Virtualization

Dave Eckhardt
Dave O'Hallaron

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

Nov, 2016

What is Virtualization?

- **Virtualization is a technique for abstracting a resource by interposing on accesses to that resource.**

- **By interposing on accesses to some physical resource, the system presents users with a virtual (or logical) resource.**

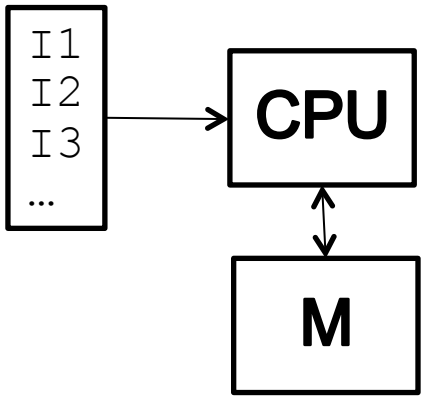
- **Lots of examples in systems…**

Example: x86 ISA

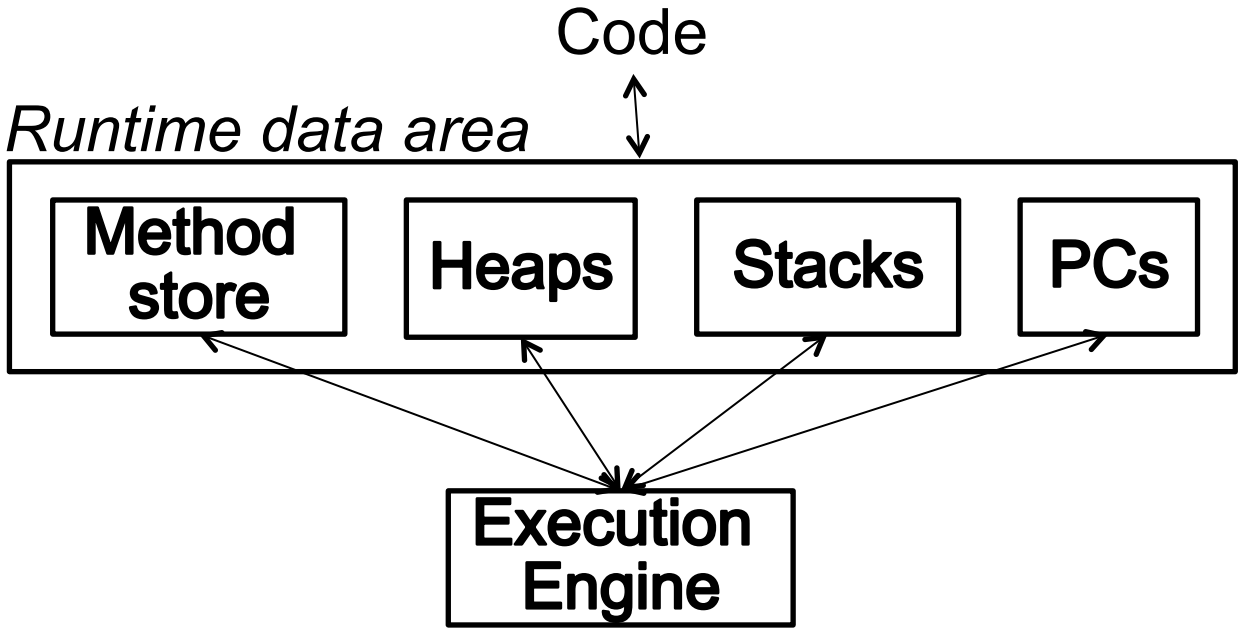
Physical resource	*Interposed by*	*Virtual resource*
x86 uops Functional units Tagged registers TLB Cache memories	Hardware interpretor	x86 ISA GP registers Memory

Example: Java VM

Physical resource	*Interposed by*	*Virtual resource*
Physical machine	**Byte code interpreter/ JIT compiler**	**Java virtual machine**

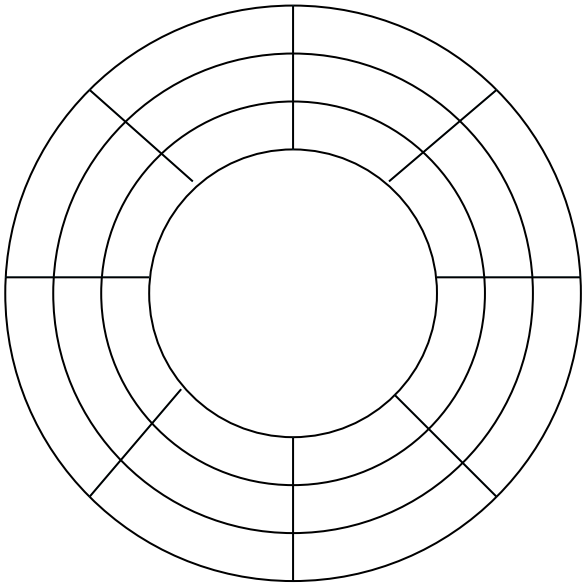


Physical machine



Example: Disk Geometry

Physical resource	*Interposed by*	*Virtual resource*
Sectors, tracks, surfaces, and cylinders	**Disk controller**	**Logical blocks**



LBN 0 LBN 1 LBN 2 ...

Example: Files

Physical resource	*Interposed by*	*Virtual resource*
Partitions **Logical blocks**	**Kernel**	**Files**

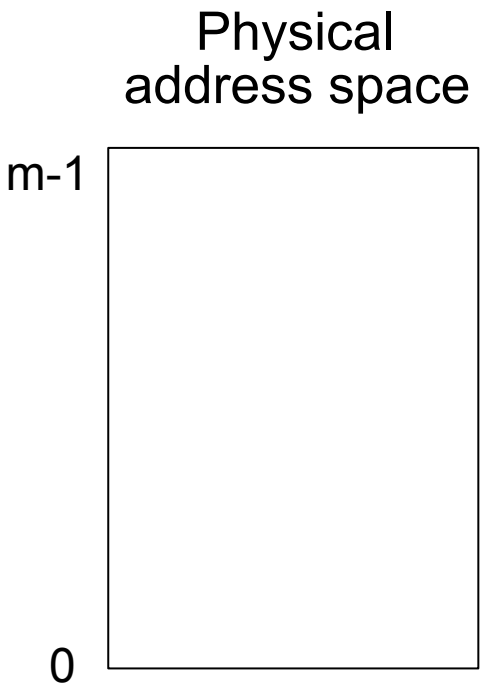
LBN 0 LBN 1 ...

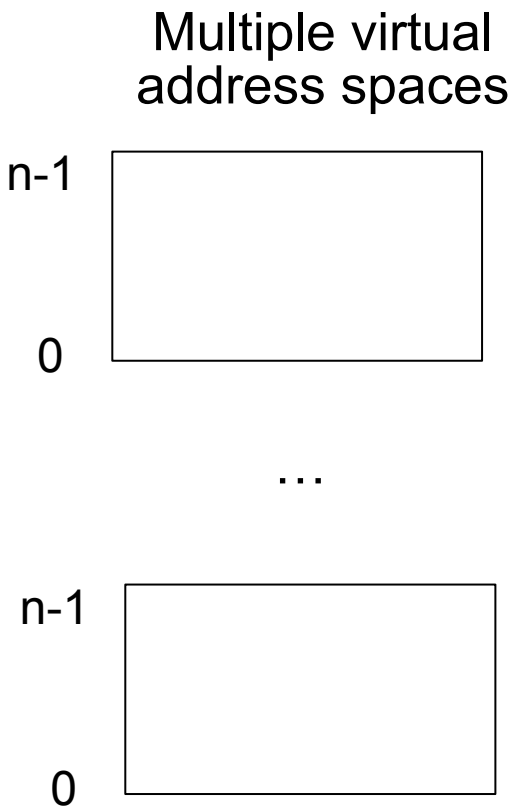
File f $f_0, f_1, \ldots, f_{n-1}$

File g $g_0, g_1, \ldots, g_{m-1}$

Example: Virtual Memory

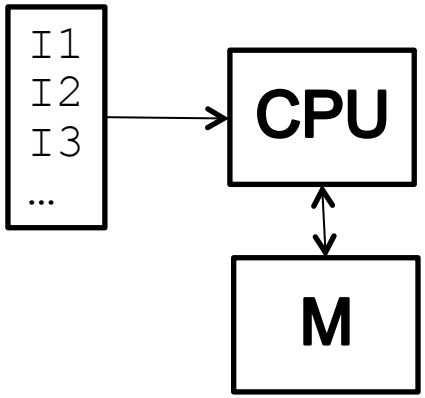
Physical resource	*Interposed by*	*Virtual resource*
DRAM	**MMU/kernel**	**Virtual memory**



