

# **CSE 120**

# **Principles of Operating Systems**

**Spring 2018**

**Lecture 16: Virtual Machine Monitors**

Geoffrey M. Voelker


# Virtual Machine Monitors

---



# Virtual Machine Monitors

---

- Virtual Machine Monitors (VMMs) are a  hot topic in industry and academia
  - ♦ Industry commitment
    - » Software: VMware, Xen, Microsoft Virtual PC
    - » Hardware: Intel VT, AMD-V
      - If Intel and AMD add it to their chips, you know it's serious...
  - ♦ Academia: lots of VMM-based projects and papers
- An old idea, actually: developed by IBM in 60s and 70s
- Today
  - ♦ What is it, what problems have to be solved, how to solve them
  - ♦ Survey some virtualization systems

# What is a VMM?

---

- We have seen that an OS already virtualizes
  - ♦ Syscalls, processes, virtual memory, file system, sockets, etc.
  - ♦ Applications program to this interface
- A **VMM** virtualizes an entire physical machine
  - ♦ Interface supported is the hardware
    - » In contrast, OS defines a higher-level interface
  - ♦ VMM provides the **illusion** that software has full control over the hardware (of course, VMM is in control)
  - ♦ VMM “applications” run in **virtual machines** (c.f., OS processes)
- Implications
  - ♦ You can boot an operating system in a virtual machine
  - ♦ Run multiple instances of an OS on same physical machine
  - ♦ Run different OSes simultaneously on the same machine
    - » Linux on Windows, Windows on Mac, etc.

# Why VMMs?

---

- Resource utilization
  - ♦ Machines today are powerful, want to multiplex their hardware
    - » Cloud hosting can divvy up a physical machine to customers
  - ♦ Can migrate VMs from one machine to another without shutdown
- Software use and development
  - ♦ Can run multiple OSes simultaneously
    - » No need to dual boot
  - ♦ Can do system (e.g., OS) development at user-level
- Many other cool applications
  - ♦ Debugging, emulation, security, speculation, fault tolerance...
- Common theme is manipulating applications/services at the granularity of a machine
  - ♦ Specific version of OS, libraries, applications, etc., as package

# VMM 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, much better today)

# Rough VMM Model

---

- VMM runs with privilege
  - ♦ OS in VM runs at “lesser” privilege (think user-level)
  - ♦ VMM multiplexes resources among VMs
- Want to run OS code in a VM directly on CPU
  - ♦ Think in terms of making the OS a user-level process
  - ♦ What OS code can run directly, what will cause problems?
- Ideally, want privileged instructions to trap
  - ♦ Exception vectors to VMM, it emulates operation, returns
  - ♦ Nothing modified, running unprivileged is transparent
  - ♦ Known as trap-and-emulate
- Unfortunately on architectures like x86, not so easy

# Virtualizing the x86

---

- Ease of virtualization influenced by the architecture
  - ♦ x86 is perhaps the last architecture you would choose
  - ♦ But it's what everyone uses, so...that's what we deal with
- Issues
  - ♦ Unvirtualizable events
    - » `popf` does not trap when it cannot modify system flags
  - ♦ Hardware-managed TLB
    - » VMM cannot easily interpose on a TLB miss (more in a bit)
  - ♦ Untagged TLB
    - » Have to flush on context switches (just a performance issue)
- Why Intel and AMD have added virtualization support

