

# Virtualization

## Virtualization Introduction

### \* Virtualization Examples

1. Memory systems
2. Data centers
3. JVM
4. VirtualBox
5. IBM VM/370
6. Google Glass
7. Cloud computing
8. Dalvik
9. VMWare Workstation
10. The movie "Inception"

\* Platform Virtualization: Instead of running on actual hardware, can we run on virtual hardware without loss of performance?

\* Utility Computing: Sharing hardware resources across different communities

- This means different users can have different OS needs
- Resource usage typically isn't constant and doesn't require 100% of system resources at a time
- AWS, Microsoft Azure, GCP
- Virtualization is the logical extension to the extensibility that was implemented by Exokernel, SPIN, and L3
  - + Entire OS though, as opposed to individual services

\* Hypervisors: Arbitrates resources across different client OS and protects them from one another

- Running multiple OS on same hardware resources
- Also called virtual machine managers (VMM)
- Two types: Native (bare metal) and hosted
  - + Native: Hypervisor running on hardware, guest OS runs on top of hypervisors
  - + Hosted: Run on top of host OS as application process, not hardware
- VMWare Workstation and VirtualBox are hosted hypervisors
- Native hypervisors provide minimal interference with guest OS

\* History of Virtualization

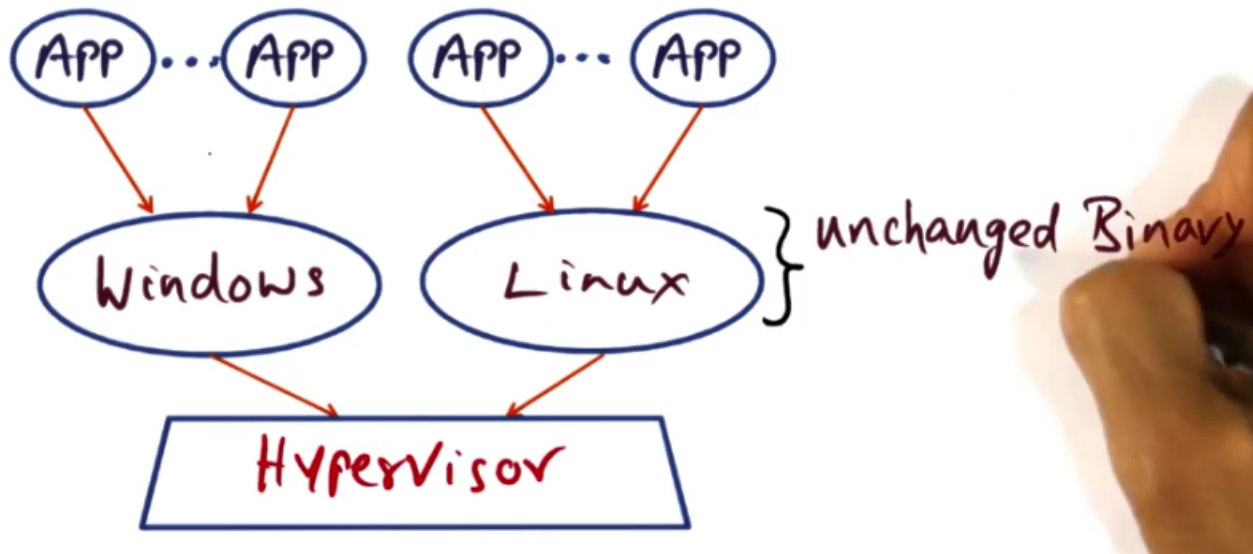
- IBM VM 370: Provide illusion that the entire computer belonged to a single user, even though they were sharing it (70s)
- Microkernels: Kernel provides minimal functionality, everything else sits on top of this microkernel (late 80s and 90s)
- Extensibility of OS: Library OS running on top of minimal kernel (90s)
- SIMOS: Basis for VMWare, idea of virtual OS (late 90s)
- Xen and VMWare: Virtualization as we know today (early 2000s)
- Today: Virtualization is used widely in data centers

\* Full Virtualization: Leave the OS untouched so you can run an unchanged binary of the OS on top of the hypervisor

