CSIE 5310

Lecture 6

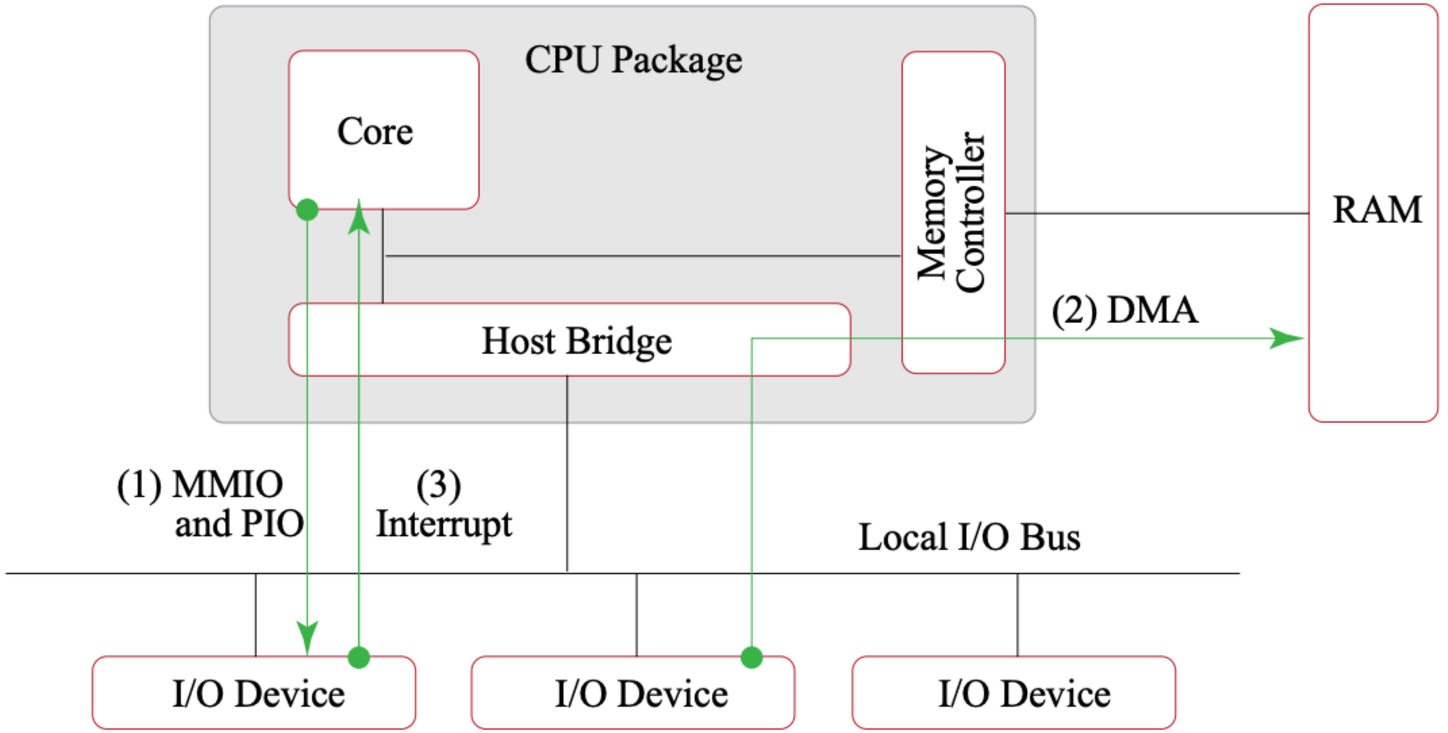
Prof. Shih-Wei Li

Department of Computer Science and Information Engineering
National Taiwan University

Agenda

- **Interrupt Virtualization**
- KVM walkthrough (part 2)

Review: Interactions between I/O devices, CPU, and memory



Interrupt Controller (1)

