
# Security of Cloud Computing

Fabrizio Baiardi
f.baiardi@unipi.it

# Syllabus

- Cloud Computing Introduction
- Security
- Supporting Technologies
  - Virtualization Technology
  - Scalable Computing = Elasticity
- Security
  - New Threat Model
  - New Attacks
  - Countermeasures

# Virtualization

- **What is virtualization?**
- **Traditional virtualization techniques.**
- **Overview of Software VMM.**
- **Overview of Hardware VMM.**
- **Evaluation of VMMs.**

# Overview

Fundamental idea – abstract hardware of a single computer into several different execution environments

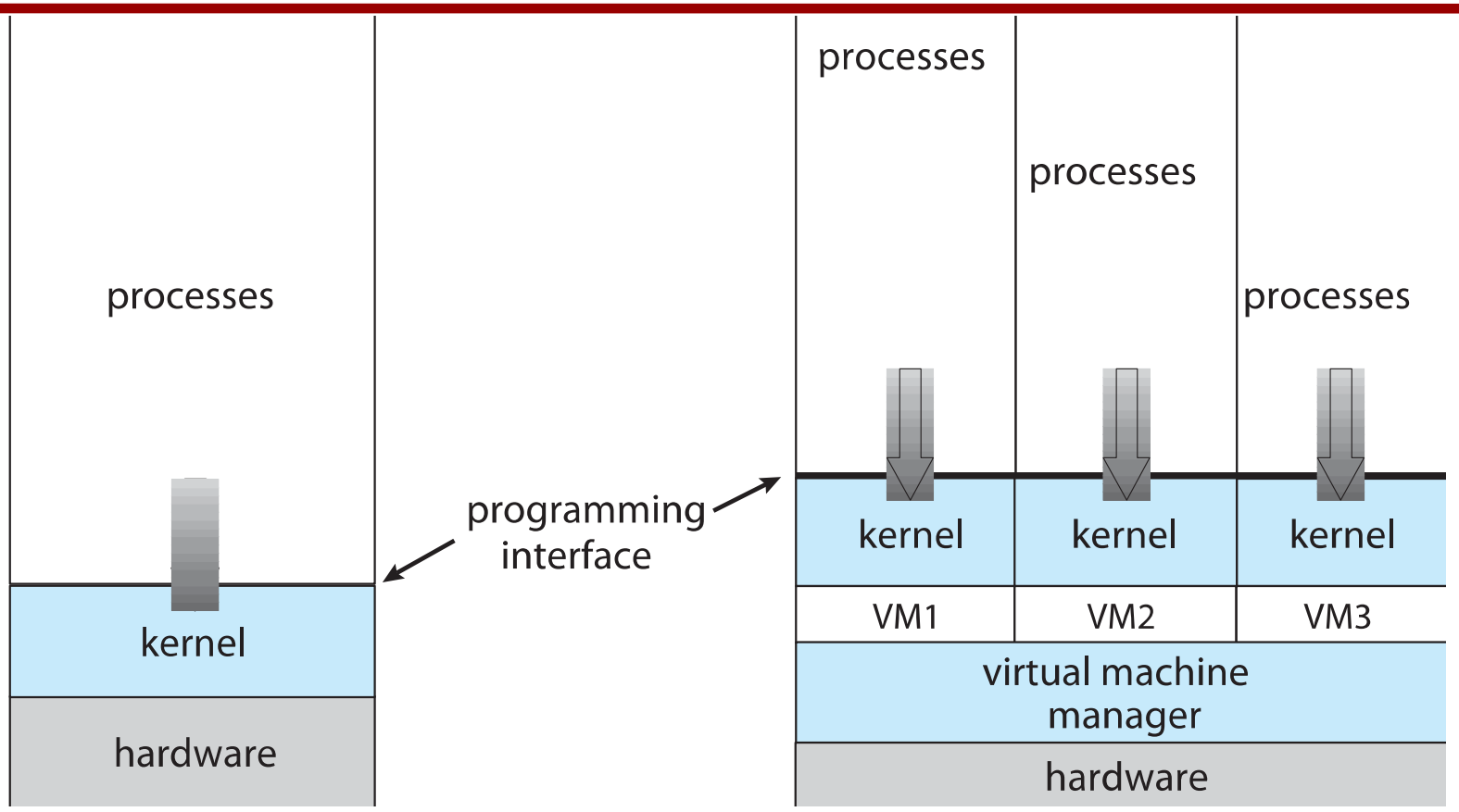
- Similar to layered approach
- But layer creates virtual system (**virtual machine**, or **VM**) on which operation systems or applications can run

Several components

- **Host** – underlying hardware system
- **Virtual machine manager** (**VMM**) or **hypervisor** – creates and runs virtual machines by providing interface that is ***identical*** to the host
  - (Except in the case of paravirtualization)
- **Guest** – process provided with virtual copy of the host
  - Usually an operating system

Single physical machine can run multiple operating systems concurrently, each in its own virtual machine

# System Models




(a) Nonvirtual machine

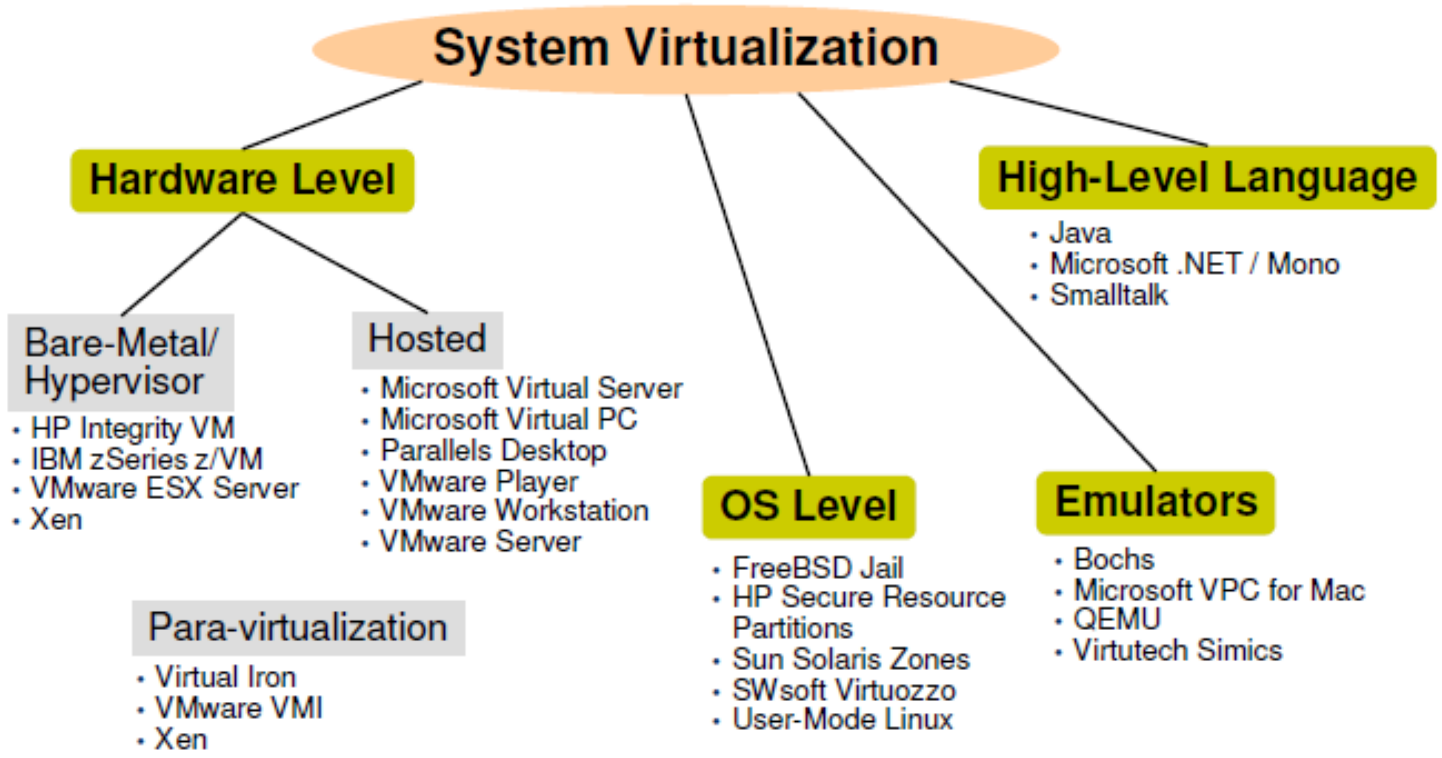
(b) Virtual machine

# Implementation of VMMs

Vary greatly, with options including:

- **Type 0 hypervisors** - Hardware-based solutions that provide support for virtual machine creation and management via firmware
  - IBM LPARs and Oracle LDOMs are examples
- **Type 1 hypervisors** - Operating-system-like software built to provide virtualization
  - Including VMware ESX, Joyent SmartOS, and Citrix XenServer
- **Type 1 hypervisors** – Also includes general-purpose operating systems that provide standard functions as well as VMM functions
  - Including Microsoft Windows Server with HyperV and RedHat Linux with KVM
- **Type 2 hypervisors** - Applications that run on standard operating systems but provide VMM features to guest operating systems
  - VMware Workstation and Fusion, Parallels Desktop, and Oracle VirtualBox

# Classification



# Implementation of VMMs (cont.)

Other variations include:

- **Paravirtualization** - Technique in which the guest operating system is modified to work in cooperation with the VMM to optimize performance
- **Programming-environment virtualization** - VMMs do not virtualize real hardware but instead create an optimized virtual system
- **Emulators** – Allow applications written for one hardware environment to run on a different hardware environment, such as a different CPU
- **Application containment** - Not virtualization at all but rather provides virtualization-like features by segregating applications from the operating system, making them more secure, manageable
  - Including Oracle Solaris Zones, BSD Jails, and IBM AIX WPARs

Much variation due to breadth, depth and importance of virtualization in modern computing

# Implementation of VMMs (cont.)

The Linux Containers (LXC) feature I

- a lightweight virtualization mechanism that does not require you to set up a virtual machine on an emulation of physical hardware.
- takes the cgroups resource management facilities as its basis and adds POSIX file capabilities to implement process and network isolation.
- You can run
  - a single application within a container (an application container) whose name space is isolated from the other processes on the system in a similar manner to a chroot jail.
  - a complete copy of the Linux operating system in a container (a system container) without the overhead of running a level-2 hypervisor such as VirtualBox.
  - the container is sharing the kernel with the host system, so its processes and file system are completely visible from the host. When you are logged into the container, you only see its file system and process space.

# Types of Virtual Machines and Implementations

Many variations as well as HW details

- Assume VMMs take advantage of HW features
  - HW features can simplify implementation, improve performance

Whatever the type, a VM has a lifecycle

- Created by VMM
- Resources assigned to it (number of cores, amount of memory, networking details, storage details)
- In type 0 hypervisor, resources usually dedicated
- Other types dedicate or share resources, or a mix
- When no longer needed, VM can be deleted, freeing resouces simpler, faster than with a physical machine install
- Can lead to **virtual machine sprawl** with lots of VMs, history and state difficult to track and manage

# Types of VMs – Type 0 Hypervisor

Old idea, under many names by HW manufacturers

- “partitions”, “domains”
- A HW feature implemented by firmware
- OS need to nothing special, VMM is in firmware
- Smaller feature set than other types
- Each guest has dedicated HW

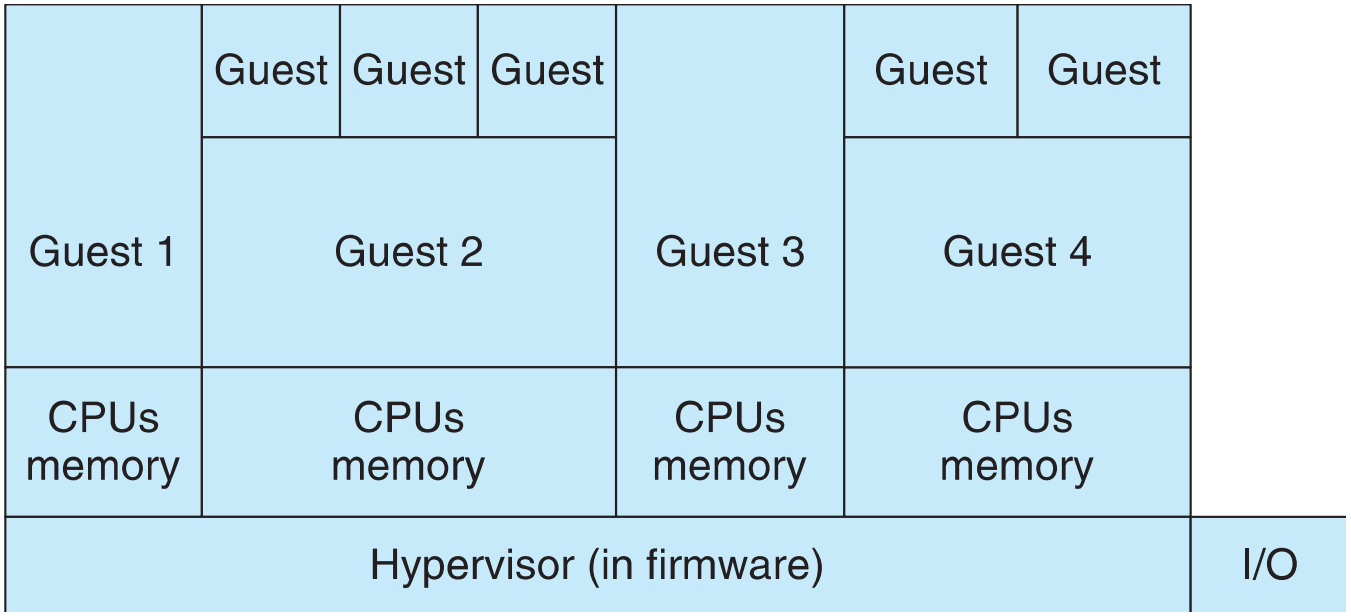
I/O a challenge as difficult to have enough devices, controllers to dedicate to each guest

