CSCI3150 Introduction to Operating Systems

Lecture 16: Virtualization

Hong Xu

https://github.com/henryhxu/CSCI3150

Acknowledgement

- Based on slides from Brad Cambell@U. Virginia for CS6456

What is virtualization?

- Virtualization is the ability to run multiple operating systems on a single physical system and share the underlying hardware resources[1]
- Allows one computer to provide the appearance of many computers.
- Goals:
  - Provide flexibility for users
  - Amortize hardware costs
  - Isolate completely separate users

[1] VMWare white paper, *Virtualization Overview*

Requirements for Virtualizable Architectures

- "First, the VMM provides an environment for programs which is essentially identical with the original machine;
- second, programs run in this environment show at worst only minor decreases in spe ed;
- and last, the VMM is in complete control of system resources."
- VMM: virtual machine monitor (from VMware); a.k.a. hypervisor

VMM Platform Types

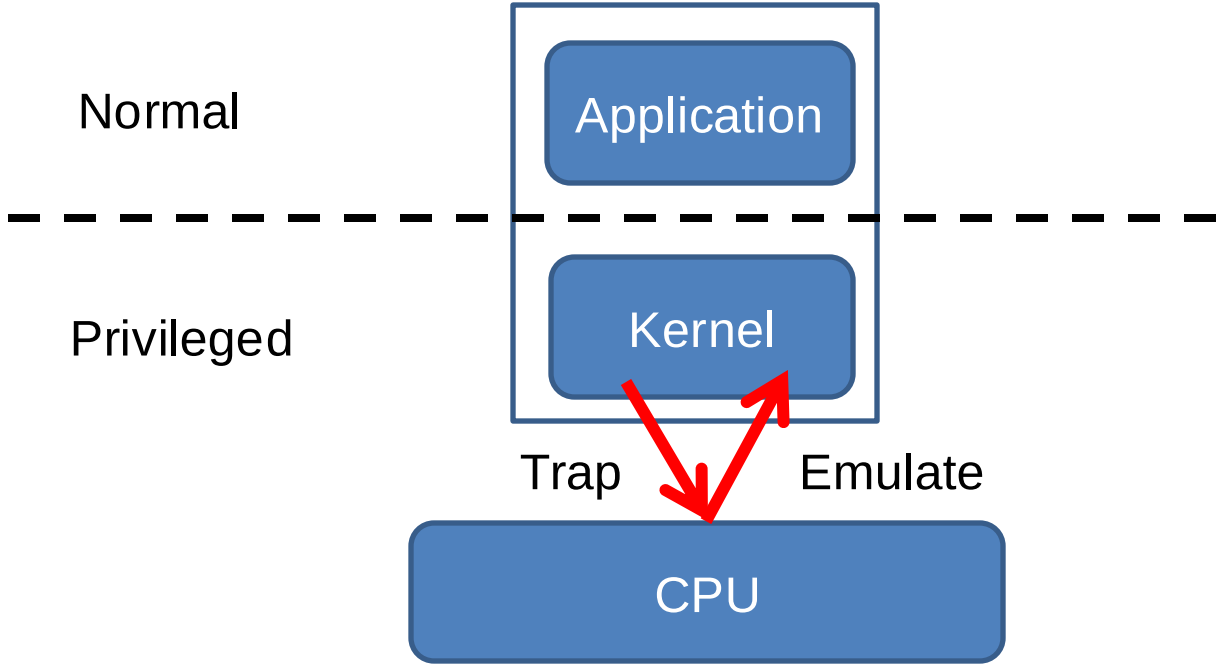
- **Hosted Architecture**
- Install as application on existing x86 "host" OS, *e.g.* Windows, Linux, OS X
- Small context-switching driver
- Leverage host I/O stack and resource management
- Examples: VMware Player/Workstation/Server, Microsoft Virtual PC/Server, Parallels Desktop
- **Bare-Metal Architecture**
- "Hypervisor" installs directly on hardware
- Acknowledged as preferred architecture for high-end servers
- Examples: VMware ESX Server, Xen, Microsoft Viridian (2008)

Virtualization: rejuvenation

- 1960's: first track of virtualization
  - Time and resource sharing on expensive mainframes
  - IBM VM/370
- Late 1970s and early 1980s: became unpopular
  - Cheap hardware and multiprocessing OS
- Late 1990s: became popular again
  - Wide variety of OS and hardware configurations
  - VMWare
- Since 2000: hot and important
  - Cloud computing
  - Docker containers

IBM VM/370

- Technology: trap-and-emulate



Trap and Emulate Virtualization on x86 architecture

- Challenges
  - Correctness: not all privileged instructions produce traps!
  - Performance:
    - System calls: traps in both enter and exit (10X)
    - I/O performance: high CPU overhead
    - Virtual memory: no software-controlled TLB

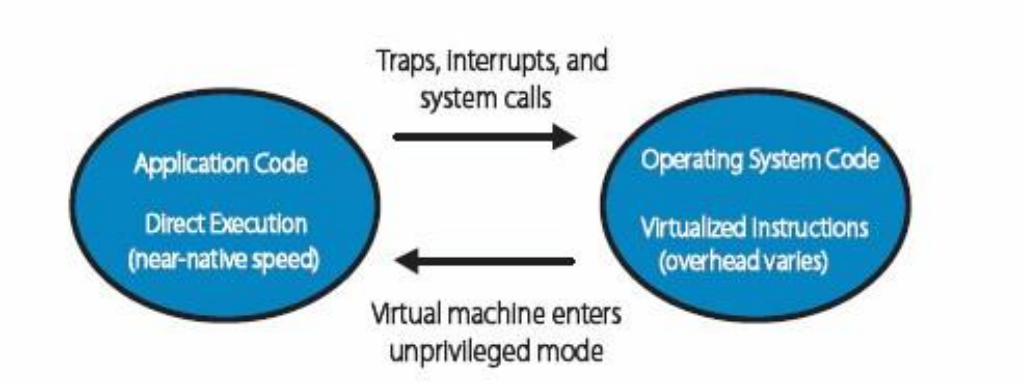
Virtualization on x86 architecture

- Solutions:
  - Dynamic binary translation & shadow page table
  - Para-virtualization (Xen)
  - Hardware extension

