# Processor and Virtual Machine Security

## Fall 2024

R. Sekar

# Processor Security: Key Principles

- Processors operate at multiple privilege levels
  - *At least two levels needed: privileged and unprivileged*
  - Often, four or more levels supported.
    - Ring 0 is highest privilege
    - Ring 3 is lowest privilege
- OS kernel executes in privileged mode
- User level code executes in unprivileged mode
  - Applies to all processes, including those owned by root

# Processor Security: Key Principles

- Privileged instructions can execute successfully only if the processor is operating in privileged mode.
  - *Important processor state can be changed only through the execution of privileged instructions*
    - Page tables
    - I/O devices
- As a result, only the kernel code can change critical processor state.
  - Enables the OS to control and manage system resources and share them safely across user-level processes.
  - Resources are often "virtualized:" for most resources, it is as if a user level process has an exclusive, private copy of the resource.
    - memory, display, keyboard, ...

# Processor Security: Key Principles

- No control transfers across privilege levels
  - Can't secure privileged code if unprivileged code can call/jump to it
    - Difficult to get things right even for control transfers in the opposite direction!
- So, privileged crossings are usually effected via interrupts
  - hardware interrupts: often used to respond to device requests
  - software interrupts: system calls (user code calling kernel code)
- Interrupts are like request messages.
  - The sender does not have any ability to control whether the receiver examines or processes requests
  - Nor can they influence the environment in which they are processed
    - the registers, stack, heap etc. are separate for the kernel
    - kernel code can access user process memory, but it takes extreme care in doing so.

# Virtualization in OSes

- Creation of logical instances of physical resources.
  - Logical instances have the same functions
  - differ in size, performance, availability, cost etc.
  - often used to create a dedicated instance of a resource from a shared physical resource
- Resources to virtualize
  - CPU
  - Memory
  - I/O devices (mouse, display, network, ...)
- Some resources are shared using high level interfaces rather than virtualization, e.g., file system.
- OSes already virtualize most resources for user processes
  - Can we extend this so that the whole system is virtualized

# System Virtualization

- System virtualization creates several virtual systems within a single physical system
  - System = complete computer system, including the processor and all the peripherals contained within
  - This virtual should still provide privileged instructions, so that OS kernels can run on top.
- *VMM (or hypervisor)*
  - Virtual machine monitor is the software layer providing the virtualization.
- *VM* Virtual machine is the virtual system running on top of the VMM.

# Brief History

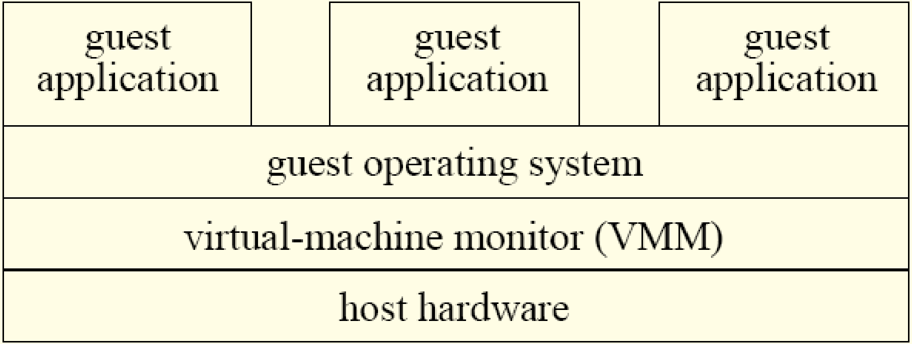
- 1960s, first introduced, for main frames
  - Motivation: hardware cost etc.
  - Dearth of knowledge on multi-user OS
- 1970s, an active research area
- 1980s, underestimated
  - Multitask modern operating systems took its place, and delivered better combination of price and performance
- late 1990s, resurgence: software techniques for x86 virtualization
  - Many motivations: mixed-OS develop environment, security, fault tolerance etc.
- since mid 2000s: hardware support from both Intel and AMD.

