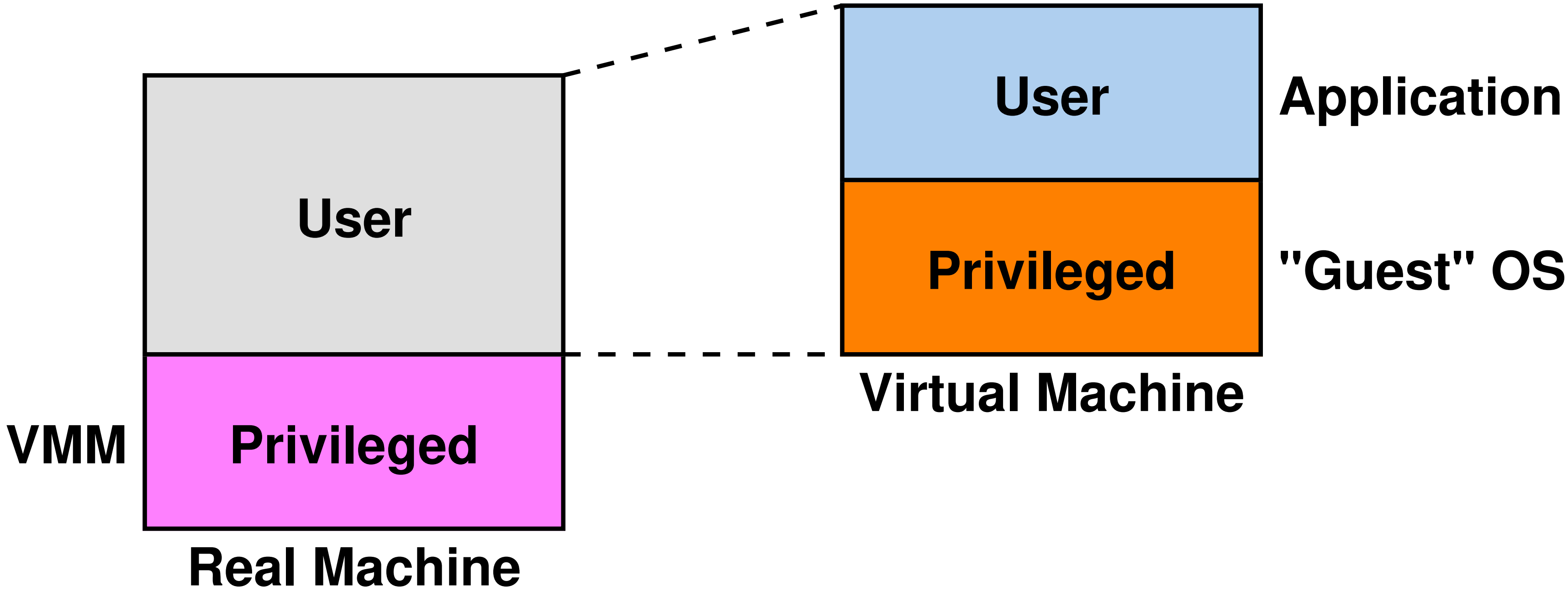
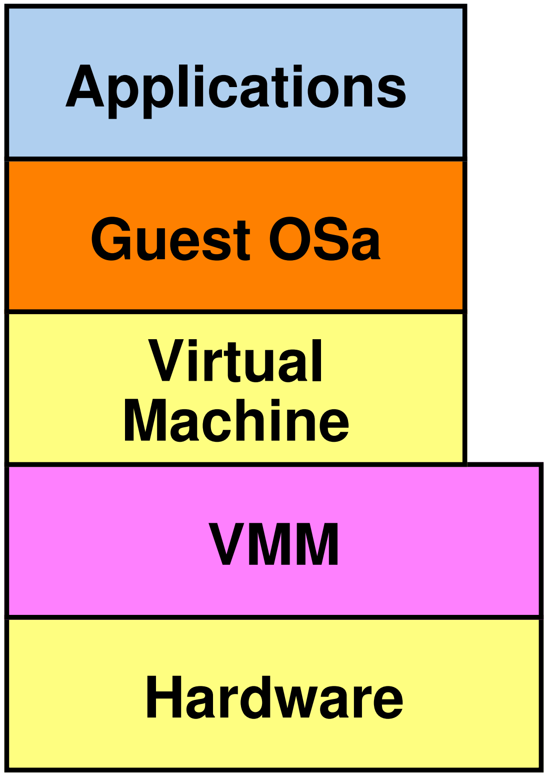
How?



- Run the *entire virtual machine* in *user mode* of the real machine
  - *VMM* runs in the *privileged* mode of the *real* machine
- VMM keeps track of whether each virtual machine is in the *virtual privileged mode* or in the *virtual user mode*
  - *OS* runs in the *(virtual) privileged* mode of the *virtual* machine
  - Applications runs in the *(virtual) user* mode of the *virtual* machine

VMM Operations

- Execute a *non-privileged* instruction
  - e.g., "add", "mul", pointer manipulation

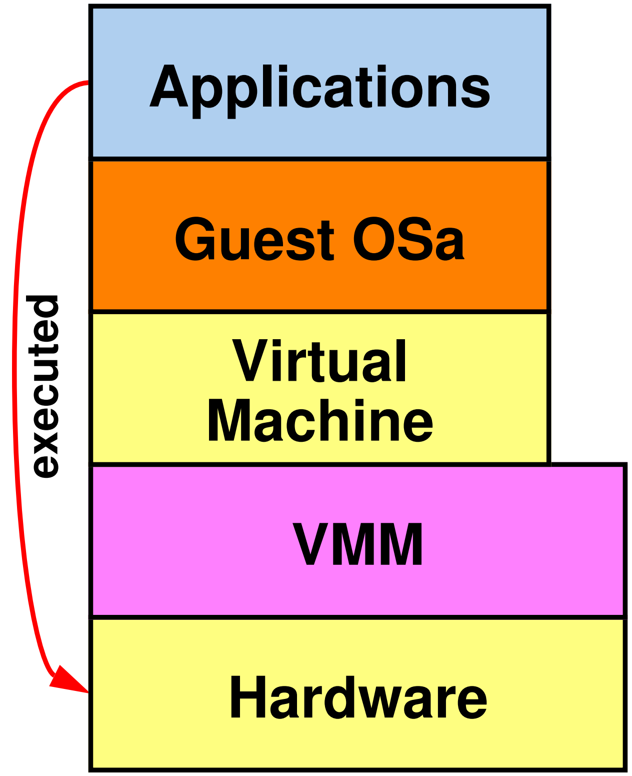


VMM Operations

- Execute a *non-privileged* instruction
  - e.g., "add", "mul", pointer manipulation

