Virtualization

CS 4740: Cloud Computing

Fall 2024

Lecture 13

Yue Cheng

University of Virginia

Some material taken/derived from:

- IIT Bombay CS 695 by Mythili Vutukuru

Virtualization techniques

Overall objectives

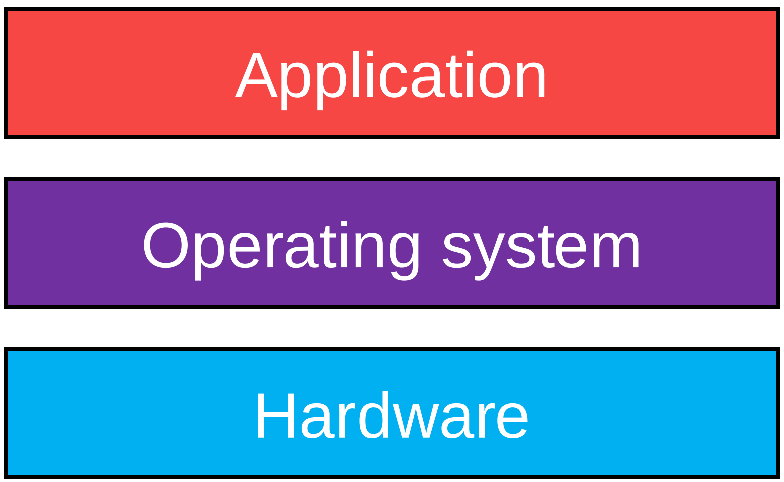
1. Isolation
2. Performance

Virtualization techniques

- Process-level virtualization
- OS-level virtualization
- Whole-machine virtualization

