

# Virtualization

Thierry Sans

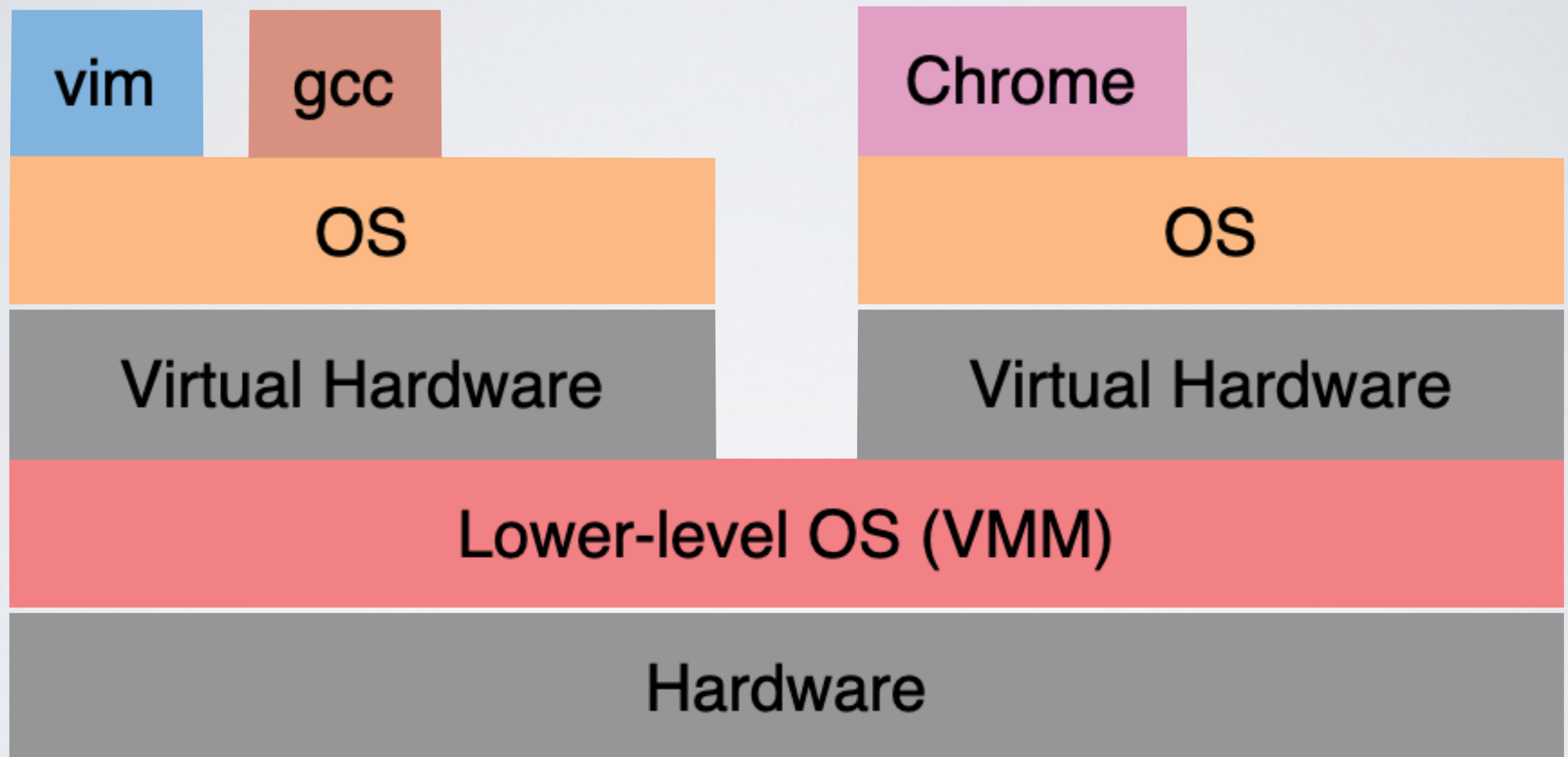
## (recap) What is an OS?



