CS250B: Modern Computer Systems

Privileged Operations and Virtualization

Sang-Woo Jun

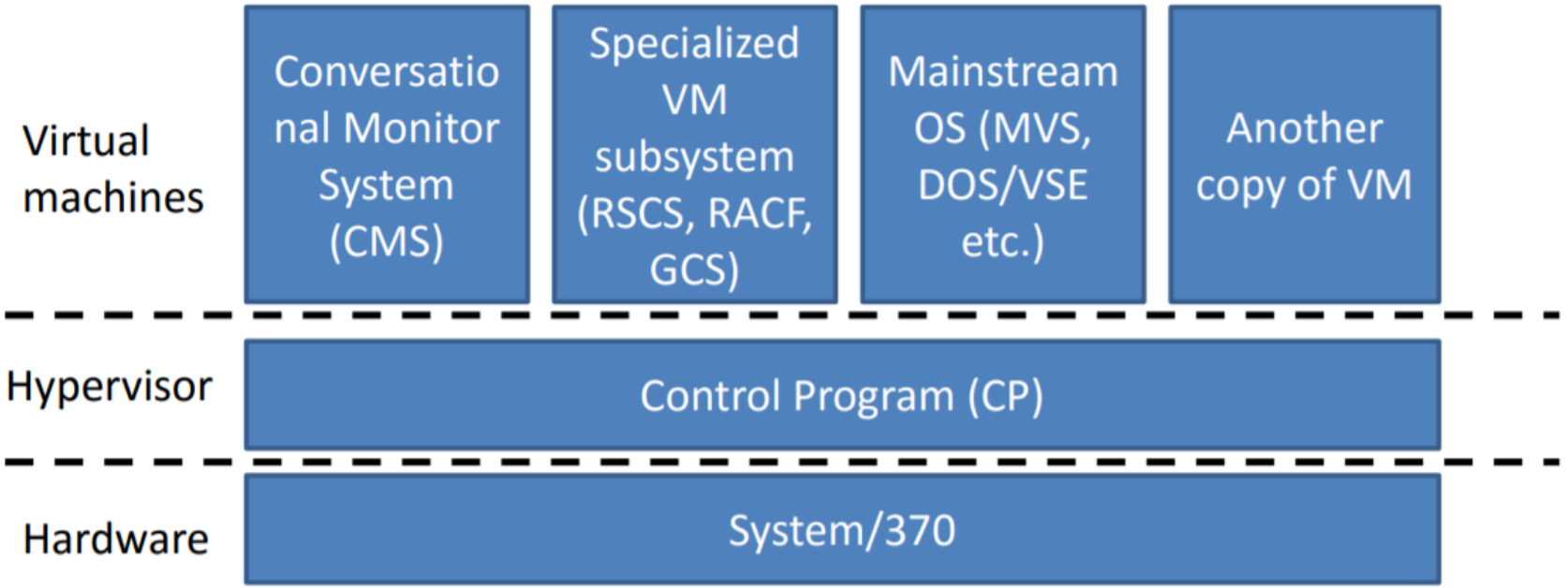
UCI

Historical Uses of Virtualization

- Application virtualization
  - Improves portability by running on virtual environment
  - JVM, .net, …
- System virtualization (topic for today)
  - Emulates full system hardware in software to create one or more virtual machine instances on a single hardware instance
  - Security/isolation, manageability, OS development, efficient use of resources (important topic!)
  - IBM VM/370, vmware, qemu, Linux KVM, …
- In the middle: OS-level virtualization (Docker, …)

IBM VM/370 Example

One of the first hardware-assisted virtualization instances



Oliver.Obi “3D Rendering of computer center with IBM System/370-145 and IBM 2401 tape drives”

Zhiming Shen, “Virtualization Technology,” CS 6410: Advanced Systems Fall 2016, Cornell University

Virtualization in the Cloud

- Virtualization is a fundamental piece of elastic clouds
- Reduces resource fragmentation, helps load balancing
  - For example, in a 8-core physical machine, four 2-core virtual machines can be spawned to efficiently use its resources
  - Without virtual machines, clouds will have to extremely accurately predict customer use cases, or suffer resource waste due to fragmentation
  - Reduce resource fragmentation, enabling efficient resource utilization for elastic resource allocation → Economy of scale that makes clouds viable
- Conveniently spawn and kill instances
- We will now focus only on system virtualization