Sometimes VMM implements a **control partition** running daemons that other guests communicate with for shared I/O

Can provide virtualization-within-virtualization (guest itself can be a VMM with guests

- Other types have difficulty doing this

# Type 0 Hypervisor



# Types of VMs – Type 1 Hypervisor

Commonly found in company datacenters

- In a sense becoming “datacenter operating systems”
  - Datacenter managers control and manage OSes in new, sophisticated ways by controlling the Type 1 hypervisor
  - Consolidation of multiple OSes and apps onto less HW
  - Move guests between systems to balance performance
  - Snapshots and cloning

Special purpose operating systems that run natively on HW

- Rather than providing system call interface, create run and manage guest OSes
- Can run on Type 0 hypervisors but not on other Type 1s
- Run in kernel mode
- Guests generally don’t know they are running in a VM
- Implement device drivers for host HW because no other component can
- Also provide other traditional OS services like CPU and memory management

# Types of VMs – Type 1 Hypervisor (cont.)

Another variation is a general purpose OS that also provides VMM functionality

- RedHat Enterprise Linux with KVM, Windows with Hyper-V, Oracle Solaris
- Perform normal duties as well as VMM duties
- Typically less feature rich than dedicated Type 1 hypervisors

In many ways, treat guests OSes as just another process

- Albeit with special handling when guest tries to execute special instructions

# Types of VMs – Type 2 Hypervisor

Less interesting from an OS perspective

- Very little OS involvement in virtualization
- VMM is simply another process, run and managed by host
  - Even the host doesn't know they are a VMM running guests
- Tend to have poorer overall performance because can't take advantage of some HW features
- But also a benefit because require no changes to host OS
  - Student could have Type 2 hypervisor on native host, run multiple guests, all on standard host OS such as Windows, Linux, MacOS

# Types of VMs – Paravirtualization

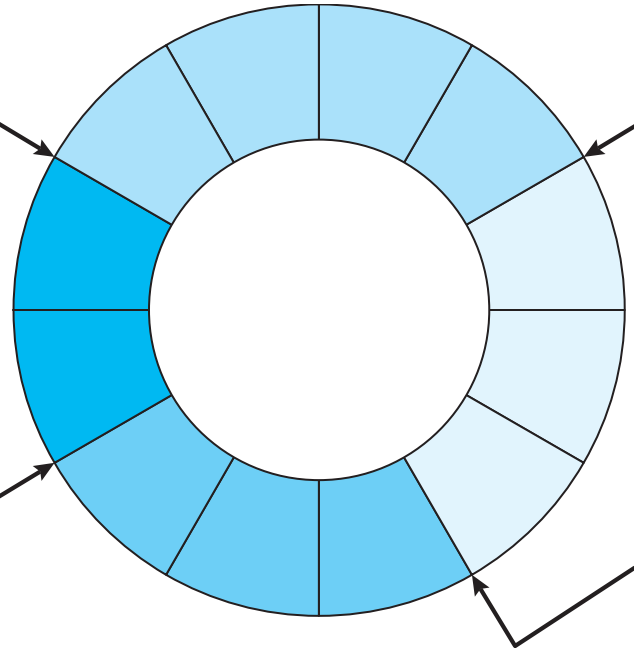
Does not fit the definition of virtualization – VMM not presenting an exact duplication of underlying hardware

- But still useful!
- VMM provides services that guest must be modified to use
- Leads to increased performance
- Less needed as hardware support for VMs grows

Xen, leader in paravirtualized space, adds several techniques

- For example, clean and simple device abstractions
  - Efficient I/O
  - Good communication between guest and VMM about device I/O
  - Each device has circular buffer shared by guest and VMM via shared memory

# Xen I/O via Shared Circular Buffer



*Request Consumer*
Private pointer
in Xen

*Request Producer*
Shared pointer
updated by guest OS

*Response Producer*
Shared pointer
updated by
Xen

*Response Consumer*
Private pointer
in guest OS

**Request queue** - Descriptors queued by the VM but not yet accepted by Xen

**Outstanding descriptors** - Descriptor slots awaiting a response from Xen

**Response queue** - Descriptors returned by Xen in response to serviced requests

**Unused descriptors**

# Benefits and Features

Host system protected from VMs, VMs protected from each other

- I.e. A virus less likely to spread
- Sharing is provided though via shared file system volume, network communication

Freeze, **suspend**, running VM

- Then can move or copy somewhere else and **resume**
- Snapshot of a given state, able to restore back to that state
  - Some VMMs allow multiple snapshots per VM
- **Clone** by creating copy and running both original and copy

Great for OS research, better system development efficiency

Run multiple, different OSes on a single machine

- **Consolidation**, app dev, …

# Benefits and Features (cont.)

**Templating** – create an OS + application VM, provide it to customers, use it to create multiple instances of that combination

**Live migration** – move a running VM from one host to another!

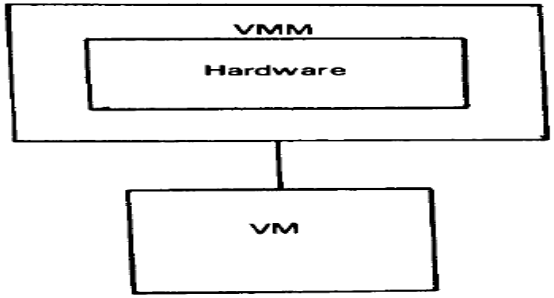
- No interruption of user access

All those features taken together -> **cloud computing**

- Using APIs, programs tell cloud infrastructure (servers, networking, storage) to create new guests, VMs, virtual desktops

# Origins - Principles

Fig. 1. The virtual machine monitor.




"an *efficient, isolated duplicate* of the real machine"

**Efficiency**

Innocuous instructions should execute directly on the hardware

**Resource control**

Executed programs may not affect the system resources

**Equivalence**

The behavior of a program executing under the VMM should be the same as if the program were executed directly on the hardware (except possibly for timing and resource availability)

## Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

**Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.**

Communications of the ACM, vol 17, no 7, 1974, pp.412-421

# History

First appeared in IBM mainframes in 1972

Allowed multiple users to share a batch-oriented system

Formal definition of virtualization helped move it beyond IBM

1. A VMM provides an environment for programs that is essentially identical to the original machine
2. Programs running within that environment show only minor performance decreases
3. The VMM is in complete control of system resources

In late 1990s Intel CPUs fast enough for researchers to try virtualizing on general purpose PCs

- **Xen** and **VMware** created technologies, still used today
- Virtualization has expanded to many OSes, CPUs, VMMs

# Origins - Principles

Instruction types (security some time ago :-D) )

Privileged

an instruction traps in unprivileged (user) mode but not in privileged (supervisor) mode.

Sensitive

- Control sensitive –
  attempts to change the memory allocation or privilege mode
- Behavior sensitive
  - Location sensitive – execution behavior depends on memory location
  - Mode sensitive – execution behavior depends on privilege mode

Innocuous – an instruction that is not sensitive

Theorem

For any conventional computer, a virtual machine monitor may be built if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

Significance = The IA-32/x86 architecture is not virtualizable.

# Origins - Technology

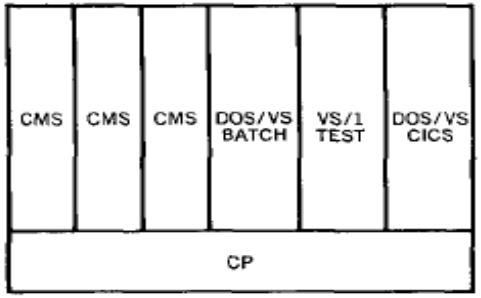

**VM/370—a study of multiplicity and usefulness**

**by L. H. Seawright and R. A. MacKinnon**

The productivity of data processing professionals and other professionals can be enhanced through the use of interactive and time-sharing systems. Similarly, system programmers can benefit from the use of system testing tools. A systems solution to both areas can be the virtual machine concept, which provides multiple software replicas of real computing systems on one real processor. Each virtual machine has a full complement of input/output devices and provides functions similar to those of a real machine. One system that implements virtual machines is IBM's Virtual Machine Facility/370 (VM/370).[1]

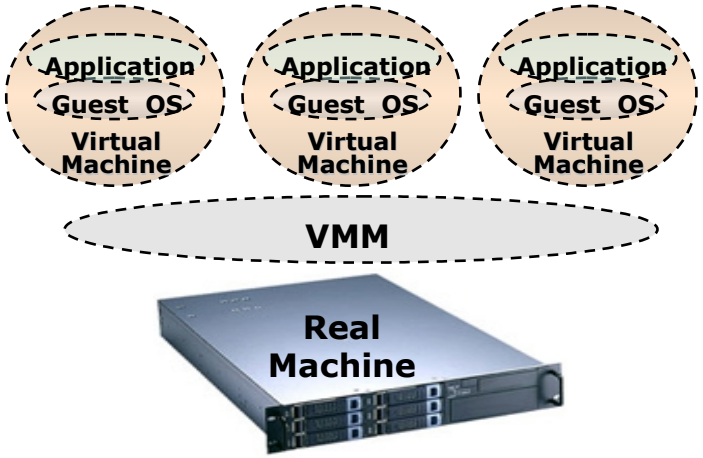
IBM Systems Journal, vol. 18, no. 1, 1979, pp. 4-17.

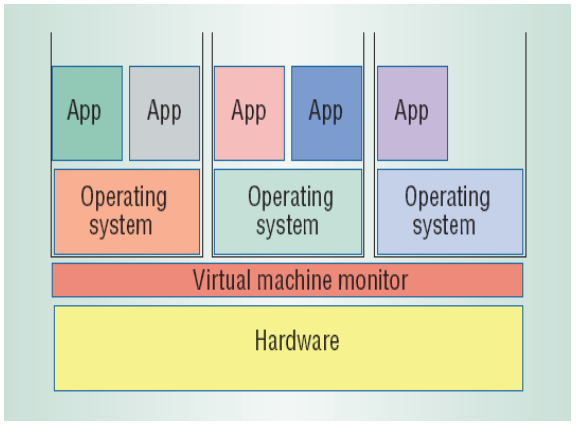
Figure 1 A VM/370 environment




- Concurrent execution of multiple OSes
- Testing and development of experimental systems
- Adoption of new systems with continued use of legacy systems
- Ability to accommodate applications requiring special-purpose OS
- Introduced notions of "handshake" = transparency and "virtual-equals-real mode" to allow sharing of resource control information with CP
- Leveraged ability to co-design hardware, VMM, and guestOS

# VMMs Rediscovered

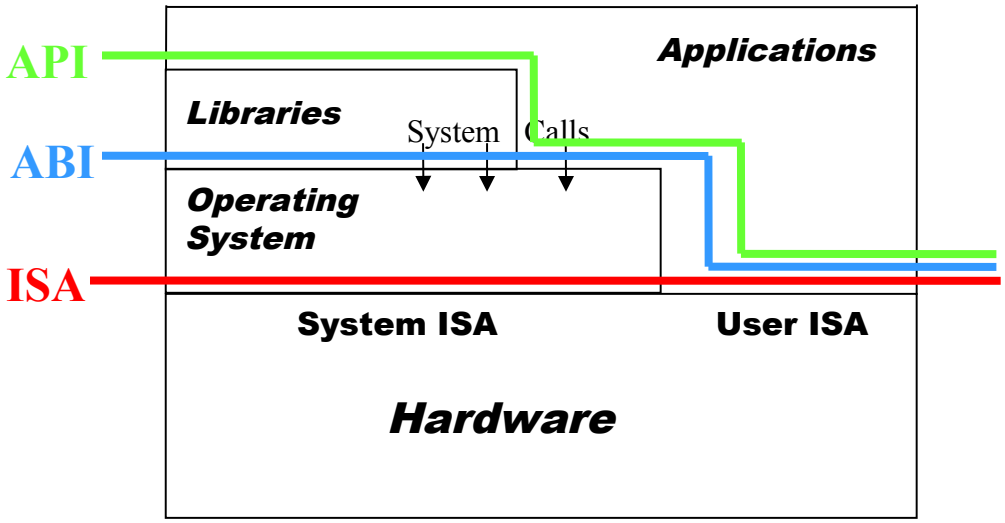



Server/workload consolidation (reduces “server sprawl”)
Compatible with evolving multi-core architectures
Simplifies software distributions for complex environments
“Whole system” (workload) migration
Improved data-center management and efficiency
Additional services (workload isolation) added “underneath” the OS
  security (intrusion detection, sandboxing,…)
  fault-tolerance (checkpointing, roll-back/recovery)

