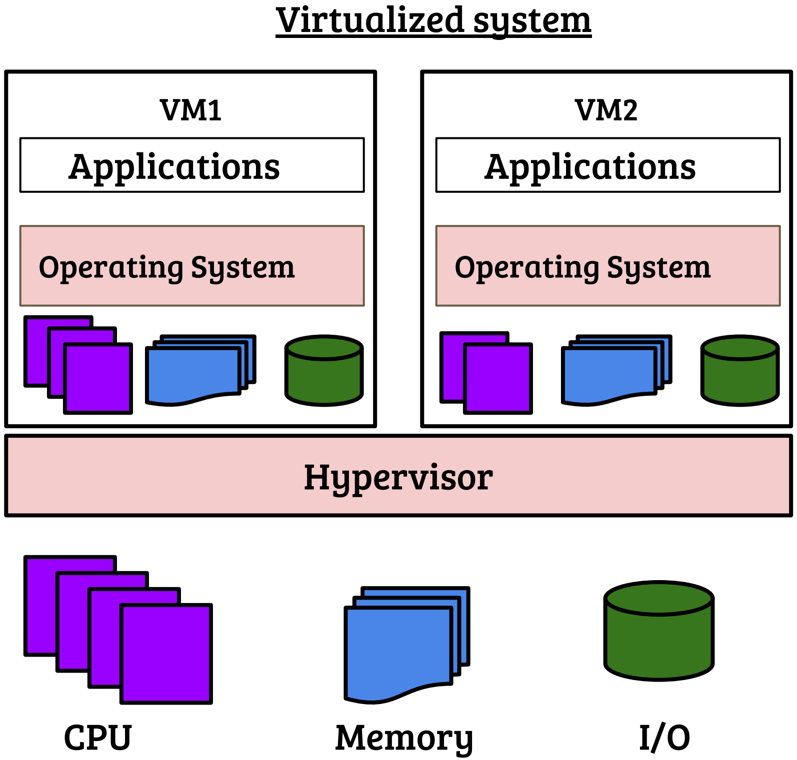
Topics in Operating Systems

Advanced isolation: Virtualization (I/O)

Debadatta Mishra, CSE, IITK

Virtualization: Resource multiplexing with isolation



- Definition [1] “Not physically existing as such but made by software to appear to do so.”
- Objectives
  - Equivalence
  - Isolation
  - Resource control
  - Efficiency

1. Oxford dictionary : https://en.oxforddictionaries.com/definition/virtual

I/O virtualization is different

Characteristics

- Speed mismatch between I/O and CPU
- CPU may not accesses the I/O device like memory (inefficient)
- I/O events depends on external factors
- Considered to be at the periphery of the core OS

!= CPU | Memory

- No hardware state save and restore support
- No in-device partitioning support like memory (traditional I/O devices)
- Involvement of the system software (OS) is more prominent

I/O virtualization requirements

Equivalence

- Strict: Device driver for physical device should work for virtual device
- Relaxed: Generic device layer (HAL) should work in a seamless manner