Virtualization techniques

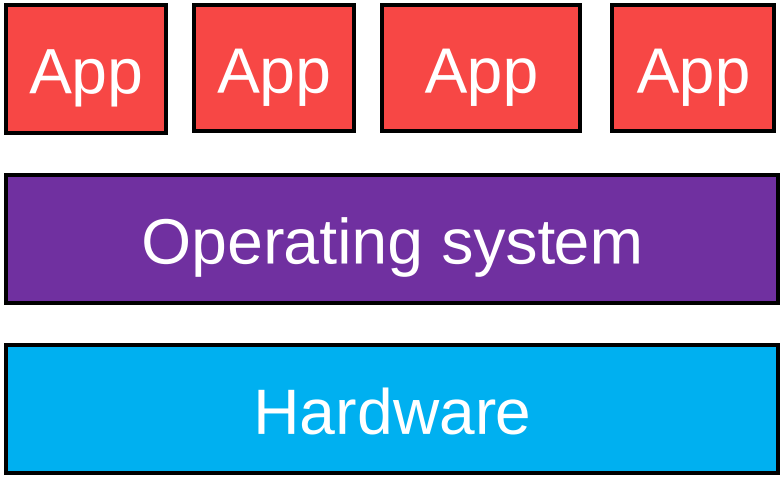
- Process-level virtualization



Virtual memory
CPU scheduling
File system management

Virtualization techniques

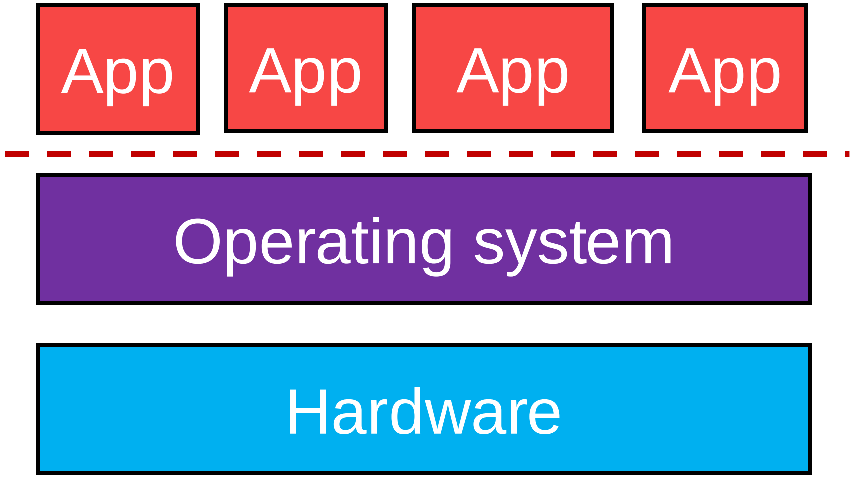
- Process-level virtualization



Virtual memory
CPU scheduling
File system management

Virtualization techniques

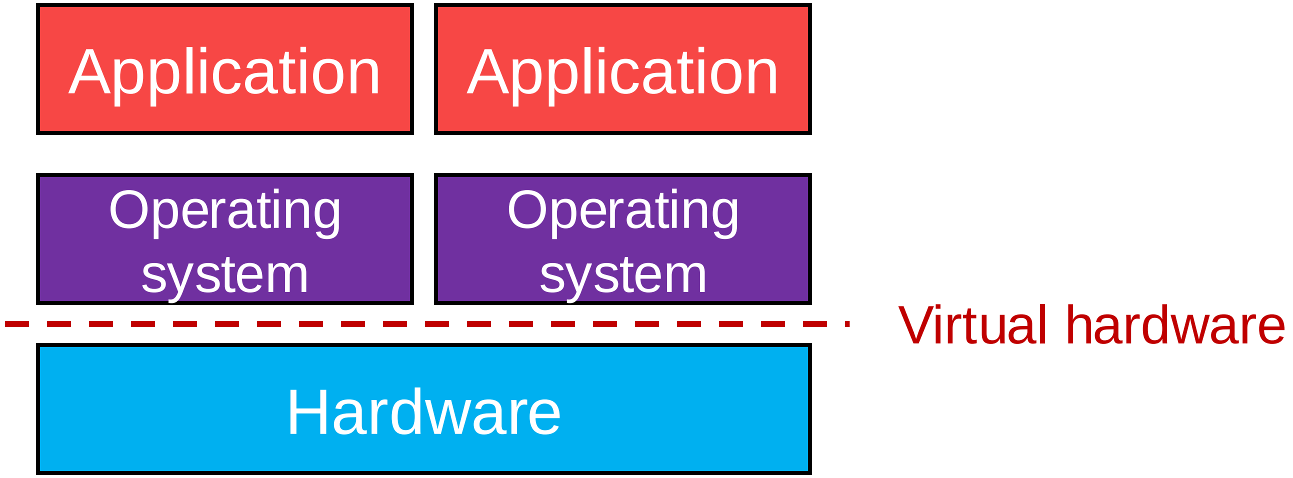
- OS-level virtualization



Sharing the OS, but uses OS features (cgroups, namespaces) for isolation

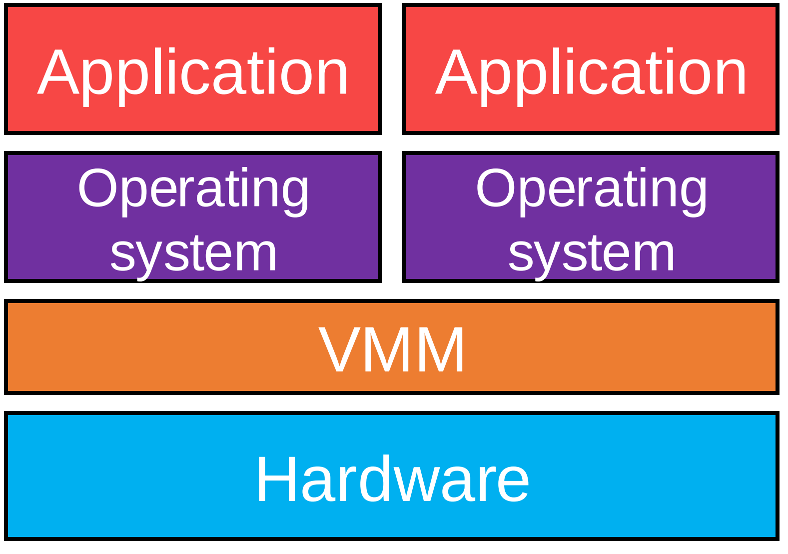
Virtualization techniques

- Whole-machine virtualization

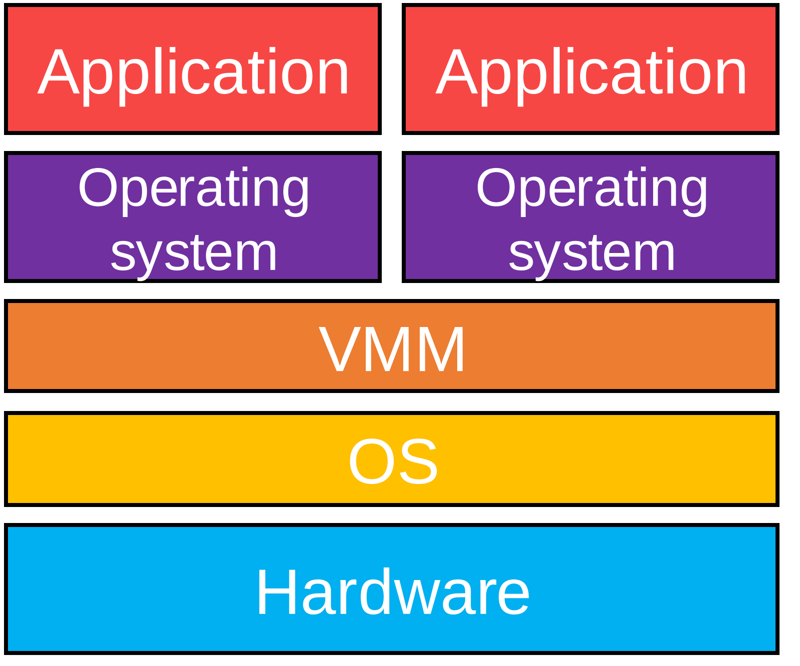


Virtual machines (VMs)