Dynamic binary translation

- Idea: intercept privileged instructions by changing the binary
- Cannot patch the guest kernel directly (would be visible to guests)
- Solution: make a copy, change it, and execute it from there
  - Use a cache to improve the performance

Binary translation

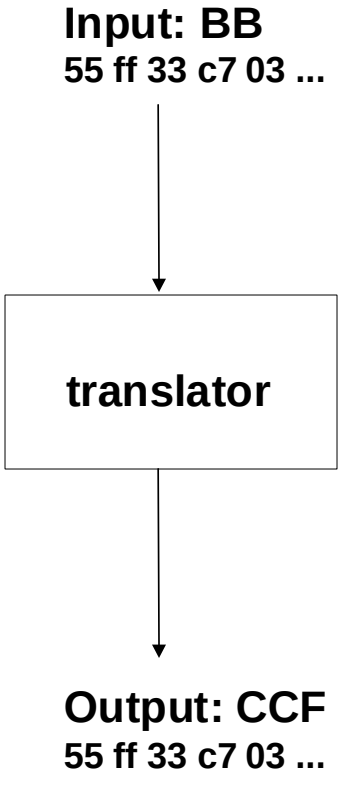


- **Directly execute unprivileged guest application code**
- Will run at full speed until it traps, we get an interrupt, etc.
- **"Binary translate" all guest kernel code, run it unprivileged**
- Since x86 has non-virtualizable instructions, *proactively* transfer control to the VMM (no need for traps)
- Safe instructions are emitted without change
- For "unsafe" instructions, emit a controlled emulation sequence
- VMM translation cache for good performance

How does VMWare do this?

- **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

Convert unsafe operations and cache them



Each Translator Invocation

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

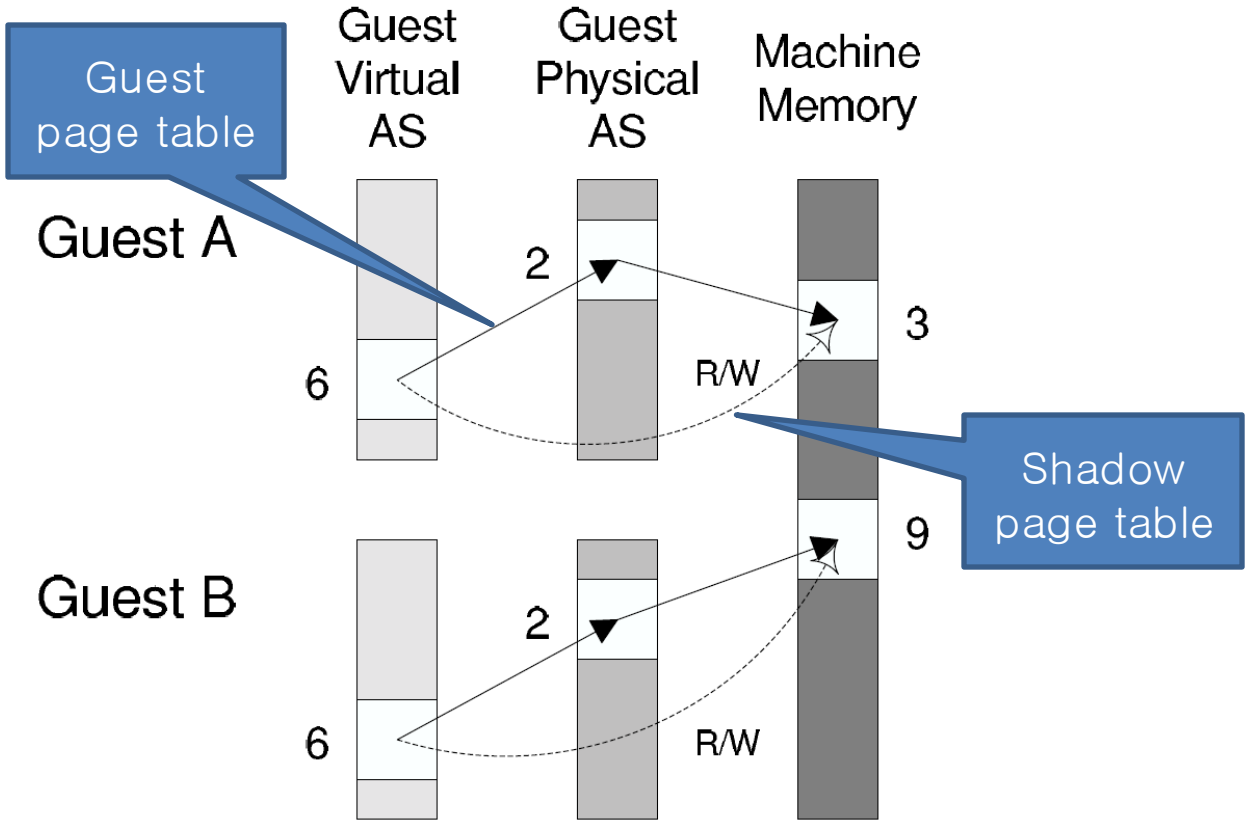
Store CCF in Translation Cache

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

Dynamic binary translation

- Pros:
  - Make x86 virtualizable
  - Can reduce traps
- Cons:
  - Overhead
  - Hard to improve system calls, I/O operations
  - Hard to handle complex code