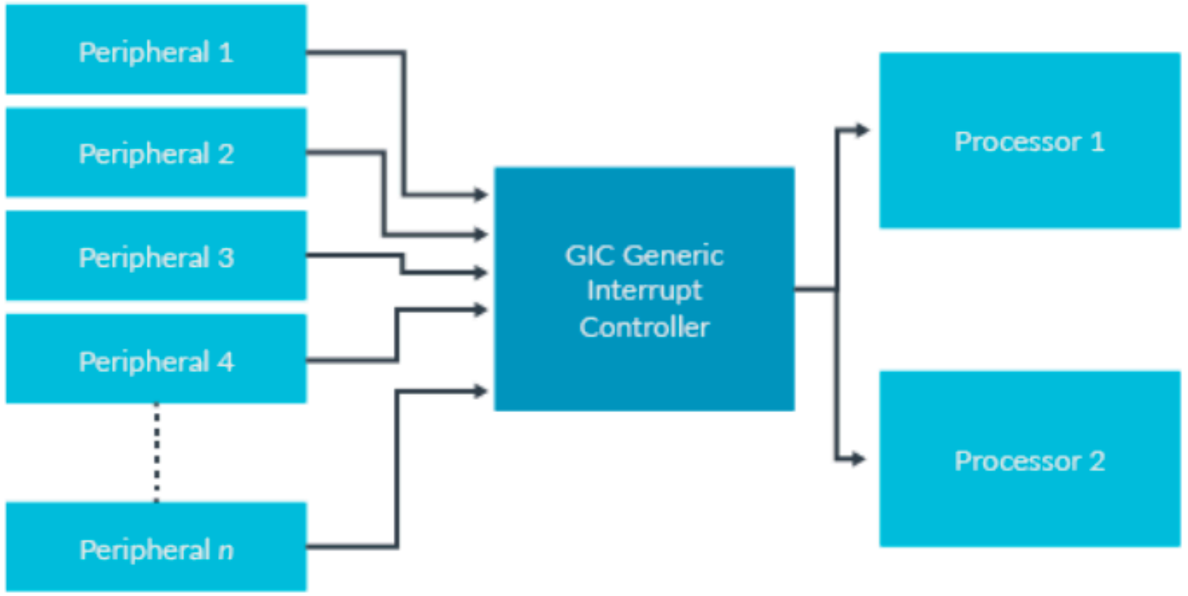
- Handle interrupt related operations performed by the OS kernel or hypervisor:
  - Ex: enabling/disabling interrupts; notify hardware for interrupt handling completion; sending IPIs, etc.
- x86 provides Local Advanced Programmable Interrupt Controller (***LAPIC***)
  - Per CPU core; includes a set of registers for different features:
    - *Interrupt request register (IRR)*: contains the fired interrupts that have not been handled by the core
    - *In-service register (ISR)*: contains the interrupts that are currently being handled
    - *Interrupt command register (ICR)*: allows a core to send interprocessor interrupts (IPI) to other cores
    - *End of Interrupt register (EOI)*: used by the OS or hypervisor to signal the completion of the handling of the interrupt; LAPIC then clears the ISR for the interrupt

Interrupt Controller (2)

- CPU accesses to LAPIC:
  - The previous LPAIC interfaces (ex: xAPIC) be accessed by load/store instructions to the respective MMIO area
  - The newest LAPIC interface (x2APIC) can be accessed via read/write operations of model specific registers (MSRs)

Interrupt Controller (3)

- Arm defines the Generic Interrupt Controller (GIC) architecture



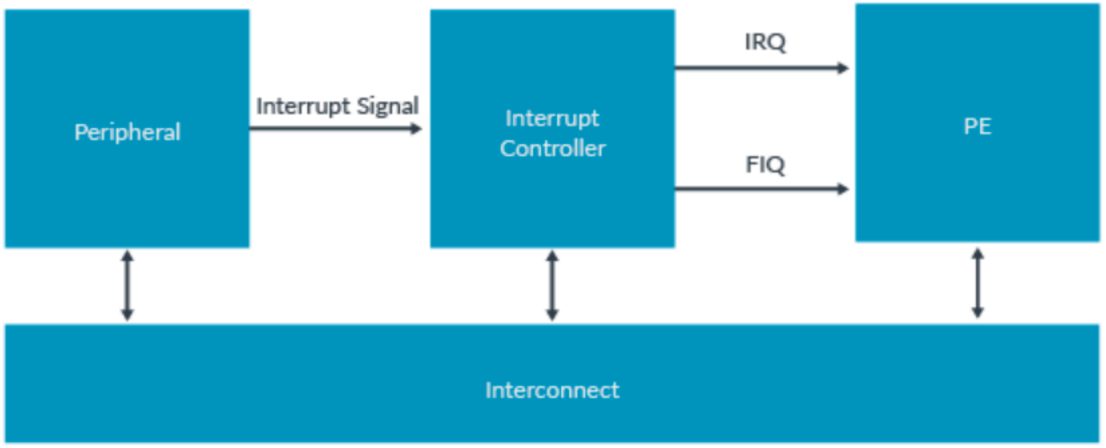
https://documentation-service.arm.com/static/636e3b314e6cf12278ad8d30?token=

Interrupt Controller (4)