# Types of Virtualization

- Process virtualization (virtualize one process)
  - The VM supports an application binary interface: user instructions plus "system calls"
    - JVM, ...
- OS or Namespace virtualization (multiple logical VMs that share the same OS kernel)
  - Isolates VMs by partitioning all objects (not just files) into namespaces
  - Linux containers and vServer, Docker
- System (or full) virtualization (whole system: OS+apps)
  - The VM supports a complete ISA: user+system instructions
  - Classic VMs, whole system emulators (and many others we discuss in next slides)

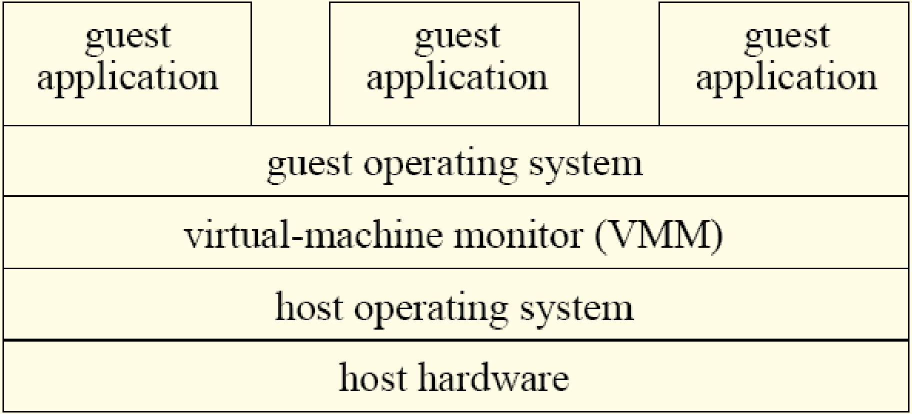
# VMM Architectures

Type I: The VMM runs on bare hardware (“bare-metal hypervisor”)



# VMM Architectures

Type II: The VMM runs as an ordinary application inside host OS (hosted hypervisor)

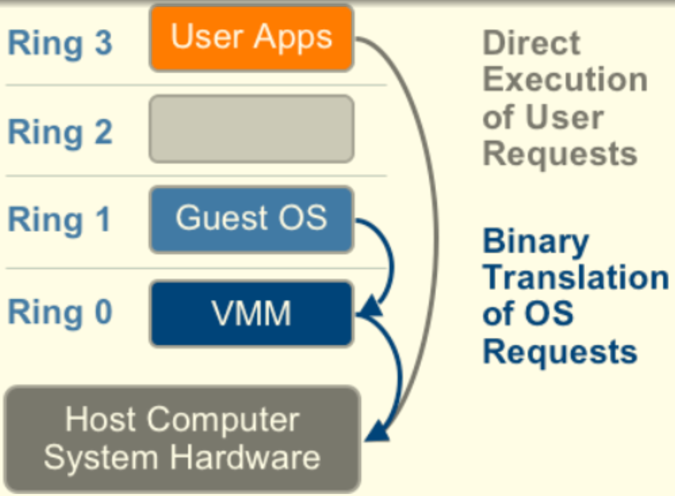


# Key Issues in CPU Virtualization

- Protection levels
  - Ring 0 (most privileged)
  - Ring 3 (user mode)
- Requirement for efficient/effective virtualization
  - Privileged instructions: Trap if executed in user mode
  - Sensitive instructions: affect important "system state"
  - If privileged==sensitive, can support efficient "trap and emulate" approach
    - Virtualized execution = native execution+exception handling code that emulates privileged instructions
- For x86, not all sensitive instructions are privileged
  - Some instructions simply exhibit different behaviors in user and privileged mode