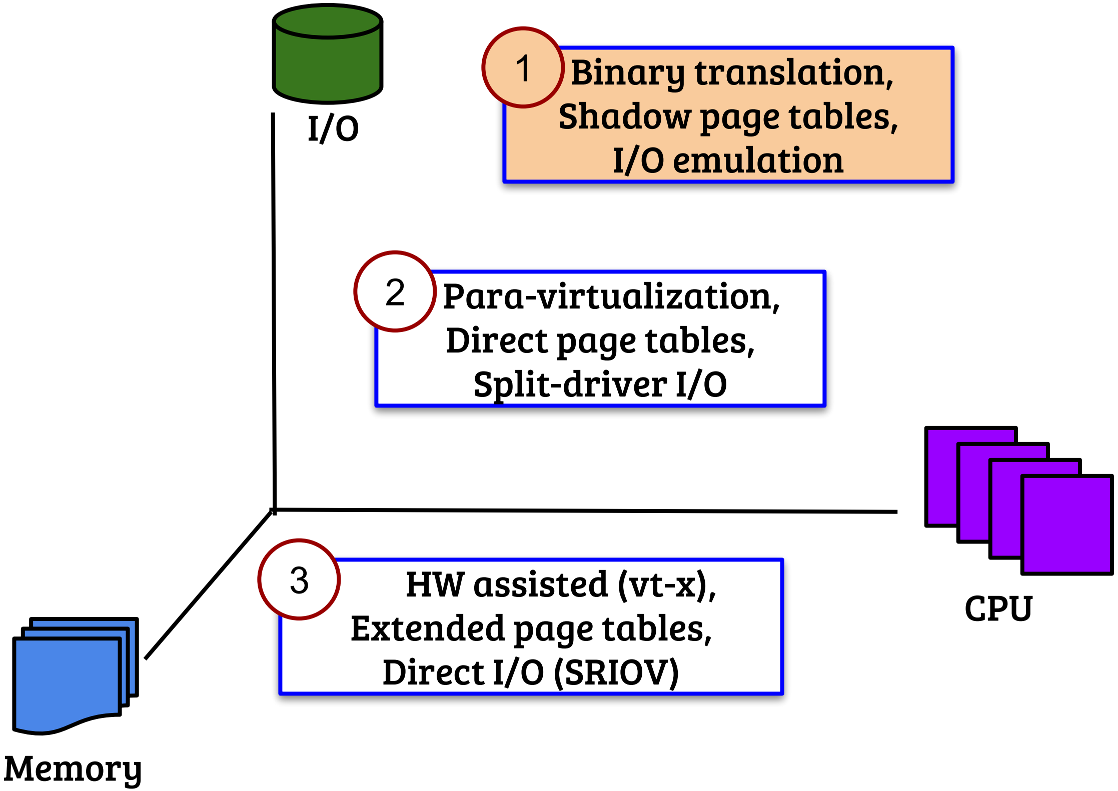
Resource control and isolation

- Already achieved by native systems - OS intervention to handle application I/O requests and I/O notifications

Efficiency

- Metrics: Drive the device capacity, other resource (CPU, memory) utilization

Overview of virtualization approaches



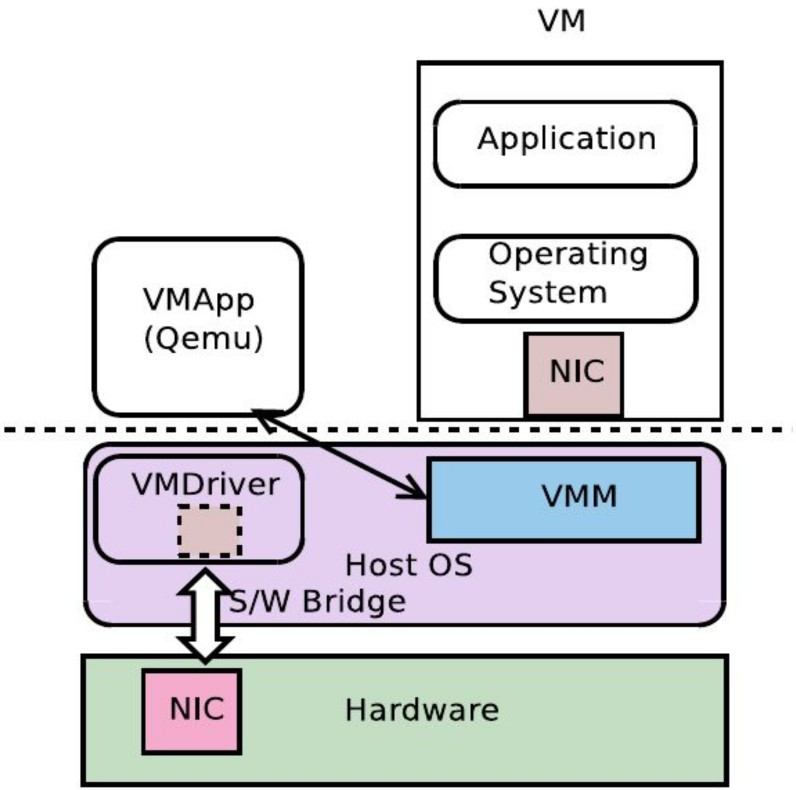
① Binary translation,
Shadow page tables,
I/O emulation

② Para-virtualization,
Direct page tables,
Split-driver I/O

③ HW assisted (vt-x),
Extended page tables,
Direct I/O (SRIOV)

- Agenda for today's lecture: I/O virtualization
- Software only techniques: device emulation, split-driver PV devices
- Hardware assisted: IOMMU and SRIOV

Emulated I/O [1]