Shadow page table



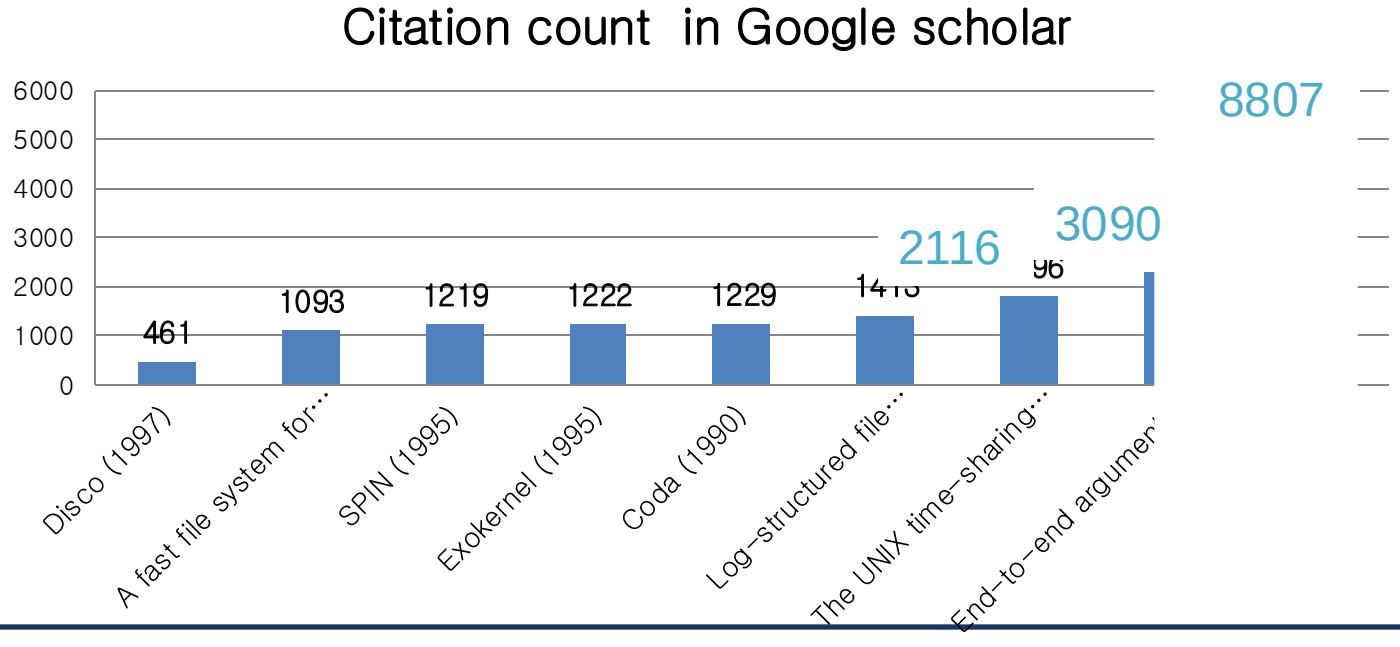
Shadow page table

- Pros:
  - Transparent to guest VMs
  - Good performance when working set is stable
- Cons:
  - Big overhead of keeping two page tables consistent
  - Introducing more issues: hidden fault, double paging …

Xen

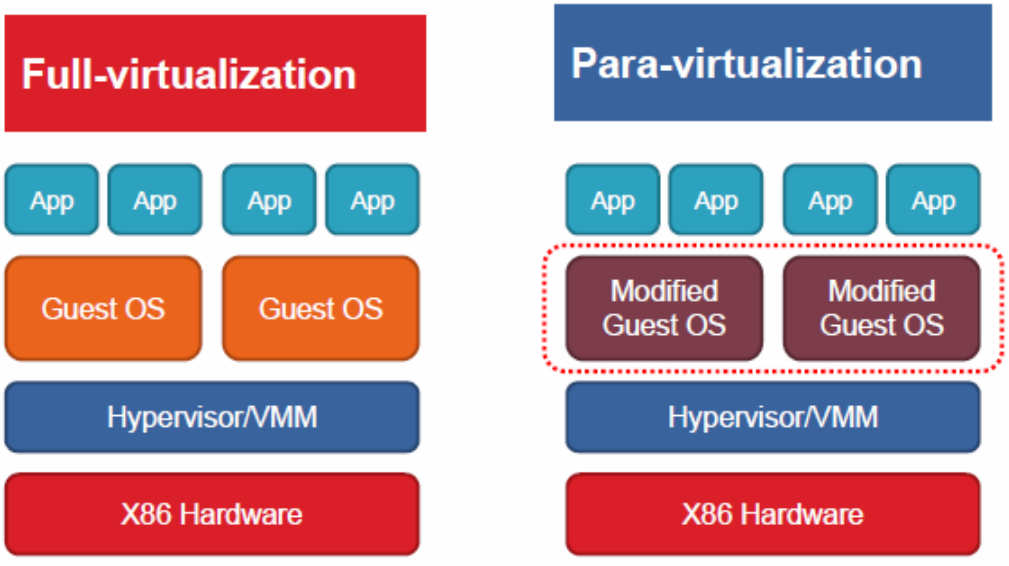
Xen and the art of virtualization

- SOSP'03
- Very high impact (data collected in 2013)



