# CS 318 Principles of Operating Systems

## Fall 2020

## Lecture 18: Virtual Machine Monitors

Prof. Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

# Administrivia

- **Last lab is out**
- **Good news:**
  - Given everything going on this semester, Lab 4 will be optional for 418/618 sections as well
  - Still encourage you to do this lab to better understand file systems
- **Lab 4 overview session**
  - Wednesday 11/18 8-9 PM

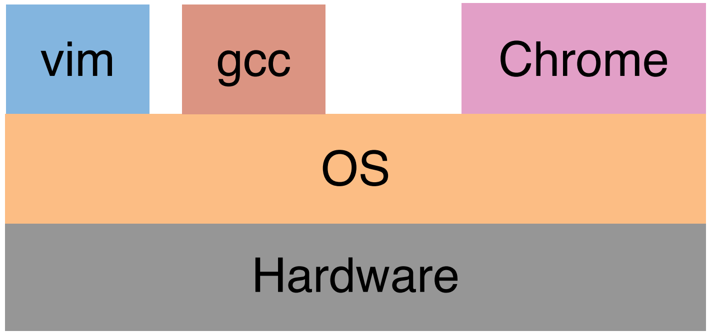
# So Far...

- **We've covered the three fundamental concepts in OS**
  - Concurrency
  - Virtualization
  - Persistency

✅ **A major milestone of the course**

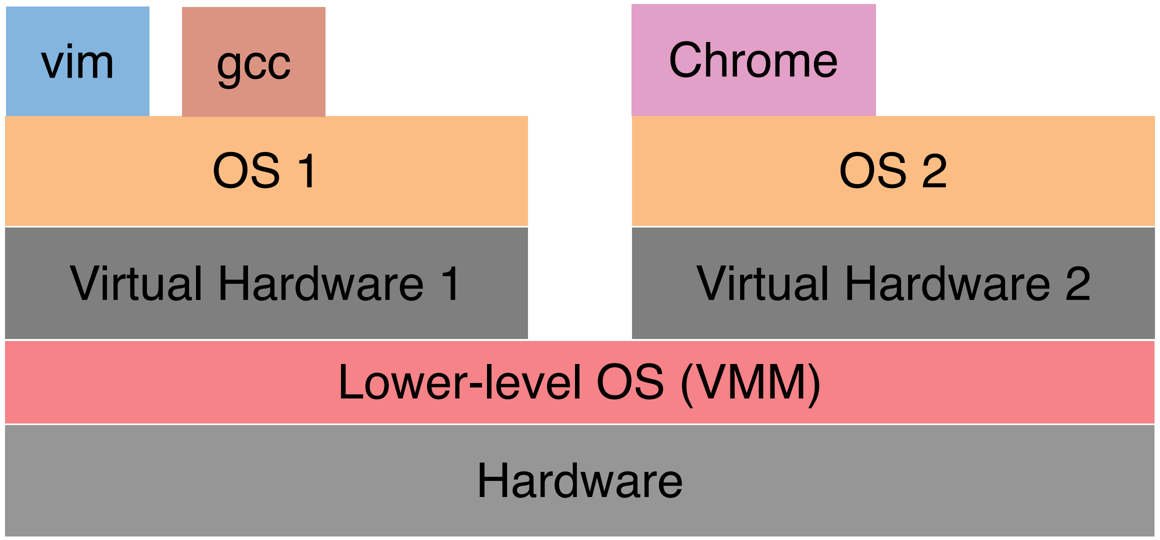
- **Remaining lectures are slightly advanced (but important) OS topics**

# Review: What Is An OS




- **OS is software between applications and hardware**
  - Abstracts hardware to makes applications portable
  - Makes finite resources (memory, # CPU cores) appear much larger
  - Protects processes and users from one another

# What If…




- **The process abstraction looked just like hardware?**

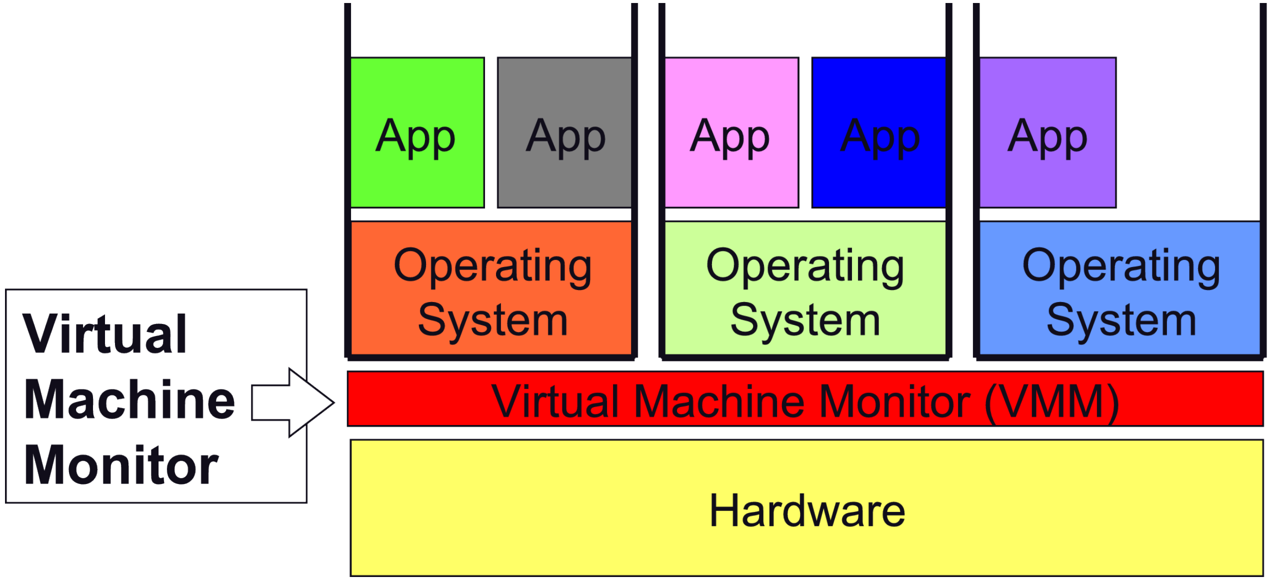
# How Do Process Abstraction & H/W Differ

