Virtual Machines
Part 1: 61 years ago

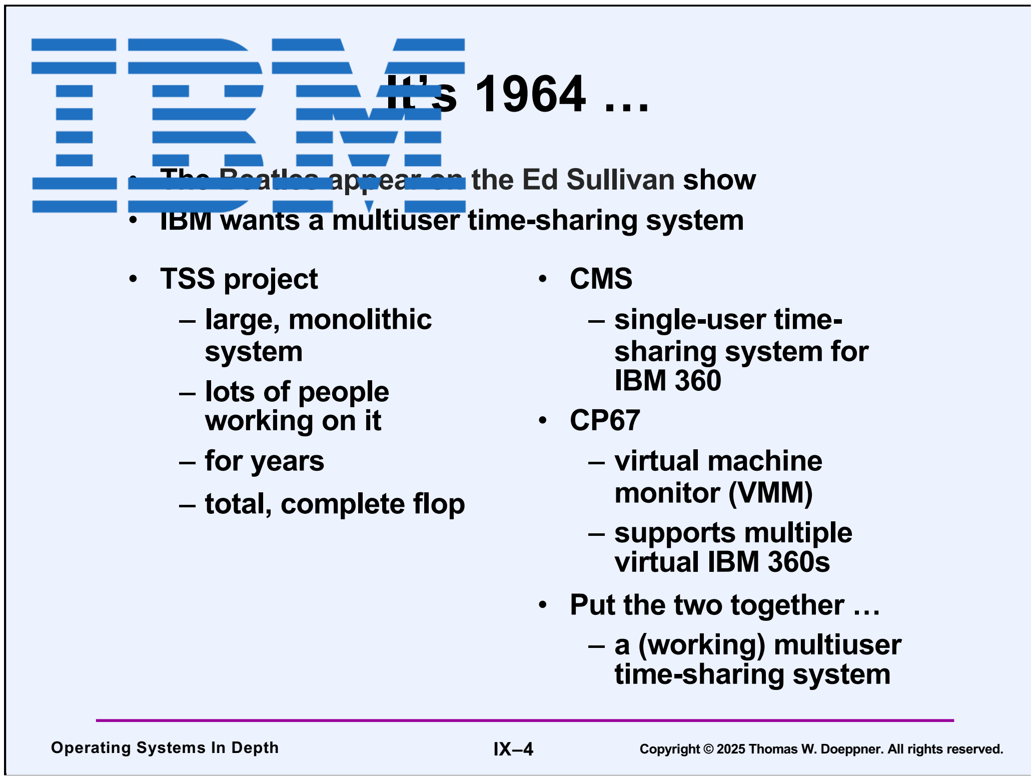
It’s 1964 …

- The Beatles appear on the Ed Sullivan show

It’s 1964 …

- The Beatles appear on the Ed Sullivan show

The video is from https://www.edsullivan.com/artists/the-beatles/.



It's 1964 …

- The Beatles appear on the Ed Sullivan show
- IBM wants a multiuser time-sharing system

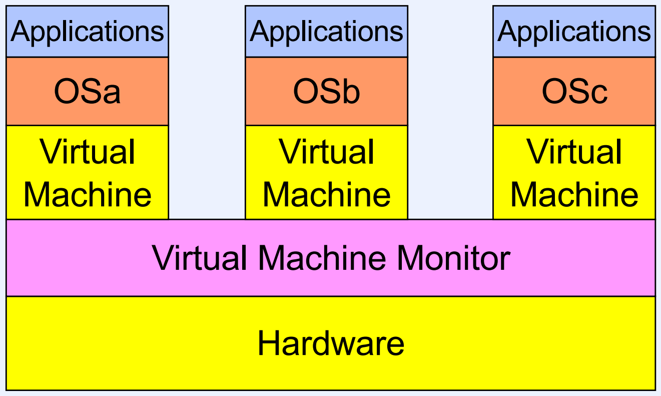
- TSS project
  - large, monolithic system
  - lots of people working on it
  - for years
  - total, complete flop
- CMS
  - single-user time-sharing system for IBM 360
- CP67
  - virtual machine monitor (VMM)
  - supports multiple virtual IBM 360s
- Put the two together …
  - a (working) multiuser time-sharing system

The Beatles made their first live appearance on American TV on the Ed Sullivan Show on February 9, 1964. If you haven't heard of either the Beatles or the Ed Sullivan show, you're totally out of it, culturally. Your Professor was 12 years old at the time, was totally out of it, culturally, and didn't watch it (having never heard of the Beatles). He sort of found out about it the next day at school.

He also had barely heard of IBM (they had something to do with typewriters and computers) and didn't hear about TSS, CMS, and CP67 until many years after he first listened to the Beatles.

TSS stands for, oddly, time-sharing system. (This is the same company that developed a programming language called PL/1 (programming language one). There never was a PL/2.)

Virtual Machines



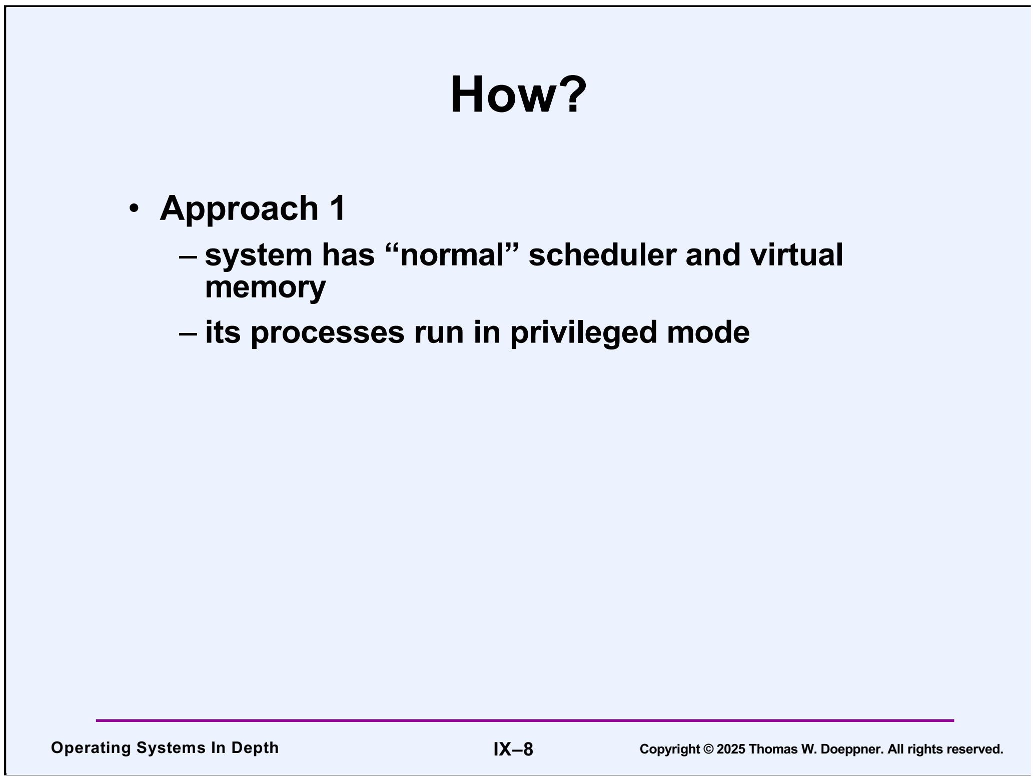
Virtual machines are an interesting extension of the virtual-memory concept: not only do they give processes the illusion that they have all of memory to themselves, but also, they give them the illusion that they have an entire machine to themselves. The kernel, known as a **virtual machine monitor**, is a relatively simple piece of software that provides environments, called **virtual machines**, that look and behave exactly like real machines. Within a virtual machine, since it behaves just like a real machine, one can run a standard operating system, or perhaps a specialized operating system.

Why?

