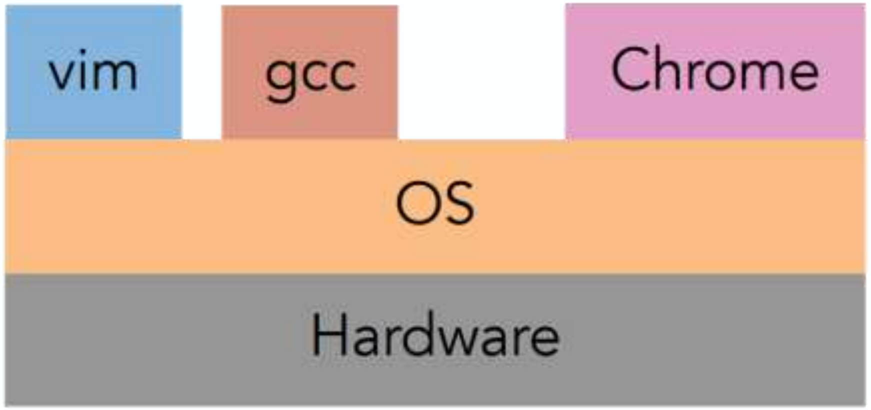
CS-446/646

Virtual Machine Monitors

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Virtual Machines

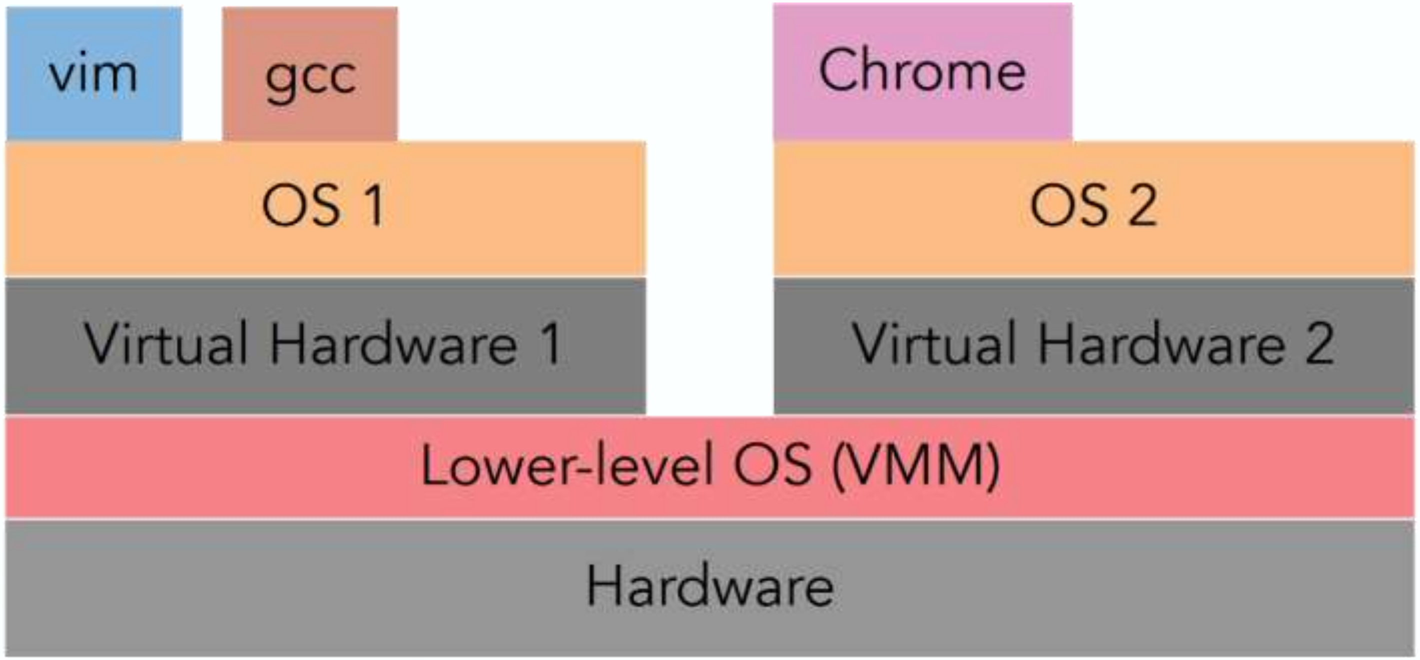
Remember: **What is an OS**



- OS is *Middleware* – Software between Applications and Hardware
  - Abstracts Hardware to makes Applications portable
  - Makes finite resources (*Memory*, # of CPU cores) appear much larger
  - Protects *Processes* and Users from one another

Virtual Machines

What if…



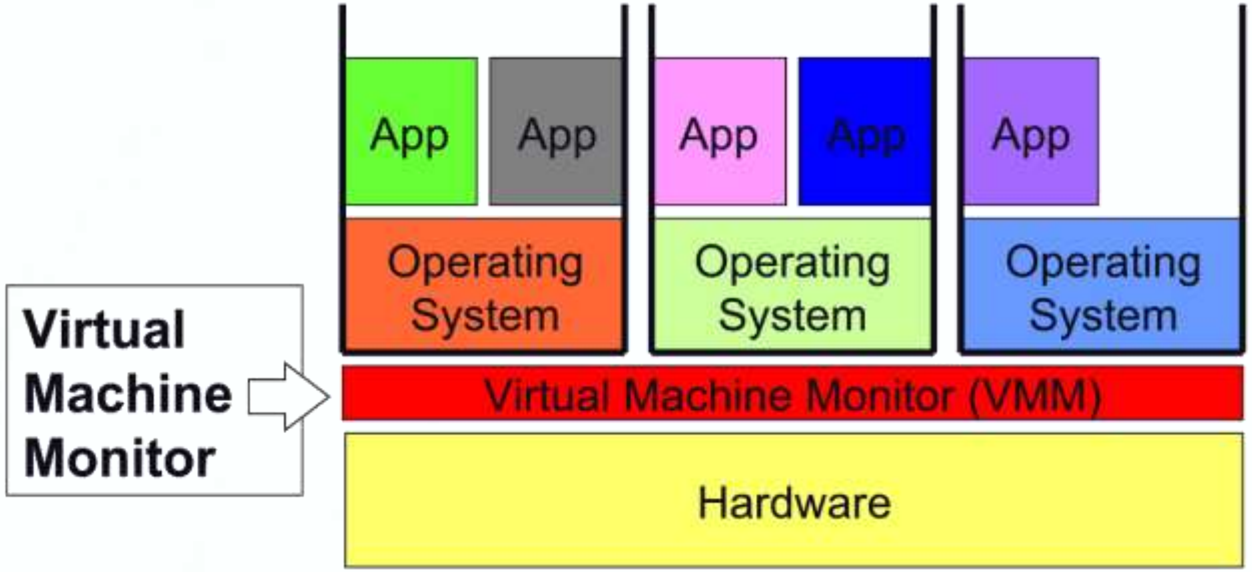
- The *Process* Abstraction looked just like Hardware?

