4.2 Rethinking Operating-System Structure

- Virtual Machines
- Microkernel

Monolithic Kernel

- Major advantage of monolithic kernel
  - performance
- Major down side of monolithic kernel
  - reliability (i.e., buggy kernel)
- Proposal to fix the reliability problem
  - shrink the code in "privileged mode"
- Two major approaches
  - *virtual machines*
  - *microkernel*

Virtual Machines

- A nicely designed and implemented monolithic OS is great
  - but that's not the reality
- Major problem with a monolithic OS *implementation*
  - bugs in one component can adversely affect another component
    - worse if large number of programmers contribute code
      - some coders are not as good as others
      - good coders have bad days
- Modern OSs *isolate* applications from one another
  - code executing in the privileged mode can do things the user mode code cannot
    - e.g., invoking privileged instructions
  - if you invoke a privileged instruction in user mode, you will cause a violation and trap into the kernel
- Can the same kind of *isolation* be provided for *OS components*?
  - if yes, at what cost? (there is no free lunch)

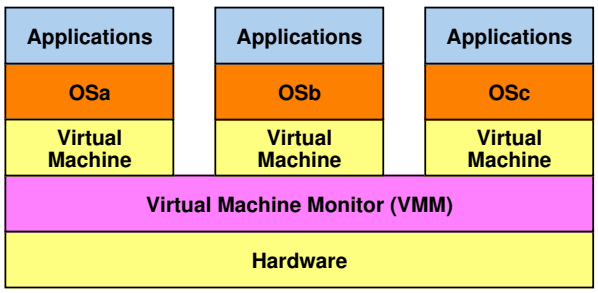
Virtual Machines Part 1: > 45 Years Ago

- Had a different motivation

It's 1964 ...

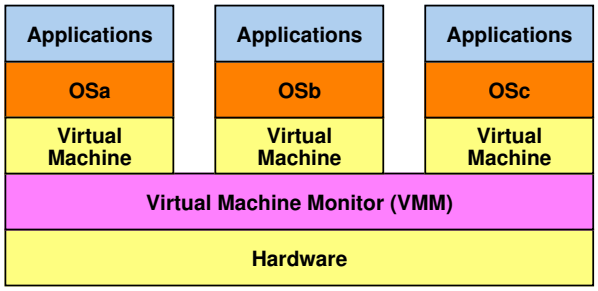
- IBM wants a *multiuser time-sharing system*
- TSS (Time-Sharing System) project
  - it's a very difficult system to build
  - large, monolithic system
  - lots of people working on it
  - for years
  - total, complete flop
- CMS
  - *single-user time-sharing system* for IBM 360
- CP67
  - *virtual machine monitor (VMM)*
  - supports multiple virtual IBM 360s
- Put the two together ...
  - a (working) *multiuser time-sharing system*

Virtual Machines



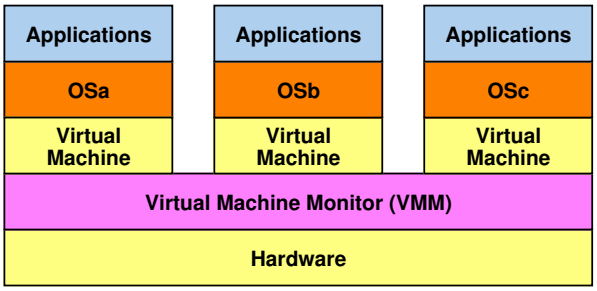
- In OS, a *"monitor"* is a *synchronization construct* that allows executing entities to have both *mutual exclusion* and the ability to *wait* (block) for a certain condition to become true
- What abstraction does a *virtual machine* provide?

Virtual Machines



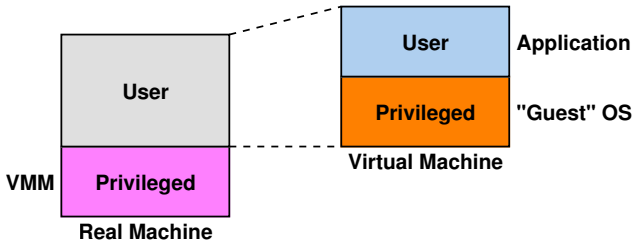
- In OS, a *"monitor"* is a *synchronization construct* that allows executing entities to have both *mutual exclusion* and the ability to *wait* (block) for a certain condition to become true
- What abstraction does a *virtual machine* provide?
  - *hardware*

Virtual Machines



- A single user time-sharing system could be developed independently of the VMM
  - and it can be tested on a real machine (which behaves identical to the VM)
  - no ambiguity about the interface th VMM must provide to its applications - *identical* to the *real machine!*

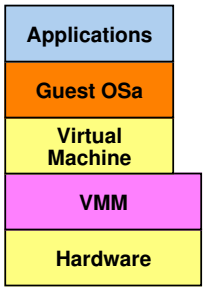
How?



