Design and Engineering of Computer Systems

Lecture 20: Memory and I/O Virtualization

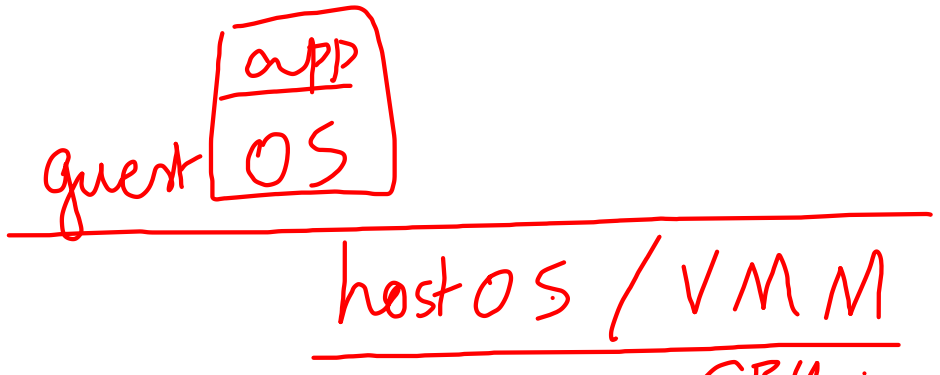
Mythili Vutukuru

IIT Bombay

Recap: Containers

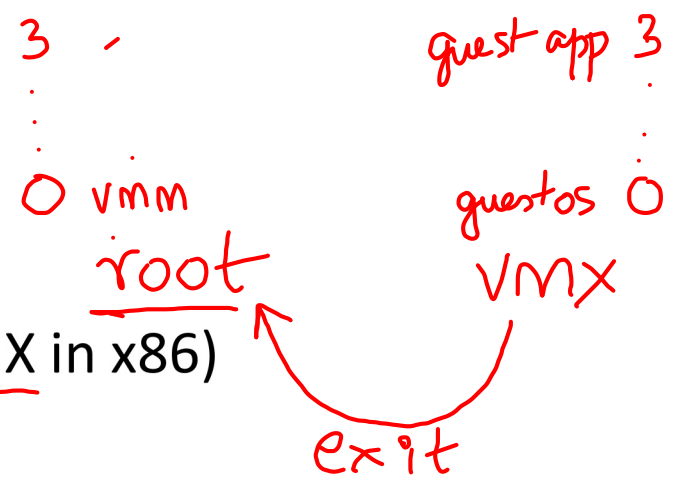
- Containers = lightweight virtualization
- Multiple containers share same base OS, provide isolated view
- Processes in a container are isolated from other containers
  - Separate process tree containing only processes in container
  - Separate root filesystem, network resources, libraries, …
  - Resource limits enforced for isolation
- Container frameworks like LXC and Docker used to create containers
  - Based on Linux namespaces and Cgroups
- Container orchestration frameworks (e.g. Kubernetes) manage lifecycle of multiple containers of an application spread across multiple physical nodes
  - Containers make it easy to deploy and manage large applications with many components in a distributed system

Recap: Virtual Machines

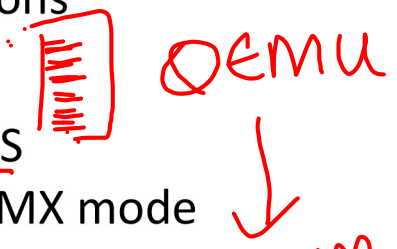


- Virtual machine: illusion of multiple machines over one physical machine
  - Useful to provide complete isolation of clients sharing servers in cloud computing
- Virtual machine monitor (VMM) / hypervisor multiplexes multiple guest OS on the underlying hardware and host OS
  - Guest OS and its applications isolated from other guests
- Basic idea of VMM: trap-and-emulate
  - Guest OS cannot be granted unrestricted access to hardware, must run at lower privilege
  - Privileged operations of guest OS trap to VMM, emulated by VMM
- Existing OS and CPUs not built for easy virtualization using trap-and-emulate, guest OS does not run correctly at lower privilege level
  - Paravirtualization: modify guest OS code to work at lower privilege level
  - Full virtualization: translate guest OS binary dynamically at run time
  - Hardware assisted virtualization: change underlying CPU to support virtualization