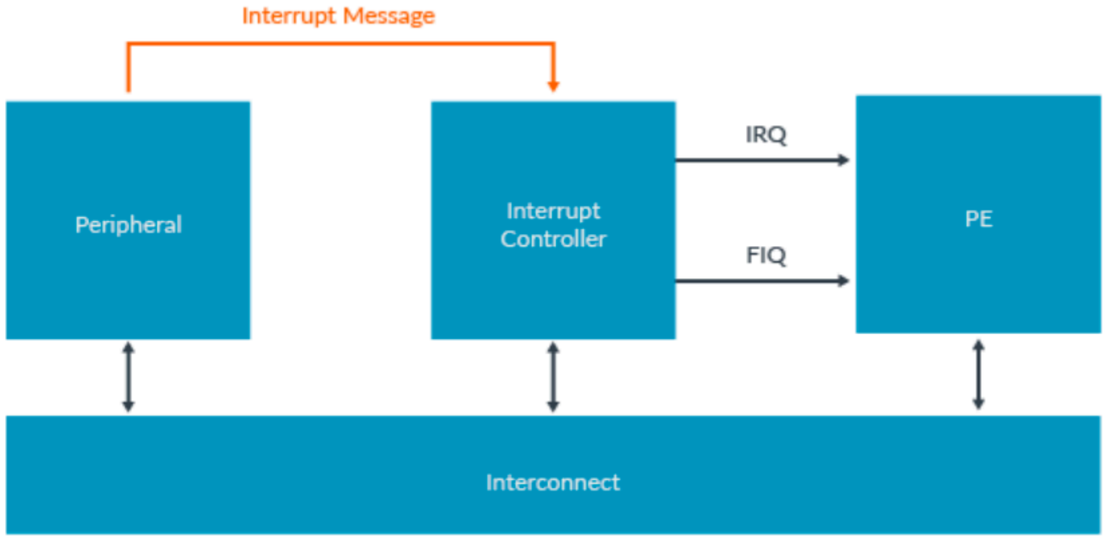
- Arm GIC has various versions (v1, v2, v3, and v4)
  - GICv3 and GICv4 are supported by more recent Arm CPUs (Cortex-A3x, CortexA5x, CortexA7x cores)
- Each interrupt source is identified by an ID number, referred to as an *INTID*

Interrupt Controller (5)

- Interrupts on Arm are signaled from a peripheral to the GIC using either a *dedicated hardware signal*, or a message (*message-signaled interrupts, MSI*)



Dedicated interrupt signal:
https://developer.arm.com/e9eee214-50a8-4b2a-a9d7-20b4e773630f



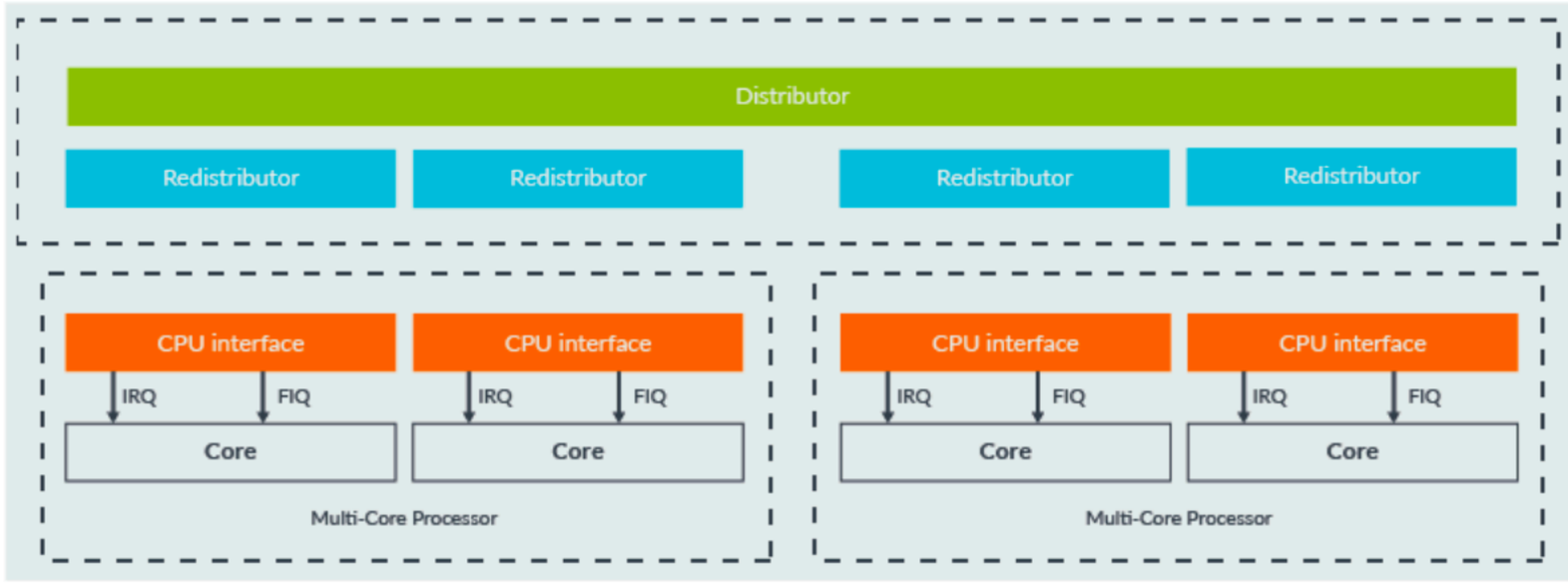
Message signaled interrupt signal:
blob:https://developer.arm.com/b8a2e485-e8fd-44e7-8818-444392190c97