executes directly on hardware

- from application's perspective, no difference running in VM or on hardware

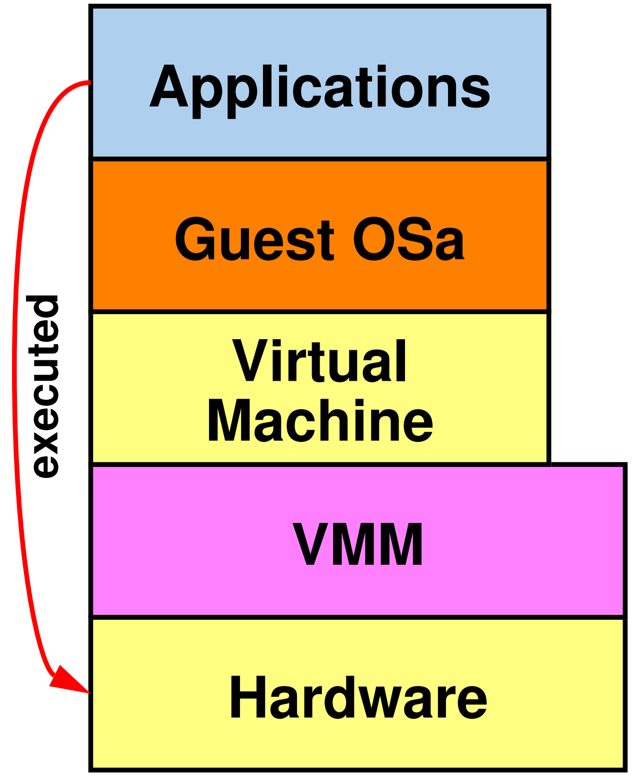


VMM Operations

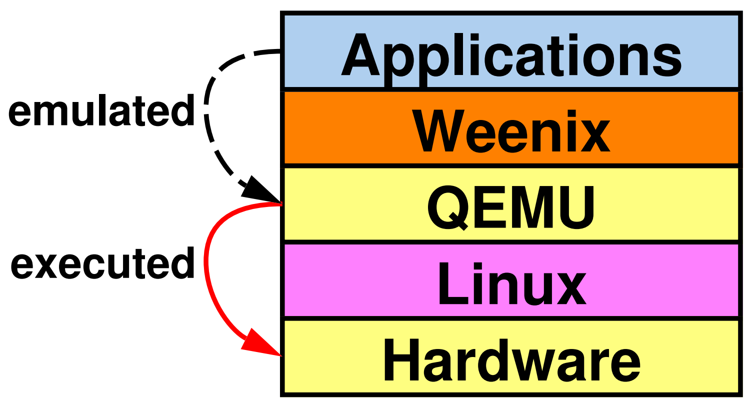
- Execute a ***non-privileged*** instruction
  - e.g., "add", "mul", pointer manipulation

executes directly on hardware

- from application's perspective, no difference running in VM or on hardware



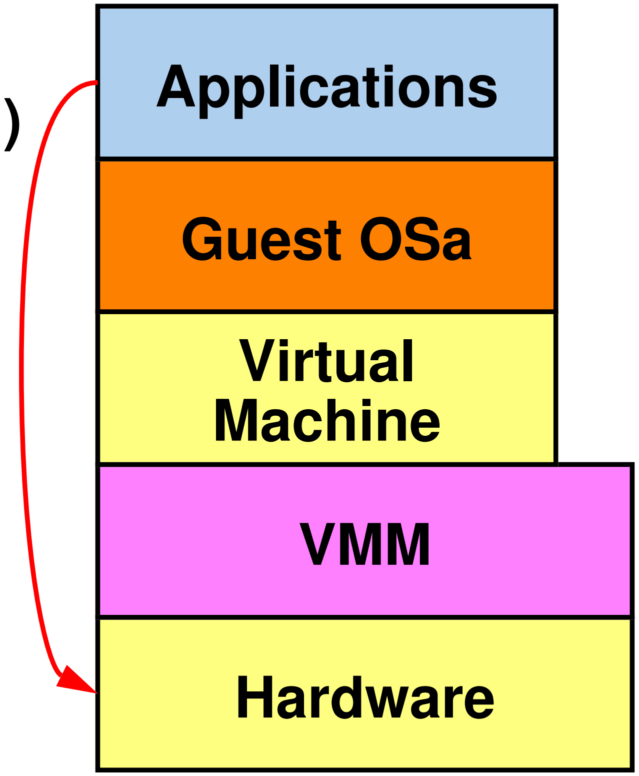
- Note: this looks like our kernel assignments (but quite different)



a ***emulator program*** for the x86 instruction set

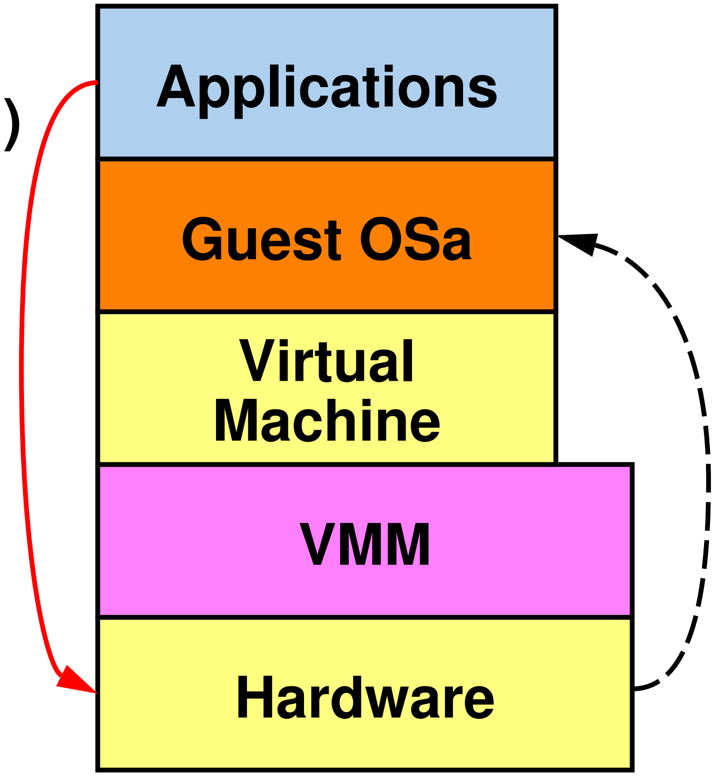
VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)