- **Process**
  - Non-privileged registers and instructions
  - Virtual memory
  - Errors and signals
  - File systems, directories, files, raw devices

- **Hardware**
  - All registers and instructions
  - Both virtual and physical memory, MMU functions, TLB/page tables,..
  - Trap, interrupts
  - I/O devices accessed through programmed I/O, DMA, interrupts

# Virtual Machine Monitor

- **Thin layer of software that virtualizes the hardware**
  - Exports a virtual machine abstraction that looks like the hardware
  - Provides the illusion that software has full control over the hardware
    - Run multiple instances of an OS or different OSes simultaneously on the same physical machine



# Old Idea from The 1970s

- **IBM VM/370 – A VMM for IBM mainframe**
  - Multiplex multiple OS environments on expensive hardware
  - Desirable when few machines around
- **Interest died out in the 1980s and 1990s**
  - Hardware got cheap
  - Compare Windows NT vs. N DOS machines
- **Revived by the Disco [SOSP '97] work**
  - Led by Mendel Rosenblum, later lead to the foundation of VMware
- **Another important work Xen [SOSP '03]**

# VMMs Today

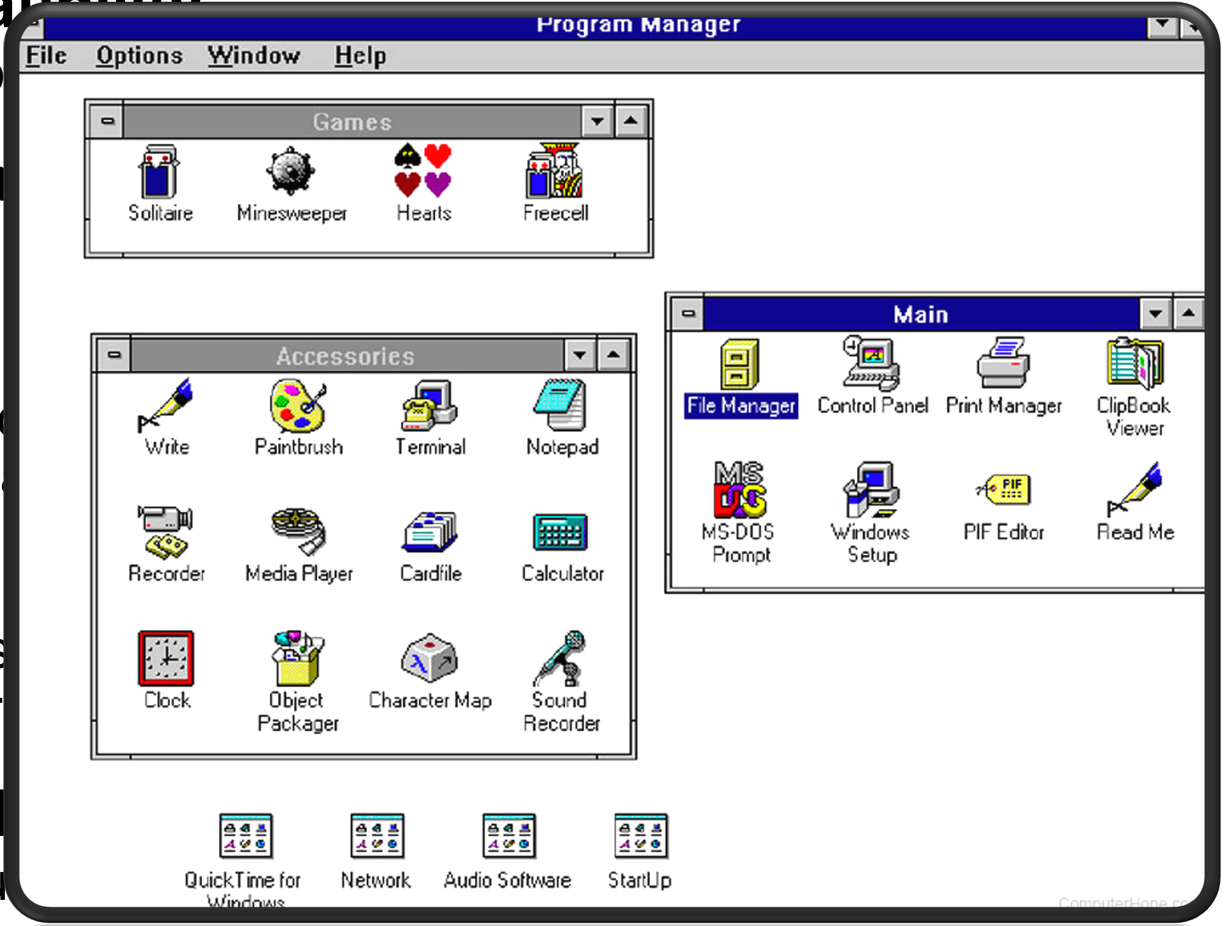
- **Today VMs are used everywhere**
  - Popularized by cloud computing
  - Used to solve different problems
- **VMMs are a hot topic in industry and academia**
  - Industry commitment
    - Software: VMware, Xen,…
    - Hardware: Intel VT, AMD-V
      - If Intel and AMD add it to their chips, you know it's serious…
  - Academia: lots of related projects and papers