How do *Process* Abstraction & Hardware differ?

- ***Process***
  - Non-*Privileged Registers* and *Instructions*
  - *Virtual Memory*
  - *Errors*, *Signals*
  - *Filesystem*, *Directories*, *Files*, Raw-Data *Devices*
- ***Hardware***
  - All *Registers* & *Instructions*
  - *Both Virtual* and *Physical memory*, MMU functions, TLB/*Page Tables*, etc.
  - *Trap* Architecture, *Interrupts*
  - I/O *Devices* accessed using Programmed I/O, *Direct Memory Access (DMA)*, *Interrupts*

Virtual Machine Monitor (VMM)

- Thin layer of Software that *Virtualizes* the Hardware
  - Exports a *Virtual Machine* (VM) 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 same *Physical* Machine



Virtual Machine Monitor (VMM)

- Old idea from the 1960s
  - See [Goldberg] from 1974
- IBM VM/370 – A *Virtual Machine Monitor* for the 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
  - Just put a Windows machine on every Desktop
- Revived by the *Disco* [SOSP '97] work
  - Led by Mendel Rosenblum, later lead to the foundation of *VMware*
- Another important work: *Xen* [SOSP '03]

Virtual Machines

Virtual Machine Monitor **(VMM)**

- Today VMs are used everywhere
  - Popularized by Cloud Computing
  - Used to solve different problems
- *Virtual Machine Monitors* are a hot topic in industry and academia
- Industry commitment
  - Software: VMware, Xen,…
  - Hardware: Intel VT, AMD-V
    - Integration of support in CPUs means it's serious…
  - Academia: lots of related projects and papers

Virtual Machine Monitor (VMM) Benefits

- Software compatibility
  - *Virtual Machine Monitors* 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…

Virtual Machine Monitor (VMM) Applications

- *Backwards Compatibility* is bane of new OSes
  - Huge effort require to innovate but not break
- *Security* considerations may make it impossible
  - Choice: Close *Security* hole and break Apps, or let known *Security* flaw just be
- *Example*: Windows XP at End-of-Life
  - 4.59% of machines were still running 17-year-old Windows XP back in 2018
  - Eventually Hardware running WinXP would die
  - What to do with legacy WinXP Applications?
  - Not all Applications will be able to run on later Windows
  - Given the number of WinXP Applications, practically any OS change will break something
- Solution: Use a VMM to run both WinXP and Win10
  - Obvious for OS migration as well: Windows → Linux

Virtual Machine Monitor (VMM) Applications

- Logical Partitioning of *Servers*
  - Run multiple Servers on same box (e.g. Amazon EC2)
    - Modern CPUs more powerful than most services need
    - *Virtual Machine Monitors* let you give away less than one Machine
    - "*Server* Consolidation" trend: N Machines → 1 real Machine
    - 0.10U Rack-Space Machine – Less power, cooling, space, 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
  - Side-by-side Linux, FreeBSD, Windows, etc.

Note:
In practice not so simple because of *Side-Channel Attacks*
[Ristenpart] [Meltdown/Spectre]

Implementing Virtual Machines

Requirements

- *Fidelity*
  - OSes and Applications should work the same without modification
    - (although we may modify the OS a bit)
- *Isolation*
  - *Virtual Machine Monitor* protects *Resources* and VMs from each other
- *Performance*
  - *Virtual Machine Monitor* is another layer of Software…and therefore adds overhead
    - As with OS, want to minimize this overhead
  - Example: *VMware* (early):
    - CPU-intensive Apps : 2-10% overhead
    - I/O-intensive Apps : 25-60% overhead (much better today)