Para-virtualization

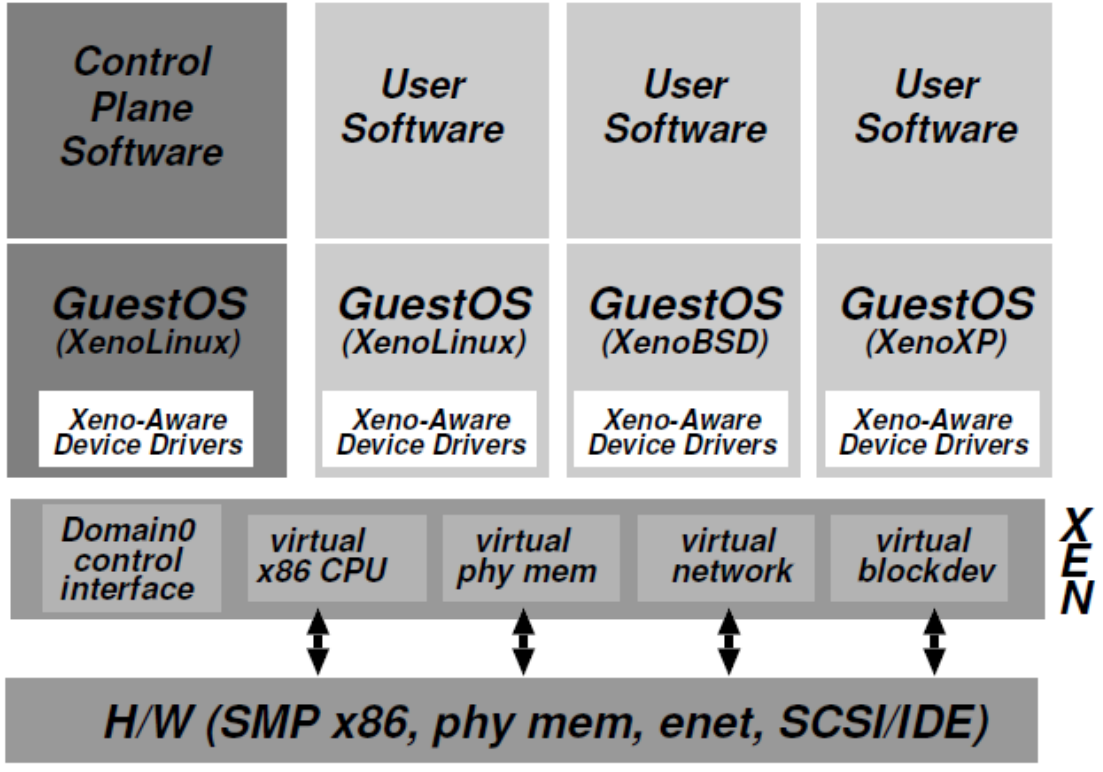
- Full vs. para virtualization



Overview of the Xen approach

- Support for unmodified application binaries (but not OS)
  - Keep Application Binary Interface (ABI)
- Modify guest OS to be aware of virtualization
  - Get around issues of x86 architecture
  - Better performance
- Keep hypervisor as small as possible
  - Device driver is in Dom0

Xen architecture



Virtualization on x86 architecture

- Challenges
  - Correctness: not all privileged instructions produce traps!
  - Performance:
    - System calls: traps in both enter and exit (10X)
    - I/O performance: high CPU overhead
    - Virtual memory: no software-controlled TLB

CPU virtualization

- Protection
  - Xen in ring0, guest kernel in ring1
  - Privileged instructions are replaced with hypercalls
- Exception and system calls
  - Guest OS registers handlers validated by Xen
  - Allowing direct system calls from app into guest OS
  - Page faults: redirected by Xen

Memory virtualization

- Xen exists in a 64MB section at the top of every address space
- Guest sees the mapping to real machine address
- Guest kernels are responsible for allocating and managing the hardware page tables.
- After registering the page table to Xen, all subsequent updates must be validated.

Porting effort is quite low

OS subsection	# lines	
	Linux	XP
Architecture-independent	78	1299
Virtual network driver	484	–
Virtual block-device driver	1070	–
Xen-specific (non-driver)	1363	3321
Total	**2995**	**4620**
(Portion of total x86 code base	**1.36%**	**0.04%)**

Table 2: The simplicity of porting commodity OSes to Xen. The cost metric is the number of lines of reasonably commented and formatted code which are modified or added compared with the original x86 code base (excluding device drivers).

