Virtualization

Dave Eckhardt
Dave O'Hallaron

based on material from:
Mike Kasick
Roger Dannenberg
Glenn Willen
Mike Cui

Nov. 20, 2017

Outline

- Introduction
  - What, why?
- Basic techniques
  - Simulation
  - Binary translation
- Kinds of instructions
- Virtualization
  - x86 Virtualization
  - Paravirtualization
- Summary

What is Virtualization?

- **Virtualization:**
  - **Practice of presenting and partitioning computing resources in a *logical* way rather than partitioning according to *physical* reality**
- **Virtual Machine:**
  - **An execution environment (logically) identical to a physical machine, with the ability to execute a full operating system**

Process vs. Virtualization

- The *Process abstraction* is a “weak, fuzzy” form of virtualization
  - Many process resources exactly match machine resources
    - %eax, %ebx, …
  - Some machine resources are not visible to processes
    - %cr0
  - Some process resources are “inspired by” hardware
    - SIGALARM
  - Some process resources are “invented” - don't match any hardware feature
    - “current directory” and “umask”
- Virtualization is “more like hardware” than processes
  - What runs inside virtualization is an operating system

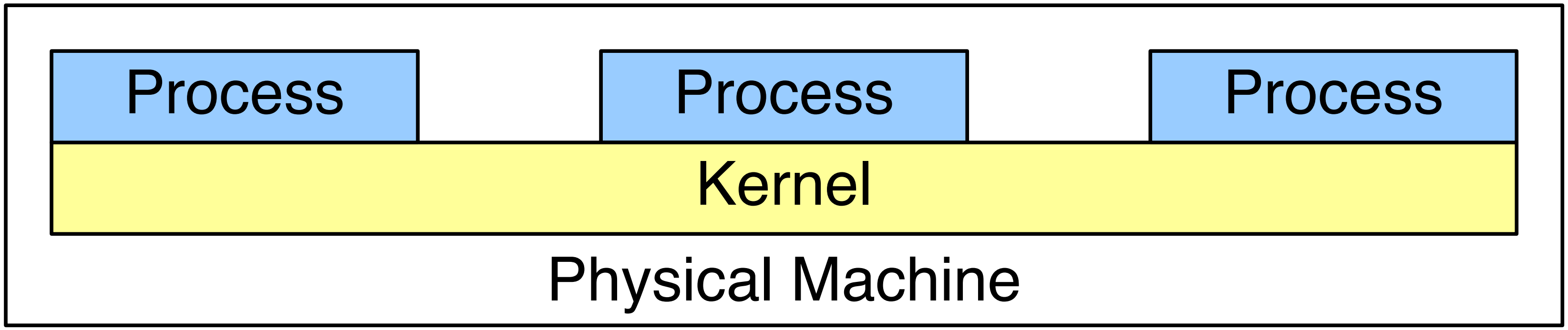
Process : Kernel :: Kernel : ?

Process vs. Virtualization

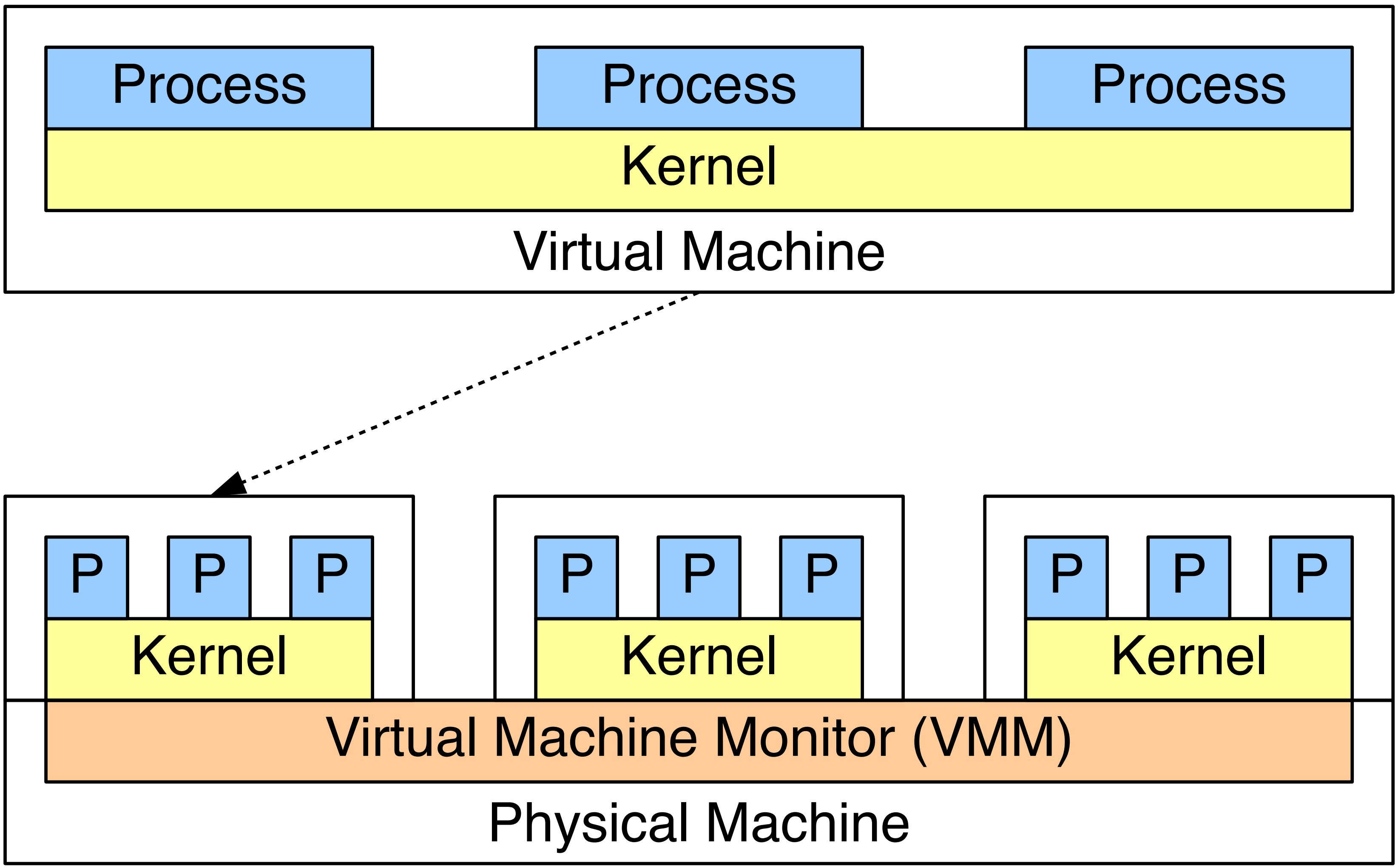
- **The *Process abstraction* is a "weak, fuzzy" form of virtualization**
  - **Many process resources exactly match machine resources**
    - **%eax, %ebx, …**
  - **Some machine resources are not visible to processes**
    - **%cr0**
  - **Some process resources are "inspired by" hardware**
    - **SIGALARM**
  - **Some process resources are "invented" - don't match any hardware feature**
    - **"current directory" and "umask"**
- **Virtualization is "more like hardware" than processes**
  - **What runs inside virtualization is an operating system**