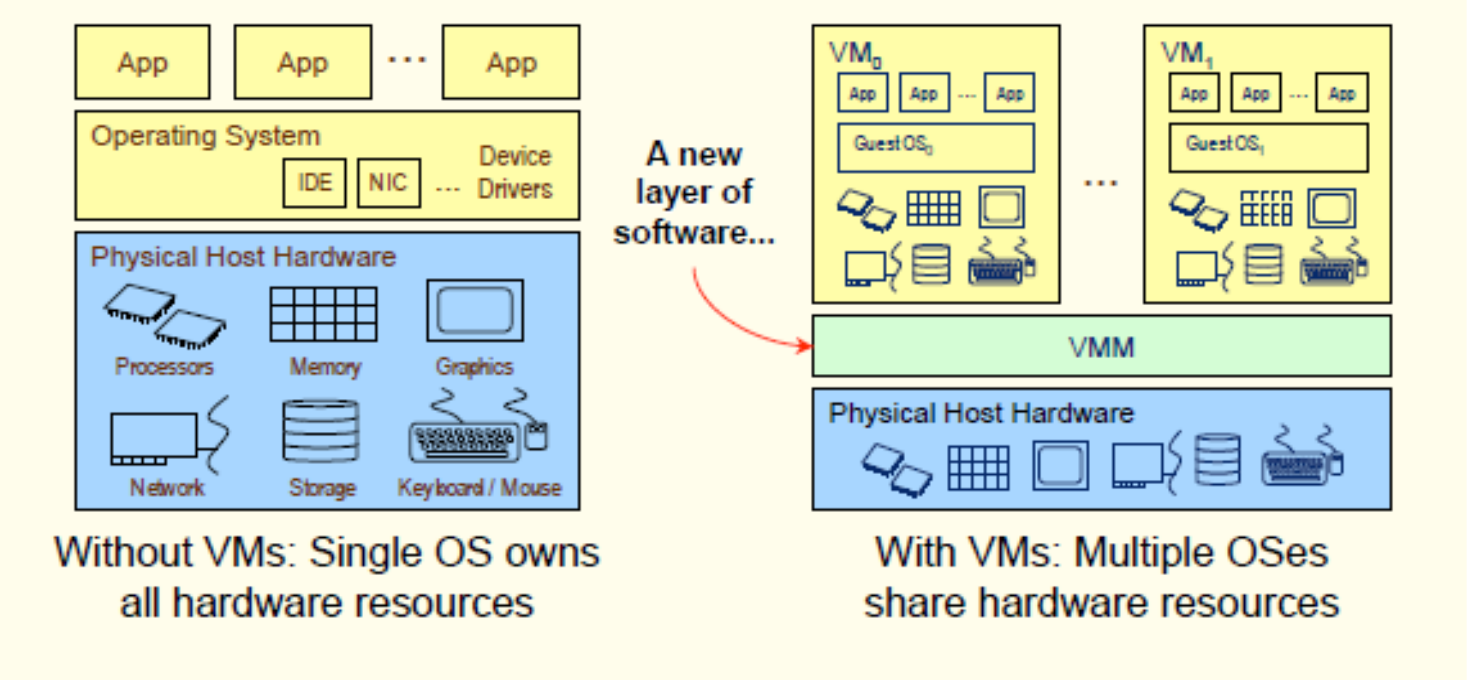
# Architecture & Interfaces

Architecture: formal specification of a system's interface and the logical behavior of its visible resources.



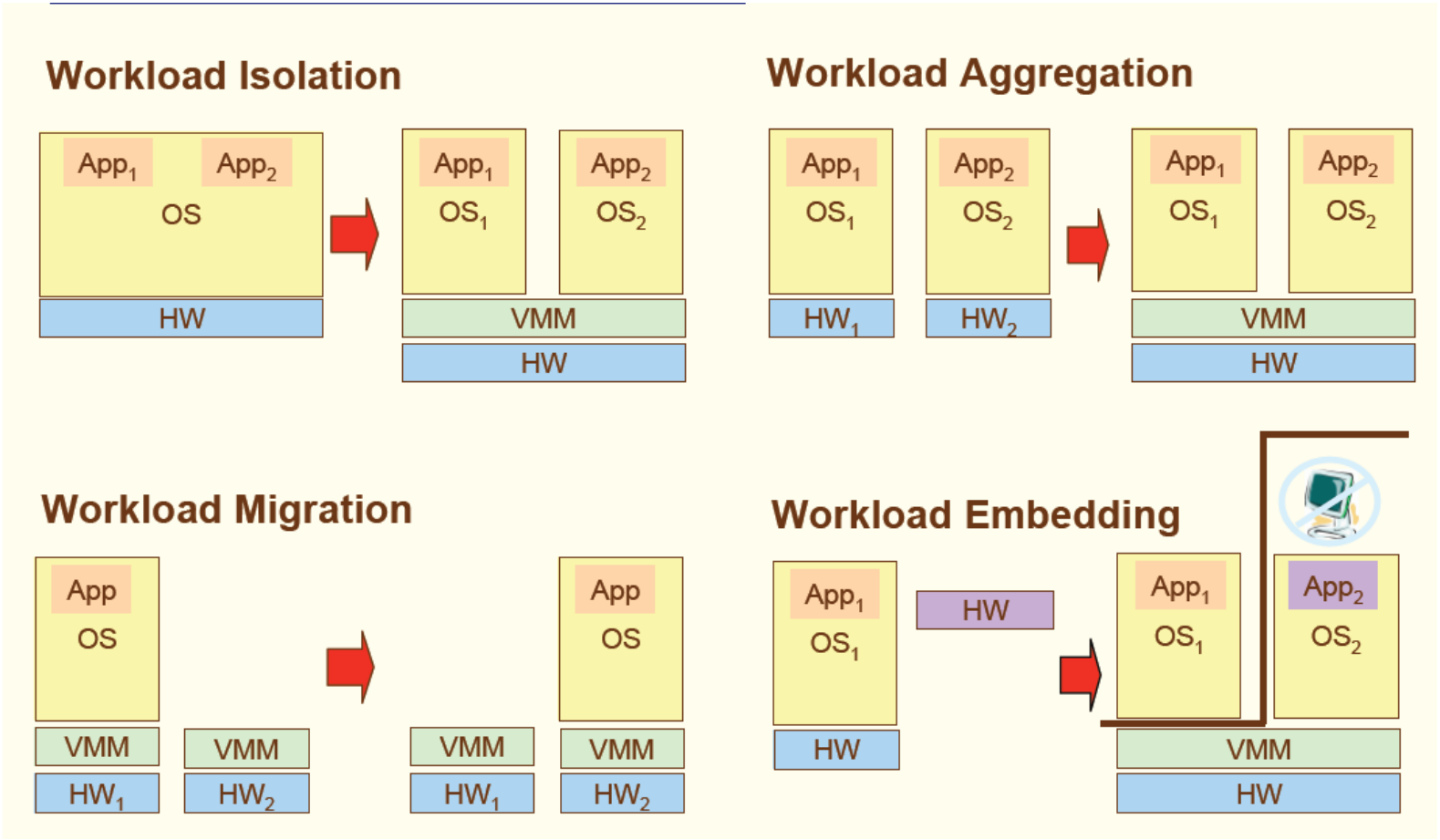

- API – application programming interface
- ABI – application binary interface
- ISA – instruction set architecture

# Our focus on system VMs




A new software layer is introduced that honors the existing ISA to create distinct physical machines

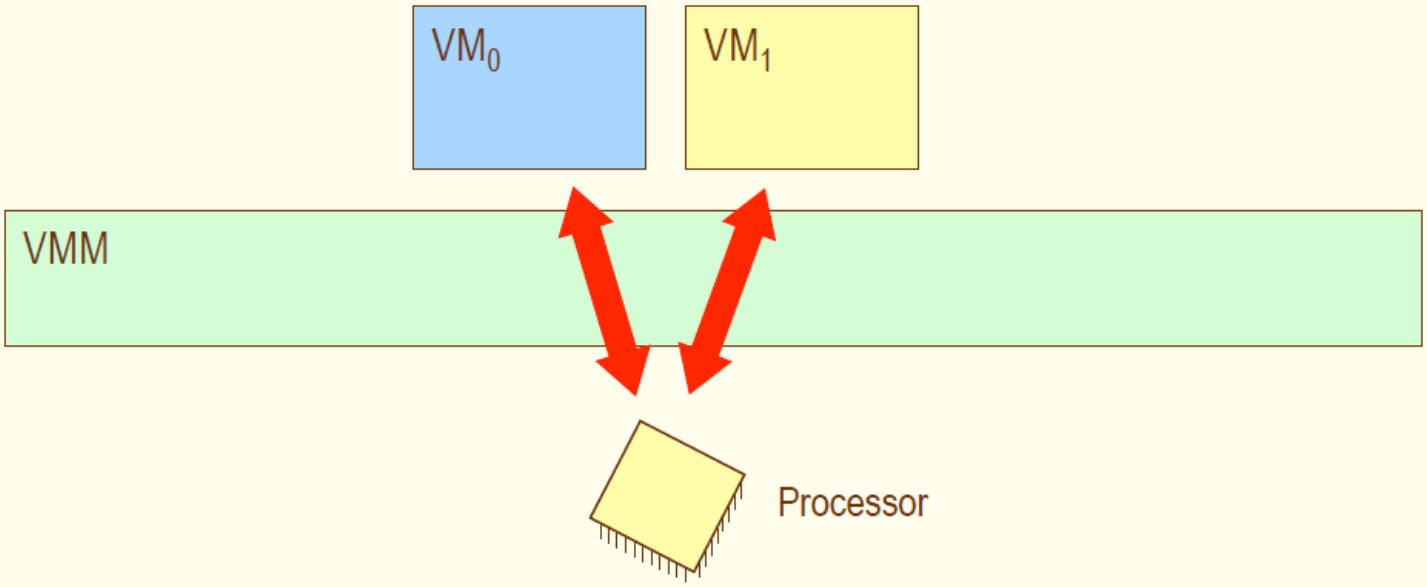
# Basic Capabilities



# Basic VMM requirements

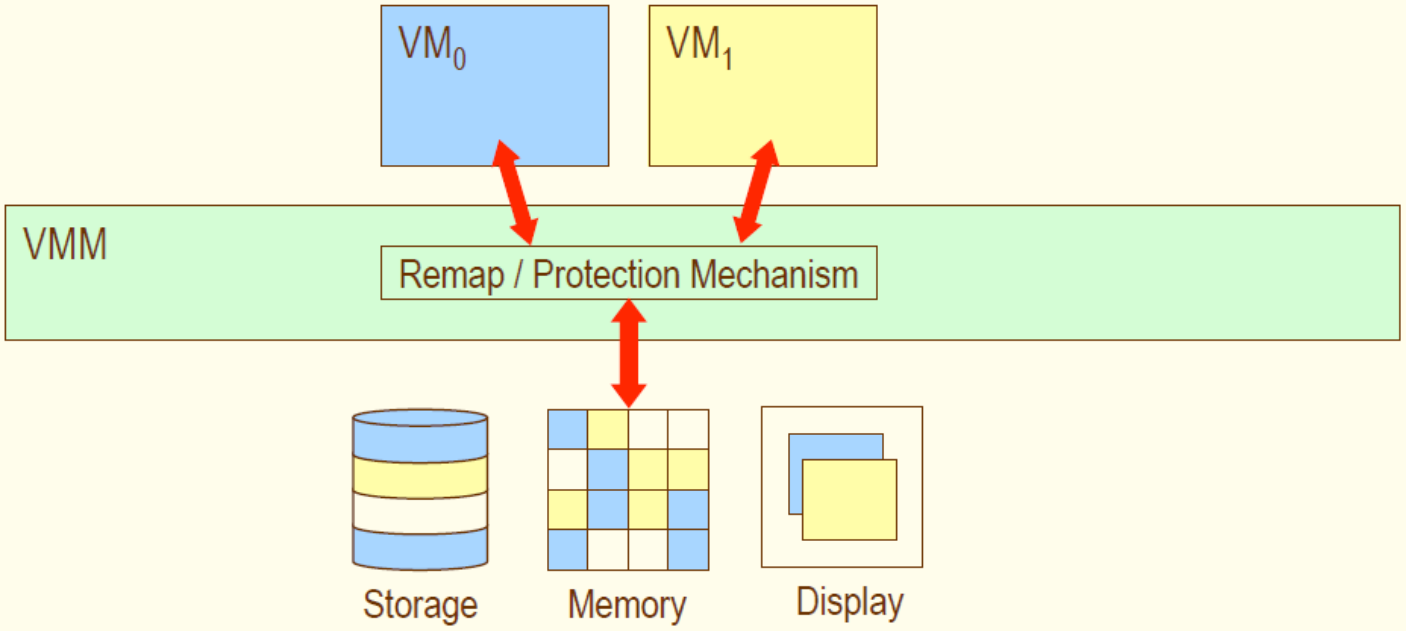
- **A VMM must be able to:**
  - Protect itself from guest software
  - Isolate guest software stacks (OS + Apps) from one another
  - Present a (virtual) platform interface to guest software
- **To achieve this, VMM must control access to:**
  - CPUs, Memory and I/O Devices
- **Ways that a VMM can share resources between VMs**
  - Time multiplexing
  - Resource partitioning
  - Mediating hardware interfaces