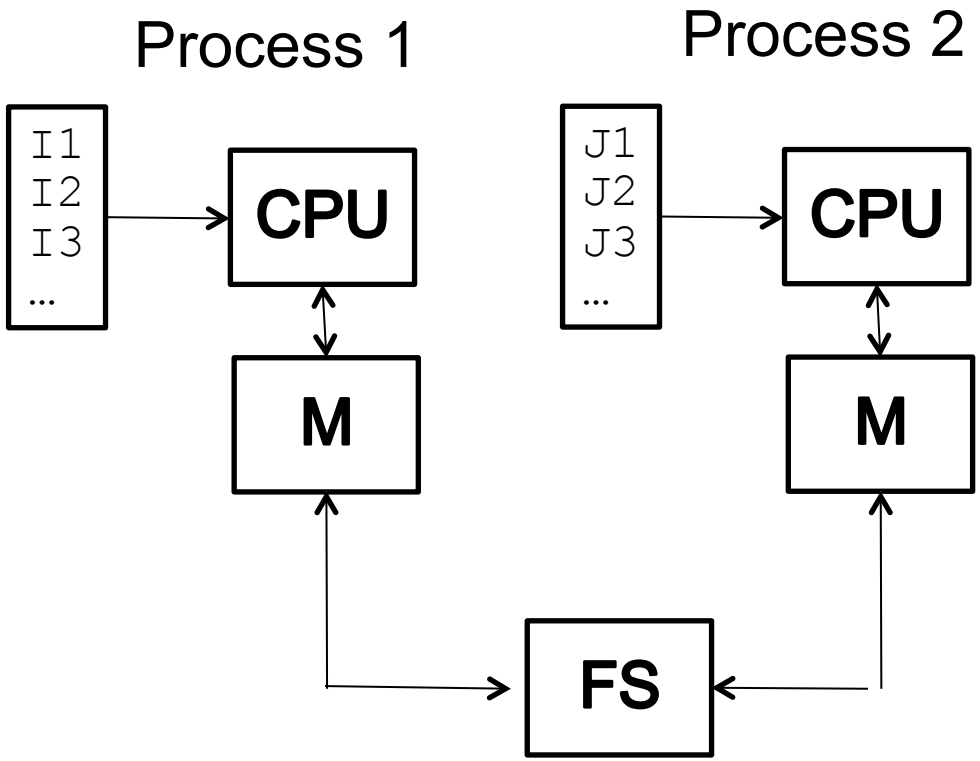


Example: Partial Machine (Process)

Physical resource	*Interposed by*	*Virtual resource*
Physical machine	**Timer, Kernel, and MMU**	**Process**



Physical machine



Advantages of the Process Abstraction

- **Program is provided with the illusion that it has**
  - **exclusive use of the CPU, and**
  - **its own private address space protected from other processes**

- **Good memory isolation, but…**

Disadvantages of Process Abstraction

- Processes share the file system
  - Difficult to simultaneously use different versions of:
    - Programs, libraries, configurations
- Single machine owner:
  - root is *the* superuser
  - Any process that attains superuser privileges controls all processes
- Processes share the same kernel
  - Hard to run different OSs on the same machine
- Processes have limited degree of protection, even from each other:
  - Processes can have performance impact on each other
  - Linux “OOM killer” can kill one process if another uses lots of memory
- Overall, processes aren't *that* isolated from each other...

Isolating Processes (classic)

- `chroot()`
  - Venerable Unix system call
  - Sets the process's new root directory
  - Only a minimal sandbox, still sharing:
    - /proc and /sys
    - Resources: I/O bandwidth, CPU time, memory, disk space, …

- `getrlimit()/setrlimit()/ulimit()`
  - get and set resource limits for a process.
  - per process only.

Isolating Processes Further (Linux)

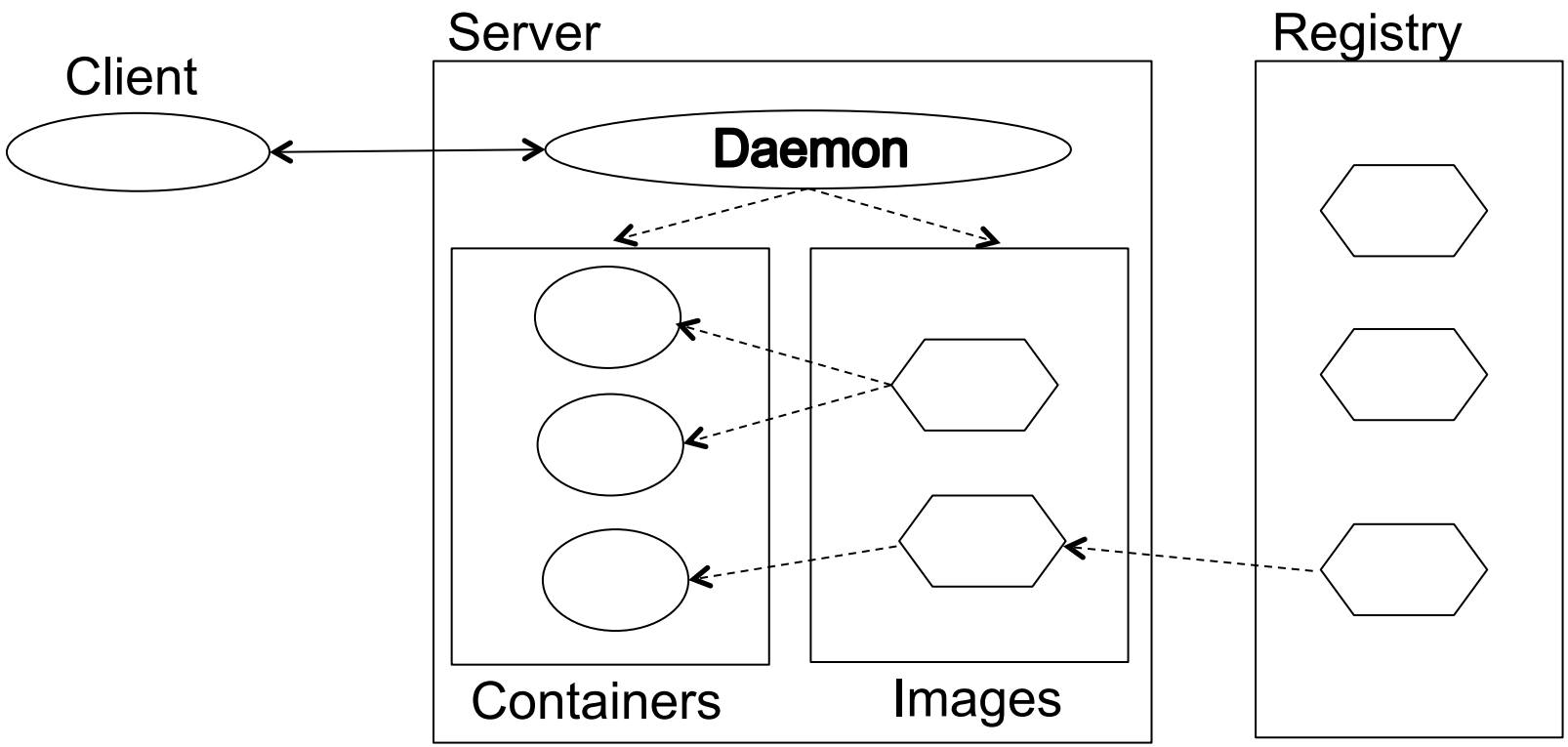
- **namespaces (via `clone()`): Control visible resources:**
  - **Process: Each task gets its own task hierarchy**
  - **Mount: Each task gets a separate file hierarchy**
  - **Network: Each task can see separate network devices**
  - **User: Task can have root access within namespace.**