Process : Kernel :: Kernel : Virtual-machine monitor

Process/Kernel Stack



Virtualization Stack



Why Use Virtualization?

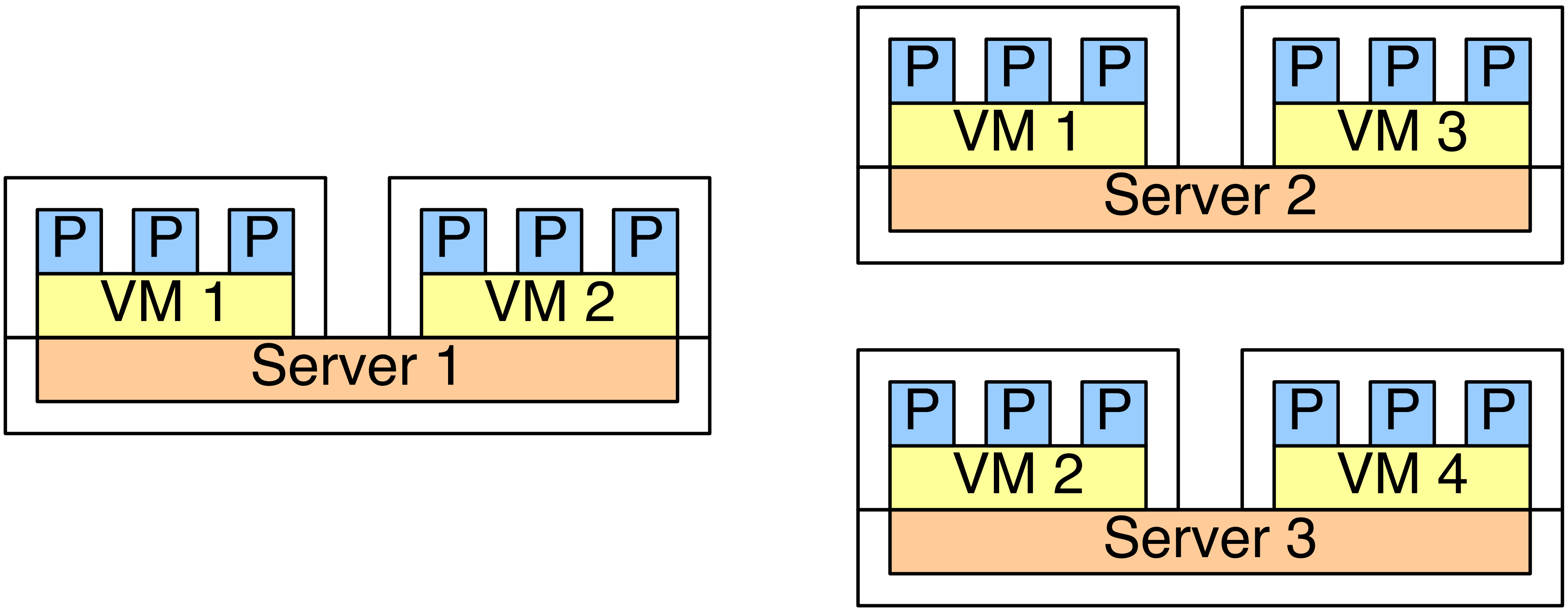
- **Run two operating systems on the same machine!**
  - **“Windows+Linux” was VMware's first business model**
  - **Hobbyists like to run ancient-history OS's**
- **Debugging OS's is more pleasant**
  - **Also: instrumenting what an OS does**
  - ***Monitoring a captive OS for security infestations***
- **“Process abstraction” at the *kernel* layer**
  - **Separate file system**
  - **Multiple machine owners**
  - **Better protection than one kernel's processes (in theory)**
    - **“Small, secure” hypervisor, “small, fair” scheduler**

Why Use Virtualization?