- Structuring technique for a multi-user system
- OS debugging and testing
- Multiple OSes on one machine
- Adapt to hardware changes in software
- Server consolidation and service isolation
- It’s cool

User vs. Privileged Mode

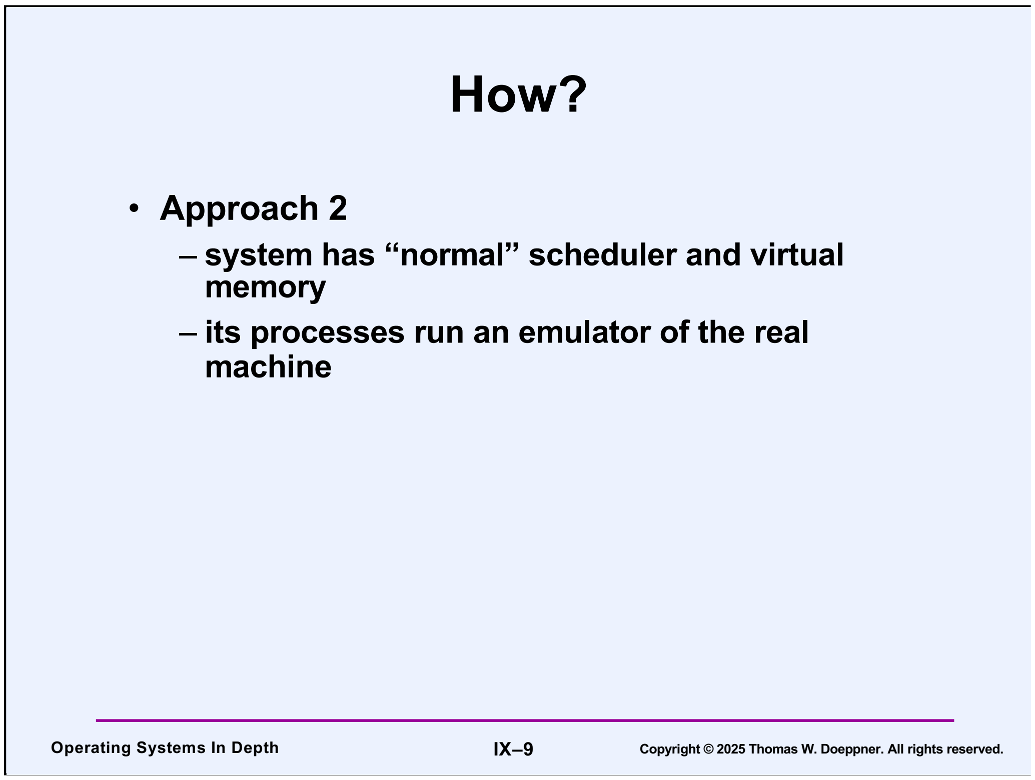
- Privileged mode
  - may run all instructions, access all registers and memory
  - for example:
    - modify address translation for virtual memory
    - access and control I/O devices
    - mask and unmask interrupts
    - start and stop system clock
- User mode
  - may run only “innocuous” instructions
  - may access only normal registers and certain designated memory



How?

- Approach 1
  - system has “normal” scheduler and virtual memory
  - its processes run in privileged mode

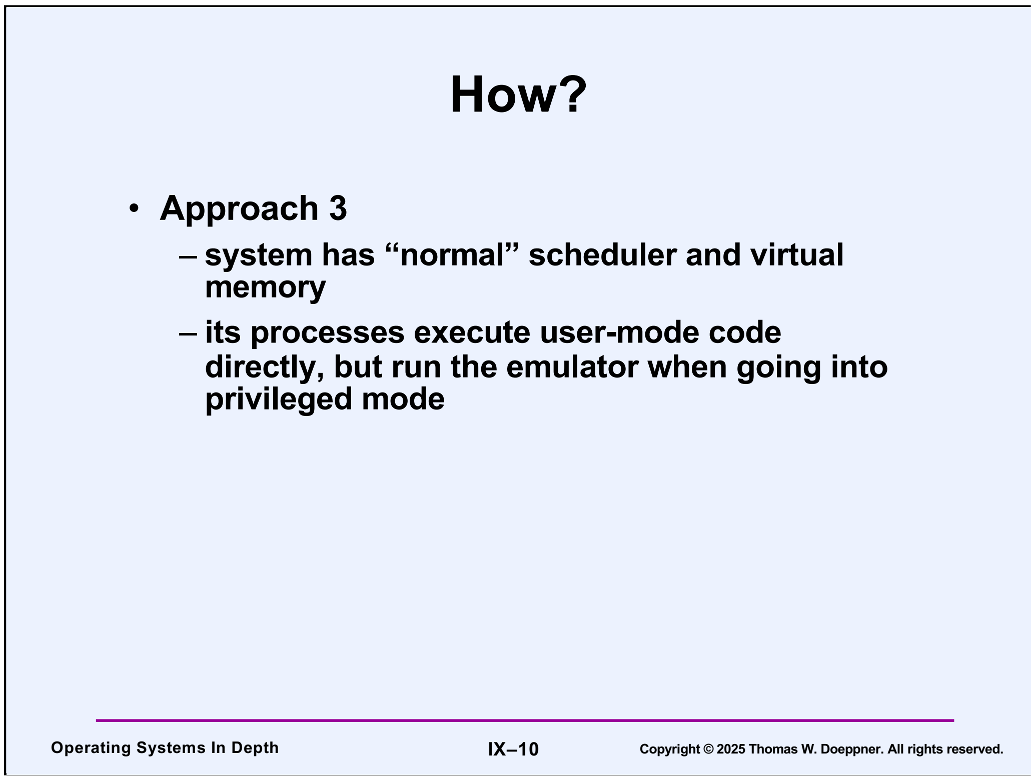
One possible approach for implementing a virtual machine monitor is for it to have processes that run completely in privileged mode. Thus, code running in these processes have full access to all the features of the hardware. But, if the processes run in privileged mode, then they can do everything, including interfering with (and attacking) all other processes and the OS.



How?

- Approach 2
  - system has “normal” scheduler and virtual memory
  - its processes run an emulator of the real machine

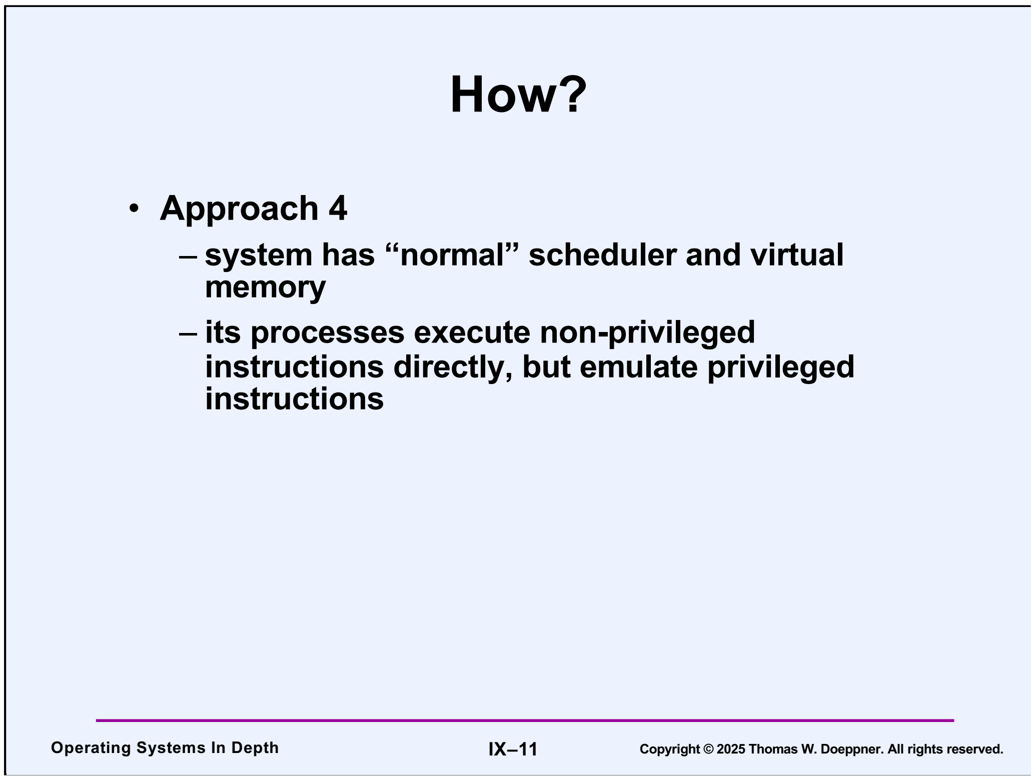
A better approach might be to have an emulator for the real machine. This is code that reads machine code instruction by instruction and executes the instructions as if they were on the standalone computer being emulated. This can be made to work but emulating the real machine would be very slow.



How?