Interrupt Controller (6)

- Message signaled v.s. Dedicated interrupt signal:
  - Message-signaled interrupts remove the requirement for a dedicated signal for each interrupt source; can support more interrupt signals in large systems

Interrupt Controller (7)

- Case Study: GICv3 consists of three groups of registers
  - Distributor interface
  - Redistributor interface
  - CPU interface



Interrupt Controller (8)

- Distributor and Redistributors provide the programming interface to configure interrupts
  - Configurations: enable/disable, set priority, affinity, etc.
- CPU interface is used to handle interrupts:
  - Supports acknowledge an interrupt or signal an EOI

Message Signal Interrupts (MSI)

- Devices can send MSIs by performing a DMA write to a dedicated memory address range:
  - 0xFEE00000 - 0xFEEFFFFFF on x86

Interrupt Overhead

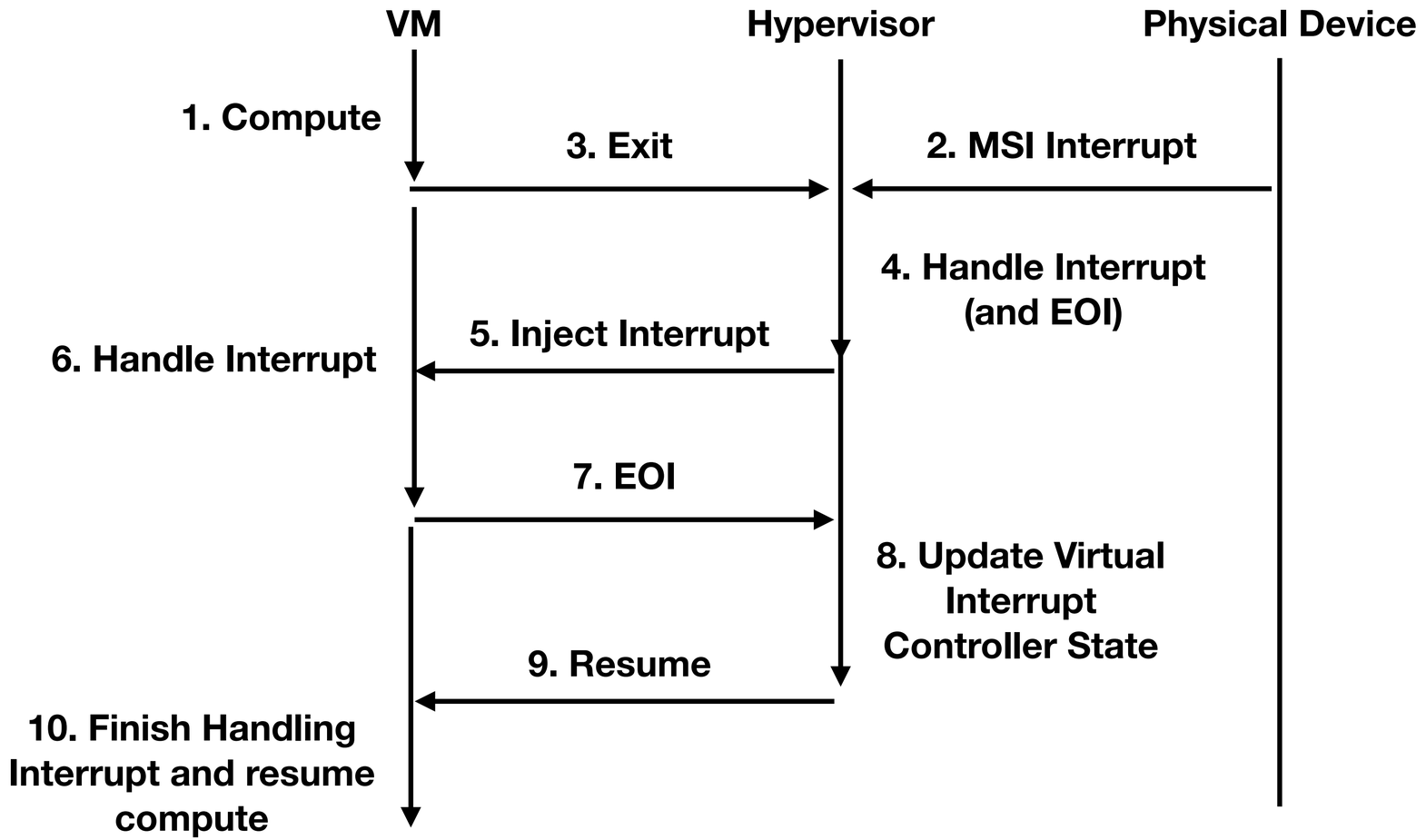
- SRIOV and IOMMU eliminates most of the VM exits that occur when guests talk to their assigned device (ex: via MMIO)