Virtual machine monitor (VMM)



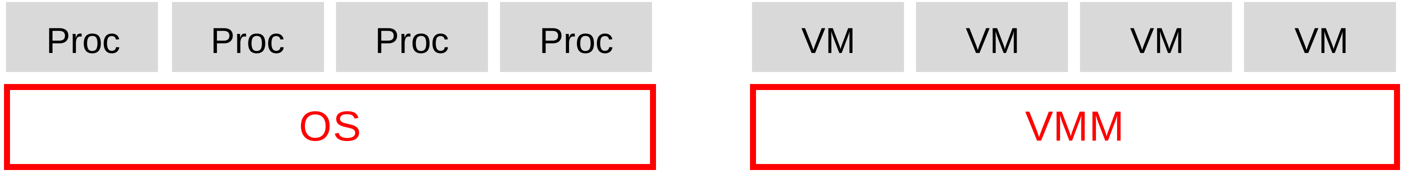
Type 1 hypervisor



Type 2 hypervisor

Virtual machine monitor (VMM)

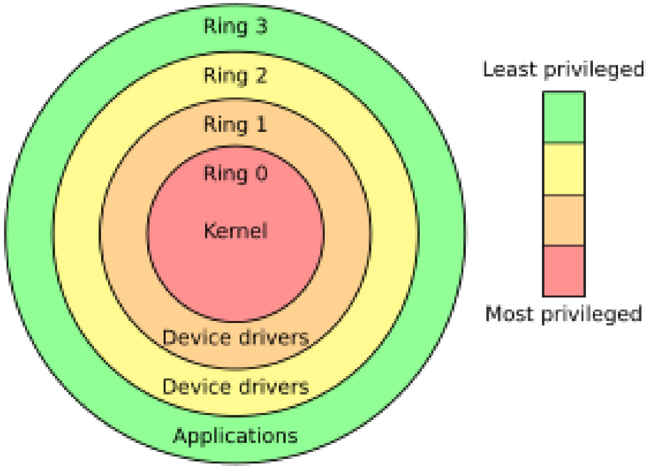
- Multiple VMs running on a physical machine (PM) – multiplexing the underlying machine
  - Similar to how OS multiplexes processes on CPU

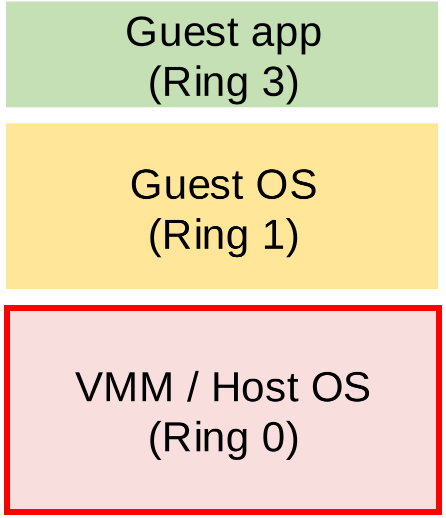


- VMM performs VM switch (much like process context switch)
  - Runs a VM a bit, save context and switch to another VM, and so on…
- What's the problem?
  - Guest OS expects to have unrestricted access to hardware, runs privileged instructions, unlike user processes