- ***Huge*** **impact on enterprise hosting**
  - **No longer need to sell whole machines**
  - **Sell machine *slices***
    - **"xx GB RAM, yy cores" - smoother than "n Dell PowerEdge 2600's"**
    - **Can put competitors on the same physical hardware**
- **Can separate instance of VM from instance of hardware**
  - **Live migration of VM from machine to machine**
    - **Deal with machine failures or machine-room flooding**
  - **VM replication to provide fault tolerance**
    - **"Why bother doing it at the application level?"**
- **Can overcommit hardware**
  - **Most VM's are not 100% busy all the time**
  - **If one suddenly becomes 100% busy, move it to a dedicated machine for a few hours, then move it back**

Virtualization in Enterprise

- **Separates product (OS services) from physical resources (server hardware)**
- **Live migration example:**



Outline

- Introduction
  - What, why?
- Basic techniques
  - Simulation
  - Binary translation
- Kinds of instructions
- Virtualization
  - x86 Virtualization
  - Paravirtualization
- Summary

Full-System Simulation (Simics 1998)

- Software simulates hardware components that make up a target machine
  - Interpreter executes each instruction & updates the software representation of the hardware state
- Approach is very accurate but very slow
- Great for OS development & debugging
  - "Break on triple fault" is better than real hardware suddenly rebooting
  - Possible to debug a driver for a hardware device that hasn't been built yet

System Emulation (Bochs, DOSBox, QEMU, fake86)

- Emulate just enough of hardware components to create an accurate "user experience"
- Typically CPU & memory are emulated
  - Buses are not
  - Devices communicate with CPU & memory directly
- Shortcuts are taken to achieve better performance
  - Reduces overall system accuracy
  - Code designed to run correctly on real hardware executes "pretty well"
  - Code not designed to run correctly on real hardware exhibits wildly divergent behavior

System Emulation Techniques

- Pure interpretation:
  - Interpret each guest instruction
  - Perform a semantically equivalent operation on host
- Static translation:
  - Translate each guest instruction to host instructions *once*
  - Example: DEC “mx” translator
    - Input: MIPS Ultrix executable
    - Output: Alpha OSF/1 executable
  - Limited applicability; self-modifying code doesn't work

System Emulation Techniques

- **Dynamic translation:**
  - Translate a block of guest instructions to host instructions just prior to execution of that block
  - Cache translated blocks for better performance
  - Like a Smalltalk/Java “JIT”
- **Dynamic recompilation & adaptive optimization:**
  - Discover which algorithm the guest code implements
  - Substitute with an optimized version on the host
  - Hard

Outline

- Introduction
  - What, why?
- Basic techniques
  - Simulation
  - Binary translation
- Kinds of instructions
- Virtualization
  - x86 Virtualization
  - Paravirtualization
- Summary

Kinds of Instructions

- "Regular"
  - ADD, XOR
  - Load, store
  - Branch, push, pop
- "Special"
  - CLI/STI, HLT, read/modify %cr3
- Devices (magic side-effects)
  - INB/OUTB
  - Stores into video RAM!
- How do we emulate?
  - "Regular", "Special" - just simulate the CPU
  - Devices – *very* difficult!
    - *Thousands* of devices exist, each one is extremely complex
    - A device emulator may be 100 lines of code, or 10,000

The Need for Speed

- "Slow" is easy
  - Simulation is naturally slow
  - Binary translation requires lots of "compilation"
- Key observation
  - "Run virtual X on physical X" should be faster than "run virtual X on physical Y"
  - "x86 on x86" should be faster than "x86 on PowerPC"
  - We don't need to *simulate* hardware if we can *use* it
    - "The best simulation of REP STOSB is REP STOSB"
- while(1):
  - Find a big block of "regular" instructions
  - Load up register values, jump to start of block
    - These instructions run at full speed
  - When something goes wrong, figure out a fix
    - This part is slow

Outline

- Introduction
  - What, why?
- Basic techniques
  - Simulation
  - Binary translation
- Kinds of instructions
- Virtualization
  - x86 Virtualization
  - Paravirtualization
- Summary

Full Virtualization

- IBM CP-40 (1967)
  - Supported 14 simultaneous S/360 virtual machines
- Later evolved into CP/CMS and VM/CMS (still in use)
  - 1,000 mainframe users, each with a private mainframe, running a text-based single-process "OS"
- Popek & Goldberg: *Formal Requirements for Virtualizable Third Generation Architectures* (1974)
  - Defines characteristics of a *Virtual Machine Monitor* (VMM)
  - Describes a set of architecture features sufficient to support virtualization

Virtual Machine Monitor

- **Equivalence:**
  - Provides an environment essentially identical with the original machine
- **Efficiency:**
  - Programs running under a VMM should exhibit only minor decreases in speed
- **Resource Control:**
  - VMM is in complete control of system resources

Process : Kernel :: VM : VMM

Popek & Goldberg Instruction Classification

- ***Sensitive* instructions**:
  - **Attempt to change configuration of system resources**
    - **Disable interrupts**
    - **Change count-down timer value**
    - **...**
  - **Illustrate different behaviors depending on system configuration**
- ***Privileged* instructions**:
  - **Trap if the processor is in user mode**
  - **Do not trap in supervisor mode**

Popek & Goldberg Theorem

"... 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."

