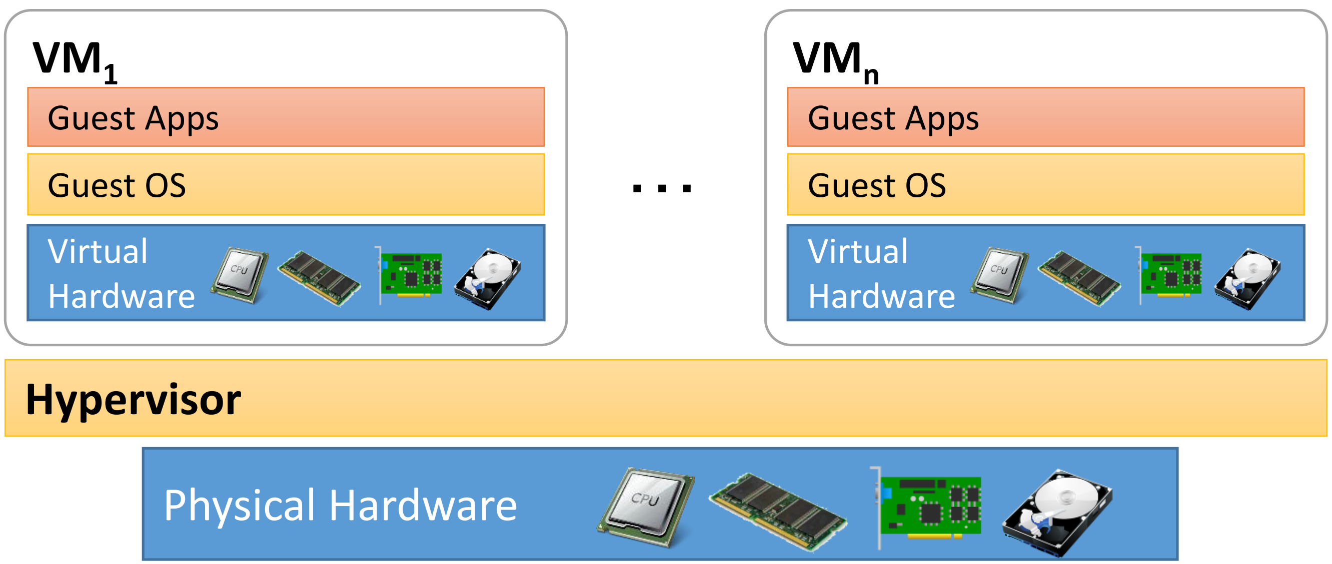
Introduction to
Virtual Machines

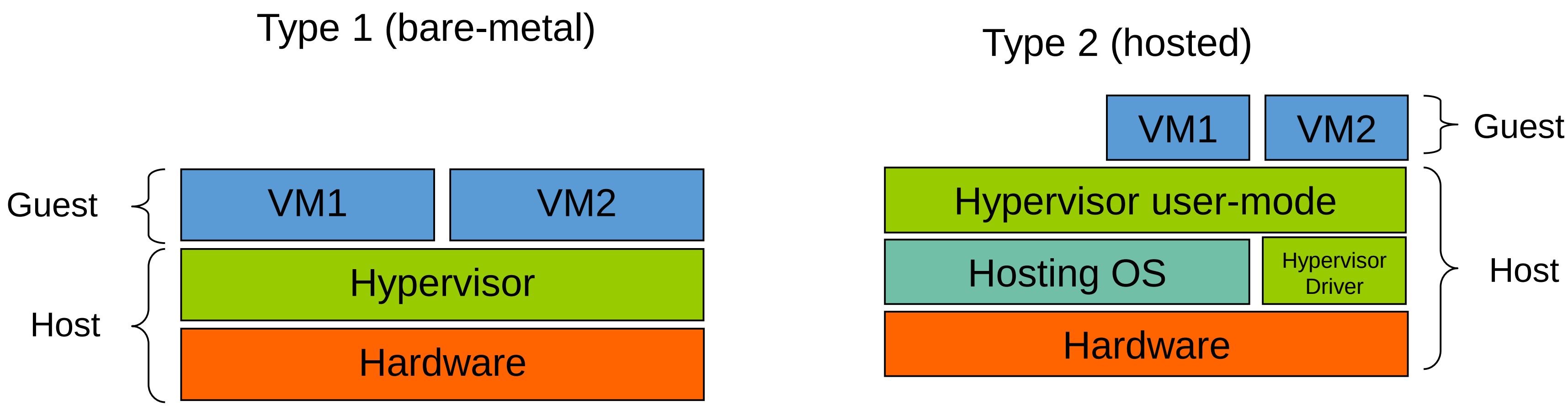
Nima Honarmand

Virtual Machines & Hypervisors

- ***Virtual Machine***: an abstraction of a complete compute environment through the combined virtualization of the processor, memory, and I/O components of a computer.
- ***Hypervisor***: system software that manages and runs virtual machines
  - Think of an OS sitting below virtual machines



Type-1 & 2 Hypervisors



VMware ESX, Microsoft Hyper-V, Xen, Qemu

VMware Workstation, Microsoft Virtual PC, Sun VirtualBox, QEMU, KVM

- Type-1: hypervisor controls the bare metal
  - mostly servers
- Type-2: hypervisor is hosted inside a host OS
  - mostly desktops

VM Use-cases (1)

1) Server consolidation: physical servers are often underutilized in data centers
   - Consolidate multiple virtual servers on a single physical server to improve utilization
   - Has numerous cost benefits: reduces hardware cost, electricity bills, maintenance overhead, deployment costs, etc.
   - Enables better fault tolerant: if a physical machine (or part of it) fails, move the VM image to a new machine
     - Improves service availability
     - Could be even done without shutting the VM down (**live migration**)

VM Use-cases (2)

2) Transparently adding services below operating systems (w/o OS or application modifications)
   - Encrypted storage
   - Logging of OS activities to provide time travel (e.g., roll-back the state) or replay features
   - Live migration

3) Software testing and development
   - Test your code on many different platforms, even ones that are not physically available
   - Developing system software on virtual platforms is much easier than physical ones
     - You’re doing something similar with JOS