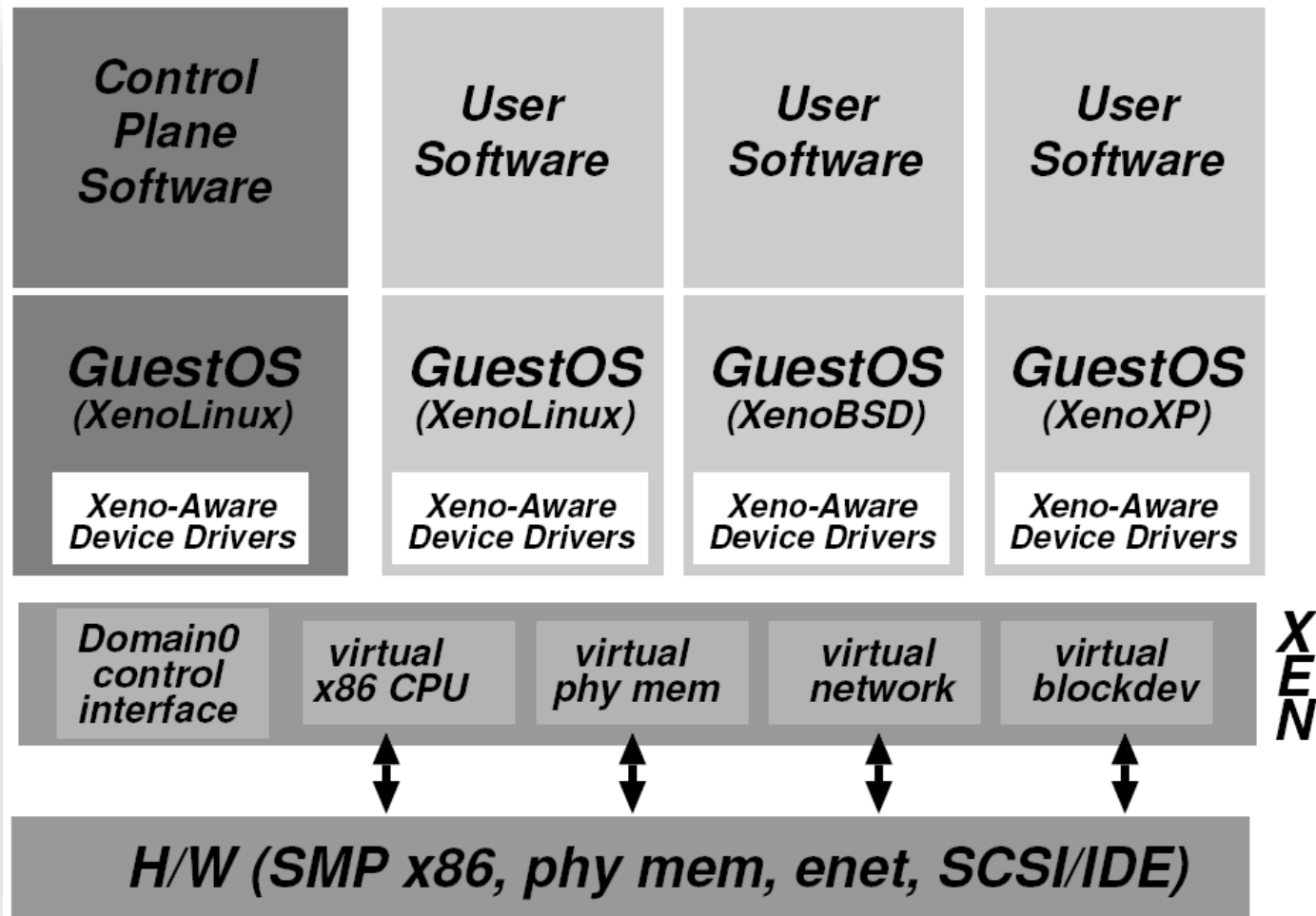


# Xen

---

- 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

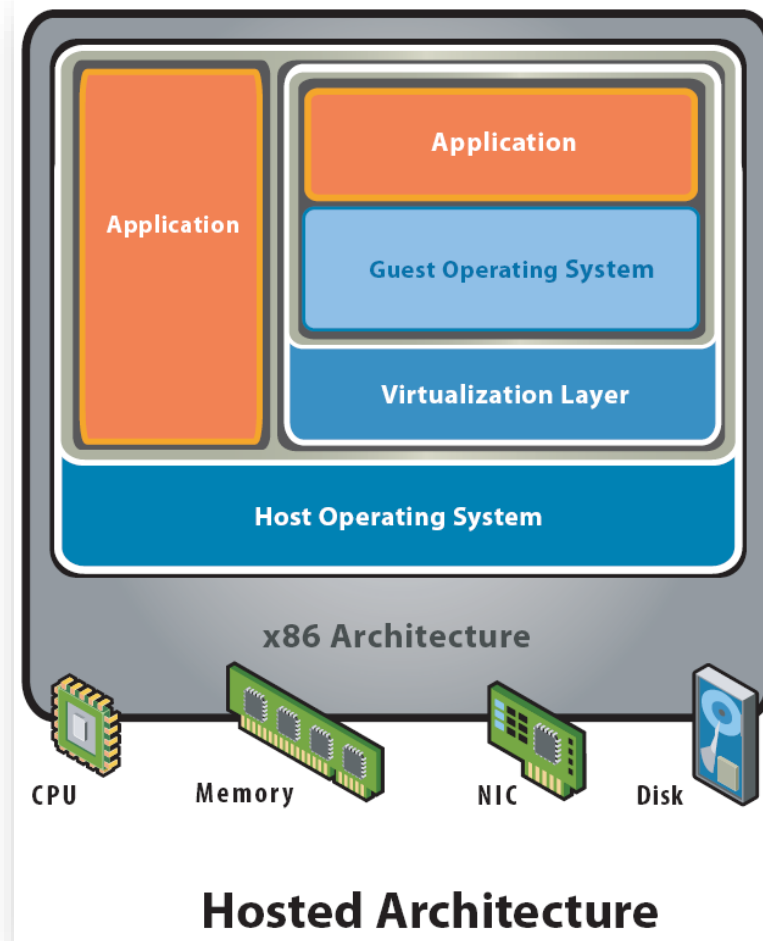


# VMware

---

- 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
    - » Rewrite privileged instructions with emulation code (may trap)
  - ♦ CPU only executes translated code
  - ♦ 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

