver
  - Base Address Registers (BAR)
    - Will specify how to find the head & tail of up to 6 ring buffers
    - Semantics is manufacturer dependent

.

In and Edge includes the primary, secondary and subordinate buses



Message Signaled Interrupts (MSI)

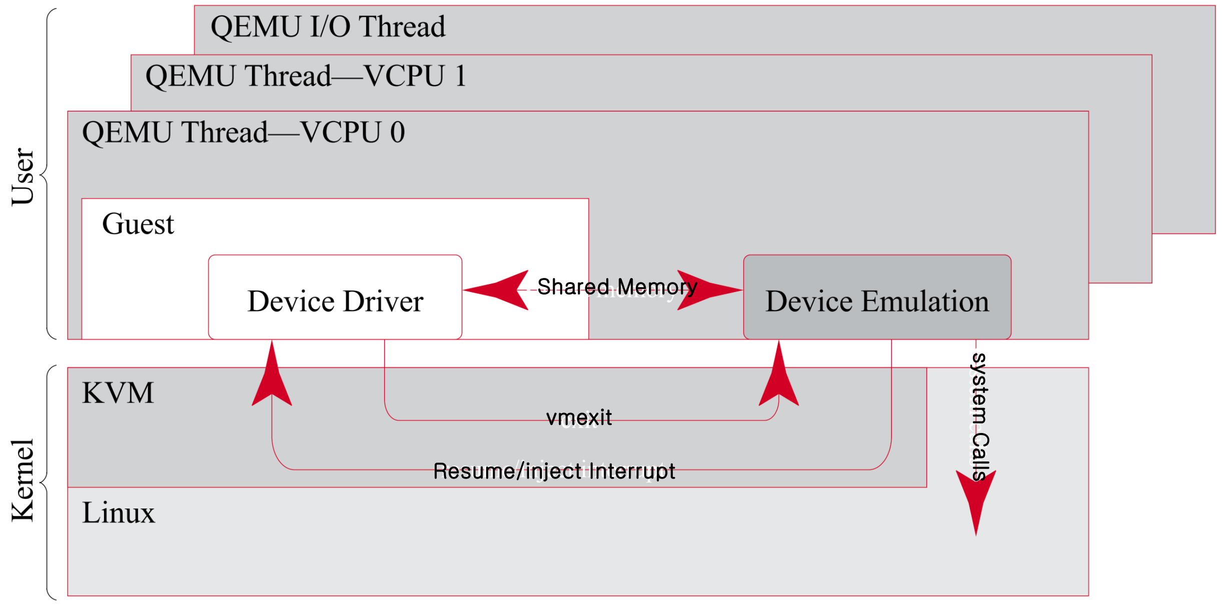
- Third type of I/O-CPU interaction (beyond PIO, MMIO)
- Allows to a device to send a PCIe **interrupt packet** whose destination is a LAPIC in a core
- Similar to a DMA operation but instead of targeting the DMA controller, **targets a particular LAPIC**
- Propagates the PCIe tree until reaches the host bridge
- The OS configures (via PIO or MMIO) the device to use MSI by writing the address of the **target LAPIC** and desired **vector interrupt** to the message-address and message-data registers in middle part of PCIe config space
    - When firing an interrupt send message-address and message-data
- MSI supports 32 interrupt per device and 2048 in MSI-X (PCI 3.0)
    - **IOAPIC/MSI-X** replaces the core-level controller by a **package level** controller

Virtual I/O Without Hardware Support

- Guest-OS still **believes** has control over I/O devices
- Hypervisor **can’t allow** that (by any means):
  - E.g., assume a shared disk between VM and hypervisor (i.e., guest can access directly to it) → VM crash → data loss (most likely)
- Hypervisor should **prevent** direct access to I/O, retaining the **illusion** of the guest-OS
  - Software defined “virtual” I/O devices (fake devices)
  - Trap guest-OS “I/O intentions” and **emulated** then in the **fake devices**