- Guest OS will be running as a user-level process, not the same level of privilege as the OS would be if running on bare metal
- Guest OS will be unaware that it doesn't have the same privileges
  - + Binary is unchanged
- Trap and emulate: When a guest OS tries to execute a privileged instruction, the hypervisor receives the trap and emulates the intended functionality
- Some privileged instructions may fail; guest OS will think it

succeeded

- + Binary translation: Hypervisor looks for these instructions and ensures the instructions are dealt with carefully through binary editing strategies
- VMWare Workstation is an example of full virtualization

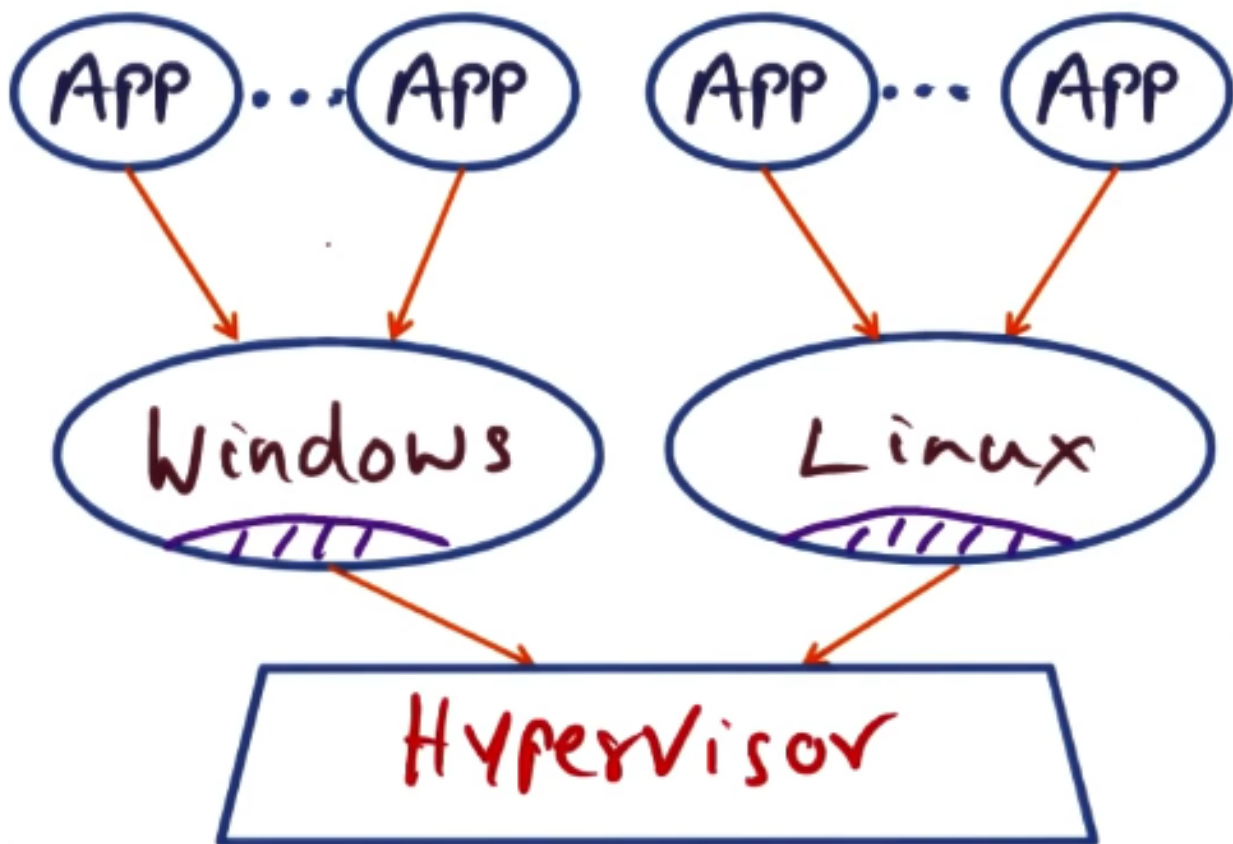


Full Virtualization

\* Para Virtualization: Modify source of guest OS so they are aware they're running in a virtualization setting

- Can introduce optimizations, such as giving guest OS access to hardware resources
- Applications seem same interfaces an OS running on bare metal would provide
- Not fully virtualized, only partially
- Less than 2% of guest OS code needs to be modified
- Xen architecture is an example of para virtualization
  - + Showed this by proof of construction