- **Each instruction must either:**
  - **Exhibit the same result in user and supervisor modes**
  - **Else trap if executed in user mode**
- **Then a VMM can *run a guest kernel in user mode!***
  - **Sensitive instructions are trapped, handled by VMM**
- **Architectures that meet this requirement:**
  - **IBM S/370, Motorola 68010+, PowerPC, others.**

x86 Virtualization

- x86 ISA (pre-2005) does not meet the Popek & Goldberg requirements for virtualization!
- ISA contains 17+ sensitive, unprivileged instructions:
  - `SGDT, SIDT, SLDT, SMSW, PUSHF, POPF, LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT, RET, STR, MOV`
  - Most simply reveal that the "kernel" is running in user mode
    - PUSHF
    - PUSH %CS
  - Some *execute inaccurately*
    - POPF
- Virtualization is still possible, requires workarounds

The "POPF Problem"

```
PUSHF                        # %EFLAGS onto stack
ANDL $0x003FFDFF, (%ESP)     # Clear IF on stack
POPF                         # %EFLAGS from stack
```

- If run in supervisor mode, interrupts are now off
- What "should" happen if this is run in user mode?

The "POPF Problem"

```
PUSHF                       # %EFLAGS onto stack
ANDL $0x003FFDFF, (%ESP) # Clear IF on stack
POPF                        # %EFLAGS from stack
```

- **If run in supervisor mode, interrupts are now off**
- **What "should" happen if this is run in user mode?**
  - **Attempting a privileged operation should trap to VMM**
  - **If it doesn't trap, the VMM can't simulate it**
    - **Because the VMM won't even know it happened**
- **What happens on the x86?**

The "POPF Problem"

```
PUSHF                          # %EFLAGS onto stack
ANDL $0x003FFDFF, (%ESP) # Clear IF on stack
POPF                           # %EFLAGS from stack
```

- If run in supervisor mode, interrupts are now off
- What "should" happen if this is run in user mode?
  - Attempting a privileged operation should trap to VMM
  - If it doesn't trap, the VMM can't simulate it
    - Because the VMM won't even know it happened
- What happens on the x86?
  - CPU "helpfully" ***ignores changes to privileged bits*** when POPF runs in user mode!
  - So that sequence does ***nothing***, no trap, VMM can't simulate

VMware (1998)

- Runs guest operating system in ring 3
  - Maintains the illusion of running the guest in ring 0
- *Insensitive* instruction sequences run by CPU at full speed:
  - `movl 8(%ebp), %ecx`
  - `addl %ecx, %eax`
- *Privileged* instructions trap to the VMM:
  - `cli`
- *Sensitive, unprivileged* instructions handled by *binary translation*:
  - `popf` ⇒ `int $99`

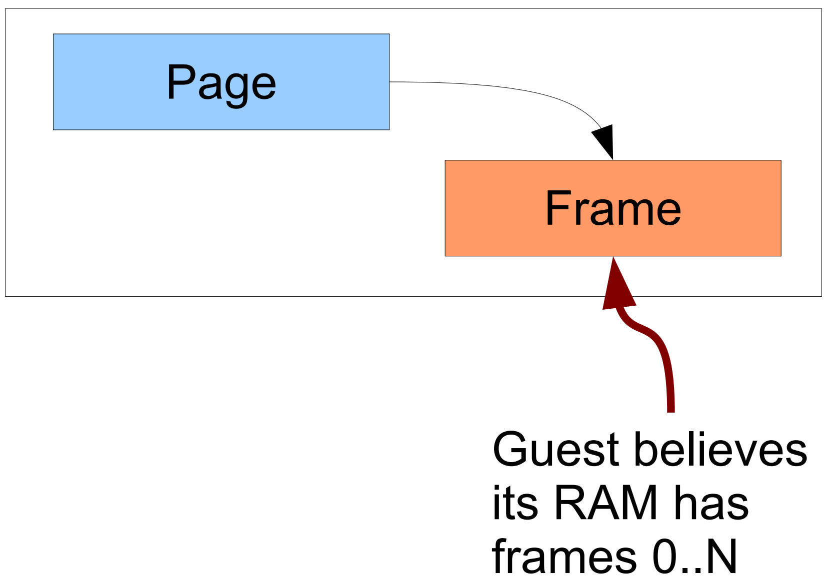
Virtual Memory

- We've virtualized instruction execution
  - How about other resources?
- Kernels use physical memory to implement virtual memory
  - How do we virtualize physical memory?
    - Each guest kernel must be protected from the others, so we can't let them access physical memory
    - Ok, use virtual memory (obvious so far, isn't it?)