# Virtualizing Privileged Insts

---

- OSes can no longer successfully execute privileged instructions
  - ♦ Virtual memory registers, interrupts, I/O, halt, etc.
- For those instructions that cause an exception
  - ♦ Trap to VMM, take care of business, return to OS in VM
- For those that do not...
  - ♦ **Xen**: modify OS to hypervisor call into VMM
  - ♦ **VMware**: rewrite OS instructions to emulate or call into VMM
  - ♦ **H/W support**: add new CPU mode, instructions to support trap and emulate

# Virtualizing the CPU

---

- VMM needs to multiplex VMs on CPU
- How? Just as you would expect
  - ♦ Timeslice the VMs
  - ♦ Each VM will timeslice its OS/applications during its quantum
- Typically relatively simple scheduler
  - ♦ Round robin, work-conserving (give unused quantum to other VMs)

# Virtualizing Events

---

- VMM receives interrupts, exceptions
- Needs to vector to appropriate VM
  - ♦ **Xen**: modify OS to use virtual interrupt register, event queue
  - ♦ **VMware**: craft appropriate handler invocation, emulate event registers

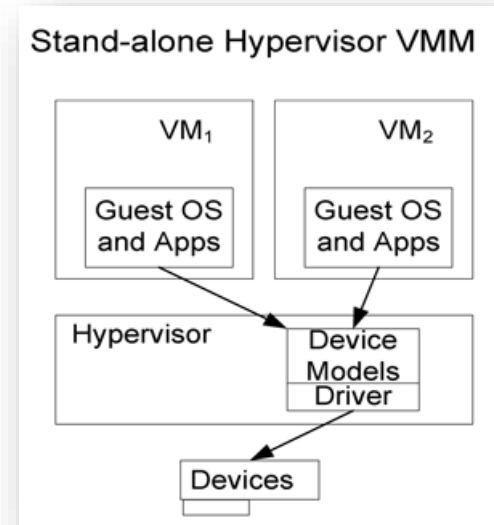
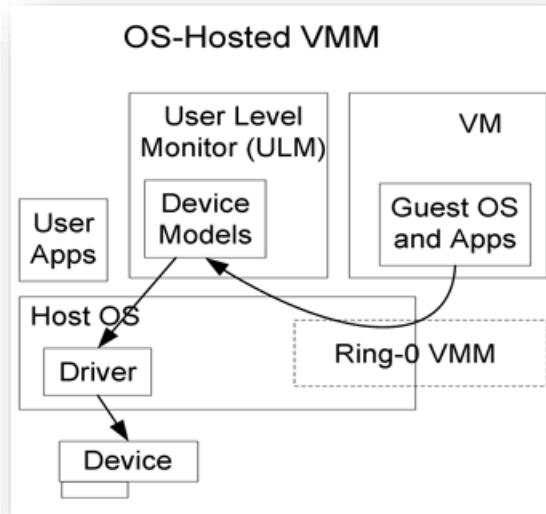
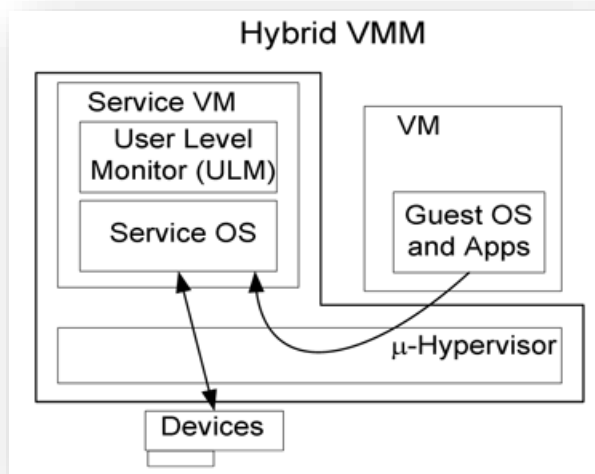


# Virtualizing I/O

---

- OSes can no longer interact directly with I/O devices
- **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

# 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

# Xen Paravirtualization

---

- Xen uses the page tables that an OS creates
  - ♦ These page tables are used directly by hardware MMU
- Xen validates all updates to page tables by OS
  - ♦ OS can read page tables without modification
  - ♦ But Xen 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 Xen when updating PTEs
    - » Batch updates to reduce overhead
- 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. If you can't...