VMM Operations

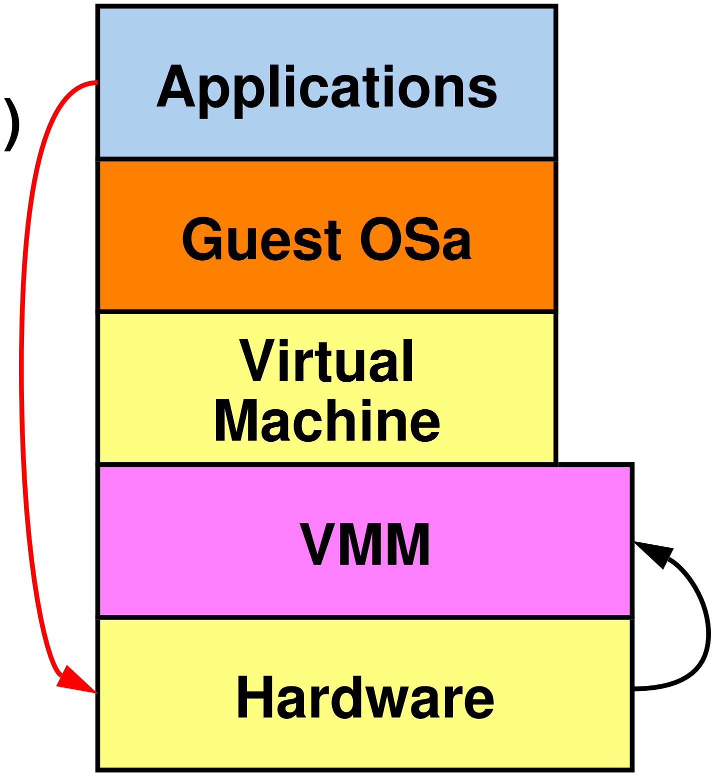
- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)
    - in a real machine, trap handler is indexed by the trap number into a hardware-mandated jump table
    - the VMM needs to find the address of the virtual machine's trap handler in the table and transfer control to it



VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)

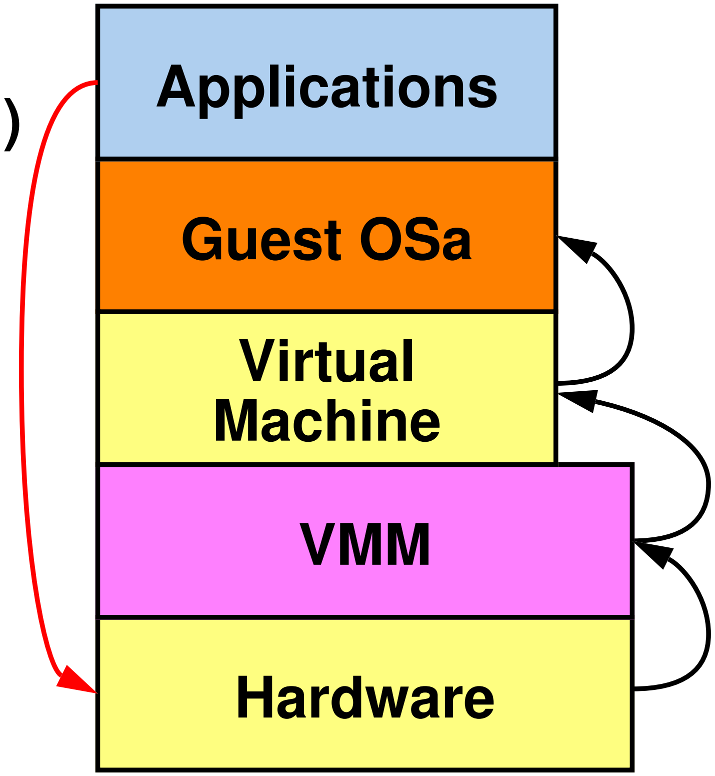
the VMM is invoked
- the VMM figures out which VM is currently executing



VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)

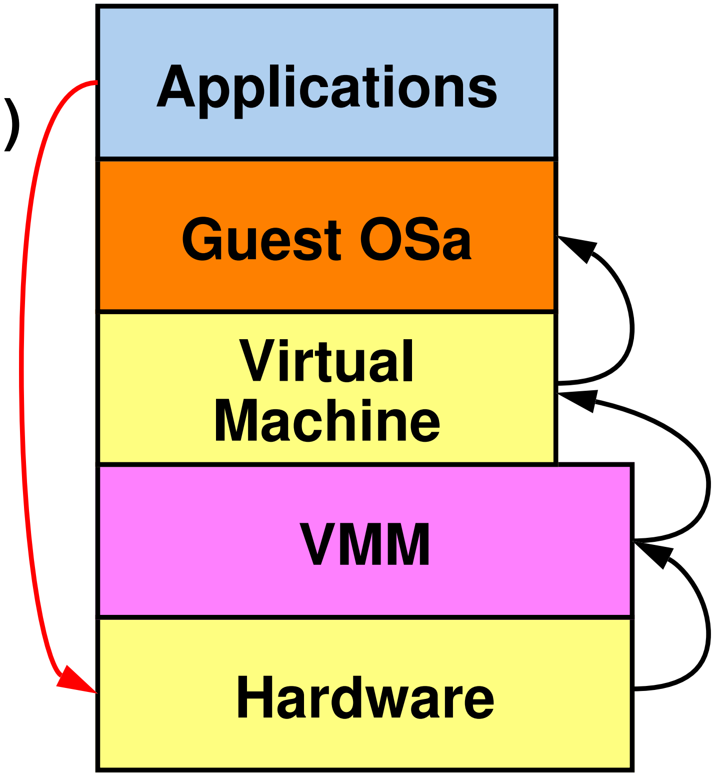
the VMM is invoked
- the VMM figures out which VM is currently executing
- VMM then asks the corresponding VM to *deliver* the trap to its OS
  - VMM should be *virtual machine independent*



VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)

the VMM is invoked
- the VMM figures out which VM is currently executing
- VMM then asks the corresponding VM to *deliver* the trap to its OS
  - VMM should be *virtual machine independent*



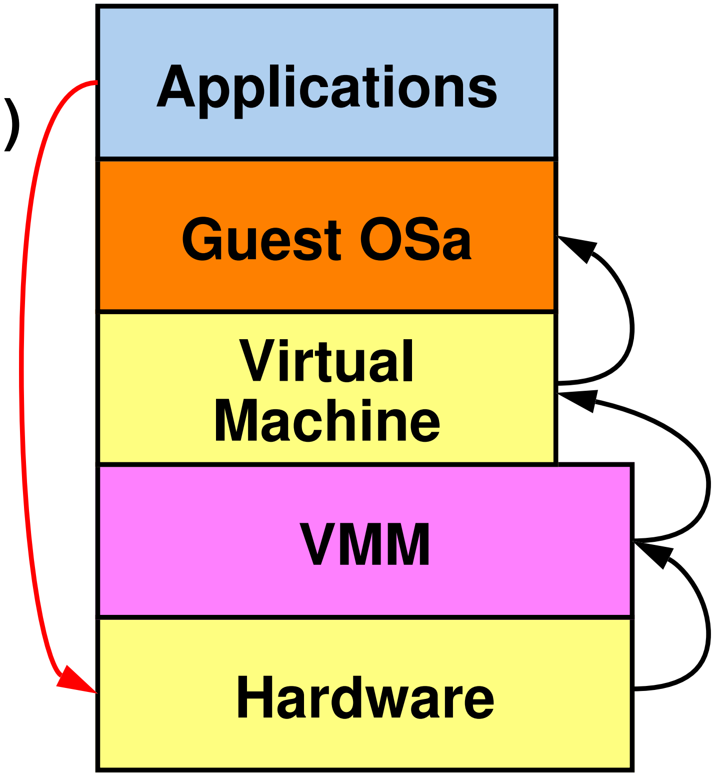
- Without VM, the application will simply traps into the OS directly
  - now it's a lot more involved (and slower)

- *Interrupts* pretty much work the same way

VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)

the VMM is invoked
- the VMM figures out which VM is currently executing
- VMM then asks the corresponding VM to *deliver* the trap to its OS
  - VMM should be *virtual machine independent*

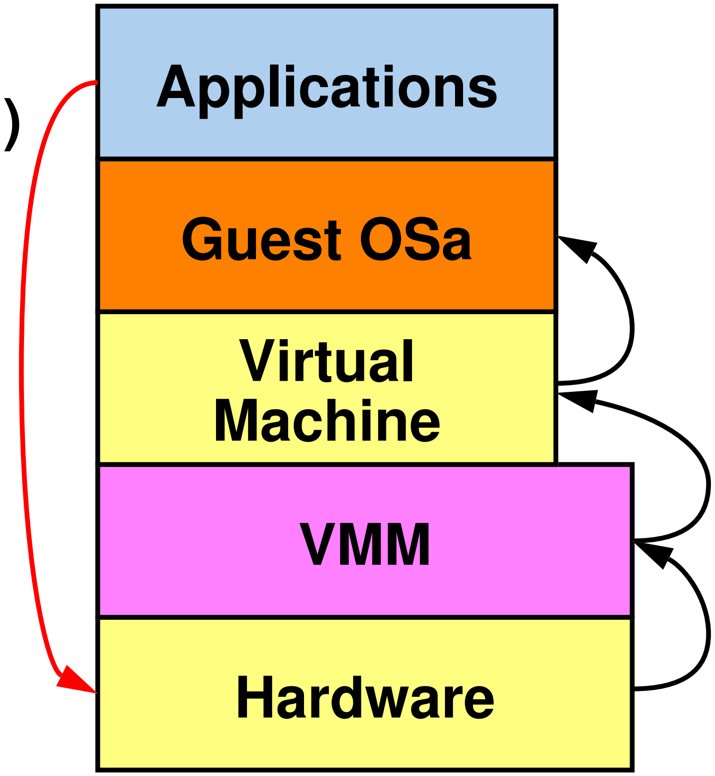


- "Virtual Machine" in the picture contains:
  - virtual CPU, virtual disk, virtual display, virtual keyboard, etc.
    - *data structures* and *code* that *represent hardware components*

VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)

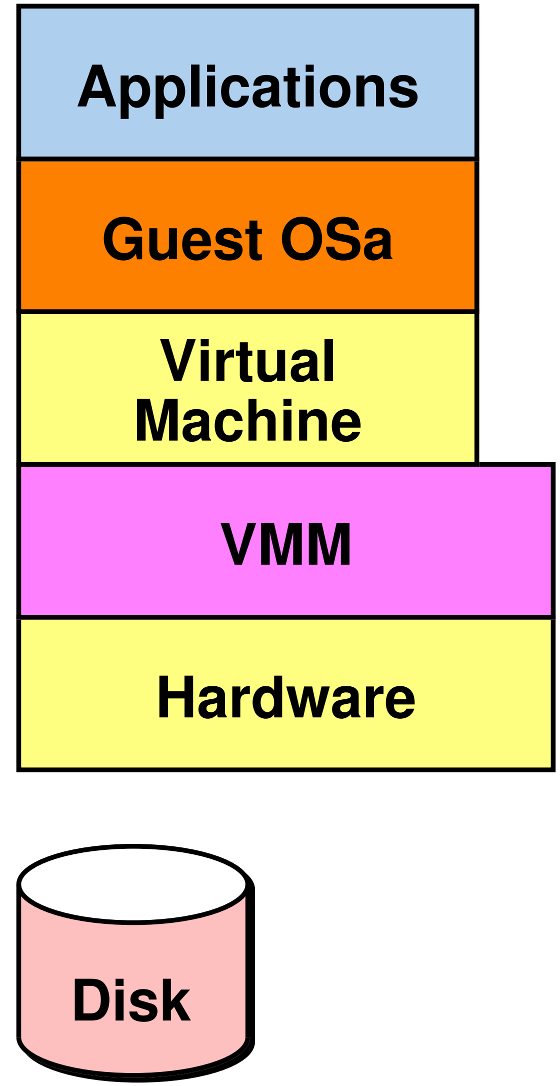
the VMM is invoked
- the VMM figures out which VM is currently executing
- VMM then asks the corresponding VM to *deliver* the trap to its OS
  - VMM should be *virtual machine independent*



- Note that most instructions the trap handler executes are *not* privileged (such as the code to setup PCB, TCB, etc.)
  - clearly, these instructions can run in non-privileged mode
  - what type of code *must* execute in *privileged* mode? (later)
    - what if "return from interrupt" is not privileged?

