

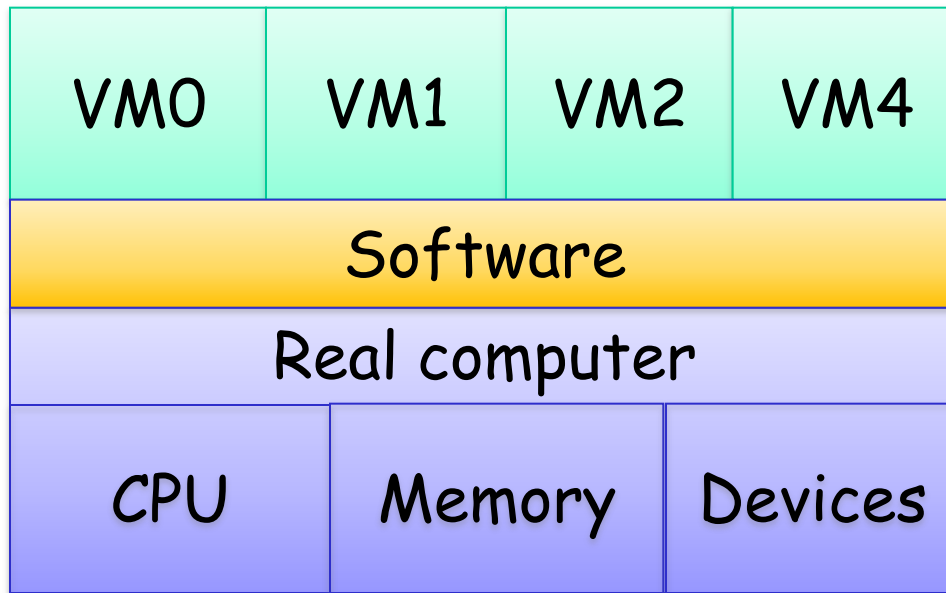
W4118: virtual machines



Instructor: Junfeng Yang

References: Modern Operating Systems (3rd edition), Operating Systems Concepts (8th edition), previous W4118, and OS at MIT, Stanford, and UWisc

Virtual machines (VM)



Why virtual machines?

- ❑ Manage big machines
 - Multiplex CPUs/memory/devices at VM granularity
 - E.g., Amazon EC2
- ❑ Multiple OS on one machine
 - E.g., use Windows on Linux OS
- ❑ Isolate faults/break-ins
 - One VM is compromised/crashes, others OK
- ❑ Kernel development
 - Like QEMU, but faster
- ❑ OS granularity checkpoint/record/replay