- 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

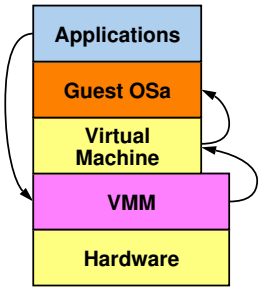
VMM

- 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



VMM

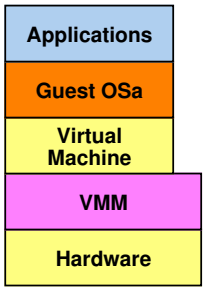
- Processor in the virtual machine is the *real* processor
  - instructions are *executed* (and not interpreted or emulated)
  - traps are generated just as they are on real machines
    - in a real machine, trap handler is indexed by the trap number into a hardware-mandated jump table
    - the VMM 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



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

VMM Operations

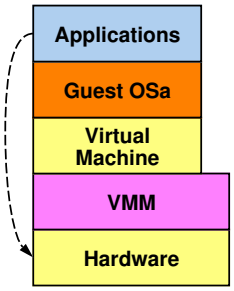
- Execute a *non-privileged* instruction
  - e.g., "add", "mul"



VMM Operations

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

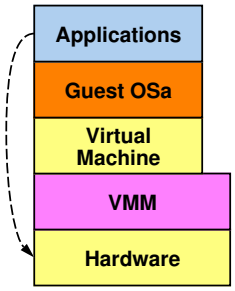
executes directly on hardware
- from application's perspective, no difference running in VM or on hardware



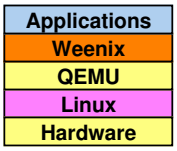
VMM Operations

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

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



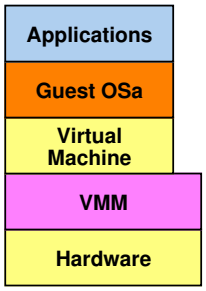
- Note: this is kind of like our kernel assignments (but quite different)



a *software emulator* for the x86 instruction set

VMM Operations

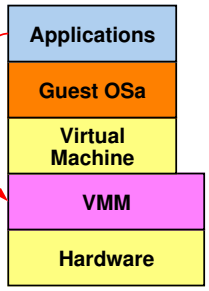
- Execute a *privileged* instruction
  - e.g., "trap" (due to "fork")



VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (due to "fork")

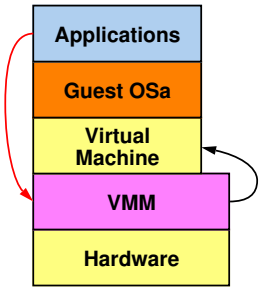
the VMM is invoked
- the VMM figures out which VM is currently executing



VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (due to "fork")

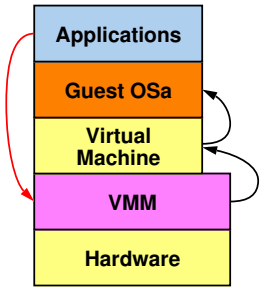
the VMM is invoked
- the VMM figures out which VM is currently executing
- VMM then asks the corresponding VM to *deliver* the trap to its OS



VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (due to "fork")

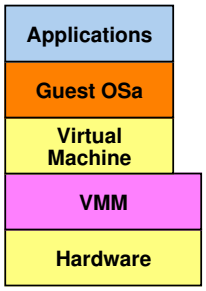
the VMM is invoked
- the VMM figures out which VM is currently executing
- VMM then asks the corresponding VM to *deliver* the trap to its OS



- Note that without VM, the application will simply traps into the OS directly
  - now it's a lot more involved
- 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?

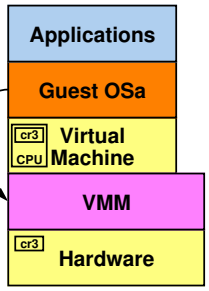
VMM Operations

- A *sensitive* instruction must execute in the privilege mode *(in the kernel)*
  - a *sensitive* instruction is an instruction that will change something in the *hardware/processor*
    - e.g., insturctions that change the mapping of virtual to real memory



VMM Operations

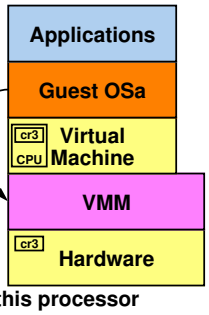
- A *sensitive* instruction must execute in the privilege mode *(in the kernel)*
  - a *sensitive* instruction is an instruction that will change something in the *hardware/processor*
    - e.g., insturctions that change the mapping of virtual to real memory



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

VMM Operations

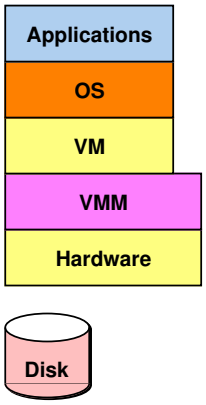
- A *sensitive* instruction must execute in the privilege mode *(in the kernel)*
  - a *sensitive* instruction is an instruction that will change something in the *hardware/processor*
    - e.g., insturctions that change the mapping of virtual to real memory



- 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 definition may be more useful