Trap and emulate VMM

- All CPUs have multiple privilege levels
  - Ring 0,1,2,3 in x86 CPUs
- Normally, user process runs in Ring 3, OS in Ring 0
  - Privileged instructions only run in Ring 0
- With VMM, user process in Ring 3, VMM/host OS in Ring 0
  - Guest OS must be protected from guest apps
  - But not fully privileged like host OS/VMM
  - Let guest OS run in Ring 1?
- Trap-and-emulate VMM: Guest OS runs at lower privilege level than VMM, traps to VMM for privileged operation

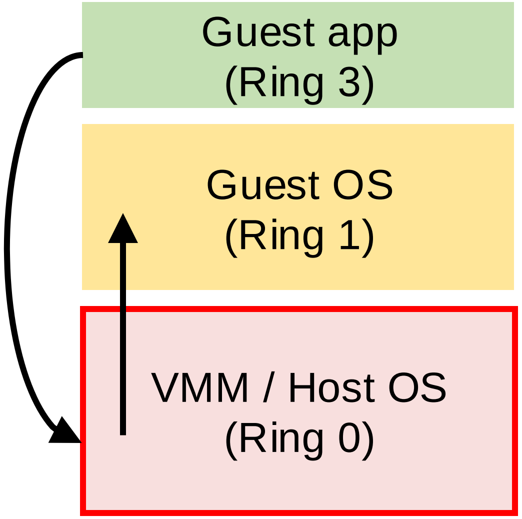




https://en.wikipedia.org/wiki/Protection_ring

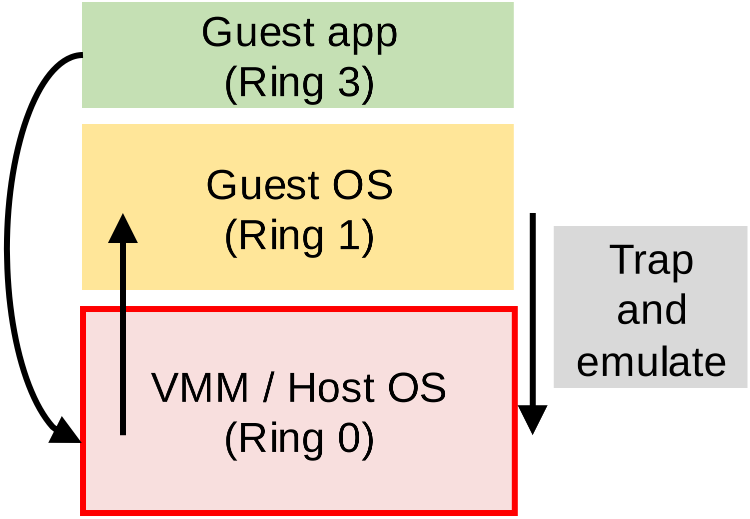
Trap and emulate VMM (cont.)

- Guest app needs to trigger syscall/interrupt
  - Special trap instr (`int n`), traps to VMM
  - VMM doesn't know how to handle trap
  - VMM jumps to guest OS trap handler
  - Trap handled by guest OS normally



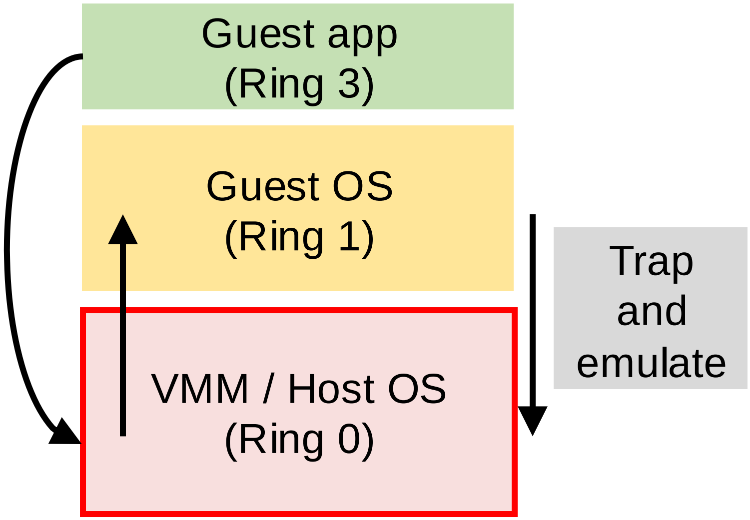
Trap and emulate VMM (cont.)

