Virtualization

15-410 Fall 2006

Glenn Willen
Mike Cui

Synchronization

- Kernel due tonight
- If you are using your late days, don't forget to register on the website
- Be alert for an opportunity to study large warm floppy disks at midnight

Outline

- Overview
- The Game (Why/How Virtualization?)
- Other Stuff
  - Virtualization on x86
  - Paravirtualization
  - Hardware Assisted Virtualization
  - Software Implementation

Virtualization

Process of presenting and partitioning computing resources in a *logical* way rather than what is dictated by their *physical* reality

Virtual Machine

An execution environment identical to a physical machine, with the ability to execute a full operating system

Q: Process : OS :: OS :
A: Virtualization layer

IBM System 370

VM/CMS ~1967

VM - Virtualization Layer

CMS - Single-user DOS-like operating system

1000 users, each user gets a personal mainframe!

Motivation

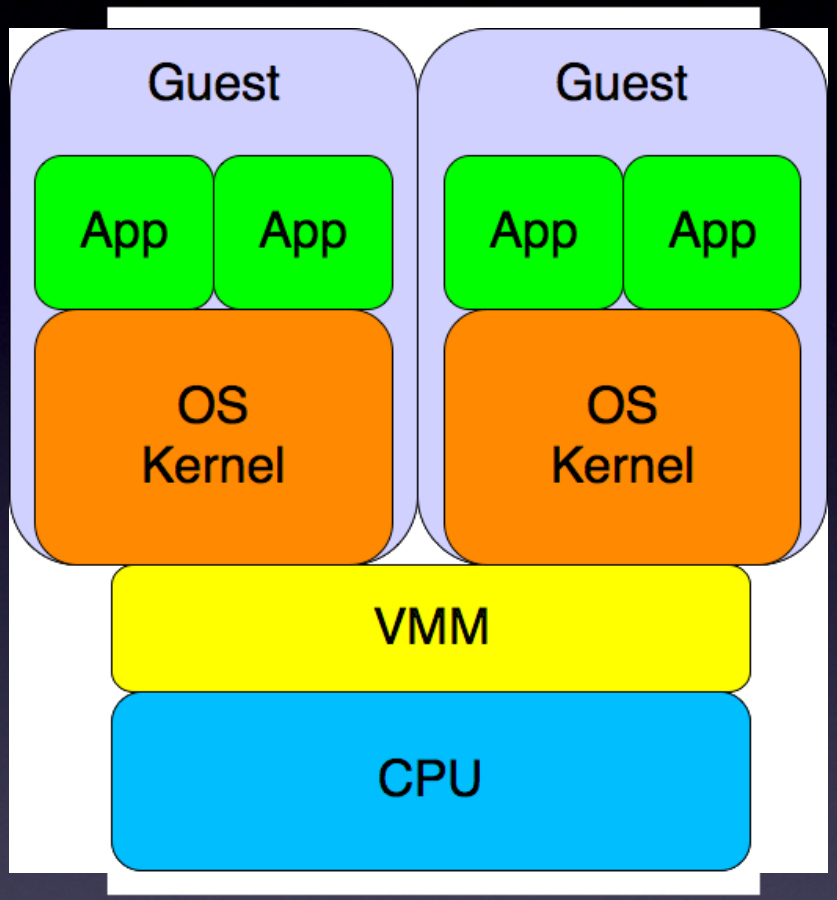
- Fake hardware (e.g. for old code)
- Multiple OSes simultaneously
- Efficient use of resources
  - Can share one machine's resources between services which require the isolation of separate machines
  - Can move virtual machines among physical machines to load-balance
- Billion-dollar industry

Motivation for You

- Virtualization is cool
- Application of OS concepts you already know from this class
- Impress VMware interviewer

Virtual Machine Layers

The virtualization layer is commonly referred to as the *Virtual Machine Monitor / Virtual Memory Monitor* (*VMM*) or *Hypervisor*



Virtualization Layer

- VMM: Virtual {Memory|Machine} Manager
- Runs with the highest privileges
- Controls and allocates hardware resources for the virtual machine(s)
- It is the “operating system”

Guest OS

- CPL 0 code running at CPL > 0
  - Code that assumes it's privileged, running without privileges
- Accesses physical memory
- Manages virtual memory
- Installs interrupt / system call handlers
- Controls hardware devices
- [Does not know about the VMM]

Why and How?