- Approach 3
  - system has “normal” scheduler and virtual memory
  - its processes execute user-mode code directly, but run the emulator when going into privileged mode

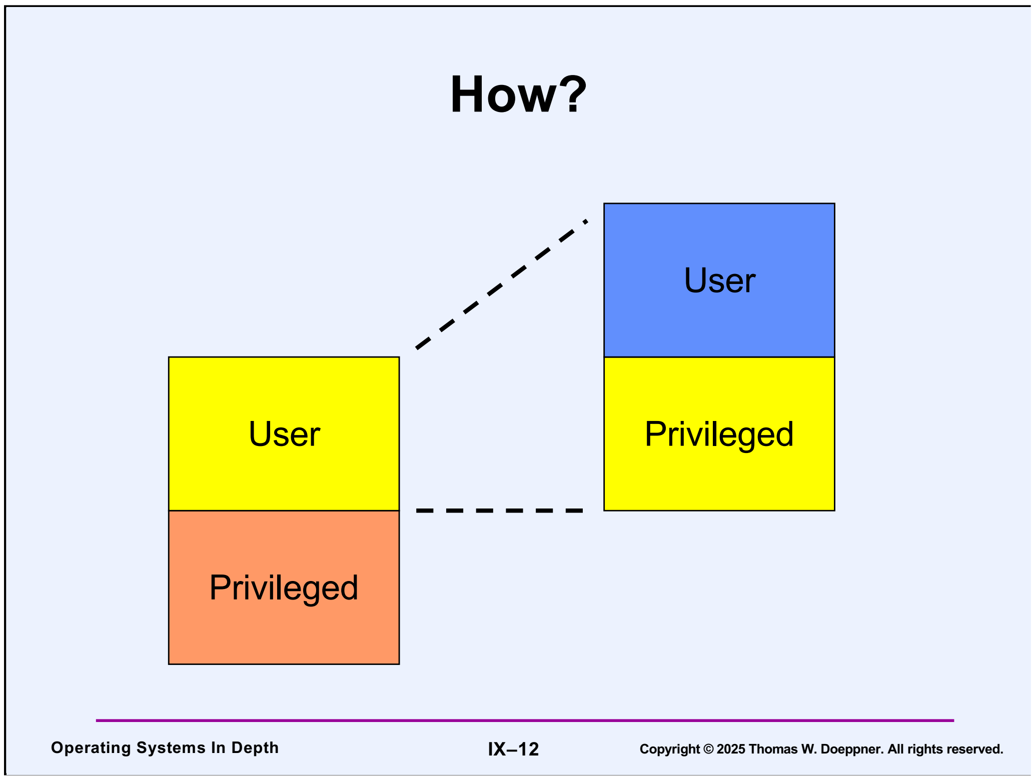
This is better, but it's still slow when in privileged mode.



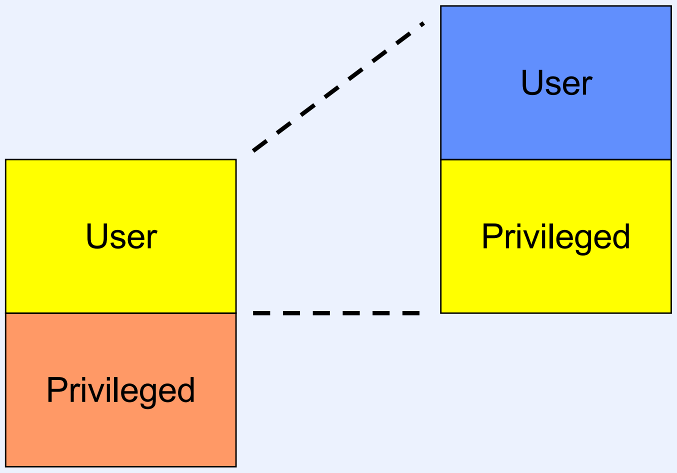
How?

- Approach 4
  - system has “normal” scheduler and virtual memory
  - its processes execute non-privileged instructions directly, but emulate privileged instructions

This actually works well – it's the basis of how virtual machines monitors are actually implemented.

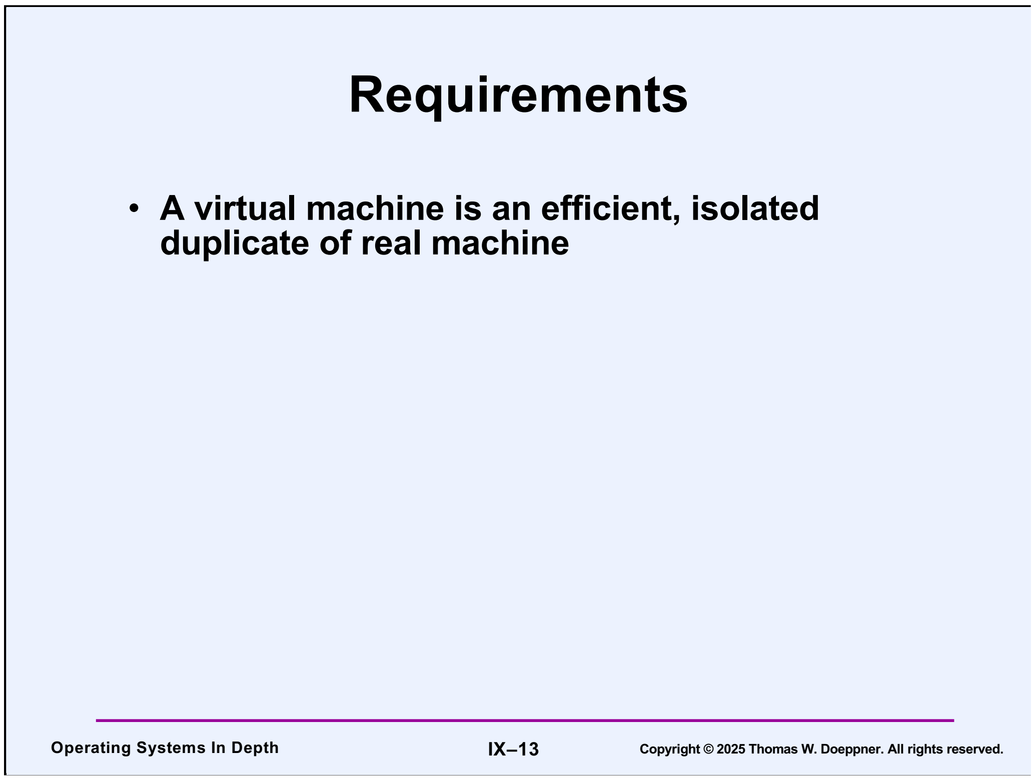


How?



The key to the design of a system supporting virtual machines is the distinction between privileged and user modes. When the processor is running in privileged mode, there is no protection of the various system components. Thus if we are trying to isolate the various virtual machines from one another, we cannot allow the virtual machines to have the processor in privileged mode. But, on the other hand, since we want the virtual machine to behave as a real machine does, we must support the execution of privileged instructions, as well as the dichotomy between privileged and user modes, on each virtual machine.

The solution is to allow only the virtual machine monitor to run in privileged mode; all other software runs in user mode. However, for each virtual machine, we maintain the concepts of "virtual privileged mode" and "virtual user mode." Whenever a virtual machine is in privileged mode, the virtual machine monitor represents the virtual machine's state as being virtual privileged mode, though the (real) processor is in user mode. When a privileged instruction is executed on a virtual machine, the real processor traps to the virtual machine monitor. The virtual machine monitor will then check the state of the virtual machine: if the virtual machine is in virtual user mode, then this attempt to execute a privileged instruction should result in a trap on the virtual machine. However, if the virtual machine is in virtual privileged mode, then the privileged instruction should be allowed to execute. The instruction is not executed directly—the virtual machine monitor must make certain that the effect of executing the instruction applies only to the virtual machine. For example, one virtual machine must not be allowed to start an operation on another virtual machine's virtual disk drive.