OS is software between applications and hardware

- Abstracts hardware to makes applications portable
- Makes finite resources (memory, # CPU cores) appear much larger and fully dedicated to running one application
- Protects processes and users from one another

What if...



➡ The process abstraction looked just like hardware?

# How do process abstraction & hardware differ

## Process

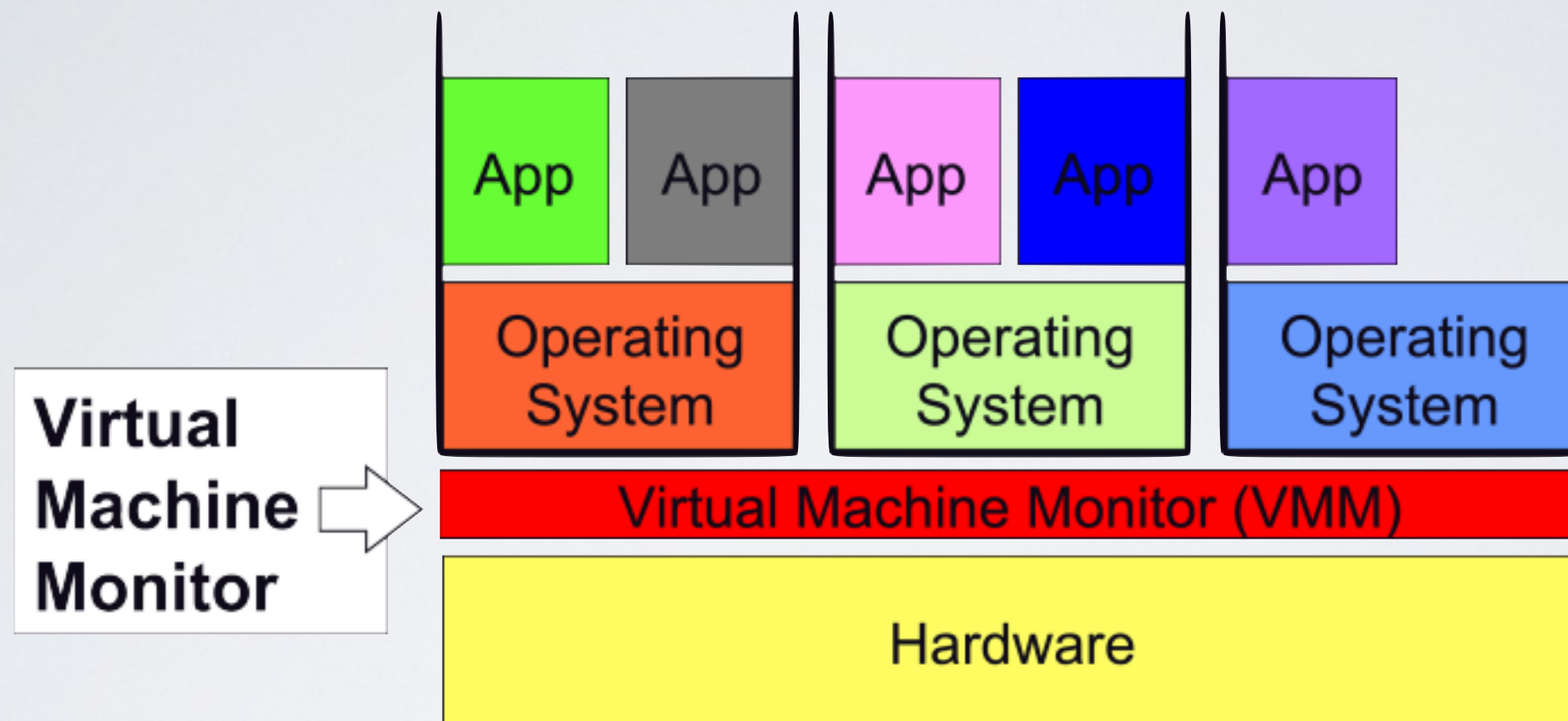
- Non-privileged registers and instructions
- Virtual memory
- Errors and signals
- File systems, directories, files, raw devices