# Time Multiplexing



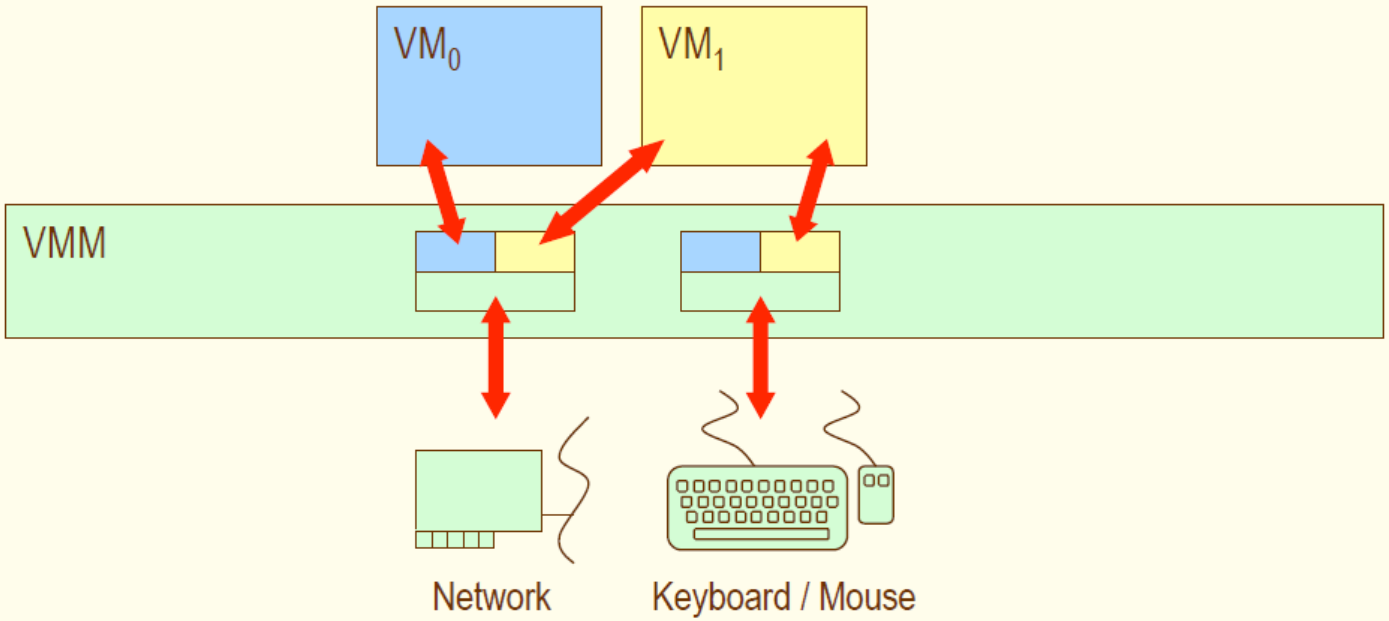

- **VM is allowed direct access to resource for a period of time before being context switched to another VM (e.g., CPU resource)**

# Resource Partitioning



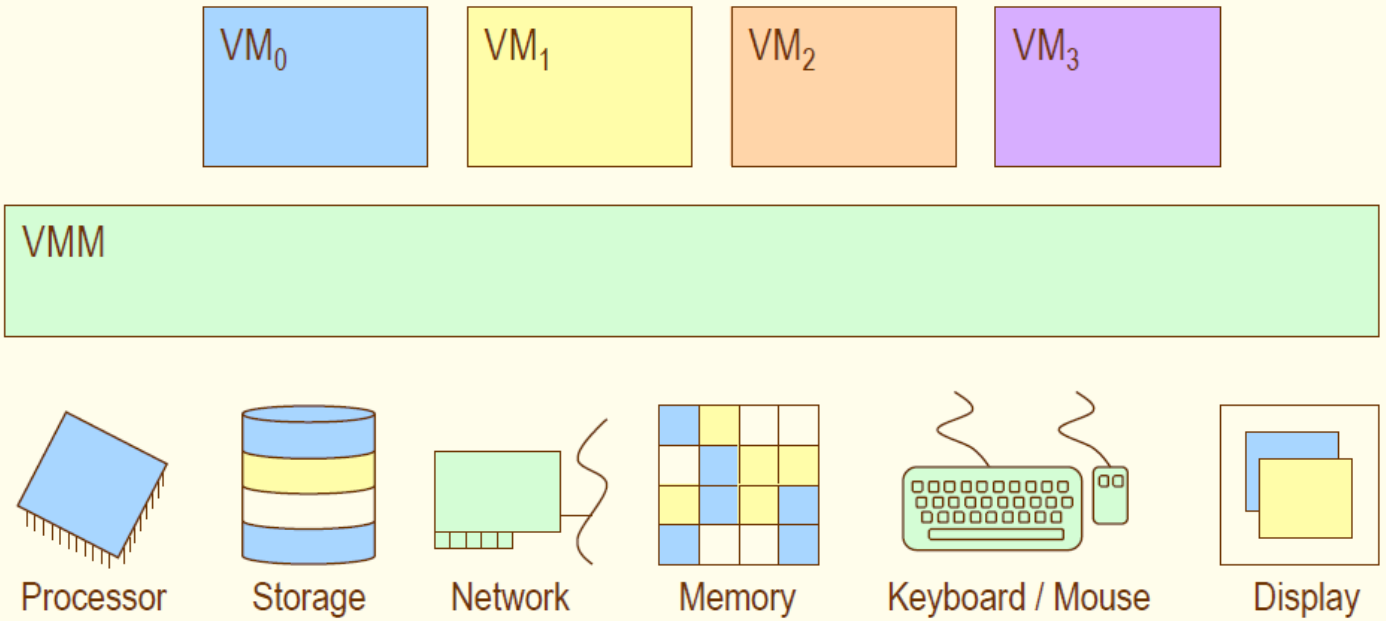

- **VMM allocates "ownership" of phys resources to VMs**
  - Typically involves some remapping and protection mechanism
  - Examples: physical memory, disk partitions, graphical display

# Mediating Access to Physical Resources



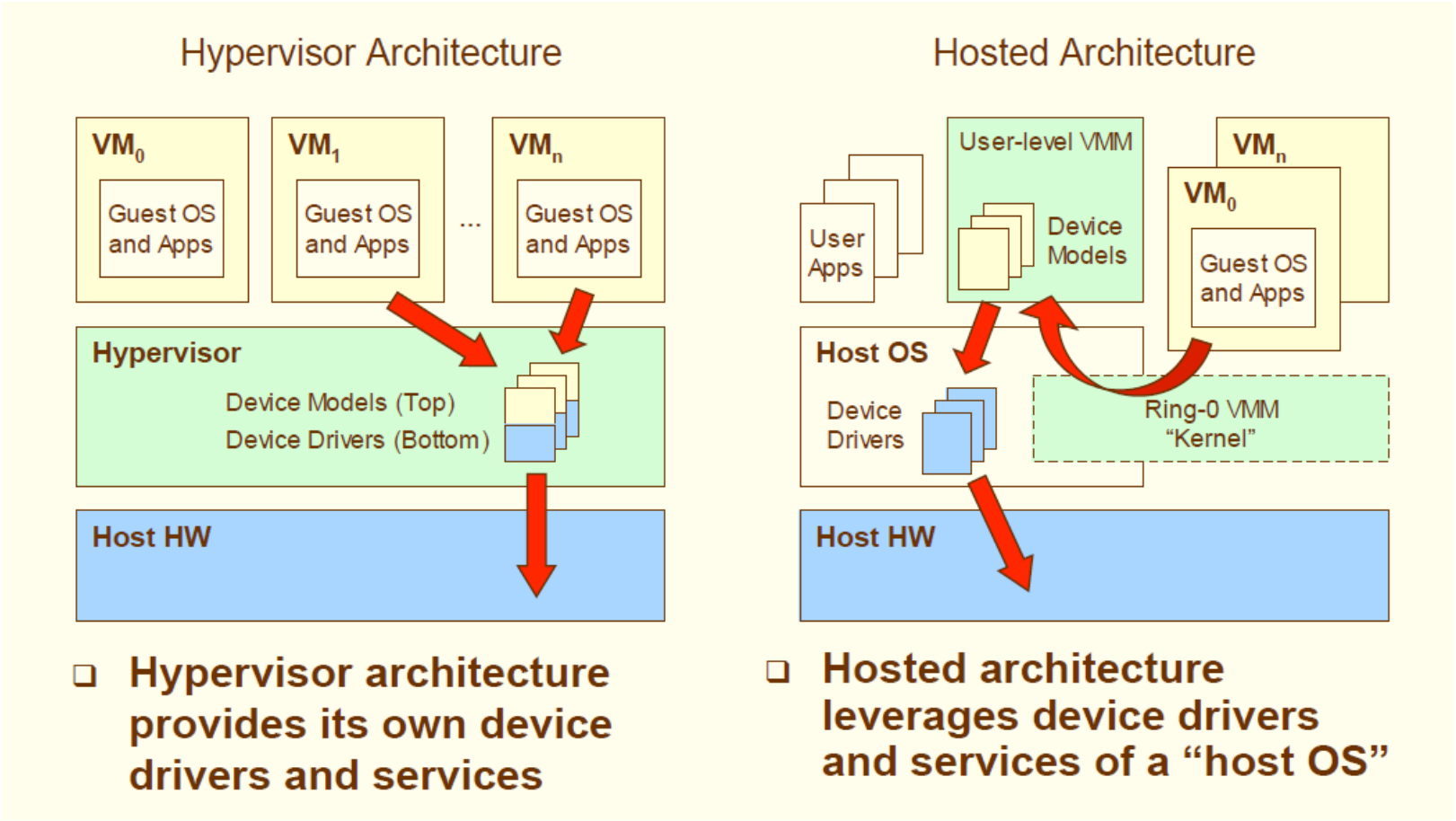

- **VMM retains direct ownership of physical resource**
  - VMM hosts device driver as well as a virtualized device interface
  - Virtual interface can be same as or different than physical device

# All together …



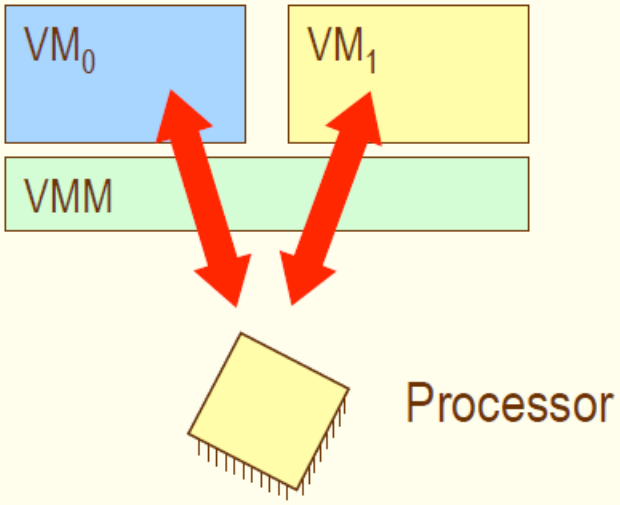

- **VMM applies all 3 sharing methods, as needed, to create illusion of platform ownership to each guest OS**

# Alternative Options




- **Hypervisor architecture provides its own device drivers and services**
- **Hosted architecture leverages device drivers and services of a "host OS"**

# CPU Virtualization



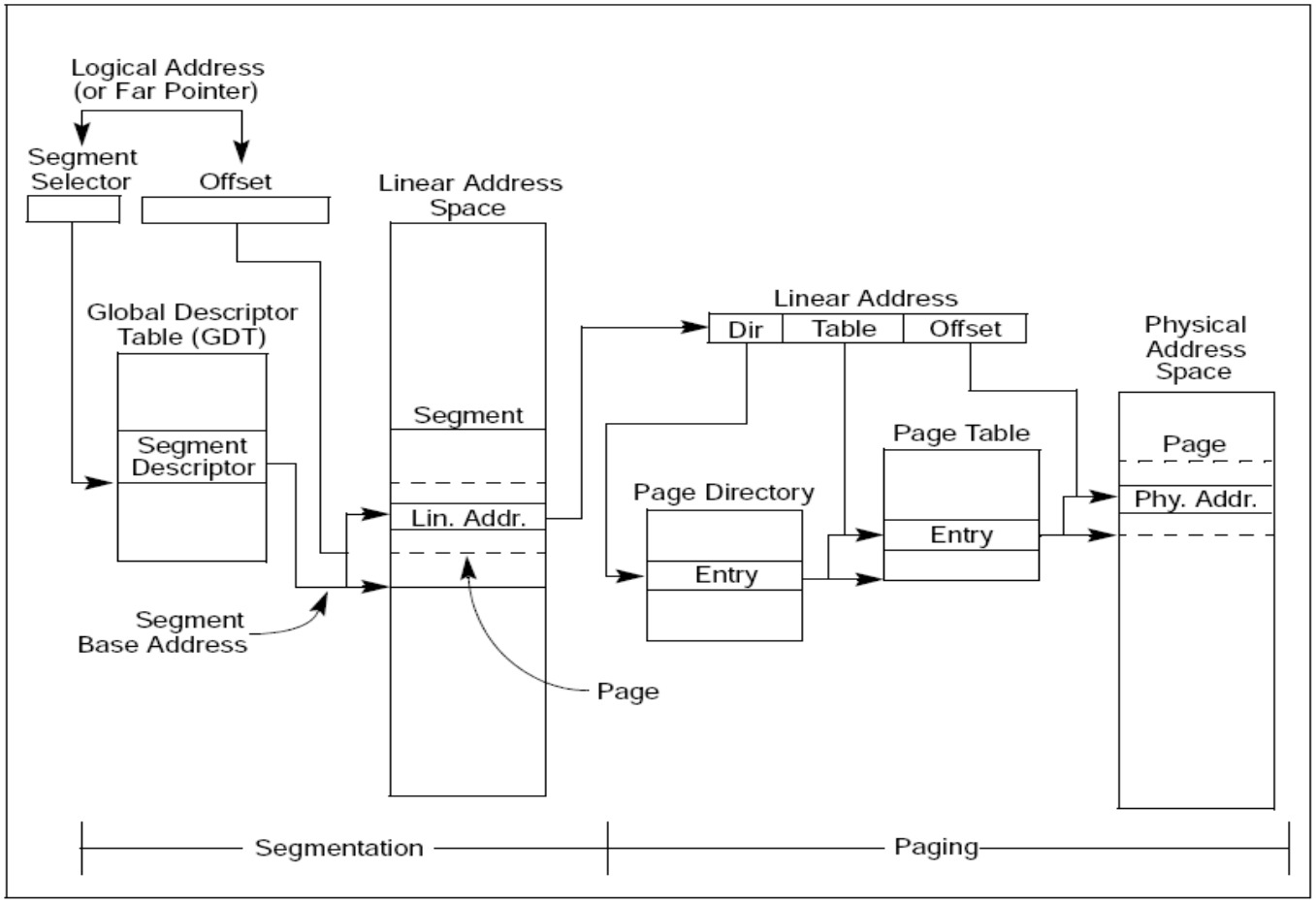

- **To virtualize a CPU, a VMM must retain control over:**
  - Accesses to privileged state (control regs, debug regs, etc.)
  - Exceptions (page faults, machine-check exceptions, etc.)
  - Interrupts and interrupt masking
  - Address translation (via page tables)
  - CPU access to I/O (via I/O ports or MMIO)