# Why Would You Do Such a Crazy Thing?

- **Software compatibility**
  - VMMs can run pretty much all software
- **Resource utilization**
  - Machines today are powerful, want to multiplex their hardware
- **Isolation**
  - Seemingly total data isolation between virtual machines
  - Leverage hardware memory protection mechanisms
- **Encapsulation**
  - Virtual machines are not tied to physical machines
  - Checkpoint/migration
- **Many other cool applications**
  - Debugging, emulation, security, speculation, fault tolerance…

# Why Would You Do Such a Crazy Thing?

- **Software compatibility**
  - VMMs can run p
- **Resource utiliza**
  - Machines today
- **Isolation**
  - Seemingly total
  - Leverage hardw
- **Encapsulation**
  - Virtual machines
  - Checkpoint/migr
- **Many other cool**
  - Debugging, emu



# OS Backwards Compatibility

- **Backward compatibility is bane of new Oses**
  - Huge effort required to innovate but not break
- **Security considerations may make it impossible**
  - Choice: Close security hole and break apps or be insecure
- **Example: Windows XP is end of life**
  - Eventually hardware running WinXP will die
  - What to do with legacy WinXP applications?
  - Not all applications will run on later Windows
  - Given the # of WinXP applications, practically any OS change will break something

```
if (OS == WinXP) ...
```

- **Solution: Use a VMM to run both WinXP and Win10**
  - Obvious for OS migration as well: Windows → Linux

# Logical Partitioning of Servers

- **Run multiple servers on same box (e.g., Amazon EC2)**
  - Modern CPUs more powerful than most services need: e.g., only 10% utilization
  - VMs let you give away less than one machine for running a service
  - Server consolidation: N machines → 1 real machine
  - Consolidation leads to cost savings (less power, cooling, management, *etc.*)
- **Isolation of environments**
  - Printer server doesn't take down Exchange server
  - Compromise of one VM can't get at data of others
- **Resource management**
  - Provide service-level agreements
- **Heterogeneous environments**
  - Linux, FreeBSD, Windows, etc.