Evaluation

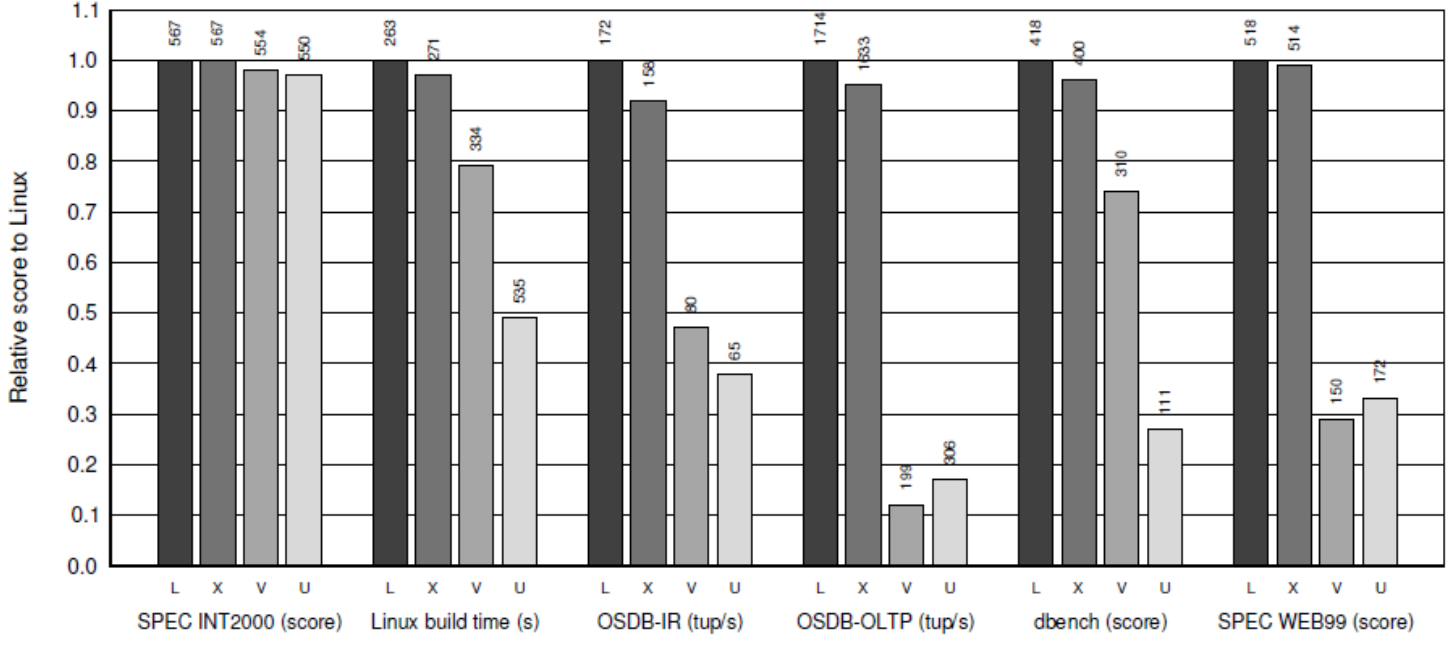


Figure 3: Relative performance of native Linux (L), XenoLinux (X), VMware workstation 3.2 (V) and User-Mode Linux (U).

Conclusion

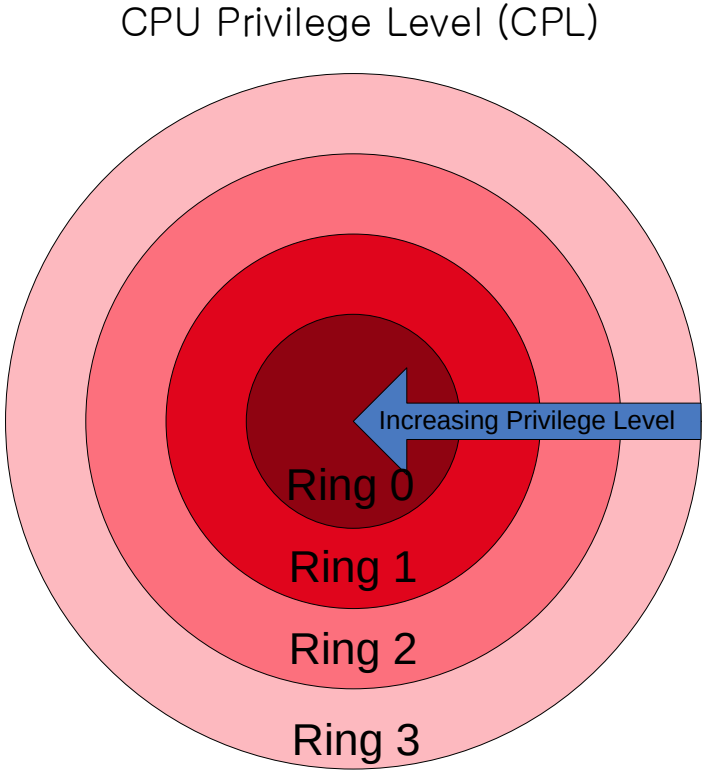
- x86 architecture makes virtualization challenging
- Full virtualization
  - unmodified guest OS; good isolation
  - Performance issue (especially I/O)
- Para virtualization:
  - Better performance (potentially)
  - Need to update guest kernel
- Full and para virtualization will keep evolving together

Instead: Leverage hardware support

- First generation - processor
- Second generation - memory
- Third generation – I/O device
  - In progress