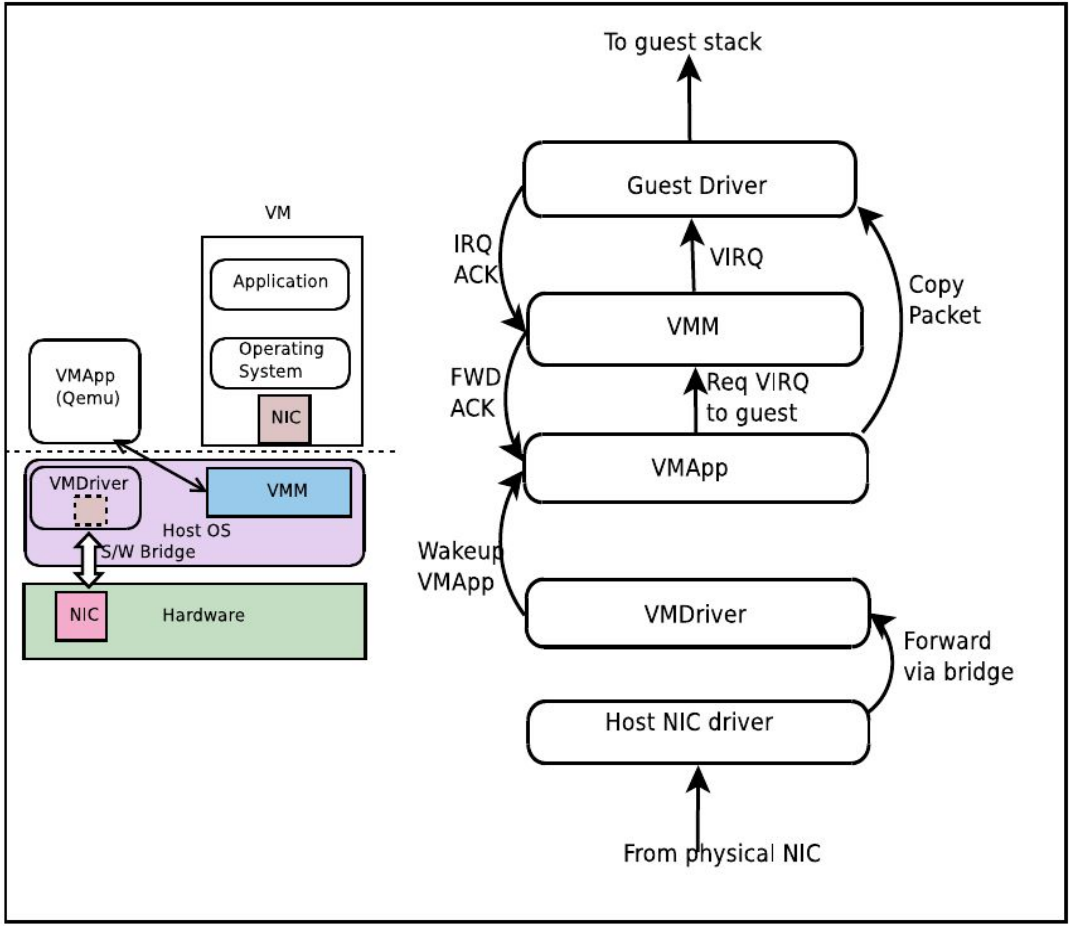


- VMM/hypervisor ⇒ CPU and memory virtualization, Emulator ⇒ BIOS and I/O
- Emulated BIOS, bus and devices allow the guest OSes discover the device like the native system
- An equivalent device state is maintained by the software emulator
- Device emulator invokes host APIs to perform the translated operation
  - Example: DD in the guest OS triggers transmission ⇒ emulator invoke send( )

---

1. J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstationâs Hosted Virtual Machine Monitor. USENIX ATC, 2001.

Emulated I/O: example packet receive



- Packet received by the emulator process through event notification mechanism (like select( ))
- VIRQ (virtual interrupt) handler for packet receive is registered by the guest OS
- Hypervisor invokes the handler after a receive complete notification by the VMApp

I/O emulation: discussion

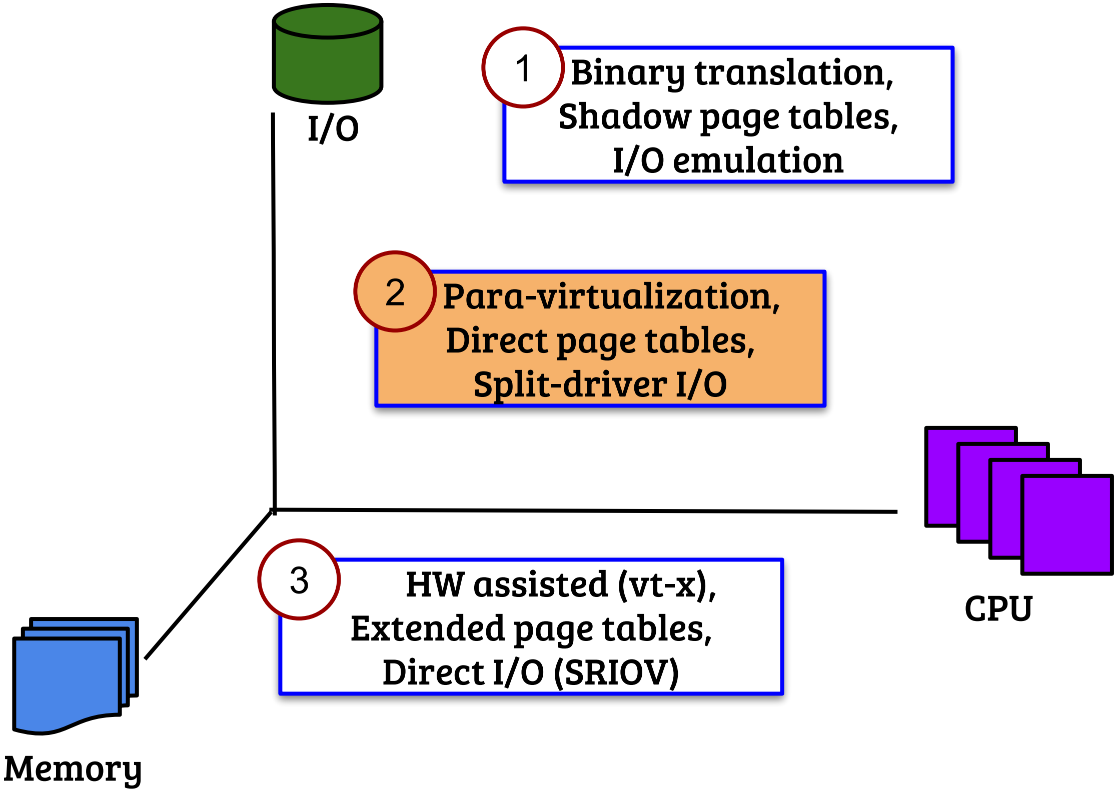
Virtualization requirements

- Equivalence is strictly adhered as device driver for physical device works for virtual device
- No extra efforts in the upper layers
- Resource control is easy as hypervisor is involved in all actions
- Not efficient ⟶ early designed could achieve 20% utilization for a 100Mbps NIC

Optimizations

- Avoid emulation of I/O instructions not resulting in meaningful I/O activity at the hypervisor (binary rewriting!)
- Packet combining and intermediate queuing
- Improved communication between emulator and hypervisor
- Device emulator ⇒ host OS?