- Guest app needs to handle syscall/interrupt
  - Special trap instr (`int n`), traps to VMM
  - VMM doesn't know how to handle trap
  - VMM jumps to guest OS trap handler
  - Trap handled by guest OS normally
- Guest OS performs return from trap
  - Privileged instr, traps to VMM
  - VMM jumps to corresponding user process



Trap and emulate VMM (cont.)

- Guest app needs to handle syscall/interrupt
  - Special trap instr (`int n`), traps to VMM
  - VMM doesn't know how to handle trap
  - VMM jumps to guest OS trap handler
  - Trap handled by guest OS normally
- Guest OS performs return from trap
  - Privileged instr, traps to VMM
  - VMM jumps to corresponding user process
- Any privileged action by guest OS traps to VMM, emulated by VMM
  - Example: set IDT (`lidt`), set CR3, access hardware, modify hardware state
  - Sensitive data structures like IDT must be managed by VMM, not guest OS



Problems with trap and emulate

- Guest OS may realize it is running at lower privileged level
  - Some registers in x86 reflect CPU privilege level (code segment/CS register)
    - E.g., in x86, 0 indicates Ring 0, while 3 indicates Ring 3
  - Guest OS can read these values and get offended!

Problems with trap and emulate

- Guest OS may realize it is running at lower privileged level
  - Some registers in x86 reflect CPU privilege level (code segment/CS register)
  - Guest OS can read these values and get offended!
- Some x86 instructions that change hardware state (sensitive instructions) run in both privileged and unprivileged modes
  - Behaves differently when guest OS in Ring 0 vs. in less privileged Ring 1
  - OS behaves incorrectly in Ring 1, will not trap to VMM

Problems with trap and emulate

- Guest OS may realize it is running at lower privileged level
  - Some registers in x86 reflect CPU privilege level (code segment/CS register)
  - Guest OS can read these values and get offended!
- Some x86 instructions that change hardware state (sensitive instructions) run in both privileged and unprivileged modes
  - Behaves differently when guest OS in Ring 0 vs. in less privileged Ring 1
  - OS behaves incorrectly in Ring 1, will not trap to VMM
- Legacy reasons
  - OSes not originally designed to run at a lower privilege level
  - Instruction set architecture (ISA) of x86 is not easily virtualizable (x86 wasn't designed with virtualization in mind)

Example

- `EFLAGS` register is a set of CPU flags
  - `IF` (interrupt flag) indicates if interrupts on/off
- Consider the `popf` instruction in x86
  - Pops a value from top of stack and set `EFLAGS`
- If executed in Ring 0, all flags set normally
- If executed in Ring 1, only some flags set
  - `IF` is not set as it is privileged flag
- `popf` is a sensitive instruction, not privileged, does not trap, behaves differently when executed in different privilege levels
  - Guest OS is buggy in Ring 1

https://en.wikipedia.org/wiki/FLAGS_register

Popek Goldberg theorem

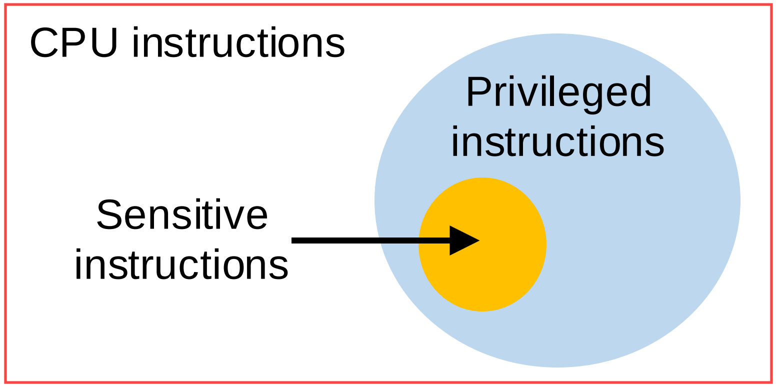
https://www.scs.stanford.edu/21wi-cs140/sched/readings/virtualization.pdf

- Sensitive instruction = changes hardware state
- Privileged instruction = runs only in privileged mode
  - Traps to Ring 0 if executed from unprivileged rings
- To build a VMM efficiently via trap-and-emulate method, sensitive instructions should be a subset of privileged instructions
  - x86 does not satisfy this criteria, so trap and emulate VMM is not possible with x86