VMM Operations

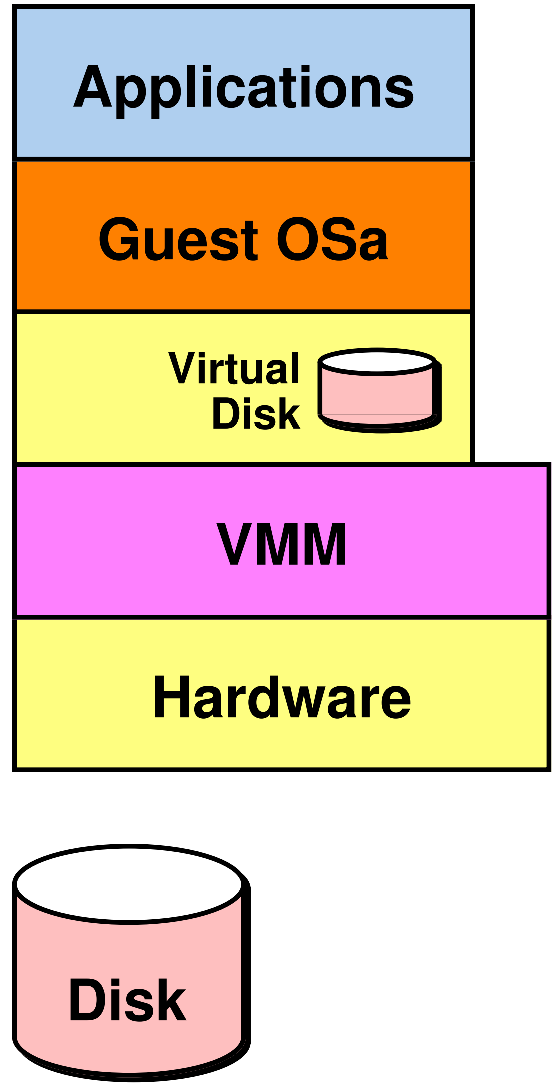
- What about I/O?
  - e.g., `read()`



VMM Operations

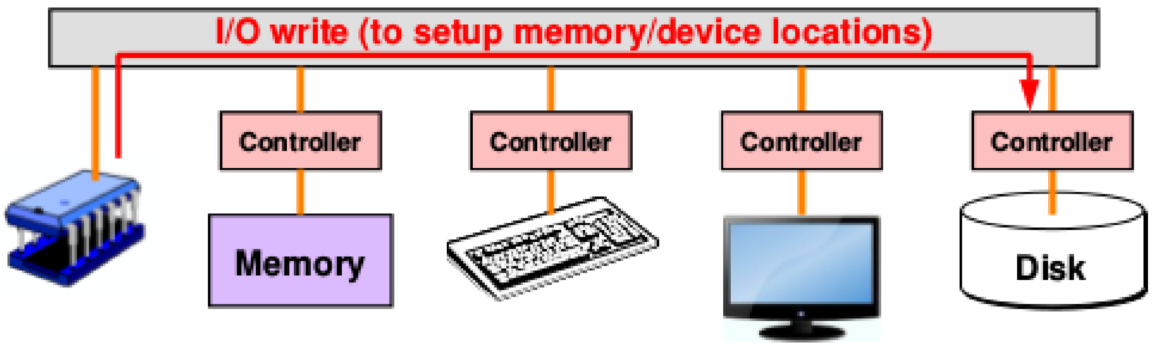
- What about I/O?
  - e.g., `read()`

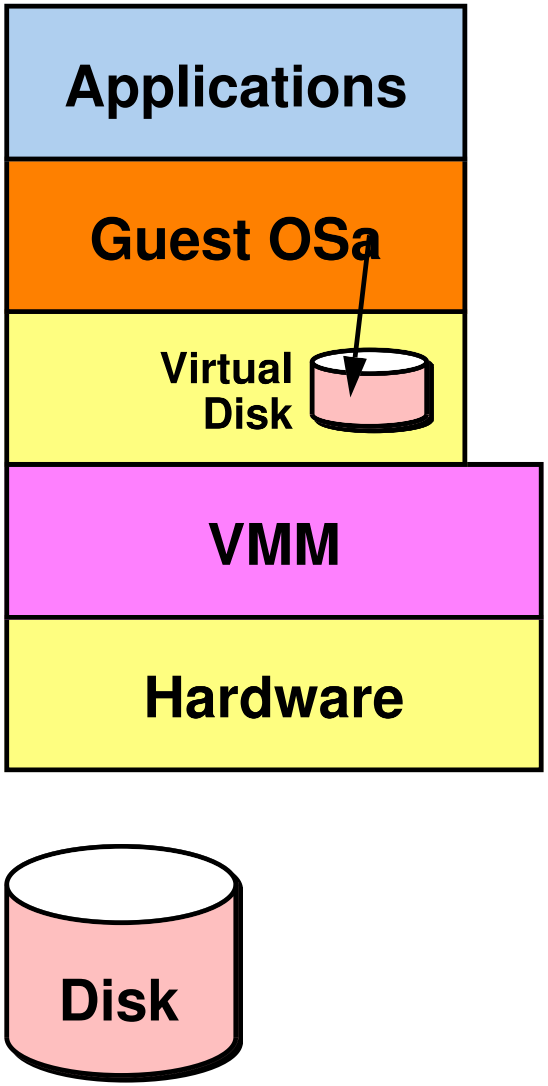
real disk is divvy up among the virtual machines
- each VM has a virtual disk



