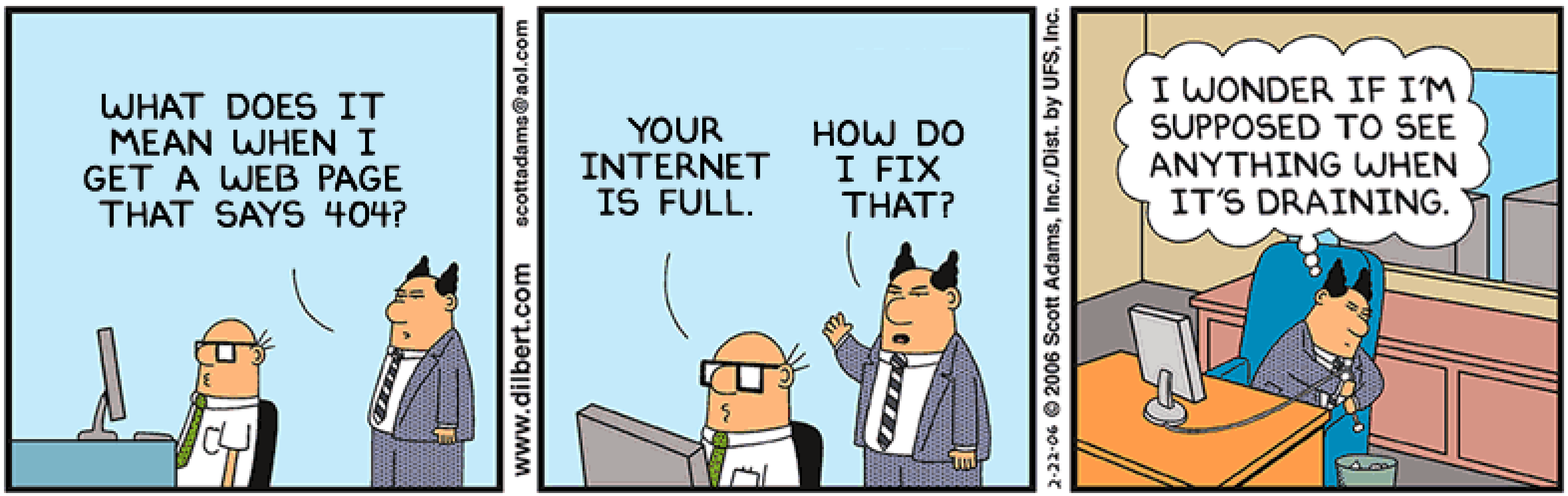
Lecture 26: Virtual Machines

Instructor: Mitch Neilsen
Office: N219D

Quote of the Day

404 NOT FOUND



Chapter 16: Virtual Machines

- **Simulation and Emulation**
- Virtualization
- x86 Virtualization
- 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-file system “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**

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**

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 Pintos on x86

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**

QEMU's Portable Dynamic Translator

- Cute hack: uses GCC to pre-generate 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**

Outline

- Simulation and Emulation
- **Virtualization**
- x86 Virtualization
- Summary

What is Virtualization?

- **Virtualization:**
  - **Process 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**
- **The *process* abstraction is related to virtualization: it's at least similar to a physical machine**

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

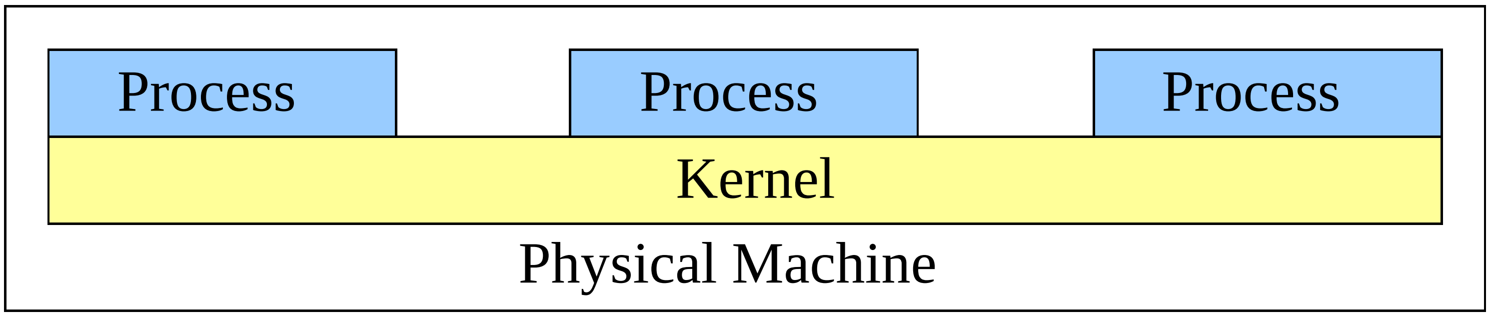
Disadvantages of the Process Abstraction

