Virtualization

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

April 11, 2011

Synchronization

Project 3 due tonight

- **Please remember to register for late days as appropriate**
  - **One registration per day**
- **Please register for the midnight floppy disc seminar in the 5th floor lobby**
  - **Registrations before 14:30 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**
  - **If you know early that you will p3extra, please register early**

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”
- Virtualization is “more like hardware”
  - 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”
- Virtualization is “more like hardware”
  - What runs inside virtualization is an operating system

Process : Kernel :: Kernel : Virtual-machine monitor

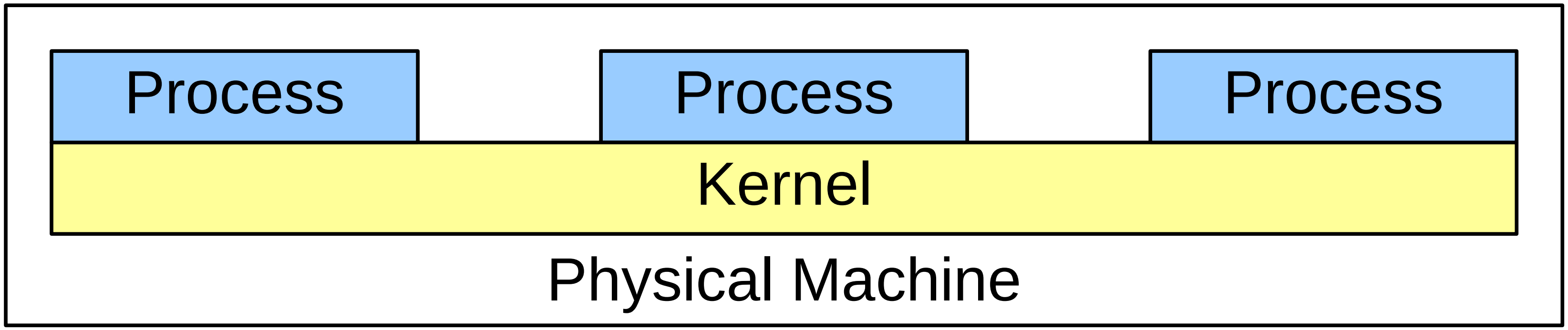
Advantages of the Process Abstraction

- Each process is a pseudo-machine
- Processes have their own registers, address space, file descriptors (sometimes)
- Protection from other processes

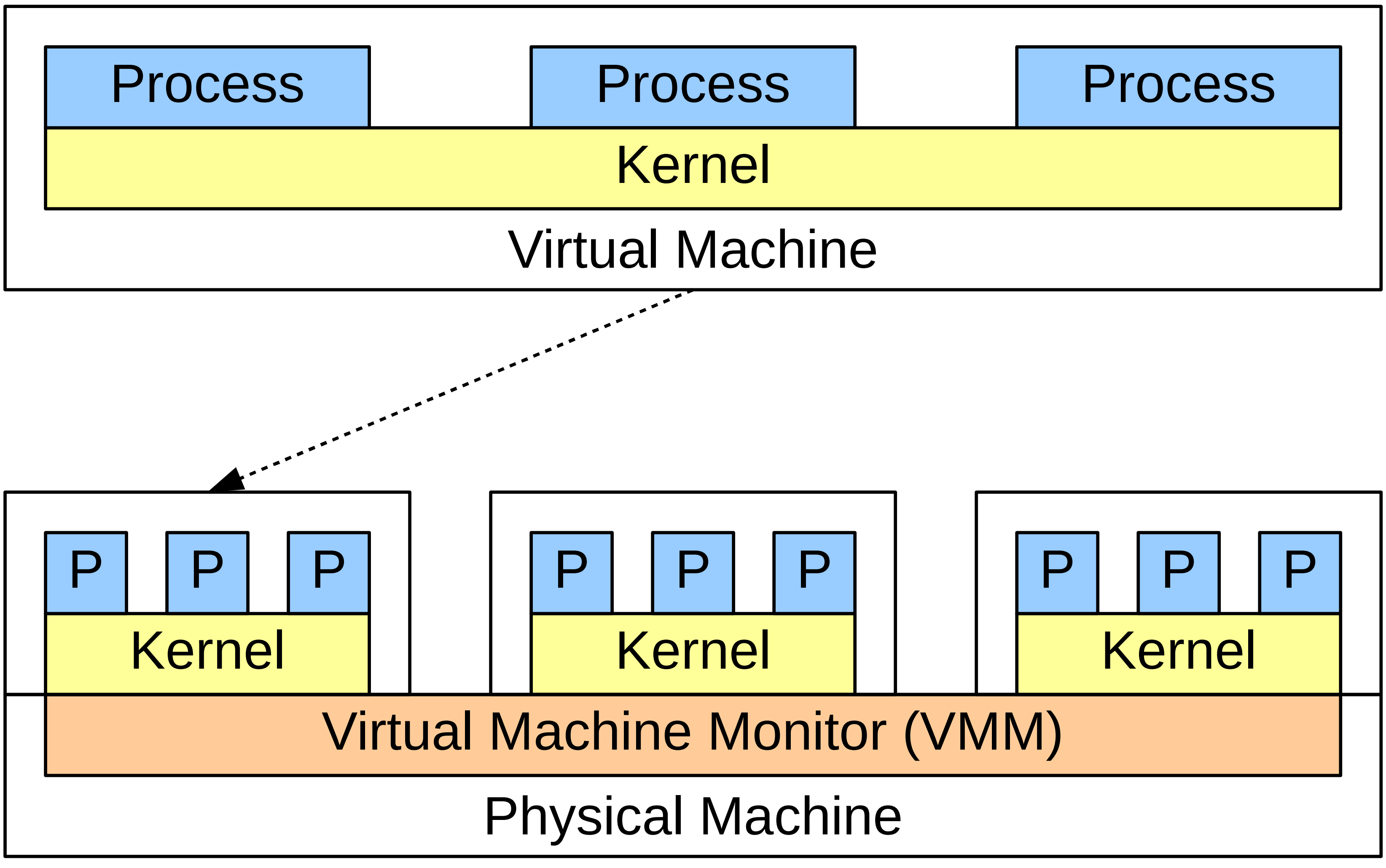
Disadvantages of the Process Abstraction