VMM Operations

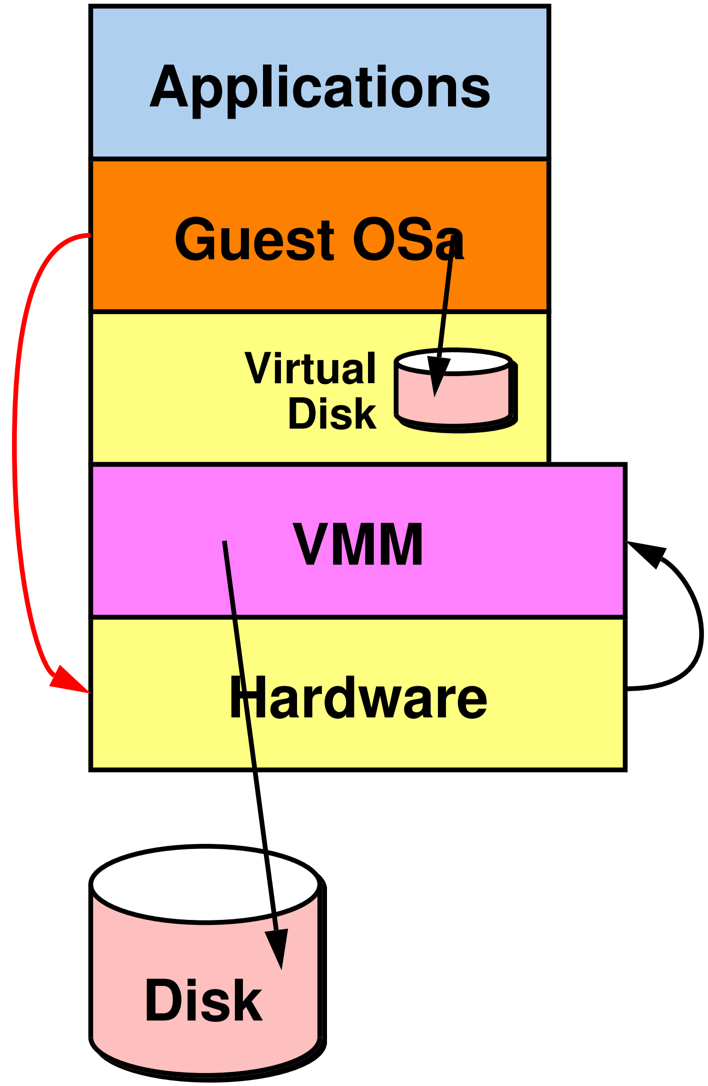
- What about I/O?
  - e.g., `read()`
  - `read()` eventually reaches the OS
  - the OS asks for a block on the virtual disk
    - in x86: memory-mapped I/O





VMM Operations

- What about I/O?
  - e.g., `read()`
  - `read()` eventually reaches the OS
  - the OS asks for a block on the virtual disk
    - in x86: memory-mapped I/O

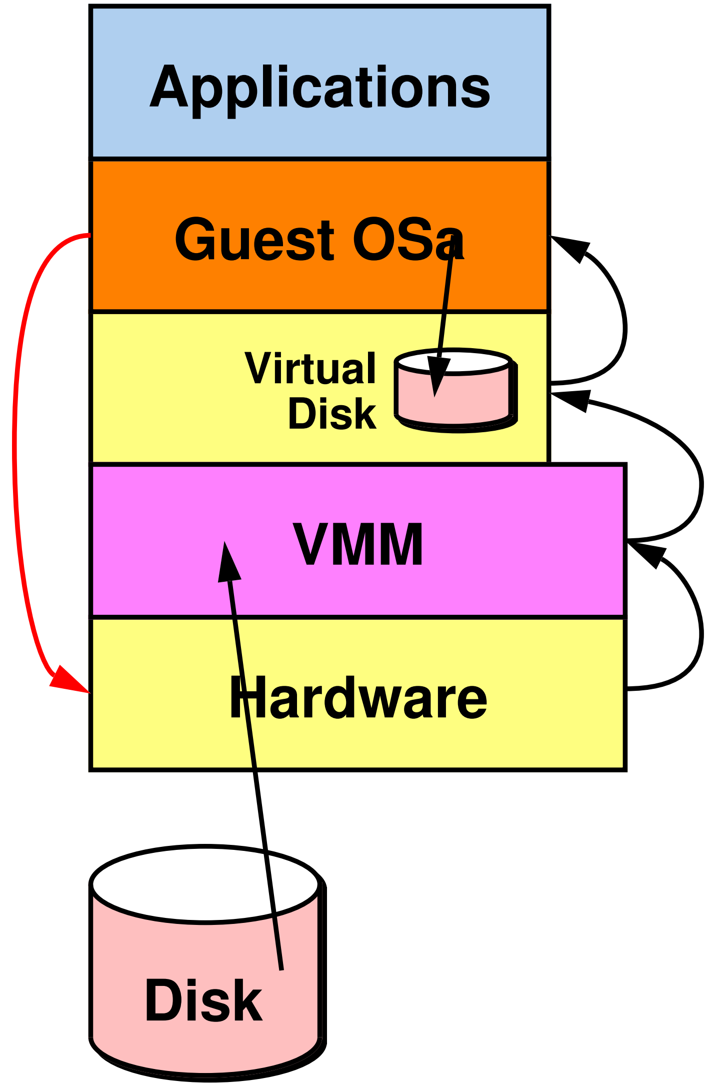


memory-mapped I/O causes a trap into VMM

- the VMM *emulates* the instruction (i.e., translates it into a request for the real disk)
  - there is *no "handler"* in the guest OS for I/O instructions
- there's really *no* disk in the VM

VMM Operations

- What about I/O?
  - e.g., `read()`
  - `read()` eventually reaches the OS
  - the OS asks for a block on the virtual disk
    - in x86: memory-mapped I/O



memory-mapped I/O causes a trap into VMM

- the VMM *emulates* the instruction (i.e., translates it into a request for the real disk)
  - there is *no "handler"* in the guest OS for I/O instructions
- there's really *no* disk in the VM
- the guest OS is expecting an interrupt

Why Virtual Machine?