- **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**
    - **Other processes aren't so isolated after all**

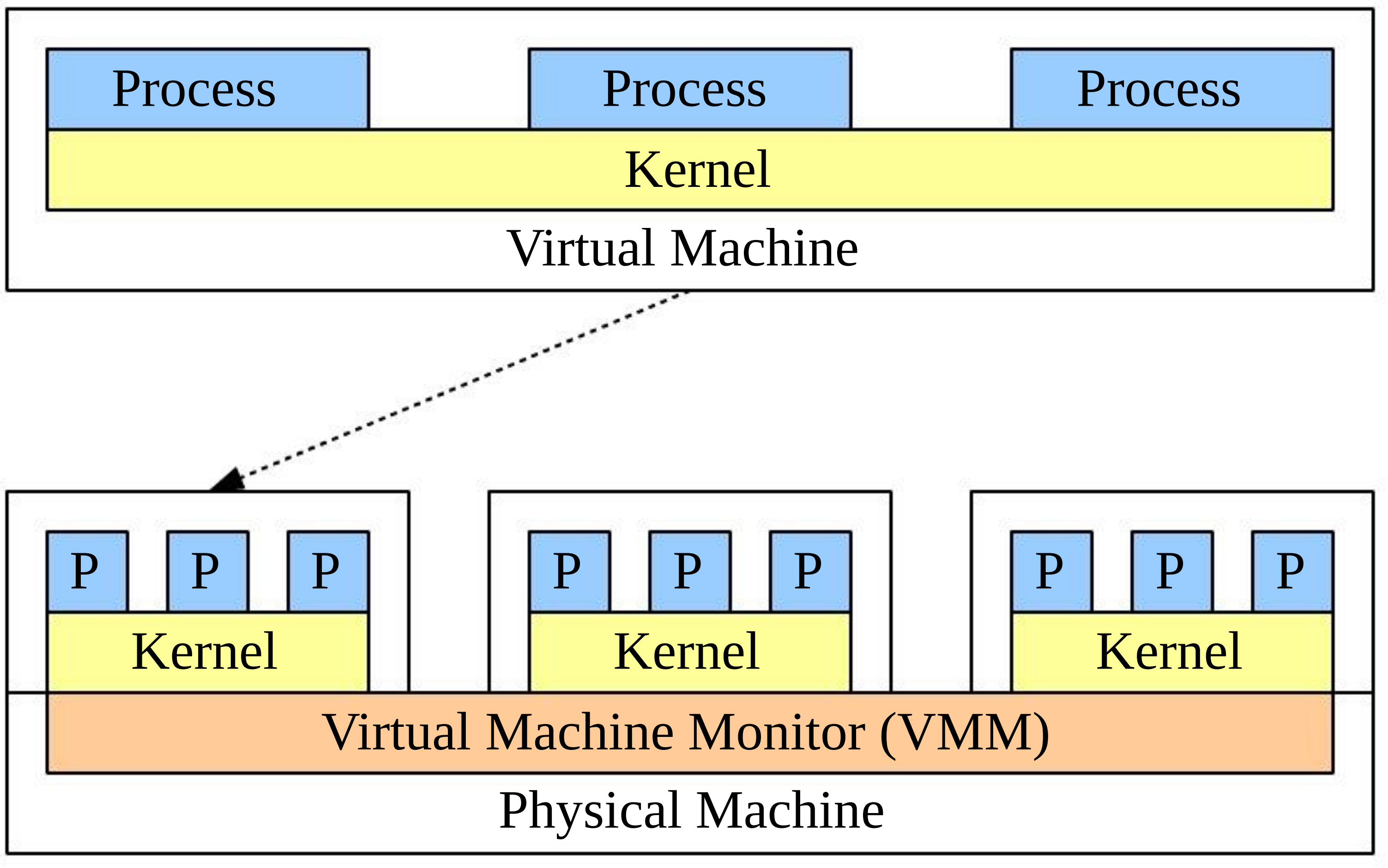
Disadvantages of the Process Abstraction