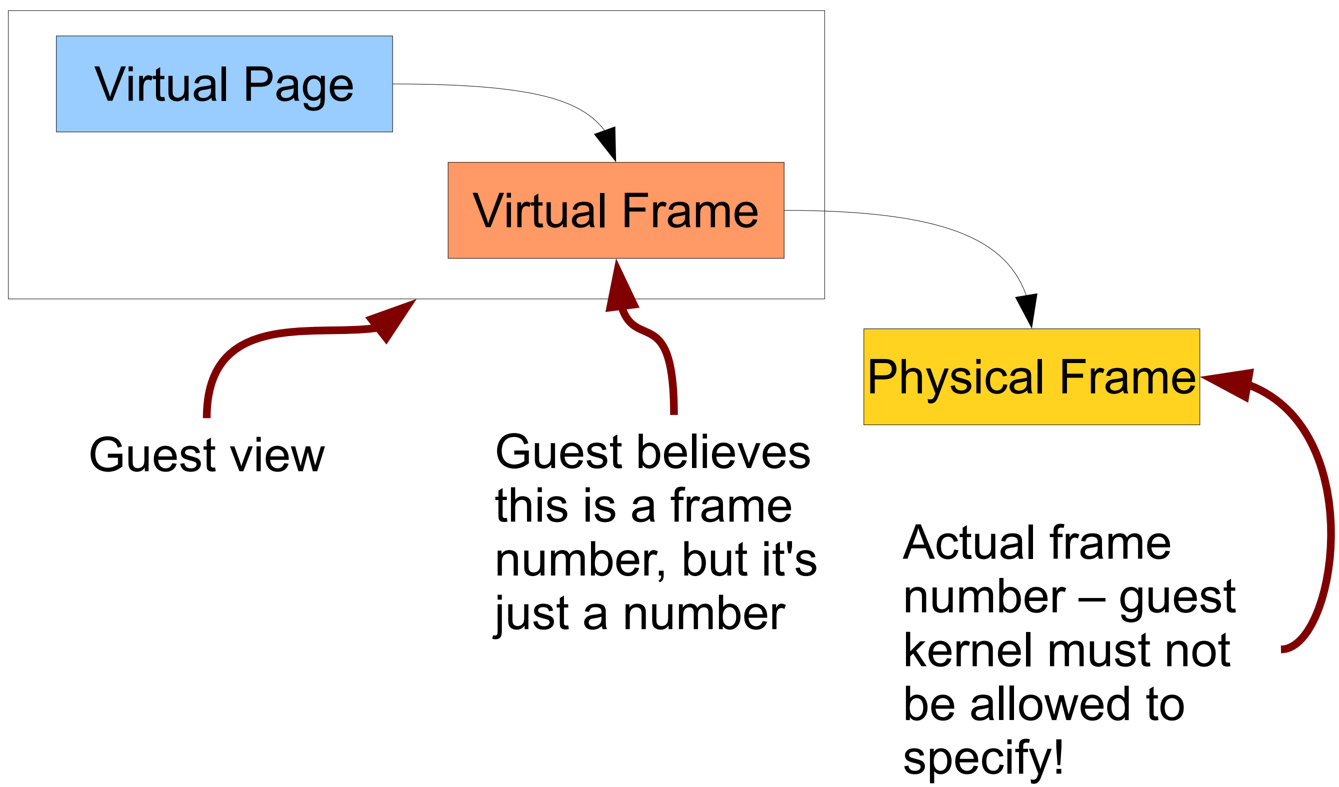
Virtual Memory

- We've virtualized instruction execution
  - How about other resources?
- Kernels use physical memory to implement virtual memory
  - How do we virtualize physical memory?
    - Each guest kernel must be protected from the others, so we can't let them access physical memory
    - Ok, use virtual memory (obvious so far, isn't it?)
  - But guest kernels themselves provide virtual memory to their processes
    - They like to "MOVL %EAX, %CR3"
    - We can't allow them to do that!
    - Can we simulate it??