# Virtualization: Implementation Strategies

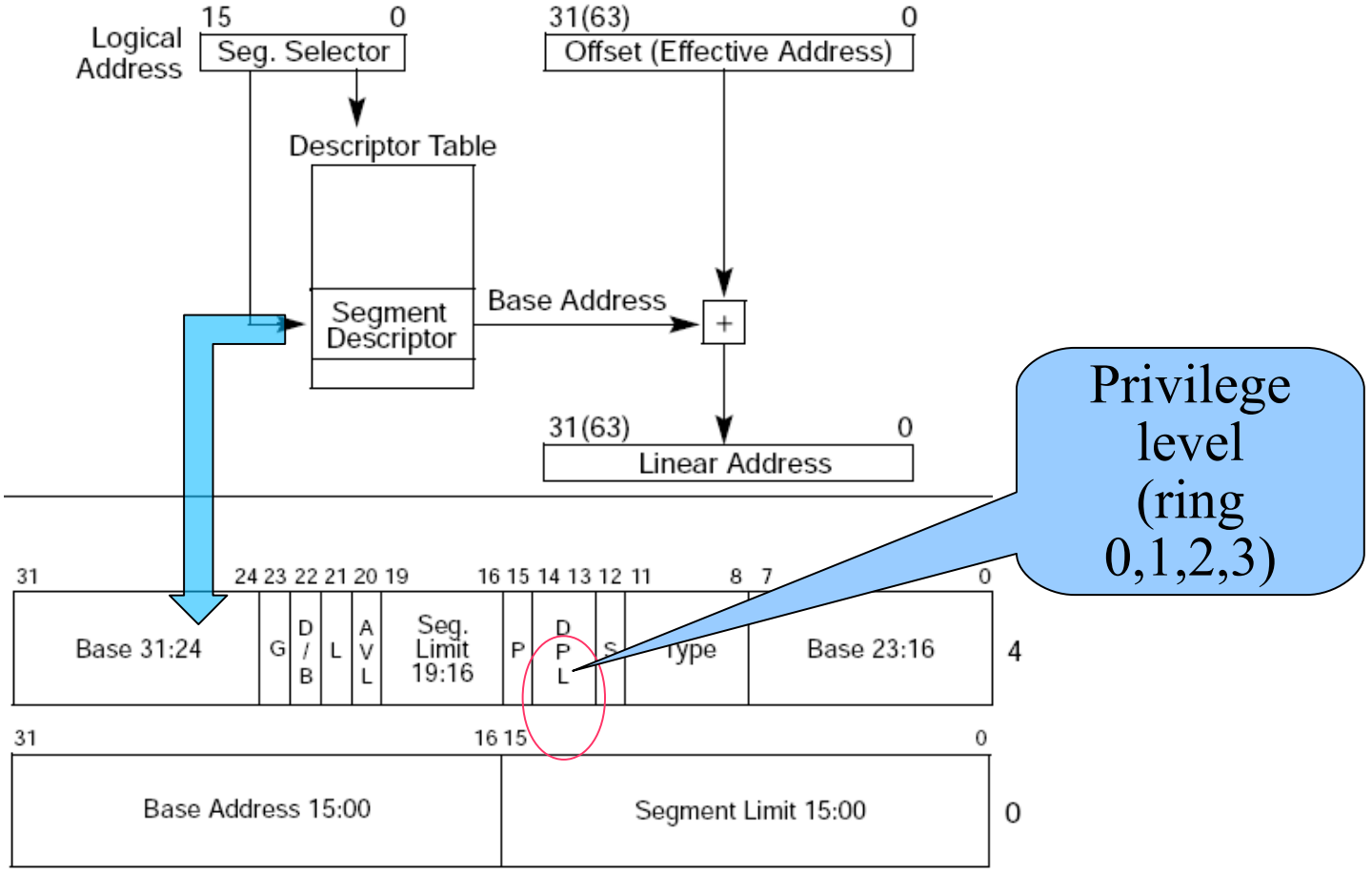
# IA-32

- **IA-32 Provides 4 Privilege Levels (Rings)**
- **Segment-based Protections**
  - Distinguish between all 4 rings
- **Page-based Protections**
  - Separate only User and Supervisor modes
  - User mode: Code running in ring 3
  - Supervisor mode: Code running in rings 0, 1, or 2

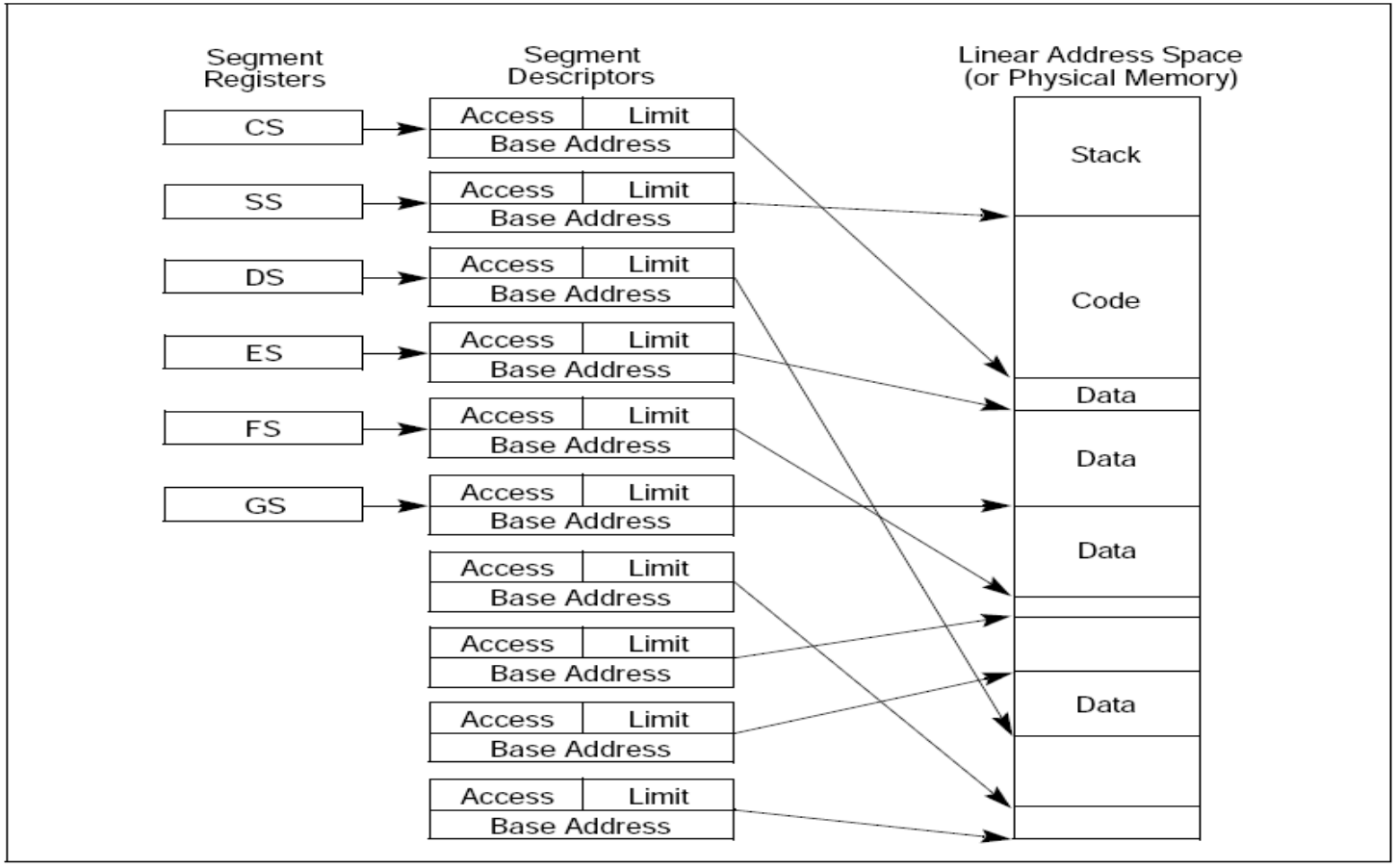
# Memory Management on IA32



# Segment descriptor



# Multi segments



# Challenges of running a VMM

1) OS and Apps in a VM don't know that the VMM exists or that they share CPU resources with other VMs.

2) VMM should isolate Guest SW stacks from one another.

3) VMM should run protected from all Guest software

4) VMM should present a virtual platform interface to Guest SW.

# Classical solution

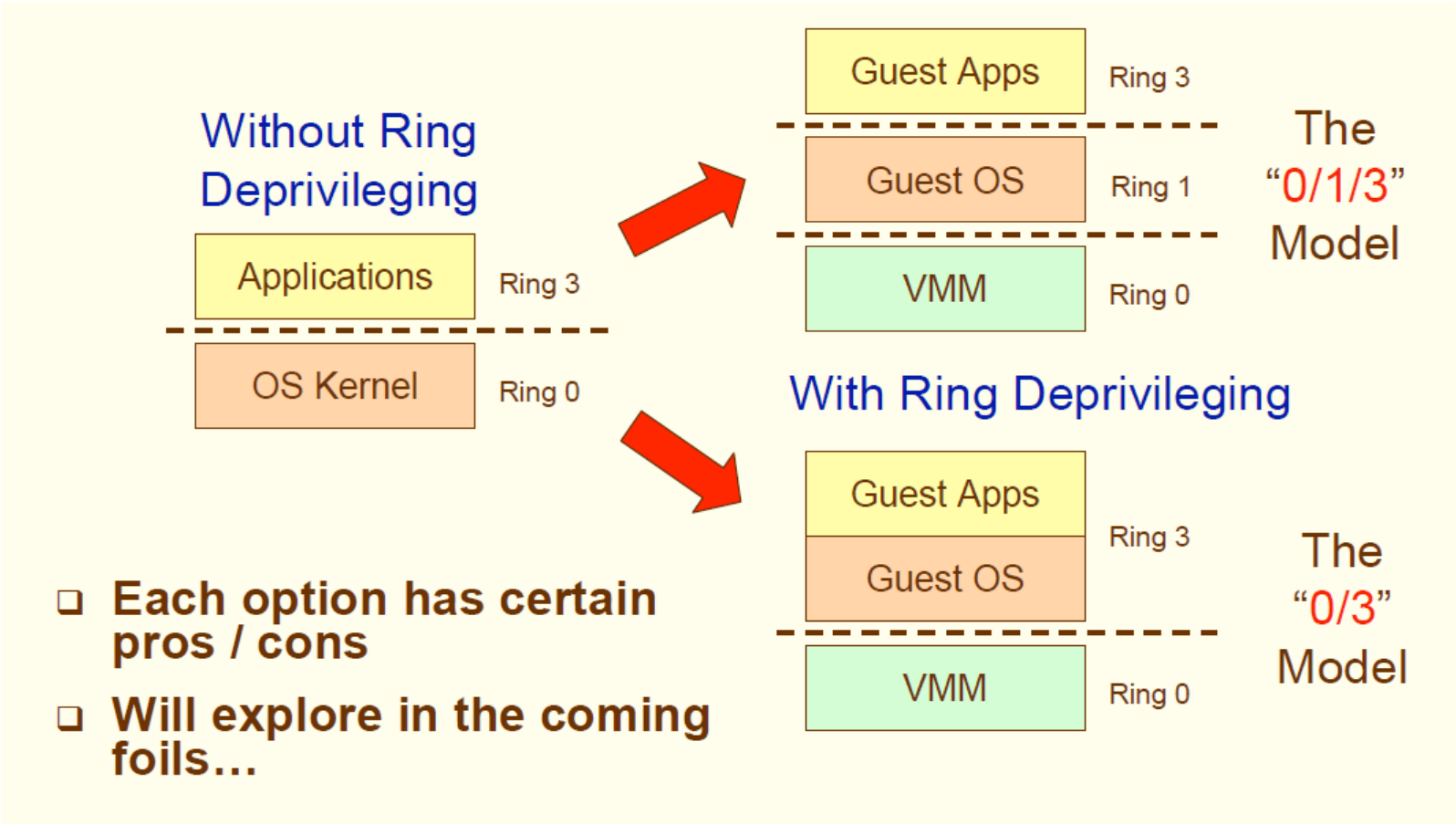
## Trap and Emulate

- Run guest operating system *deprivileged*
- All privileged instructions *trap into VMM*
- VMM emulates instructions against virtual state
  *e.g.* disable virtual interrupts, not physical interrupts
- Resume direct execution from next guest instruction