- **Processes share the same kernel**
  - **Kernel/OS specific software**
  - **Kernels are *huge*, lots of possibly buggy code**
- **Processes have limited degree of protection, even from each other**
  - **OOM (out of memory) killer (in Linux) frees memory when all else fails**

Process/Kernel Stack



Virtualization Stack



Why Use Virtualization?

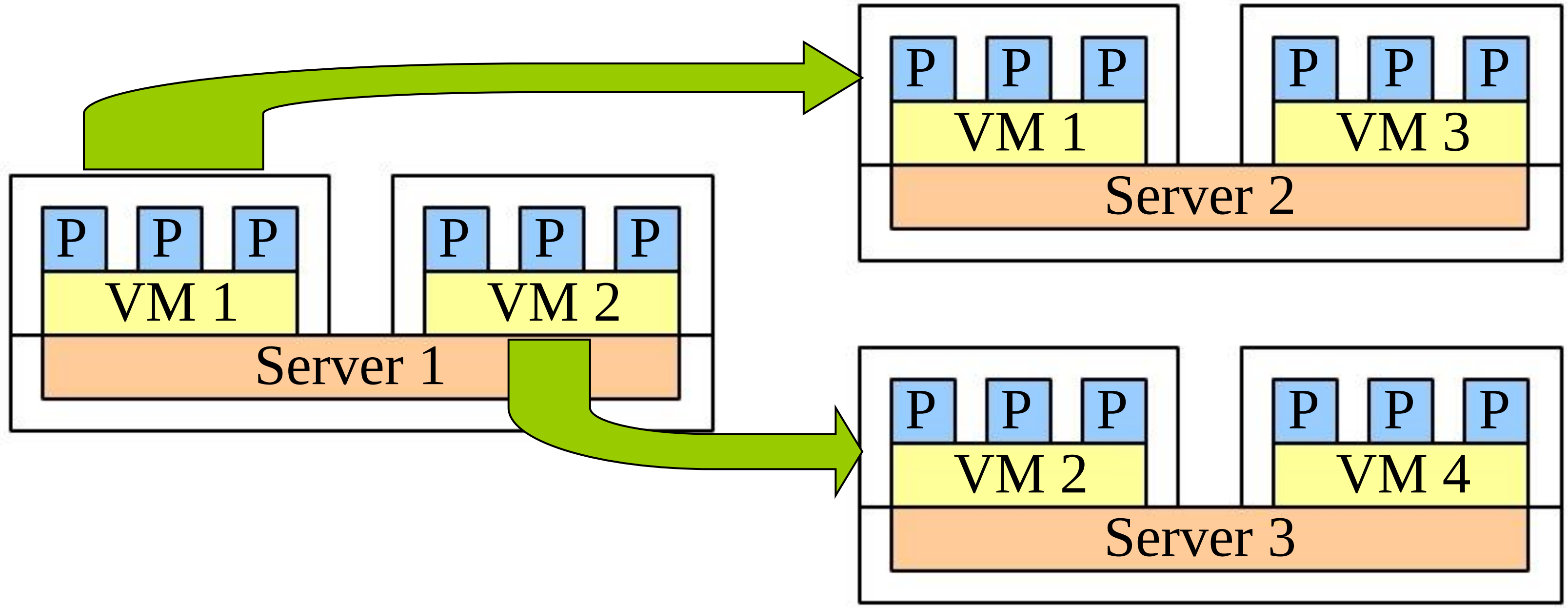
- “Process abstraction” at the *kernel* layer
  - Separate file system
  - Different machine owners
- Offers much better protection (in theory)
  - Secure hypervisor, fair scheduler
  - Interdomain DoS? Thrashing?
- Run two operating systems on the same machine!