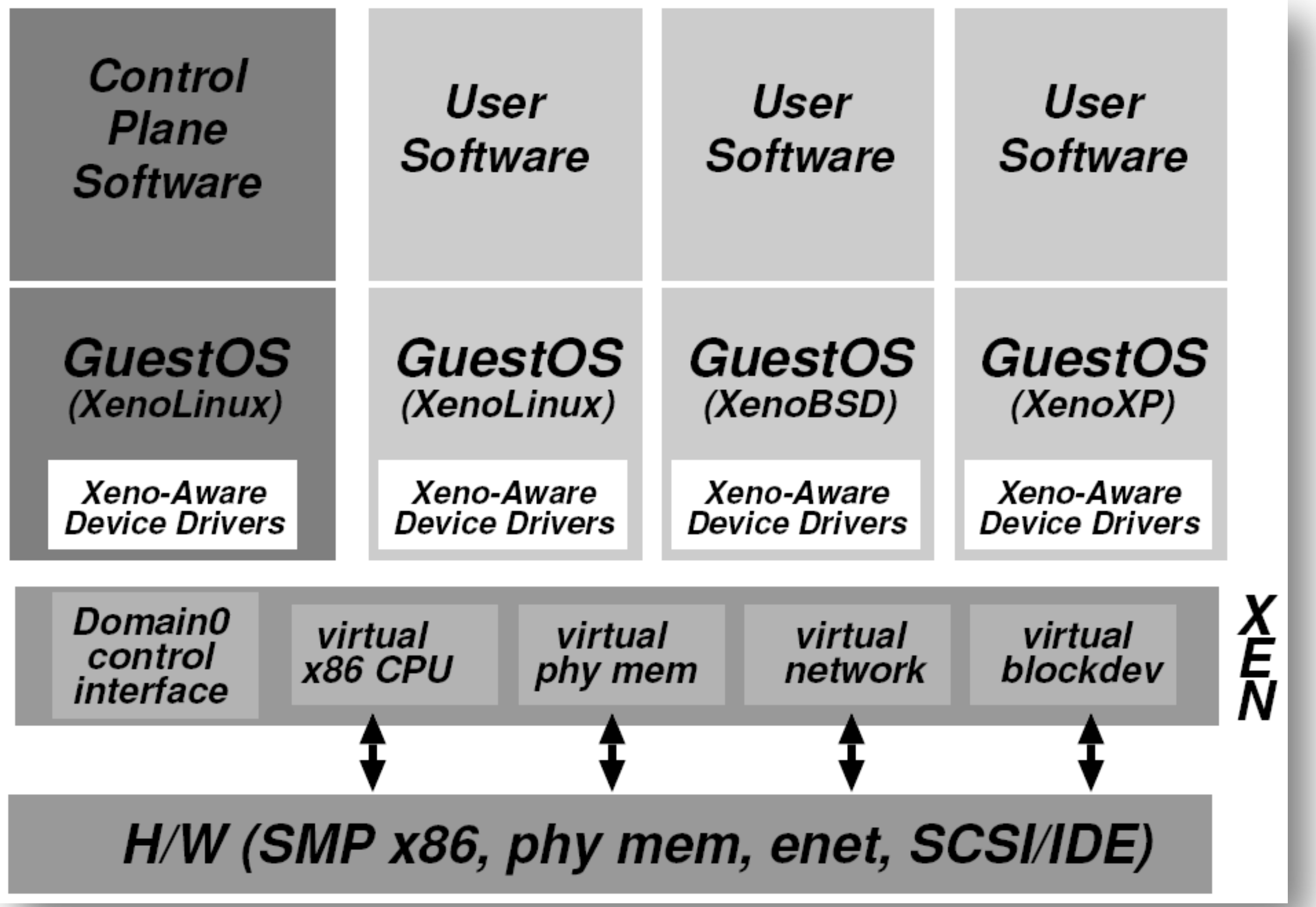
# Implementing VMMs - Requirements

- **Fidelity**
  - OSes and applications work the same without modification
    - (although we may modify the OS a bit)
- **Isolation**
  - VMM protects resources and VMs from each other
- **Performance**
  - VMM is another layer of software…and therefore overhead
    - As with OS, want to minimize this overhead
  - VMware (early):
    - CPU-intensive apps: 2-10% overhead
    - I/O-intensive apps: 25-60% overhead (much better today)

# VMM Case Study 1: Xen

- **Early versions use "paravirtualization"**
  - Fancy word for "we have to modify & recompile the OS"
  - Since you're modifying the OS, make life easy for yourself
  - Create a VMM interface to minimize porting and overhead
- **Xen hypervisor (VMM) implements interface**
  - VMM runs at privilege, VMs (domains) run unprivileged
  - Trusted OS (Linux) runs in own domain (Domain0)
    - Use Domain0 to manage system, operate devices, etc.
- **Most recent version of Xen does not require OS mods**
  - Because of Intel/AMD hardware support
- **Commercialized via XenSource, but also open source**

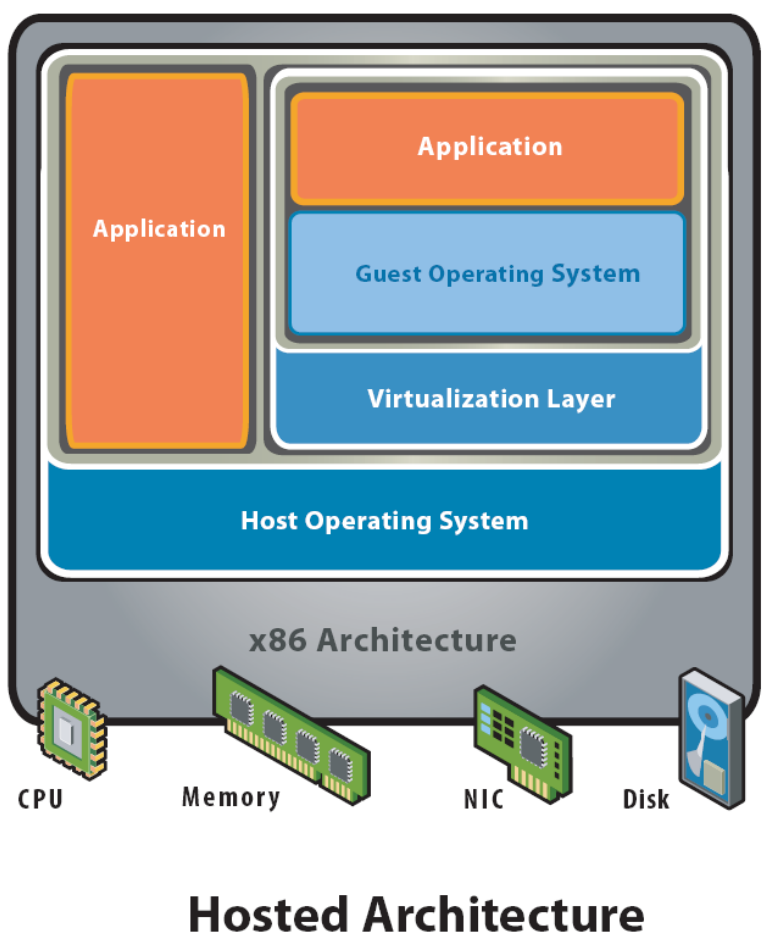
# Xen Architecture



# VMM Case Study 2: VMware

- **VMware workstation uses hosted model**
  - VMM runs unprivileged, installed on base OS (+ driver)
  - Relies upon base OS for device functionality
- **VMware ESX server uses hypervisor model**
  - Similar to Xen, but no guest domain/OS
- **VMware uses software virtualization**
  - Dynamic binary rewriting translates code executed in VM
    - Most instructions translated identically, e.g., `movl`
    - Rewrite privileged instructions with emulation code (may trap), e.g., `popf`
  - Think JIT compilation for JVM, but
    - full binary x86 → IR code → safe subset of x86
  - Incurs overhead, but can be well-tuned (small % hit)