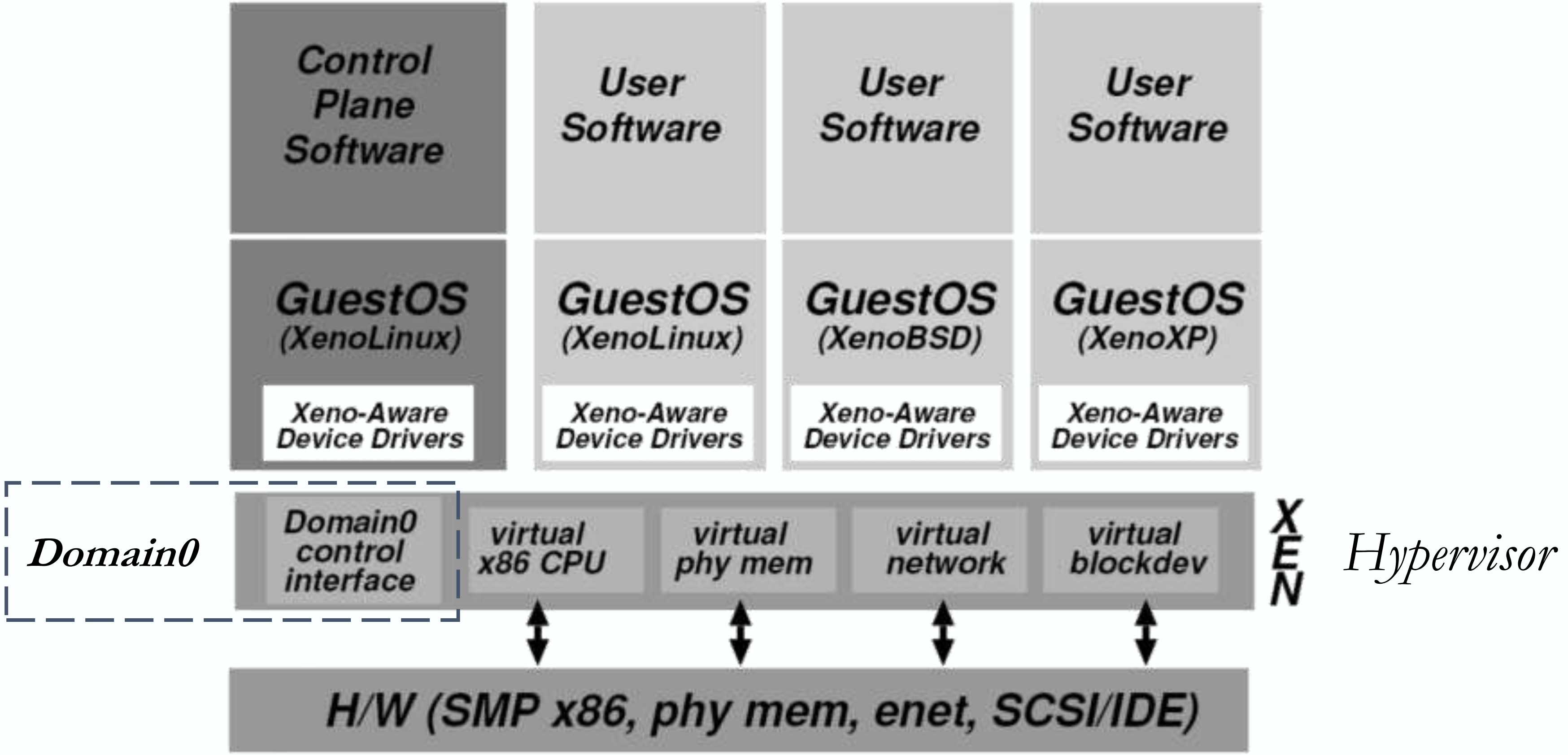
Implementing Virtual Machines

Virtual Machine Monitor Case Study 1: *Xen*

- Earlier versions and Open-Source version use ***Paravirtualization***
  - Fancy word for "**modify** & recompile the OS" to have Guest OS Instructions make "*Hypercalls*" (communicate directly with *Hypervisor* by providing Interface that minimizes overhead; makes it seem as if natively running on Host Hardware)
- *Xen* ***Hypervisor*** (*Virtual Machine Monitor*) implements this Interface
  - *Virtual Machine Monitor* runs at *Privilege*, VMs (*Domains*) run *Unprivileged*
  - Also, the *Trusted* OS (Linux) runs in own *Domain* (*Domain0*)
    - Manage System, operate *Devices*, etc.
- Most recent version of *Xen* (non-Open-Source) does not require OS **modifications**
  - "*Hardware-Assisted Virtualization*"
    - Thanks to Intel/AMD Hardware support
- Commercialized via *XenSource*, but also Open-Source

Virtual Machine Monitor Case Study 1: *Xen*

- Architecture



Implementing Virtual Machines

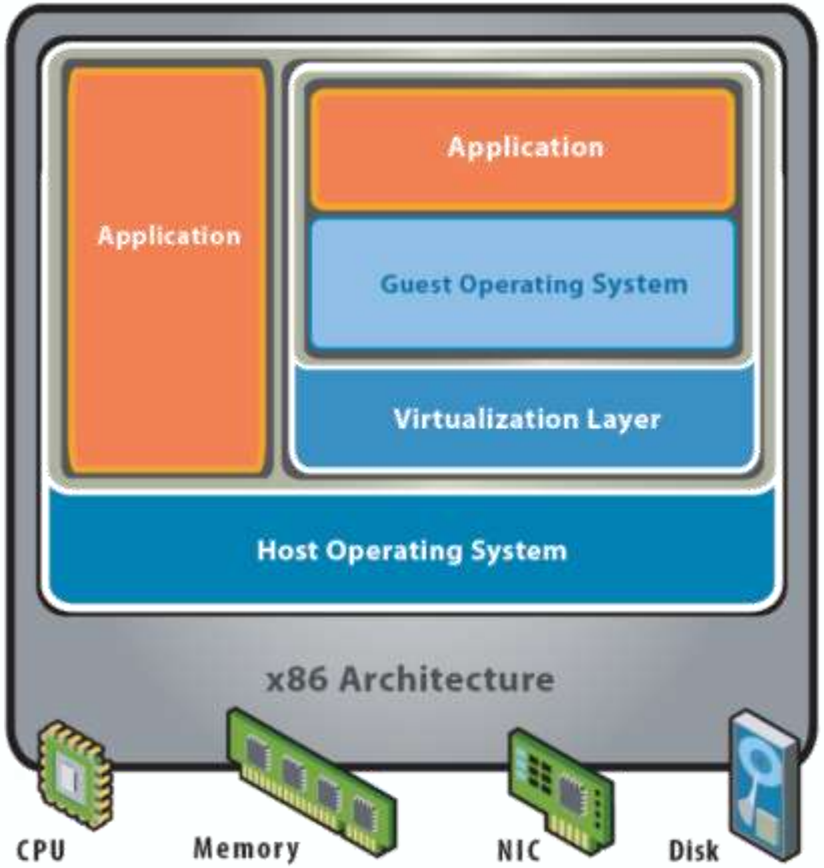
Virtual Machine Monitor* Case Study 2: *VMware

- *VMware Workstation* : Uses ***Hosted*** model
  - *Virtual Machine Monitor* runs *Unprivileged*, installed on Host OS (+ Drivers)
  - Fully relies upon Host OS for all *Device* functionality
- *VMware ESXi* : Uses ***Hypervisor*** model
  - Similar to *Xen*, but no Guest *Domain*/OS
- *VMware* uses ***Software Virtualization / Binary Translation***
  - *Dynamic Binary Rewriting* translates code executed in VM **on-the-fly**
    - Full binary x86 → *Intermediate Representation (IR)* code → Safe subset of x86
    - Software automatically modified on-the-fly by replacing original *Instructions* that "pierce the VM" with a different, VM-safe sequence of *Instructions*
  - Incurs overhead, but can be well-tuned (to minimize performance hit)

Implementing Virtual Machines

Virtual Machine Monitor Case Study 2: *VMware*

- ***Hosted*** Architecture



Implementing Virtual Machines

What needs to be *Virtualized*?