- Overhead remains in delivering virtual interrupts to VMs

Basic VT-x Interrupt Support

- VMCS stores the value of the guest's Interrupt Descriptor Table Register (IDTR):
  - Loaded to hardware upon VM enter, saved when VM exit
- The hypervisor sets the "external-interrupt exiting" bit in VMCS
  - Configure the CPU to exit from the VM when an external interrupt fires
- The hypervisor injects virtual interrupts to the VM
  - The hypervisor updates the "**interruption-information**" field in VMCS
  - If the VM disables interrupt, the hypervisor updates the "**interrupt-window exiting**" bit to deliver interrupt later when possible

Basic VT-x Interrupt Support



Assigned EOI Register

- The OS kernel programs the EOI register when completing interrupt handling
- VM traps when programming the EOI register in the emulated virtual LAPIC
  - LAPIC can be accessed by MMIO instructions (memory load/store)
  - It causes an additional trap; two traps are needed to handle one interrupt!
- The current LAPIC interface, x2APIC, allows the hypervisors to control the (the msr bitmap in) VMCS to avoid trapping the VM's access to the EOI register

Assigned Interrupts

- Exitless Interrupts (ELI)
  - Software-based approach to safely deliver interrupts for the SRIOV device to the VM without trapping to the VMM

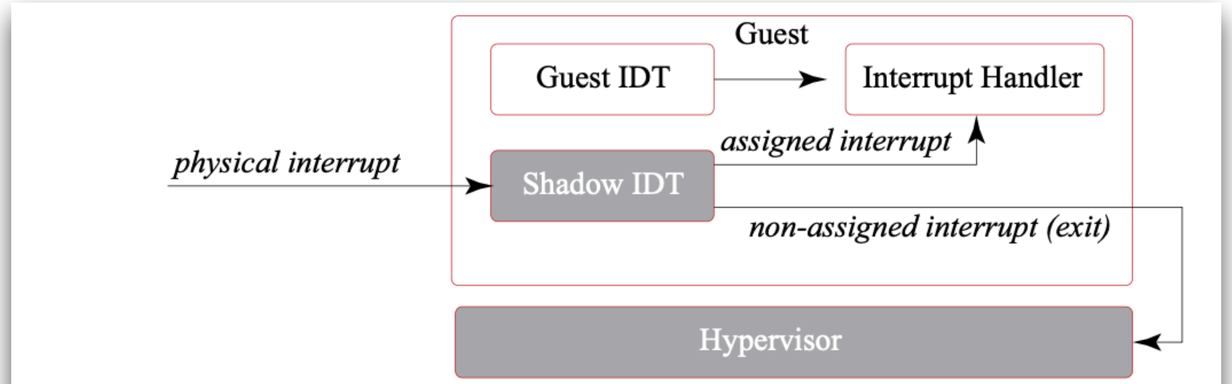
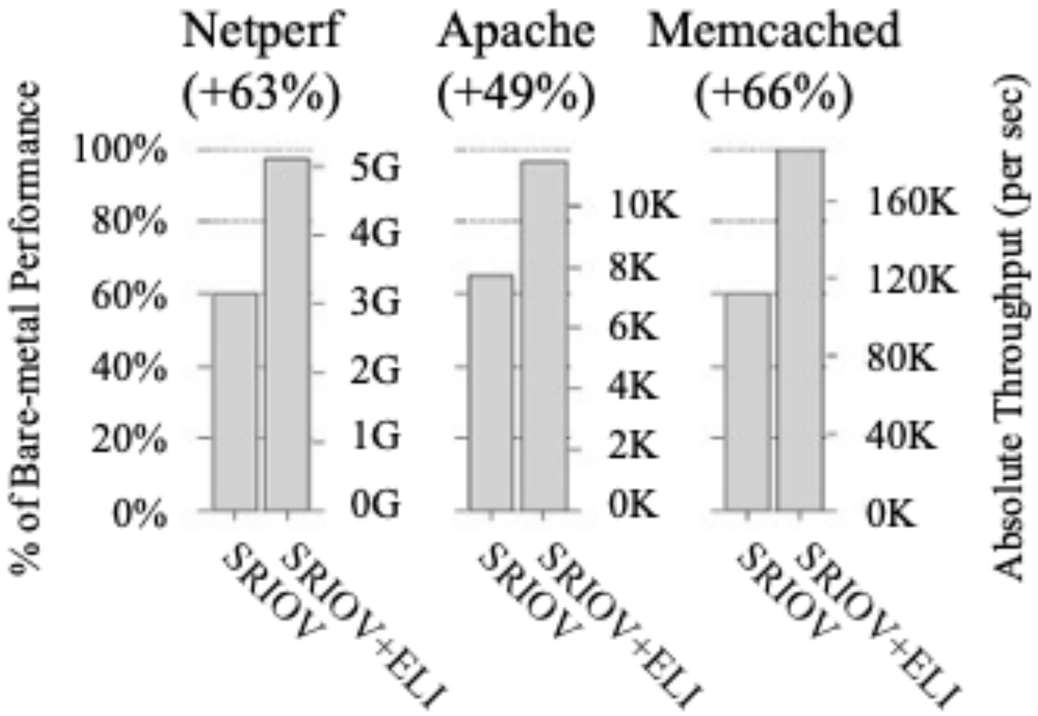