Requirements

- A virtual machine is an efficient, isolated duplicate of real machine

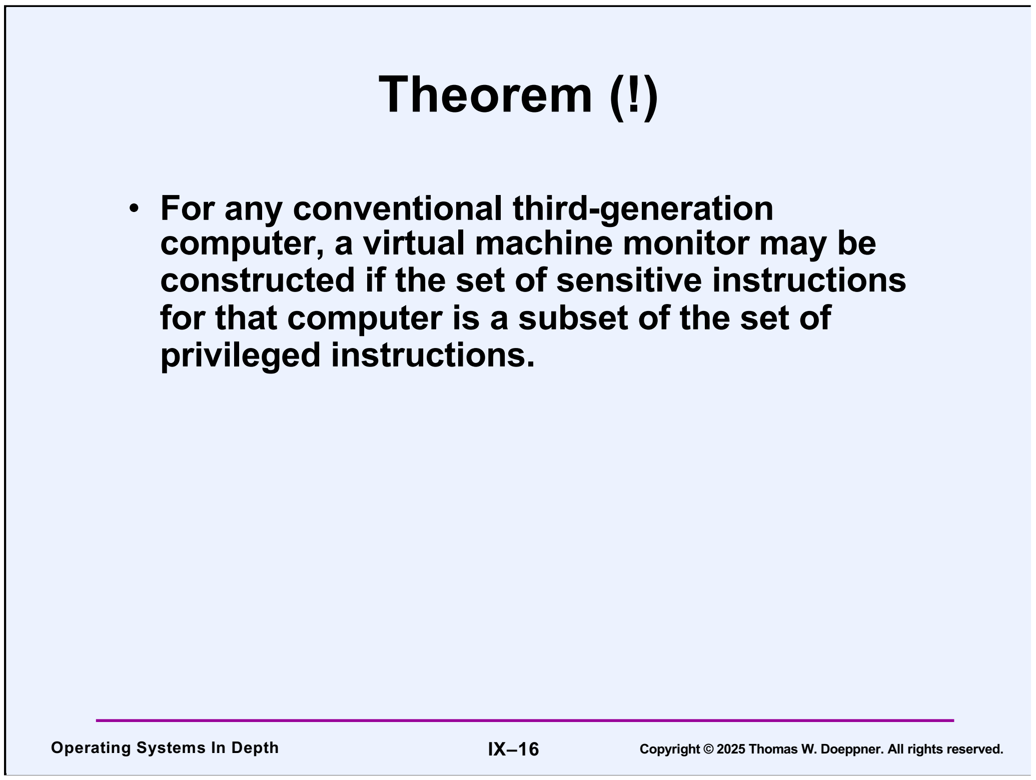
This and the next three slides are based on material from the paper: “Formal Requirements for Virtualizable Third Generation Architectures,” by G. J. Popek and R. P. Goldberg, Communications of the ACM, July 1974, Vol. 17., No. 7.

Sensitive Instructions

- **Control-sensitive instructions**
  - **affect the allocation of resources available to the virtual machine**
  - **change processor mode without causing a trap**
- **Behavior-sensitive instructions**
  - **effect of execution depends upon location in real memory or on processor mode**

Privileged Instructions

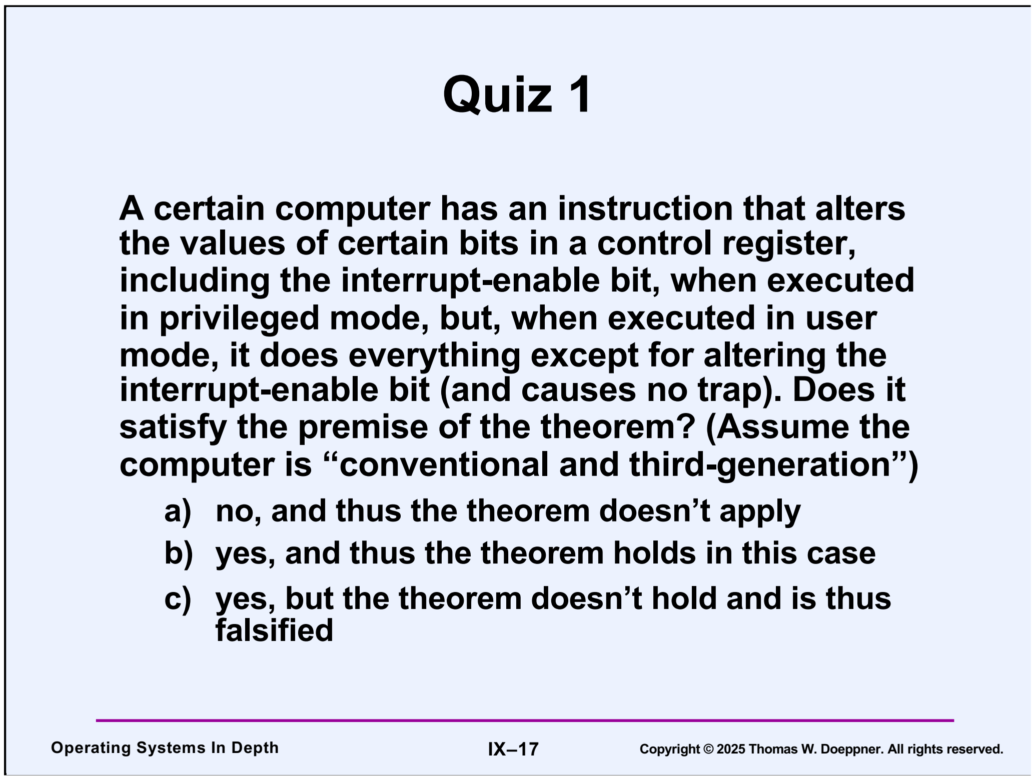
- Cause a fault in user mode
- Work fine in privileged mode



Theorem (!)

- For any conventional third-generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

A "conventional third-generation computer" was the sort that was common in the early 1970s.



Quiz 1

A certain computer has an instruction that alters the values of certain bits in a control register, including the interrupt-enable bit, when executed in privileged mode, but, when executed in user mode, it does everything except for altering the interrupt-enable bit (and causes no trap). Does it satisfy the premise of the theorem? (Assume the computer is “conventional and third-generation”)

a) no, and thus the theorem doesn’t apply

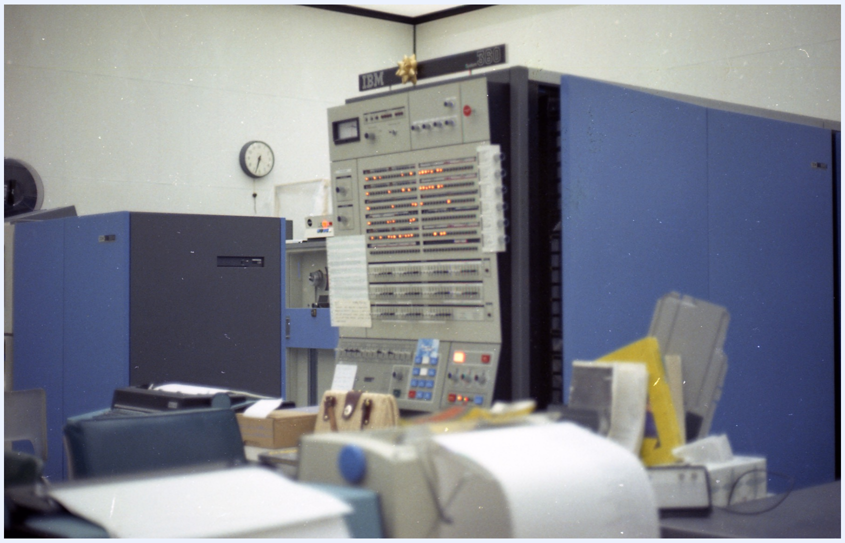
b) yes, and thus the theorem holds in this case

c) yes, but the theorem doesn’t hold and is thus falsified

The computer in question is the Intel X86 (and X86-64). If the interrupt-enable bit is zero, interrupts won't happen (they're deferred). If the bit is one, they will happen. Thus we have a sensitive instruction that, rather being privileged, behaves differently in privileged mode than it does in non-privileged mode.

Note that the theorem says, "if A then B". But if A is false, there are no implications about the truth of B.