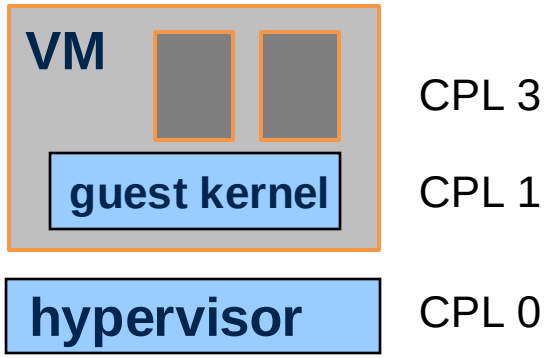
Protection Rings

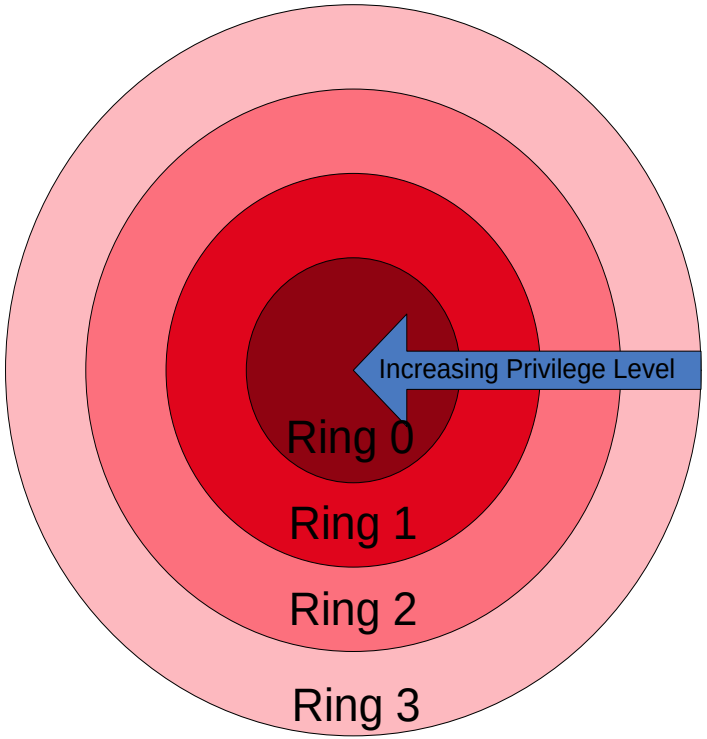
- Actually, x86 has four protection levels, not two (kernel/user).
- X86 rings (CPL)
  - Ring 0 – “Kernel mode” (most privileged)
  - Ring 3 – “User mode”
  - Ring 1 & 2 – Other
- Linux only uses 0 and 3.
  - “Kernel vs. user mode”
- Pre-VT Xen modified to run the guest kernel to Ring 1: reserve Ring 0 for hypervisor.



[Fischbach]

Why aren’t protection rings good enough?





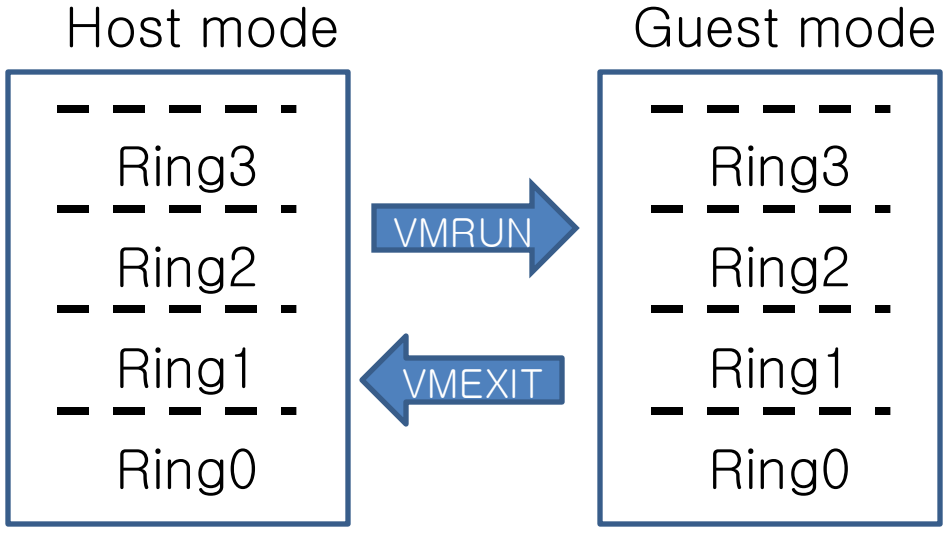
A short list of pre-VT problems

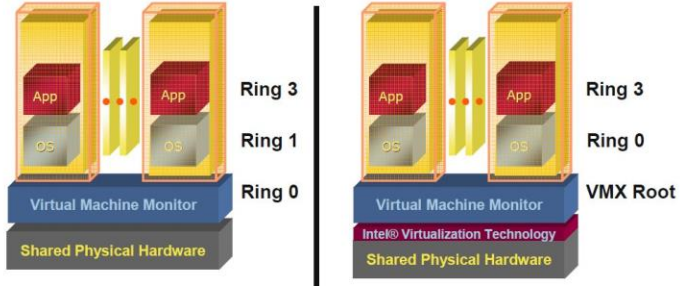
Early hypervisors (VMware, Xen) had to emulate various machine behaviors and generally bend over backwards.

- IA32 page protection does not distinguish CPL 0-2.
  - Segment-grained memory protection only.
- Ring aliasing: some IA instructions expose CPL to guest!
  - Or fail silently…
- Syscalls don't work properly and require emulation.
  - sysenter always transitions to CPL 0. (D'oh!)
  - sysexit faults if the core is not in CPL 0.
- Interrupts don't work properly and require emulation.
  - Interrupt disable/enable reserved to CPL0.