# VMware Hosted Architecture



# What Needs to Be Virtualized?

- **Exactly what you would expect**
  - CPU
  - Events (exceptions and interrupts)
  - Memory
  - I/O devices
- **Isn't this just duplicating OS functionality in a VMM?**
  - Yes and no
  - Approaches will be similar to what we do with OSes
    - Simpler in functionality, though (VMM much smaller than OS)
  - But implements a different abstraction
    - Hardware interface vs. OS interface

# Approach 1: Complete Machine Simulation

- **Simplest VMM approach, used by `bochs`**
- **Run the VMM as a regular user application atop a host OS**
- **Application simulates all the hardware (i.e., a simulator)**
  - CPU – A loop that fetches each instruction, decodes it, simulates its effect

```
while (1) {
  curr_instr = fetch(virtHw.PC);
  virtHw.PC += 4;
  switch (curr_instr) {
    case ADD:
      int sum = virtHw.regs[curr_instr.reg0] +
        virtHw.regs[curr_instr.reg1];
      virtHw.regs[curr_instr.reg0] = sum;
      break;
    case SUB:
      //...
```

  - Memory – Memory is just an array, simulate the MMU on all memory accesses
  - I/O – Simulate I/O devices, programmed I/O, DMA, interrupts

# Approach #1: Complete Machine Simulation

- **Simplest VMM approach, used by `bochs`**
- **Run the VMM as a regular user application atop a host OS**
- **Application simulates all the hardware (i.e., a simulator)**
- **Problem: Too slow!**
  - CPU/Memory – 100x CPU/MMU simulation
  - I/O Device – < 2× slowdown.
  - 100× slowdown makes it not too useful
- **Need faster ways of emulating CPU/MMU**

# Approach #2: Direct Execution w/ Trap and Emulate

- **Observations: Most instructions are the same regardless of processor privileged level**
  - Example: `incl %eax`
- **Why not just give instructions to CPU to execute?**
  - One issue: Safety – How to get the CPU back? Or stop it from stepping on us? How about `cli/halt`?
  - Solution: Use protection mechanisms already in CPU
- **Run virtual machine's OS directly on CPU in unprivileged user mode**
  - "Trap and emulate" approach
  - Most instructions just work
  - Privileged instructions trap into monitor and run simulator on instruction
  - Makes some assumptions about architecture: processor is "virtualizable"

# Virtualizable Processor (1)

- **Sensitive instructions access low-level machine states**
- **Virtualizable CPU: all sensitive instructions are privileged**
- **For many years, x86 chips were *not* virtualizable**
  - On the Pentium chip, 17 instructions were not virtualizable
  - Example: `push` instruction pushes a register value onto the top of the stack
    - `%cs` register contains (among other things) 2 bits representing the current privilege level
    - A guest OS in Ring 1 could push `%cs` and see that the privilege level isn't Ring 0!
    - To be virtualizable, `push` should cause a trap when invoked from Ring 1, allowing the VMM to push a fake `%cs` value which indicates that the guest OS is running in Ring 0

# Virtualizable Processor (2)

- **For many years, x86 chips were *not* virtualizable**
  - On the Pentium chip, 17 instructions were not virtualizable
  - Example: `push` instruction pushes a register value onto the top of the stack
  - Another example: `pushf/popf` read/write the `%eflags`
    - Bit 9 of `%eflags` enables interrupts
    - In Ring 0, `popf` can set bit 9, but in Ring 1, CPU silently ignores `popf`!
    - To be virtualizable, `pushf/popf` should cause traps in Ring 1 so that the VMM can detect when guest OS wants to changes its interrupt level

# Virtualizing Traps

- **What happens when an interrupt or trap occurs**
  - Like normal kernels: we trap into the monitor
- **What if the interrupt or trap should go to guest OS?**
  - Example: Page fault, illegal instruction, system call, interrupt
  - Re-start the guest OS execution simulating the trap
- **x86 example:**
  - Give CPU an IDT that vectors back to VMM
  - Look up trap vector in VM's "virtual" IDT
    - How does VMM know this?
  - Push virtualized `%cs, %eip, %eflags`, on stack
  - Switch to virtualized privileged mode

# Virtualizing Memory

- **OSes assume they have full control over memory**
  - Managing it: OS assumes it owns it all
  - Mapping it: OS assumes it can map any virtual page to any physical page
- **But VMM partitions memory among VMs**
  - VMM needs to assign hardware pages to VMs
  - VMM needs to control mappings for isolation
    - Cannot allow an OS to map a virtual page to any hardware page
    - OS can only map to a hardware page given to it by the VMM
- **Hardware-managed TLBs make this difficult**
  - When the TLB misses, the hardware automatically walks the page tables in memory
  - As a result, VMM needs to control access by OS to page tables

# One Solution: Direct Mapping

- **VMM uses the page tables that a guest OS creates**
  - These page tables are used directly by hardware MMU
- **VMM validates all updates to page tables by guest OS**
  - OS can read page tables without modification
  - But VMM needs to check all PTE writes to ensure that the virtual-to-physical mapping is valid
    - That the OS "owns" the physical page being used in the PTE
  - Modify OS to hypervisor call into VMM when updating PTEs
- **Page tables work the same as before, but OS is constrained to only map to the physical pages it owns**
- **Works fine if you can modify the OS (used in Xen paravirtualization)**
- **If you can't…**