Why Use Virtualization?

- Huge impact on enterprise hosting
  - No longer need to sell whole machines
  - Sell machine slices
  - 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
  - No more maintenance downtime
- VM replication to provide fault tolerance
  - "Why bother doing it at the application level?"

Virtualization in Enterprise

- Separates product (OS services) from phsyical 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?

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 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."

- **All instructions must either:**
  - **Exhibit the same result in user and supervisor modes**
  - **Or, they must 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.**

Outline

- Simulation and Emulation
- Virtualization
- **x86 Virtualization**
- Summary

x86 Virtualization

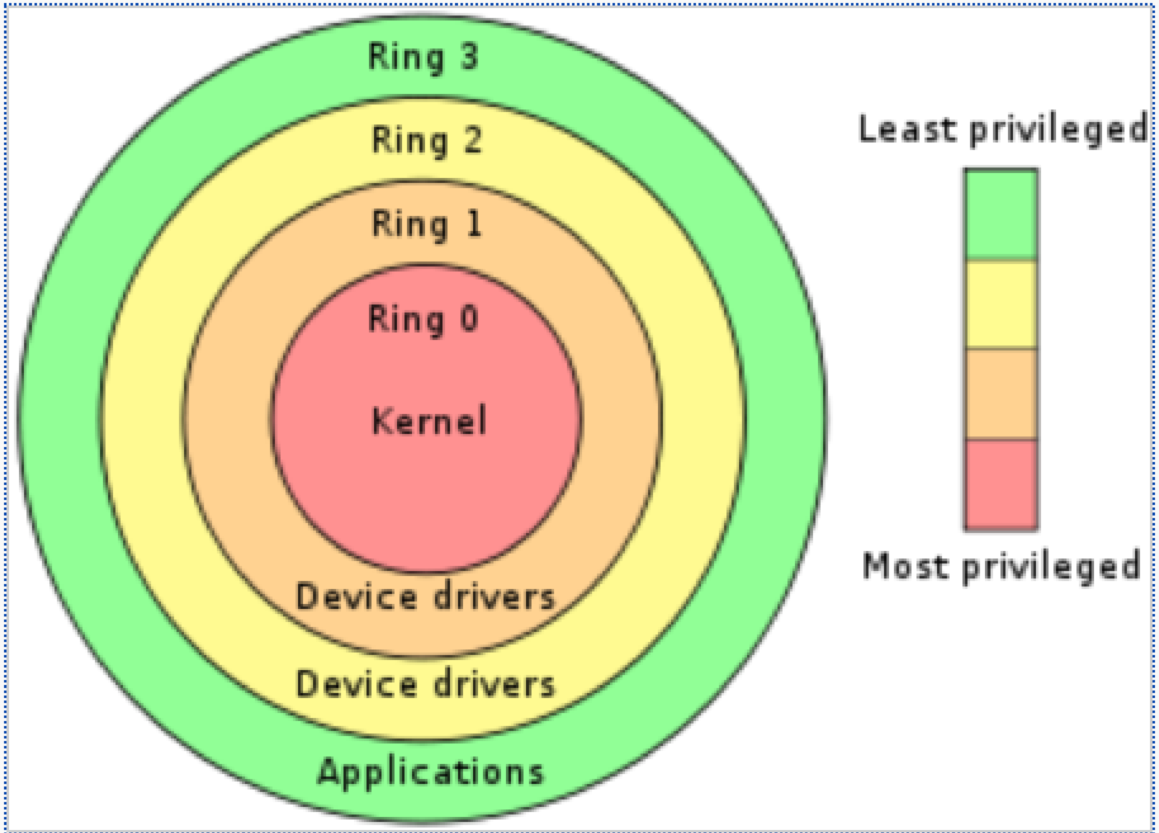
- **x86 ISA does not meet the Popek & Goldberg requirements for virtualization**
- **ISA contains 17+ sensitive, unprivileged instructions:**
  - **SGDT, SIDT, SLDT, SMSW, PUSHF, POPF, LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT, RET, STR, MOV**
  - **Most simply reveal the processor's CPL (Current Privilege Level)**
- **Virtualization is still possible, requires a workaround**