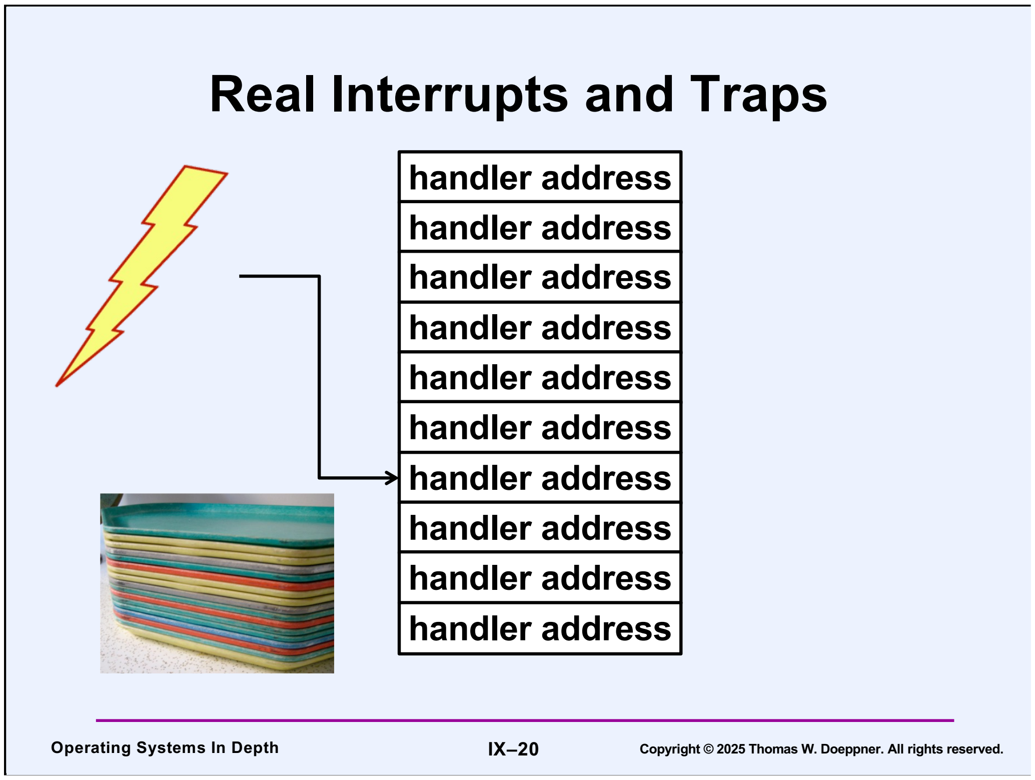
IBM 360/67



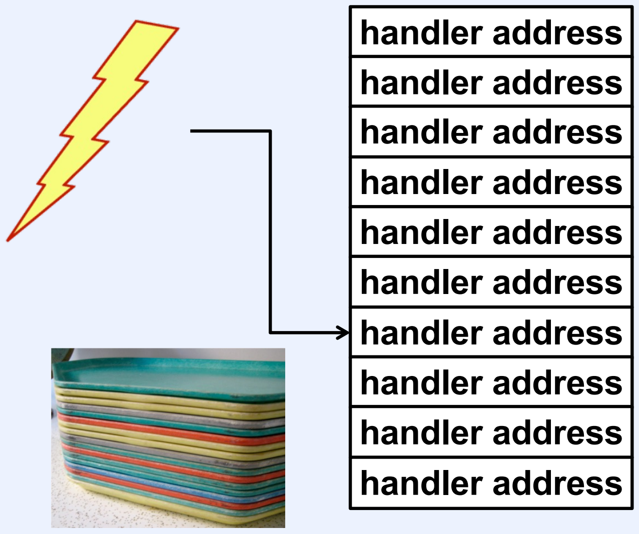
This is a photo of Brown's IBM 360 Model 67, which Brown owned from January 1969 through mid 1978. It ran IBM's CP67 OS, a virtual machine monitor, with many virtual machines running CMS and others running other OS's. (It was housed in 180 George St., which was then the Brown University Computer Center, but which now houses the Brown Center for Computation and Visualization (CCV).)

The (Real) 360 Architecture

- Two execution modes
  - supervisor and problem (user)
  - all sensitive instructions are privileged instructions
- Memory is protectable: 2k-byte granularity
- All interrupt vectors and the clock are in first 512 bytes of memory
- I/O done via channel programs in memory, initiated with privileged instructions
- Dynamic address translation (virtual memory) added for Model 67

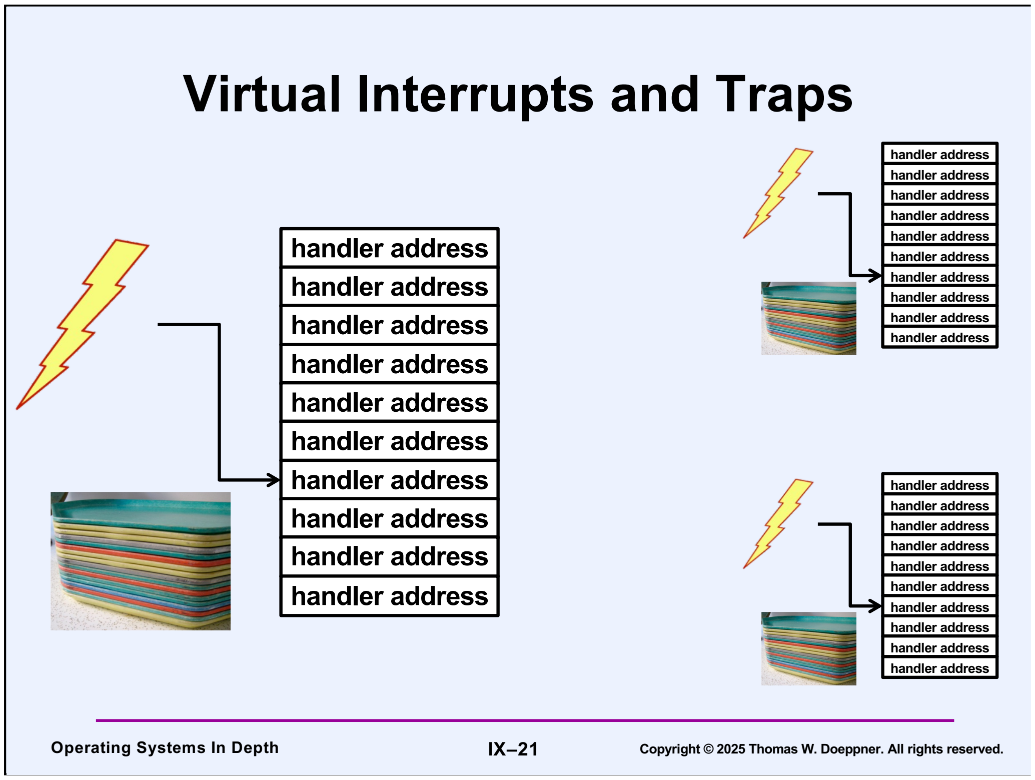


Real Interrupts and Traps



When an interrupt or trap occurs, the current state is pushed on the kernel stack. Control is transferred to the location given in the interrupt vector; the particular element of which is selected by the type of interrupt. When the handler returns, the saved state is popped off the stack and restored, thus returning control to whatever was interrupted.

Note: the IBM 360 architecture did not directly support stacks. They had to be implemented explicitly in software. What's in each element of the interrupt vector is a **program status word** (PSW) that specifies the address of the handler and other information.



Virtual Interrupts and Traps

A real interrupt or trap occurs, of course, on the real machine. The VMM might determine that it should be handled by a virtual machine, as if the interrupt or trap had happened on that machine. The VMM simulates the interrupt or trap on the virtual machine by causing to happen what happens on the real machine: it saves the current state of the virtual machine on the kernel stack, looks up the interrupt or trap in virtual machine's interrupt vector, then passes control to the virtual machine at that location.

Actions on Real 360

	User mode	Privileged mode
non-sensitive instruction	**executes fine**	**executes fine**
errant instruction	**traps to kernel**	**traps to kernel**
sensitive instruction	**traps to kernel**	**executes fine**
access low memory	**traps to kernel**	**executes fine**

Actions on Virtual 360

	User mode	Privileged mode
non-sensitive instruction	executes fine	executes fine
errant instruction	traps to VMM; VMM causes trap to occur on guest OS	traps to VMM; VMM causes trap to occur on guest OS
sensitive instruction	traps to VMM; VMM causes trap to occur on guest OS	traps to VMM; VMM verifies and emulates instruction
access low memory	traps to VMM; VMM causes trap to occur on guest OS	traps to VMM; VMM verifies and emulates/translates access

Quiz 2

Can a VMM (supporting other virtual machines) run on a virtual machine?

a) it requires some changes to a VMM for it to run on a virtual machine

b) yes, no problem

c) no, can’t be done

Virtual Devices?

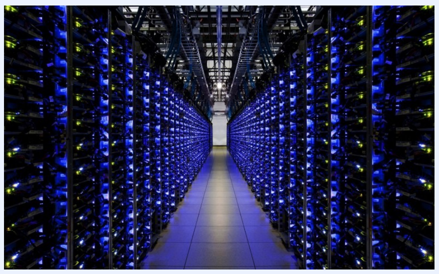
- Terminals
  - connecting (real) people
- Networks
  - didn’t exist in the 60s
  - (how did virtual machines communicate?)
- Disk drives
  - CP67 supported “mini disks”
  - extended at Brown into “segment system”
- Interval timer
  - virtual or real?