- CPU
- *Events* (*Exceptions* and *Interrupts*)
- *Memory*
- I/O *Devices*

- Isn't this just duplicating OS functionality in a *Virtual Machine Monitor*?
  - Yes and No
  - Approaches will be similar to what we do with OSes
    - Simpler in functionality, though (VMM much smaller than OS)
  - But *Virtual Machine Monitor* implements a different Abstraction
    - Hardware Interface –vs– OS Interface

Implementing Virtual Machines

Approach 1: *Complete Machine Emulation*

- Simplest *Virtual Machine Monitor* approach, used by **`bochs`**
- Build an *Emulation* of all the Hardware
  - CPU : A loop that fetches each *Instruction*, decodes it, *Emulates* its effect on Machine State
  - Memory : *Physical Memory* is just an array, *Emulate* the MMU on all *Memory* accesses
  - I/O : *Emulate* I/O *Devices*, Programmed I/O, DMA, *Interrupts*
- Problem: Too slow!
  - CPU/*Memory* – 100x CPU/MMU *Emulation*
  - I/O Device – Worse-than 2× slowdown.
  - 100× slowdown makes it not too useful
- Need faster ways of emulating CPU/MMU

Implementing Virtual Machines

Approach 2: *Direct Execution* with *Trap–&–Emulate*

- Observations: Most *Instructions* are the same regardless of Processor *Privilege 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 over us?
  How about `cli` *(Clear Interrupt Flag)* / `hlt` *(Halt)*?
- Solution: Use *Protection* mechanisms already in CPU
- Run *Virtual Machine's* OS directly on CPU in *Unprivileged* (*User*) *Mode*
  - "*Trap–&–Emulate*" approach
  - Most *Instructions* will just work
  - *Privileged Instruction* will *Trap* into VMM and we can run *Emulation* on that *Instruction*
  - Need "*Virtualizable*" Processor Architecture

Implementing Virtual Machines

Virtualizable Processor Architecture

- 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 (operating in Ring 1) could perform **`push %cs`** as part of its *Kernel Mode* code
      - But then the CPU *Privilege Level* in **`%cs`** wouldn't actually correspond to a Ring 0 value
    - To be *Virtualizable*, **`push`** should instead cause a *Trap* when invoked from Ring 1, allowing then the *Virtual Machine Monitor* to appropriately handle it by eventually pushing a different **`%cs`** value

Implementing Virtual Machines

Virtualizable Processor Architecture

- 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*
  - Another Example:
    `pushf`/**`popf`** *Instructions* can read/write the **`%eflags`** *Register*
    - Bit 9 of **`%eflags`** (**`IF`**) enables *External Interrupts*
    - In Ring 0, **`popf`** can set bit 9; but in Ring 1, CPU **silently** ignores **`popf`**!
    - To be *Virtualizable*, **`pushf`**/**`popf`** *Instructions* should instead cause *Traps* in Ring 1, so that the *Virtual Machine Monitor* can detect when Guest OS (operating in Ring 1) wants to change its *Interrupts* level

Implementing Virtual Machines

Virtualizable Processor Architecture

- *Virtualizable* CPU : All sensitive *Instructions* are *Privileged* and should *Trap*

- *Privilege* Level should not be visible to Software
  - Guest OS shouldn't be able to query and find out it's in a VM environment
    - x86 problem: **`movw %cs, %ax`**
      - Raises Invalid **`opcode`** *Exception* (**`UD`**) in *User Mode*, (can modify **`%cs`** only in *Kernel Mode*)

Note: **`%ax`** is lower-16-bits part of **`%eax`** / **`%rax`**, *Instruction* performs partial write of full *Registers*

- *Trap* should be transparent to Software in VM
  - Guest OS (in VM environment) shouldn't be able to tell if *Instruction Trapped*
    - x86 problem: *Traps* can destroy Machine State
    - e.g. if Guest OS state (internal *Segment Register*) becomes out of sync with *Global Descriptor Table*

- See [Goldberg] for a discussion

Virtualizing Traps

- What happens when an *Interrupt* or *Trap* occurs
  - Like normal Kernels: But we *Trap* into the *Virtual Machine Monitor*
- What if the *Interrupt* or *Trap* should go to Guest OS?
  - Example: *Page Fault*, Illegal *Instruction*, *System Call*, *Interrupt*
  - Restart the Guest OS execution, *Emulating* the *Trap*
- x86 example:
  - Provide an *Interrupt Descriptor Table* (IDT) so that CPU vectors back to VMM
  - Lookup *Trap* vector of Guest OS' (in VM environment) "*Virtual*" IDT
    - How can *Virtual Machine Monitor* know this?
      Location of IDT is kept in **`%idtr`** (IDT *Register*, loaded using the **`lidt`** *Instruction*)
  - Push *Virtualized* **`%cs`**, **`%eip`**, **`%eflags`** on *Stack*
  - Switch to *Virtualized Privileged Mode*

Implementing Virtual Machines

Virtualizing Memory

- OS assumes it has 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 *Virtual Machine Monitor* partitions *Memory* among VMs
  - *Virtual Machine Monitor* needs to assign *Physical Pages* to VMs
  - *Virtual Machine Monitor* needs to control mappings for Isolation
    - Cannot allow a Guest OS to map a *Virtual Page* to any *Physical Page*
    - Guest OS can only map to a *Physical 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, *Virtual Machine Monitor* needs to control access by Guest OS to the *Page Tables*

Implementing Virtual Machines

Virtualizing Memory

One Solution: *Direct Mapping*

- The Guest OS creates *Page Tables* and the *Virtual Machine Monitor* uses these
  - These *Page Tables* are **used directly by the MMU Hardware**
- *Page Tables* work the same as before, but Guest OS has to be constrained to only map to the *Physical Pages* it "owns"
- *Virtual Machine Monitor* responsible to **validate all updates** to *Page Tables* by Guest OS
  - Guest OS can read *Page Tables* without modification
  - But *Virtual Machine Monitor* needs to check all *Page Table Entry* (PTE) writes to ensure that the *Virtual*-to-*Physical* mapping is valid
    - i.e. that the Guest OS actually "owns" the *Physical Page* being used in that *Page Table Entry*
  - Have to modify Guest OS to perform *Hypervisor* call into VMM when updating PTEs
- Works fine if you can **modify** the OS
  - Used in *Xen Paravirtualization*