VM Use-cases (3)

4) Desktop virtualization
   - Simultaneously have multiple OSes on your desktop to use their native apps

5) Support multiple users with larger isolation
   - Compared to sharing the same OS between multiple users
   - Each user can customize their OS the way they like

6) What else can you think of?

High-Level Requirements

- Security and Isolation
  - Hypervisor should be in complete control of the machine
  - VMs should be protected from each other
  - Hypervisor should be protected from VMs
- Performance
  - VM performance should be close to native (non-virtualized) execution
  - Means most VM instructions (both OS and apps) should execute directly on the processor
- Sounds familiar?
  - We said the same things when discussing kernel vs. application requirements (hypervisor → kernel, guest → application)

Problem with VMs

- What is special about the VMs then?
- ***Abstraction***
  - OS was free to choose the resource abstraction it exposed to applications
    - High-level, easy-to-provide abstractions such as <u>threads</u>, <u>files</u> and <u>sockets</u> instead of <u>processors</u>, <u>disks</u>, <u>interrupts</u>, <u>I/O devices</u>, etc.
  - VMs are an after-thought; Guest OS has already been written assuming a particular hardware model
    - Low-level CPU and MMU details are hard-coded in the OS
    - Same for I/O devices
  - Hypervisor has to provide the abstraction expected by guest OS (or something very close)

Approach #1: Full Virtualization

- Hardware exposed to guest mimics a real hardware configuration
  - No change required to the guest OS
- Does not have to be exactly the same as underlying machine
  - E.g., can have smaller memory, fewer processor cores, different I/O devices, etc.
  - But should look and feel like some real machine