# Shadow Page Tables

---

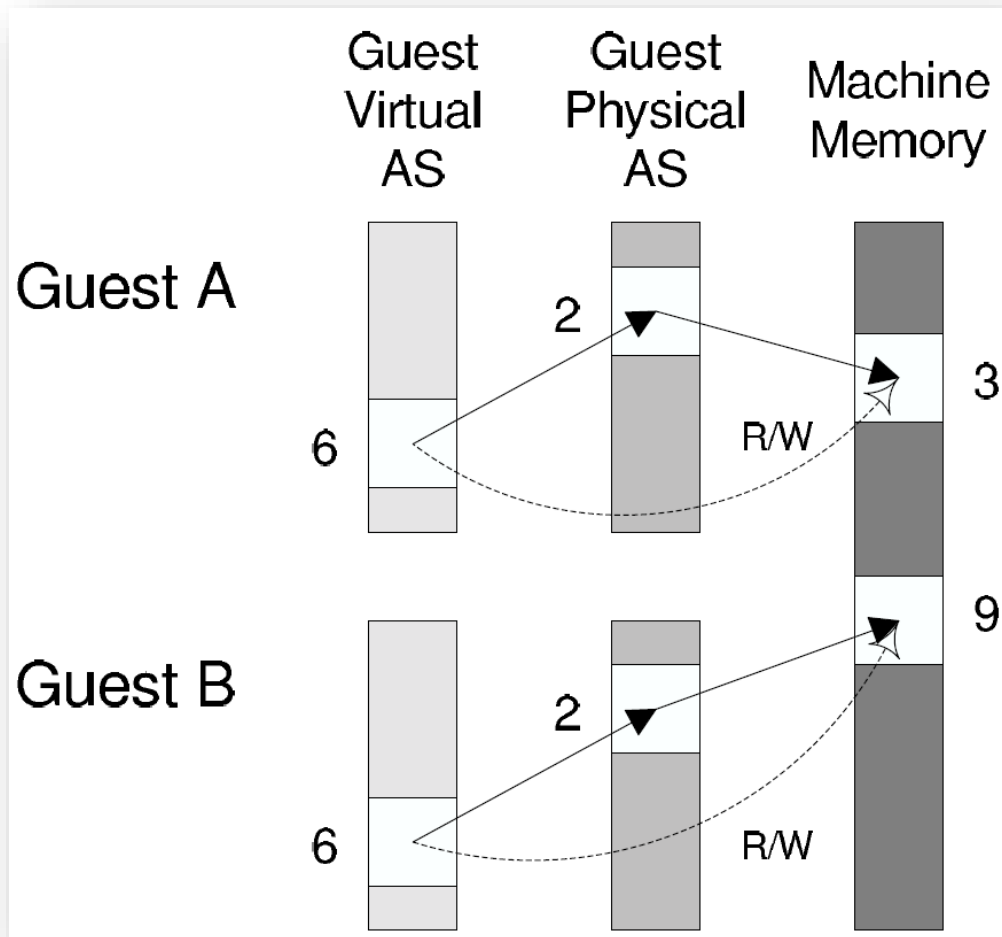
- Three abstractions of memory
  - ♦ Machine: actual hardware memory
    - » 16 GB of DRAM
  - ♦ Physical: abstraction of hardware memory managed by OS
    - » If a VMM allocates 512 MB to a VM, the OS thinks the computer has 512 MB of contiguous physical memory
    - » (Underlying machine memory may be discontinuous)
  - ♦ Virtual: virtual address spaces you know and love
    - » Standard  $2^{32}$  or  $2^{64}$  address space
- 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 (2)

---

- 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  $V \rightarrow M$  tables consistent with changes made by OS to its  $V \rightarrow P$  tables
  - ♦ VMM maps OS page tables as read only
  - ♦ When OS writes to page tables, trap to VMM
  - ♦ VMM applies write to shadow table and OS table, returns
  - ♦ Also known as [memory tracing](#)
  - ♦ Again, more overhead...

# Shadow Page Tables (3)



# 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



# 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
- Execution model
  - ♦ New execution mode: guest mode
    - » Direct execution of guest OS code, including privileged insts
  - ♦ Virtual machine control block (VMCB)
    - » Controls what operations trap, records info to handle traps in VMM
  - ♦ New instruction `vmenter` enters guest mode, runs VM code
  - ♦ When VM traps, CPU executes new `vmexit` instruction
  - ♦ Enters VMM, which emulates operation

# 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
    - » No need to trap to VMM when OS updates its page tables
  - ♦ New tables map physical to machine pages
    - » Managed by VMM
  - ♦ 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 DEV: exclude pages (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
- Lesson: Never underestimate the power of indirection