The “POPF Problem”

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

- **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**
  - **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 run in user mode!**

Operating System Protection Rings



Privilege rings for the x86 available in protected mode

- from Wikipedia

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

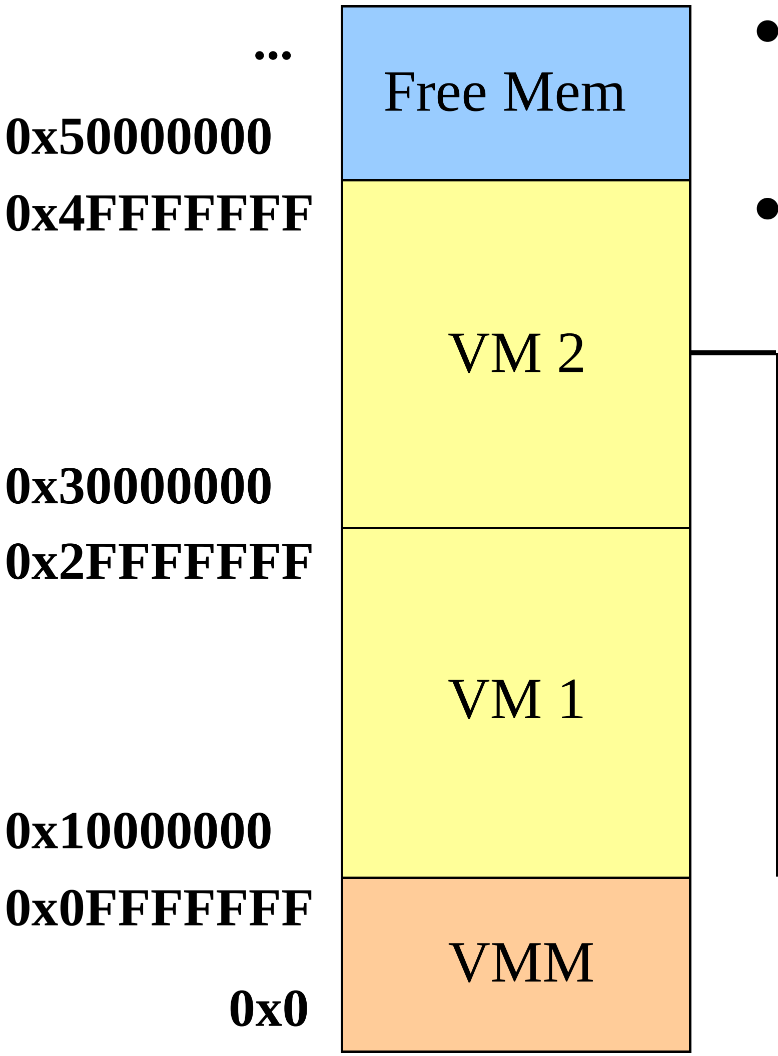
Software Fault Isolation (Lucco, UCB, 1993)

VX32 (Ford & Cox, MIT, 2008)