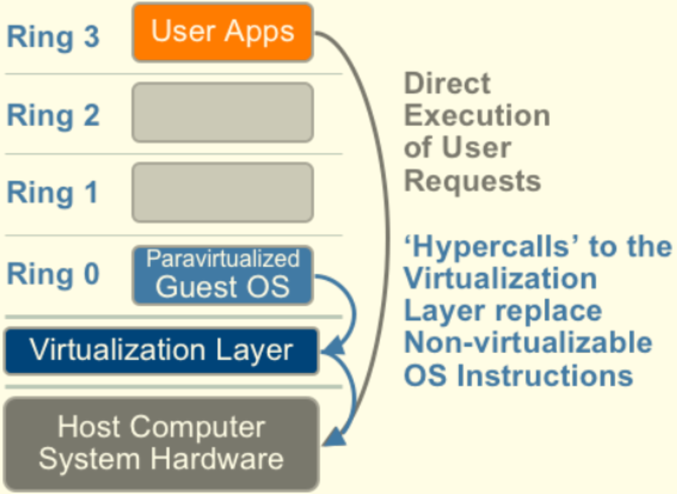
# Virtualization Approaches

- Full virtualization using binary translation:
  - Problem instructions translated into a sequence of instructions that achieve the intended function
  - Example: early VMware, QEMU
- Need to disassemble the binary, identify problem instructions and patch them
- Rely dynamic disassembly and translation in order to make disassembly tractable, and to support dynamic changes to code.



# Virtualization Approaches

- Paravirtualization: OS modified to run on VMM
  - Example: Xen

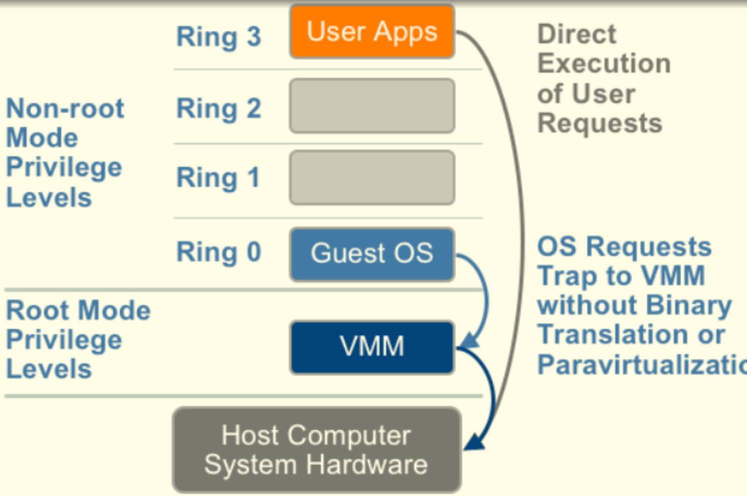


# Paravirtualization

- No longer 100% interface compatible, but better performance
  - Guest OSes must be modified to use VMM's interface
  - Note that ABI is unchanged
    - Applications need not to be modified
- Guest OSes are aware of virtualization
  - privileged instructions are replaced by hypervisor calls
  - therefore, no need for trapping or binary translation

# Virtualization Approaches

- Hardware-assisted virtualization
  - Most VMMs today



# Hardware-assisted virtualization

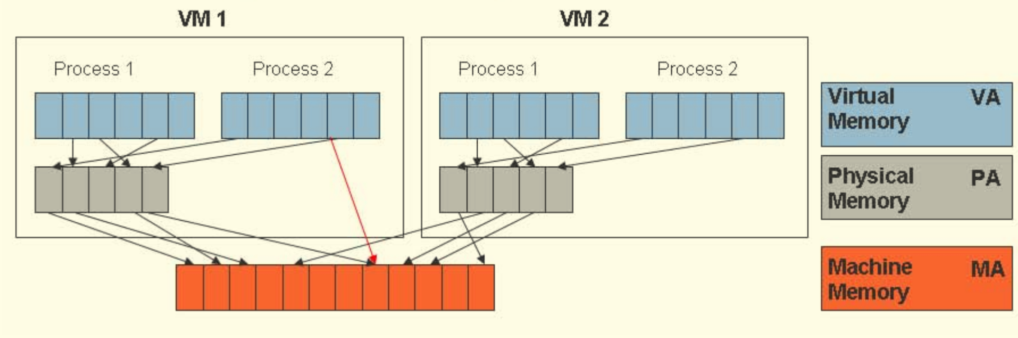
- Separates CPU execution into two modes
  - hypervisor executes in host mode
  - all VMs execute in guest mode
- Both hypervisor and VMs can execute in any of the four rings
- Hypervisor can
  - explicitly switch from host mode to guest mode
  - specify which events (e.g. interrupts) cause exist from guest mode