Subsystem	Linux	XP
Architecture Independent	78	1299
Virtual Network Driver	484	0
Virtual Block Device Driver	1070	0
Xen Specific	1363	3321
Total	2995	4620
% Total Codebase	1.36	0.04



## Para Virtualization

---

### \* Big picture

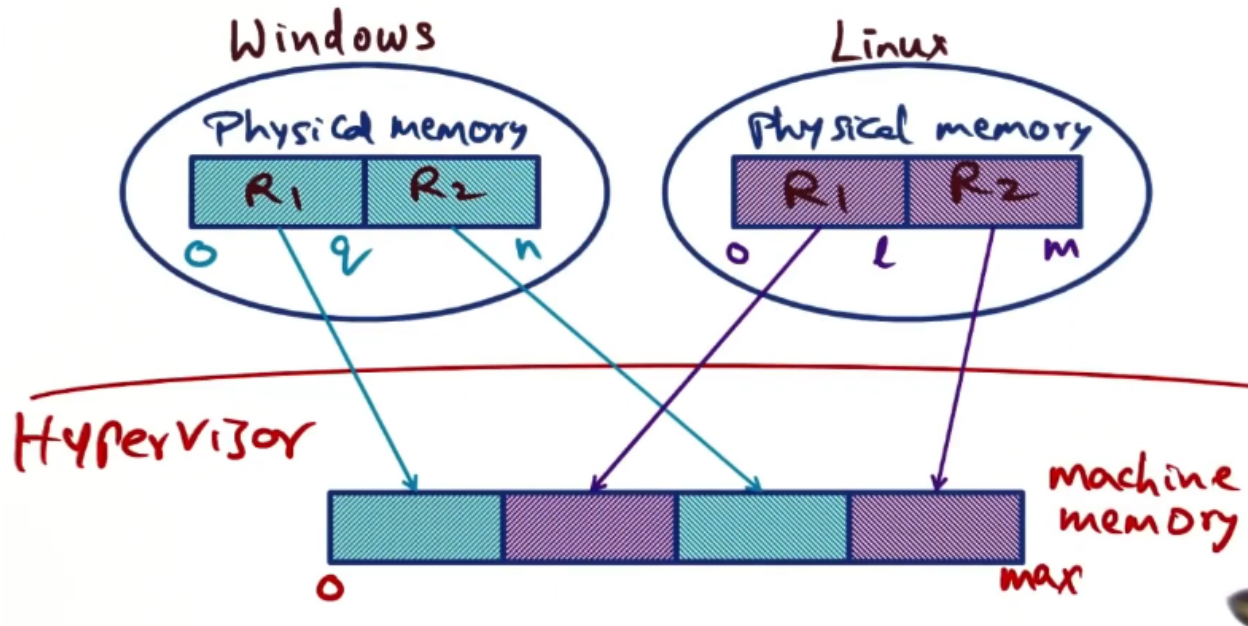
- Virtualize hardware
  - + Memory hierarchy
  - + CPU
  - + Devices
- Effect data and control between guests and hypervisor