## 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



# VMM - 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

## ➔ **The Virtual Machine Monitor (a.k.a Hypervisor)**

runs multiple OSes simultaneously on the same physical machine

# Motivations

# Old idea from the 70s

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 80s and 90s

- Hardware got cheap
- The era of the personal computer (Windows)

➡ Revived by the Disco work  
led by *Mendel Rosenblum*, later lead to the foundation of VMware

➡ Another important work *Xen*

# VMMs 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, VirtualBox, ...*  
Hardware : *Intel VT, AMD-V*
- Academia - lots of related projects and papers





# Why would you do such a crazy thing?

- **Software compatibility**

VMMs can run pretty much all software (since they can pretty much all OSes)

- **Resource utilization**

Machines today are powerful, want to multiplex their hardware

- **Isolation**

Seemingly total data isolation between virtual machines

- **Encapsulation**

Virtual machines are not tied to physical machines

- **Many other cool applications**

Debugging, emulation, security, speculation, fault tolerance...

# OS backwards compatibility

Backward compatibility is bane of new OSes as it requires huge efforts to innovate but not break

➡ Security considerations may make it impossible in practice

# 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**

- Safety - printer server failure doesn't take down Exchange server
- Security - compromise of one VM cannot get at data of others

## **Resource management**

- Provide service-level agreements

## **Heterogeneous environments**

- Linux, FreeBSD, Windows, etc.

# Implementation



# 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 (that needs to be minimized)

# 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) approaches will be similar to what we do with OSes  
simpler in functionality, though (VMM much smaller than OS)
- (and no) but implements a different abstraction  
hardware interface vs. OS interface

# Approach 1 : complete machine simulation

➔ Simplest VMM approach, used by *bochs*

Build a simulation of all the hardware

- CPU – a loop that fetches each instruction, decodes it, simulates its effect on the machine state (no direct execution)
- Memory – physical memory is just an array, simulate the MMU on all memory accesses
- I/O – simulate I/O devices, programmed I/O, DMA, interrupts

⦿ Too slow!

- CPU/Memory – **100x slowdown**
- I/O Device – 2x slowdown

➔ Need faster ways of emulating CPU/MMU

# Approach 2 : virtualizing the CPU/MMU

- ➡ Observations - most instructions are the same regardless of processor privileged level e.g. `incl %eax`

Why not just give instructions to CPU to execute?

- Problem - safety  
How to prevent privilege instructions from interfering with hypervisor and other OSes?
- Solution - use protection mechanisms already in CPU

- ➡ "Trap and emulate" approach

- run virtual machine's OS directly on CPU in unprivileged user mode
- privileged instructions trap into monitor and run simulator on instruction



# Virtualizing interrupts

OS assumes to be in control of interrupts via the interrupt table

So what happens when an interrupt or trap occurs in a virtual environment?

- ➡ The VMM handles the interrupt (in kernel mode) using the "virtual" interrupt handler table of the running OS
- ✓ Some interrupt can be shadowed

# Virtualizing memory

OS assumes to be in full control over memory via the page table

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 way - direct mapping

- ➔ VMM uses the page tables that a guest OS creates (direct mapping by MMU)

VMM validates all updates to page tables by guest OS

- OS can read page tables without modification
  - but VMM needs to check all page table entry writes to ensure that the virtual-to-physical mapping is valid
- Requires to modify the OS to patch updates to the page table (used in *Xen* paravirtualization)

# Another way - 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 discontinuous)
- Virtual - virtual address spaces (similar to virtual memory)  
standard  $2^{32}$  or  $2^{64}$  address space

➔ Translation - from **VM's Guest VA** to **VM's Guest PA** to **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