- **cgroups (via `/sys/fs/cgroup`): Control resource usage:**
  - **Relative CPU usage**
  - **Memory usage**
  - **Disk I/O**

Containers

***A container* is a group of related isolated processes.**

- **Use kernel features to voluntarily limit resource visibility and usage**
- **Stronger isolation than process abstraction**
  - **However, all containers still share the same kernel**
- **Examples: Docker, Kubernetes**



Advantages and Disadvantages of Container Abstraction

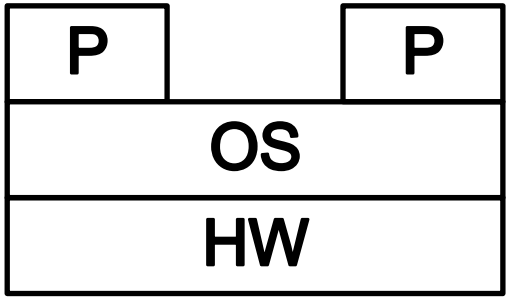
- **Advantages**
  - **Isolation level is often good enough**
  - **Lightweight and relatively simple**

- **Disadvantages**
  - **Isolation level is sometimes not good enough**
  - **Can’t be migrated from one machine to another**
  - **Precludes running multiple different kernels**

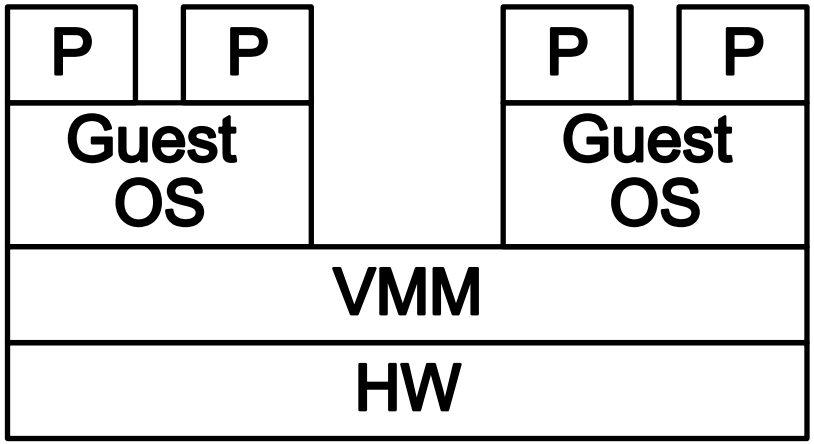
- **Stronger isolation requires full-machine virtualization**
  - **Resource being interposed on is the entire machine state.**

Example: Full-Machine Virtualization

Physical resource	*Interposed by*	*Virtual resource*
Physical machine	**Virtual-machine Monitor (VMM)/ Hypervisor**	**Virtual Machine**

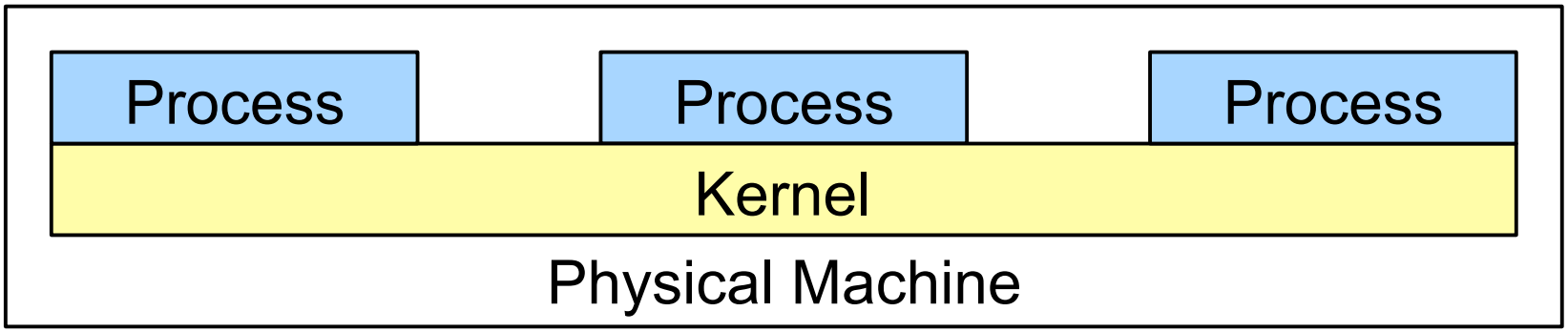


Physical machine

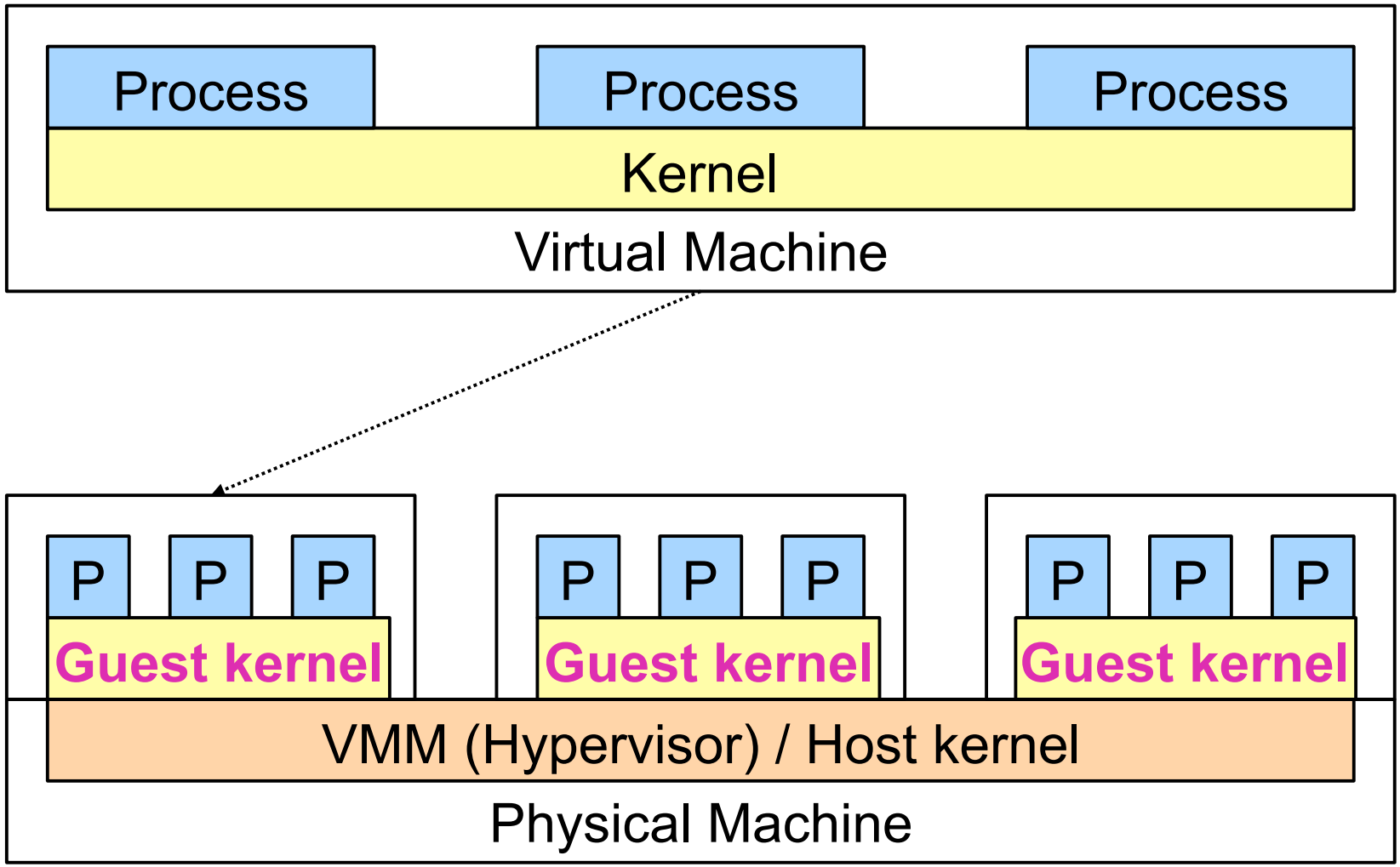


Virtual machines

Process/Kernel Stack



Virtualization Stack



Why Use Virtual Machines?

