Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 25: Virtual Machines

Prof. Ryan Huang

Administration

Final exam on April 28th 7-9 pm

- Mark on your calendar and don't get the time wrong

In-person, on paper like midterm

- Logistics details will be posted next week

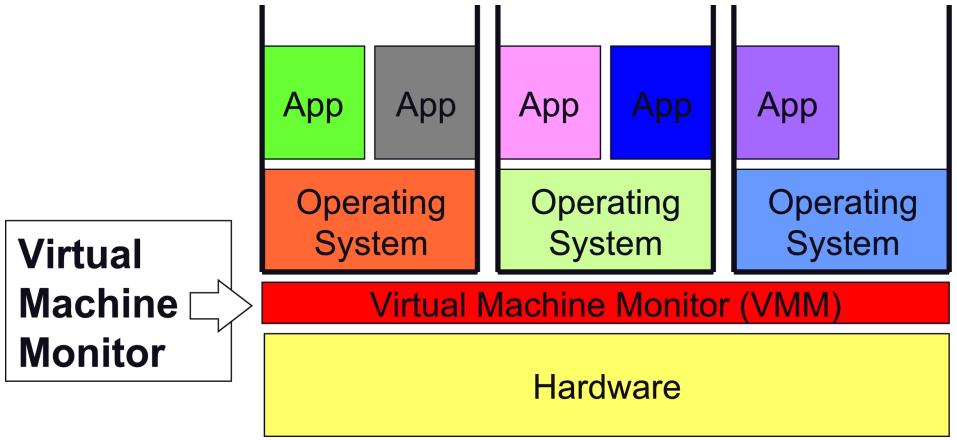
Practice exams released

- Review session this Friday

Virtual Machine Monitor

Thin layer of software that virtualizes the hardware

- Exports a virtual machine abstraction that looks like the h/w
- Provides illusion to the OS (it has full control over the h/w)
  - Run multiple instances of an OS or different OSes simultaneously on the same physical machine



Why Use Virtual Machines?

Software compatibility

- VMMs can run pretty much all software

Resource utilization

- Machines today are powerful, want to multiplex their hardware

Isolation

- Seemingly total data isolation between virtual machines
- Leverage hardware memory protection mechanisms

Encapsulation

- Virtual machines are not tied to physical machines
- Checkpoint/migration

Many other cool applications

- Debugging, emulation, security, speculation, fault tolerance…

What Needs to Be Virtualized?

Exactly what you would expect

- CPU
- Events (exceptions and interrupts)
- Memory
- I/O devices

Isn't this just duplicating OS functionality in a VMM?

- Yes and no
- Approaches will be similar to what we do with OSes
  - Simpler in functionality, though (VMM much smaller than OS)
- But implements a different abstraction
  - Hardware interface vs. OS interface

Approach 1: Machine Simulation

Simplest VMM approach, used by `bochs`

Run the VMM as a regular user application atop a host OS

Application simulates all the hardware (i.e., a simulator)

- CPU – A loop that fetches an instruction, decodes it, simulates its effect

```
while (1) {
  curr_instr = fetch(virtHw.PC);
  virtHw.PC += 4;
  switch (curr_instr) {
    case ADD:
      int sum = virtHw.regs[curr_instr.reg0] +
        virtHw.regs[curr_instr.reg1];
      virtHw.regs[curr_instr.reg0] = sum;
      break;
    case SUB:
      //...
```

- Memory – Just an array, simulate the MMU on all memory accesses
- I/O – Simulate I/O devices, programmed I/O, DMA, interrupts

Approach 1: Machine Simulation

Problem: Too slow!

- CPU/Memory – 100x CPU/MMU simulation
- I/O Device – < 2x slowdown.
- 100x slowdown makes it not too useful

Need faster ways of emulating CPU/MMU

Approach 2: Direct Execution w/ Trap & Emulate

Observations: most instructions are the same regardless of processor privileged level

- Example: `incl %eax`

Why not just give instructions to CPU to execute?

- One issue: Safety – How to get the CPU back? Or stop it from stepping on us? How about `cli/halt`?
- Solution: Use protection mechanisms already in CPU

Run VM's OS directly on CPU in unprivileged user mode

- "Trap and emulate" approach
- Most instructions just work
- Privileged instructions trap into VMM
- Assumptions: all sensitive instructions are privileged
  - Not always true! E.g., push %cs, pushf/popf

Virtualizing Memory

OSes assume they have full control over memory

- Managing it: OS assumes it owns it all
- Mapping it: OS assumes it can map any virtual page to any physical page

But VMM partitions memory among VMs

- VMM needs to assign hardware pages to VMs
- VMM needs to control mappings for isolation
  - Cannot allow an OS to map a virtual page to any hardware page
  - OS can only map to a hardware page given to it by the VMM