VMM Operations

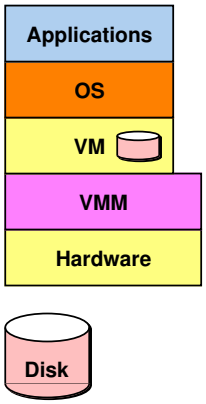
- What about I/O?
  - e.g., `read()`



VMM Operations

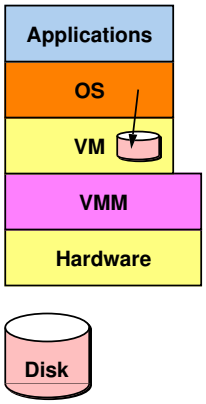
- What about I/O?
  - e.g., `read()`

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



VMM Operations

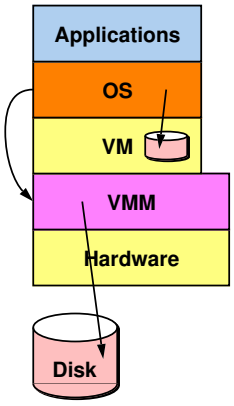
- What about I/O?
  - e.g., `read()`
  - `read()` eventually reaches the OS
  - the OS asks for a block on the virtual disk



VMM Operations

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

traps into VMM
- the VMM *emulates* the instruction (i.e., translates it into a request for the real disk)
- there's really *no* disk in the VM



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

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

Processor Virtualization Requirements

- 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

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 privileged mode, what's to prevent it from changing how the memory-mapping resources that have been set up?
- Need to distinguish between *sensitive instructions* and *privileged instructions*

Processor Virtualization Requirements

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

Processor Virtualization Requirements

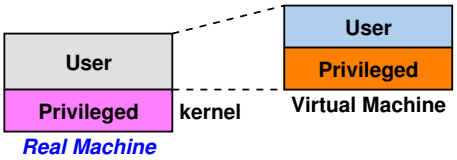
- [Popek and Goldberg, 1974] *proved* that the *sufficient condition* to be able to construct a virtual machine is simply the following:
  - a computer's set of *sensitive instructions* is a *subset* of its *privileged instructions*
  - i.e., if you execute a sensitive insturction in user mode, you will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

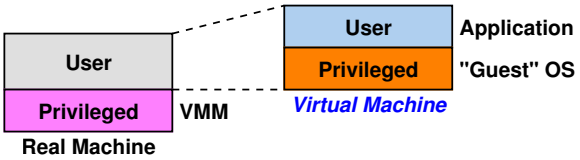
- Two execution modes
  - supervisor and problem (user)
  - all sensitive instructions are privileged instructions
- Memory is protectable: 2KB granularity
- All interrupt vectors and the clock are in first 512 bytes of memory
- I/O done via channel programs in memory, initiated with privileged instructions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

Actions on Virtual 360



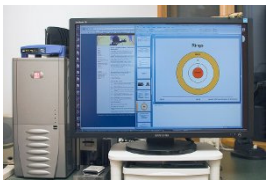
	Virtual user mode	Virtual privileged mode
non-sensitive instruction	executes fine	executes fine
errant instruction	traps to VMM; VMM causes trap to occur on Guest OS	traps to VMM; VMM causes trap to occur on Guest OS
sensitive instruction	traps to VMM; VMM causes trap to occur on Guest OS	traps to VMM; 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?

Virtual Machines Part 2: Now

≠



How They Are Different

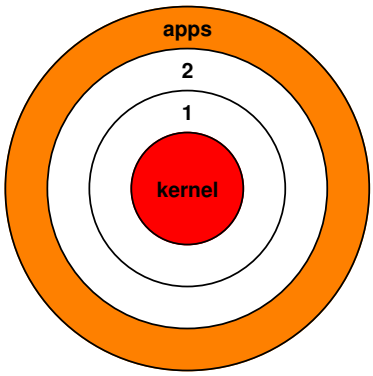
IBM 360

- Two execution modes
  - supervisor and problem (user)
  - *all* sensitive instructions are privileged instructions
- Memory is protectable: 2k-byte 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

Intel x86

- Four execution modes
  - rings 0 through 3
  - *not all* sensitive instructions are privileged instructions
- Memory is protectable: segment system + virtual memory
- Special register points to interrupt table
- I/O done via *memory-mapped* addresses
- Virtual memory is standard

Rings



- An x86 processor can be in one of 4 *modes*

A Sensitive x86 Instruction

- `popf`
  - pops flags (word) off stack, setting processor flags according to word's content
    - sets all flags if in ring 0
      - including interrupt-disable flag
    - just some of them if in other rings
      - ignores interrupt-disable flag
  - bad news: if invoked in *user* mode, does *not* cause a *trap!*
    - therefore, this instruction will execute differently in the OS when it's running on top of a VM (as compared to running on a real machine)
      - since the OS is running in user mode under the virtual memory scheme
    - this is one of the major problem to virtualize x86 systems
    - there is another major problem (related to device I/O)

What to Do?