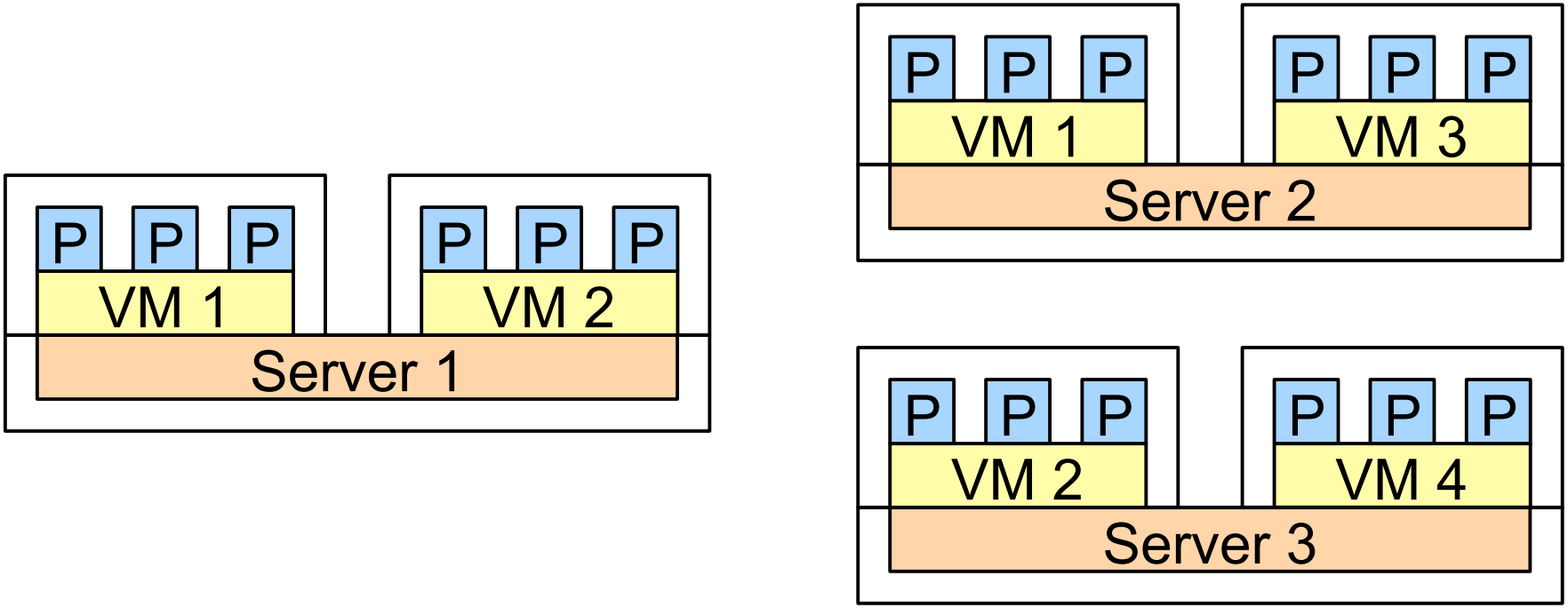
- **Run two operating systems on the same machine!**
  - **“Windows+Linux” was VMware's first business model**
  - **Hobbyists like to run ancient-history OS's**
- **Debugging OS's is more pleasant**
  - **Also: instrumenting what an OS does**
  - **Monitoring a captive OS for security infestations**

Why Use Virtual Machines?

- *Huge* impact on enterprise hosting and cloud services
  - 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:**



Disadvantages of Virtual Machines

- **Feasibility issues**
  - **Hardware support? OS support?**
  - **Admin support?**
  - **Popularity of virtualization platforms argues these can be handled**
- **Performance issues**
  - **Is a 5-20% performance hit tolerable?**
  - **Can your NIC or DRAM or disk keep up with the load of multiple virtual machines?**
- **"Nothing fails like success"**
  - **VMMs are getting larger, and potentially home to security bugs**

Full Machine 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.
  - But not legacy x86.

Legacy 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`

VMware (1998)

Privileged instructions trap to the VMM:

```
cli
```

actually results in General Protection Fault (IDT entry #13), handled:

```
void gpf_exception(int vm_num, regs_t *regs)
{
    switch (vmm_get_faulting_opcode(regs->eip))
    {
        ...
        case OP_CLI:
            /* VM doesn't want interrupts now */
            vmm_defer_interrupts(vm_num);
            break;
        ...
    }
}
```

VMware (1998)

We wish `popf` trapped, but it doesn't.

Scan “code pages” of executable, translating

`popf` ⇒ `int $99`

which gets handled:

```
void popf_handler(int vm_num, regs_t *regs) {
    unsigned int oldef = regs->eflags;
    unsigned int newef = *(regs->esp);
    if (!vm->pl0 && (newef & EFLAGS_SENSITIVE))
        gpf_handler(...);
    regs->eflags = newef;
    regs->esp++;
    if (!(oldef&EFLAGS_IF) && (newef&EFLAGS_IF)
        deliver_pending_interrupts(vm);
    ...
}
```

Modern x86 Virtualization

- **Modern x86's after 2005 *do* meet Popek & Goldberg requirements**
  - **Intel VT-x (2005, 2008), AMD-V (2006)**

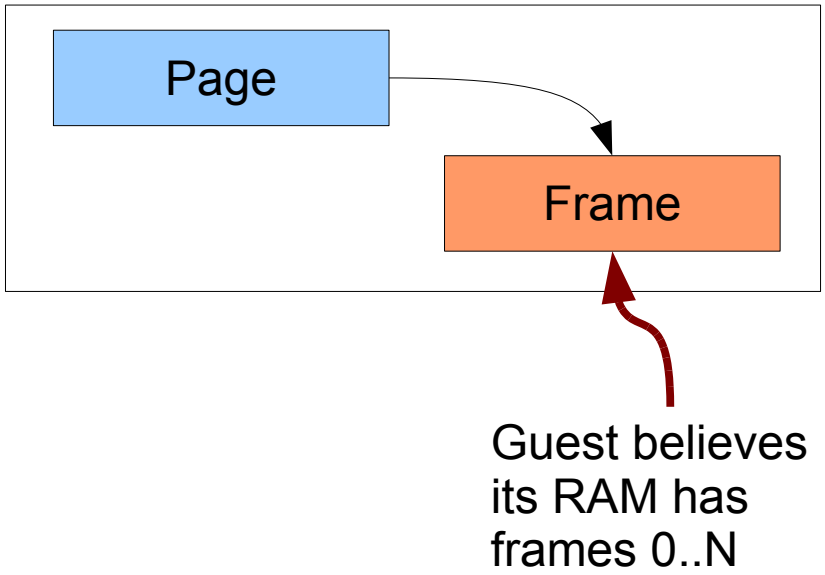
- **VT-x introduces two new operating modes:**
  - **"VMX root" operation & "VMX non-root" operation**
  - **VMM runs in VMX root, guest OS runs in non-root**
    - **Both modes support all privilege rings**