Overview of virtualization approaches



I/O

1 Binary translation,
Shadow page tables,
I/O emulation

2 Para-virtualization,
Direct page tables,
Split-driver I/O

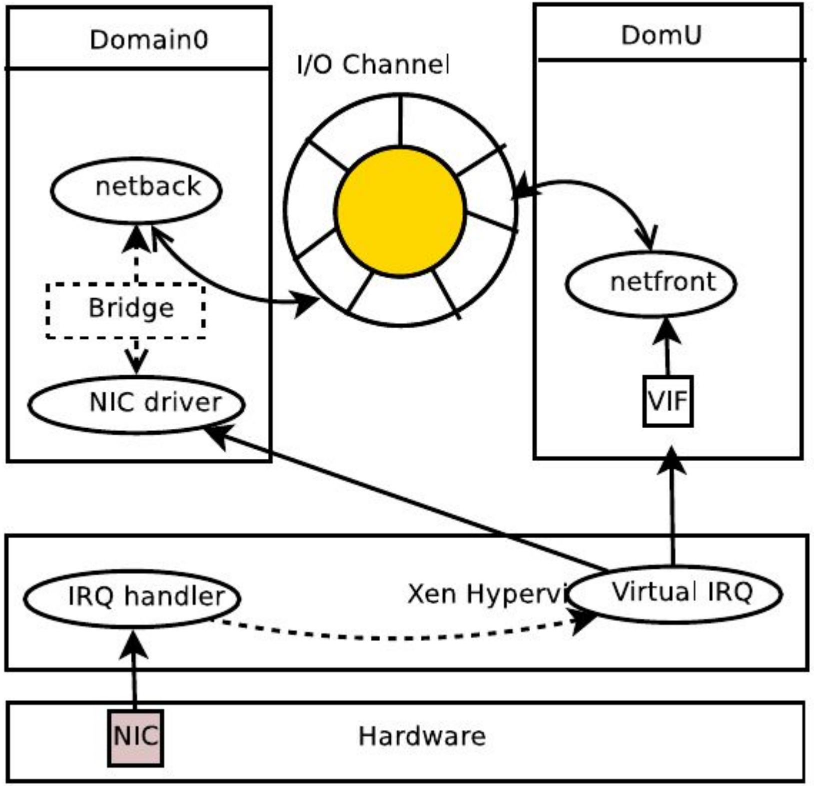
CPU

3 HW assisted (vt-x),
Extended page tables,
Direct I/O (SRIOV)

Memory

- Agenda for today’s lecture: I/O virtualization
- Software only techniques: device emulation, split-driver PV devices
- Hardware assisted: IOMMU and SRIOV

Xen domain-0 and split driver model [1]

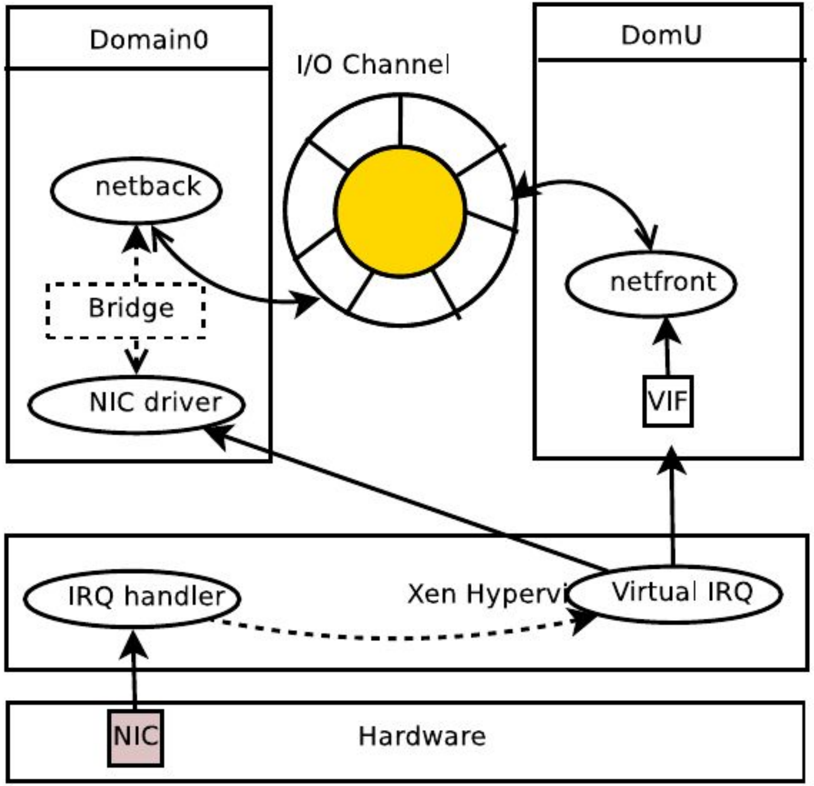


- Domain-0 is the management domain responsible for
  - Hosting device drivers
  - VM management
- Xen netback: backend device driver hosted in domain-0
- Xen netfront: frontend device driver hosted in other VMs (domU)
- In KVM (virtio_*)
  - Backend is in the host
  - Frontend is in the VM