## Implementation Technique

- This is just one technique
- Popek and Goldberg criteria permit others

# Some Options

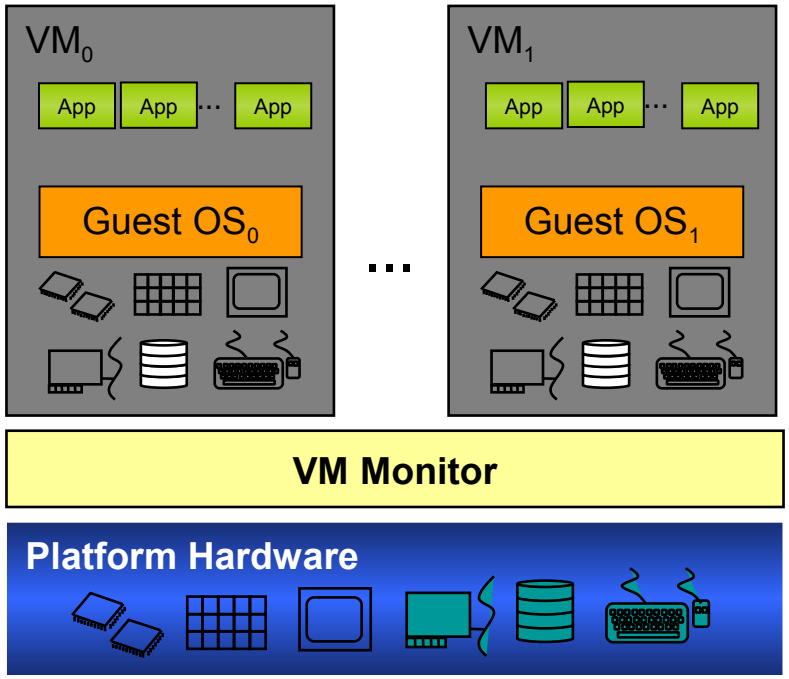



- **Each option has certain pros / cons**
- **Will explore in the coming foils…**

# Solution for IA-32 arch

## Ring Deprivileging =

- all guest software should be run at a privilege level greater than 0.
- privileged instructions generate faults = VMM runs in Ring-0 as a collection of fault handlers.
- the guest OS should not be able to update the VMM




- The VMM interprets in software privileged instructions that would be executed by an OS.
- Any non privileged instruction issued by an OS or Application Environment is executed directly by the machine.
- A guest OS could be deprivileged in two distinct ways:
  - it could run either at privilege level 1 (the 0/1/3 model) or ,
  - It could run at privilege level 3 (the 0/3/3 model).

# Virtualization challenges

## Ring Aliasing

Problems if software is run at a privilege level other than the privilege level for which it was written.

- The CS register points to the code segment.
- If the *PUSH* is executed with the CS register, the register content (with the current privilege level) is pushed on the stack

→ A guest OS could easily determine that it is not running at privilege level 0.

## Address-Space Compression

OSs expect to have access to the processor's full virtual address space (in IA-32. linear address space)

- The VMM could run entirely within the guest's virtual-address space but it would use a substantial amount of the guest's virtual address space.
- The VMM could run in a separate address space, but it must use a minimal amount of the guest's virtual address space to manage transitions between guest software and the VMM (IDT and GDT for IA-32)

To preserve its integrity, the VMM must prevent guest access to those portions of the guest's virtual address space that it is using.

# Virtualization challenges

## Excessive Faulting

Ring deprivileging interferes with the effectiveness of facilities in the IA-32 architecture that accelerate the delivery and handling of transitions to OS software.

- The IA-32 *SYSENTER* and *SYSEXIT* instructions support low-latency system calls.
- *SYSENTER* always effects a transition to privilege level 0, and *SYSEXIT* faults if executed outside that ring

⇒ With VMM it does traps to the OS but to the VMM that emulates every execution of SYSENTER and SYSEXIT to implement interactions with the OS causing serious performance problems.

## Non-Trapping Instructions

Some instructions access privileged state and do not fault when executed with insufficient privilege.

- the IA-32 registers GDTR, IDTR, LDTR, and TR contain pointers to data structures that control CPU operation. Software can execute the instructions that read, or store, from these registers at any privilege level.

# Virtualization challenges

## Interrupt Virtualization

- The mechanisms of masking external interrupts for preventing their delivery when the OS is not ready for them is a big challenge for the VMM design.
- The VMM must manage the interrupt masking in order to prevent an OS from masking the external interrupts because this prevents any guest OS to receive interrupts.
    - IA-32 uses the interrupt flag (IF) in EFLAGS register to control interrupt masking. IF= 0 interrupts are masked.

## Access to Hidden State

- Some components of the processor state are not represented in any software-accessible register.
    - the IA-32 has the hidden descriptor caches for segment registers. A segment-register load copies of the GDT and LDT into this cache, which is not modified if software later writes to the descriptor tables.

# Virtualization challenges

## Ring Compression

Ring deprivileging uses privilege-based mechanisms to protect the VMM from guest software. IA-32 includes two mechanisms: segment limits and paging:

- Segment limits do not apply in 64-bit mode.
- Paging must be used.
  - Problem: IA-32 paging does not distinguish privilege levels 0-2.
    - » The guest OS must run at privilege level 3 (the 0/3/3 model).
    - » The guest OS is not protected from the guest applications.

## Frequent Access to Privileged Resources

The performance is compromised if the privileged resources are accessed too many times generating too many faults that must be intercepted by the VMM.

- For example: the task-priority register (TPR), in IA-32 located in the advanced programmable interrupt controller (APIC), is accessed with very high frequency by some OSs.

# Alternative solutions

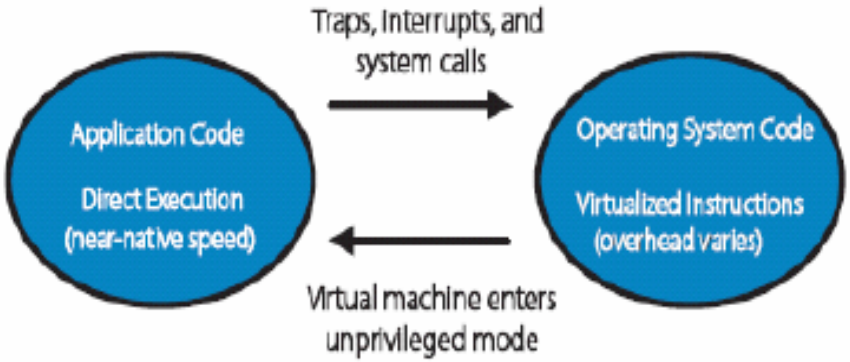
Interpretation
- Problem – too inefficient
- x86 decoding slow

Code Patching
- Problem – not transparent
- Guest can inspect its own code

Binary Translation (BT)
- Approach pioneered by VMware
- Run any unmodified x86 OS in VM

Extend x86 Architecture

# Software VMM




Direct execute unprivileged guest application code

- Will run at full speed until it traps, we get an interrupt, etc.

"Binary translate" all guest kernel code, run it unprivileged

- Since x86 has non-virtualizable instructions, *proactively* transfer control to the VMM (no need for traps)
- Safe instructions are emitted without change
- For "unsafe" instructions, emit a controlled emulation sequence
- VMM translation cache for good performance

# Solution adopted by VMware

**Binary** – input is x86 “hex”, not source

**Dynamic** – interleave translation and execution

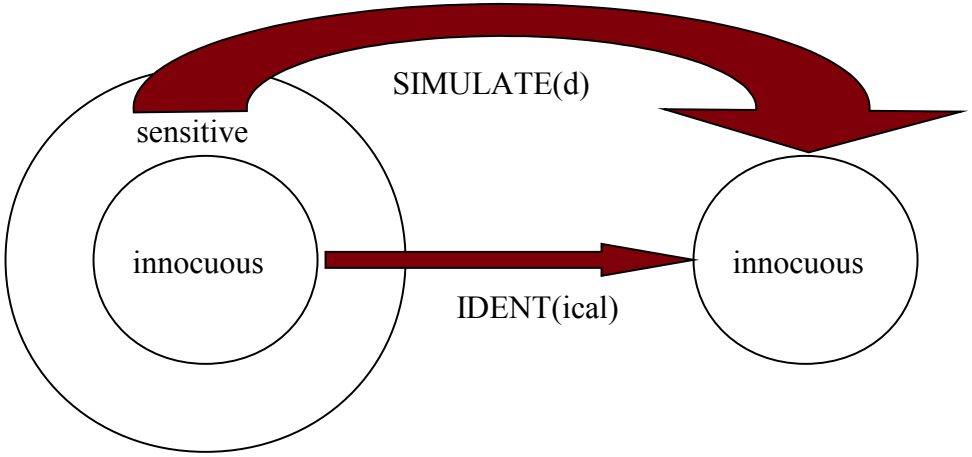
**On Demand** – translate only what about to execute (lazy)

**System Level** – makes no assumptions about guest code

**Subsetting** – full x86 to safe subset

**Adaptive** – adjust translations based on guest behavior

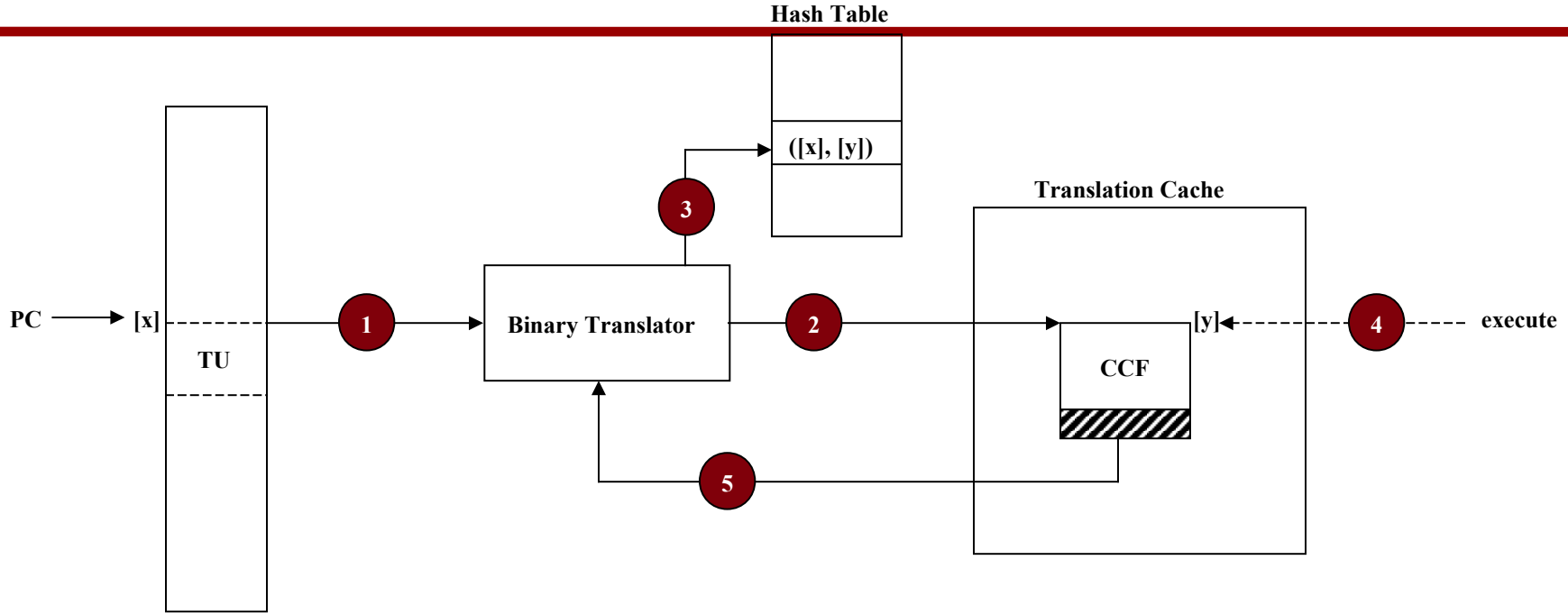
# Binary Translation



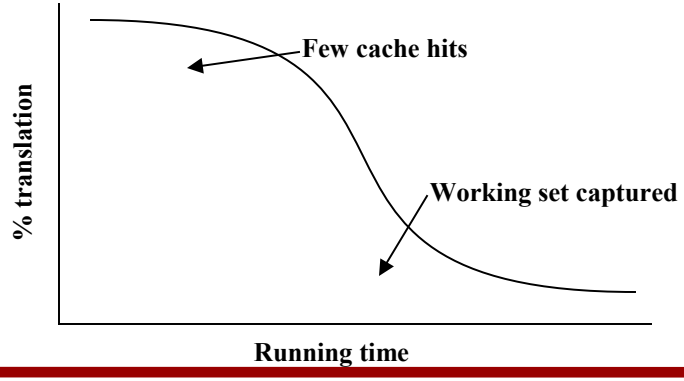

## Characteristics

Binary – input is machine-level code
Dynamic – occurs at runtime
On demand – code translated when needed for execution
System level – makes no assumption about guest code
Subsetting – translates from full instruction set to safe subset
Adaptive – adjust code based on guest behavior to achieve efficiency