Implementing Virtual Machines

Virtualizing Memory

Second Approach (as usual) : Add a *Level of Indirection*

- Define 3 Abstractions of *Memory*
  - ***M**achine*: Actual Hardware *Memory*
    - 16 GB of DRAM
  - ***P**hysical*: Abstraction of Hardware *Memory* managed by Guest OS
    - If a *Virtual Machine Monitor* allocates 512 MB to a VM, the Guest OS thinks the computer has 512 MB of Contiguous *Physical Memory* (underlying Machine *Memory* may not be Contiguous)
  - ***V**irtual*: *Virtual Address Spaces* we already know by now
    - Our standard 2^{32} or 2^{64} *Address Space*

Translation: VM's Guest ***V**irtual Address* → VM's Guest ***P**hysical Address* → Host ***M**achine Address*

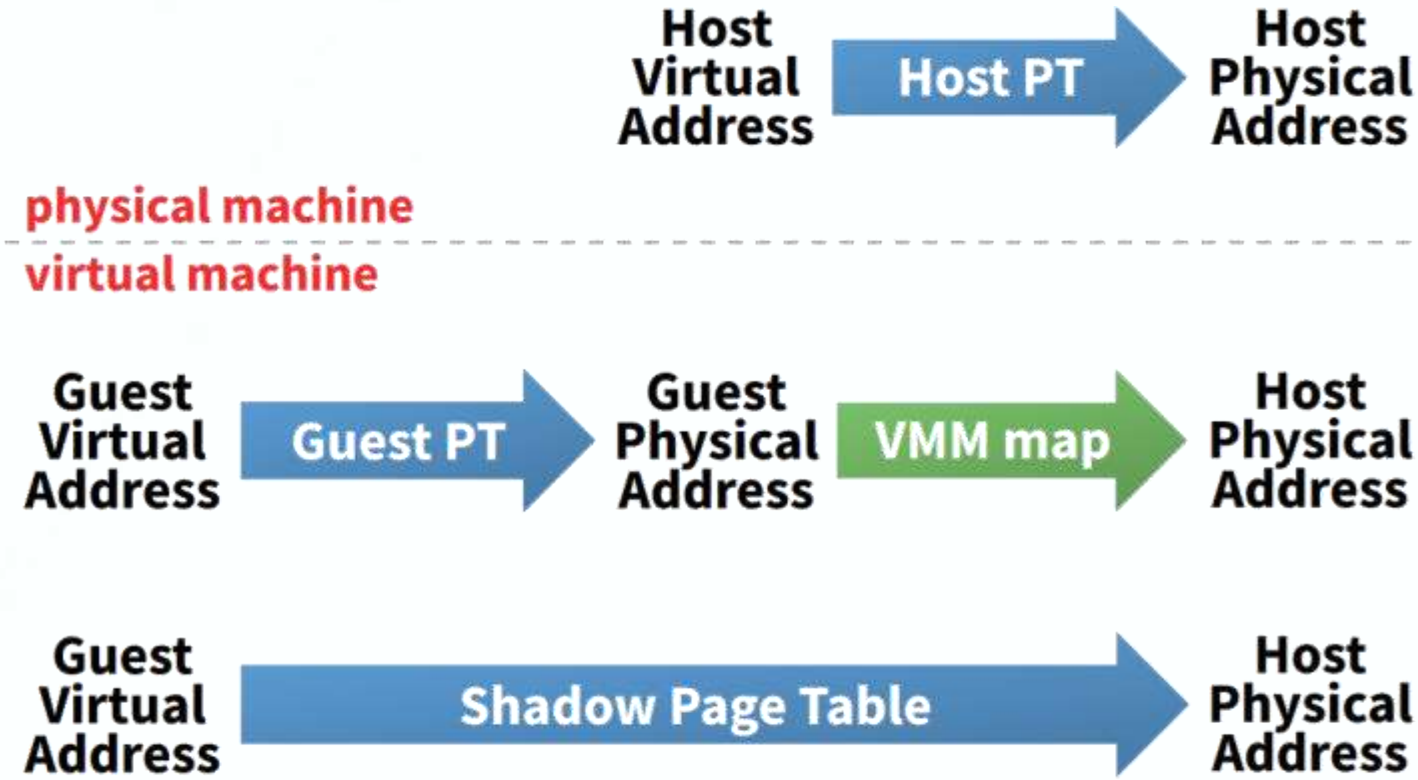
- In each VM, the Guest OS creates and manages *Page Tables* for its *Virtual Address Spaces* as it normally does
  - But these *Page Tables* are **not used by the MMU Hardware**