Hardware-managed TLBs make this difficult

- When the TLB misses, the hardware automatically walks the page tables in memory
- As a result, VMM needs to control access by OS to page tables

One Solution: Direct Mapping

VMM uses the page tables that a guest OS creates

- These page tables are used directly by hardware MMU

Page tables work the same as before, but OS is constrained to only map to the physical pages it owns

VMM validates all updates to page tables by guest OS

- OS can read page tables without modification
- But VMM needs to check all PTE writes to ensure that the virtual-to-physical mapping is valid
  - That the OS "owns" the physical page being used in the PTE
- Modify OS to hypervisor call into VMM when updating PTEs

Works fine if you can modify the OS

If you can't…

Second Approach: Level of Indirection

Three abstractions of memory

- Machine: actual hardware memory (e.g., 16 GB of DRAM)
- Physical: abstraction of hardware memory managed by OS
  - If a VMM allocates 512 MB to a VM, the OS thinks the computer has 512 MB of contiguous physical memory
    - Underlying machine memory may be discontiguous
- Virtual: virtual address spaces you know and love

Translation: VM's Guest VA → VM's Guest PA → Host PA

In each VM, OS creates and manages page tables for its virtual address spaces *without modification*

- But these page tables are not used by the MMU hardware

Shadow Page Tables

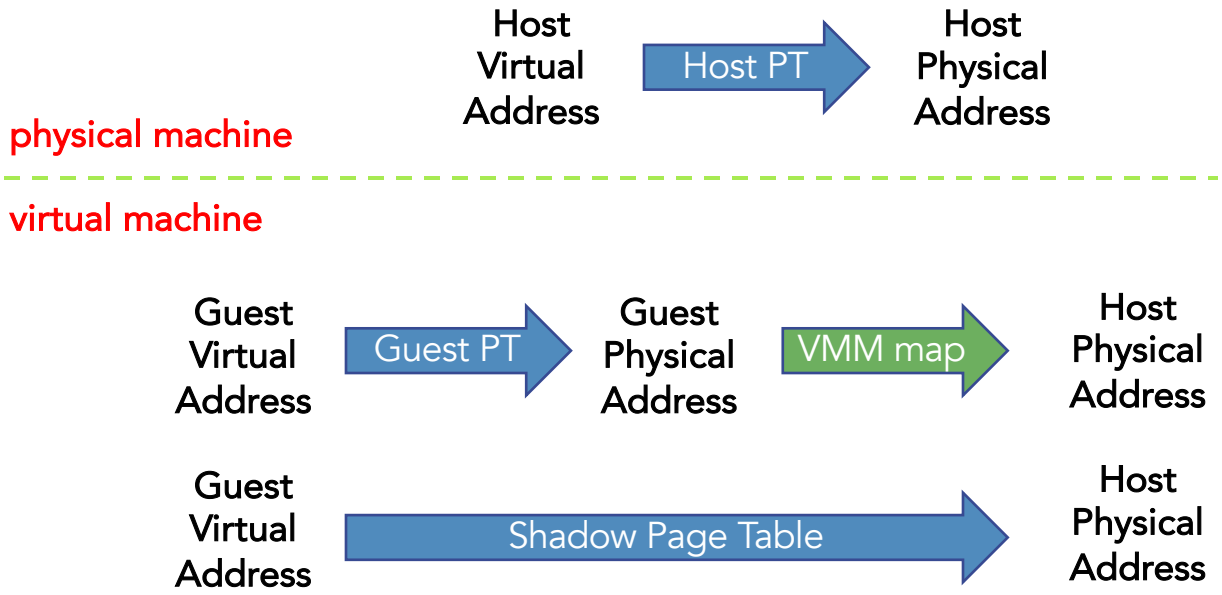
VMM creates and manages another page table that maps virtual pages directly to machine pages

- These tables are loaded into the MMU
- VMM page tables are the shadow page tables

VMM needs to keep its V→ M tables consistent with changes made by OS to its V→ P tables

- VMM maps OS page tables as read-only
- When OS writes to page tables, trap to VMM
- VMM applies write to shadow table and OS table, returns
- Also known as memory tracing

Memory Mapping Summary



More on Shadow Page Table

Shadow page tables are essentially a cache

VMM is responsible for maintaining the consistency

Two kinds of page faults

- *True page faults* when page not in VM's guest page table
- *Hidden page faults* when just misses in shadow page table

On a page fault, VMM must:

- Lookup guest VPN → guest PPN in guest's page table
- Determine where guest PPN is in host physical memory
- Add guest VPN → host PPN mapping in shadow page table

Lesson

Never underestimate the power of indirection!