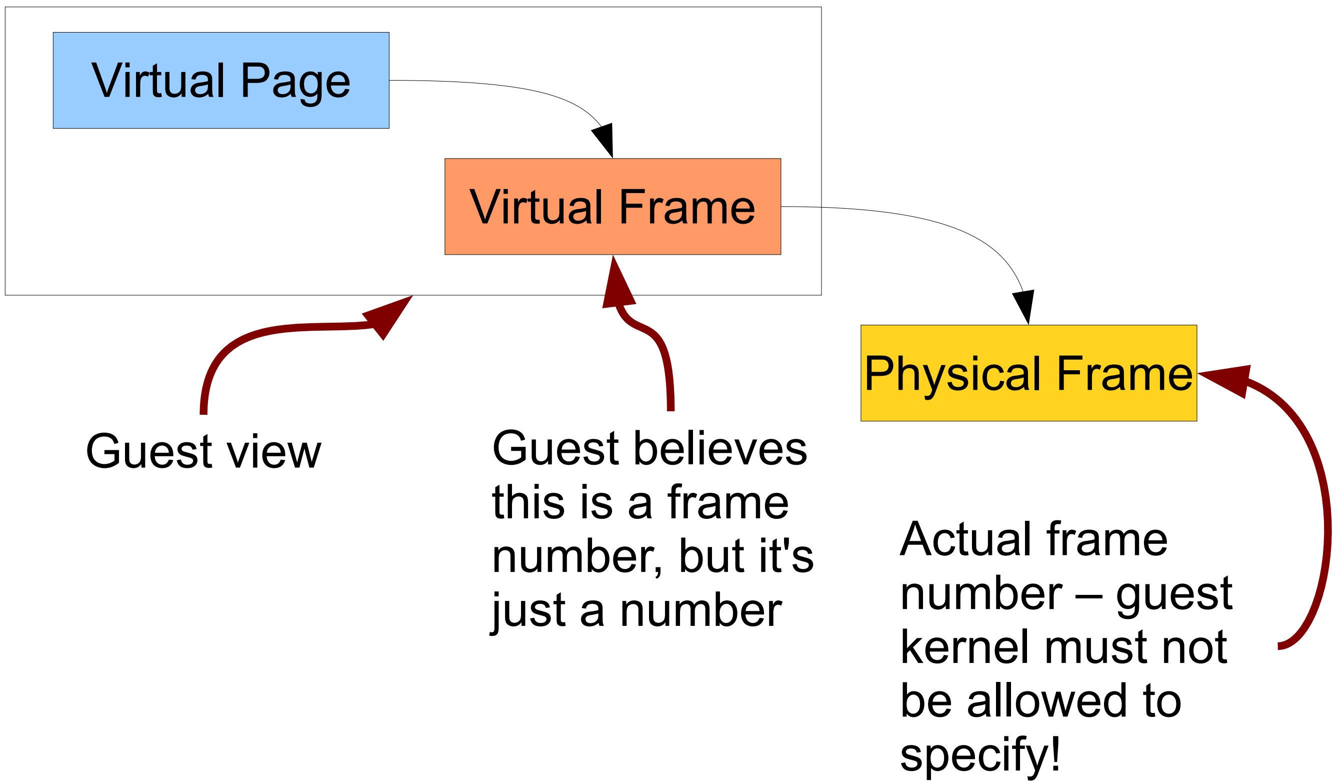
VM – Guest-kernel view



VM – Fiction vs. Reality

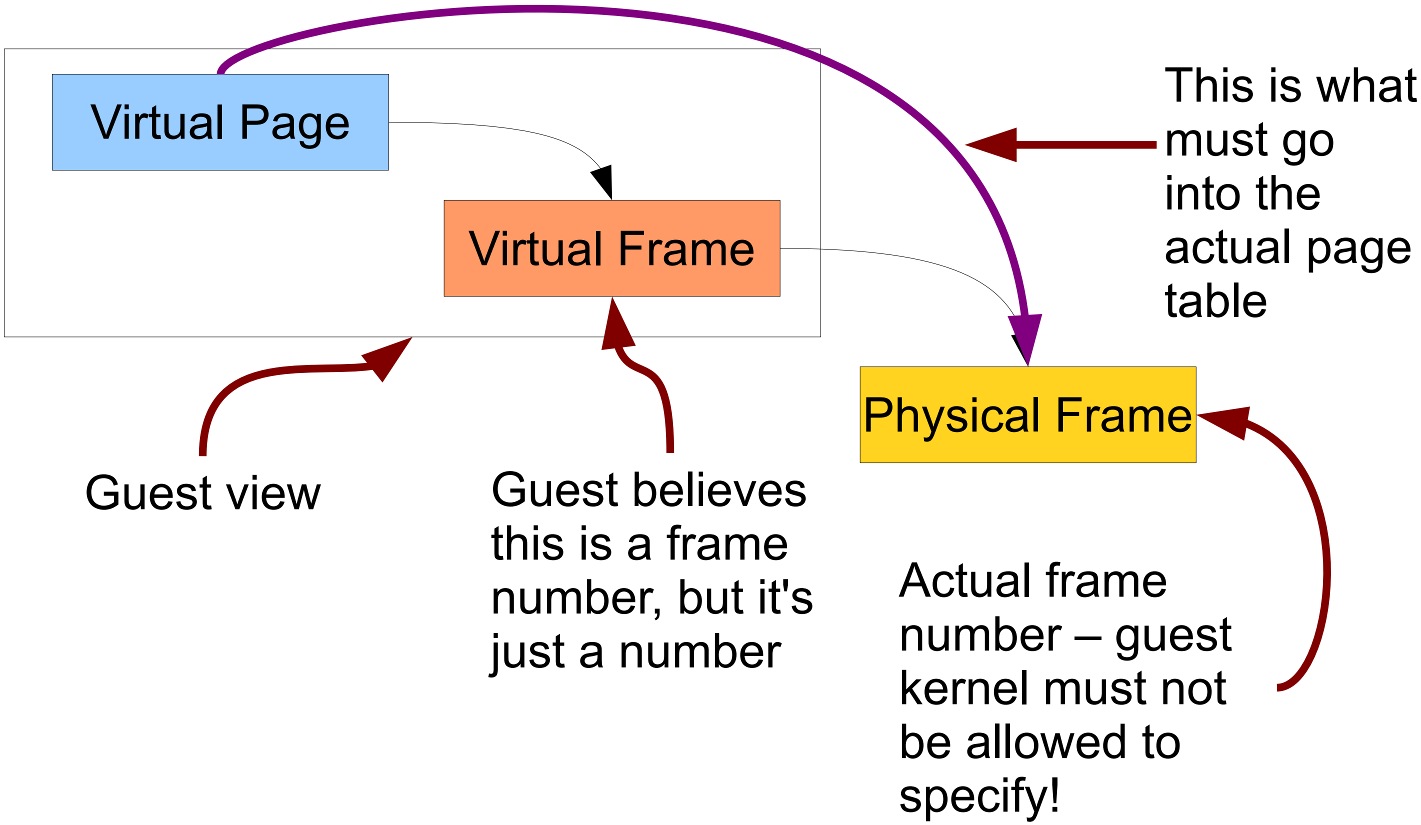


VM – How to do it?