Coping

- Invent new devices
  - recognized by VMM as not real, but referring to additional functionality
    - e.g., mini disks
- Provide new VM facilities not present on real machine
  - e.g., Brown segment system
  - special instructions on VM to request service from VMM
    - sort of like system calls (supervisor calls on 360), but ...
      - hypervisor calls
        - 360 had an extra, unused privileged instruction
          - the *diagnose* instruction

Virtual Machines
Part 2: starting ~20 years ago

$\neq$



How They’re Different

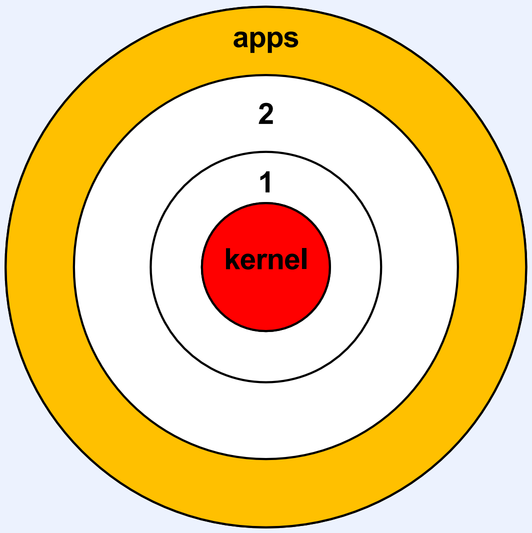
IBM 360

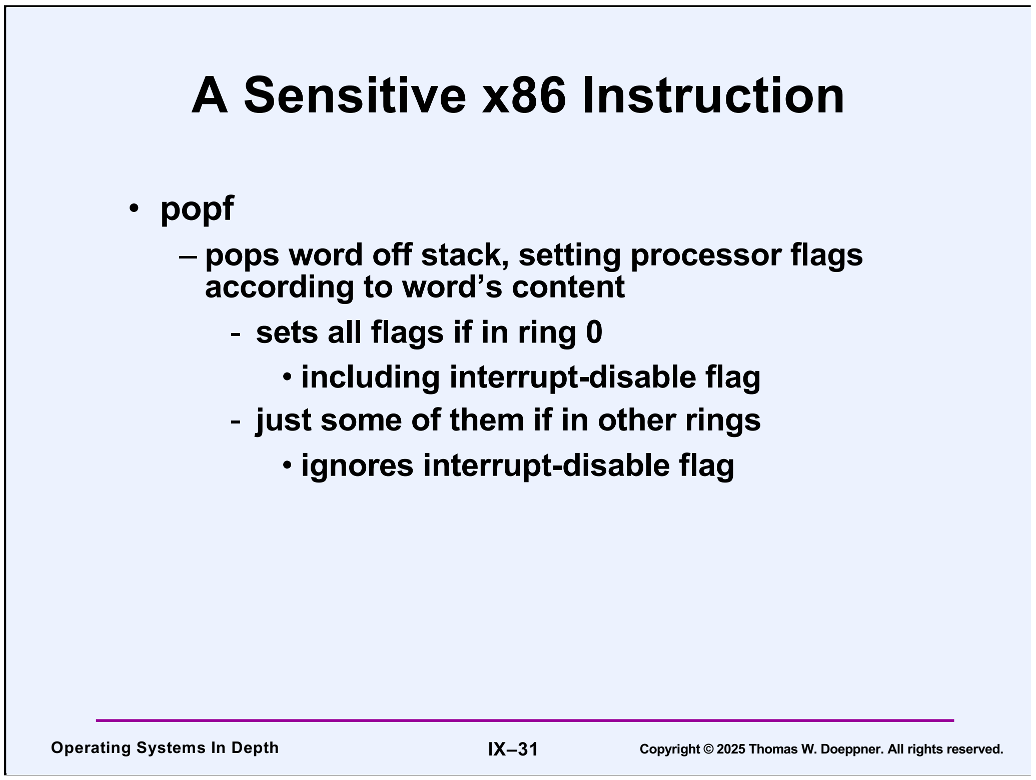
- Two execution modes
  - supervisor and problem (user)
  - all sensitive instructions are privileged instructions
- Memory is protectable: 2k-byte granularity
- All interrupt vectors and the clock are in first 512 bytes of memory
- I/O done via channel programs in memory, initiated with privileged instructions
- Dynamic address translation (virtual memory) added for Model 67

Intel x86

- Four execution modes
  - rings 0 through 3
  - not all sensitive instructions are privileged instructions
- Memory is protectable: segment system + virtual memory
- Special register points to interrupt vector
- I/O done via memory-mapped registers
- Virtual memory is standard

Rings





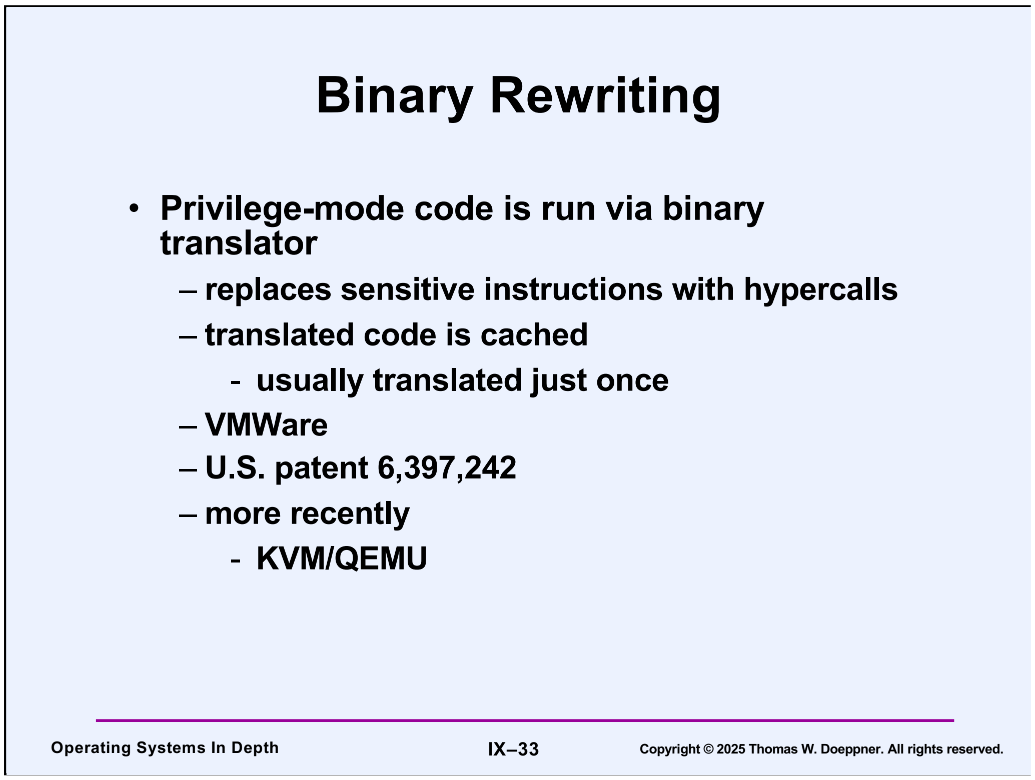
A Sensitive x86 Instruction

- popf
  - pops word off stack, setting processor flags according to word's content
    - sets all flags if in ring 0
      - including interrupt-disable flag
    - just some of them if in other rings
      - ignores interrupt-disable flag

What's important is that this instruction is sensitive, but not privileged. Thus, it doesn't satisfy the premise of our theorem.

What to Do?

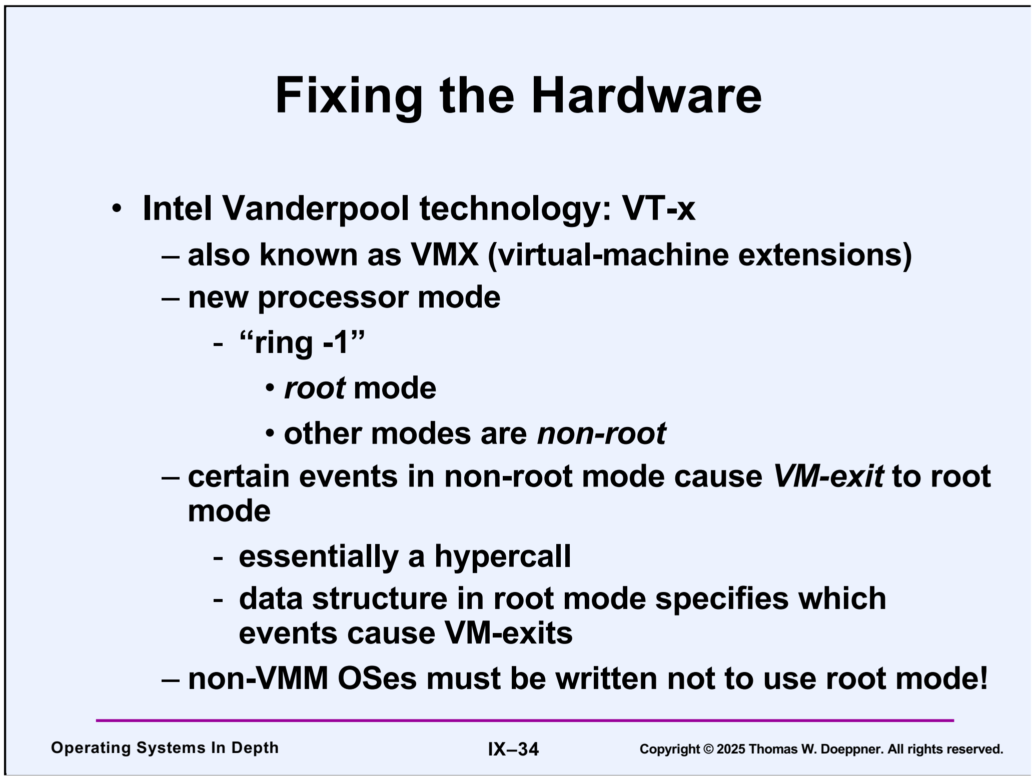
- Binary rewriting
  - rewrite kernel binaries of guest OSes
    - replace sensitive instructions with hypercalls
    - do so dynamically
- Hardware virtualization
  - fix the hardware so it's virtualizable
- Paravirtualization
  - virtual machine differs from real machine
    - provides more convenient interfaces for virtualization
    - *hypervisor* interface between virtual and real machines
    - guest OS source code is modified