- 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
  - Kernels are *huge*, lots of possibly buggy code
- Processes have limited degree of protection, even from each other
  - Linux “OOM killer” can kill one process if another uses lots of memory
- Overall, processes aren't *that* isolated from each other...

Process/Kernel Stack



Virtualization Stack



Why Use Virtualization?

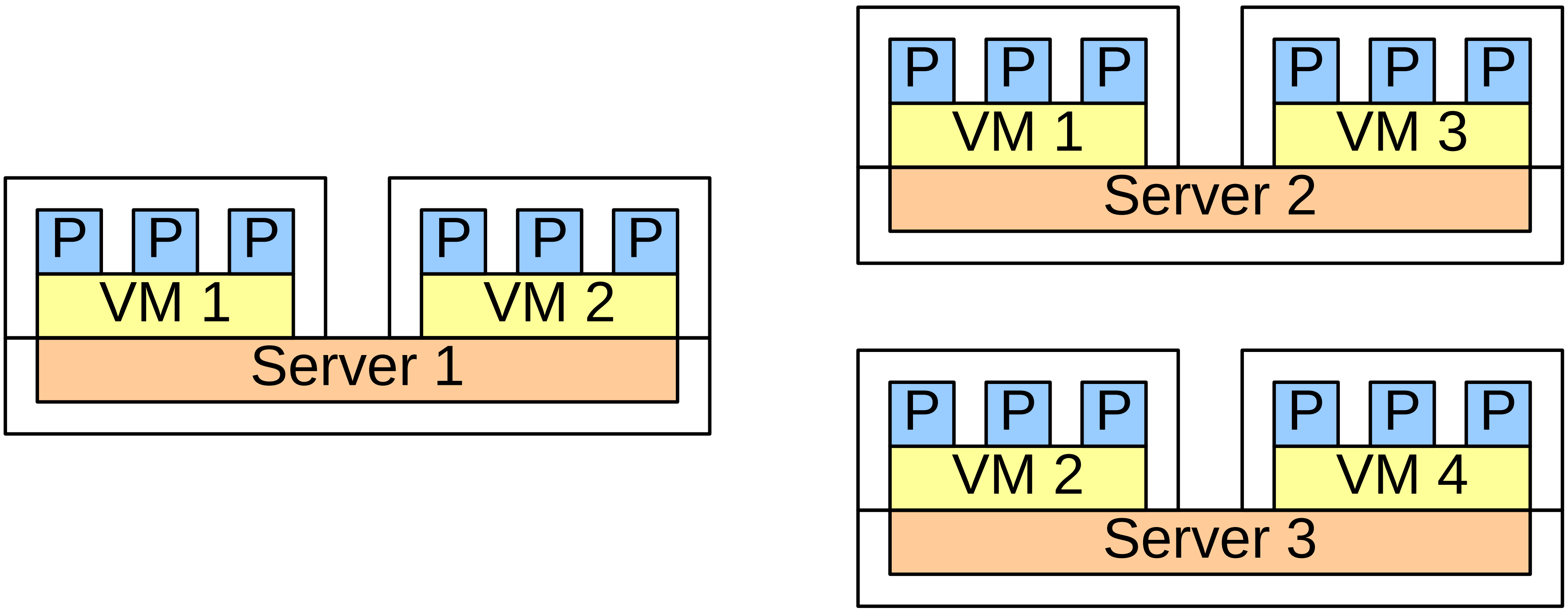
- Run two operating systems on the same machine!
  - “Windows+Linux” was VMware's first business model
  - Hobbyists like to run ancient-history OS's
- Debugging OS's is more pleasant
  - Also: instrumenting what an OS does
  - Monitoring a captive OS for security infestations
- “Process abstraction”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