- **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**

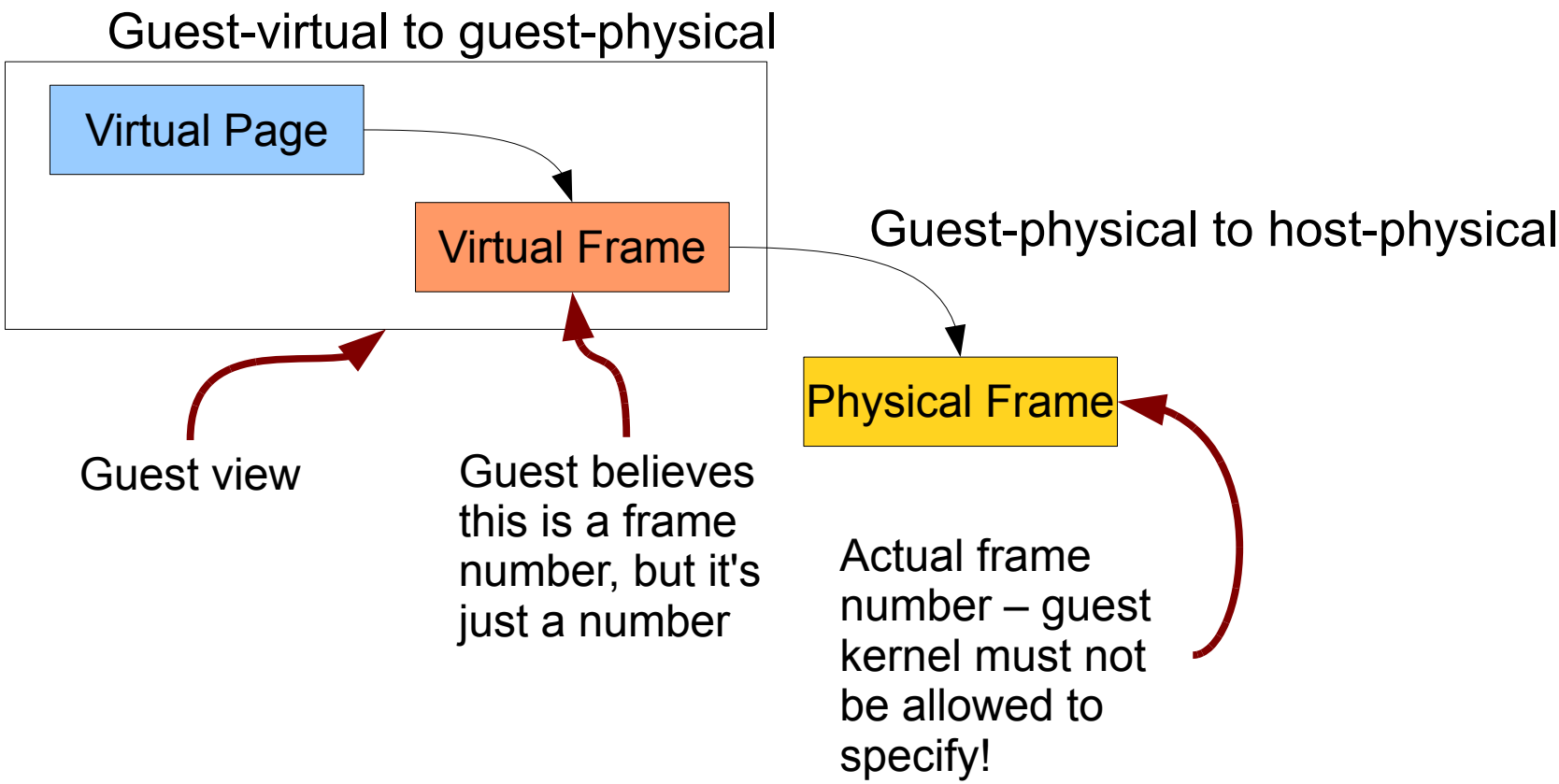
Virtualizing Memory

- We've virtualized instruction execution
  - How about other resources?
- Kernels use physical memory to implement virtual memory
- How do we virtualize physical memory?
  - As always, a level of indirection can help.

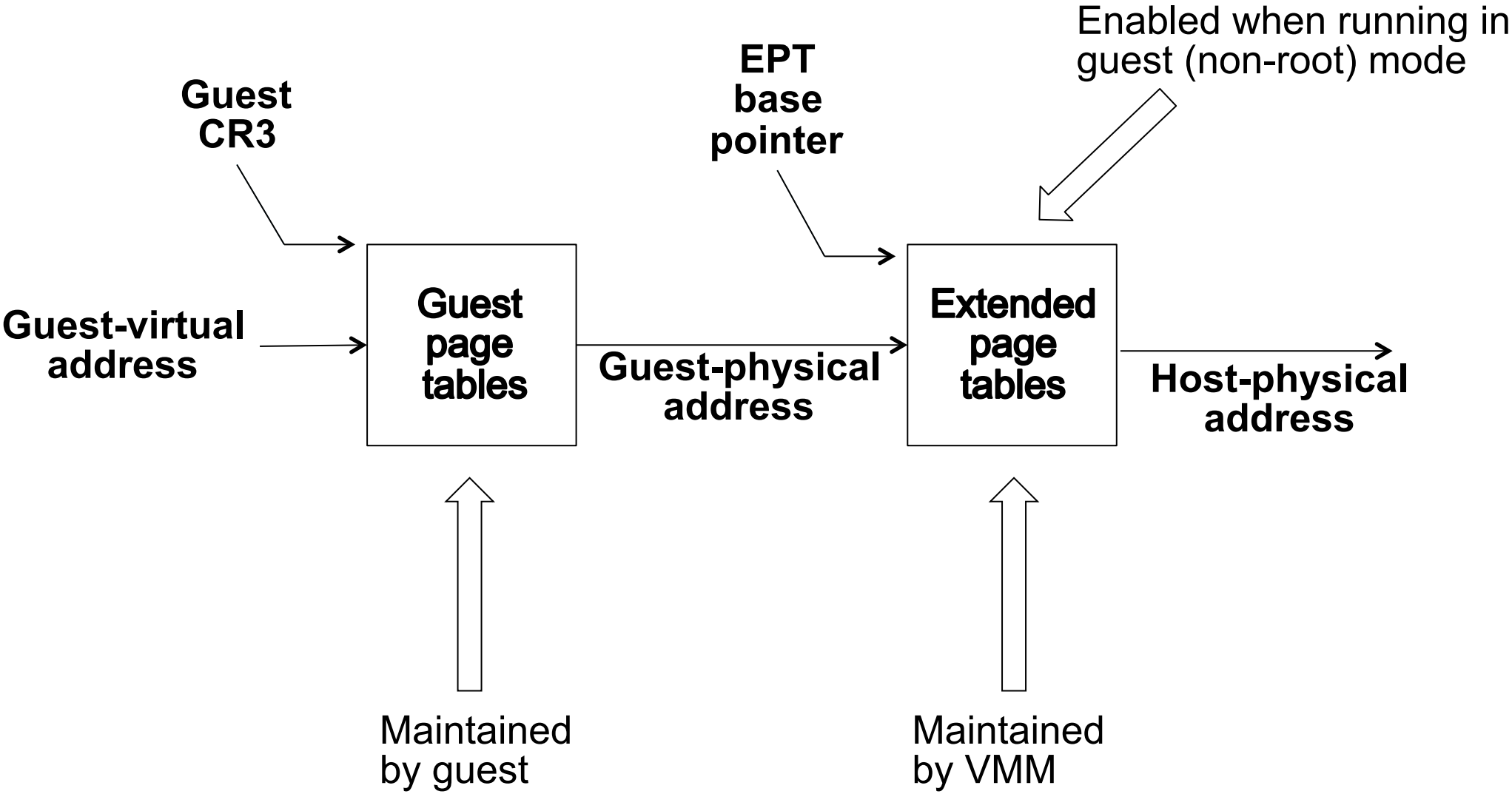
VM – Guest-kernel view



VM – Fiction vs. Reality



x86 Nested Page Tables



Summary

- Virtualization is a general technique for providing abstract views of resources
- Full machine virtualization virtualizes all of the state in the machine
  - Isolation from other processes
  - Debugging, security analysis
  - Enterprise
    - Efficiency
    - Reliability (outage resistance)
- Many 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.