- It's a good structuring technique for a multi-user system
  - many advantages
- OS debugging and testing
  - run a production OS in a VM, accessible to users
  - test a new OS in a separate VM, accessible to developers
- *Adapt to hardware changes in software*
- *Multiple OSes on one machine*
  - one type of applications run really well in one OS
  - another type of applications run really well in a different OS
  - one physical machine can support both, no user need to suffer
  - today, it's common that a machine in the cloud would run multiple Linux OS instances and multiple Windows OS instances
- *Server consolidation* and *service isolation*
  - web hosting, security concerns
  - cloud computing

Virtualization Requirements

- A virtual machine is an efficient, isolated duplicate of real machine
  - requires faithful virtualization of pretty much all components
    - processor
    - memory
    - interval timers
    - I/O devices
    - etc.
  - this is "*pure*" virtualization
    - costly
- ***Paravirtualization:***
  - virtualized entity is a bit different from the real entity
    - so as to enhance scalability, performance, and simplicity
    - it is probably aware that it's not running on a real machine

Pure Processor Virtualization Requirements

- Can all processors be virtualized?
- Virtualizing the processor requires:
  1) multiplexing the real processor among the virtual machines
     - relatively straightforward
  2) making each virtual machine behaves just like a real machine
     - all instructions must work identically
     - generation of and response to traps and interrupts must be identical as well

Pure Processor Virtualization Requirements

- Processor in the virtual machine is the real processor
  - instructions are executed (and not interpreted or emulated)
  - traps are generated just as they are on real machines
    - in a real machine, trap handler is indexed by the trap number into a hardware-mandated jump table
    - the VMM needs to find the address of the virtual machine's trap handler in the table and transfer control to it
    - interrupts pretty much work the same way

- Pretty much everything can be worked out except for one problem
  - if a virtual machine is *executing* in the virtual privileged mode, what's to prevent it from changing things like memory-mapping *(which can affect the execution the virtual machine)?*
    - or what if "return from interrupt" is *not* privileged?
      - this may not be a realistic example because, clearly, "return from interrupt" must be privileged
    - such instructions must be identified and make sure that they work properly under virtualization

Pure Processor Virtualization Requirements

- Under virtualization, we must distinguish between *sensitive instructions* and *privileged instructions*

- *Privileged instructions:*
  - cause privileged-instruction *trap* when executed in *user mode* but *execute fully* when the processor is in *privileged mode*

- *Sensitive Instructions:*
  - ?

Pure Processor Virtualization Requirements

- *Sensitive Instructions:*
  - *Control-sensitive instructions:*

    1) instructions that affect allocation of (real) system resources
       - such as insturctions that change the mapping of virtual to real memory
  - *Behavior-sensitive instructions:*

    2) instructions whose effect depends on the allocation of (1)
       - such as insturctions that returns the real address of a location in virtual memory

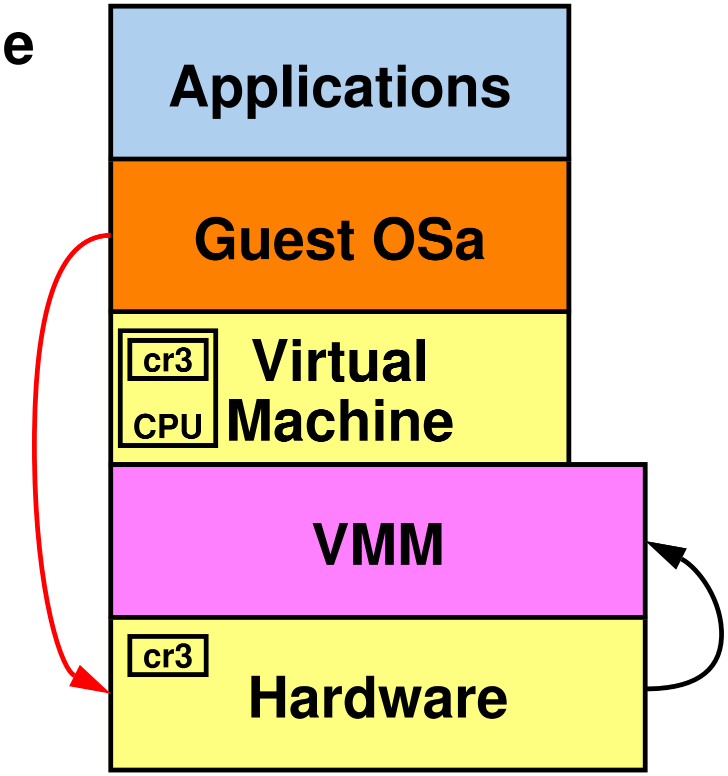
    3) instructions whose effect depends on the current processor mode
       - such as x86's `popf` insturctions that sets a set of processor flags when run in privileged mode, but set a different set of flags otherwise

Sensitive Instruction Example

- A ***sensitive*** **instruction must execute in the privilege mode** ***(in the kernel)***
  - **e.g., insturctions that change the mapping of virtual to real memory**

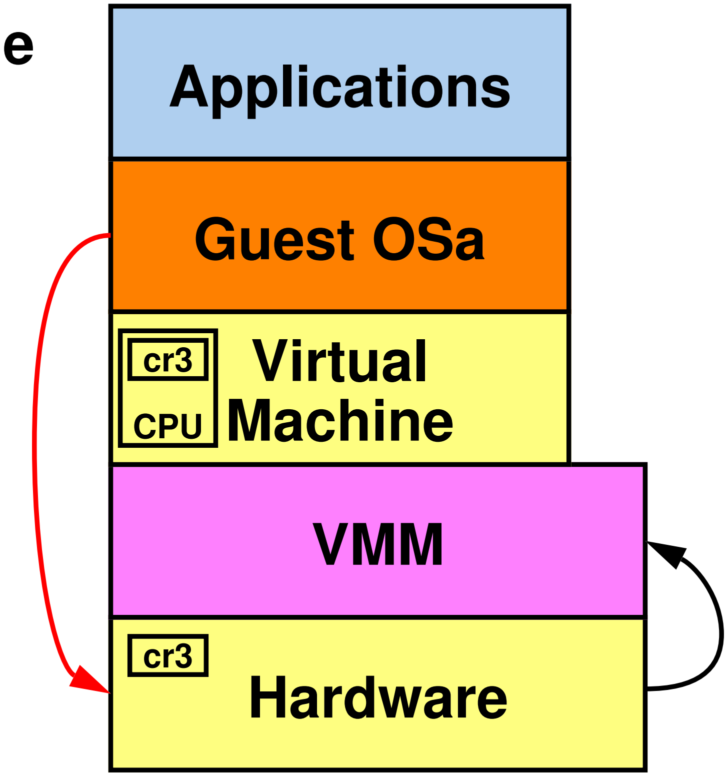
Guest OS runs in the ***user*** **mode of the** ***real*** **processor**

- **executing a** ***sensitive*** **instruction will cause a trap into the VMM**
- **must** ***not*** **execute such instruction**
- **the VMM** ***emulates*** **the instruction**



Sensitive Instruction Example

- A *sensitive* instruction must execute in the privilege mode *(in the kernel)*
  - e.g., insturctions that change the mapping of virtual to real memory