Usual VM goals

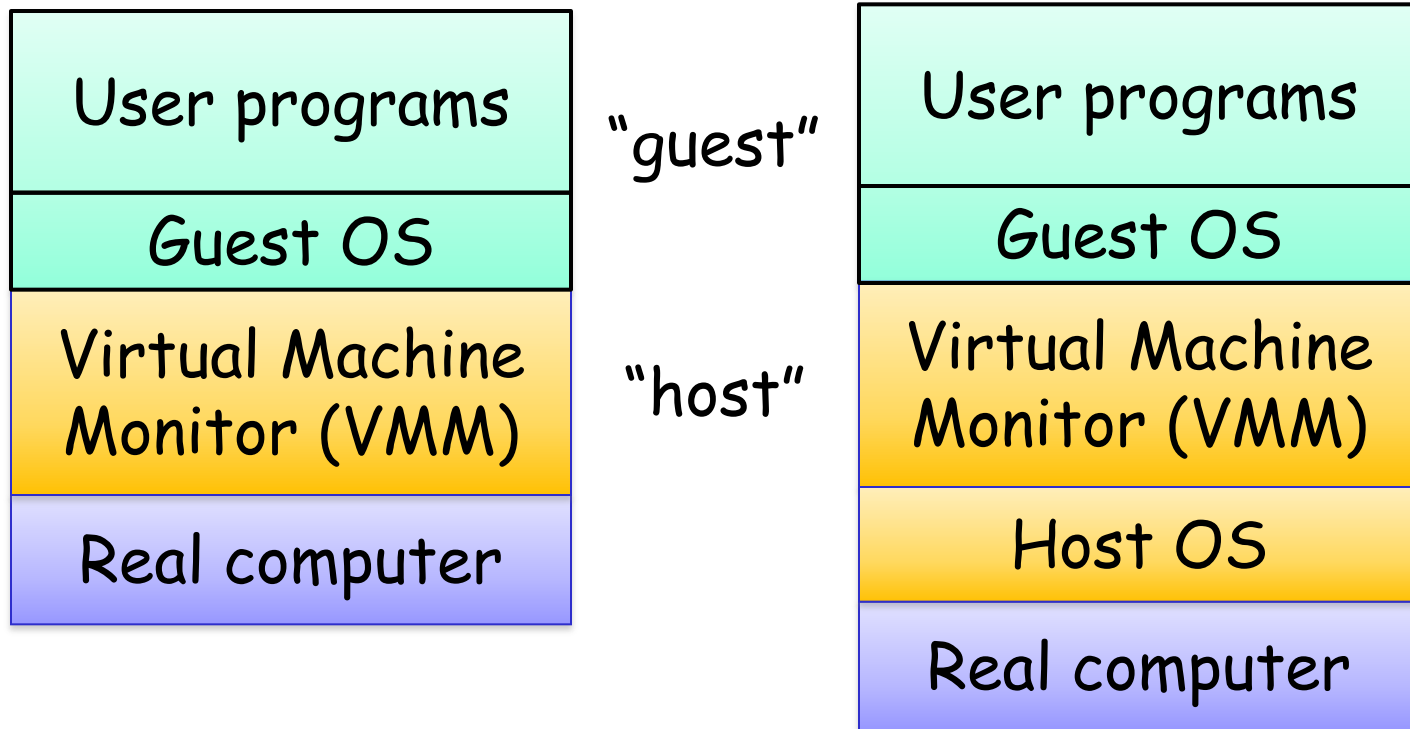
- ❑ Accurate
 - Guest can't distinguish VM from real computer
- ❑ Isolated
 - Guest can't escape VM
- ❑ Fast
- ❑ Some VM implementations require guest kernel modifications
 - E.g., Xen

Virtual machine lineage

- ❑ 1960s: IBM used VMs to share mainframe
 - VM/370, today's z/VM
 - Still in use!

- ❑ 1990s: VMWare re-popularized VMs for x86
 - VMWare ESX servers
 - VMWare work station
 - ...

Virtual machine structures



VMM responsibilities

- ❑ Time-share CPU among guests
- ❑ Space-share memory among guests
- ❑ Simulate disk, network, and other devices
 - Often multiplex on host devices

Naïve approach: simulation

```
int32_t regs[8];
#define REG_EAX 1;
#define REG_EBX 2;
#define REG_ECX 3;
...
int32_t eip;
int16_t segregs[4];
...

for (;;) {
    read_instruction();
    switch (decode_instruction_opcode()) {
    case OPCODE_ADD:
        int src = decode_src_reg();
        int dst = decode_dst_reg();
        regs[dst] = regs[dst] + regs[src];
        break;
    case OPCODE_SUB:
        int src = decode_src_reg();
        int dst = decode_dst_reg();
        regs[dst] = regs[dst] - regs[src];
        break;
    ...
    }
    eip += instruction_length;
}
```

- ❑ Interpret each guest instruction
- ❑ Maintain each VM state purely in software
- ❑ Problem: **too slow!**

2nd approach: trap-and-emulate

- ❑ Execute guest instructions on real CPU when possible
 - E.g., `addl %eax, %ebx`
- ❑ Run guest OS in unprivileged mode
- ❑ Privileged instructions trap, and VMM emulates
 - E.g., `movl %eax, %cr3`
- ❑ VMM hides real machine state from guests
 - E.g., virtual `%cr3` set by guest, real `%cr3` set by VMM,
 - More: page table, privilege level, interrupt flag, ...

Trap-and-emulate: tricky on x86

- ❑ Not all instructions that should be emulated cause traps
- ❑ Instructions have **different effects** depending on privilege mode
- ❑ Instructions **reading** privileged state don't trap
- ❑ **Page table modifications** don't trap
- ❑ Trap them all → **slow**

Real x86 state to hide&protect

- ❑ **CPL** (low bits of CS) = 3, but guest expects 0
- ❑ **Physical memory**: guest expects 0..PHYSTOP, VMM maps to one slice of physical memory
- ❑ **Page tables**: don't map to physical addresses expected by guest OS
 - **Shadow page table**
- ❑ **%cr3**: points to shadow page table

Real x86 state to hide&protect (cont.)