---

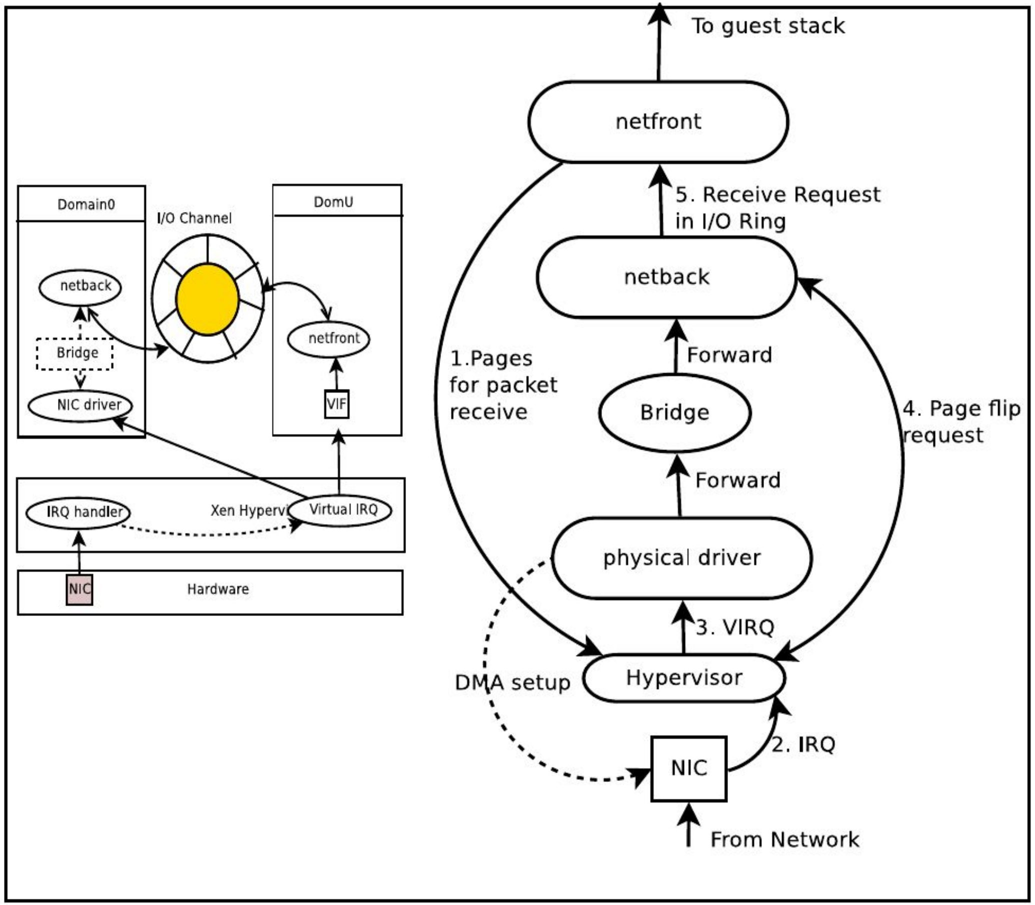
1. Xen and the art of virtualization, https://dl.acm.org/citation.cfm?id=945462

Xen domain-0 and split driver model



- Virtual interface is a stripped down version of a typical physical network (guest OS knows it!)
- I/O channels (a.k.a. I/O rings ^{1}) is realized by shared memory structures between the frontend and backend for communication
- Interrupt delivery is taken care by the hypervisor --- shadow IDT load on VCPU to PCPU assignment

Split driver receive



- DMA setup by physical device driver in domain-0
- IRQ and VIRQ raised by device and hypervisor, respectively
- (1) frontend provide pages to receive packets
- (4) ownership flip{ page containing the packet, front end provided page}
- (5) netback fills up the receive descriptor in I/O ring and raise VIRQ to the guest