Figure 6.23: ELI interrupt delivery flow.

Assigned Interrupts



Posted Interrupts

- ELI was mostly an academic exercise
- Hardware provides support that allows the hypervisors to assign specific hardware interrupts of specific devices to guest VMs; allows ***interrupts to be delivered to the VM without hypervisor intervention***
- Intel VT-d provides posted interrupts + a virtual APIC for the VM to handle interrupts (e.g., EOI) without trapping
  - The hypervisor configures the VMCS of the guest to point to a 4KB memory area, denoted as the “virtual APIC page” (that houses virtual APIC registers), which the processor uses to virtualize access to APIC registers and track their state, and manage virtual interrupts

Posted Interrupts

- Posted interrupts include two components:
  - CPU posted interrupts
    - Corresponds to IPIs
  - IOMMU posted interrupts
    - Interrupts delivered from I/O devices to the VM

Agenda

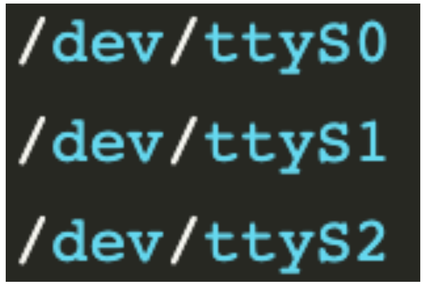
- Interrupt Virtualization
- **KVM walkthrough (part 2)**

KVM: the Linux Virtual Machine Monitor

- Stands for Kernel-based Virtual Machine; came out in 2006/2007
- Type-2 VMM; integrated with a host Linux kernel
  - Reuse existing kernel functionality; i.e., schedulers, NUMA support, timers
  - Reuse existing Linux process management infrastructure; admin can *kill* VMs
- Adds VMM capability to Linux; leverages x86's hardware virtualization support
- Integrates with a user space program called QEMU

Linux character device

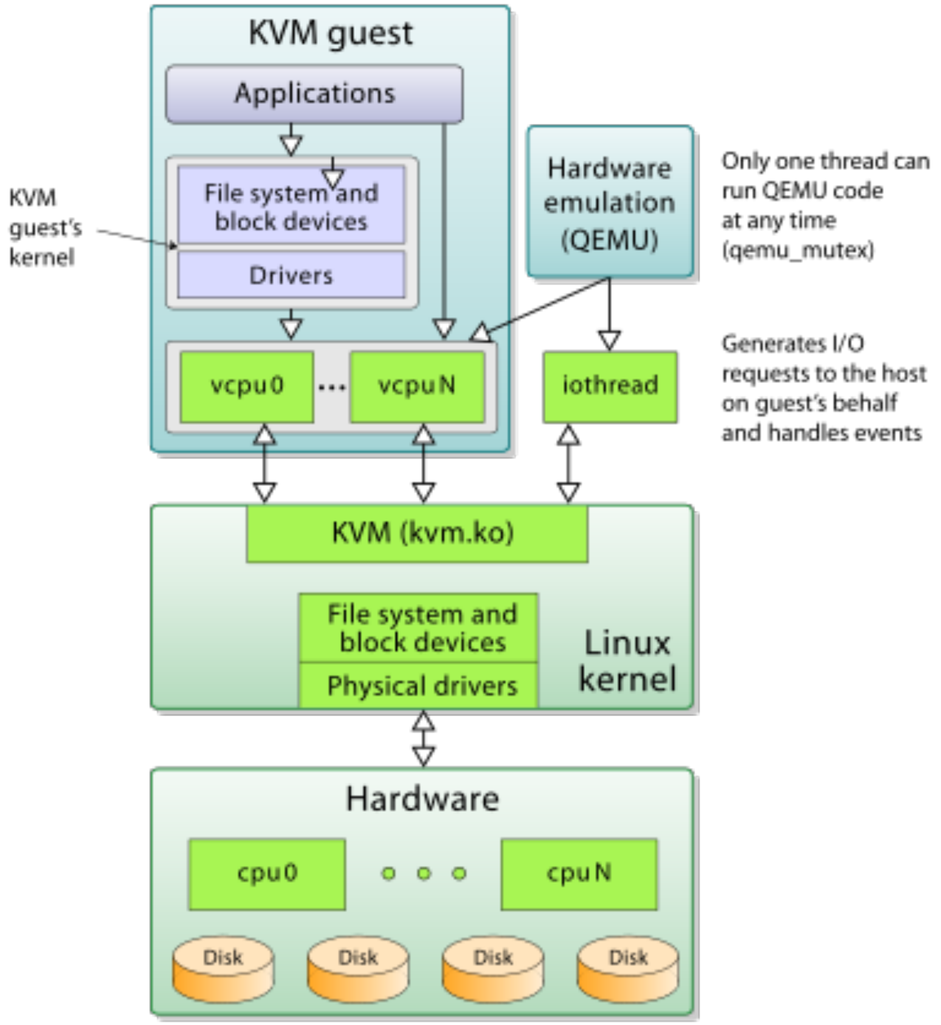
- One of the simplest way for user space to communicate with a Linux kernel module
  - Users can access the device file (/dev/xxx)
  - Support regular file operations (open, read, write, ioctl, etc.)