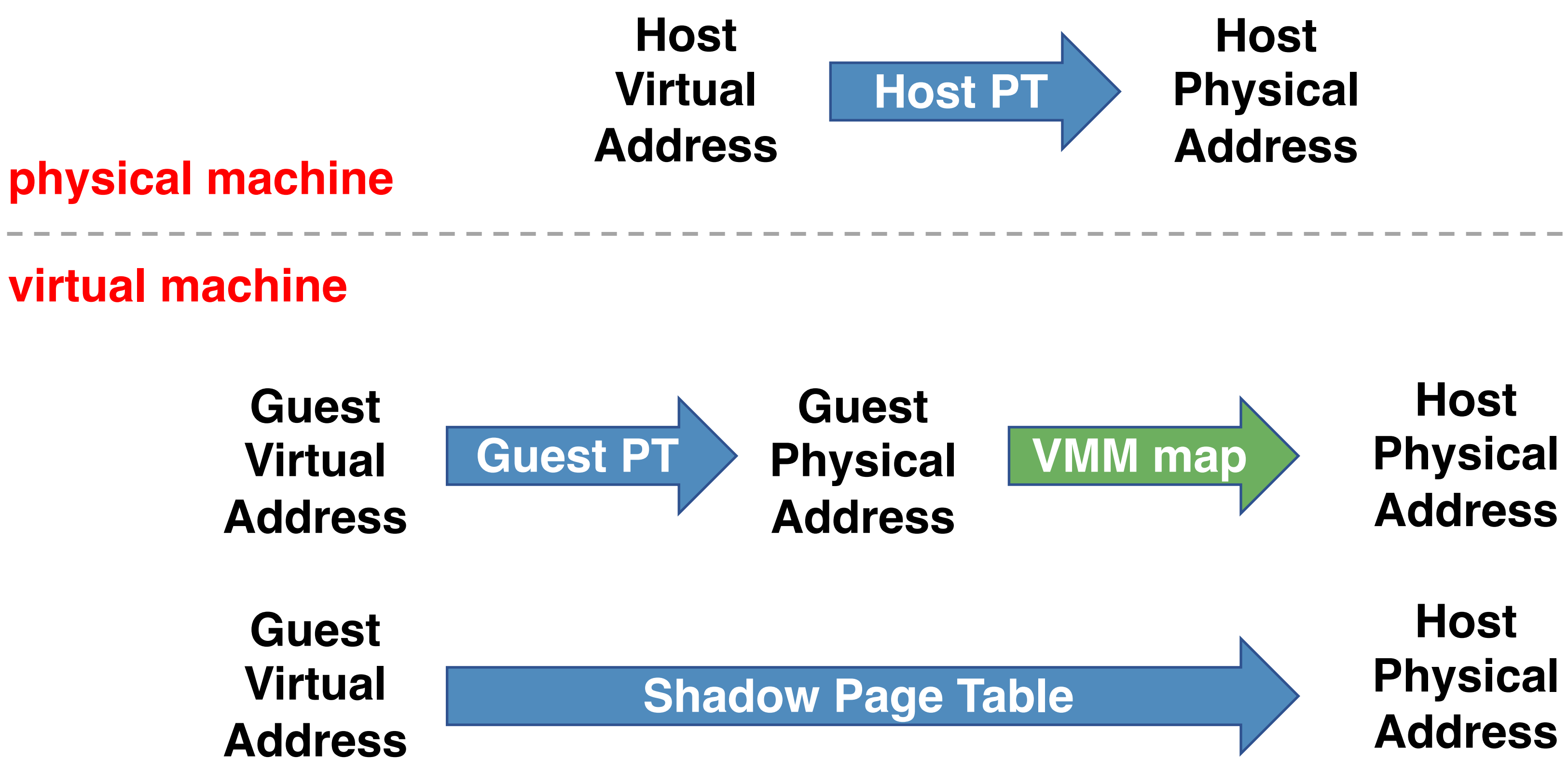
# Second Approach: Level of Indirection

- **Three abstractions of memory**
  - Machine: actual hardware memory
    - 16 GB of DRAM
  - Physical: abstraction of hardware memory managed by OS
    - If a VMM allocates 512 MB to a VM, the OS thinks the computer has 512 MB of contiguous physical memory
    - (Underlying machine memory may be discontiguous)
  - Virtual: virtual address spaces you know and love
    - Standard $2^{32}$ or $2^{64}$ address space
- **Translation: VM's Guest VA → VM's Guest PA → Host PA**
- **In each VM, OS creates and manages page tables for its virtual address spaces without modification**
  - But these page tables are not used by the MMU hardware