Popek Goldberg theorem

https://www.scs.stanford.edu/21wi-cs140/sched/readings/virtualization.pdf

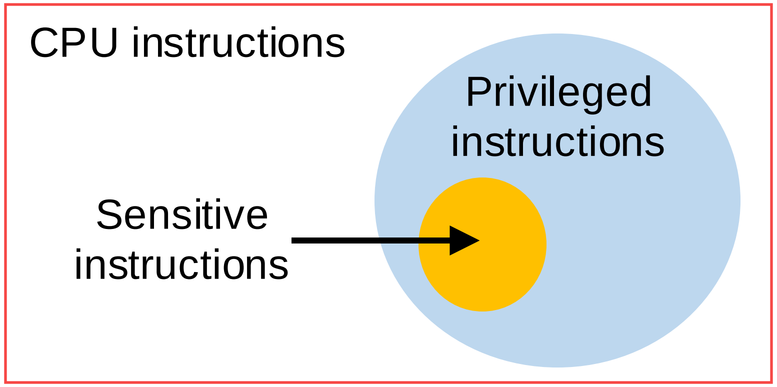
- Sensitive instruction = changes hardware state
- Privileged instruction = runs only in privileged mode
  - Traps to Ring 0 if executed from unprivileged rings
- To build a VMM efficiently via trap-and-emulate method, sensitive instructions should be a subset of privileged instructions
  - x86 does not satisfy this criteria, so trap and emulate VMM is not possible with x86

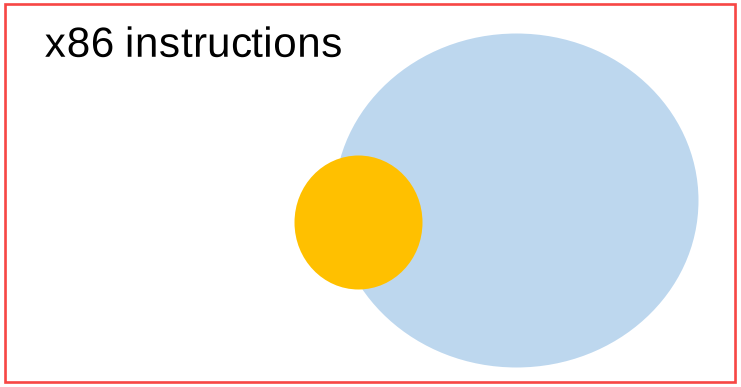


Popek Goldberg theorem

https://www.scs.stanford.edu/21wi-cs140/sched/readings/virtualization.pdf

- Sensitive instruction = changes hardware state
- Privileged instruction = runs only in privileged mode
  - Traps to Ring 0 if executed from unprivileged rings
- To build a VMM efficiently via trap-and-emulate method, sensitive instructions should be a subset of privileged instructions
  - x86 does not satisfy this criteria, so trap and emulate VMM is not possible with x86





Techniques to virtualize x86

- Paravirtualization: rewrite guest OS code to be virtualizable
  - Guest OS won't invoke privileged instructions, but makes "hypercall" to VMM
  - Needs OS source code changes, cannot work with unmodified OS
  - Example: Xen hypervisor

https://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf

Techniques to virtualize x86

- Paravirtualization: rewrite guest OS code to be virtualizable
  - Guest OS won't invoke privileged instructions, but makes "hypercall" to VMM
  - Needs OS source code changes, cannot work with unmodified OS
  - Example: Xen hypervisor

https://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf

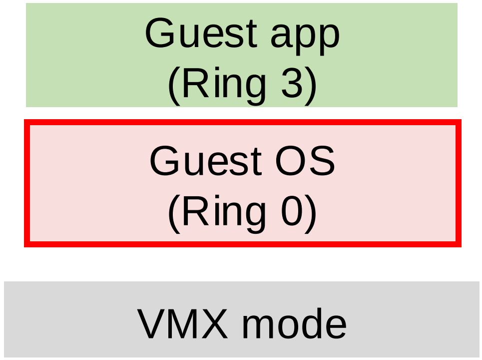
- Full virtualization: CPU instructions of guest OS are translated to be virtualizable
  - Sensitive instructions translated to trap to VMM
  - Dynamic (on the fly) binary translation, so works with unmodified OS
  - Higher overhead than paravirtualization
  - Example: VMWare workstation

Techniques to virtualize x86 (cont.)