## Memory Virtualization

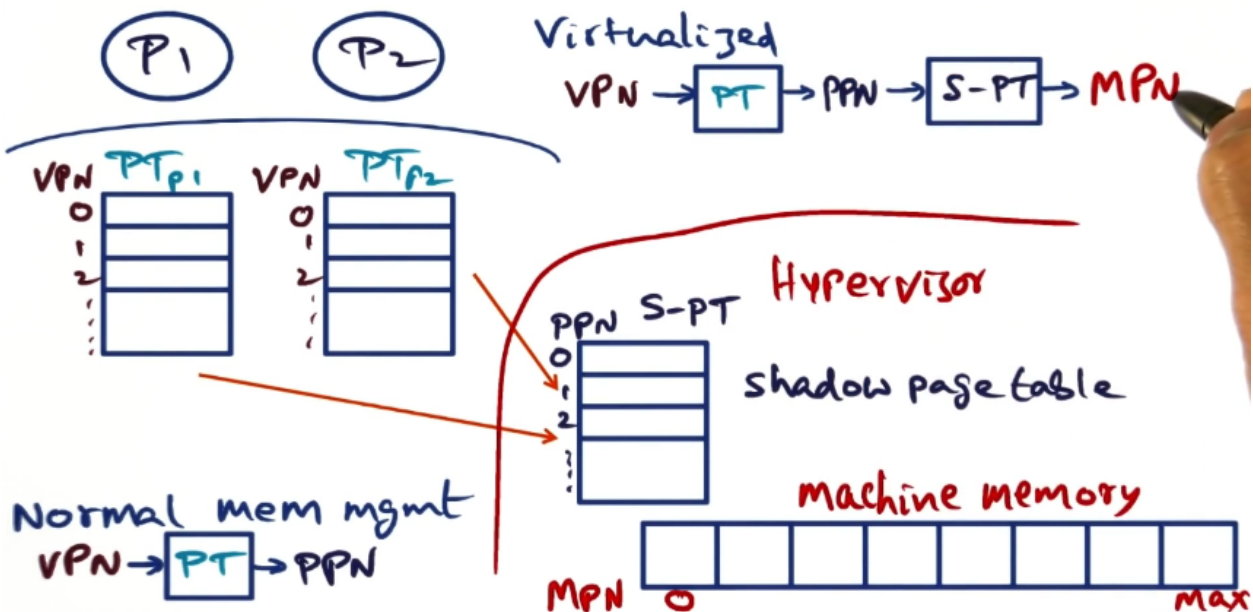
### \* Memory Management and the Hypervisor

- Hypervisor sits between guest OS and hardware
- Each process (guest OS) is in its own protection domain
  - + OS maintains separate page table for each process
- Hypervisor isn't aware of the different processes running in each guest OS, so it doesn't know about their individual page tables
- Guest OS assumes its physical memory is contiguous, but hypervisor does not provide this guarantee (not all memory can start at 0)
- Normally, a virtual page number is mapped to a physical frame number through the page table
  - + When virtualized, another level of indirection is required
  - + Hypervisor tracks physical frame numbers of guest OS and how they are mapped to machine memory

- + Physical page number to machine page number is kept in the "shadow page table"
- In a fully virtualized system, the shadow page table is kept by the hypervisor
- In a para virtualized system, the shadow page table is kept by the guest OS (could still be hypervisor though)