Hardware-assisted virtualization



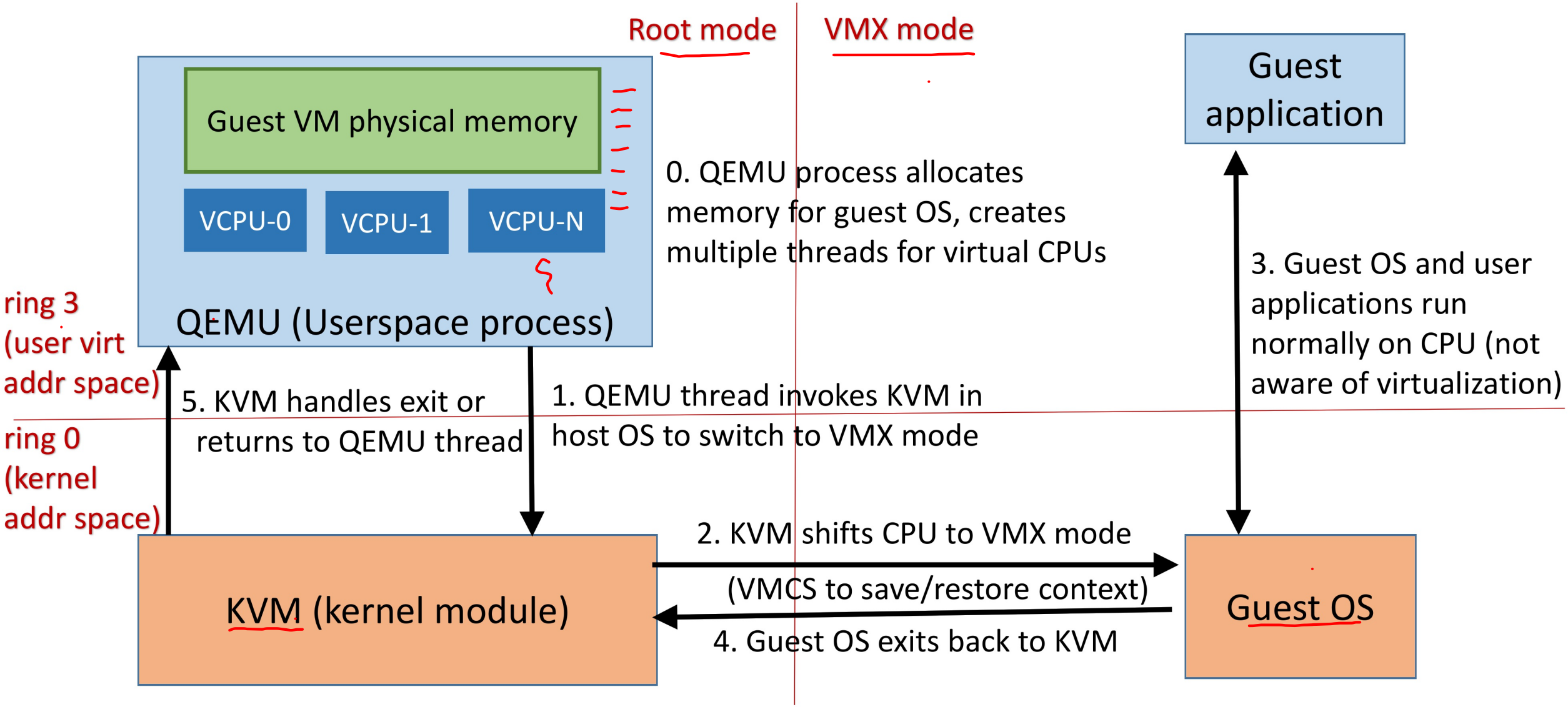
- Modern CPUs have special mode to run virtual machines (VMX in x86)
- VMX mode in x86 (similar modes exist in other architectures)
  - Separate rings 0-3 in regular (root) mode and VMX mode
  - Guest OS runs in VMX ring 0 (less powerful than regular ring 0)
  - VMM/host OS can configure guest VM to exit to regular mode for privileged operations
- Example: QEMU/KVM in Linux
  - QEMU is userspace process, allocates memory that serves as "RAM" for the guest OS
  - When QEMU process runs on CPU core, instructs KVM kernel module to switch to VMX mode
  - CPU core running QEMU now runs guest OS code (part of QEMU memory image) in VMX ring 0
  - QEMU has multiple threads to run guest OS code in parallel, one for each virtual CPU