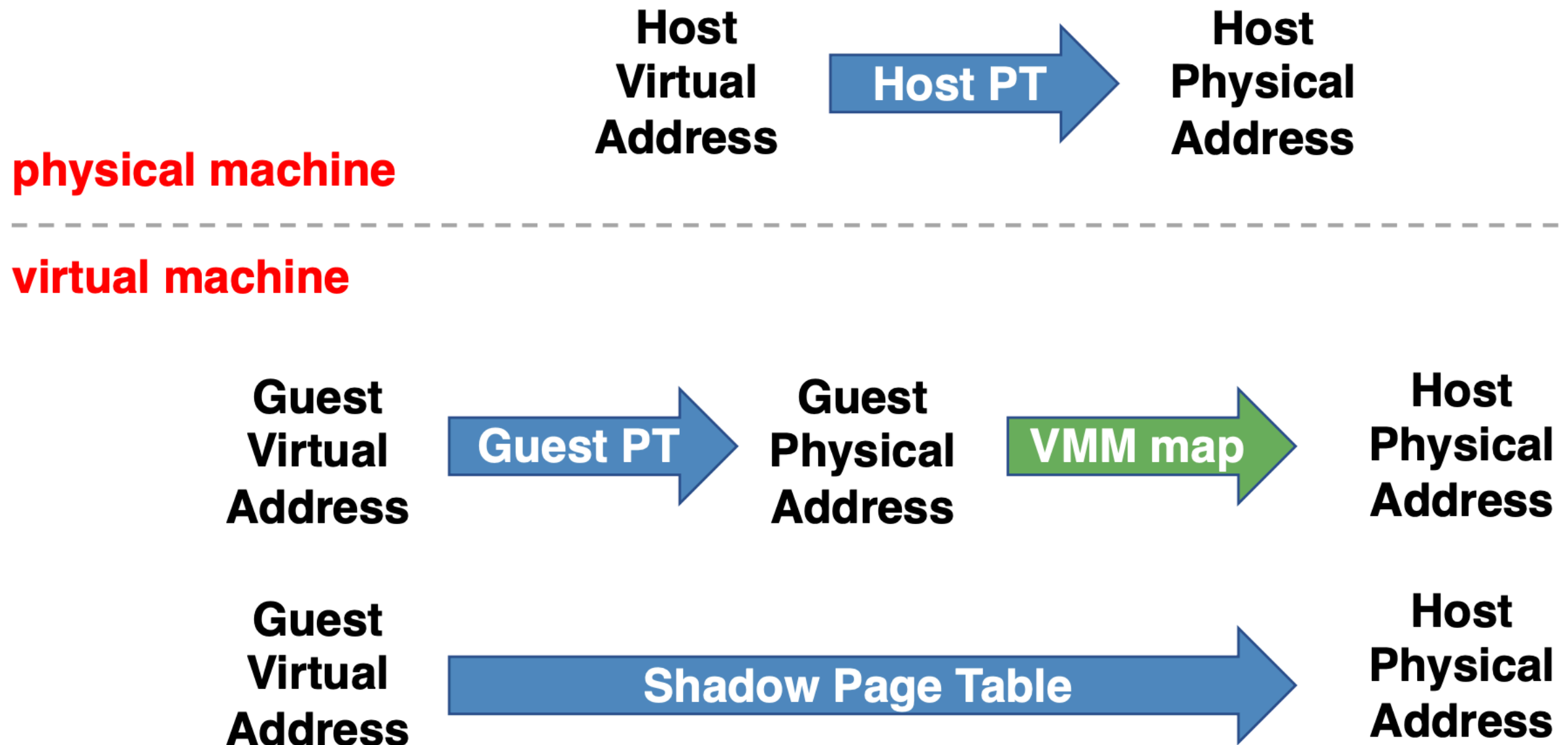
VMM creates and manages page tables that map virtual pages directly to machine page these tables are loaded into the MMU on a context switch

## → **VMM page tables are the shadow page tables**

VMM needs to keep its virtual to machine tables consistent with changes made by OS to its virtual to physical tables

- VMM maps OS page tables as read-only (i.e., write-protected)
- When OS writes to page tables, trap to VMM
- Memory tracing :VMM applies write to shadow table and OS table and returns
- Memory-mapped devices must be protected for both read- and write- protected

# Memory mapping summary



# 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)

# 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

1. Make in/out trap into VMM and use tracing for memory-mapped I/O

2. Run simulation of I/O device

- Interrupt – tell CPU simulator to generate interrupt
- DMA – copy data to/from physical memory of virtual machine