- *Binary rewriting*
  - rewrite kernel binaries of guest OSes
    - replace sensitive instructions with hypercalls
    - do so dynamically (i.e., *dynamic* binary rewriting)
      - VMware does this
    - no need to modify guest OS
- *Hardware virtualization*
  - fix the hardware so it's virtualizable
- *Paravirtualization*
  - virtual machine differs from real machine
    - provides more convenient interfaces for virtualization
    - *hypervisor* interface between virtual and real machines
      - please note that some would use the term *"hypervisor"* to simply mean *"VMM"* (even without paravirtualization)
    - guest OS *source code* is *modified* (and recompiled)

Binary Rewriting

- Privilege-mode code run via binary translator
  - guest OS is unmodified
  - replaces sensitive instructions with hypercalls
  - translated code is cached
    - usually translated just once
  - *VMWare*
  - U.S. patent 6,397,242
- VirtualBox appears to do something similar to VMWare
  - see https://www.virtualbox.org/manual/ch10.html#idp58764736 for more details

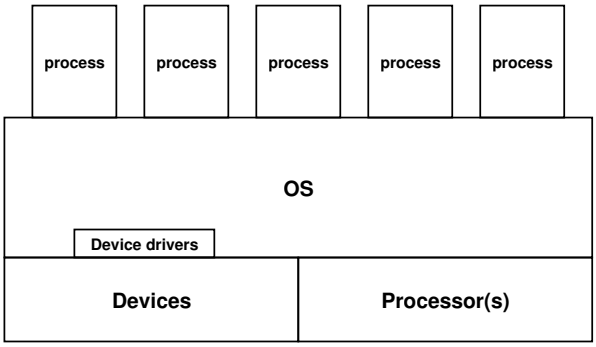
Fixing the Hardware

- Intel Vanderpool technology: VT-x
  - new processor mode
    - "ring -1"
      - *root* mode
      - other modes are *non-root*
  - certain operations and events in non-root mode cause *VM-exit* to root mode
    - essentially a hypercall
    - code in root mode specifies which operations and events cause VM-exits
      - e.g., `popf`, page fault
  - non-VMM OSes must not be written to use root mode!

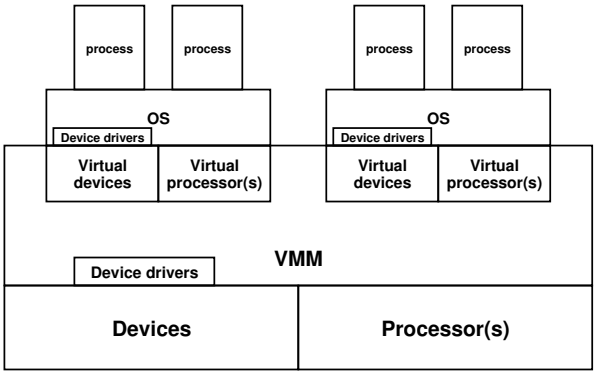
I/O Virtualization

- Channel programs were generic for IBM 360
  - can be emmulated in the VMM
- I/O via memory-mapped registers is not
  - lots and lots and lots of device drivers
  - must VMM handle all of them?

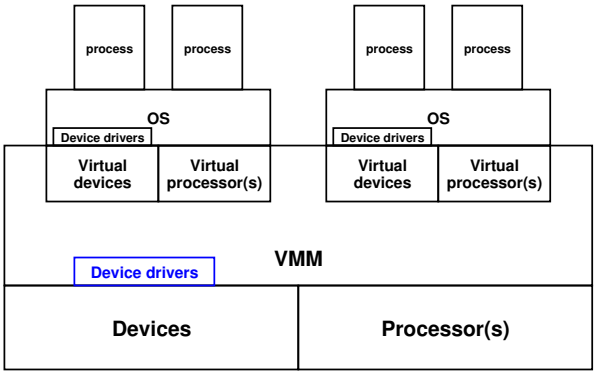
Real-Machine OS Structure



On a Virtual Machine ...

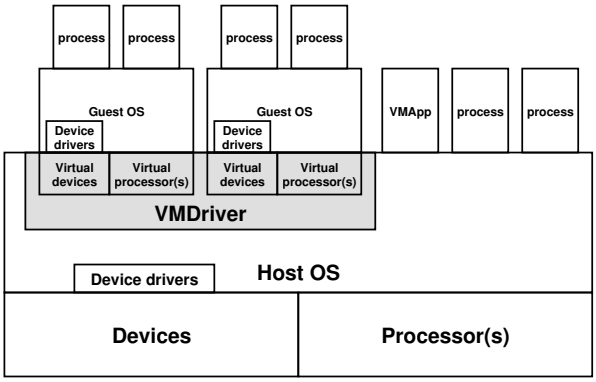


On a Virtual Machine ...



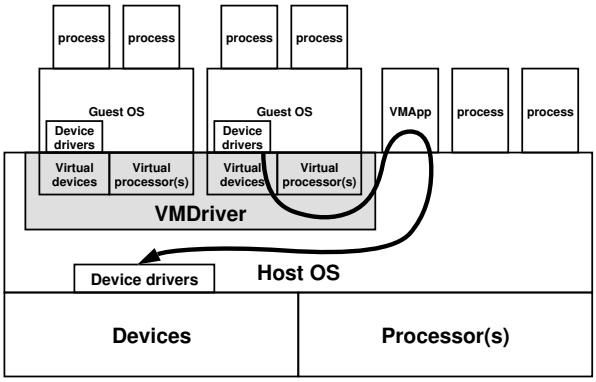
- More suitable for server machines (higher performance)
  - problem: who is going to write device drivers for VMM in lower-end machines?