Scenario: Virtualization to fake different hardware
(Rationale: Simpler example than multiple guest OSs)

All we need to do is control and redirect access to hardware

- Make the guest OS see our virtual hardware instead of the real thing

…how hard can it be?

What Won't Work

Absolutely cannot run the guest OS in kernel mode (ring 0). Why?

What Won't Work

Absolutely cannot run the guest OS in kernel mode (ring 0). Why?

- No way for us to keep control of the machine!
  - No way to keep the guest away from physical devices!
  - Once we let it run, we never get control back; guest runs I/O instructions, nothing we can do.

So What's Left?

No choice but to run the guest kernel in user mode (ring 3)

… and strictly maintain the illusion that it's in kernel mode!

- This part is slightly difficult…

What do we need to do (and how do we do it?)

The Game

- Virtualization requirements, OR
- “How to play pretend with a kernel”

- In order of importance:

  - Protection
  - Illusion
  - Performance

The Game (Part 1)

- Protection: Virtual machine _must not_
  - Alter the state of the VMM (directly)
  - Interact directly with the hardware
  - Alter the state of any other VMs running side-by-side with it
- Rule: VM must not be able to alter anything outside the simulation
  - Otherwise it's a pretty bad simulation!

The Game (Part 2)

- Illusion:
  - VM must not (with the one exception of timing) be able to determine that it is running in a simulation
  - Say it with me: Otherwise it's a pretty bad simulation!
  - Critical for software which is not aware, and cannot be made aware, that it is being simulated (i.e. closed source)

The Game (Part 3)

- Performance:
  - A secondary requirement, but still important
  - The system must be fast enough to use, otherwise nobody will!
    - If we didn't care about speed, we would be simulating in pure software.
  - Difficult to make this a hard requirement -- very much best-effort

How?

- *Keep guest OS off of I/O instructions*
  - *Basically free: they are privileged, will automatically trap into the VMM*
- Keep guest OS off of other “sensitive” instructions
  - Which ones are these?
- Keep guest OS out of VMM’s memory
  - Slightly more difficult…

How, Part I (Trap and Emulate)