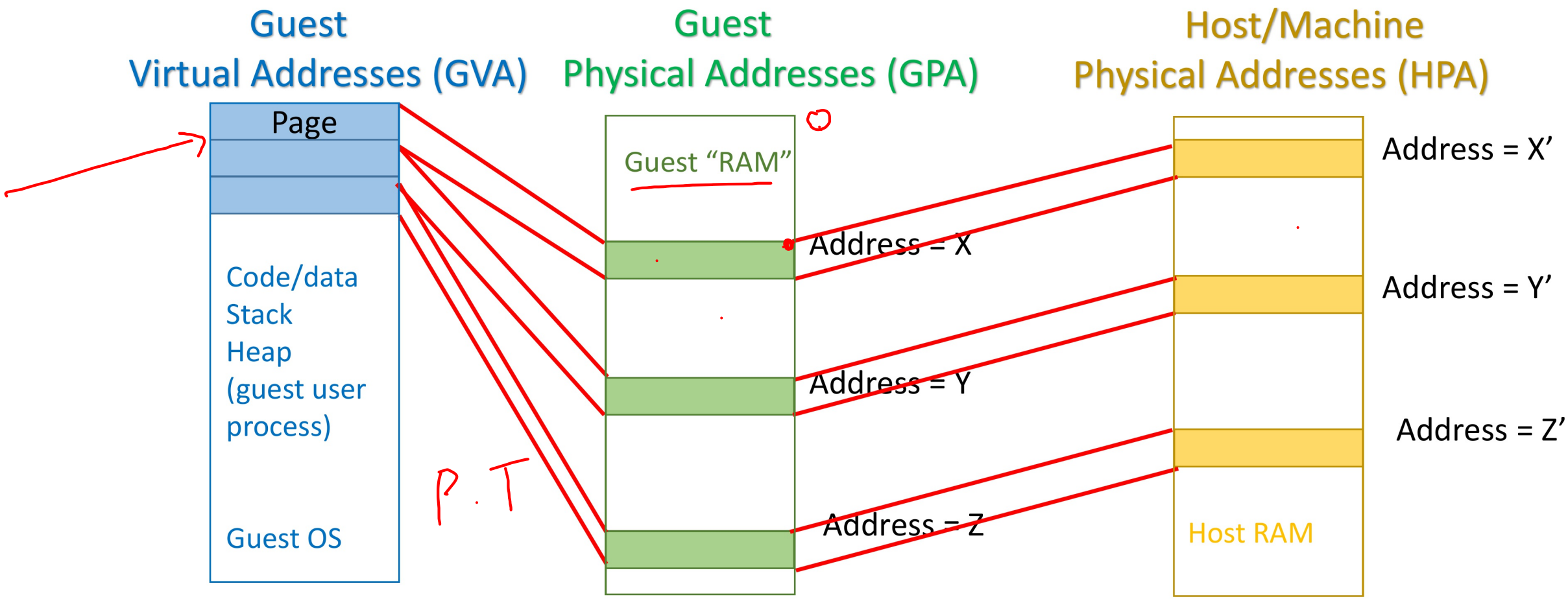
- What happens when KVM switches to VMX mode on CPU core?
  - Host OS stops running on CPU core, guest OS runs natively in VMX mode ring 0
  - Guest/host context is saved/restored during this "machine switch"
  - Context of host/guest stored in memory data structure called VMCS (VM control structure)

host ⟷ guest

Example: QEMU/KVM hypervisor



Memory management in VMs

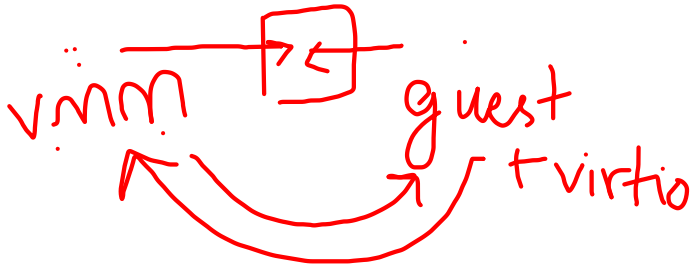


Guest "physical memory" is actually memory of the userspace hypervisor process running on the host, which is mapped to host physical memory by the host's page table