VMware Workstation - Host/Guest Model



- VMware's solution is to use a host/guest model
  - VMDriver takes the place of the VMM
  - plenty of device drivers already available on host OS

VMware Workstation - Host/Guest Model

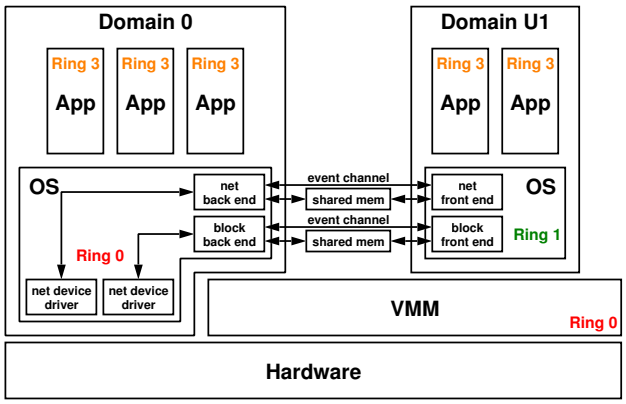


- More suitable for "workstations" - more variety of devices
  - *convenience* over *performance*

Paravirtualization

- Sensitive instructions replaced with hypervisor calls
  - traps to VMM
- Virtual machine provides higher-level device interface
  - guest machine has *no device drivers*
    - OS is changed already, might as well change I/O, if there are sufficient benefits

Xen

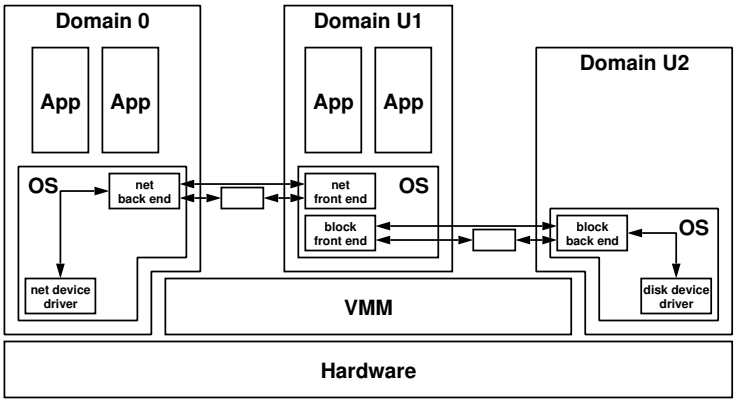


- In a way, the Domain 0 OS is like the Host OS in VMware
  - it directly talks to the hardware

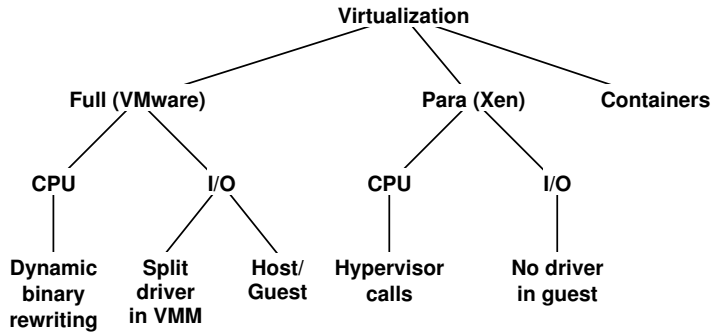
Additional Applications

- Sandboxing
  - isolate web servers
  - isolate device drivers
- Migration
  - VM not tied to particular hardware
  - easy to move from one (real) platform to another

Xen with Isolated Driver

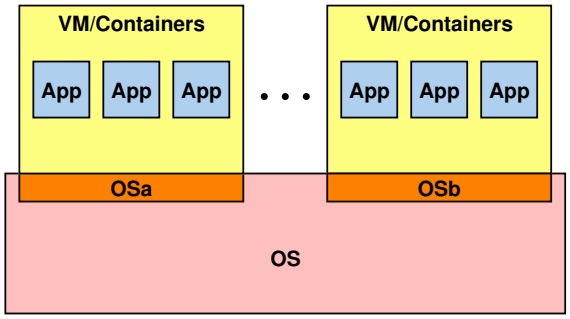


Summary



One More Kind Of Virtualization

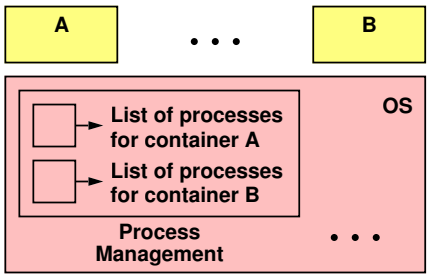
- Containerized OS (or OS Containers)
  - not covered in textbook



  - the OS provides the abstraction that each container runs on top of a separate OS (but there is really only one OS)
  - e.g., OpenVZ, Linux Containers (LXC), Docker