But first and foremost, virtualization should be fast. Otherwise, it's pointless for the cloud

How Does Virtualization Work? The Naïve Way

- Write a software interpreter
  - A piece of software completely implements the CPU ISA and surrounding hardware
  - e.g., Bochs system emulator
- Pros:
  - Completely isolated, user-space implementation
  - Can emulate guest systems unrelated to host
  - Bochs is very useful for operating system development
- Cons: Very very slow!
  - Typically 100x slower

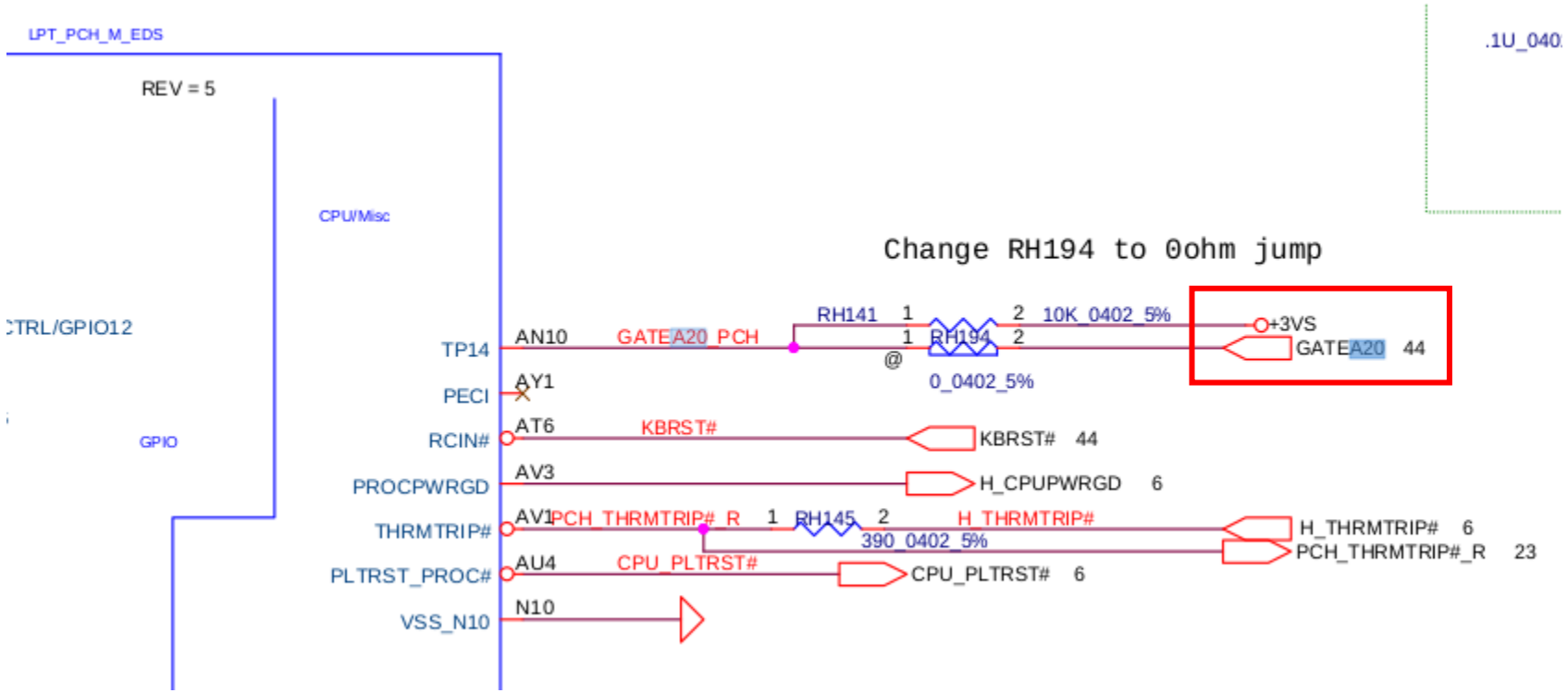
Bochs logo

Before We Go On – Protected Mode Recap

- Modern x86 CPUs have “real mode” and “protected mode”
  - On boot, BIOS/UEFI loads bootloader from storage into memory, and CPU starts executing it in real mode
  - Real mode has 1 MB addressable memory, no virtual memory or memory protection
  - The bootloader loads the kernel and executes it, which populates the virtual memory data structures for the CPU, among other bookkeeping, and switches forever into protected mode by setting a control register
  - From here, all memory accesses are through virtual memory (via TLB and virtual memory table)