# 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 some privileged instructions
- 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

# Hardware Support

## 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
- 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

# Virtualizing I/O - Three Models

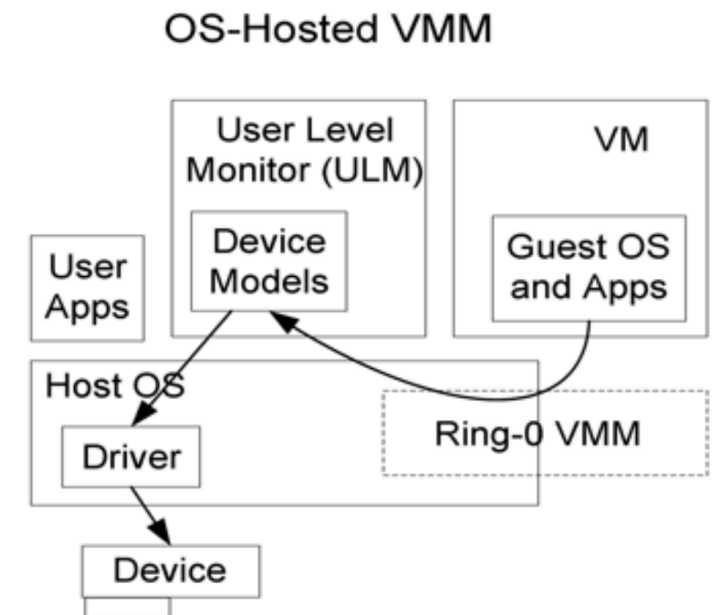
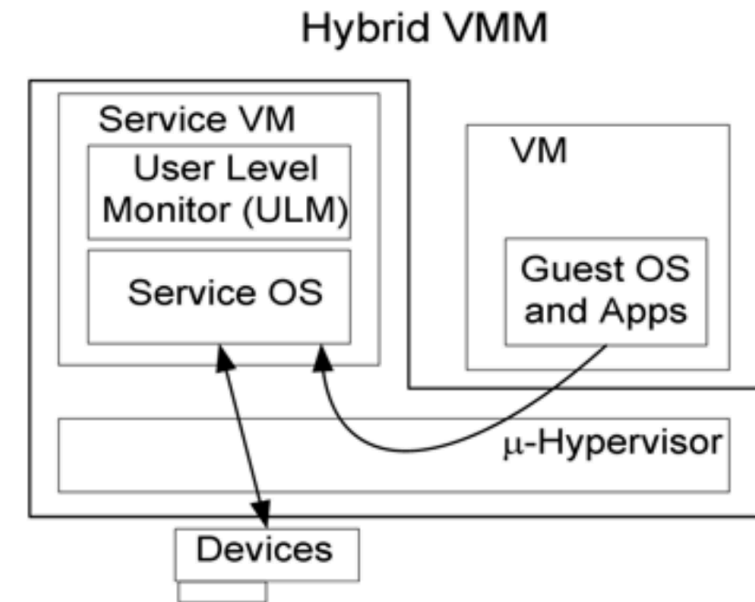
*Xen* : modify OS to use low-level I/O interface (hybrid)

- 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

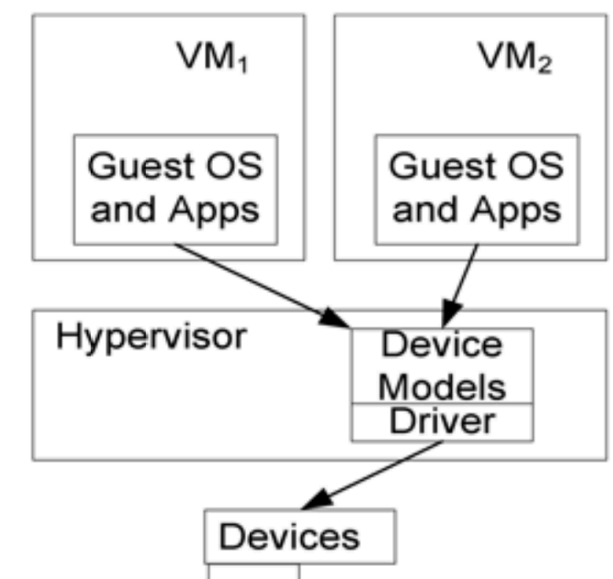
*VMware* :VMM supports generic devices (hosted)