Problems w/ Full Virtualization

- To protect hypervisor, guest kernel may not run in privileged mode
- Idea: let’s run the whole guest (both kernel and user) in unprivileged mode
- But any kernel will have to perform privileged operations; what we should do about them?
- Idea: ***Trap-and-Emulate***
  - CPU will fault when guest OS tries to execute a privileged operation
  - Hypervisor then takes over, decodes the operation and emulates its effect
  - Examples: a system call on the guest OS

Problems w/ Full Virtualization

- Two problems with trap-and-emulate:
  1) Too many traps will cause severe performance degradation
  2) Not all *sensitive* instructions will generate traps in all architectures
     - Example: SIDT and POPF in x86
     - Guest OS (running in Ring 3) can read, and be confused by, privileged state belonging to hypervisor
- A solution to (2): ***Binary translation***
  - The hypervisor will analyze all of guest code, and replace non-trapping sensitive instructions w/ explicit traps
  - This idea gave birth to VMware, enabling it to virtualize x86 efficiently

Approach #2: Para-virtualization

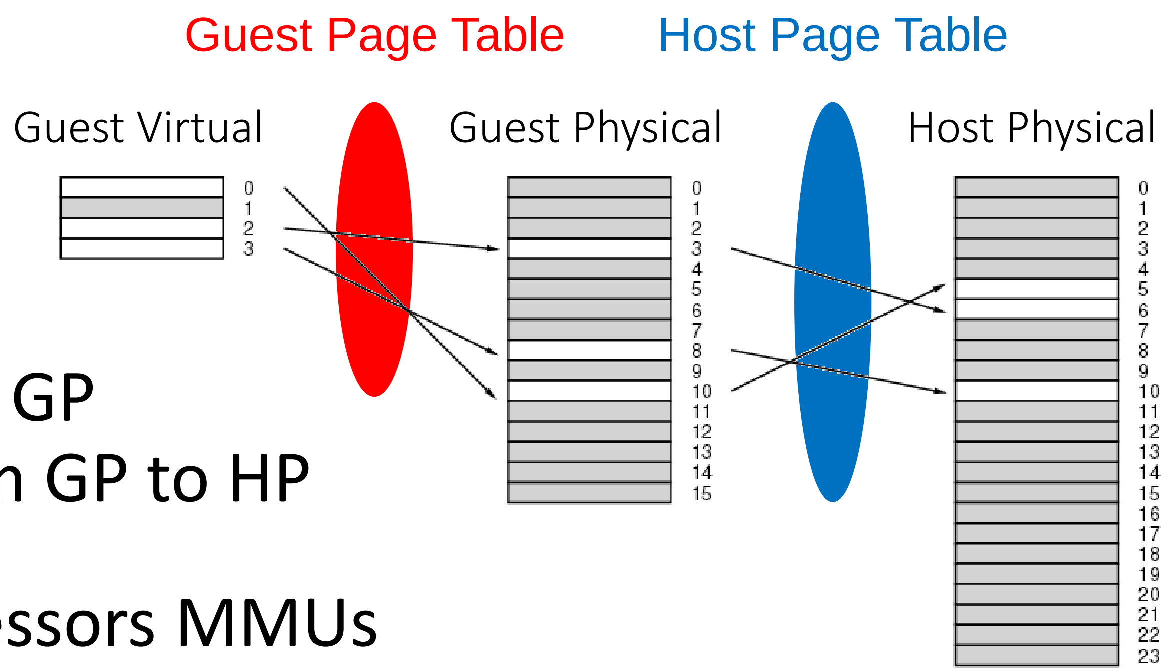
- Expose a different, virtualization-friendly abstraction to the guest OS
  - Will require changes to the guest OS (hopefully not so big)
  - Xen reported 1.5% code change for Linux and 0.04% for Windows XP
- Guest OS knows that it is being virtualized and run w/ reduced privilege
  - It is careful w/ unprivileged sensitive operations
  - It avoids most trap-and-emulate situations by using explicit ***hypercalls*** to the hypervisor

What to Virtualize?