Implementing Virtual Machines

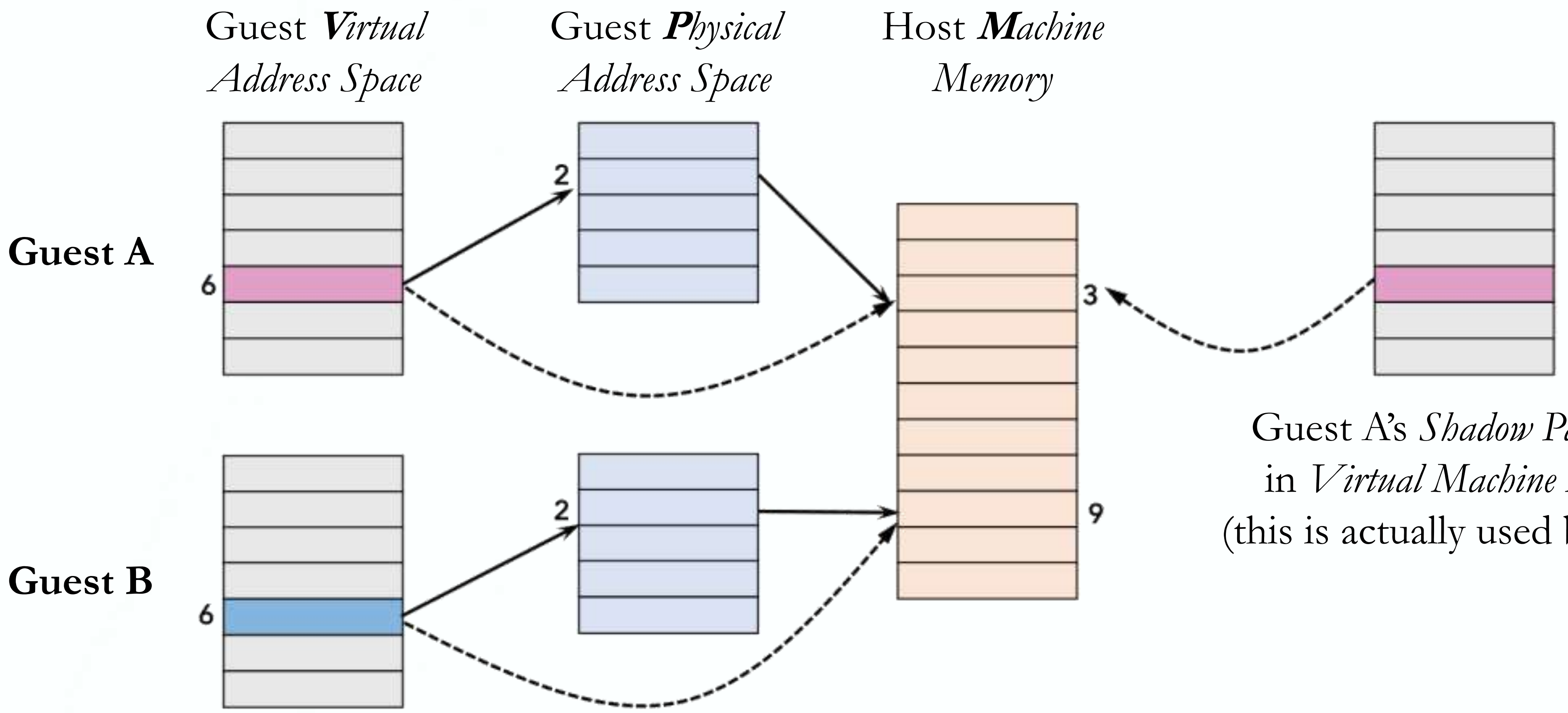
Shadow Page Tables

- The *Page Tables* **actually used by the MMU Hardware**
- *Virtual Machine Monitor* creates and manages "*Shadow*" *Page Tables* that directly map Guest OS ***V**irtual Pages* → Host ***M**achine Pages*
  - Avoid the *Translation* step of Guest ***V**irtual Address* → VM's Guest ***P**hysical Address*
  - These *Shadow Page Tables* are the ones loaded on a *Context Switch* (and used by MMU)
- *Virtual Machine Monitor* is responsible to keep the *Shadow Page Tables* Consistent
  - ***V→P*** Consistency changes may be made by Guest OS
    - e.g. changing Guest *Page Table* to update *Page Table Entries* and while allocating *Pages*
  - ***V→M*** Consistency changes may be made by Hardware
    - e.g. changing *Accessed/Dirty* bits on *Page* access at Host ***M**achine Memory*
  - Also any necessary TLB *Flushes* need to be managed by *Virtual Machine Monitor*
    - i.e. during *Context-Switching*

Memory Mapping Summary



Shadow Page Table Example



Guest A's *Shadow Page Table* in *Virtual Machine Monitor* (this is actually used by MMU)

Implementing Virtual Machines

More on *Shadow Page Tables*

- VM Guest OS cannot be allowed access to *Page Tables* in Host ***M**achine Memory*
  - VMM has to keep track of state in which the VM Guest OS thinks its *Page Tables* should be
- Two classes of *Page Faults* (from the viewpoint of Guest OS)
  - ***True*** *Page Faults* when *Page* not in VM's Guest OS *Page Table*
  - ***Hidden*** *Page Faults* when just *Misses* in *Shadow Page Table*

VMM progressively builds up *Shadow Page Tables* by tracking *Page Faults* generated by Guest OS

- e.g. Guest writes a new ***V**irtual Page* → ***P**hysical Page Number* mapping in its Guest *Page Tables*…

  … but the Hardware will actually use the *Shadow Page Tables*
  - *Page Fault* caused by *Invalid* Guest ***V**irtual Page Number* $\not\rightarrow$ Host ***M**achine Memory*
  - *Page Fault* is "forwarded" to the *Virtual Machine Monitor*
  - Compares Guest *Page Table* & *Shadow Page Table*, notices no such ***V→M*** mapping yet
  - Sets up Guest ***V**irtual Page Number* → Host ***M**achine Page Number* mapping for Hardware, and inserts it into the *Shadow Page Table*