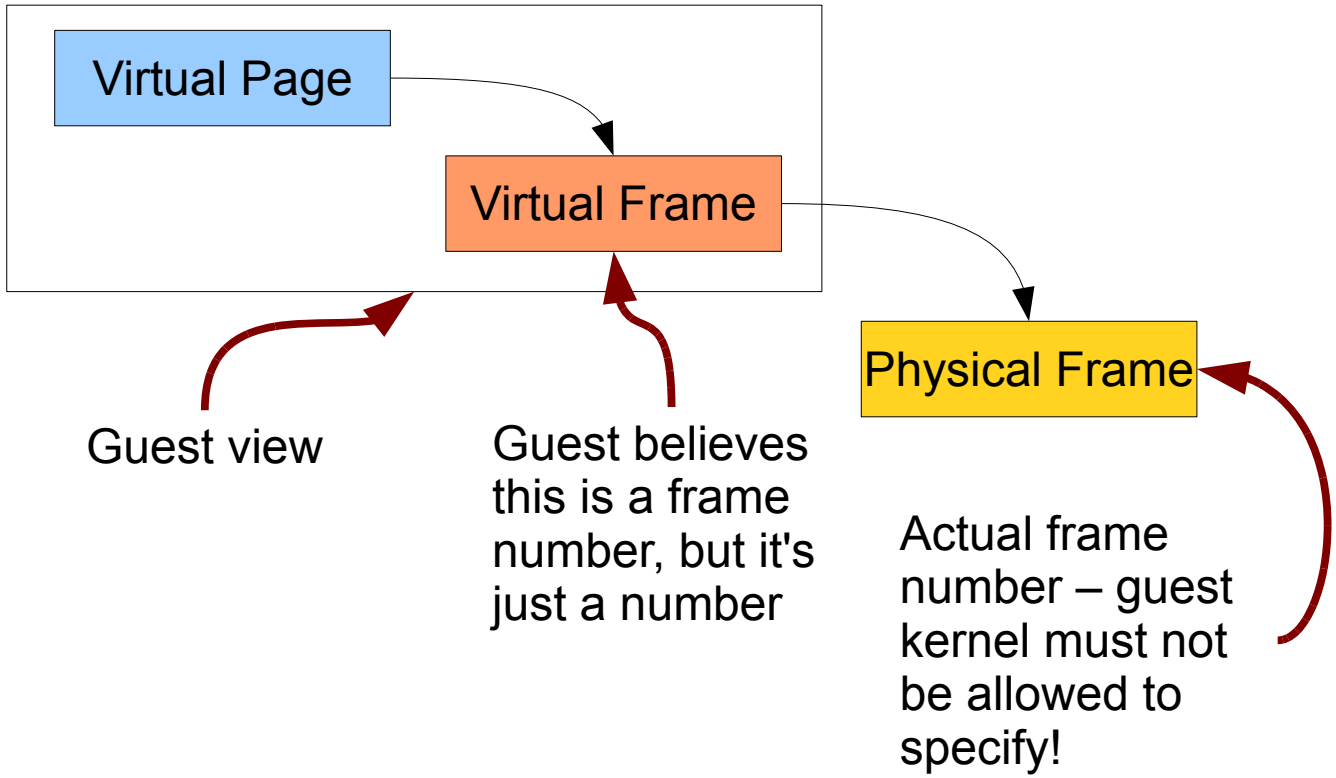
- Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield.
  Xen and the Art of Virtualization.
  In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164-177, Bolton Landing, NY, October 2003.

- 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.

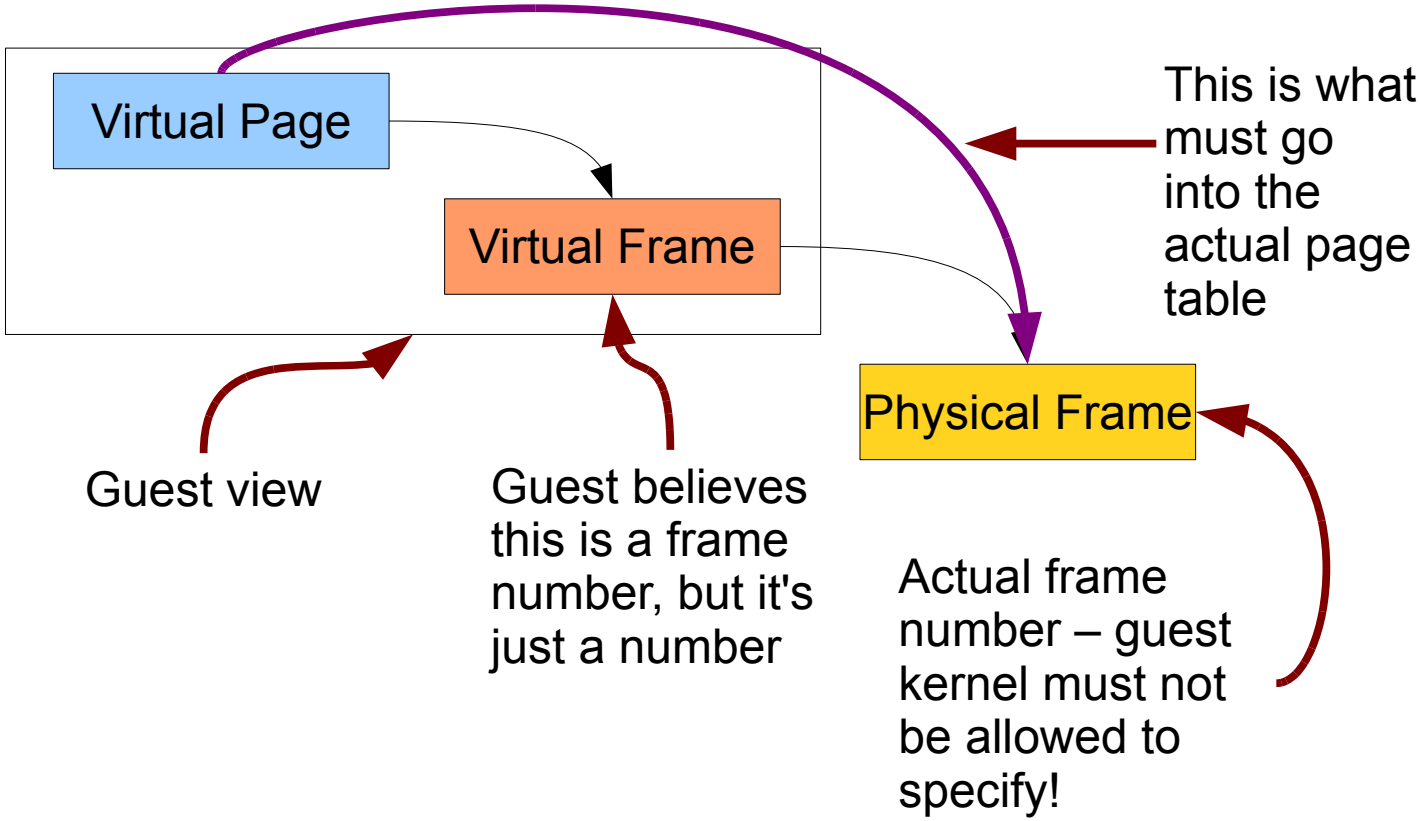
- Fabrice Bellard.
  QEMU, a fast and portable dynamic translator.
  In *Proceedings of the 2005 USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

VM – How to do it?