- Three things:
  1) CPU (everything, including the privileged state)
  2) Memory Management Unit (MMU)
  3) I/O Devices
- (1) and (3) are simpler
  - Just trap on relevant accesses, or binary translate them, or have the guest OS make hypercalls
- (2) is more complicated
  - It requires ***shadow page tables*** in absence of virtualization-hardware support

MMU Virtualization Problem

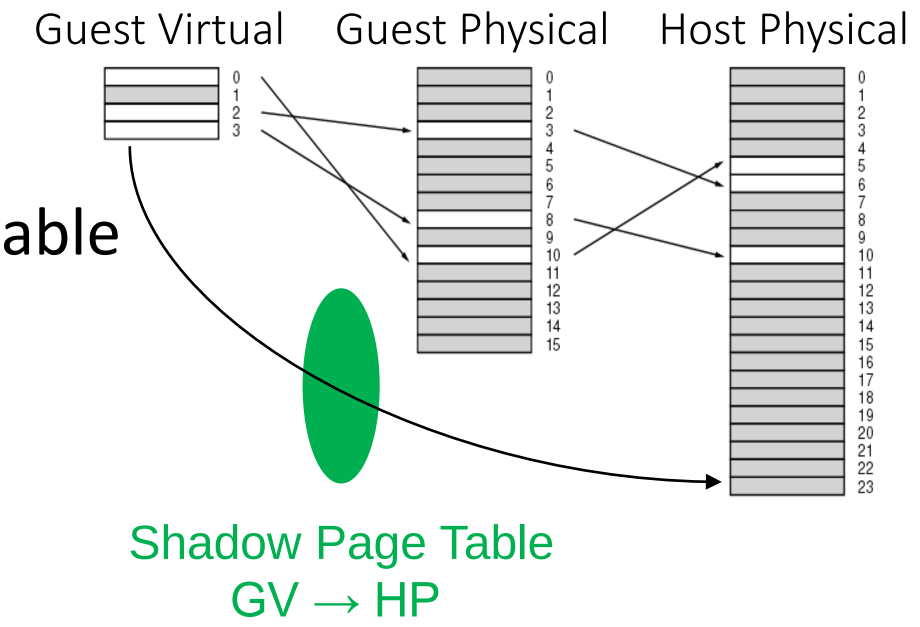
- We deal with three types of memory spaces in a virtualized environment
  - Guest Virtual
  - Guest Physical
  - Host Physical



- Need one page table from GV to GP and another from GP to HP
- But (older) processors MMUs can only deal with one page table

Shadow Page Tables

- For each (GV → GP → HP) need to combine the two translations into a single translation kept in a ***shadow page table***
- For this, the hypervisor needs to know of any change to the guest page table
  - I.e., GV → GP translation
- **Full virtualization**: mark guest page table pages read only → trap on every page table change
- **Para-virtualization**: make a hypercall to the hypervisor to change the shadow page table



Approach #3: Hardware-Assisted Virtualization

- Trap-and-emulate is expensive
- Too many changes to guest (for para-virt) not always desirable

→ Change hardware to make it virtualization friendly
  - E.g., **Intel VT-x** and **AMD-V** technologies

- Hardware support to eliminate most trap-and-emulate situations
  - CPU: duplicate the entire architecturally visible state of the processor in separate ***root*** (for the hypervisor) ***and non-root*** (for guests) modes
    - Makes most traps into hypervisor unnecessary
  - MMU: HW supports second layer of page table (managed by hypervisor)
    - Makes shadow page tables unnecessary
  - I/O: add support for high-performance I/O through IOMMU and SR-IOV
    - Enables direct hardware access by guests
- State of the art
  - Will discuss extensively in the student presentations

Lightweight Virtualization

- Also known as ***containers***
  - Examples: Docker, OpenVZ, Linux Containers (LXC)
- All processes on the machine share the same kernel
- Each container will use a different user-mode image of a compatible OS
  - For example, different Linux distributions (as long as they work w/ underlying kernel)
  - Each container typically has a different root directory
  - Recall per-process roots in VFS lecture
- Not real virtualization; just different user-mode OS images sharing the same HW & kernel