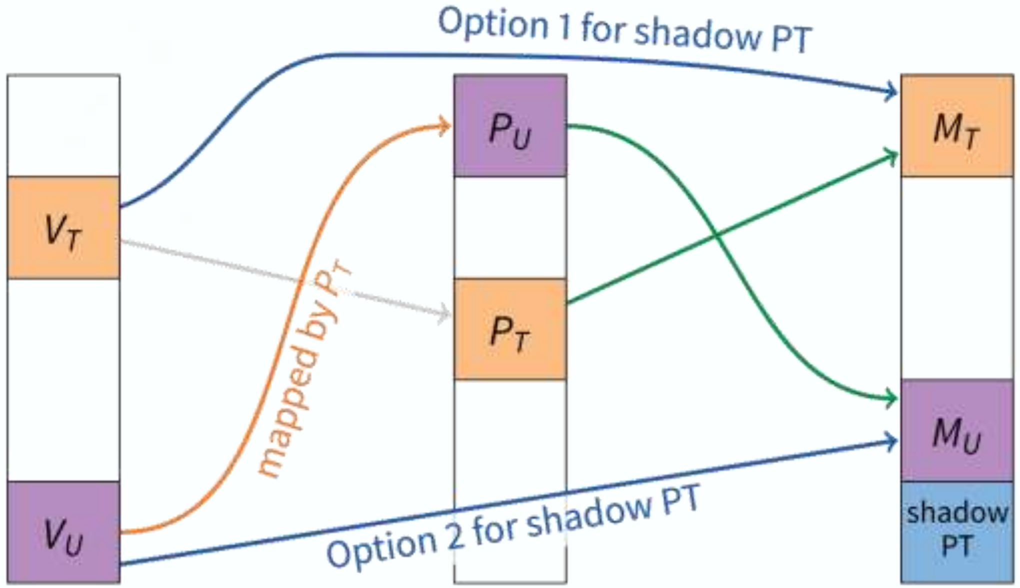
Implementing Virtual Machines

Shadow Page Table **Issues**

- Hardware only ever sees *Shadow Page Table*
  - Guest OS only sees its own VM *Page Table*, never *Shadow Page Table*
- Consider the following:
  - Guest OS (*User-Level*) has a *Page Table* ***T*** mapping $\mathbf{V_U} \rightarrow \mathbf{P_U}$
  - Guest OS (*Kernel-Level*) *Page Table* ***T*** itself resides at Guest ***P****hysical Address* $\mathbf{P_T}$
    - Need another Guest *Page Table Entry* to map $\mathbf{V_T} \rightarrow \mathbf{P_T}$
      - e.g. in Pintos 1-to-1 simple mapping: $\mathbf{V_T} = \mathbf{P_T}$ + **`PHYS_BASE`**
  - *Virtual Machine Monitor* stores $\mathbf{P_U}$ in Host ***M****achine Address* $\mathbf{M_U}$, and $\mathbf{P_T}$ in Host ***M****achine Address* $\mathbf{M_T}$
- What can *Virtual Machine Monitor* put in *Shadow Page Table*?
  - In *Shadow Page Table*, safe to map User *Page* ($\mathbf{V_U} \rightarrow \mathbf{M_U}$) **–or–** map *Page Table* ($\mathbf{V_T} \rightarrow \mathbf{M_T}$)
- But **not safe to map both** simultaneously!
  - If OS changes *Page Table* ***T*** at $\mathbf{P_T}$, may make $\mathbf{V_U} \rightarrow \mathbf{M_U}$ in *Shadow Page Table* incorrect
  - If OS reads/writes $\mathbf{V_U}$, may require *Accessed/Dirty* bits to be changed in *Page Table* $\mathbf{P_U}$ (Hardware only accesses and can thus only change the *Shadow Page Table* $\mathbf{M_U}$)

Implementing Virtual Machines

Shadow Page Table Issues – Illustration



- *Option 1:* Guest *Page Table* ***T*** accessible at $\mathbf{V_T} \rightarrow \mathbf{M_T}$, but changes (done by Guest OS) won't be reflected in *Shadow Page Table* or TLB $\Rightarrow$ Access to $\mathbf{V_U} \rightarrow \mathbf{M_U}$ dangerous
- *Option 2:* $\mathbf{V_U} \rightarrow \mathbf{M_U}$ accessible, but Hardware sets *Accessed/Dirty* bits only in *Shadow Page Table*, not in Guest *Page Table* ***T*** at $\mathbf{V_T} \rightarrow \mathbf{M_T}$ (Hardware unaware of Guest *Page Tables*)

Implementing Virtual Machines

Memory Tracing

- *Virtual Machine Monitor* needs to get control on some *Memory* accesses
- Guest OS changes previously used mapping in its VM *Page Table*
  - Must intercept to invalidate stale mappings in *Shadow Page Table*, TLB
  - *Note:* Guest OS should use **`invlpg`** *Instruction*, which would eventually *Trap* to VMM (thus we would have an easy way to detect this) – but in practice many/most OSes are sloppy about this
- Guest OS accesses *Page* when its VM *Page Table* is accessible
  - *Accessed/Dirty* bits in VM *Page Table* may no longer be correct
  - Must intercept to fix up VM *Page Table* (or make VM *Page Table* inaccessible)
- Solution: "*Memory Tracing*"
  - To track *Page* access, mark *Virtual Page Numbers* as *Invalid* in *Shadow Page Table*
  - If Guest OS accesses *Page*, will *Trap* to *Virtual Machine Monitor* with a *Page Fault*
  - *Virtual Machine Monitor* can *Emulate* the result of *Memory* access & restart Guest OS, just as an OS restarts a *Process* after a *Page Fault*

Implementing Virtual Machines

Virtualizing I/O

- Guest OS can no longer interact directly with I/O *Devices*
- Types of communication
  - Special *Instructions* – `in`/`out`
  - *Memory-Mapped* I/O
  - *Interrupts*
  - *Direct Memory Access (DMA)*
- Make `in`/`out` *Trap* to *Virtual Machine Monitor*
- Use *Memory Tracing* for *Memory-Mapped* I/O
- Run *Emulation* of I/O *Device*
  - *Interrupt* – Tell CPU *Emulator* to generate *Interrupt*
  - *DMA* – Copy Data to/from *Physical Memory* of *Virtual Machine*