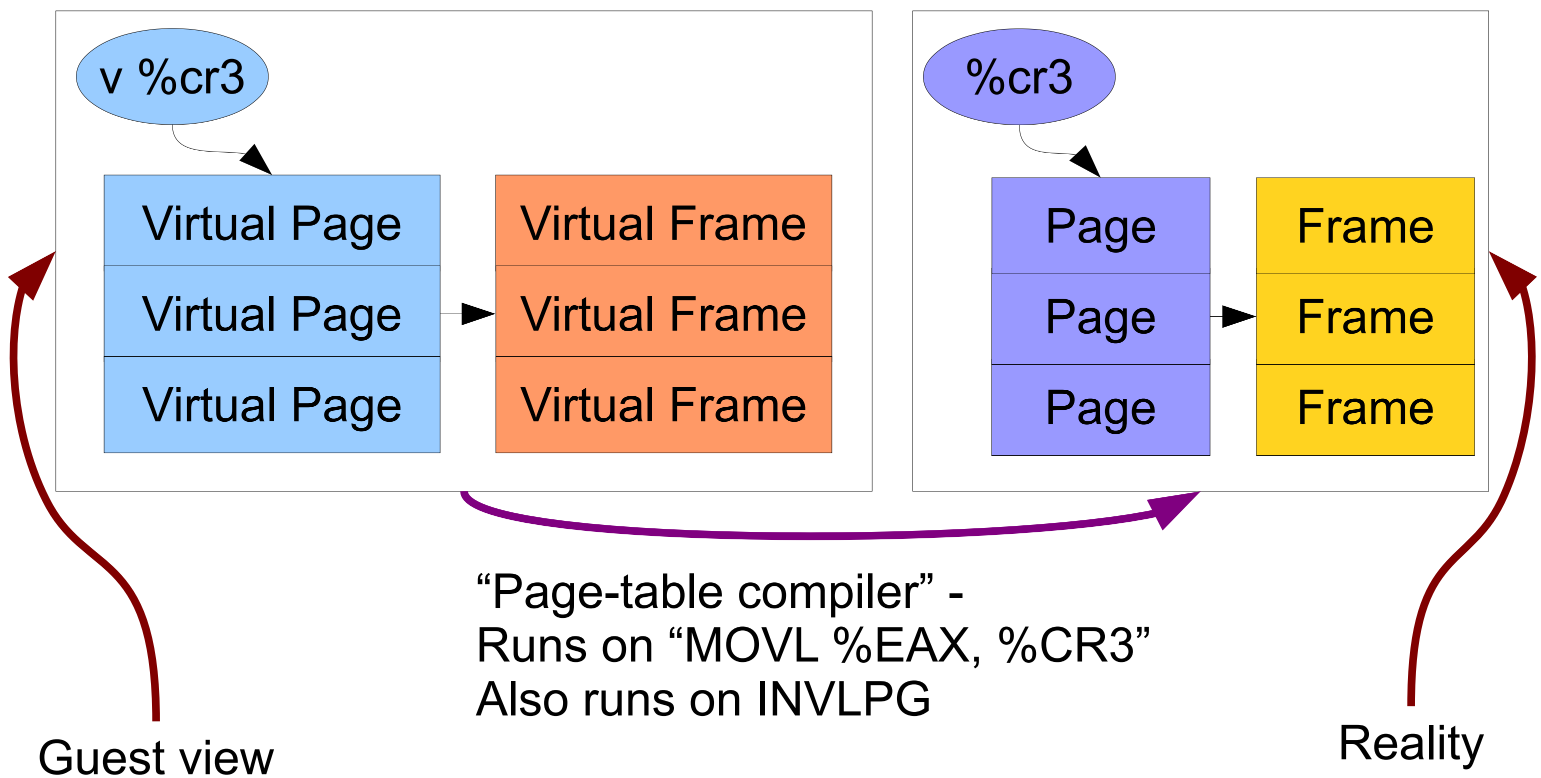


Note: traditional x86 VM hardware does not implement “map, then map again’

VM – How to do it?



VM – Shadow Page Tables



Shadow Page Tables

- **Accesses to %cr3 are trapped by hardware**
  - **Store into %cr3?**
    - **“Compile” guest-kernel page table into real page table**
      - **Map guest frame numbers into actual frame numbers**
    - **Secretly set %cr3 to point to real page table**
  - **Fetch from %cr3?**
    - **Return the guest-kernel “physical” address of the virtual page table in guest-kernel virtual memory, not the physical address of the actual page table in physical memory**

Shadow Page Tables

- **Accesses to %cr3 are trapped by hardware**
  - **Store into %cr3?**
    - **"Compile" guest-kernel page table into real page table**
      - **Map guest frame numbers into actual frame numbers**
    - **Secretly set %cr3 to point to real page table**
  - **Fetch from %cr3?**
    - **Return the guest-kernel "physical" address of the virtual page table in guest-kernel virtual memory, not the physical address of the actual page table in physical memory**
- **Accesses to guest-kernel page tables are special too!**
  - **It's ok for the guest kernel to examine its fake page table**
  - **But if guest *stores* into a fake PTE, we must re-compile**
  - **So virtual page tables are read-only pages for the guest**

Shadow Page Tables

- Accesses to %cr3 are trapped by hardware
  - Store into %cr3?
    - "Compile" guest-kernel page table into real page table
      - Map guest frame numbers into actual frame numbers
    - Secretly set %cr3 to point to real page table
  - Fetch from %cr3?
    - Return the guest-kernel "physical" address of the virtual page table in guest-kernel virtual memory, not the physical address of the actual page table in physical memory
- Accesses to guest-kernel page tables are special too!
  - It's ok for the guest kernel to examine its fake page table
  - But if guest *stores* into a fake PTE, we must re-compile
  - So virtual page tables are read-only pages for the guest
- Guest kernel sets some pages to "kernel only"
  - Each guest page table compiles to *two* real page tables
    - guest-kernel-mode has all pages, guest-user-mode doesn't

Wow, This is Hard!