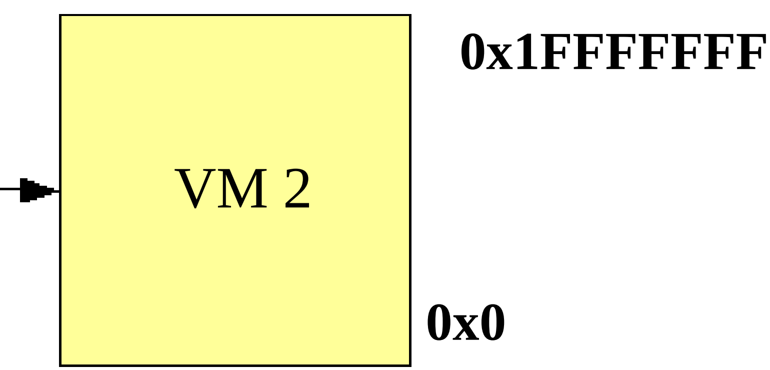
Virtual Memory

- We've virtualized instruction execution
  - How about other resources?
- Kernels access physical memory and implements virtual memory.
  - How do we virtualize physical memory?
    - Use virtual memory (obvious so far, isn’t it?)
  - If guest kernel runs in virtual memory, how does it provide virtual memory for processes?

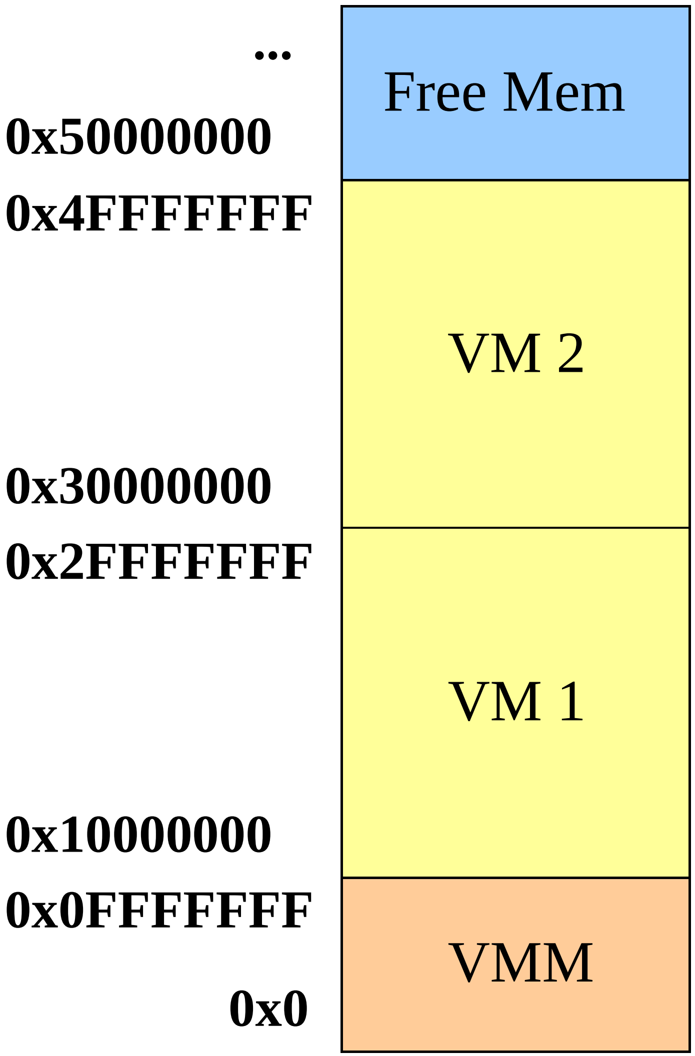
Physical Memory Map



- **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**

Other Virtual Memory Issues

- **Consistency**
  - **How do guest's "active" page table updates propagate to shadow copy?**
    - **Trap TLB flushes (cr3 & invlpg) and update**
    - **Trap writes w/read-only guest tables, update on page faults**
- **Privileges**
  - **VMM > kernel > user, but only one permission bit**
  - **One solution: Separate tables for guest kernel & user**
- **Many tricks played to improve performance**
- **Dirty & accessed bits? Won't get into them.**

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 faster than VT
  - int $99 is a “regular” trap, faster than a “special trap”

Summary

- Virtualization is big in enterprise hosting
- {Full, hardware assisted, para-}virtualization

Next:

- Wednesday: Final Exam Out
- Friday: Programming Lab 3: Android OS on Galaxy Tab 10.1

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 por table dynamic translator.
  In *Proceedings of the 2005 USENIX Annual Technical Conference*, Anaheim, CA, April 2005.