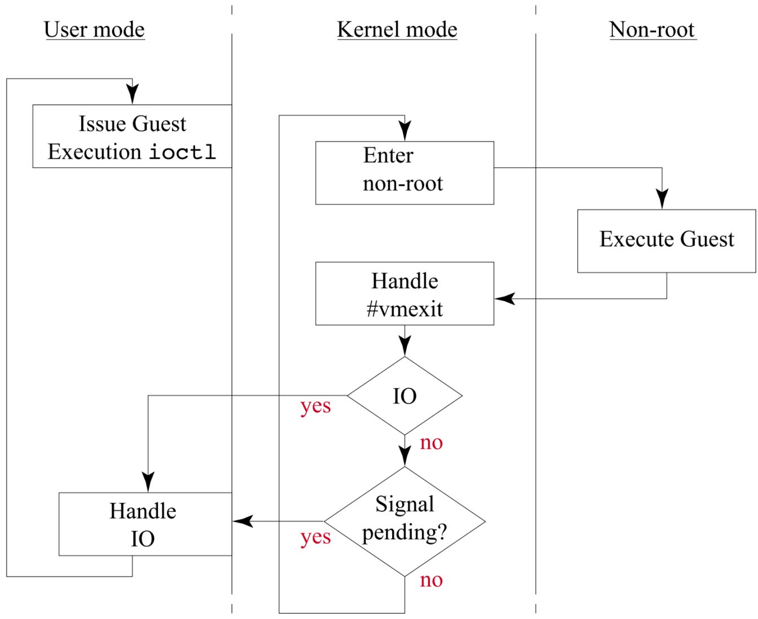
I/O Emulation (Full Virtualization)

- **Guest-OS**: (1) "talks" to I/O via MMIO or PIO and (2) I/O responds back triggering interruptions and reading/writing Memory via DMA
- **Hypervisor**: (1) can intercept guest-OS I/O operations and (2) inject "fake" responses of the emulated I/O devices into the guest-OS
- DMA can be trivially emulated by Hypervisor (can read and write in guest-OS mem)
- PIO Traffic will trap (`IN` and `OUT` will do it, configuring VMCS to do so)
- MMIO operations will trap, mapping them into restricted zones (e.g., `pte.us:1`)



I/O Emulation In KVM

- Every hosted VM is **encapsulated in a QEMU process**
  - Internally QEMU process "handles" each **VCPU** as a **separate thread**
    - Two execution contexts: QEMU host process and VCPU
  - For each **individual I/O device** of the VM, QEMU **uses a thread** (I/O thread)
    - Handles the async activity associated to the device (e.g., net)
- VCPU context issues MMIO/PIO request to the device
  - Triggers trap to KVM
  - KVM relays back the operation to the QEMU host of the thread
  - QEMU hosts drives this to the emulated device driver. Using system calls are redirected to the real devices
  - Returns to KVM via `ioctl` over `/dev/kvm`
- When device comes back, hypervisor deliver the response to the corresponding guest-OS
  - Copies from/back from "real" DMA regions to guest-OS expected addresses
  - Emulates the interruption in the corresponding VCPU



Virtual I/O devices exposed to the VM

- Hypervisor will decide to do it at boot time (according config)
  - Via emulated UEFI/BIOS (i.e., not real ACPI tables but the generated by the hypervisor)
- `lspci` output (default QEMU)

```
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
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
```

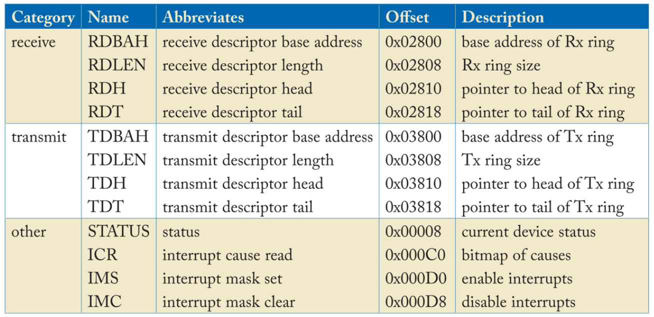
- Details for ethernet controller

```
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
```

MMIO BARs

Intel e1000 (82540EM Gigabit Ethernet Controller)

- 82450EM is a PCI device (2002) supported by
  - Host bridge (440FX) [root of the hierarchy] is also PCI not PCIe (QEMU exposes a PCI tree)
  - No influence on performance (it's a **software construct**)
- e1000 driver is used by a large family of devices and supported by most guest-OS
- `0xfebc0000` is the physical address of the MMIO BAR