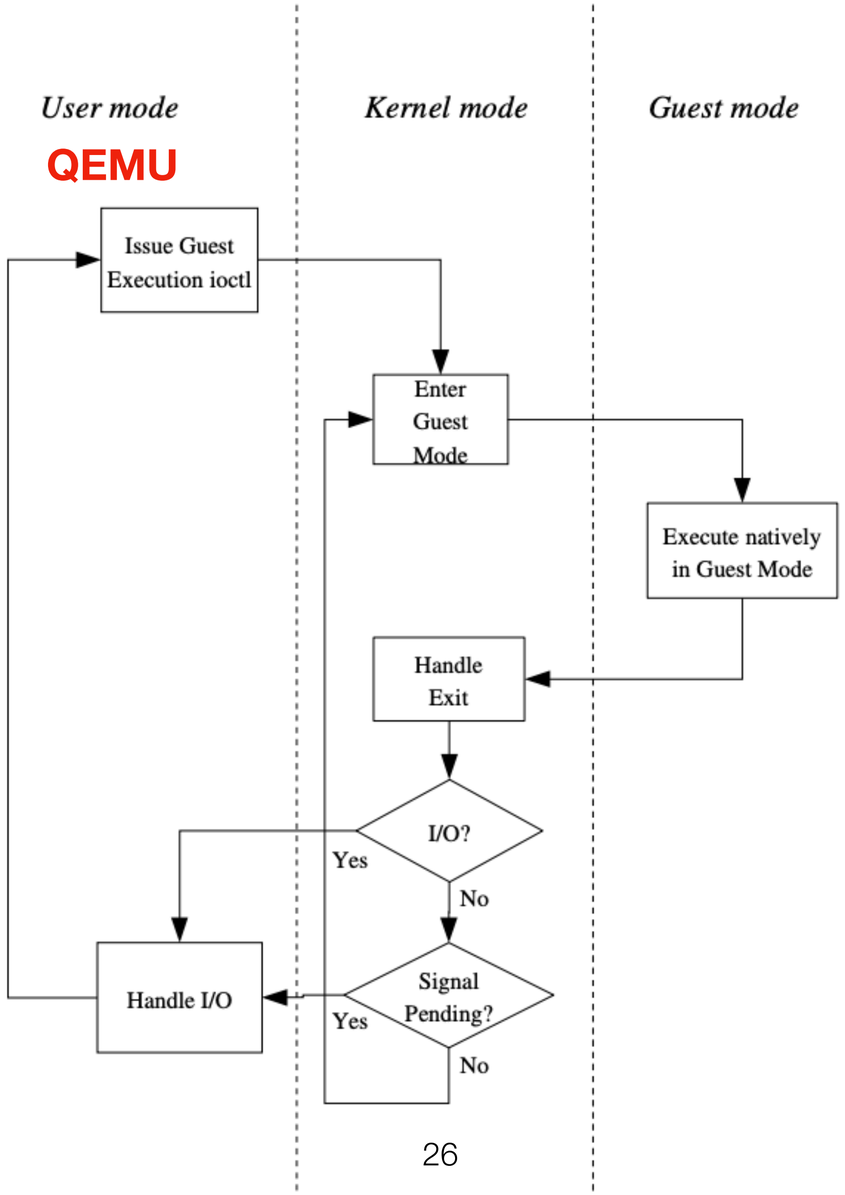
KVM device node

- KVM exposes a /dev/kvm device node to userspace
- Userspace can use the device node to create and run virtual machines
- Operations provided by /dev/kvm:
  - Creation of a new VM
  - Allocation of memory to a VM
  - Reading and writing from/to virtual CPU registers
  - Running a VCPU