Containerized OS

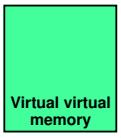
- Within the OS, the management of resources for each container is separated
  - e.g., processes for container A is kept separate from processes for container B



- Others may consider this "virtual machine", but we shouldn't
  - because "guest OS" does not run in user space

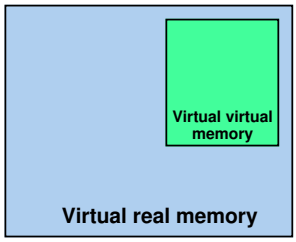
7.2.6 Virtualizing Virtual Memory

Virtual Machines Meet Virtual Memory



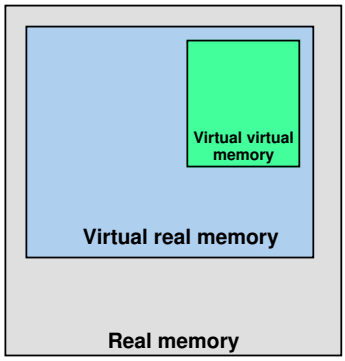
- A user process thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory*

Virtual Machines Meet Virtual Memory



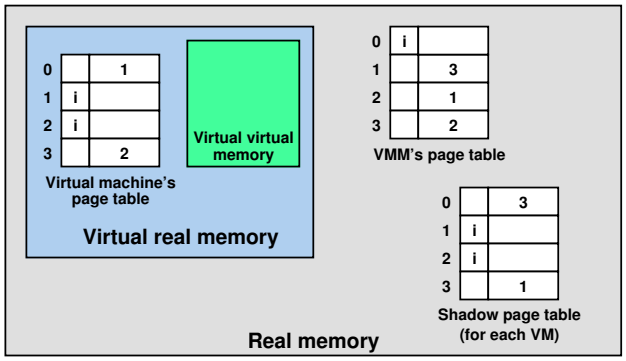
- A user process thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory*
- The OS in a VM thinks it's managing real memory
  - but it's really dealing with *virtual real memory*

Virtual Machines Meet Virtual Memory



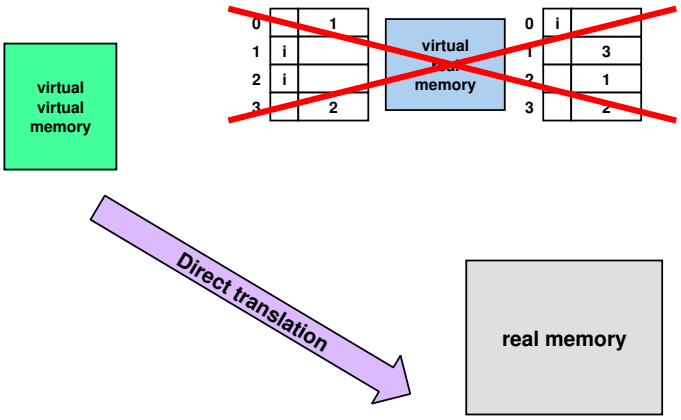
- A user process thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory*
- The OS in a VM thinks it's managing real memory
  - but it's really dealing with *virtual real memory*
- VMM needs to manage real memory
  - how can we *virtualize virtual memory*?

Virtual Machines Meet Virtual Memory

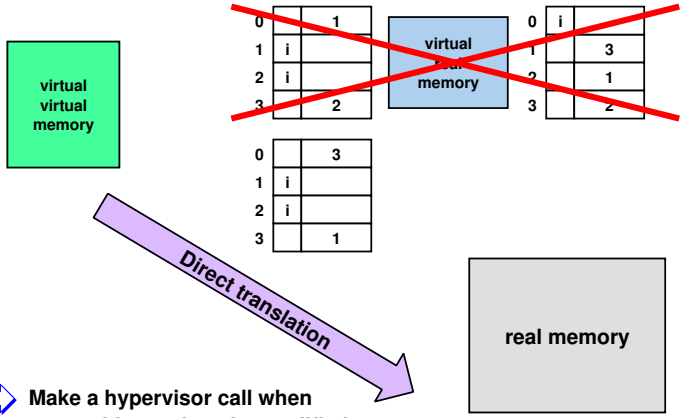


- When a VM changes its page table, VMM must update the corresponding *Shadow Page Table*
  - main problem: poor performance