Binary Rewriting

- Privilege-mode code is run via binary translator
  - replaces sensitive instructions with hypercalls
  - translated code is cached
    - usually translated just once
  - VMWare
  - U.S. patent 6,397,242
  - more recently
    - KVM/QEMU

This was mainly the work of Mendel Rosenblum, who is a CS professor at Stanford. He and his wife, Diane Greene, founded VMWare – Mendel was CTO and his wife was CEO. He became a fellow of the Association for Computing Machinery (ACM) for "contributions to reinventing virtual machines" (see https://en.wikipedia.org/wiki/Mendel_Rosenblum).

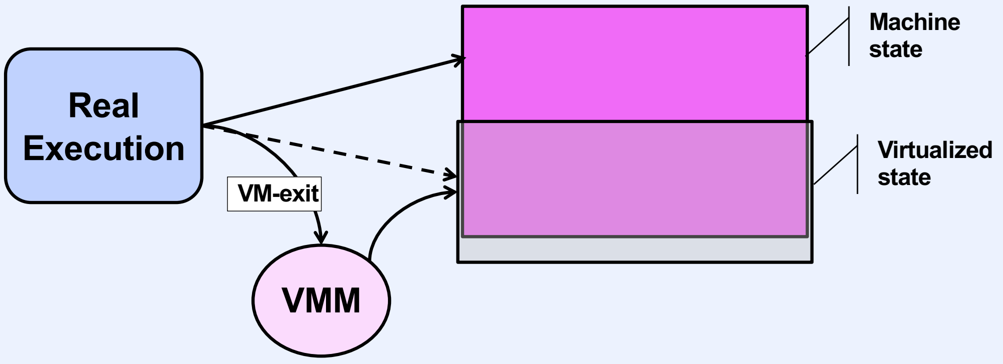


Fixing the Hardware

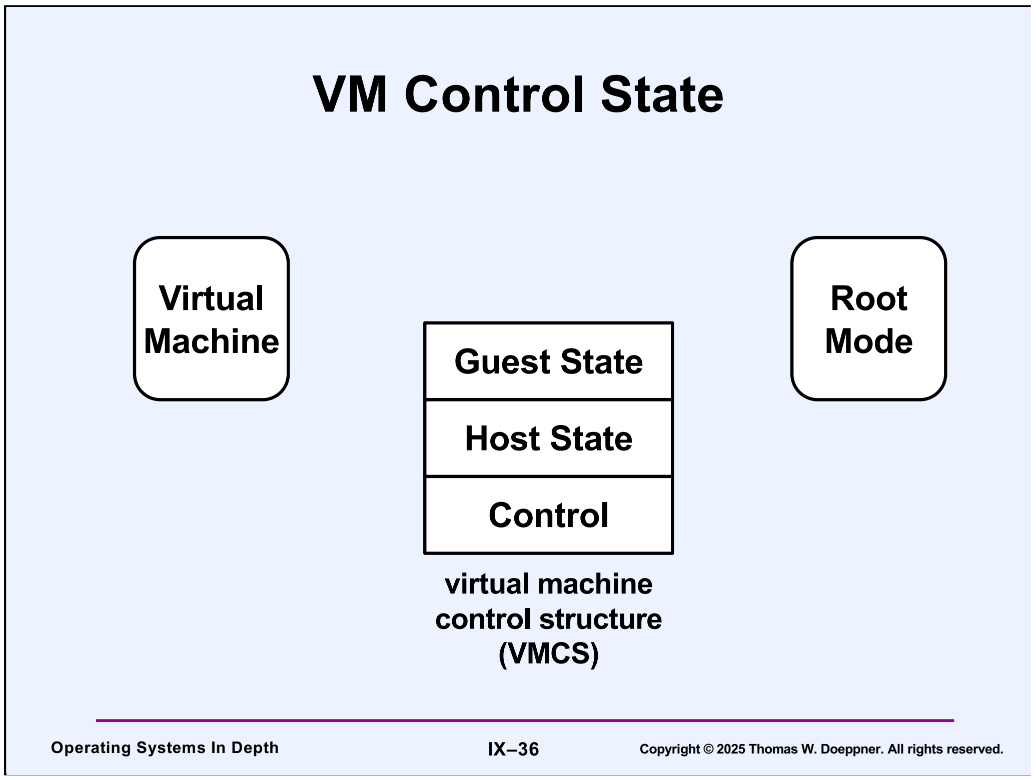
- Intel Vanderpool technology: VT-x
  - also known as VMX (virtual-machine extensions)
  - new processor mode
    - "ring -1"
      - *root* mode
      - other modes are *non-root*
  - certain events in non-root mode cause *VM-exit* to root mode
    - essentially a hypercall
    - data structure in root mode specifies which events cause VM-exits
  - non-VMM OSes must be written not to use root mode!

A detailed description of the VMX architecture can be found in Volume 3, Chapters 23 – 30 in the Intel 64 and IA-32 Architectures Software Developer's Manual, available at http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. While the concepts behind VMX are straightforward, the details are not: eight chapters covering 276 pages in the software developer's manual are required to describe it.

Virtual-Machine State

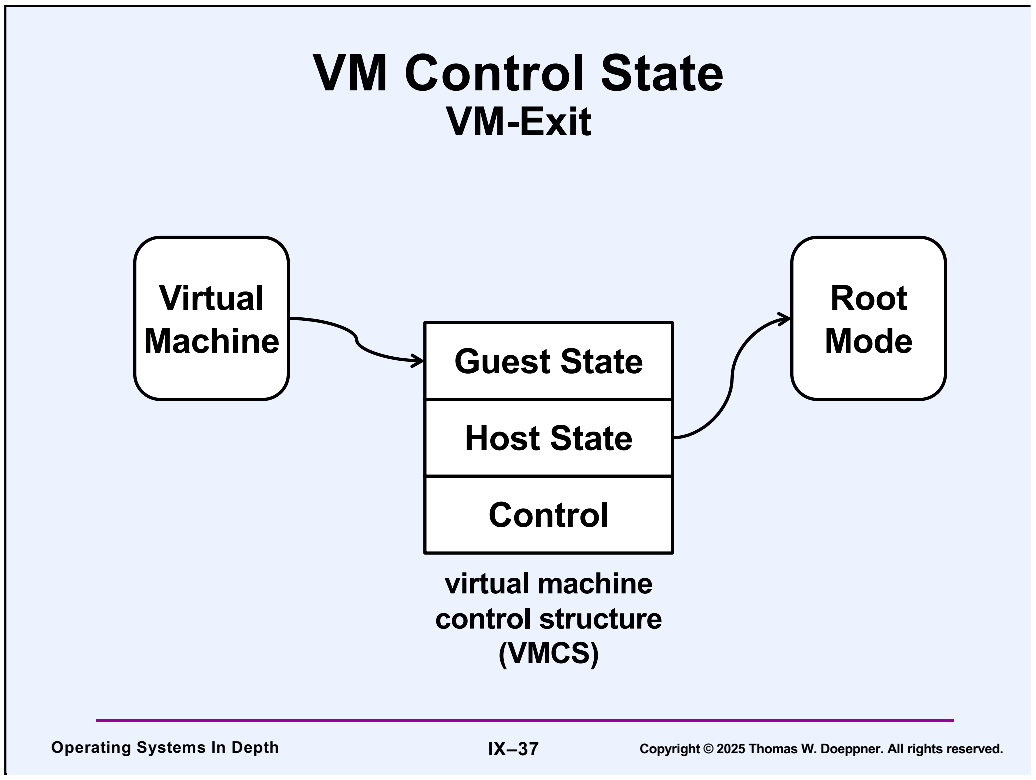


The status of a virtual machine (and of a real machine) is defined by the various state information, including memory, general-purpose registers, control registers, interrupt vectors, etc. For a real machine, all of this is manipulated directly via machine instructions. For a virtual machine, some of it (such as general-purpose registers and much of memory) is manipulated directly, but others (such as control registers) are manipulated via the virtual machine monitor as an intermediary. Intel's **vmx** architecture (and the equivalent of AMD) allow the system to designate which state information of the virtual machine is manipulated by direct execution within the virtual machine and which must be handled via the VMM. Any attempt by the virtual machine to manipulate the latter results in a **vm-exit**, which is a trap into the VMM, which then modifies the "virtualized" state.