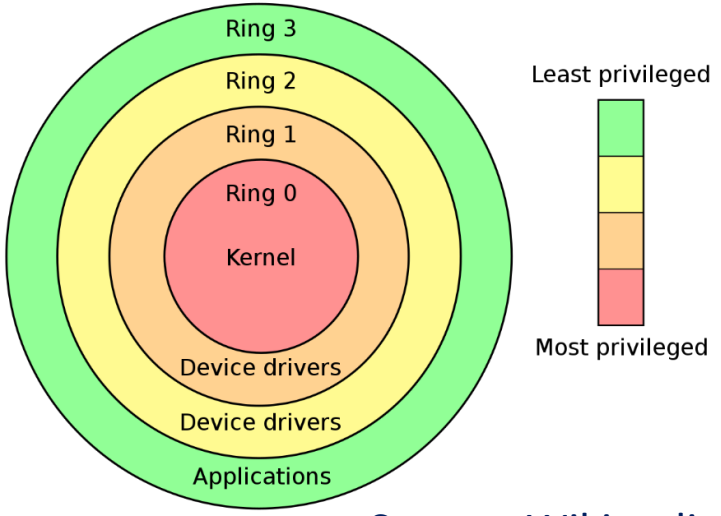
Also, A20 line… ☹

Tangent: The A20 Physical Pin?!



At least still existent on some Haswell (~2015) processors!
Skylate doesn't have it, it seems

Before We Go On – Protection Rings Recap



Source: Wikipedia

- Modern CPUs assign different levels of access per process/thread
  - A process‘s ring determines which subset of instructions it can execute
  - Lower levels are more privileged, can execute all instructions that upper rings can
  - x86 CPUs have four rings, but most OSs use only two (0 : “Supervisor mode”, and 3 : “User mode”)
- “Privileged Instructions” can only execute while in ring 0 (Kernel)
  - Managing virtual memory mappings, modify control registers, etc
  - Attempting one in user mode results in “general protection fault” exception
    - GPF can be for many other reasons as well…

Before We Go On – Exceptions Recap

- OS must supply the CPU with exception handlers
  - On x86, a table (“Interrupt Descriptor Table”) of pointers to each handler
  - On an exception (e.g., GPF), execution jumps to corresponding handler with information about where it happened
  - Handler runs in ring 0, and can do what it wants to handle or not handle the exception

Back To Virtualization – Native Execution

- If virtual and host ISA is identical, most instructions can be run as-is
- Virtual Machine Manager creates a virtual system environment, (memory, display, etc) in userspace, and tries to execute OS code as if it is user software
  - Privileged instruction attempts are caught via exceptions, and handled by VMM to emulate what should have happened
  - The VMM must have kernelspace access! – Typically what is called Hypervisor
- Pros: Very high performance – Almost no overhead for computation-bound applications

Some Issues With Native Execution

- Some privileged instructions don’t generate exceptions in user mode
  - popf (Pop flags) fails silently
- Guest virtual memory is cumbersome
  - Another layer of translation: Guest virtual memory -> Guest physical memory (host virtual memory) via virtual page table -> Host physical memory via physical page table

Binary Translation

- Typically used as performance optimization for cross-platform virtualization
- All software that is to run on a VM is translated during load to work better with the VM
  - Translated software (even OSs) can run just like normal software
  - Software for different ISA is translated to host ISA
  - Example: JVM JIT
- Special instructions are changed to point to handlers in VM
  - Interrupts, privileged instructions, etc now call handlers – Solves the silent failure problem for native execution
  - Jump targets are overwritten