- All *sensitive* instructions must also be *privileged*
  - but what if it's not?
    - you *cannot* build a virtual machine for this processor
  - this gives us another definition of *"sensitive instruction"*
    - it's an instruction that if it's not privileged, it will cause the guest OS (inside a virtual machine) to execute *incorrectly*
      - this *operational definition* may be more useful for an introductory class like ours

Pure Processor Virtualization Requirements

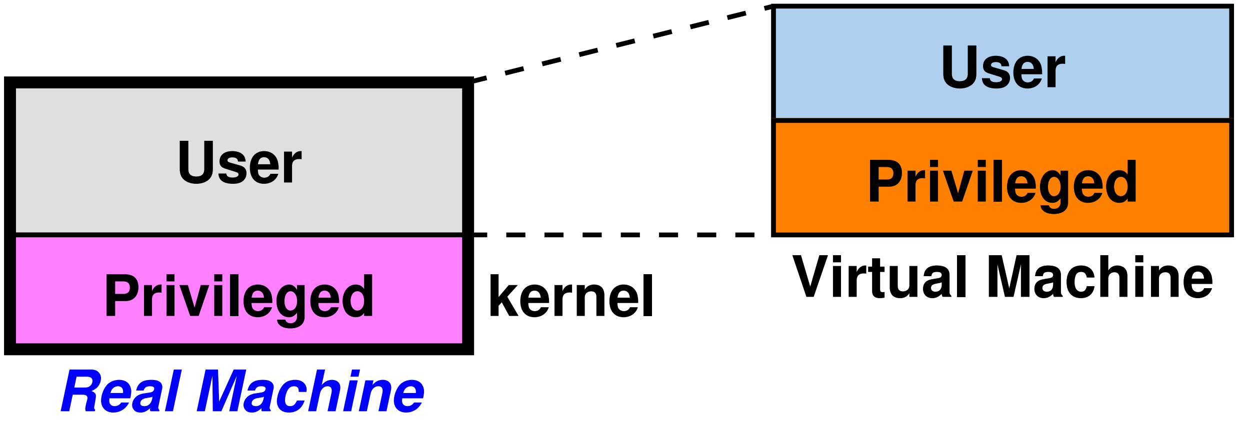
- [Popek and Goldberg, 1974] ***proved*** that the ***sufficient condition*** to be able to construct a virtual machine is simply the following:
  - a computer's set of ***sensitive instructions*** is a ***subset*** of its ***privileged instructions***
  - i.e., if you execute a sensitive insturction in user mode, you will trap into the kernel
    - more importantly, if you execute a sensitive insturction in ***virtual user or virtual privileged mode***, you will trap into ***VMM***

- The above theorem holds for the IBM 360
  - virtual machines can be constructed for it
- The above theorem does ***not*** hold for the x86 processors
  - cannot build virtual machines for x86

The (Real) 360 Architecture

- Two execution modes
  - supervisor and problem (user)
  - all sensitive instructions are privileged instructions
- Memory is protectable: 2KB granularity
- All interrupt vectors and the clock are in first 512 bytes of memory
- I/O done via channel programs in memory, initiated with privileged instructions
- Dynamic address translation (virtual memory) added for Model 67

Actions on Real 360

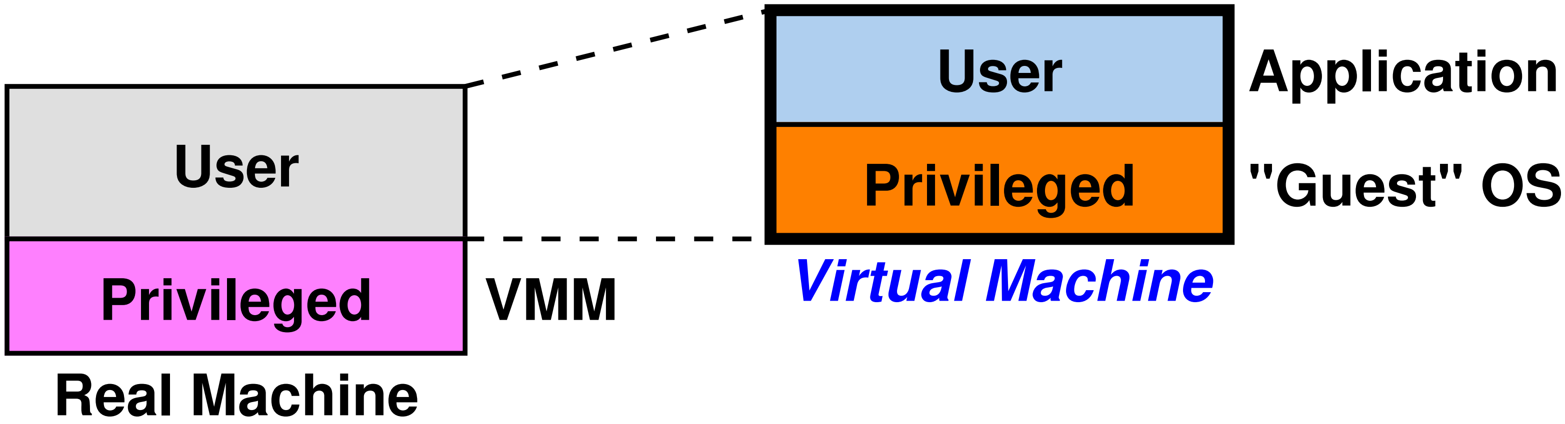


	User Mode	Privileged Mode
non-sensitive instruction	executes fine	executes fine
"errant" instruction	traps to kernel	traps to kernel
sensitive instruction	traps to kernel	executes fine

privileged but not sensitive

since all sensitive instructions are privileged for IBM 360

Actions on Virtual 360



	Virtual User Mode	Virtual Privileged Mode
non-sensitive instruction	executes fine	executes fine
"errant" instruction	traps to VMM; VMM delivers trap to the Guest OS	traps to VMM; VMM delivers trap to the Guest OS
sensitive instruction	traps to VMM; VMM delivers trap to the Guest OS	traps to VMM; VMM verifies and *emulates* instruction

Virtual Devices?

- Terminals
  - connecting (real) people
- Networks
  - didn't exist in the 60s
  - (how did virtual machines communicate?)
- Disk drives
  - CP67 supported "mini disks"
  - extended at Brown into "segment system"
- Interval timer
  - virtual or real?