Para-virtualized I/O: discussion

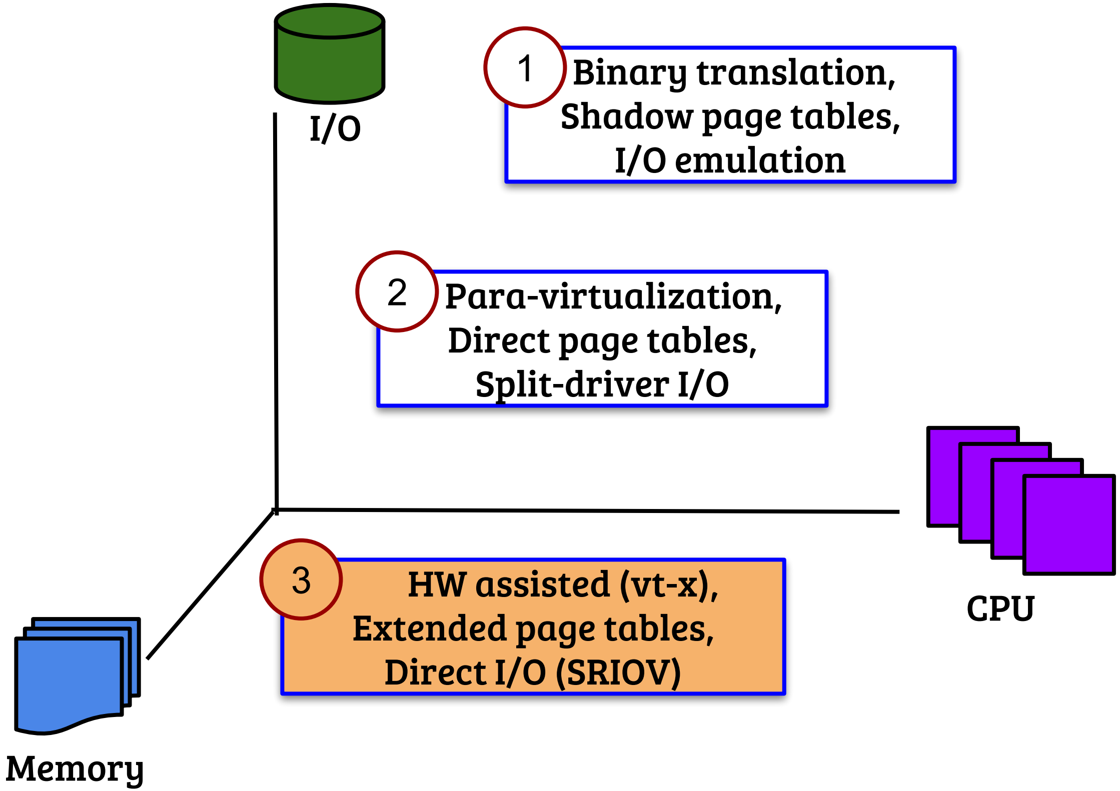
Virtualization requirements

- Equivalence is not strictly adhered, but everything above netfront remains unchanged
- Resource control is easy as hypervisor is involved in all actions
- Comparatively efficient w.r.t. I/O emulation, still a lot of overheads

Optimizations

- Page flipping replaced by page grant mechanism
- Event coalescing at different levels
- Leverage Multi Queue NIC support

Overview of virtualization approaches



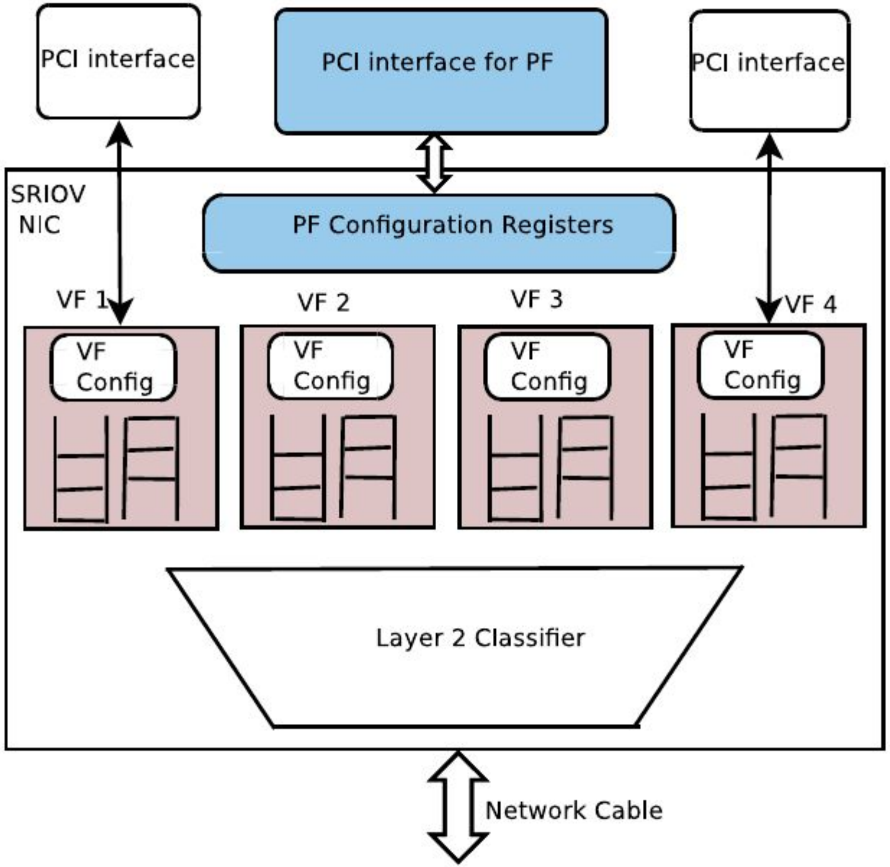
① Binary translation,
Shadow page tables,
I/O emulation

② Para-virtualization,
Direct page tables,
Split-driver I/O

③ HW assisted (vt-x),
Extended page tables,
Direct I/O (SRIOV)