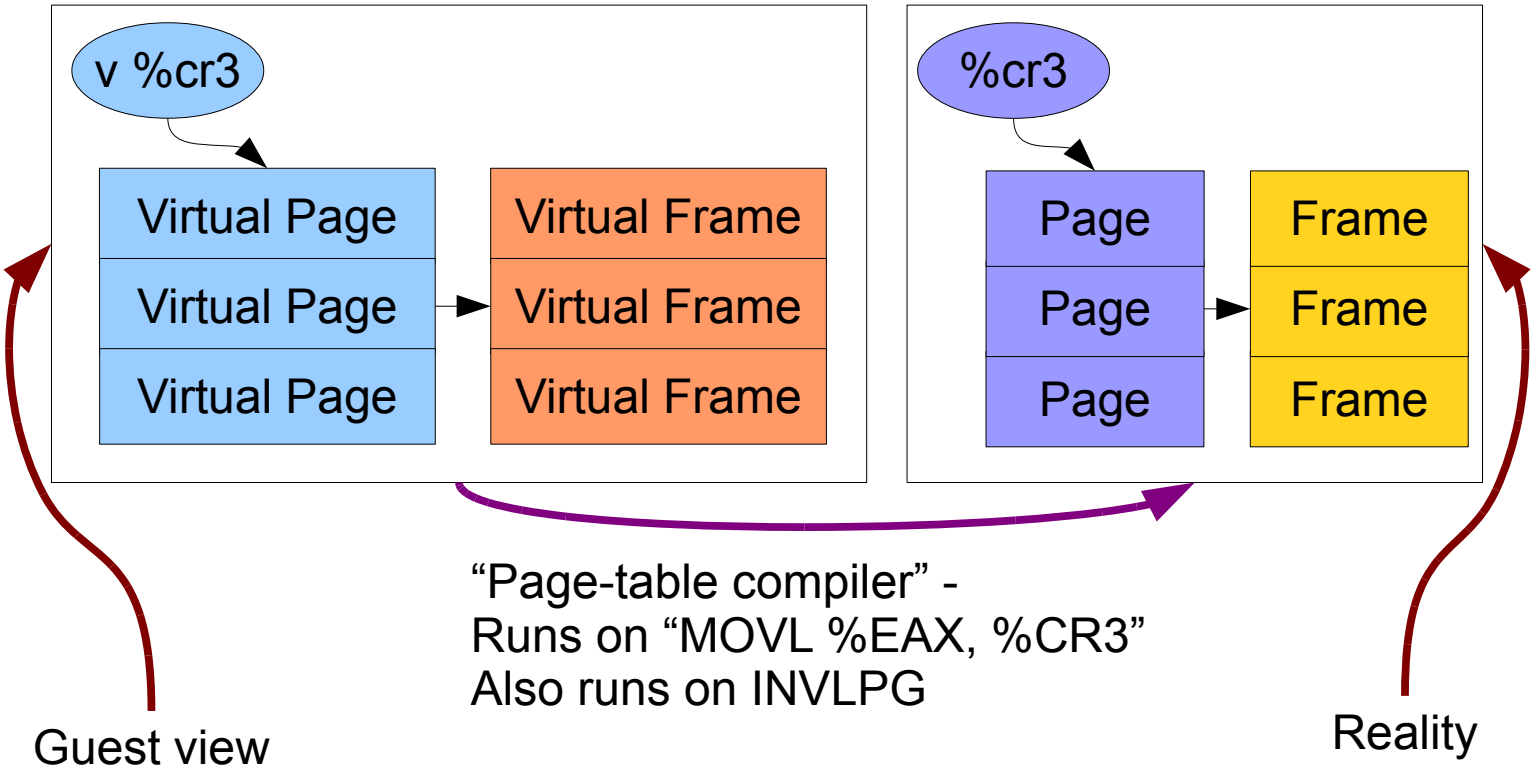


Note: traditional x86 VM hardware does not implement “map, then map again’

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
- Guest kernel may be able to tell it is running in a VM
  - Some sensitive instructions may leak user-mode-ness
  - Virtual devices may behave subtly wrong
  - Time dilation may be observed

- 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
- Guest kernel may be able to tell it is running in a VM
  - Some sensitive instructions may leak user-mode-ness
  - Virtual devices may behave subtly wrong
  - Time dilation may be observed

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

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 OS in rings 1-3**
    - **No illusion about running in a virtual environment**
    - **Guests 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

VMware vs. Paravirtualization

- **Kernel's device communication with VMware (emulated):**

```
void nic_write_buffer(char *buf, int size)
{
    for (; size > 0; size--) {
        nic_poll_ready();           // many traps
        outb(NIC_TX_BUF, *buf++);   // many traps
    }
}
```

- **Kernel's device communication with hypervisor (hypercall):**

```
void nic_write_buffer(char *buf, int size)
{
    vmm_write(NIC_TX_BUF, buf, size); // one trap
}
```

Xen (2003)

- Popular hypervisor supporting paravirtualization
  - Hypervisor runs on hardware
  - Runs two kinds of kernels
  - Host kernel runs in domain 0 (dom0)
    - Required by Xen to boot
    - Hypervisor contains no peripheral device drivers
    - dom0 needed to communicate with devices
    - Supports all peripherals that Linux or NetBSD do!
  - Guest kernels run in unprivileged domains (domU's)

Xen (2003)

- Provides virtual devices to guest kernels
  - Virtual block device, virtual ethernet device
  - Devices communicate with hypercalls & ring buffers
  - Can also assign PCI devices to specific domUs
    - Video card
- Also supports hardware assisted virtualization (HVM)
  - Allows Xen to run unmodified domU's
  - Useful for porting an OS to Xen PV
  - Also used for “certain OSes” with closed source
- Supports Linux & NetBSD as dom0 kernels
- Linux, FreeBSD, NetBSD, and Solaris as domU's

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?

Bonus Slides
(Material not covered in lecture)

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
- Virtualization
- x86 Virtualization
- Paravirtualization
- Alternatives for Isolation
- Alternatives for “running two OSes on same machine”
- Summary

chroot()

- Venerable Unix system call
- Runs a Unix process with a different root directory
  - Almost like having a separate file system
- Share the same kernel & non-filesystem “things”
  - Networking, process control
- Only a minimal sandbox.
  - /proc, /sys
  - Resources: I/O bandwidth, cpu time, memory, disk space, …

User-mode Linux

- Runs a guest Linux kernel as a user space process under a regular Linux kernel
- Requires highly modified Linux kernel
- No modification to application code
- Used to be popular among hosting providers
- More mature than Xen, roughly equivalent, but much slower because Xen is designed to host kernels

Container-based OS Virtualization

- Allows multiple instances of an OS to run in isolated *containers* under the same kernel
- Assumptions:
  - Want strong separation between "virtual machines"
  - But we can trust the kernel
  - Every "virtual machine" can use the same kernel version
- It follows that:
  - Don't need to virtualize the kernel
  - Instead, beef up naming and partitioning inside the kernel: Each *container* can have:
    - User id, pid, tid space
    - Domain name
    - Isolated file system, OS version, libraries, etc.
- Total isolation between containers without virtualization overhead.
- VServer, FBSD Jails, OpenVZ, Solaris Containers (aka "Zones")

Outline

- Introduction
- Virtualization
- x86 Virtualization
- Paravirtualization
- Alternatives for Isolation
- Alternatives for “running two OSes on same machine”
- 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 a reset

System Emulation (Bochs, DOSBox, QEMU)

- Seeks to emulate just enough of system hardware components to create an accurate “user experience”
- Typically CPU & memory subsystems are emulated
  - Buses are not
  - Devices communicate with CPU & memory directly
- Many shortcuts 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
- E.g. run legacy 680x0 code on PowerPC, run Windows on ??

System Emulation Techniques

- **Pure interpretation:**
  - **Interpret each guest instruction**
  - **Perform a semantically equivalent operation on host**
- **Static translation:**
  - **Translate each guest instruction to host once**
  - **Happens at startup**
  - **Limited applicability, no self-modifying code**

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**
- **Dynamic recompilation & adaptive optimization:**
  - **Discover what algorithm the guest code implements**
  - **Substitute with an optimized version on the host**
  - **Hard**

Vx32/9vx (2008)

- Vx32 is a library which sandboxes execution of 32-bit x86 code
  - Runs in user mode, no privileges (like QEMU, unlike VMware)
  - Two tricks
    - Memory is virtualized through x86 segmentation
      - Allocate some memory in your virtual address space
      - Request kernel to create a segment which maps (0,...) to (base +0,...)
      - Guest code runs in that segment
    - Sensitive instructions handled by binary translation
- When guest code is launched, it runs until an exception
  - Hypervisor examines reason, takes action

Vx32/9vx (2008)

- 9vx is a port of the Plan 9 OS kernel to Unix
  - Runs in user mode, no privileges
  - 9vx kernel uses host services (disk, network)
  - Plan 9 binaries are run without modification
    - Vx32 provides memory encapsulation
    - Execution proceeds until system call or page fault
    - 9vx kernel reacts by emulating syscall, paging memory in, etc.

QEMU's Portable Dynamic Translator

- Cute hack: uses GCC to pregenerate translated code
- Code executing on host is generated by GCC
  - Not hand written
- Makes QEMU easily portable to architectures that GCC supports
  - “The overall porting complexity of QEMU is estimated to be the same as the one of a dynamic linker.”