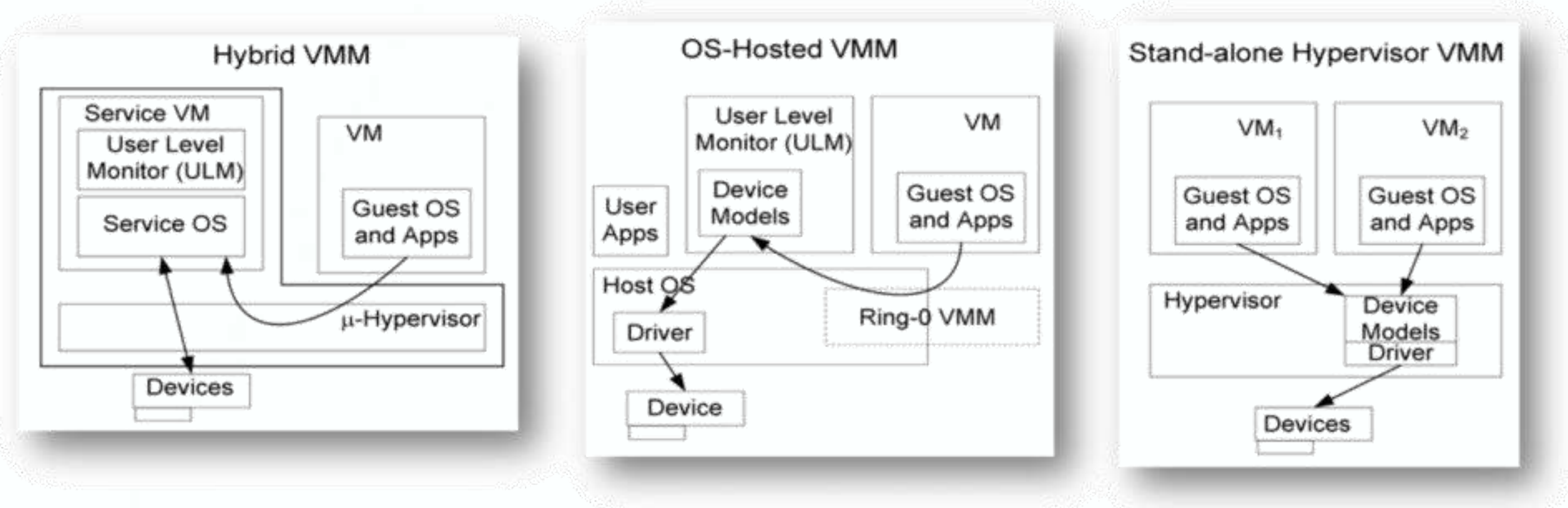
Implementing Virtual Machines

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* : *Virtual Machine Monitor* supports generic *Devices* (***Hosted***)
    - e.g. AMD Lance chipset/PCNet Ethernet *Device*
  - Load *Virtual Device Driver* into OS in VM
  - *Virtual Device Driver* is aware of *Virtual Machine Monitor*, cooperates to pass-on work to a real *Device Driver* (e.g. on underlying Host OS)
- *VMware ESX Server* : *Drivers* run in *Virtual Machine Monitor* (***Hypervisor***)

Implementing Virtual Machines

Virtualizing I/O : Three Models



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

Hardware-assisted Virtualization

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*–&–*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` *Register*
  - Less *Privileged* than *Host Mode* (where *Virtual Machine Monitor* runs)
  - Direct execution of Guest OS code, including *Privileged Instructions*
  - Some sensitive *Instructions Trap* to "*Guest Mode*"
    - e.g. `load %cr3` (*Remember: Context-Switching* & *Page Directories*)
  - Hardware also keeps "*Shadow State*" for many things
    - e.g. `%eflags`

Hardware-assisted Virtualization

Guest Mode

- *Virtual Machine Control Block* (VMCB)
  - Controls what operations *Trap*
  - Records info to handle *Traps* in *Virtual Machine Monitor* via saving the Guest state

Saved Guest state:

- Full *Segment Registers* (i.e. *Base*, *Lim*, *Attr*, not just *Selectors*)
- Full GDT *Register*, LDT *Register*, Task *Register* (Remember: *Segmentation*), the IDT *Register*
- Guest **`%cr3`**, **`%cr2`**, and other **`cr`**/**`dr`** *Registers*
- Guest **`%eip`** & **`%eflags`** *Registers* (**`%rip`** & **`%rflags`** for 64-bit)
- Guest **`%eax`** *Register* (**`%rax`** for 64-bit)

Hardware-assisted Virtualization

Guest Mode

*ENTER*ing and *EXIT*ing *Guest Mode*:

- New *Instruction* (e.g. AMD **`vmrun`**) to enter *Guest Mode*
  - Loads state from Hardware-defined 1-KiB VMCB Data Structure
- Various events (e.g. a Guest VM *Trap*) cause *EXIT* back to Host mode
  - On *EXIT*, Hardware saves state back to VMCB
- Enters *Virtual Machine Monitor*, which can now use the VMCB (saved Guest state) to *Emulate* operation
- Entering / exiting *Virtual Machine Monitor* more expensive than *System Call*
  - Have to save and restore large VMCB Data Structure

Hardware-assisted Virtualization

Hardware Support

- *Memory*
  - Intel *Extended Page Tables (EPT)*, AMD *Nested Page Tables (NPT)*
    - Original *Tables* map ***V**irtual* to Guest ***P**hysical Pages*, managed by Guest OS
    - New *Tables* map Guest ***P**hysical* to Host ***M**achine Pages*, managed by VMM
  - No need to *Trap* to *Virtual Machine Monitor* when Guest OS updates its *Page Tables*
    - Avoid overhead associated with Software-managed *Shadow Page Tables*
  - Tagged TLB w/ *Virtual Process Identifiers (VPIDs)*
    - Tag VMs with VPID, no need to flush TLB on VM/VMM *Context Switch*
- *I/O*
  - Constrain DMA operations only to *Pages* owned by specific VM
  - AMD *Device Exclusion Vector* (DEV) (compare to *Xen*'s *Memory Paravirtualization*)
  - Intel VT-d: IOMMU – Address Translation support for DMA

Memory Management Optimizations

Memory Allocation

- *Virtual Machine Monitors* tend to have simple Hardware *Memory Allocation* policies
  - Static: VM gets 512 MB of *Physical Memory* for life
  - No dynamic adjustment based on load (OSes not designed to handle changes in *Physical Memory*)
  - VMM usually not desired to –itself– have to perform *Swapping* (*"Hypervisor Swapping"*)
- *ESX* Trick of Overcommitting with "*Balloon Driver*"
  - Special pseudo-*Device Driver* running in (supported) Guest OS; consumes *Physical Pages*
  - Communicates with *Virtual Machine Monitor* through special Interface
    - When VMM needs *Memory*, the *Balloon Driver* allocates many *Pages* in Guest OS
      - Forces Guest OS to *Swap* to *Disk* any *Pages* that are least valuable to it
    - *Balloon Driver* informs VMM which *Pages* it has claimed, so that they can be recycled
- Identifying identical Guest *Physical Pages* (e.g. all-zeroes) –even across multiple VMs– and mapping them to a single Host *Physical Page*
  - Map those *Pages* as *Copy-on-Write* (across all mapped VMs)

CS-446/646

Time for Questions !