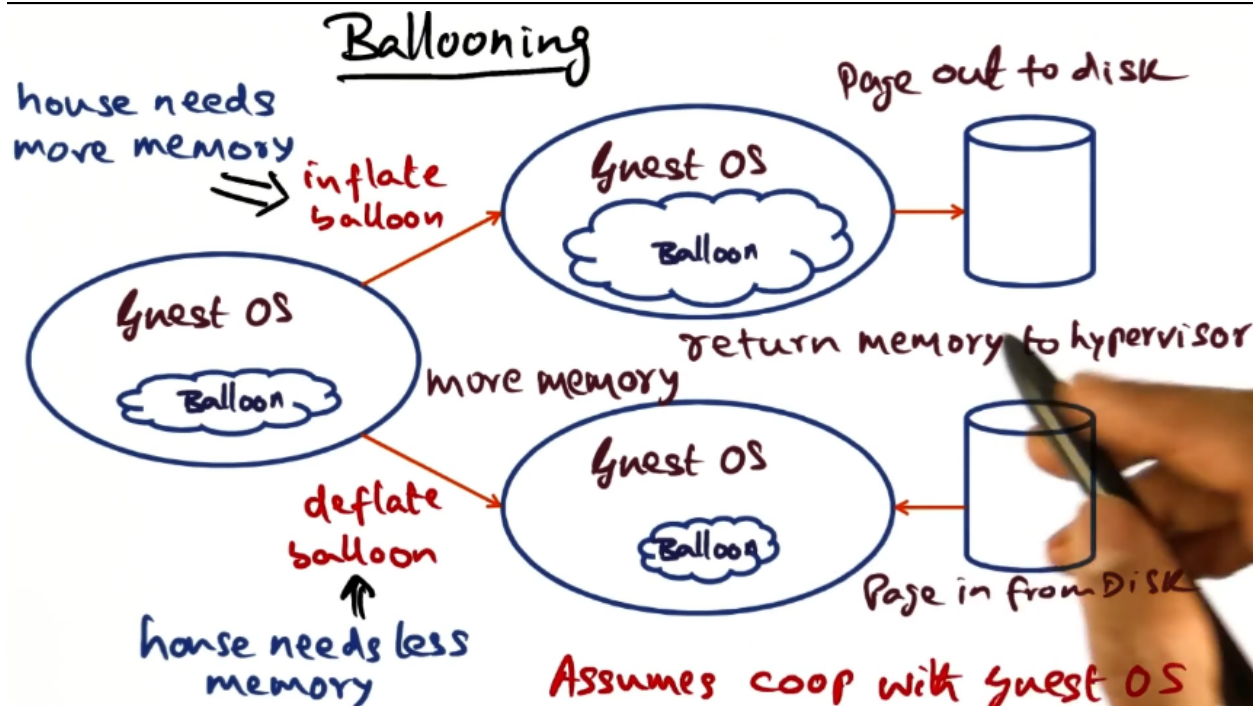


Memory Manager (Macro view)



Memory Manager (Micro view)

- \* Shadow Page Table
  - In many architectures (x86) CPU uses page table for address translation
    - + Both TLB and page table are data structures used for address translation
  - This means the hardware page table is really the shadow page table
- \* Efficient Mapping (Full Virtualization)
  - Typically, the guest OS would map VPN -> PT -> PPN, then the hypervisor would map PPN -> SPT -> MPN
  - For efficiency, is there a way to eliminate the first level of indirection?
  - Any time guest OS tries to update page table/TLB, it causes a trap
    - + Then, the hypervisor can update its shadow page table
    - + This means it can all be done at once
    - + Translations installed into TLB/hardware page table
    - + Hypervisor can directly translate VPN to MPN through TLB and hardware page table
- \* Efficient Mapping (Para Virtualization)
  - Shift burden to guest OS
    - + Maintain contiguous "physical memory"
    - + Map to discontiguous hardware pages
  - Xen provides a set of "hypercalls" to tell the hypervisor about changes to the hardware page table
    - + Create page table (on process initialization, allocate page frame)
    - + Switch page table (on context switch)
    - + Update page table (on page fault, VPN->PFN needs to change)
- \* Dynamically Increasing Memory
  - Hypervisor must be able to allocate machine memory to guest OS on demand
  - When on guest OS needs more memory, it issues a request to the hypervisor
    - + Hypervisor can reclaim memory from another guest OS, but this can result in undefined behavior for the other guest
    - + Can request that the other guest gives up some memory voluntarily
- \* Ballooning
  - Special device driver installed in every guest OS (even in fully virtualized system) called balloon
  - When the host needs more memory, it can "inflate" the balloon device driver, causing it to request memory from guest OS
    - + If guest OS memory is full and balloon asks for more memory, it will page out memory to disk
    - + Then, the excess memory claimed by the balloon is returned to the hypervisor
  - When the host no longer needs the memory, it can contact the balloon driver to "deflate"
    - + This will release the pages in the guest OS and it will page in relevant pages from disk so its processes will have more resources
  - Assumes cooperation with guest OS
  - Analogous to airlines selling more seats than they have



Ballooning

#### \* Sharing Memory Across Virtual Machines

- Desire to share memory between virtual machines to optimize usage if two guest OS's are using the same data
- Can't affect integrity of the guest OS
- If two guests are running the exact same process, we can set their page tables to point to the same physical memory location
  - + Avoids duplications
  - + Particularly true for core pages of a process (immutable)
- On write, hypervisor can make a copy of the pages to prevent one guest from corrupting the memory of another (copy on write)

#### \* VM Oblivious Page Sharing

- Hypervisor keeps a hash table of the content of the machine pages
- Creates a hash of the content and check the "hint frame" (hash table) to see if the content is already in memory
- Doesn't guarantee that the contents are identical, just a hint
  - + The page may have been updated since the hint frame was created
- Once there's a match, we do a full comparison to guarantee that the pages are identical
- Modify the physical page number to machine page number for guest
  - + Update reference counter in hash table
  - + Mark as "copy on write," meaning they can only share the frame as long as they're only reading from it
  - + Frees up a machine page, but this is an expensive operation
- Scanning pages is performed as a background activity of the server
- Works in both fully virtualized and para virtualized settings