Solution 1: Paravirtualization to the Rescue

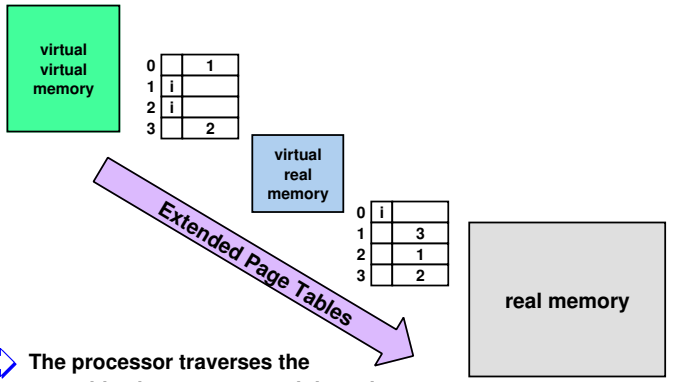


Solution 1: Paravirtualization to the Rescue



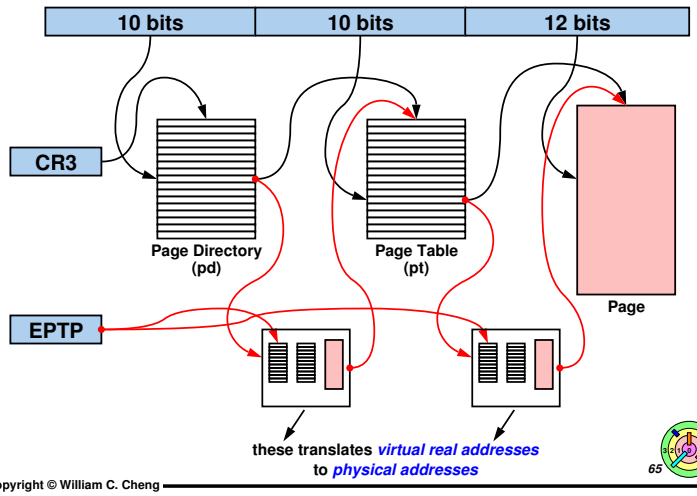
- Make a hypervisor call when page table needs to be modified
  - helps a bit, but not much faster

Solution 2: Hardware to the Rescue



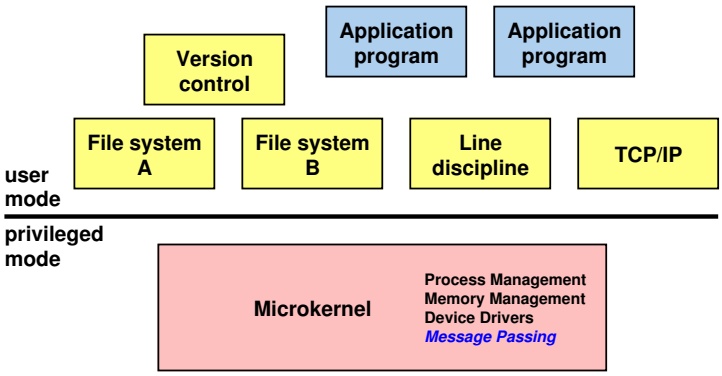
- The processor traverses the two tables in sequence and does the conversion all by itself

x86 Paging with EPT



Microkernels (Back to Section 4.2)

OS Services as User Apps



Why?

- It's cool ...
- Assume that OS coders are incompetent, malicious, or both ...
  - OS components run as protected user-level applications
- Extensibility
  - easier to add, modify, and extend user-level components than kernel components

Implementation Issues

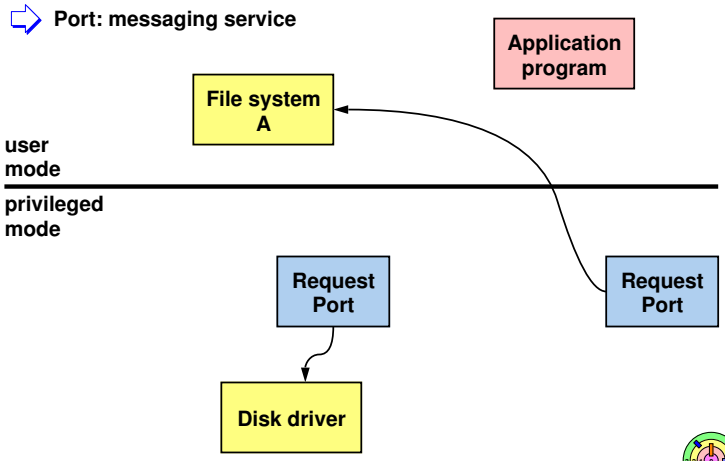
- How are modules linked together?
  - e.g., how would you implement `read()/write()`?
    - can't use system calls any more!
  - e.g., which file system supports `read()/write()`?
- How is data moved around efficiently?

Mach

- Developed at CMU, then Utah
- Early versions shared kernel with Unix
  - basis of NeXT OS
- Later versions still shared kernel with Unix
  - basis of OSF/1
    - basis of Macintosh OS X
- Even later versions actually functioned as working microkernel
  - basis of GNU/HURD project
    - HURD: HIRD of Unix-replacing daemons
    - HIRD: HURD of interfaces representing depth