Techniques for memory virtualization

- Guest page table has GVA→GPA mapping
  - Each guest OS thinks it has access to all RAM starting at address 0
- VMM / Host OS has GPA→HPA mapping
  - Guest "RAM" pages are distributed across host physical memory
- Which page table should MMU use?
- Shadow paging: VMM creates a combined mapping GVA→HPA and MMU is given a pointer to this page table
  - VMM tracks changes to guest page table and updates shadow page table
- Extended page tables (EPT): MMU hardware is aware of virtualization, takes pointers to two separate page tables
  - MMU walks both page tables during address translation
- EPT is more efficient but requires hardware support

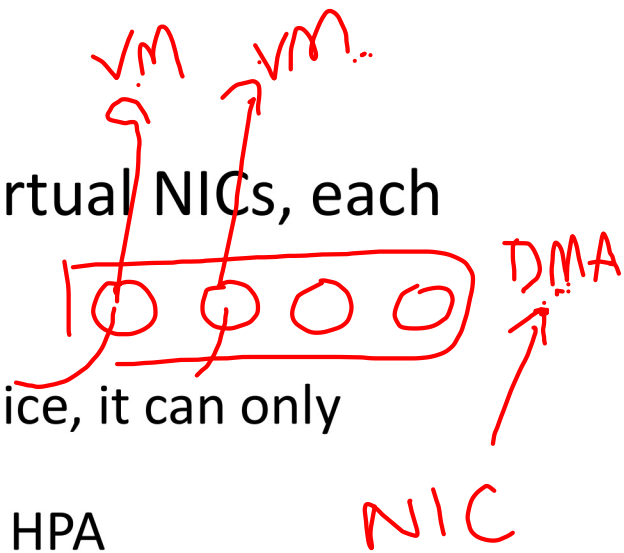
I/O Virtualization (1)



- Guest OS needs to access I/O devices, but cannot give full control of I/O to any one guest OS – how to perform I/O in guests?
- I/O Emulation: guest OS I/O operations trap to and emulated by VMM
  - Every time guest device driver gives command to I/O device, VM exit, VMM will issue command to I/O device on behalf of guest OS
  - Every time interrupt occurs, VM exit, VMM handles interrupt and injects it into guest VM it is destined for
  - I/O data DMA into host OS first, copied into guest later
  - Simple emulation slows I/O access down due to frequent VM exits, data copy
- Virtio: device drivers optimized for virtualization
  - Device driver batches I/O requests, exit once per batch of I/O requests
  - I/O data shared between guest and host via shared memory region, no copy

I/O Virtualization (2)

- Another way for guest OS to perform I/O: assign a slice of I/O device directly to each VM, each VM operates on its slice of device
  - Slices configured by VMM, does not compromise isolation of VMs
- Example: NICs with SR-IOV technology can create multiple virtual NICs, each slice is assigned to separate VM
  - Packet DMA directly into guest VM memory, host OS not involved
  - Challenge: when guest device driver provides DMA buffers to NIC slice, it can only provide guest physical addresses (GPA) of the buffer
  - NIC cannot access the DMA buffer memory using GPA alone, needs HPA
  - Solution: SR-IOV capable NICs have an inbuilt MMU (IOMMU) to translate from GPA to HPA
- Device passthrough techniques need hardware support, whereas virtio-based device drivers require only software upgrades for better performance



Summary

- In this lecture:
  - Recap of containers and virtual machines
  - Techniques for CPU, memory, I/O virtualization
- Install a VMM, start and run VMs. Understand the various techniques used for CPU, memory and I/O virtualization in your VMM.