- Hardware-assisted virtualization: KVM/QEMU in Linux
  - CPU has a special VMX mode of execution
  - x86 has 4 rings on non-VMX root mode, another 4 rings in VMX mode

Techniques to virtualize x86 (cont.)

- Hardware-assisted virtualization: KVM/QEMU in Linux
  - CPU has a special VMX mode of execution
  - x86 has 4 rings on non-VMX root mode, another 4 rings in VMX mode

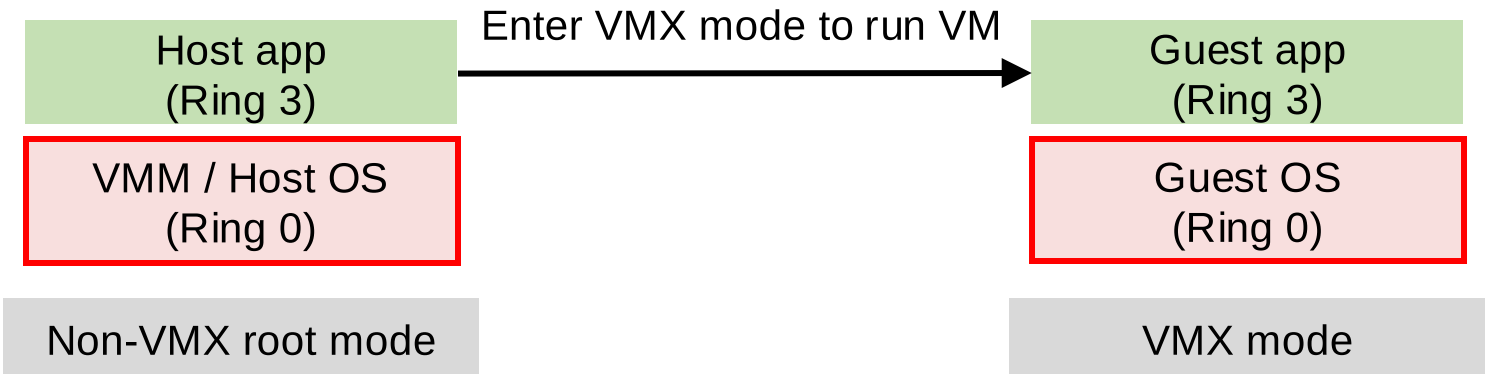


Techniques to virtualize x86 (cont.)

- Hardware-assisted virtualization: KVM/QEMU in Linux
  - CPU has a special VMX mode of execution
  - x86 has 4 rings on non-VMX root mode, another 4 rings in VMX mode

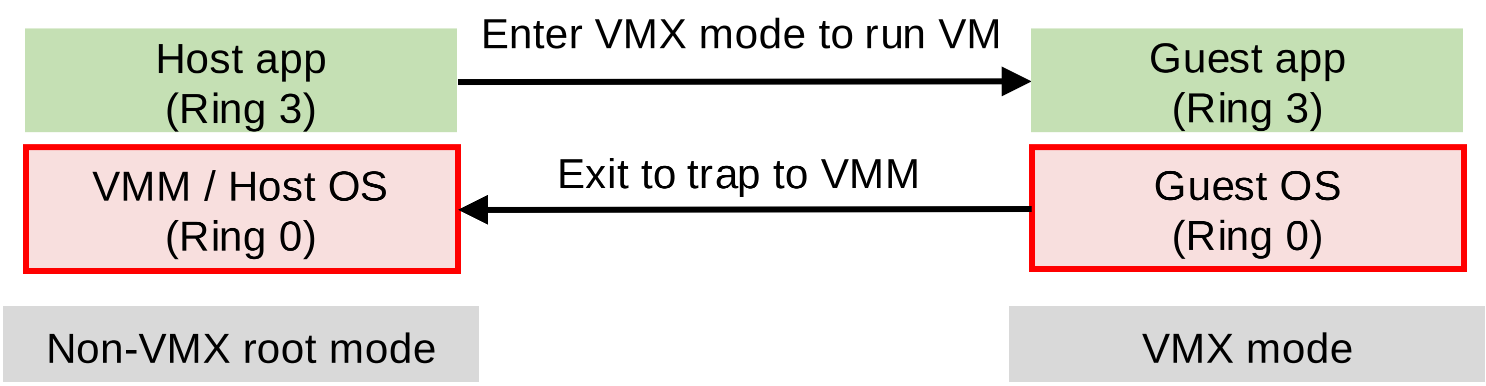
Techniques to virtualize x86 (cont.)