- Guest tries:
- ```
  outb(TIMER_PERIOD_IO_PORT, timerperiod >>
  8);    /* timer period MSB */
  ```
- To allow this would violate *protection*: the guest is attempting direct hardware access
- We're saved: the OUT instruction is privileged; since the guest is running in ring 3, a general protection fault occurs. TRAP!
  ```

How, Part I (Trap and Emulate)

- VMM now must emulate the instruction that the guest was attempting, to maintain the *illusion*.
  - Guest is attempting to change period of the system timer
  - Instead, write down new period for guest's virtual timer
  - The *real* (VMM) timer handler will "call" (read: fake a trap into) the guest's handler as appropriate, to maintain the illusion that the guest has control of the real hardware timer.
  - If the guest tries to read the period of the hardware timer, same game in reverse: tell it the period of its virtual timer device instead. Maintain the *illusion*!

How?

- Keep guest OS off of I/O instructions
  - Basically free: they are privileged, will automatically trap into the VMM
- *Keep guest OS off other "sensitive" instructions*
  - *Which ones are these?*
- Keep guest OS out of VMM's memory
  - Slightly more difficult…

How, Part 2 (Sensitive instructions)

- Any instruction which, used by the guest, puts the simulation at risk, and which we therefore must simulate
- Handle these the same as I/O instructions (which are really just an example of sensitive instructions.) But which instructions? E.g.:
  - `cli`
  - `outb`
  - `movl %eax, %cr3`
  - Luckily, all the examples given here will trap into the VMM if run by the guest in ring 3, so there's no problem
- Q: What do we do with other guest kernel instructions?
  - `addl $3, %eax`
  - A: Just run them!

How?

- Keep guest OS off of I/O instructions
  - Basically free: they are privileged, will automatically trap into the VMM
- Keep guest OS off other “sensitive” instructions
  - Which ones are these?
- *Keep guest OS out of VMM’s memory*
  - *Slightly more difficult…*

How, Part 3 (Memory Protection)

- Physical memory is a shared resource
- Do not allow guest to access it directly
- Instead, map *fake* "physical" pages for the guest
  - Provide guest with the *illusion* of access to all of physical memory