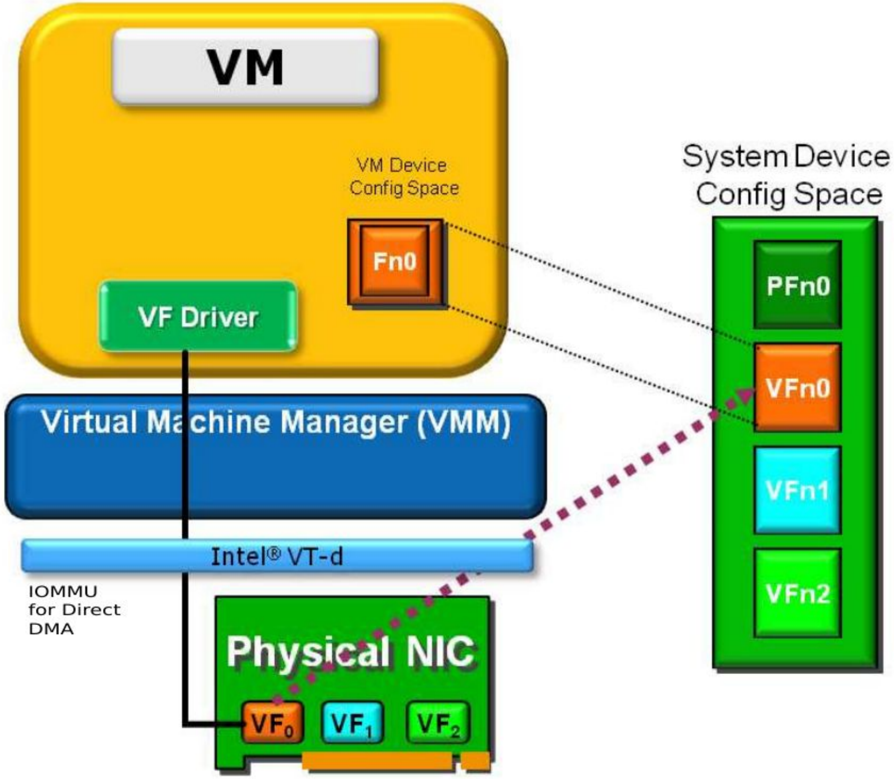
- Agenda for today's lecture: I/O virtualization
- Software only techniques: device emulation, split-driver PV devices
- Hardware assisted: IOMMU and SRIOV

Multifunction I/O devices



- H/W supports in-device partitioning of hardware resources
- Terminology
  - Physical function (PF)
  - Virtual function (VF)
- Each VF can be addressed through a separate PCI address (bus - dev - fn)

Multifunction I/O devices [1]