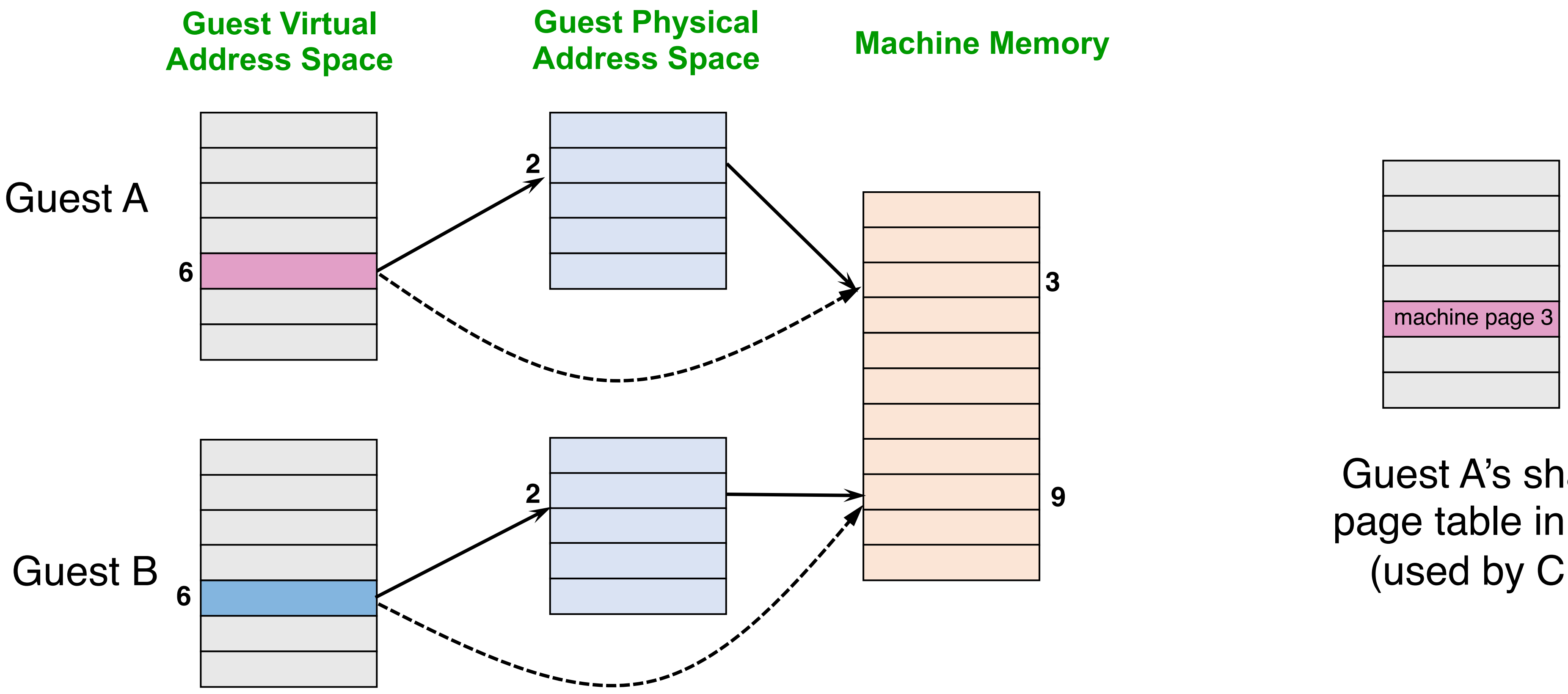
# Shadow Page Tables

- **VMM creates and manages page tables that map virtual pages directly to machine pages**
  - These tables are loaded into the MMU on a context switch
  - VMM page tables are the shadow page tables
- **VMM needs to keep its V→M tables consistent with changes made by OS to its V→ P tables**
  - VMM maps OS page tables as read-only (i.e., write-protected)
  - When OS writes to page tables, trap to VMM
  - VMM applies write to shadow table and OS table, returns
  - Also known as memory tracing
  - Memory-mapped devices must be protected for both read- and write- protected

# Memory Mapping Summary



# Shadow Page Table Example




Guest A's shadow page table in VMM (used by CPU)

# More on Shadow Page Table

- **Shadow page tables are essentially a cache**
- **VMM is responsible for maintaining the consistency**
- **Two kinds of page faults**
  - *True page faults* when page not in VM's guest page table
  - *Hidden page faults* when just misses in shadow page table
- **On a page fault, VMM must:**
  - Lookup guest VPN→guest PPN in guest's page table
  - Determine where guest PPN is in host physical memory
  - Insert guest VPN→host PPN mapping in shadow page table

# Memory Allocation

- **VMMs tend to have simple hardware memory allocation policies**
  - Static: VM gets 512 MB of hardware memory for life
  - No dynamic adjustment based on load
    - OSes not designed to handle changes in physical memory…
  - No swapping to disk
- **More sophistication: Overcommit with balloon driver**
  - Balloon driver runs inside OS to consume hardware pages
    - Steals from virtual memory and file buffer cache (balloon grows)
  - Gives hardware pages to other VMs (those balloons shrink)
- **Identify identical physical pages (e.g., all zeroes)**
  - Map those pages copy-on-write across VMs