- Guest never knows about *real* machine frames
- How?

Virtual Memory Virtualization

- Guest OS itself will need to use virtual memory
- Need 2 level virtual address translation
  - virtual page to *fake* "physical" page
  - "physical" page to real machine frame
- But the memory management hardware only gives us one…

Virtual Memory Virtualization

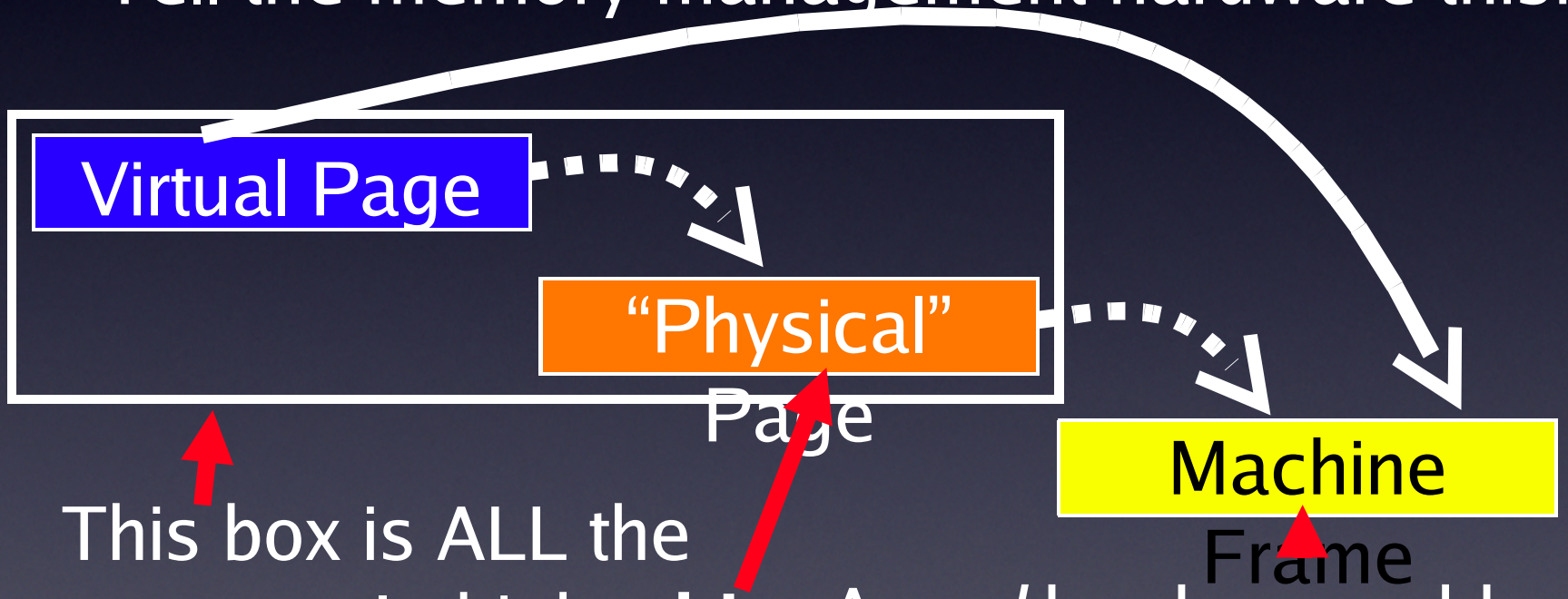
- *Need 2 level virtual address translation*
  - *virtual page to fake "physical" page*
  - *"physical" page to real machine frame*

Virtual Memory Virtualization

- But how can we get the memory-management hardware to do this for us? It can only handle a single level mapping.

Virtual Memory Virtualization

Tell the memory management hardware this:



This box is ALL the guest sees. It thinks **this** is a physical address.

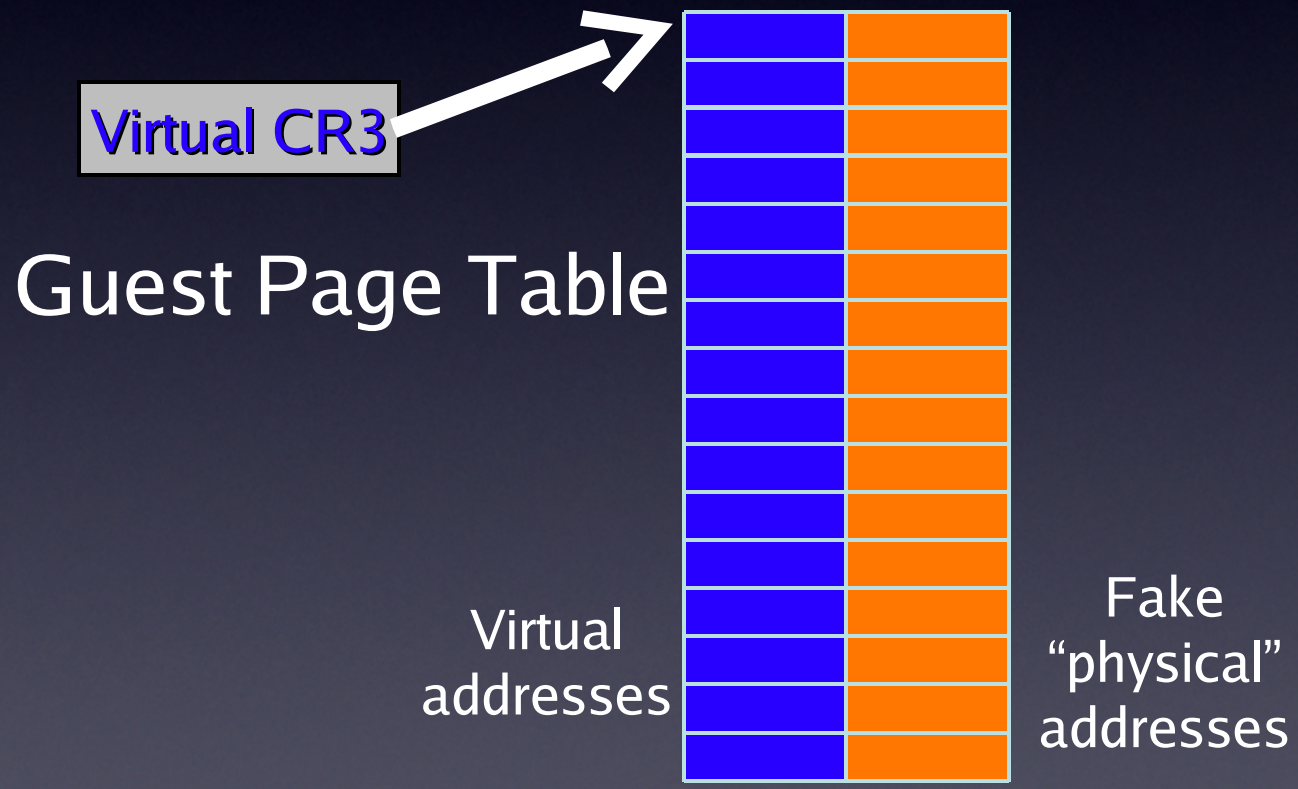
A *real* hardware address: The guest NEVER sees this.

Shadow Page Tables

- Let guest maintain page tables according to the *illusion*, mapping virtual pages to fake "physical" pages
- VMM tracks mappings from the fake "physical" pages to the real machine frames
- For the memory management hardware, maintain a set of "shadow" page tables mapping guest virtual pages => machine frames
  - Every time the guest sets CR3, or updates its page tables:
    - Look up the fake "physical" => real machine page mapping(s)
    - Update the "shadow" page tables to match

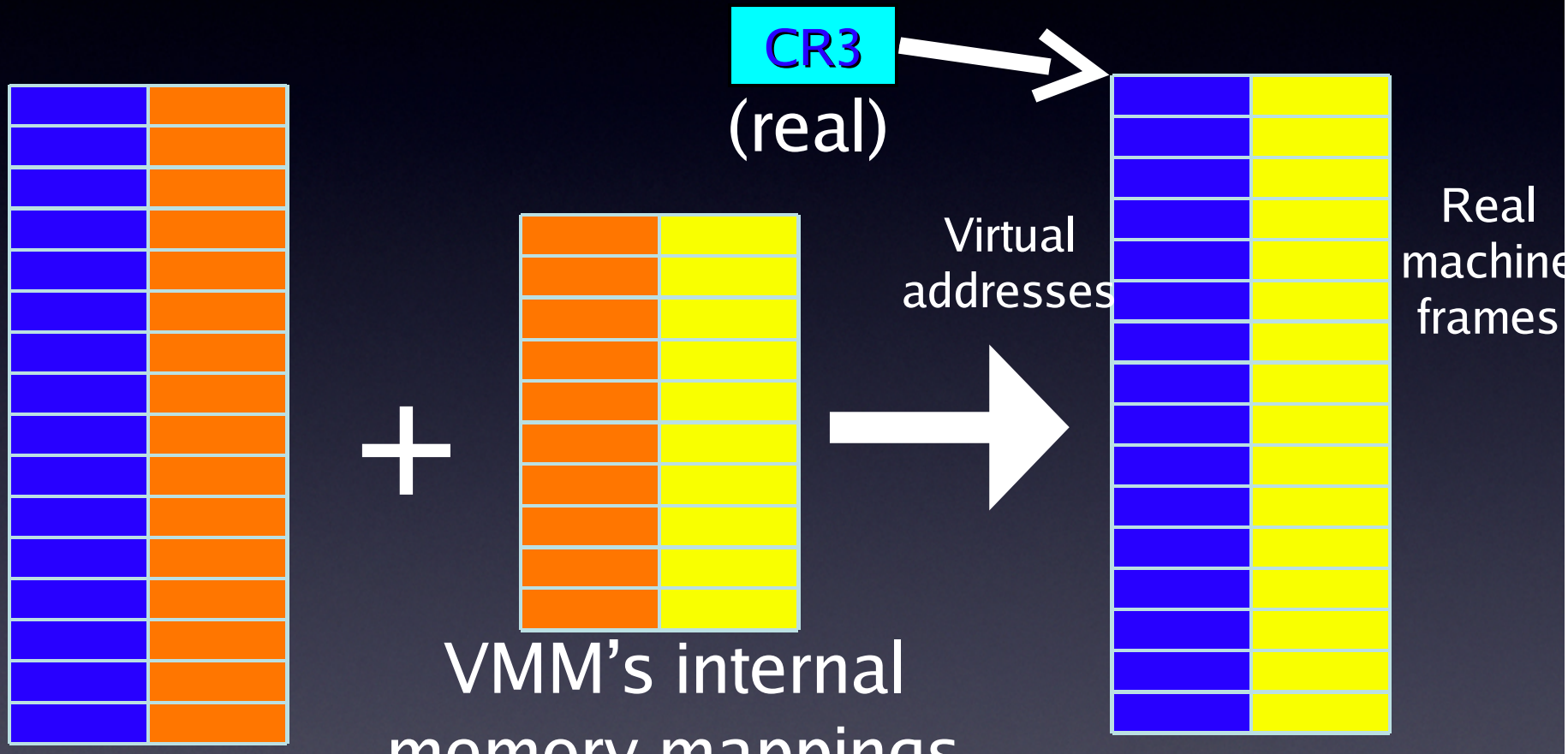
Shadow Page Tables

- Let guest maintain page tables according to the *illusion*, mapping virtual pages to fake "physical" pages



Shadow Page Tables

- For the memory management hardware, maintain a set of "shadow" page tables mapping guest virtual pages to machine frames



Shadow Page Tables

- *Protection:* The guest OS cannot (directly) affect the real page tables
  - Page tables are protected from editting
  - More important: the rest of the universe is protected from the guest
    - Guest has no way to touch parts of machine memory that it shouldn't know about

Shadow Page Tables

- *Illusion:*
- Need the guest to think it can read/write CR3
- That's a privileged instruction, so it traps into the VMM, we simulate it in sofware and advance EIP to the next instruction