Category	Name	Abbreviates	Offset	Description
receive	RDBAH	receive descriptor base address	0x02800	base address of Rx ring
	RDLEN	receive descriptor length	0x02808	Rx ring size
	RDH	receive descriptor head	0x02810	pointer to head of Rx ring
	RDT	receive descriptor tail	0x02818	pointer to tail of Rx ring
transmit	TDBAH	transmit descriptor base address	0x03800	base address of Tx ring
	TDLEN	transmit descriptor length	0x03808	Tx ring size
	TDH	transmit descriptor head	0x03810	pointer to head of Tx ring
	TDT	transmit descriptor tail	0x03818	pointer to tail of Tx ring
other	STATUS	status	0x00008	current device status
	ICR	interrupt cause read	0x000C0	bitmap of causes
	IMS	interrupt mask set	0x000D0	enable interrupts
	IMC	interrupt mask clear	0x000D8	disable interrupts

- QEMU emulates the NIC (`hw/net/e1000.c` in QEMU code base)
  - E.g., when a **guest reads** the ICR, a `vmexit` is raised→KVM→QEMU (**ICR has to be cleared** in each read, according specification)
  - QEMU analyzes the instruction that triggered the exit, and handles the case:

```
static uint32_t mac_icr_read(E1000State *s)
{
    uint32_t ret = s->mac_reg[ICR];
    s->mac_reg[ICR] = 0;
    return ret;
}
```

I/O Paravirtualization

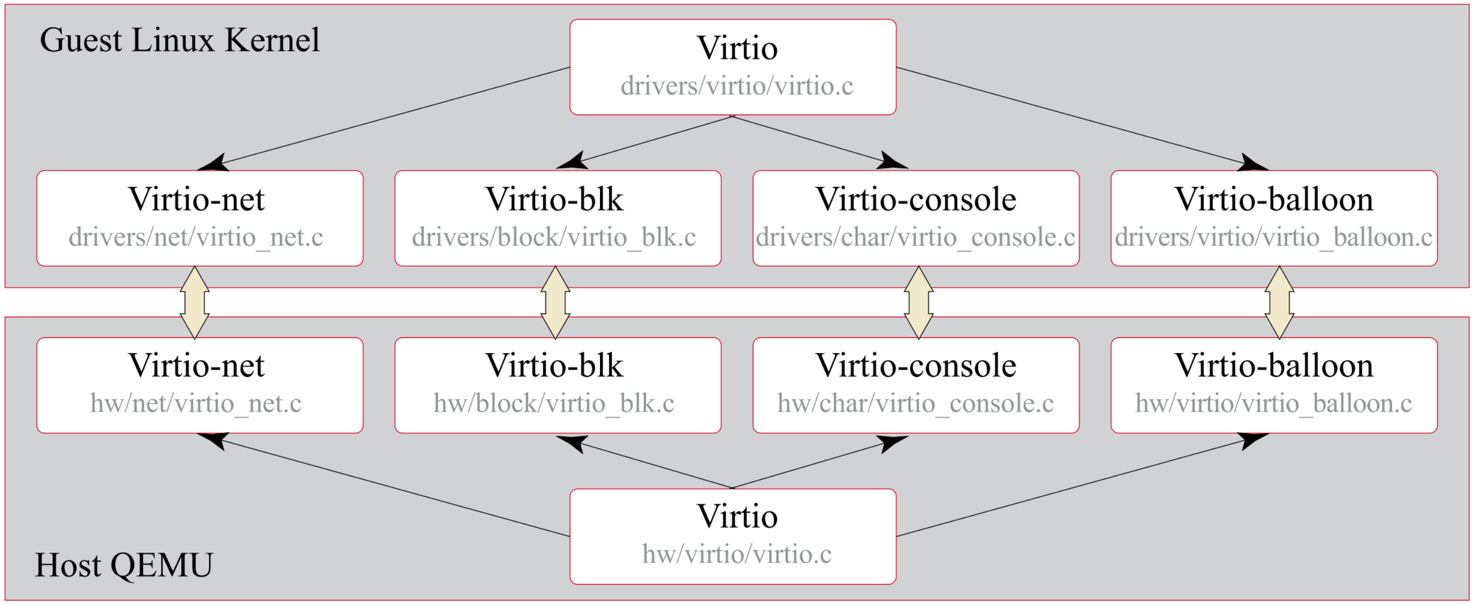
- I/O emulation its correct, but induces substantial **overheads**
  - Hardware designers may not have considered the possibility of emulation.
  - E.g., e1000 send/rcv a single ethernet frame requires multiple status register accesses (i.e., frequent `vmexit`)
- The **lack of specialization** misses the opportunity for optimization
  - e1000 operation requires frequent reads to the STATUS register (e.g., twice every send)
  - Direct access from the guest OS might have reduced emulation overhead, but unfortunately, ICR and STATUS are on the same page.
- Overhead might be eliminated if devices are designed with **virtualization friendly interfaces**
  - **Its impractical** from the standpoint of the **physical** devices
  - Might be **feasible** from the standpoint of the **virtual** devices → **I/O paravirtualization** goal is define such virtual devices
- Paravirtualization I/O improves (significantly) performance but requires specific drivers in the guest-OS (might **affect portability, less stable in critical systems, hypervisor developers should maintain them**)