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:**



Disadvantages of Virtual Machines

- **Attempt to solve what really is an abstraction issue somewhere else**
  - **Monolithic kernels**
  - **Not enough partitioning of global identifiers**
    - **pids, uids, etc**
  - **Applications written without distribution and fault tolerance in mind**
- **Provides some interesting mechanisms, but may not directly solve “the problem”**

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 10-20% performance hit tolerable?**
  - **Can your NIC or disk keep up with the load?**

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)

- 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 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 what 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* diff cult!**
    - **Thousands of devices exist, each one is extremely complex**

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**
- **Eff ciency:**
  - **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 if 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**
- **Enables a VMM to 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 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 a workaround

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`**
- **VMware performs *binary translation* on guest code to work around sensitive, unprivileged instructions:**
  - **`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 CLI_OP:
            /* 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)
{
    regs->eflags = *(regs->esp);
    regs->esp++;
    // Defer or deliver interrupts as appropriate
}
```

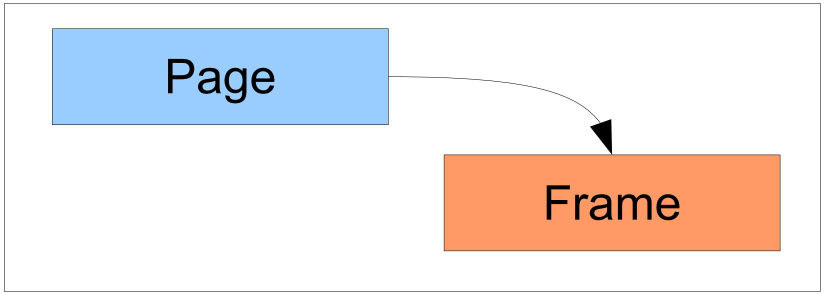
Related technologies

- Software Fault Isolation (Lucco, UCB, 1993)
- VX32 (Ford & Cox, MIT, 2008)

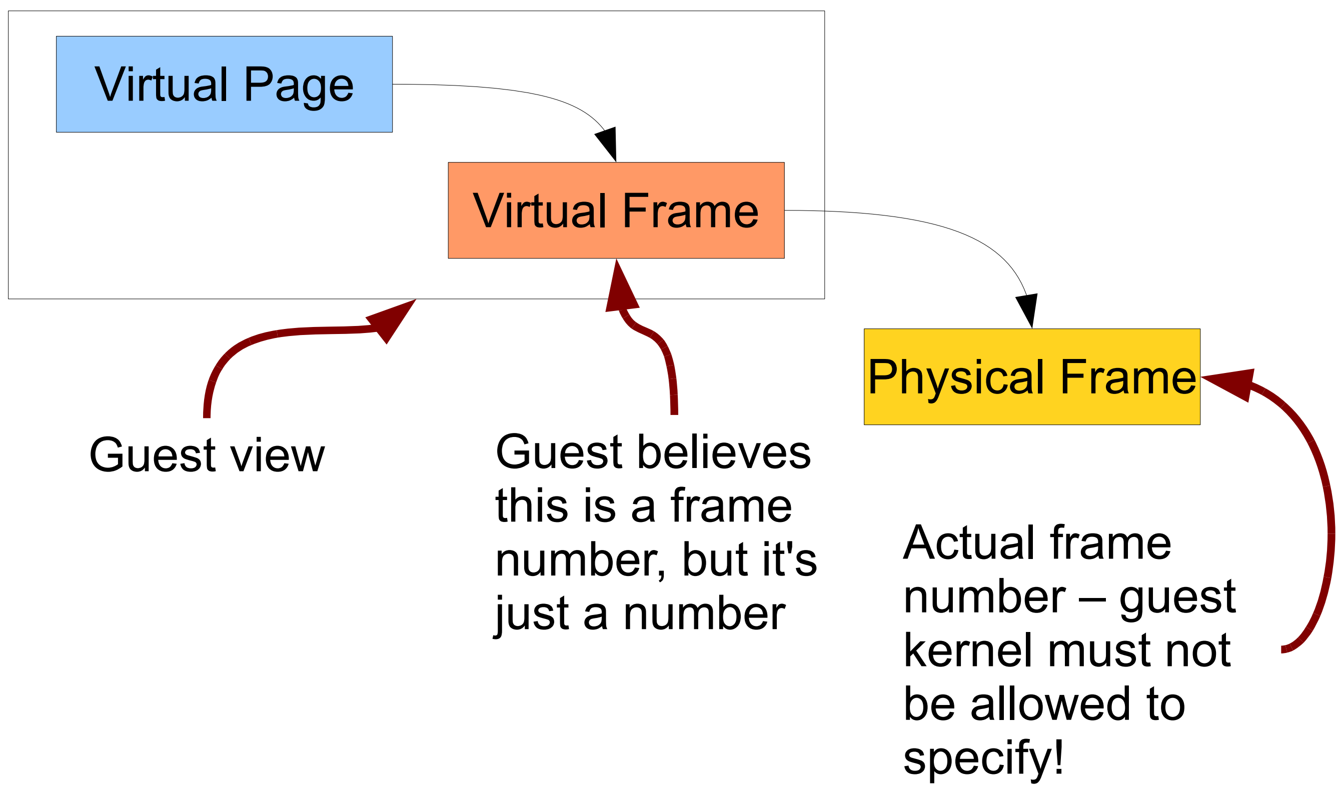
Virtual Memory

- 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??

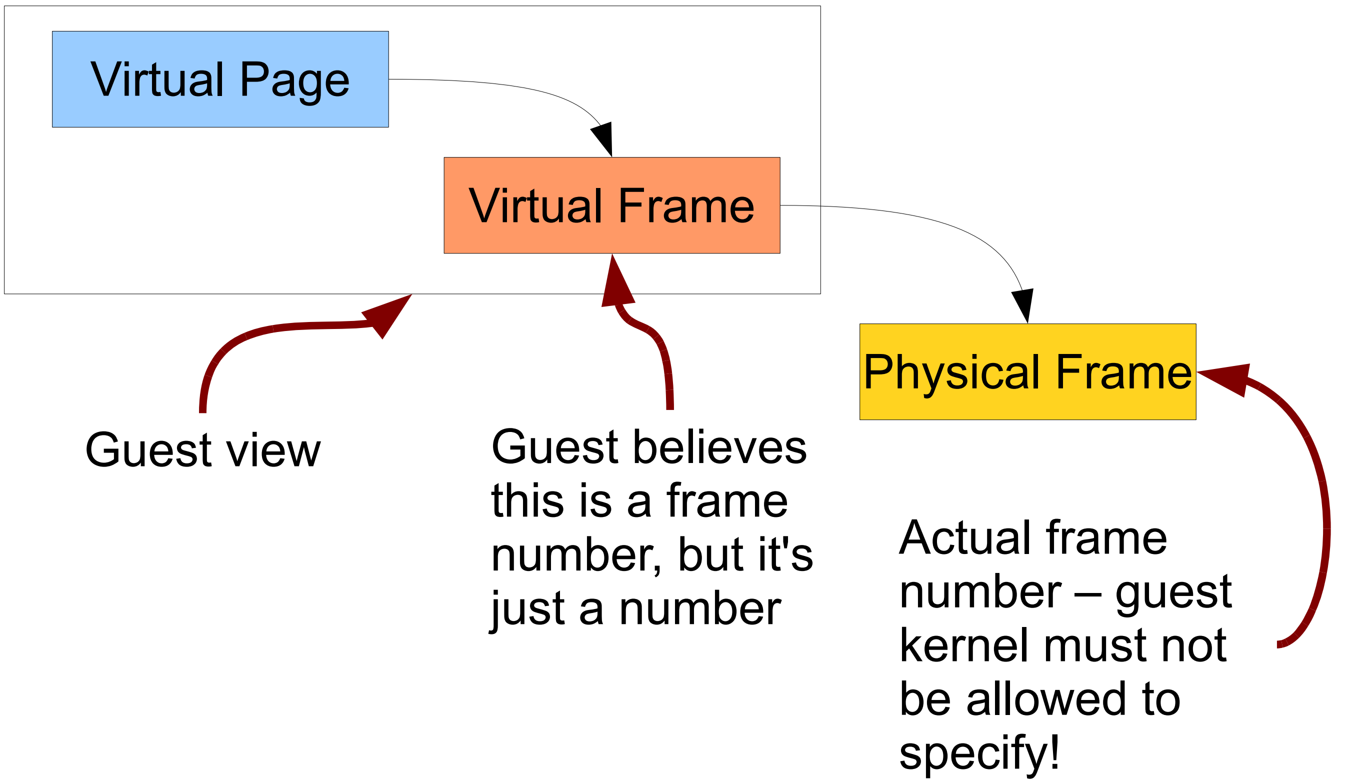
VM – Guest-kernel view



VM – Fiction vs. Reality

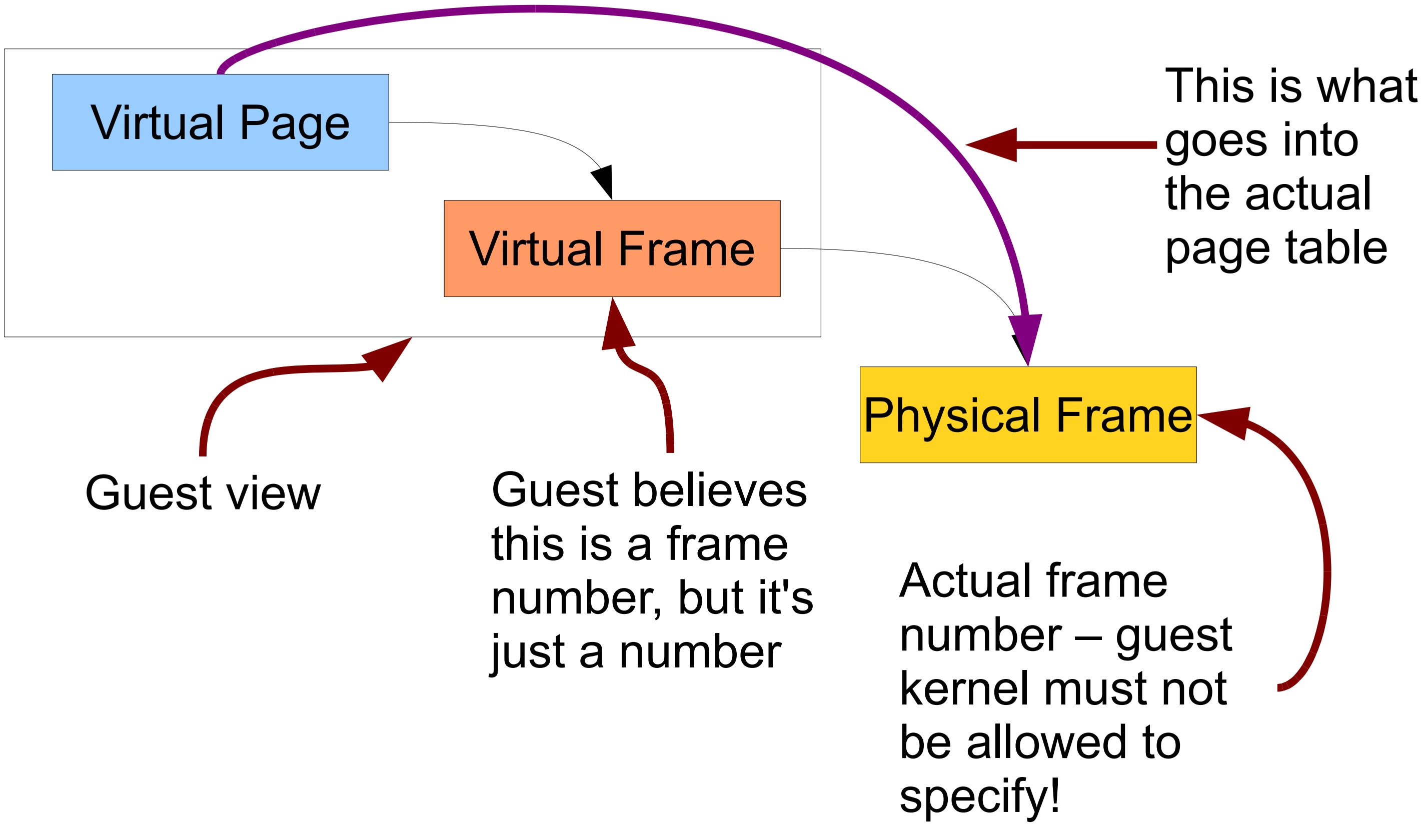


VM – How to do it?

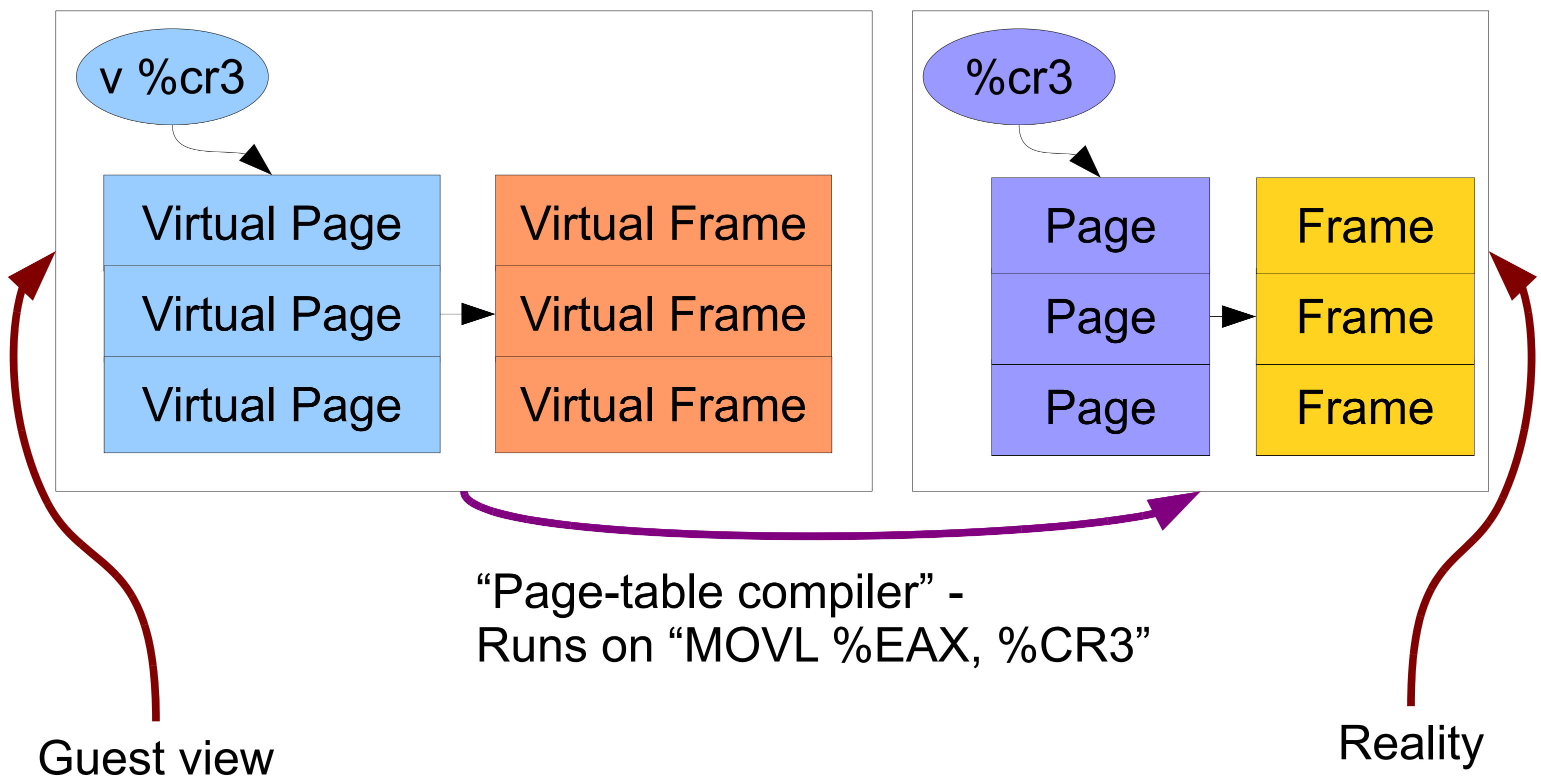


Note: VM hardware does not implement "map, then map again"

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**
    - **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 PTE, we must re-compile**
  - **So virtual page tables are read-only 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 has some**

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?**
  - **Fix the hardware**
  - **Blur the hardware ("paravirtualization")**

Hardware Assisted Virtualization

- Recent variants of the x86 ISA 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, no illusions necessary
- At least initially, binary translation was faster than VT
  - `int $99` is a “regular” trap, faster than a “special trap”

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();
        outb(NIC_TX_BUF, *buf++);
    }
}
```