# Binary Translation




TU: translation unit (usually a basic block)
CCF: compiled code fragment
: continuation



# Eliminating faults/traps

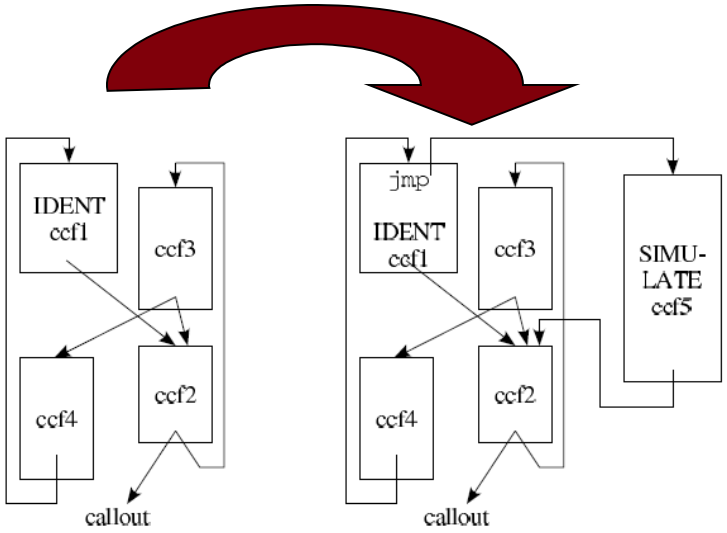
Expensive traps/faults can be avoided

Example: Pentium privileged instruction (rdtsc)

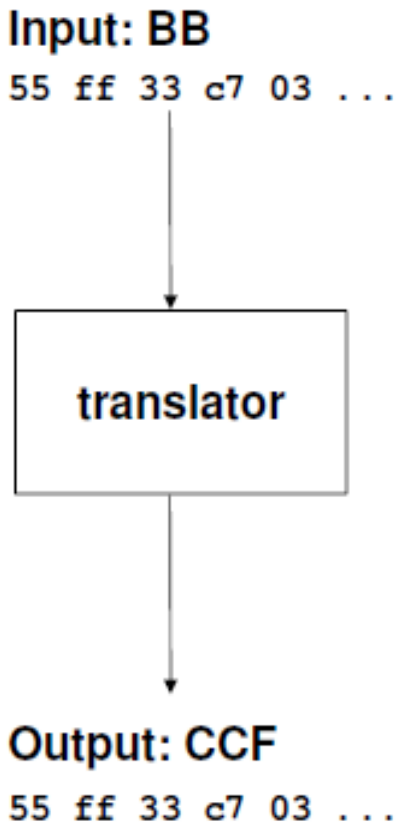
- Trap-and-emulate: 2030 cycles
- Callout-and-emulate: 1254 cycles
- In-TC emulation: 216 cycles

Process

- Privileged instructions – eliminated by simple binary translation (BT)
- Non-privileged instructions – eliminated by adaptive BT
  - (a) detect a CCF containing an instruction that trap frequently
  - (b) generate a new translation of the CCF to avoid the trap (perhaps inserting a call-out to an interpreter), and patch the original translation to execute the new translation



# Binary Translation Process




Each Translator Invocation

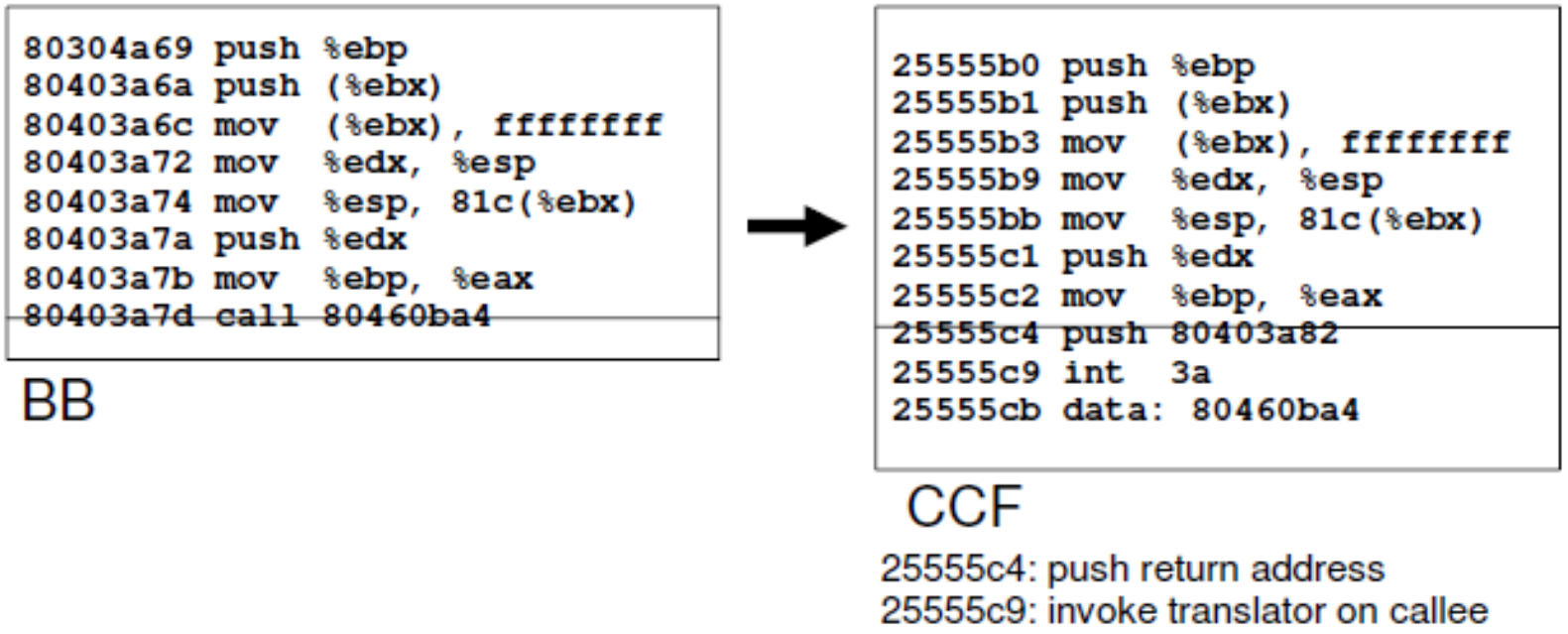
- Consume a basic block (BB)
- Produce a compiled code fragment (CCF)

Store CCF in Translation Cache

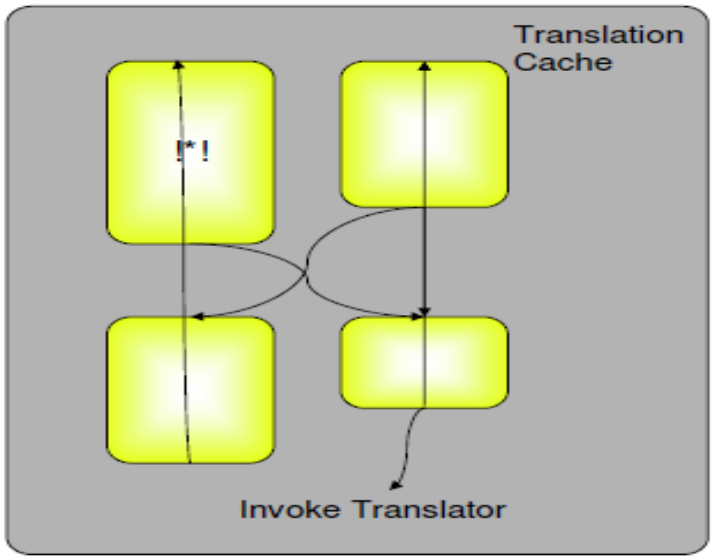
- Future reuse
- Capture working set of guest kernel
- Amortize translation costs
- Not "patching in place"

At most 12 instructions

# Binary Translation Process



# Adaptive Binary Translation Process




Translated Code Is Fast
- Mostly IDENT translations
- Runs "at speed"

Except Writes to Traced Memory
- Page fault (shown as !*!)
- Decode and interpret instruction
- Fire trace callbacks
- Resume execution
- Can take 1000's of cycles

Cache tables are protected
= trace memory

Detect instructions that trap frequently
Adapt the translation of these instructions =
- Re-translate to avoid trapping.
- Jump directly to translation.

Adaptive Binary Translation tries to eliminate more and more traps over time.

# Trace Memory

## Shadow Page Table

- Derived from primary page table in guest
- VMM must keep primary and shadow coherent

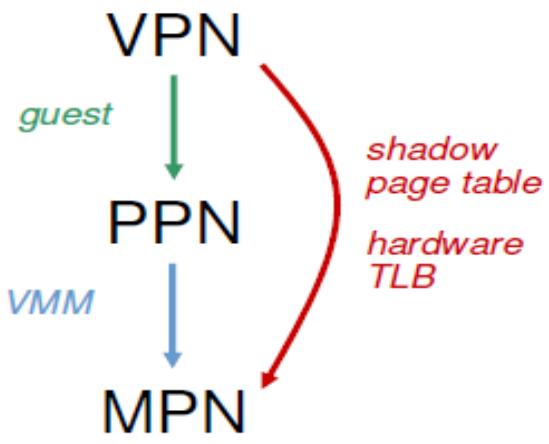
## Trace = Coherency Mechanism

- Write-protect primary page table
- Trap guest writes to primary
- Update or invalidate corresponding shadow
- Transparent to guest

# Shadow Page Table

- Shadow page tables are used by the hypervisor to keep track of the state in which the guest "thinks" its page tables should be.
- The guest can't be allowed access to the hardware page tables because then it would essentially have control of the machine
- The hypervisor keeps
  - the "real" mappings guest virtual → host physical in the hardware when the relevant guest is executing
  - a representation of the page tables that the guest thinks it's using "in the shadows," they are not used by the hardware

# Shadow structures




Traditional VMM Approach

Extra Level of Indirection

- Virtual → "Physical"
  Guest maps VPN to PPN
  using primary page tables
- "Physical" → Machine
  VMM maps PPN to MPN

Shadow Page Table

- Composite of two mappings
- For ordinary memory references
  Hardware maps VPN to MPN
- Cached by physical TLB

Built incrementally at hidden faults

A shadow structure records the state of the emulated machine

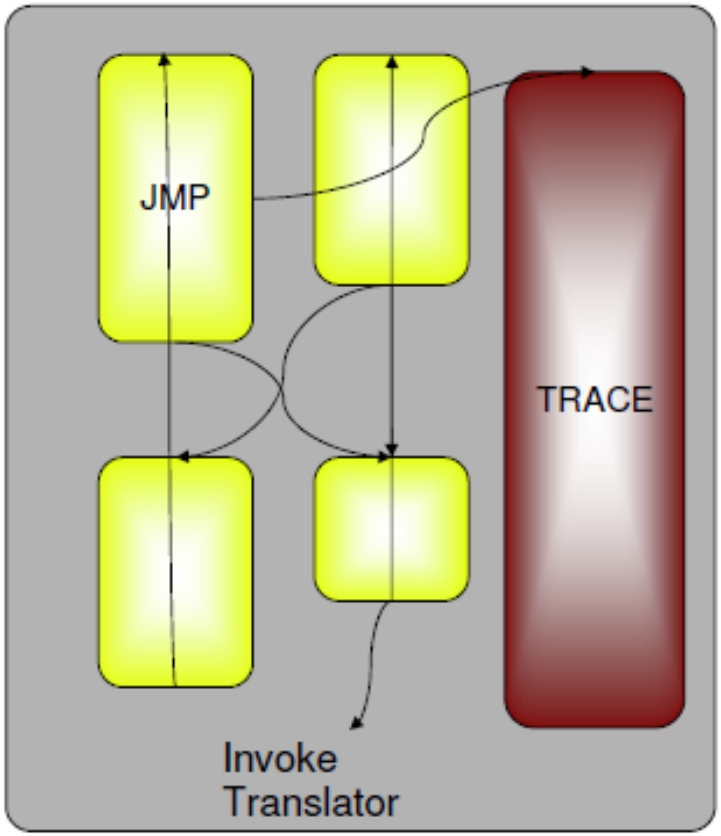
VPN= virtual page number,
PPN=physical page number
MPN= machine page number
True fault = faults in the emulated machine
Hidden fault = due to the shadow page table

# VMM memory management

- VMMs tend to have simple hardware memory allocation policies
  - Static: VM gets 512 MB of hardware memory for life
  - No dynamic adjustment based on load because OSes not designed to handle changes in physical memory…
  - No swapping to disk
- Balloon driver runs inside OS to consume hardware page:

  "ESX Server controls a balloon module running within the guest directing it to allocate guest pages and pin them in ``physical'' memory.The machine pages backing this memory can then be reclaimed by ESX Server. Inflating the balloon increases memory pressure, forcing the guest OS to invoke its own memory management algorithms. Deflating the balloon decreases pressure, freeing guest memory."
- Identify identical physical pages (e.g., all zeroes) and map those pages copy-on-write across VMs

# Adaptive Binary Translation Process




Detect and Track Trace Faults

Splice in TRACE Translation

- Execute memory access in software
- Avoid page fault
- No re-decoding
- Faster resumption

Faster Traces

- 10x performance improvement
- Adapts to runtime behavior

# Expected Performance for Hardware Solutions

## Native Speed Except for Traps

- No overhead in direct execution
- Overhead = trap frequency × average trap cost

## Trap Sources

- *Most frequent:* Guest page table traces
- Privileged instructions
- Memory-mapped device traces

# Hardware Solution = Intel® Virtualization Technology

- VT-x: Support for IA-32 processor virtualization
- VT-i: Support for Itanium processor virtualization

# Hardware VMM

## Recent x86 Extension

- 1998 – 2005: Software-only VMMs using binary translation
- 2005: Intel and AMD start extending x86 to support virtualization

## First-Generation Hardware

- Enables classical trap-and-emulate VMMs
- Intel VT, aka "Vanderpool Technology"
- AMD SVM, aka "Pacifica"

## Performance

- VT/SVM help avoid BT, but not MMU ops (actually slower!)
- Main problem is efficient virtualization of MMU and I/O, *Not* executing the virtual instruction stream

BT=binary translation

# VT-x Modes

**VMX root** operation:

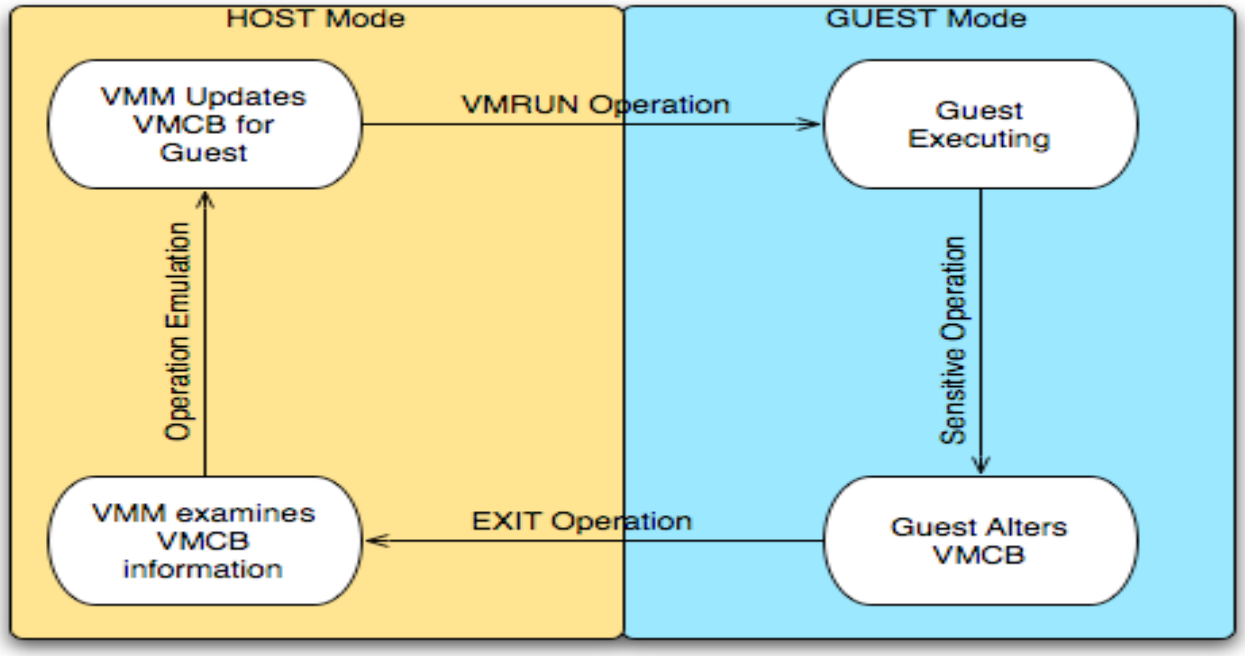
Full privileged, intended for Virtual Machine Monitor

**VMX non-root** operation:

Not fully privileged, intended for guest software

- Both forms of operation support all four privilege levels from 0 to 3

# x86 Architecture Extensions

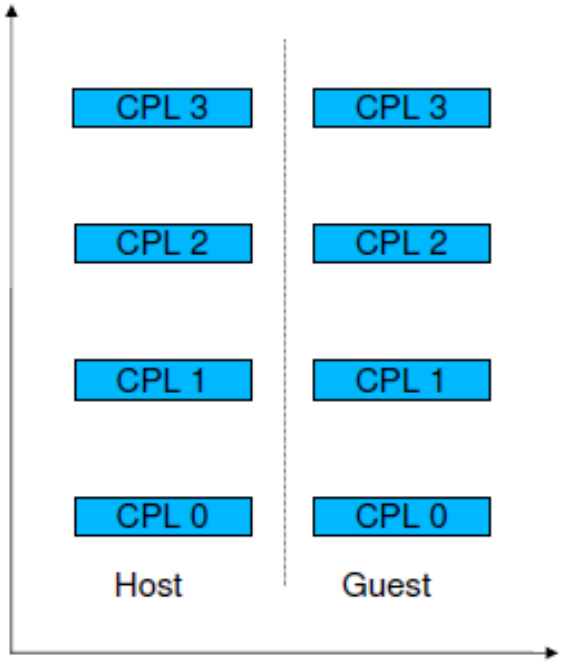



Four rings
For each mode

- VM Exit -> trap to hypervisor (enter host mode)
- VM run -> run the guest OS (enter guest mode)

# Hardware VMM

Trap sensible instruction, VM exit




## Diagram

- Y-axis: old school x86 privilege (CPL)
- X-axis: virtualization privilege

## Guest Mode

- Runs unmodified OS
- Sensitive operations "exit" (trap out) to host mode

## VMCB

- Virtual Machine Control Block
- VMM-controlled, hardware-walked
- Buffers simple exits

# Virtual Machine Control Structure (VMCS)

1. Data structure to manages VM entries and VM exits.
2. VMCS is logically divided into:
    1. Guest-state area=info on the VM CPU
    2. Host-state area.
    3. VM-execution control fields
    4. VM-exit control fields
    5. VM-entry control fields
    6. VM-exit information fields
3. VM entries = load processor state from the guest-state area.
4. VM exits = 1) save processor state to the guest-state area and the exit reason,
    2) load processor state from the host-state area.

# VT-x New instructions

VMXON, VMXOFF : To enter and exit VMX-root mode.

VMLAUNCH : initial transition from VMM to Guest , Enters VMX non-root operation mode

VMRESUME : Used on subsequent entries
Enters VMX non-root operation mode
Loads Guest state and Exit criteria from VMCS

VMEXIT : Used on transition from Guest to VMM
Enters VMX root operation mode
Saves Guest state in VMCS
Loads VMM state from VMCS

VMPTRST, VMPTRL: Read and Write the VMCS pointer.

VMREAD, VMWRITE, VMCLEAR : Read from, Write to and clear a VMCS.

# Solving Virtualization Challenges with VT-x

Address-Space Compression

With VT-x

– every transition between guest software and the VMM can change the linear-address space, allowing guest software full use of its own address space.

– VMX transitions are managed by the VMCS, which resides in the physical-address space, not the linear address space.

Ring Aliasing and Ring Compression

VT-x allow VMM to run guest software at its intended privilege level, this fact:

- Eliminates ring aliasing problems: an instruction such as PUSH (of CS) cannot reveal that software is running in a VM.
- Eliminates ring compression problems that arise when a guest OS executes at the same privilege level as guest applications

# Solving Virtualization Challenges with VT-x

Nonfaulting Access to Privileged State

VT-x avoid this problem in two ways:

- Generating VMExits on each sensitive execution
- Provides configuration of interrupts and exceptions disposition

Guest System Calls

Problems occur with the IA-32 instructions SYSENTER and SYSEXIT when guest OS run outside privilege level 0. This problem is solved because with VT-x, a guest OS can run at privilege level 0.

# Solving Virtualization Challenges with VT-x

Interrupt Virtualization

VT-x provide explicit support for interrupt virtualization

- It includes an external-interrupt exiting VM-execution control.
  - When this control is set to 1, a VMM prevents guest control of interrupt masking without gaining control of every guest attempt to modify EFLAGS.IF.
- It includes an interrupt-window exiting VM-execution control.
  - When this control is set to 1, a VM exit occurs whenever guest software is ready to receive interrupts. A VMM can set this control when it has a virtual interrupt to deliver to a guest.

Access to Hidden State

VT-x includes, in the guest-state area of the VMCS, fields corresponding to CPU state not represented in any software-accessible register.

- The processor loads values from these VMCS fields on every VM entry and saves into them on every VM exit.

# Solving Virtualization Challenges with VT-x

Frequent Access to Privileged Resources

VT-x allow a VMM to avoid the overhead of high-frequency guest access to the TPR (task priority register):

- A VMM can configure the VMCS so that the VMM is invoked only when required: when the value of the TPR shadow associated with the VMCS drops below that of a TPR threshold in the VMCS.

# Qualitative Comparison

Software wins in…

- Trap elimination via adaptive BT.
  - HW replaces traps w/ exits.
- Emulation speed.
  - Translations and call-outs essentially jump to pre-decoded emulation routines.
  - HW VMM must fetch VMCB and decode trapping instructions before emulating.

# Qualitative Comparison

Hardware wins in…

- Code density.
  - No translation = No replicated code segments
- Precise exceptions.
  - BT approach must perform extra work to recover guest state for faults and interrupts.
  - HW approach can just examine the VMCS.
- System calls.
  - [Can] run w/o VMM intervention.

# Qualitative Comparison (Summary)

Hardware VMMs…

- Native performance for things that avoid exits.
- However exits are still costly (currently).
  - Strongly targeted towards “trap-and-emulate” style.

Software VMMs…

- Carefully engineered to be efficient.
- Flexible (because it isn’t HW).

# More Quantitative Comparison

K. Adams and O. Agesen (2006). A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems*. ASPLOS-XII. ACM Press, New York, NY, 2-13.

- 3.8 GHz Intel Pentium 4.
- The contenders…
  - Mature commercial Software VMM.
  - Recently developed Hardware VMM.

# SPECint & SPECjbb

Primarily user-level computations.

Unaffected by VMMs

Therefore, performance should be near native.

Experimental results confirm this.

- 4% average slowdown for Software VMM.
- 5% average slowdown for Hardware VMM.

The cause is “host background activity”.

Windows test closer to native than Linux test.

# Benchmarks

a) A synthetic suite of microbenchmarks used to pinpoint various aspects of workstation performance

- Large RAM test - exhausts memory to test paging capability

  SW VMM wins.
- 2D Graphics test - hits system calls

  HW VMM wins.“

b) Less” synthetic test = Compilation time of Linux Kernel, Apache, etc.

SW VMM beats the HW VMM again.

Big compilation job w/ lots of files = Lots of page faults.

SW VMM is better at this than HW VMM.

Compared to native speed…

SW VMM is ~60% as fast.

HW VMM is ~55% as fast.

# ForkWait Test

```
int main(int argc, char *argv[]) {
  for (int i = 0; i < 40000; i++) {
    int pid = fork();
    if (pid < 0)  return -1;
    if (pid == 0) return 0;
    waitpid(pid);
  }
  return 0;
}
```

- Test to stress process creation/destruction.
  - System calls, context switching, page table modifications, page faults, context switching, etc.
- Native = 6.0 seconds.
- SW VMM = 36.9 seconds.
- HW VMM = 106.4 seconds.

# Conclusions

a) Hardware extensions now allow x86 to execute guests directly (trap-and-emulate style).

b) Comparison of SW and HW VMMs…

1) Both can execute computation-bound workloads at near native speed.
2) When I/O and process management is involved, SW prevails.
3) When there are a lot of system calls, HW prevails.

c) SW VMM techniques are very mature and very flexible.

d) New x86 extensions are relatively immature and present a fixed (inflexible) interface.

e) Future work on HW extensions promises to improve performance.

f) Hybrid SW/HW VMMs promise to provide benefits of both worlds.

g) There is no “clear” winner at this time.

# Paravirtualization

## Full Virtualization

- No modifications to guest OS
- Excellent compatibility, good performance, but complex

## Paravirtualization Exports Simpler Architecture

- Term coined by Denali project in '01, popularized by Xen
- Modify guest OS to be aware of virtualization layer
- Remove non-virtualizable parts of architecture
- Avoid rediscovery of knowledge in hypervisor
- Excellent performance and simple, but poor compatibility

## Ongoing Linux Standards Work

- "Paravirt Ops" interface between guest and hypervisor
- Small team from VMware, Xen, IBM LTC, etc.

# Two Ways to Handle Non-virtualizable Instructions

Paravitualization

- Modify VMM interface to use instructions that can be virtualized

  Xen, Denali
- Rewrite portions of the guest OS to delete this kind of instruction; replace with other instructions that are virtualizable.
- It affects the guest OS, but not applications that run on it – the API is unchanged

Binary Translation

Monitor execution of kernel code and dynamically replace non-virtualizable instructions with other instructions

# Paravirtualization