- Need the guest to get the "right" value back when it reads CR3
- When we simulate the read, give back the address of the fake guest page tables instead of the real shadow page tables
- Extra-subtle: What kind of address is it?

Managing Shadow Page Tables

- On every write to the page directory or a page table by the guest, we need to
  - Maintain an *illusion*: update the guest's fake page table, so the next read will give the right answer
  - Maintain *protection*: update the real shadow page tables to contain the right thing
  - We do this by *tracing* accesses to the guest's page tables

Tracing

- Use memory protection to trap writes to page tables
  - Very similar to copy-on-write
  - Tell the guest it can write to the memory areas it thinks are page tables
  - Mark them read-only in reality, so writes to them trap into the VMM
  - Do the write that faulted, and then take the opportunity to update the shadow page table to keep it in sync

Tracing

- *Use memory protection to trap writes to page tables*
- `pte[17] =`
- `  alloc_new_frame() | PTE_PAGE_PRESENT`
- Guest tries to add a mapping for a new frame into the page table
- Address from alloc_new_frame() is not *really* a machine frame, it is a fake “physical” page which the VMM will map to an actual frame. (The guest kernel does not know this!)
- This assignment will cause a page fault, trapping into the VMM
  - VMM does the write on behalf of the guest (to maintain the *illusion* that the guest controls the page tables)
  - VMM updates the shadow page tables to contain the machine frame address corresponding to the fake “physical” frame the guest is trying to map

Performance?

- Quite slow to rebuild shadow page tables every time CR3 changes, and trap every time the guest writes its page tables. Can we speed this up?
  - Caching: When CR3 changes, keep old shadows around, for recycling when it changes back
    - Subtle problem: garbage collection! How long do we keep old shadow page tables around, before deciding that the guest has freed that context and will never switch to it again?
    - Subtle solution: Guess.
      * Some patterns of memory access look like a page table. Most of them don't.
      * If we haven't seen it in awhile, maybe it's not coming back