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

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

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 bootstrapping**
  - **Also used for “certain OSes” that can't be source modified**
- **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**
- **Open question**
  - **How *best* to get isolation, machine independence?**

Summary

- What virtualization does
  - Multiple OS's on one laptop
  - Debugging, security analysis
  - Enterprise
    - Eff ciency
    - Reliability (outage resistance)
- The problem
  - Kinds of instructions
- Solutions
  - Binary translation (useful for light-weight uses)
  - {Full, hardware assisted, para-}virtualization

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.

Bonus Slides
(Material not covered in lecture)

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)
```

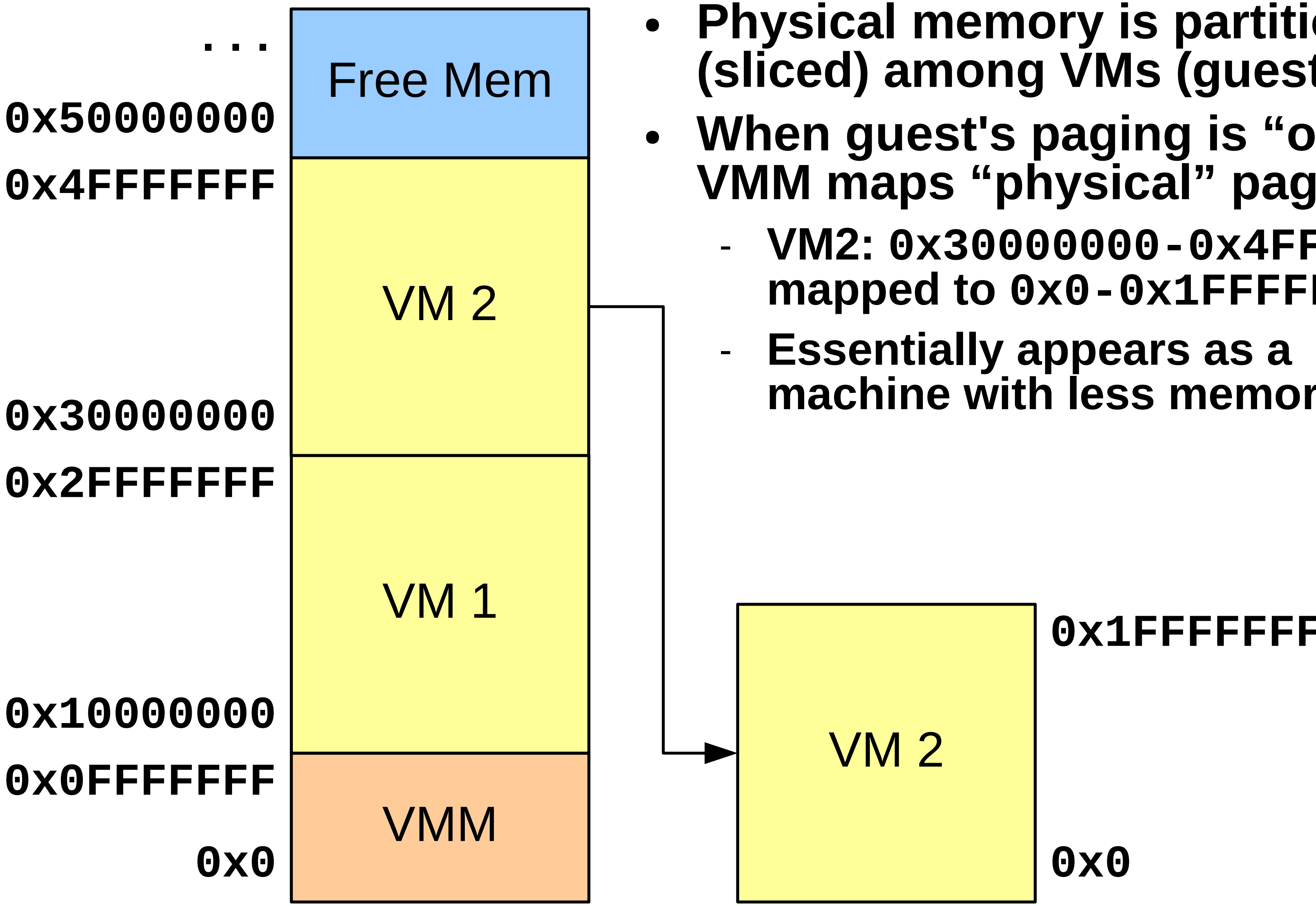
QEMU's Portable Dynamic Translator

- 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

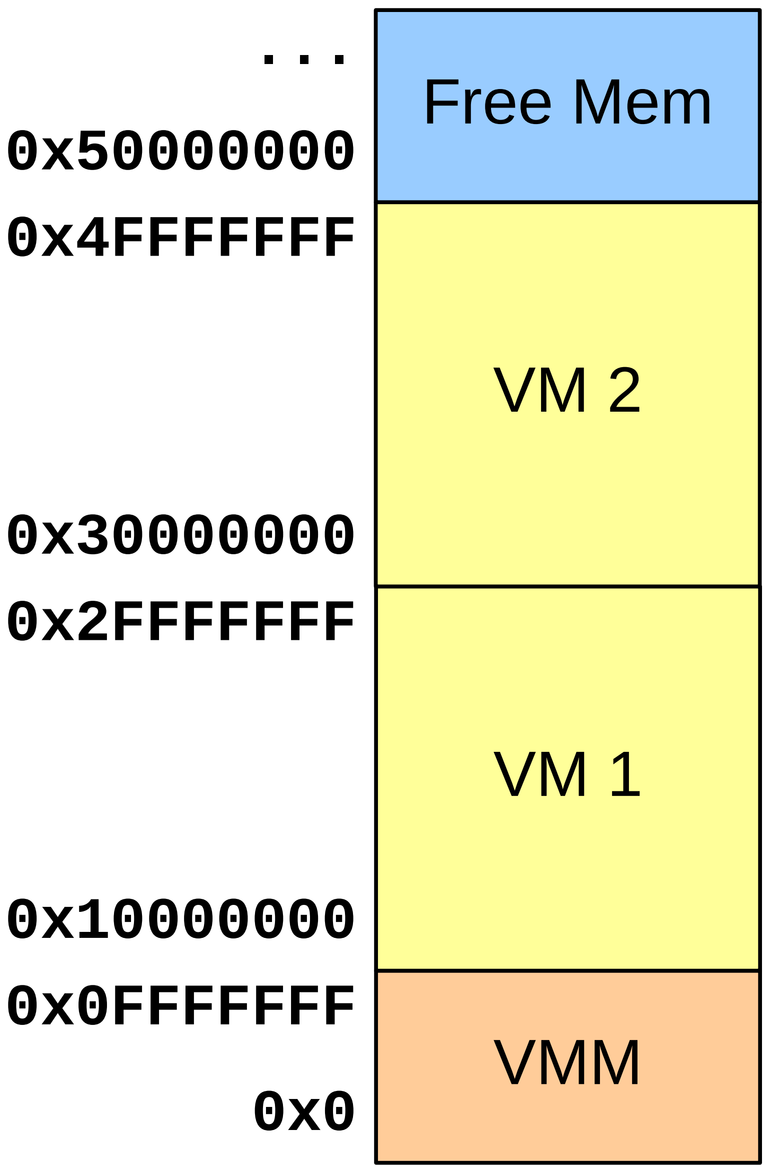
- Virtualization is big in enterprise hosting
- {Full, hardware assisted, para-}virtualization
- Containers: VM-like abstraction with high efficiency
- Emulation is a slower alternative, more flexibility

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**