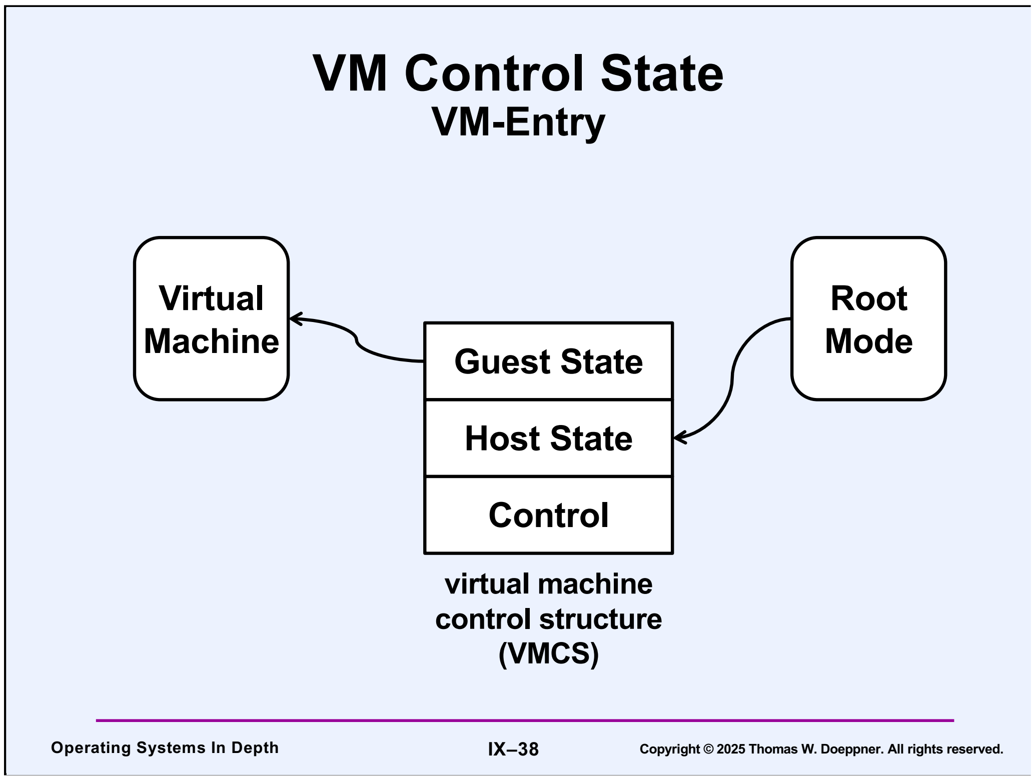
VM Control State

The VMM, running in root mode, maintains the Virtual Machine Control Structure (VMCS), one for each VM. The guest-state and host-state portions contain saved registers as described in the next slides. The Control fields describe what events cause VM-Exits, what information is saved on VM-Exits, and what information is restored on VM-Enters.



VM Control State
VM-Exit

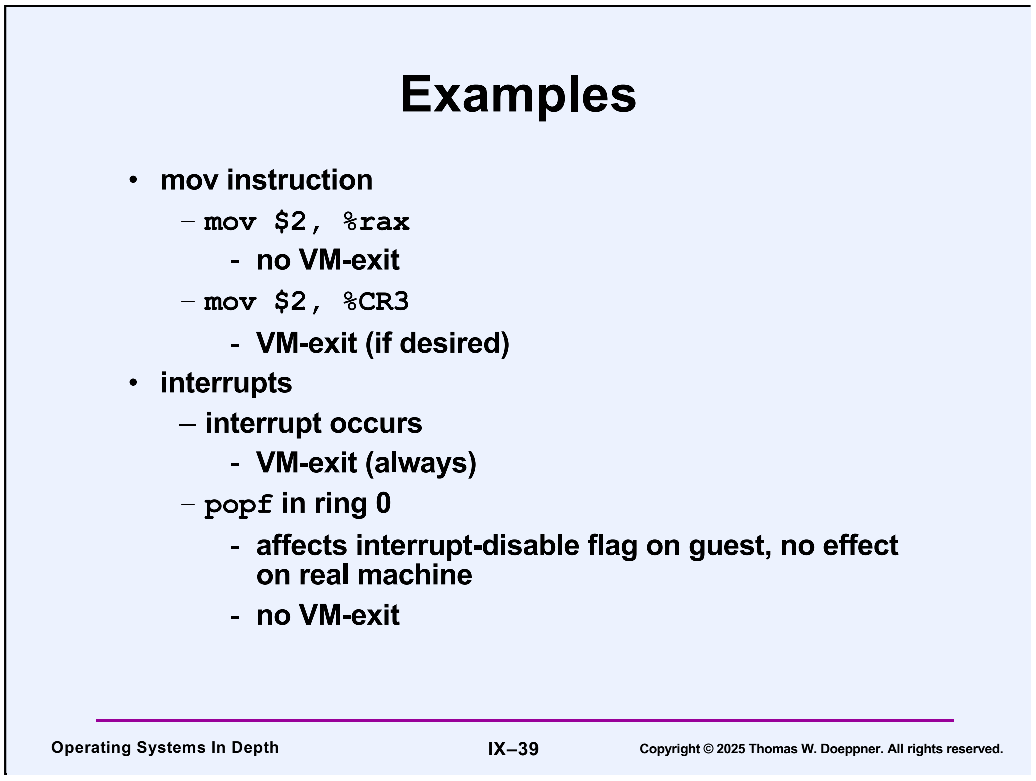
On a VM-Exit, the register state of the VM is saved in the guest state area; the saved register state of the VMM is restored from the host state area. At this point, the VMM may modify the actual machine state in response to what the VM is attempting to do.



VM Control State

VM-Entry

On a VM-Entry, the VMM's register state is saved in the host state area, the VM's register state is restored from the guest-state area.



Examples

- mov instruction
  - `mov $2, %rax`
    - no VM-exit
  - `mov $2, %CR3`
    - VM-exit (if desired)
- interrupts
  - interrupt occurs
    - VM-exit (always)
  - `popf` in ring 0
    - affects interrupt-disable flag on guest, no effect on real machine
    - no VM-exit

The CR3 (control register 3) register contains the address of the top-level page table. If this is modified, it has an immediate effect on the execution of instructions. The VMM can arrange that a VM-exit occur when the CR3 is modified, but, as we will see later in the semester, this might not be necessary.

The root VMM controls whether interrupts are enabled on the real machine. When an interrupt occurs, it's not necessarily the case that it has anything to do with the current virtual machine; thus it must be handled by the root VMM. If it is to be handled by (and is not masked by) the current virtual machine, the VMM would then push the VM's current state on its current kernel stack and enter the virtual machine at the appropriate interrupt handler.

If a guest OS, running in a virtual machine in ring 0 (non-root mode), attempts to modify the interrupt-enable flag (perhaps by executing the **popf** instruction), there's no need for a VM-exit. That interrupts are disabled are noted in the **rflags** register, which is among the registers stored in the guest-state field of the VMCS. When an interrupt actually occurs, assuming it's enabled on the real machine, a VM-exit occurs. The VMM is now running (in root mode) and it can check whether interrupts are enabled on the virtual machine by looking at the saved rflags register in the VMCS. If they are, then it can emulate the effect of an interrupt on the virtual machine.

Quiz 3

We've implemented recursive virtualization: VMM$_i$ runs on a VM supported by VMM$_{i-1}$, which runs on a VM supported by VMM$_{i-2}$, …, which runs on a VM supported by VMM$_0$, which runs on the real hardware. A VM-Exit takes place on a VM running on VMM$_i$.

a) It's handled first on VMM$_0$, is then handled on VMM$_1$, …, and finally on VMM$_i$.

b) It's handled first on VMM$_i$, which then VM-Exits to VMM$_{i-1}$, which the VM-Exits to VMM$_{i-2}$, …, which VM-exits to VMM$_0$.