Binary Translation

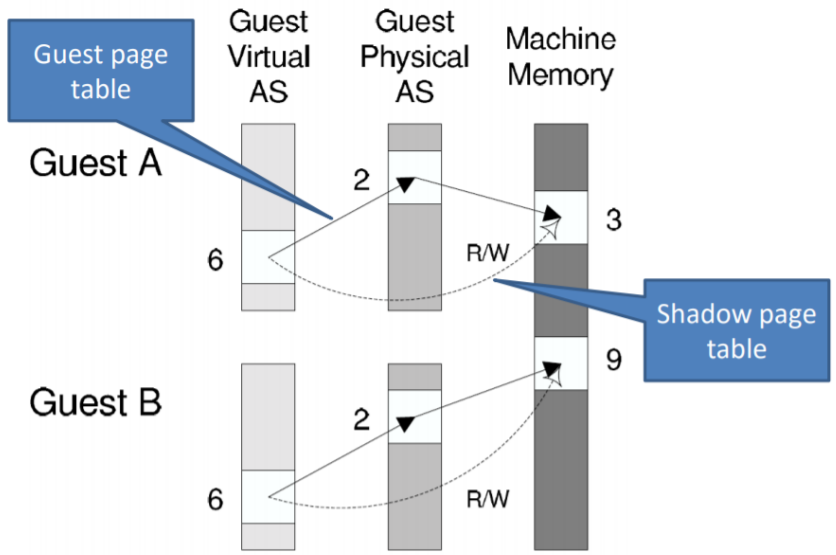
- Issue: Indirect jumps
  - Jump targets depending on runtime variable is difficult to predict
  - Re-translating every time has a high performance overhead
  - We could create an index of the addresses of all original instructions and their translations – Intractable overhead!
  - Typically a balance of the two
  - Not an issue with native execution
- Issue: Self-modifying code
  - Sometimes need to check for modifications and fall back to software interpreter

Shadow Page Table

- In a naively virtualized system, there are two page tables for the same guest memory access
  - Page table in the virtual CPU, pointing to virtual physical memory (host virtual memory)
  - Page table in the host CPU, pointing to host physical memory
  - During virtual virtual memory access, virtual CPU needs to do translation, harming performance
- For performance, a VMM can store guest memory mappings directly in host page table (guest virtual memory to host physical memory)
  - Guest MMU does no translation, and simply depends on host MMU to do the right thing

Shadow Page Table

- Guest OS page table is write-protected in host memory
  - Host OS intercepts page table updates and populates a “shadow” table
- Virtual CPU ignores its page table, and forwards requests to host
  - Single layer of translation, using host’s physical virtual memory hardware
  - Shadow table not visible to guest!



The Modern Way – Hardware-Assisted Virtualization

- Newer CPUs have hardware support for virtualization, which renders many of the above unnecessary
  - Intel VT-x, AMD-V
- Introduces the concept of ring -1, and a few more instructions
  - Hypervisor boots into ring -1, and uses ring -1 instruction (VMLAUNCH, etc) to spawn/manage/terminate VMs
  - VMs start in ring 0, thinking it has full control of CPU
- Interrupts are delivered to hypervisor for it to manage
  - Timer interrupts, etc used to bring execution back to hypervisor

The Modern Way – Hardware-Assisted Virtualization

- Virtual memory management also moved to ring -1
  - Second Level Address Translation (SLAT), or “nested paging”
  - Intel Extended page table (EBT), AMD Stage-2 MMU
- Now virtual memory translation can be nested in hardware
  - Hardware performs the translation from the guest physical address to host virtual address
  - Separate hardware registers for specifying guest and VMM VM location

Virtualizing Peripherals

- Network, storage, etc,…
- Typically a small selection of generic virtual devices are provided to the virtual machine
  - Only the hypervisor knows of the actual hardware
  - Hypervisor performs scheduling as it sees fit
- When raw access must be given to a guest, the access is exclusive
  - Class of devices a generic catalog was not provided for
  - hypervisor acts as a raw bridge
- Some modern peripherals come with their own virtualization support
  - Per-VM queues and contexts

Paravirtualization

- Guest OS is modified to communicate with the hypervisor
  - Guest OS sees physical memory, and must work with hypervisor to cooperatively manage memory
  - Privileged instructions are changed to requests to hypervisor (hypercalls)
- Can greatly simplify hypervisor, improve performance