Virtio

- **Framework** for paravirtualized I/O devices KVM/QEMU
  - Allows to "pairs" emulated devices with guest-OS devices
  - Exposed to the guest-OS via ACPI table (example BDF 00:04.

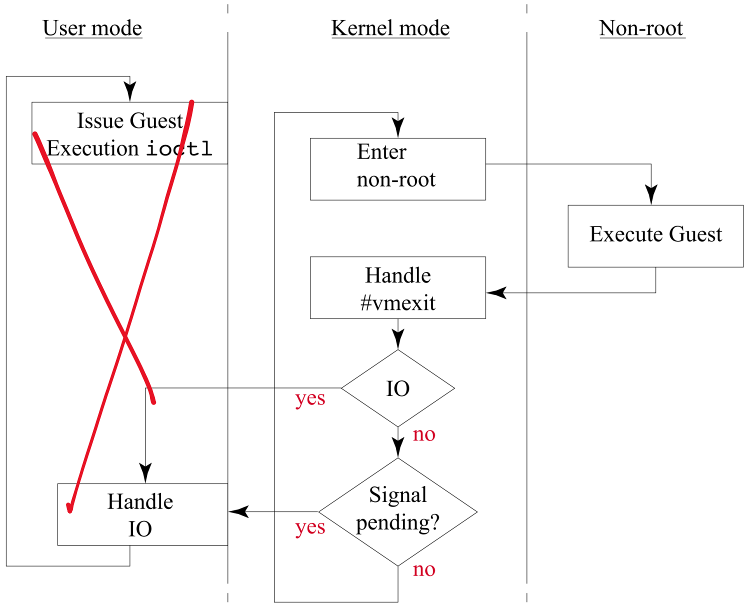
```
00:04.0 Ethernet controller: Red
```

- Fundamental construct is `virtqueue` (ring buffer) post by the guest consumed by the host (per device)
  - Guess-OS access to `virtqueue` **does not raises a `vmexit`** (unless driver needs to do it with a `virtio_kick`). (In emulated devices each PIO/MMIO access raise a `vmexit`)
  - The `virtqueue` can operate in two modes (**minimizes interrupts** and **overheads** respectively)
    - In **NO_INTERRUPT** the host side can't deliver interrupts to the guess-OS (until disables the mode) e.g., Used for TX in Virtio-net. Guest is not interested in knowing when the transmission finishes
    - Symmetrically **NO_NOTIFY** host side tells the guess-OS to not `virtio_kick`. E.g. When guest-OS needs to send a burst of frames, just `virtio_kick` the first (the remaining frames are processed automatically) (TCP traffic is bursty)



Network Paravirtualization Exception

- The most demanding device
  - Staggering throughput of modern NIC (40/50/100GbE)
  - Million of packets per second to handle
- KVM/QEMU makes an exception with `vhost-net`
  - Instead of forwarding the traffic to user space (QEMU thread) the packets are directly handled by the kernel of the host
  - No outer loop: **just inner loop**
  - `vhost-net` runs in **kernel space**