# Memory Virtualization

- Access to MMU needs to be virtualized
  - Otherwise guest OS may directly access physical memory and/or otherwise subvert VMM
- Physical Memory is divided among multiple VMs
  - Two levels of translation
    - Guest OS: guest virtual addr $\longrightarrow$ guest physical addr
    - VMM: guest physical addr $\longrightarrow$ machine addr

# Memory Virtualization

- Shadow page table needed to avoid 2-step translation
  - When guest attempts to update, VMM intercepts and emulate the effects on the corresponding shadow page table



# I/O Virtualization

- The VMM intercepts guest's I/O-performing instructions
- Performs necessary actions to emulate their effect.
  - Processor hardware cannot help that much here: support is provided using software within the VMM
  - This software "emulations" leads to low performance for most I/O operations.
- But CPU and memory operations perform near that of native code.

# VMs: Security Applications and Concerns

# Security Applications

- Honeypot systems and Malware analysis
  - VM technology provides strong isolation that is necessary to run malware without undue risks
    - Strong resource isolation: CPU, memory, storage
    - Snapshot/restore features to speed up testing and recovery
- High-assurance VMs
  - On a single workstation, can run high assurance VMs that support some security functions, but may not provide general-purpose functions
    - single-purpose VM scheme facilitates stricter security policies
    - In contrast, security policies that are compatible with the range of desktop applications being used today will likely be too permissive.

# More security applications

- Protection from compromised OSes
  - Modern OSes are too complex to secure
  - Malware-infested OS may subvert security software (virus and malware scanners)
  - Instead, rely on VMM
    - run malware and rootkit detection techniques in VMM
    - enforce security properties from within the VMM

# Security Challenges in Virtualized Environments

Virtualization leads to co-tenancy

- VMs belonging to distinct principals use the same hardware
  - Strong isolation is necessary or else attacks become too easy
  - Containers don't offer enough security if some principals can be downright malicious
- Even with strong isolation, provides increased opportunities for side-channel attacks
- Denial of service is difficult to prevent
  - But often, it is not a problem in practice as bad behavior is expensive, and/or is detected and the culprit punished

# Docker Security

- Isolation of containers
  - namespaces: each container cannot see entities (files, processes, pids, network interfaces, ...) in other containers
  - Linux cgroups: enable resource accounting and limiting — including CPU, memory, disk I/O, etc.
    - one bad container cannot use up all resources
- Container infrastructure and services (docker daemon)
  - containers can share files/directories with the host OS, but this can be dangerous, e.g., allow root user in a container to change critical host OS files
  - administrative services (e.g., creation of containers) can be abused, so interface to docker daemon should be restricted

# Docker Security: Attack Vectors

- Shared kernel
  - Same OS kernel across different containers
    - May also be the same kernel as the host OS
  - Any kernel vulnerabilities may be exploited
    - Bugs in namespace isolation
    - Bugs in syscall implementations
- Docker infrastructure needs root privileges
  - Malicious processes (on host) may abuse this privilege
- Applications running within Docker may extend their reach to the host
  - By using shared folders
  - Root processes inside container can possibly execute syscalls as root on host

# Docker security practices

- Avoid root privilege
  - Use user namespaces to map docker root to non-zero uid
- Limit further using Linux capabilities
  - programs running within containers typically don't need root privilege
  - we can use Linux capabilities to take away almost all of the power of the root
- Limit further using seccomp-bpf to limit system calls that can be made by processes within the container
- And the most important of them all:
  - Avoid using software that you can't reasonably trust to be non-malicious