First generation: Intel VT-x & AMD SVM

- Eliminating the need of binary translation or modifying OSes



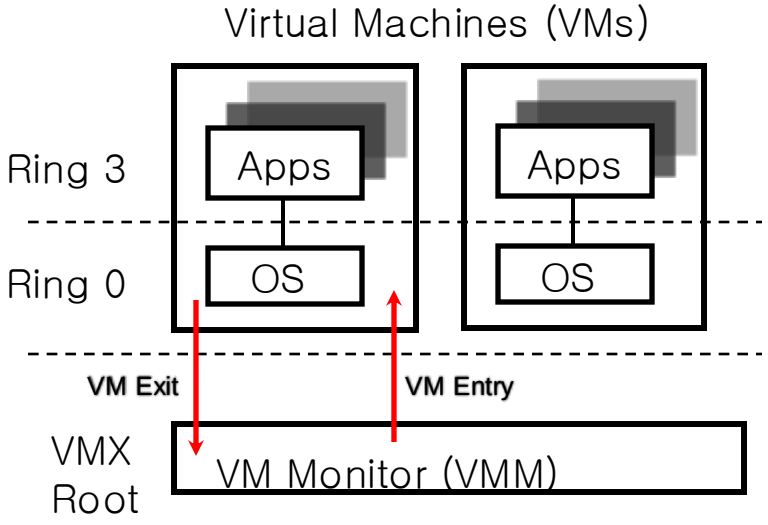


VT in a Nutshell

- New VM mode bit, Orthogonal to CPL (e.g., kernel/user mode)
- If VM mode is off → host mode
  - Machine "looks just like it always did" ("VMX root")
- If VM bit is on → guest mode
  - Machine is running a guest VM: "VMX non-root mode"
  - Machine "looks just like it always did" to the guest, BUT:
  - Various events trigger gated entry to hypervisor (in VMX root)
  - A "virtualization intercept": exit VM mode to VMM (VM Exit)
  - Hypervisor (VMM) can control which events cause intercepts
  - Hypervisor can examine/manipulate guest VM state and return to VM (VM Entry)

CPU Virtualization With VT-x

- Two new VT-x operating modes
  - Less-privileged mode (VMX non-root) for guest OSes
  - More-privileged mode (VMX root) for VMM
- Two new transitions
  - VM entry to non-root operation
  - VM exit to root operation



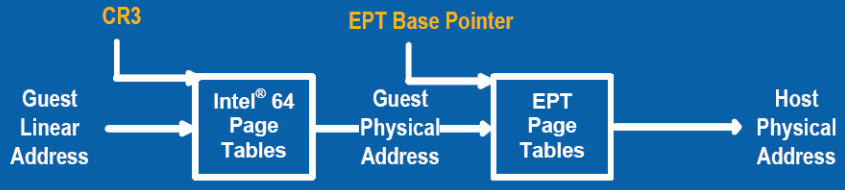
- Execution controls determine when exits occur
  - Access to privilege state, occurrence of exceptions, etc.
  - Flexibility provided to minimize unwanted exits
- VM Control Structure (VMCS) controls VT-x operation
  - Also holds guest and host state

Second generation: Intel EPT & AMD NPT

- Eliminating the need to shadow page table

Future Extensions: EPT

EPT: Overview



- Intel® 64 page tables
  - Map guest-linear to guest-physical (translated again)
  - Can be read and written by guest
- New EPT page tables under VMM control
  - Map guest-physical to host-physical (accesses memory)
  - Referenced by new EPT base pointer
- No VM exits due to page faults, INVLPG, or CR3 accesses