KVM Architecture



VM execution loop



KVM VM memory map

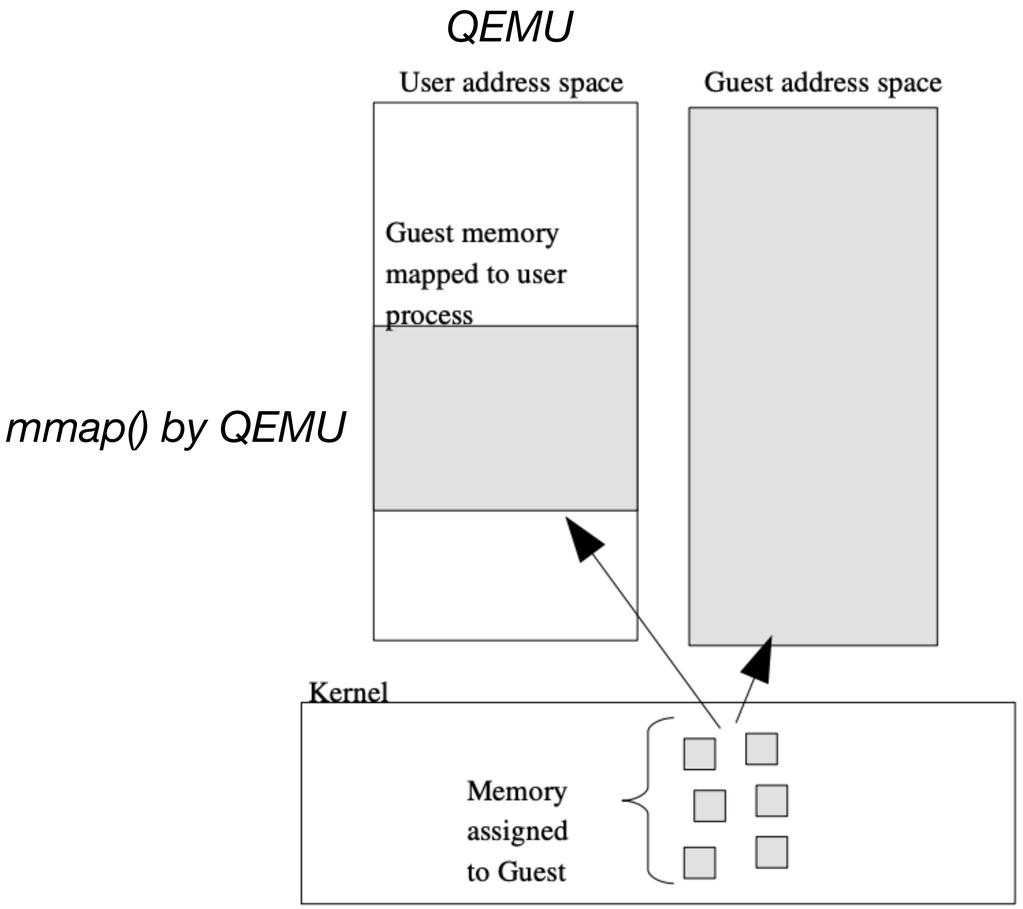


Figure 1: `kvm` Memory Map

Virtualizing the MMU

- KVM initially implements shadow page tables
  - Maps gVA to hPA
- Has to track VMs' page table updates
  - Complicates the implementation and incurs performance overhead
- Leverage nested paging hardware in later KVM version

Virtualizing I/O

- Trap and emulate MMIO and PIO instructions
  - QEMU provides the virtual I/O model
- Supports paravirtualized I/O and direct device assignment
- KVM virtualizes the interrupt controller
  - Virtual devices from QEMU can inject virtual interrupts to VM via system calls
  - Later KVM versions support in-kernel interrupt controller

KVM Code Structure

- Includes both architecturally dependent and independent code:
  - Dependent: under arch/xxx/kvm/…
  - Independent: under virt/kvm/
- Includes paravirtualized device drivers (virtio):
  - The source tree contains code for the guest and hypervisor drivers

KVM Summary

- KVM is used by Google nowadays to host their cloud platforms/services
- Support a wide range of CPU architectures and hardware platforms
  - x86, Arm, PowerPC, Risc-V, etc.
- You can find KVM from the Linux kernel source