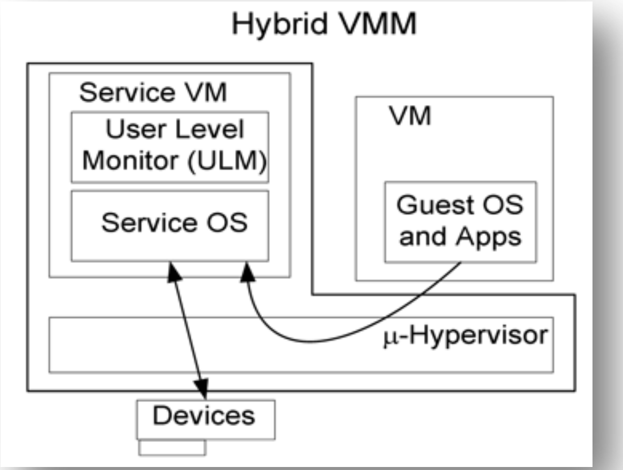
# Virtualizing I/O

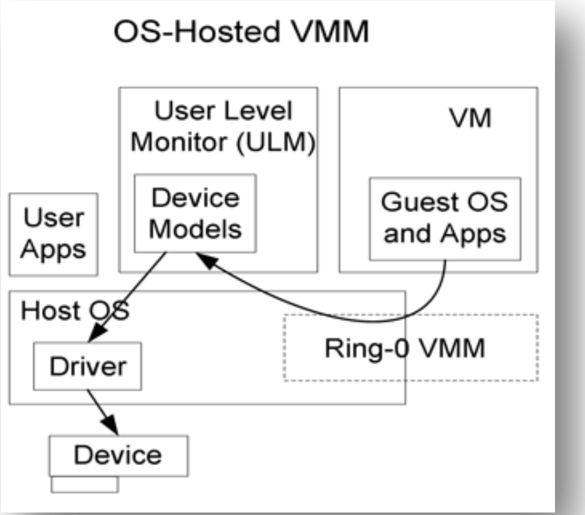
- **OSes can no longer interact directly with I/O devices**
- **Types of communication**
  - Special instruction – in/out
  - Memory-mapped I/O
  - Interrupts
  - DMA
- **Make `in/out` trap into VMM**
- **Use tracing for memory-mapped I/O**
- **Run simulation of I/O device**
  - Interrupt – Tell CPU simulator to generate interrupt
  - DMA – Copy data to/from physical memory of virtual machine

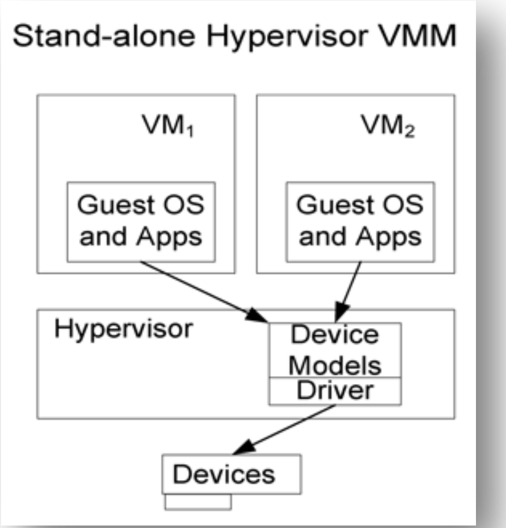
# Virtualizing I/O: Three Models

- **Xen: *modify OS* to use low-level I/O interface (hybrid)**
  - Define generic devices with simple interface
    - Virtual disk, virtual NIC, etc.
  - Ring buffer of control descriptors, pass pages back and forth
  - Handoff to trusted domain running OS with real drivers
- **VMware: VMM supports generic devices (hosted)**
  - E.g., AMD Lance chipset/PCNet Ethernet device
  - Load driver into OS in VM, OS uses it normally
  - Driver knows about VMM, cooperates to pass the buck to a real device driver (e.g., on underlying host OS)
- **VMware ESX Server: drivers run in VMM (hypervisor)**

# Virtualized I/O Models




**Abramson et al., "Intel Virtualization Technology for Directed I/O", Intel Technology Journal, 10(3) 2006**

# Hardware Support

- **Intel and AMD implement virtualization support in their recent x86 chips (Intel VT-x, AMD-V)**
  - Goal is to fully virtualize architecture
  - Transparent trap-and-emulate approach now feasible
  - Echoes hardware support originally implemented by IBM
- **These CPUs support new execution mode: guest mode**
  - This is separate from kernel/user modes in bits 0–1 of `%cs`
  - Less privileged than host mode (where VMM runs)
  - Direct execution of guest OS code, including privileged insts
  - Some sensitive instructions trap in guest mode (e.g., load `%cr3`)
  - Hardware keeps shadow state for many things (e.g., `%eflags`)

# Guest mode

- **Enter and exit guest mode**
  - New instruction `vmenter` enters guest mode, runs VM code
  - When VM traps, CPU executes new `vmexit` instruction
  - Enters VMM, which emulates operation
  - Virtual machine control block (VMCB)
    - Controls what operations trap, records info to handle traps in VMM
  - `vmenter` loads state from hardware-defined 1-KiB VMCB data structure
  - On EXIT, hardware saves state back to VMCB

# Guest State Saved in VMCB

- **Saved guest state**
  - Full segment registers (i.e., base, lim, attr, not just selectors)
  - Full GDTR, LDTR, IDTR, TR
  - Guest %cr3, %cr2, and other cr/dr registers
  - Guest %eip and %eflags
  - Guest %rax register
- **Entering/exiting VMM more expensive than syscall**
  - Have to save and restore large VM-state structure

# Hardware Support (2)

- **Memory**
  - Intel extended page tables (EPT), AMD nested page tables (NPT)
  - Original page tables map virtual to (guest) physical pages
    - Managed by OS in VM, backwards-compatible
  - New tables map physical to machine pages, managed by VMM
  - No need to trap to VMM when OS updates its page tables
  - Tagged TLB w/ virtual process identifiers (VPIDs)
    - Tag VMs with VPID, no need to flush TLB on VM/VMM switch
- **I/O**
  - Constrain DMA operations only to page owned by specific VM
  - AMD Device Exclusion Vector (DEV) (c.f. Xen memory paravirtualization)
  - Intel VT-d: IOMMU – address translation support for DMA

# Summary

- **VMMs multiplex virtual machines on hardware**
  - Export the hardware interface
  - Run OSes in VMs, apps in OSes unmodified
  - Run different versions, kinds of OSes simultaneously
- **Implementing VMMs**
  - Virtualize CPU, Memory, I/O
- **Lesson: Never underestimate the power of indirection**