8

Containers and isolation

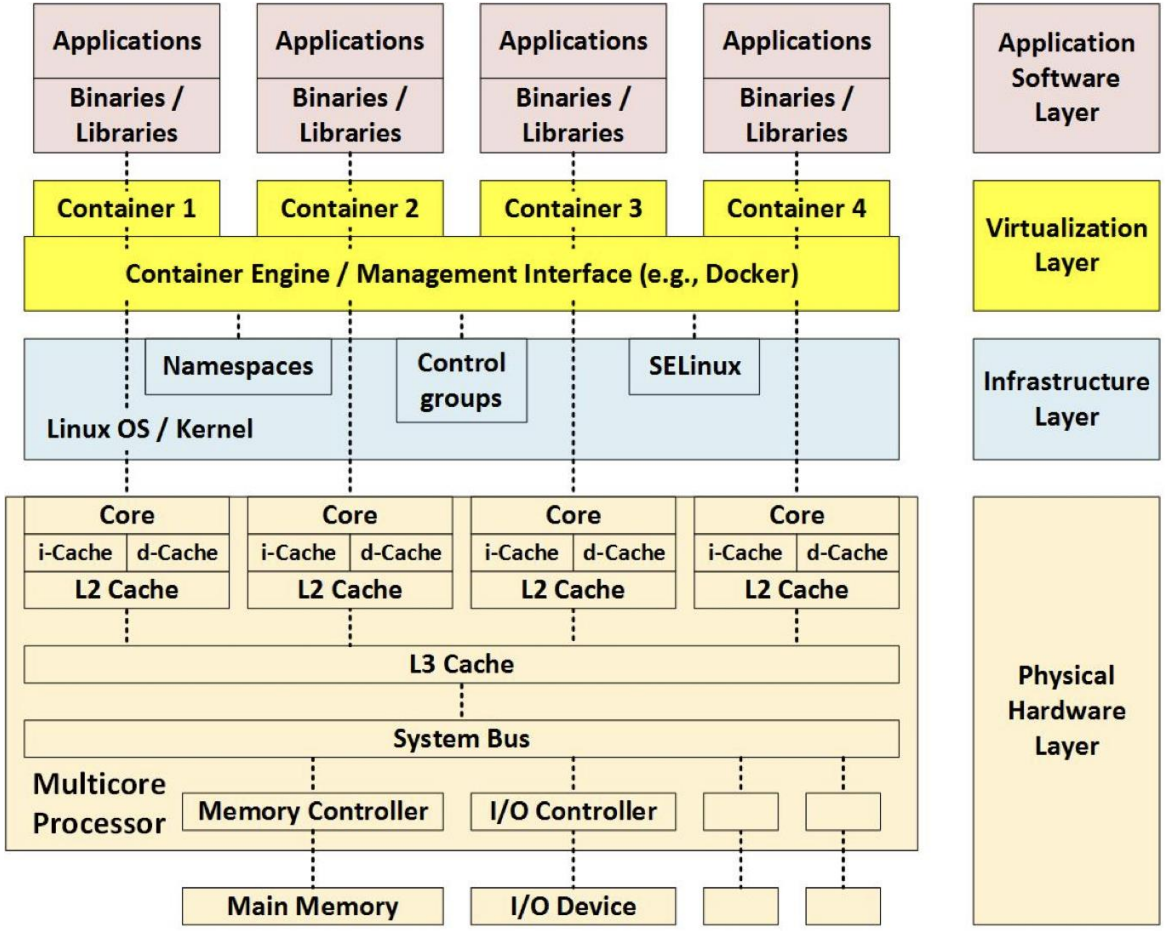
Containers: idea

- Run a process with a restricted view of system resources
  - Hide other processes
  - Limit access to system resources
- Not full virtualization
  - Process must use existing kernel and OS
- Benefits
  - Consistent environment (runtime, dependencies, etc.)
  - Low overhead, rapid launch/terminate
  - Performance

Pre-container isolation features in Linux

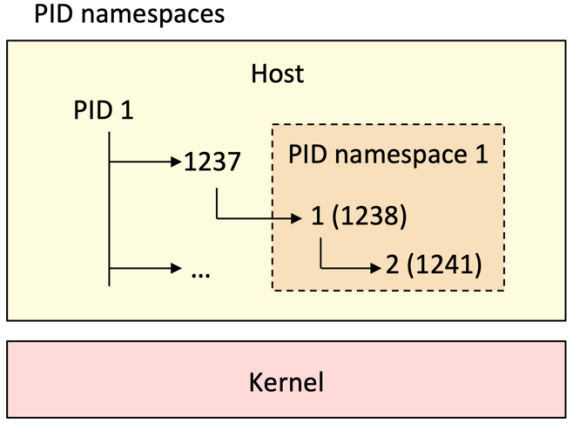
- chroot
  - Set the current root directory for processes
  - Added to Unix in 1979
- Namespaces
  - Provide processes with their own view of resources
    - Process IDs, networking sockets, hostnames, etc.
  - Akin to virtual address spaces
  - Originally introduced in 2002
- Copy-on-Write Filesystem
  - Allow a process to view existing filesystem, but any modifications result in copies then updates
  - Akin to virtual memory after fork

Key technologies



Namespace

- "A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource." – Linux man-pages
- Controls what resources a process can see, and what they are called
- Restrict visibility -> solve "inconsistent distribution"

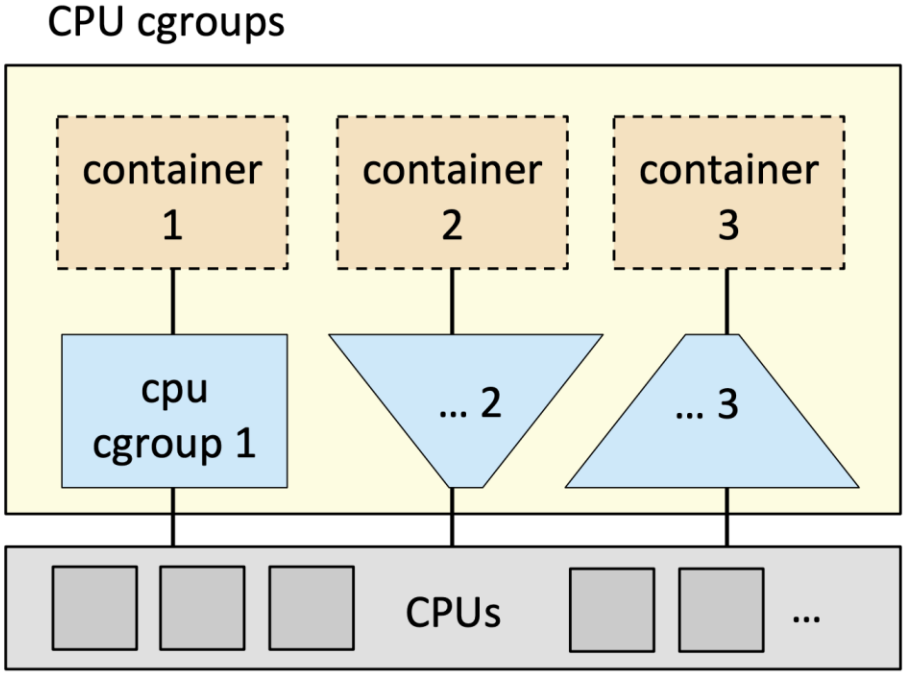


cgroups in Linux

- Enforce policies to restrict how processes use resources
  - Example: A process can only use 1 Mbps of network bandwidth
- Policies are enforced on groups of processes
- Controllers enforce various processes
- Restrict usage -> solves “resource isolation”

Controllers enforce restrictions for cgroups

- Different controllers for different system resources
- Each provides a policy for how processes should be restricted



Controllers in Linux

- *io*
  - Limit I/O requests either capped per process or proportionally.
- *memory*
  - Enforce memory caps on processes
- *pids*
  - Limit number of new processes in a cgroup
- *perf_event*
  - Allow monitoring performance
- *cpu*
  - Limit CPU usage when CPU is busy
- *freezer*
  - Allow suspending all processes in a cgroup

Linux containers = combinations of namespaces & cgroups