Performance?

- Quite slow to rebuild shadow page tables every time CR3 changes, and trap every time the guest writes its page tables. Can we speed this up?
  - Lazy updates: Don't usually need to update the *real* mappings until the guest flushes the TLB
    - Don't trace the page tables at all; instead, just trap CR3 writes and INVLPGs.
    - Guest just sees a Very Big TLB.
  - Very serious speed gains to be had here, but hugely complicated to get right

x86 Hardware

- x86 instruction set is not “virtualizable”
- 17 instructions silently behave differently in privileged versus non-privileged mode
  - Example: POPF
    - In user mode, “helpfully” pops some flags but ignores privileged flags
    - Would be much more helpful (to VMM) if it trapped instead
    - Should really be two instructions (one privileged, one not)

Virtualization on x86

- Possible, through a lot of clever hacks
- VMware (1998)
  - Binary translation: Edits problematic guest kernel instructions to be traps into VMM
    - E.g. POPF becomes INT 99
      - It's not really anywhere near that simple.
  - Directly execute guest user code
  - Only 20% performance overhead, or less

Paravirtualization Motivation

- Full virtualization is expensive and complicated
- If guest OS can be modified to work with the hypervisor, then it's unnecessary to
  - Trap and emulate
  - Shadow and trace
  - Dynamically recompile guest kernel code

Paravirtualization Implementation

Guest OSes are modified to accept the fact that they are is running inside a VM.

The VMM does not create an *illusion* that the VM owns all machine resources.

Instead, the VMM provides an *hypercall interface* to provide service to VMs.

Hypercall Interface

- Allows the guest to *voluntarily* trap into the hypervisor
- All guest access to hardware state happens through hypercalls
- The guest does not access hardware state directly at all
- No instruction decoding necessary

Physical Memory

- Guest knows that it does not own all of physical memory
- Guest requests physical memory from hypervisor
- There are no fake “physical” addresses; guest knows real frame addresses

Virtual Memory

- Guest relinquishes ownership of its own page tables
- All paging operations happen through hypercalls