- 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)

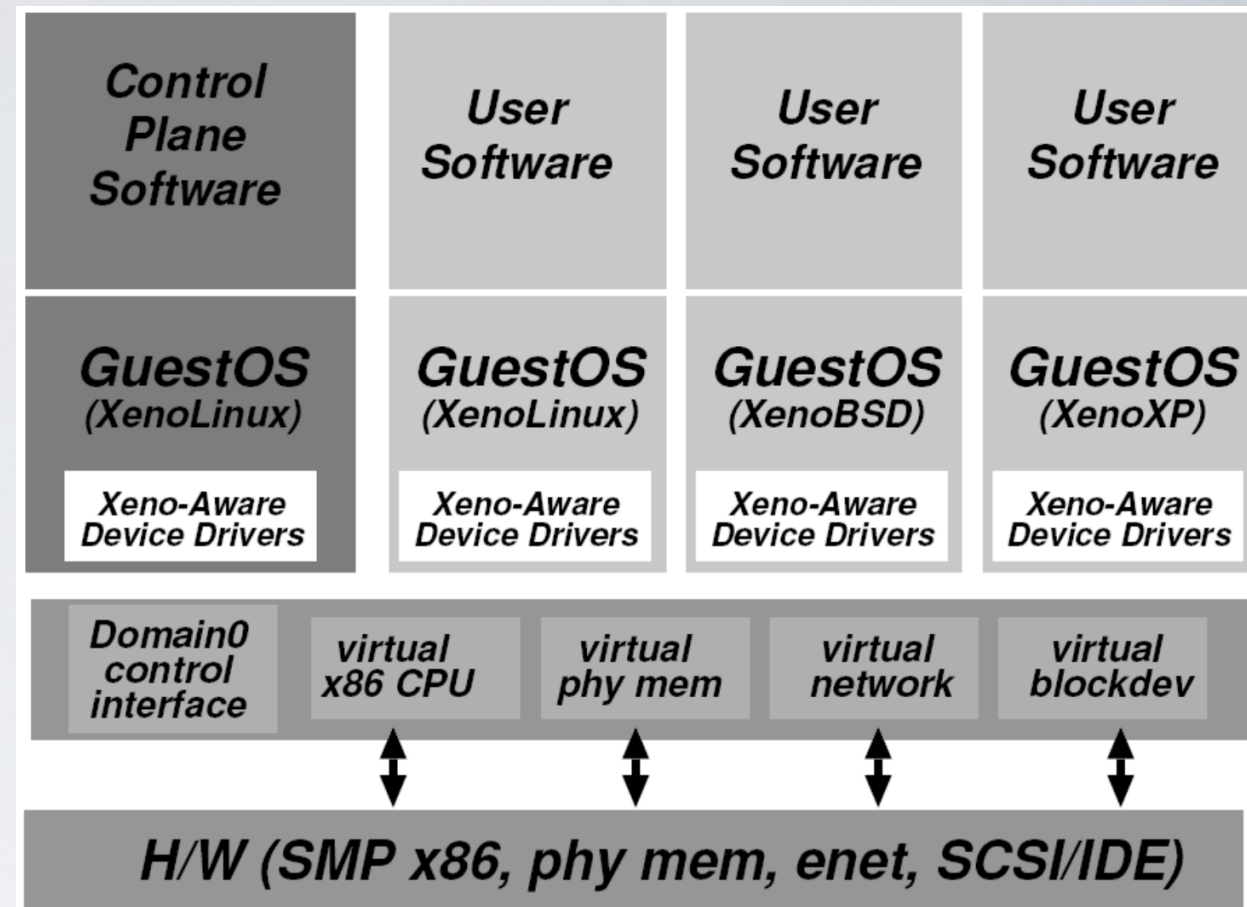


**Stand-alone Hypervisor VMM**





# VMM case study I - 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



# VMM case study 2 - VMware

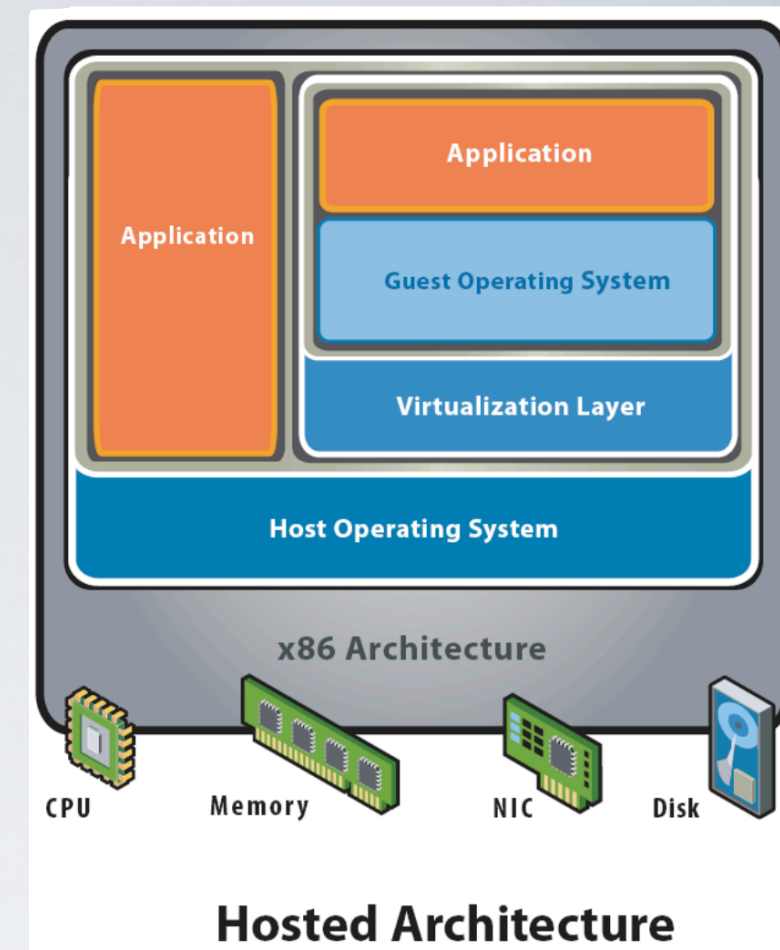
VMware uses software virtualization

- 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 to IR code to safe subset of x86
- Incurs overhead, but can be well-tuned (small % hit)

✓ 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



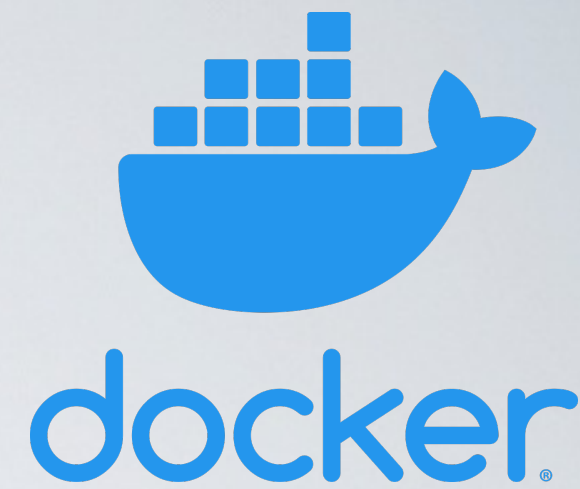
# Summary

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 means virtualizing CPU, Memory, I/O

➡ Lesson: never underestimate the power of indirection



## Another trend - containers

- Containers are not virtual OSes

- ➔ There are user applications running over the same OS kernel

Learn more about it

<https://www.codementor.io/blog/docker-technology-5x1kilcbow>

# Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*