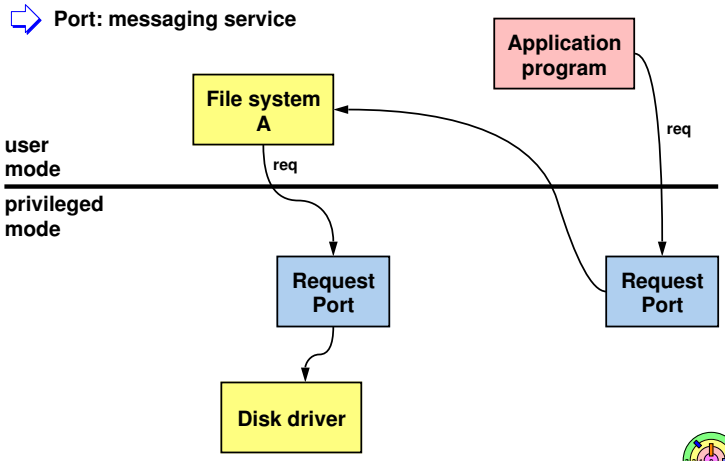
Example

- Port: messaging service



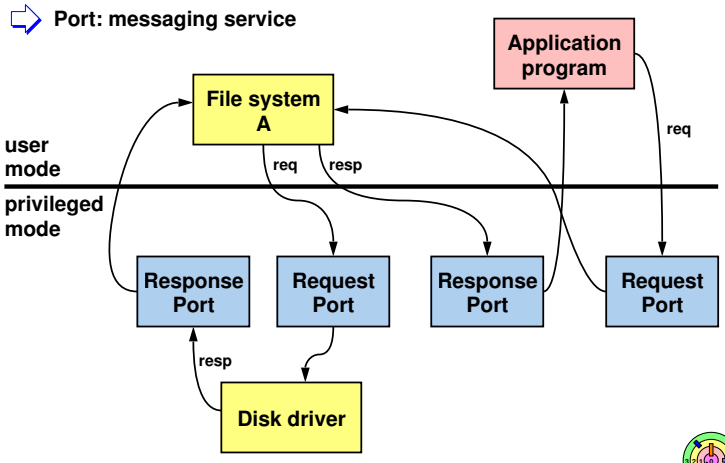
Example

- Port: messaging service



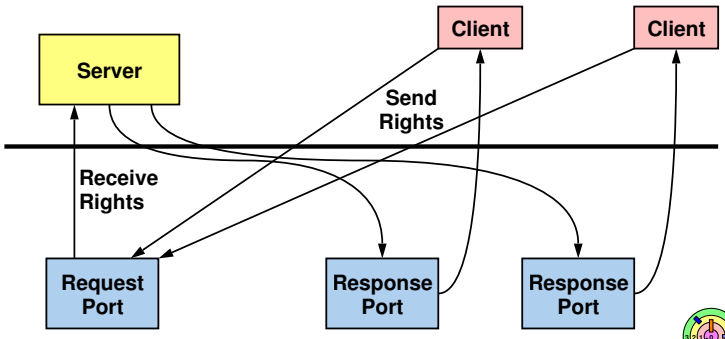
Example

- Port: messaging service



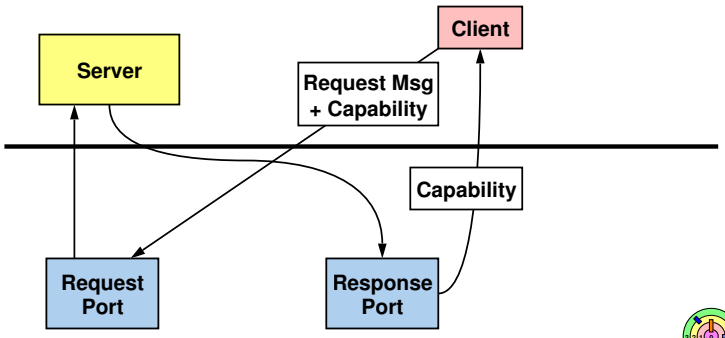
Mach Ports (1)

- Linkage construct



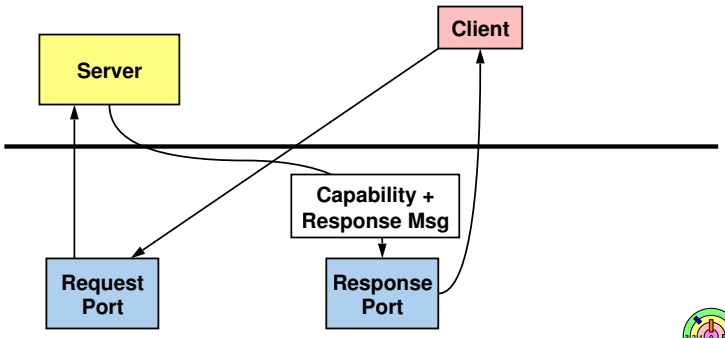
Mach Ports (1)

- Communication construct
  - client create *response port* and *capability* (like a key) to send data through it
    - include capability in the request message to server



Mach Ports (1)

- Communication construct
  - client create *response port* and *capability* (like a key) to send data through it
    - include capability in the request message to server



RPC

- Ports used to implement *remote procedure calls*
  - communication across process boundaries
  - if procedures are on same machine ...
    - local RPC

Successful Microkernel Systems

- ?
- ?
- ...

Attempts

- Windows NT 3.1
  - graphics subsystem ran as user-level process
  - moved to kernel in 4.0 for performance reasons
- Macintosh OS X
  - based on Mach
  - all services in kernel for performance reasons
- HURD
  - based on Mach
  - services implemented as user processes
  - no one uses it, for performance reasons ...