Performance: Emulation vs Paravirtualized Drivers

- `Netperf` case (~~`vhost-net`~~): MTU 1500 with **TCP segmentation offload** (TSO) 64KB
  - More exits and Interrupts
    - Network stack handles trust in the nic to split in MTU sized frames
    - In emulation much more frequent exist per segment.
    - NO_NOTIFY and NOT_INTERRUPT reduces virtio
  - Average segment size 3x
    - Is determined by the TCP/IP stack of the guest-OS dynamically. The "slow" behavior of e1000 discourages larger sizes
  - E1000 code is focused on **correctness** whereas `virt-io` has been **heavily optimized** across years
    - E.g., TSO is really unoptimized in e1000 emulation, and its software!!!

	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
TCP segments	per exit	1/9	25	225x
	per interrupt	1	118	118x
	per second	3,669	30,252	8x
	avg. size (bytes)	8,168	21,611	3x
	avg. processing time (cycles)	652,443	79,132	1/8x
Ethernet frames	per second	23,804	–	–
	avg. size (bytes)	1,259	–	–

Front-Ends and Back-Ends (modular view)

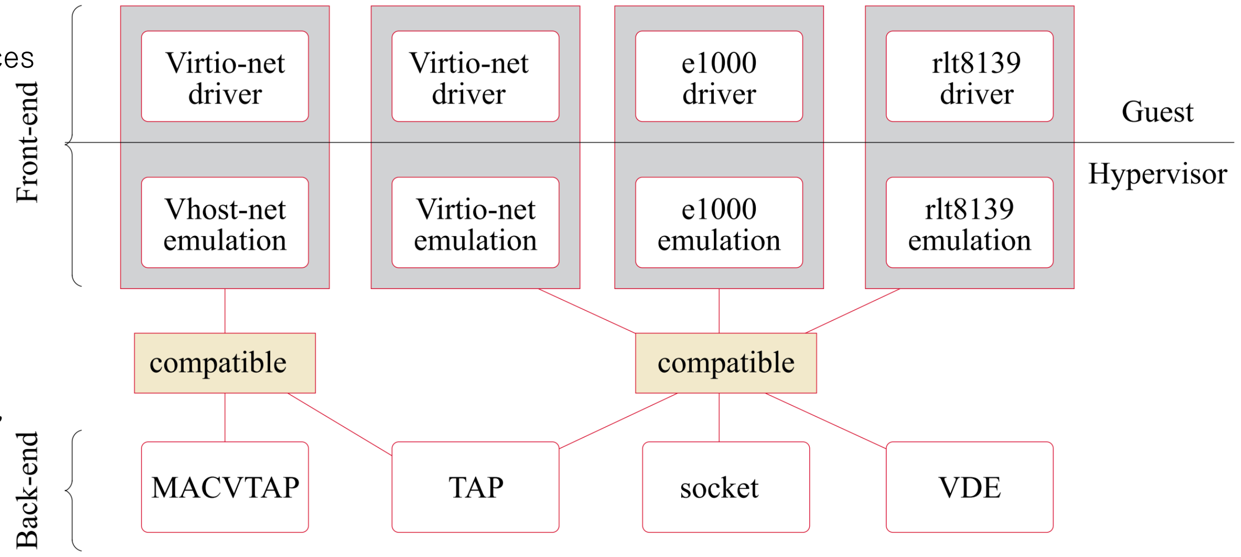
- Front-End
  - Encompasses a guest virtual device driver and a matching hypervisor emulation layer
- Back-End
  - Used by the front-end to implement the emulation of the virtual device

TAP: virtual Ethernet network device
(can forward Ethernet frames between proces
ses)

VDE: Virtual Distributed Ethernet

MACVTAP: TAP with less routing flexibility
But less abstractions layers in kernel
(no bridge: vhost-net goes into the NIC)

DISK: Local SATA, local SAS, remote iSCSI, CEPH, etc…


Front-end
Virtio-net driver
Virtio-net driver
e1000 driver
rlt8139 driver
Guest
Hypervisor
Vhost-net emulation
Virtio-net emulation
e1000 emulation
rlt8139 emulation
compatible
compatible
Back-end
MACVTAP
TAP
socket
VDE