#### \* Memory Allocation Policies

- Pure share based approach: Resources given according to payment
  - + Could lead to hoarding of resources (wasting)
- Working-set based approach: If VM needs more resources, it gets them;

when it no longer needs them, it frees them

- Dynamic idle-adjusted shares approach: Tax idle pages more than active pages
  - + 0% means idle pages aren't reclaimed
  - + 100% means all idle resources are reclaimed (use it or lose it)
  - + Somewhere in between is ideal (50-75%); used by VMWare ESX server
- Reclaim most idle memory while simultaneously allowing for sudden increases in the working set

## CPU and Device Virtualization

- \* CPU Virtualization Introduction
  - Provide the illusion of owning the CPU (not aware of other guests)
    - + Provided by hypervisor at OS granularity
  - Handle program discontinuities
- \* CPU Virtualization
  - Guest OS is already multiplexing CPU across processes, hypervisor needs to multiplex CPU across OS's
    - + Proportional share: Amount of CPU depends on agreement guest OS has with hypervisor
    - + Fair share: Amount of CPU is equal for each guest OS
  - When an external interrupt occurs, hypervisor may need to steal cycles from currently running guest and give them back later
- \* Delivering Events to Guest OS (common to full and para)
  - While an OS is running, everything should be happening at hardware speeds (no overhead due to hypervisor)
  - However, interrupts, syscalls, exceptions, and page faults while require OS intervention
    - + Hypervisor delivers events as software interrupts to guest OS
    - + Handling some of these events may require privileged access, but guest OS is running in user mode
    - + Some privileged instructions fail silently when executed at the user level instead of trapping (only issue in fully virtualized)
    - + Hypervisor must be aware of what instructions these are in the guest OS to handle appropriately
    - + Newer versions of Intel architecture provide support for virtualization
  - In a fully virtualized environment, communication between guest and hypervisor is implicit through traps
  - In a para virtualized environment, communication between guest and hypervisor is explicit through APIs

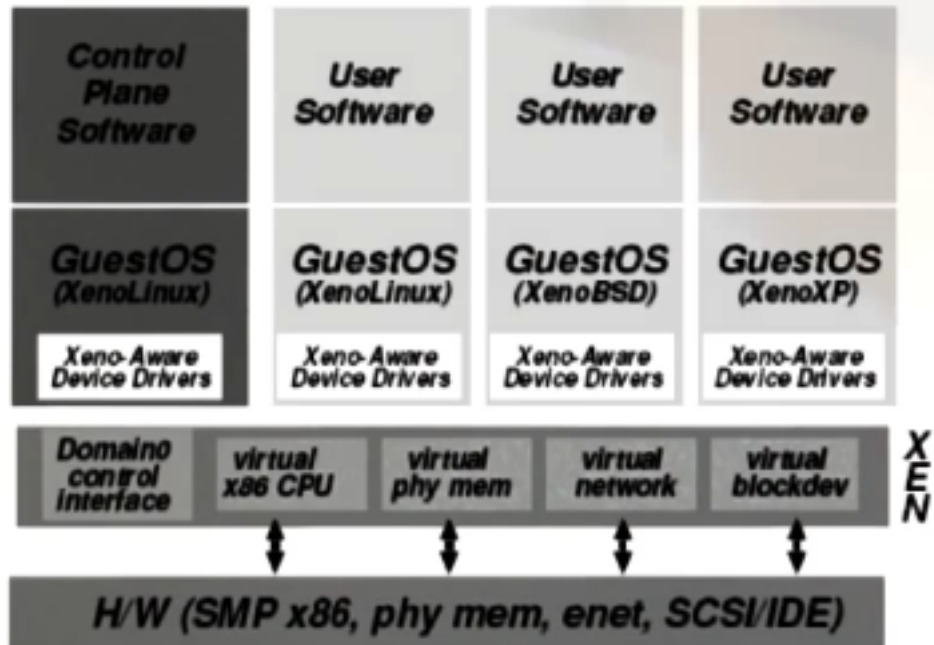
## Device Virtualization

- \* Full Virtualization
  - Guest OS thinks it owns devices; communication is handled through "trap and emulate" instructions to hypervisor
    - + Hypervisor must determine the legality of the I/O operation
- \* Para Virtualization
  - More opportunity for innovation
  - Interrupts go to hypervisor, transferred as events to guest OS
  - Hypervisor defines "hypercalls" (i.e., page table updates)
  - Hypervisor includes shared buffers to avoid multiple copying of data
- \* Control Transfer