- Hardware-assisted virtualization: KVM/QEMU in Linux
  - CPU has a special VMX mode of execution
  - x86 has 4 rings on non-VMX root mode, another 4 rings in VMX mode
- VMM enters VMX mode to run guest OS in (special) Ring 0



Techniques to virtualize x86 (cont.)

- Hardware-assisted virtualization: KVM/QEMU in Linux
  - CPU has a special VMX mode of execution
  - x86 has 4 rings on non-VMX root mode, another 4 rings in VMX mode
- VMM enters VMX mode to run guest OS in (special) Ring 0
- Exit back to VMM on triggers (VMM retains control)

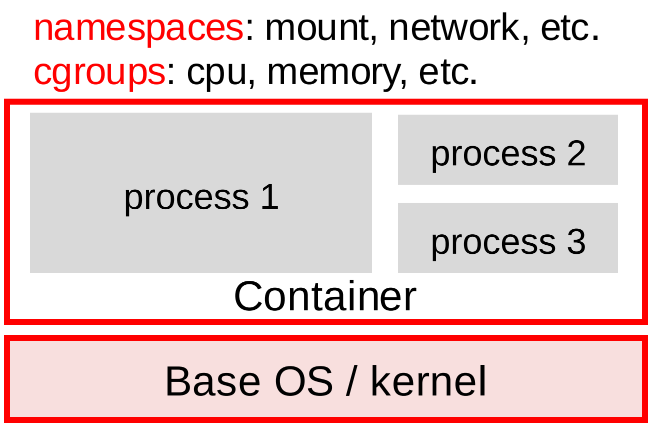


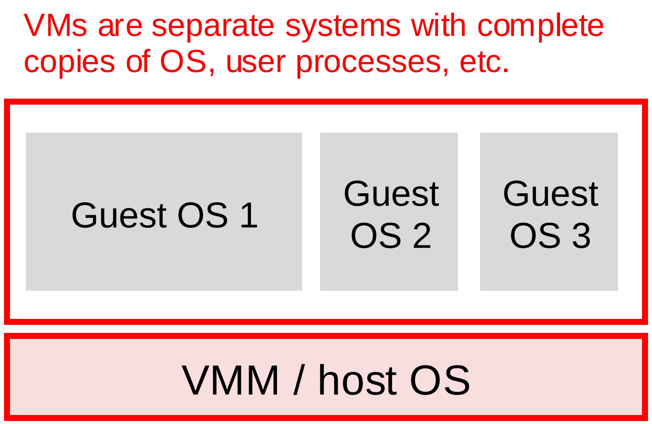
VM demo

Containers

Containers: Lightweight virtualization

- Containers share base OS, have different set of libraries, utilities, root filesystem, view of process tree, networking, etc.
  - VMs have different copies of OS itself
  - Containers have less overhead than VMs, but weaker isolation





Cgroups and namespaces

- cgroup types (resource / performance isolation)
  - cpu, memory, cpuacct, cpuset, freezer, net_cls, blkio, perf_event, net_prio, hugetlb, pids, rdma
- namespace types (namespace isolation)
  - network, mount, time, user, cgroup, IPC, PID, UTS
- Both cgroups and namespaces apply to sets of processes. Configuring all this by hand is VERY complicated.
- "Container framework": Does cgroups and namespaces configuration automatically under the hood
- One reason Docker is popular: "`docker run …`" starts a process using all these features, each with reasonable configurations.

Cgroups

- Assign resource limits on a set of processes
  - Divide processes into groups and subgroups hierarchically
  - Assign resource limits for processes in each group/subgroup
- What resources?
  - CPU, memory, I/O, CPU sets (which process can run on which CPU core), and I/O
  - Allows a user to specify what fraction of a resource can be used by each group of processes

Namespaces

- Group of processes that have an isolated/sliced view of a global resource
- Default namespace for all processes in Linux, system calls to create new namespaces and place processes in them
- What resources can be isolated / sliced?
  - mount: isolates the file system mount points seen by a group of processes
  - PID: isolates the PID number space seen by processes
  - network: isolates network resources like IP addresses, routing tables, port numbers, etc.
  - …

Docker demo