QEMU's Portable Dynamic Translator

Instructions for a given architecture are divided into micro-operations. For example:

```
addl $42, %eax # eax += 42
```

divides into:

```
movl_T0_EAX # T0   = eax
addl_T0_im  # T0  += 42
movl_EAX_T0 # eax  = T0
```

QEMU's Portable Dynamic Translator

- At (QEMU) compile time, each micro-op is compiled from C into an object file for the host architecture
  - *dyngen* copies the machine code from object files
  - Object code used as input data for code generator
- At runtime, code generator reads a stream of micro-ops and emits a stream of machine code
  - By convention, code executes properly as emitted

QEMU's Portable Dynamic Translator

Micro-operations are coded as individual C functions:

```
void OPPROTO op_movl_T0_EAX(void) { T0  = EAX }
void OPPROTO op_addl_T0_im(void)  { T0 += PARAM1 }
void OPPROTO op_movl_EAX_T0(void) { EAX = T0 }
```

which are compiled by GCC to machine code:

```
op_movl_T0_EAX:
   movl    0(%ebp), %ebx
   ret

op_addl_T0_im:
   addl    $42, %ebx
   ret

op_movl_EAX_T0:
   movl    %ebx, 0(%ebp)
   ret
```

QEMU's Portable Dynamic Translator

dyngen strips away function prologue and epilogue:

```
op_movl_T0_EAX:
   movl    0(%ebp), %ebx

op_addl_T0_im:
   addl    $42, %ebx

op_movl_EAX_T0:
   movl    %ebx, 0(%ebp)
```

QEMU's Portable Dynamic Translator

At runtime, QEMU translate the instruction:

```
add $42, %eax
```

into the micro-op sequence:

```
op_movl_T0_EAX
op_addl_T0_im
op_movl_EAX_T0
```

and then into machine code:

```
movl    0(%ebp), %ebx
addl    $42, %ebx
movl    %ebx, 0(%ebp)
```

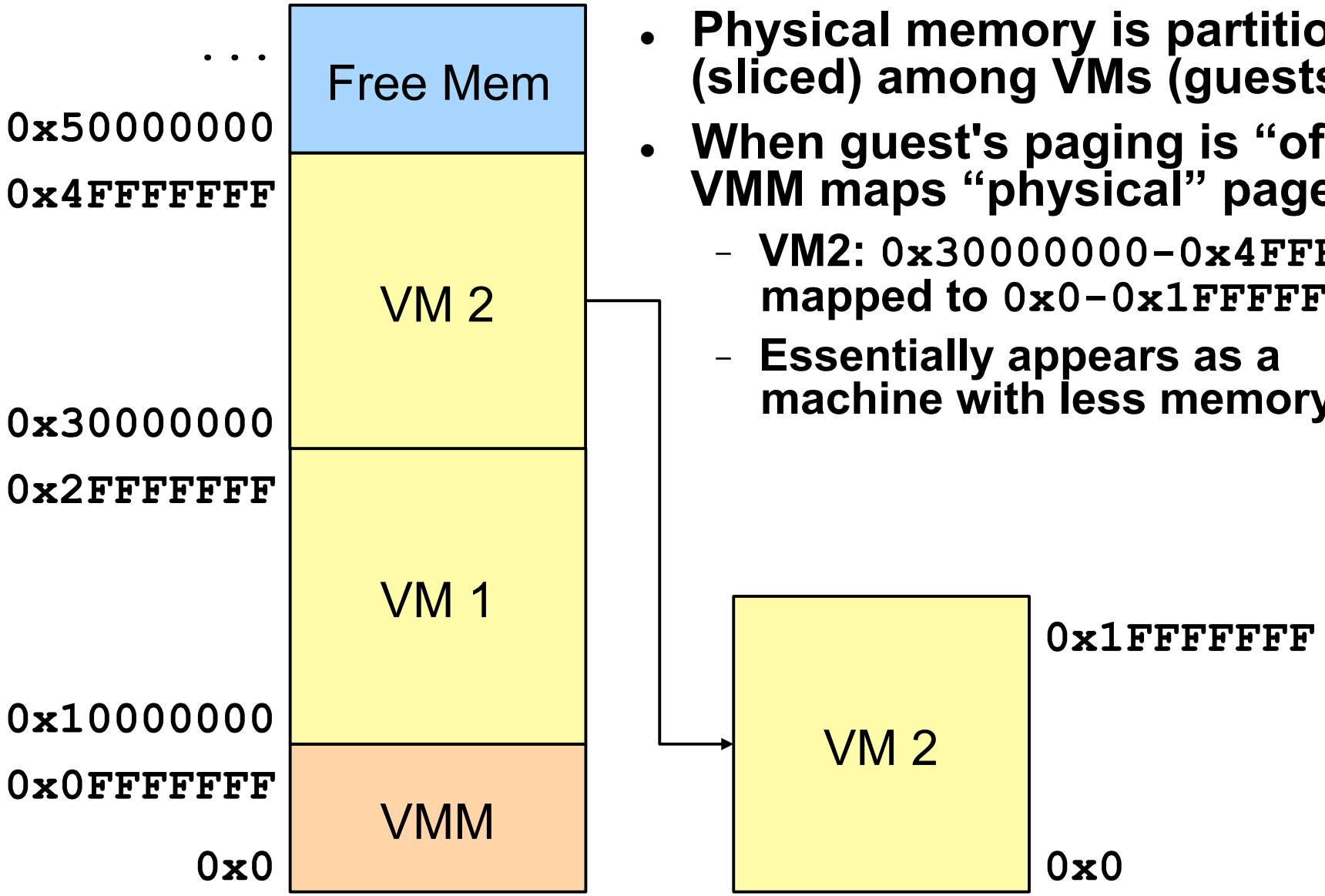
QEMU's Portable Dynamic Translator

- When QEMU encounters untranslated code, it translates each instruction until the next branch
  - Forms a single *translation block*
- After each code block is executed, the next block is located in the block hash table
  - Indexed by CPU state
  - Or, block is translated if not found
- Write protects guest code pages after translation
  - Write attempt indicates self modifying code
  - Translations are invalidated on write attempt

Summary

- Virtualization is big in enterprise hosting
- {Full, hardware assisted, para-}virtualization
- Containers: VM-like abstraction with high efficiency
- Emulation is a slower alternative, more flexibility

Physical Memory Map



- **Physical memory is partitioned (sliced) among VMs (guests)**
- **When guest's paging is "off", VMM maps "physical" pages**
  - **VM2: `0x30000000-0x4FFFFFFF` mapped to `0x0-0x1FFFFFFF`**
  - **Essentially appears as a machine with less memory**

Physical Memory Map

- **Physical memory is partitioned (sliced) among VMs (guests)**
- **When guest's paging is "off", VMM maps "physical" pages**
  - **VM2: `0x30000000-0x4FFFFFFF` mapped to `0x0-0x1FFFFFFF`**
  - **Essentially appears as a machine with less memory**
- **When guest's paging is "on", two levels of address translation are needed**

Virtual, Virtual Memory?

- **VM2 guest kernel attempts to write a page table entry:**

```
void map_page(u32 *pte, u32 phys_addr)
{
    *pte = phys_addr | 1;
}

...
map_page(pte, 0x10000000); // *pte = 0x10000001
...
```

- **But 0x10000000 isn't the right physical address!**
  - **VM2's 0x10000000 maps to physical 0x40000000**

Shadow Page Tables

- Guest kernel writes to page table, uses wrong address
  - VMM can't just “fix” the address (i.e., 0x10... => 0x40...)
    - Guest may later read page table entry (now is 0x40...)
    - Expects to see its “physical” addresss (0x10...)
- VMM keeps a shadow copy of each guest's page tables
- VMM must trap updates to cr3
  - Crawls guest page tables for updated entries
  - Writes real physical addresses to shadow table entries

Synchronization

Project 3 due Friday night

- **Please remember to register for late days as appropriate**
  - **One registration per day**
- **Please register for the midnight floppy disc seminar in the Wean 5th floor lobby**
  - **Registrations before 14:30 Friday are most useful**
- **Carefully consider the P3extra overtime**
  - **In general, getting a really solid kernel is the best thing**
    - **For your grade**
    - **For your education!**
  - **So once the dust has settled, run all the tests and carefully read the P4 requirements**
    - **p3extra is *not optional* in certain cases!**
  - **If you know early that you will p3extra, please register early**