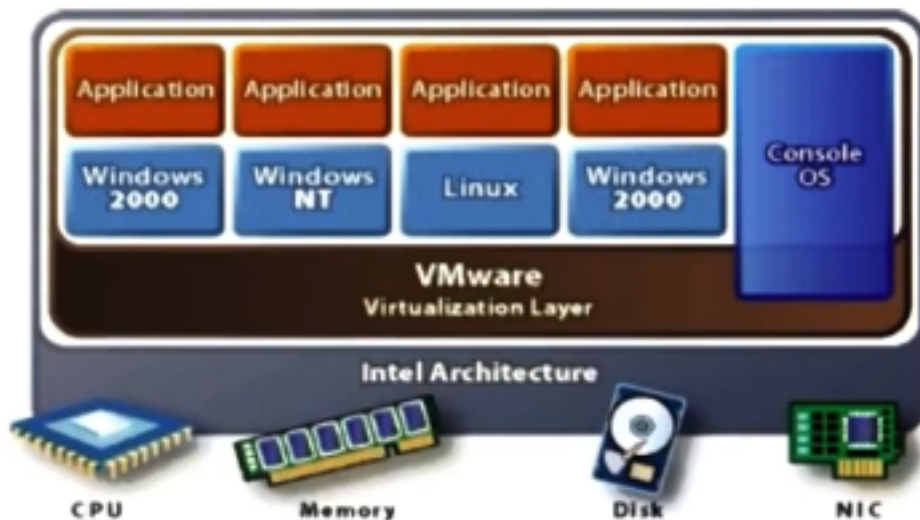
- Full virtualization
  - + Implicit (traps) in guest OS transfer control to hypervisor
  - + Software interrupts (events) in hypervisor transfer control to guest OS
- Para virtualization
  - + Explicit (hypercalls) in guest OS transfer control to hypervisor
  - + Software interrupts (events) in hypervisor transfer control to guest OS
  - + Guest has control via hypercalls on when event notifications are delivered (not possible in fully virtualized system)
- \* Data Transfer
  - Full virtualization
    - + All implicit
  - Para virtualization
    - + Explicit -> opportunity to innovate
    - + When an interrupt occurs, hypervisor must demultiplex the data that is coming from the device to the domains quickly
    - + Time issue: Which guest are CPU cycles "billed" to?
    - + Space issue: How are buffers allocated and managed?
  - Xen's Asynchronous I/O rings
    - + Data structure shared between guest and hypervisor for communication
    - + I/O ring is a set of descriptors to I/O devices
    - + Guest OS puts requests in the ring, hypervisor puts responses back
    - + Guest OS has request producer pointer that Xen can read from
    - + Hypervisor has request consumer pointer that is private to Xen
    - + Hypervisor has response producer pointer that guest can read from
    - + Guest OS has response consumer pointer that is private to guest
    - + Requests/responses have a unique ID to identify them
    - + All of this is done with pointer passing; fast
- \* Control and Data Transfer
  - Network transmit
    - + Guest OS puts transmit request in transmit ring through hypercalls
    - + Guest puts pointers to data buffers in request (no copying, page is pinned to prevent it from being paged out)
    - + Xen uses round robin packet scheduler
  - Network receipt
    - + Exchange received packet for guest OS page
    - + No copying; Xen writes to the preallocated buffers
    - + Xen can swap machine page for a page the guest already owns instead of using a preallocated buffer
- \* Disk I/O Virtualization
  - Every VM has a ring for disk I/O (private to each guest)
  - No copying into Xen
  - Request from competing domains may be reordered to make the I/O throughput more efficient
  - Xen provides a reorder barrier for guest OS semantics (disallows any reordering)
- \* Measuring Time
  - Resources shared by many clients; need a mechanism to bill them
    1. CPU usage
    2. Memory usage
    3. Storage usage
    4. Network usage
- \* Xen and Guests



- Difference from extensible OS
  - + Focus on protection and flexibility
  - + Not at the granularity of individual applications, but at the granularity of entire operating systems



Xen (Para Virtualized)



Xen (Para Virtualized)