```
int frame = hypervisor_frame_please();
 /* Politely ask for a physical frame*/
map(virtaddress, frame);
 /* Ask to map into our address space */


map_many(virtaddrs[], framenums[]);
 /* Works like set_cr3() */
```

- No shadows or traces:

Hardware Abstraction

Paravirtualization = Hardware abstraction

The hypervisor abstracts away all the hardware details from the guest operating system

Requires porting guest OS to hypervisor, as if it was another hardware platform

Hypervisor Examples

- Xen - portable hypervisor
  - Hypercall interface defined for *any* ISA
    - mmu_update()
- VMware - x86 specific
  - Hypercall interface similar to hardware
    - VMI_SetCR3()
- News Flash! VMI is now a portable interface to both VMware and Xen hypervisors

Hardware Assisted Virtualization

- Intel VT (Vanderpool) - Core Duo/Solo
- AMD SVM (Pacifica) - soon to come
- Provides a hardware mode of operation for virtual machines
- Architectural extension to make x86 virtualization *easier*
  - Does not *replace* the VMM

Hardware Assisted Virtualization

- Key feature: Supports a mode in which the 17 “problem child” instructions will trap into the VMM instead of silently doing the wrong thing
- Key problem: Offers no support (yet) for making the page table shadowing problem any easier

Hardware Assisted Virtualization

- Boneheaded: It turns out that well-optimized binary translation is currently *faster* than the weirdo-traps that happen in the special virtualized mode.
  - (It's ok hardware guys, you'll get it next time.)

Summary

- Full Virtualization
  - Trap and emulate, shadow and trace
- Paravirtualization
  - Voluntary trap, hardware abstraction
- Hardware Assisted Virtualization

Further Reading

- J.S. Robin, and C.E. Irvine, “Analysis of the Intel Pentium’s Ability to Support a Secure Virtual Machine Monitor”
- M. Rosenblum, and T. Garfinkel, “Virtual Machine Monitors: Current Technology and Future Trends"
- S. Devine, E. Bugnion, and M. Rosenblum, “Virtualization system including a virtual machine monitor for a computer with a segmented architecture”, US Patent 6,397,242
- P. Barham, et. al., “Xen and the Art of Virtualization”
- “Xen Developer’s Reference”
- “VMI Specification” from VMware
- R. Uhlig, et. al., “Intel Virtualization Technology”
- “Intel VT Specification”
- M. Rosenblum, et. al., “Optimizing the Migration of Virtual Computers”

Disclaimer: This slide not modified from last semester. But most of it’s probably still interesting.