- ❑ **GDT**: guest OS descriptors have DPL = 3, but guest expects 0
- ❑ **GDTR**: points to shadow GDT table
- ❑ **IDT descriptors**: traps go to VMM, not guest
- ❑ **IDTR**: points to shadow IDT table
- ❑ **IF in EFLAGS**: guest expects 0 after **cli**
- ❑ ...

Virtualize physical memory

□ Guest wants

- Physical address starts at $PA = 0$
- Use "all" physical memory

□ VMM must

- Space-share all physical memory among guests
- Protect one guest's memory from another

□ Idea:

- Claim DRAM smaller than real DRAM
- Ensuring paging is enabled
- Rewrite guest's PTEs to map to real PA?
- Copy guest's PTEs to shadow page table and map copied PTEs to real PA

Example: VMM allocates a guest 0x1000000-0x2000000

1. VMM allocates a guest physical memory 0x1000000 to 0x2000000
2. VMM gets trap if guest changes %cr3 (since guest kernel at CPL=3)
3. VMM copies guest's pagetable to "shadow" pagetable
4. VMM adds 0x1000000 to each PA in shadow table
5. VMM checks that each PA is < 0x2000000

Handling page table modifications

- ❑ VMM must make shadow page table entries (PTEs) consistent with guest PTEs
- ❑ *PTE loading*: copy guest PTEs to shadow PTEs on context switch
- ❑ *PTE tracing*: when guest modifies guest PTEs, modify shadow PTEs as well

PTE loading

- ❑ Naive approach: on guest %cr3 write, copy all guest PTEs
 - Problem: **slow** context switch
- ❑ Another approach: start with minimum mappings (just the PTEs of VMM), and copy on demand on “**hidden**” page faults
 - Problem: **too many** page faults
- ❑ Approach used in VMware: reuse populated shadow PTEs

PTE tracing

- ❑ Approach I: mark the memory region holding guest PTEs as readonly, and copy updates to shadow PTEs on page faults
 - Problem: too many page faults
- ❑ Approach II: binary translate code that writes to shadow PTEs to call out to VMM
 - Faster than traps

Do all instructions that read/write sensitive state cause traps at CPL = 3?

- ❑ `pushw %cs`: reveals CPL = 3, not 0
- ❑ `sgdt`: reveals real GDTR
- ❑ `sidt`: reveals real IDTR
- ❑ `pushfl`: reveals IF flag
- ❑ `popfl`: if CPL = 3, do not set IF flag (no trap)
- ❑ `iret`: no privilege mode change so won't restore SS/ESP

3rd approach: binary translation

□ Simplified idea

- Replace non-trapping instructions that read/write sensitive state with trap instruction
 - `int3`: triggers a break point exception. Shortest instruction (1 byte), doesn't change code size/layout
- Keep track of original instruction
- VMM emulate original instruction in trap

□ Problems: how does the rewriter find all code?

- Or where the instruction boundaries are,
- Or whether bytes are code or data ...

Dynamic binary translation

- ❑ Idea: disassemble code only as executed, since jump instructions reveal where code is
- ❑ When VMM first loads guest kernel, translate from entry (fixed) up to first jump
 - Replace bad instructions with equivalent instructions on virtual states
 - Replace "jmp X" with "movl X, %eax; jmp translator;"
- ❑ In translator, look where the jump goes
 - Repeat above steps
- ❑ Keep track of what we've translated to avoid re-translate
 - Store translated code in code cache (original → translated mapping)

Binary translation example

Entry:

 pushl %ebp

 popfl

 jnz x

x:

 ...

 jmp y

Entry':

 pushl %ebp

 vm->IF = ...

 popfl

 movl x, %eax

 jnz translator

x':

 ...

 movl y, %eax

 jmp translator

4th approach: hardware support

- ❑ Simplified implementation of VMM
- ❑ Hardware maintains per-guest virtual state
 - CPL, EFLAGS, idtr, etc
- ❑ Hardware knows it is in "guest mode"
 - Instructions directly modify virtual state
 - Avoids many traps to VMM

Hardware support details

- ❑ Hardware basically adds a new privilege level
 - VMM mode, $CPL=0$, $CPL=3$
 - Guest-mode, $CPL=0$ is not fully privileged
- ❑ No traps on system calls; hardware handles CPL transition
- ❑ Hardware supports **two** page tables: guest page table and VMM's page table
 - Virtual address \rightarrow guest physical address
 - Guest physical address \rightarrow host physical address