- Many tricks played to improve performance
  - Compiling page-tables is slow, so cache old compilations
  - When to garbage-collect them?
- PTE's contain dirty & accessed bits
  - Won't cover that today

- Is there an easier way??

Wow, This is Hard!

- Many tricks played to improve performance
  - Compiling page-tables is slow, so cache old compilations
  - When to garbage-collect them?
- PTE's contain dirty & accessed bits
  - Won't cover that today

- Is there an easier way??
  1. Fix the hardware
  2. Blur the hardware ("paravirtualization")

Hardware Assisted Virtualization

- Modern x86's *do* meet Popek & Goldberg requirements
  - Intel VT-x (2005), AMD-V (2006)
- VT-x introduces two new operating modes:
  - "VMX root" operation & "VMX non-root" operation
  - VMM runs in VMX root, guest OS runs in non-root
    - Both modes support all privilege rings
  - Guest OS runs in (non-root) ring 0
    - VMM tells hardware "Enter guest mode, but trap on these conditions: ..."
    - If guest kernel runs a sensitive instruction, hardware does a "VM exit" back to VMM, indicates why
- 2nd-generation VT-x has "EPT": hardware fix for VM
  - Host sets up page tables giving "virtual physical pages" to guest
  - Guest page tables map "virtual virtual pages" to them

Paravirtualization (Denali 2002, Xen 2003)

- Motivation
  - Binary translation and shadow page tables are hard
- First observation:
  - If OS is open-source, it can be modified at the source level to make virtualization explicit (not transparent), and easier
    - Replace “MOVL %EAX, %CR3” with “install_page_table()”
    - Typically only a small fraction of the guest kernel needs to be edited
    - Guest *user* code is not changed at all
- Paravirtualizing VMMs (hypervisors) virtualize only a subset of the x86 execution environment
  - Run guest kernels in rings 1-3
    - No illusion about running in a virtual environment
    - Guest kernels may not use sensitive, unprivileged instructions and expect a privileged result

Paravirtualization (Denali 2002, Xen 2003)

- **Second observation:**
  - **Regular VMMs must emulate hardware for devices**
    - **Disk, Ethernet, etc**
    - **Performance is poor due to constrained device API**
      - **To “send packet”, must emulate many device-register accesses (inb/outb or MMIO, interrupt enable/disable)**
      - **Each step results in a trap**
  - **Already modifying guest kernel, why not provide virtual device drivers?**
    - **Virtual Ethernet could export send_packet(addr, len)**
      - **This requires only one trap**
- **“Hypercall” interface:**

syscall : kernel :: hypercall : hypervisor

Outline

- Introduction
  - What, why?
- Basic techniques
  - Simulation
  - Binary translation
- Kinds of instructions
- Virtualization
  - x86 Virtualization
  - Paravirtualization
- Summary

Are We Having Fun Yet?

- Virtualization is great if you need it
  - If you must have 35 /etc/passwd's, 35 sets of users, 35 Ethernet cards, etc.
  - There are many techniques, which work (are secure and fast enough)
- Virtualization is overkill if we need only isolation
  - Remember the Java "virtual machine"??
    - Secure isolation for multiple applications
    - Old approach – Smalltalk (1980)
    - New approach – Google App Engine, Heroku, etc.
- Open question
  - How ***best*** to get isolation, machine independence?

Summary

- What virtualization does
  - Multiple OS's on one laptop
  - Debugging, security analysis
  - Enterprise
    - Efficiency
    - Reliability (outage resistance)
- The problem
  - Kinds of instructions
- Solutions
  - Binary translation (useful for light-weight uses)
  - {Full, hardware assisted, para-}virtualization
- Many things not covered today!
  - "I/O virtualization" - attaching real devices to virtual machines
  - ...

Further Reading

- Gerald J. Popek and Robert P. Goldberg.
  Formal requirements for virtualizable third generation architectures.
  Communications of the ACM, 17(7):412-421, July 1974.

- John Scott Robin and Cynthia E. Irvine.
  Analysis of the Intel Pentium's ability to support a secure virtual machine monitor.
  In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

- Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig.
  Intel Virtualization Technology: Hardware support for efficient processor virtualization.
  Intel Technology Journal, 10(3):167-177, August 2006.

- Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield.
  Xen and the Art of Virtualization.
  In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164-177, Bolton Landing, NY, October 2003.

- Yaozu Dong, Shaofan Li, Asit Mallick, Jun Nakajima, Kun Tian, Xuefei Xu, Fred Yang, and Wilfred Yu. Extending Xen with Intel Virtualization Technology.
  Intel Technology Journal, 10(3):193-203, August 2006.

- Stephen Soltesz, Herbert Potzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson.
  Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors.
  In *Proceedings of the 2007 EuroSys conference*, Lisbon, Portugal, March 2007.

- Fabrice Bellard.
  QEMU, a fast and portable dynamic translator.
  In *Proceedings of the 2005 USENIX Annual Technical Conference*, Anaheim, CA, April 2005.