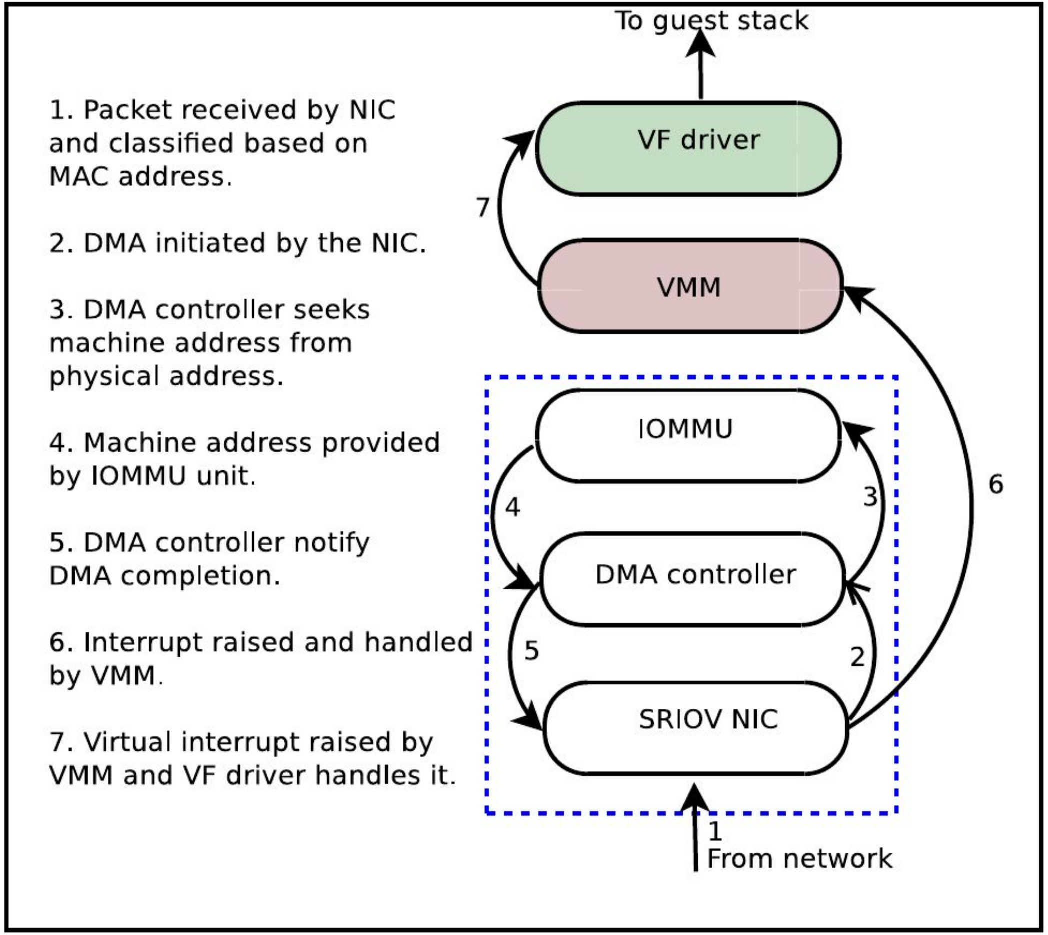


- System device configuration is performed by the hypervisor/host OS/domain-0 by loading the PF driver
- Most virtualization platforms allows direct assignment of PCI devices to the guest OS
- The guest OS loads the device driver for the VF device
  - Example: Intel igb and igbvf drivers
- IOMMU comes handy to enforce memory isolation

---

1. Intel documentation, PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html

Direct I/O receive



1. Packet received by NIC and classified based on MAC address.

2. DMA initiated by the NIC.

3. DMA controller seeks machine address from physical address.

4. Machine address provided by IOMMU unit.

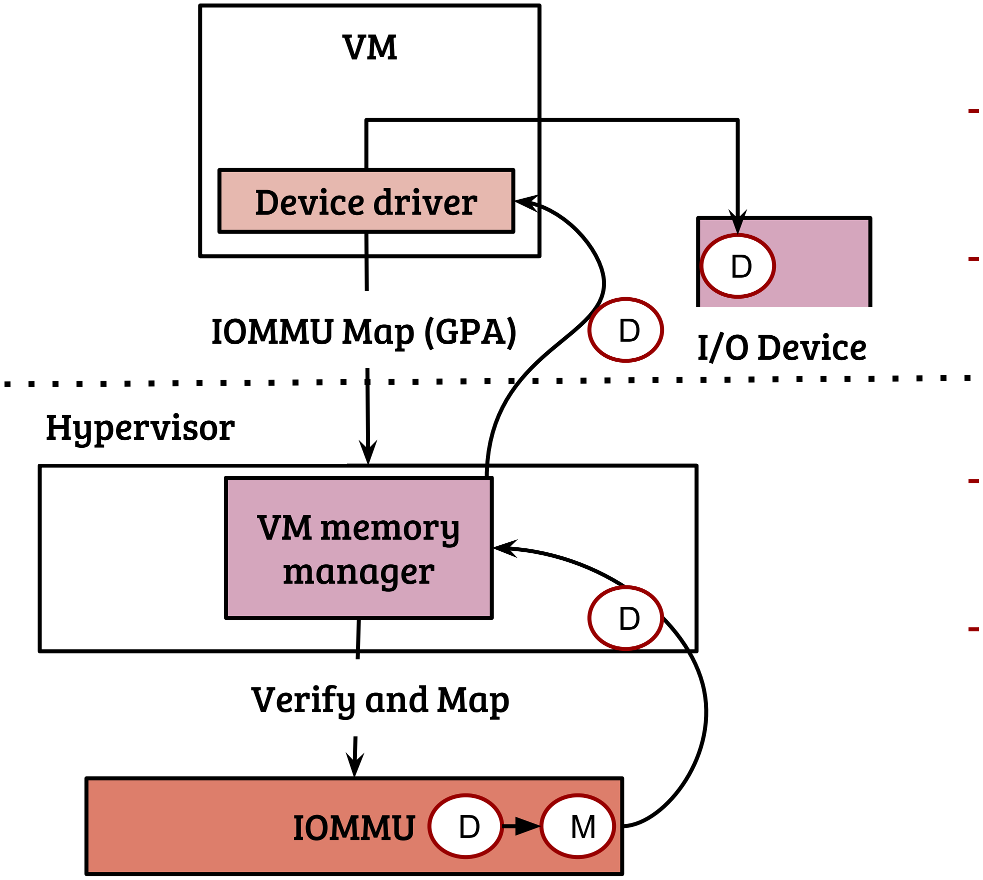
5. DMA controller notify DMA completion.

6. Interrupt raised and handled by VMM.

7. Virtual interrupt raised by VMM and VF driver handles it.

- Not completely direct I/O!
- Interrupt delivery and IOMMU setup happens through the hypervisor

Recap: IOMMU in virtualized systems



- Guest OS requests IOMMU mapping with guest physical address (GPA)
- Hypervisor validates the ownership (finds GPA ⇒ M) and performs the map and returns the DMA address (D)
- Device driver in guest OS configures the device with DMA address
- Device uses the DMA descriptor like a native system

Direct I/O: discussion

Virtualization requirements

- Equivalence is strictly adhered as device driver for physical device works for virtual device
- No extra efforts in the upper layers
- Resource control granularity is compromised (packet level ⟶ device level)
- Very efficient ⟶ early designed could achieve near native performance

Optimizations and other issues

- Interrupt delivery overhead optimizations (hardware and software)
- Broken features because of h/w dependency: migration, dynamic b/w control