

# Administrivia

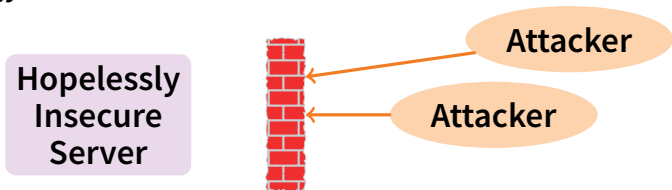
- **Last project due Friday**
- **Final exam by email Wednesday, March 18th, 3:30pm-6:30pm**
  - Likely ASCII-only exam to be submitted on AFS
  - SCPD students do not need an exam monitor
  - SCPD only can start up to 24 hours later
  - Please monitor class list and stanford mail for details
- **Exam content**
  - Open notes (except textbook), **exam may require web searches**
  - Covers all lectures including topics already on the midterm
  - Make sure you understand all answers to midterm before final
- **Final review session Friday (zoom only)**
- **Pre-exam office hours for me 2pm-4pm Monday (talky only)**

# Outline

- 1 Confining code with legacy OSES
- 2 Virtual machines
- 3 Implementing virtual machines
- 4 Binary translation
- 5 Hardware-assisted virtualization
- 6 Memory management optimizations

# Confining code with legacy OSes

- Often want to confine code on legacy OSes
- Analogy: Firewalls



- Your machine runs hopelessly insecure software
- Can't fix it—no source or too complicated
- *Can* reason about network traffic
- **Can we similarly block untrusted code *within* a machine**
  - Have OS limit what the code can interact with

# Using chroot

- `chroot (char *dir)` **“changes root directory”**
  - Kernel stores root directory of each process
  - File name `“/”` now refers to `dir`
  - Accessing `“..”` in `dir` now returns `dir`
- **Need root privileges to call chroot**
  - But subsequently can drop privileges
- **Ideally “Chrooted process” wouldn’t affect parts of the system outside of `dir`**
  - Even process still running as root shouldn’t escape chroot
- **In reality, many ways to cause damage outside `dir`**

# Escaping chroot

- **Re-chroot to a lower directory, then chroot ../..**
  - Each process has one root directory in process structure
  - Implementation special-cases / (always) & .. in root directory
  - `chroot` does not always change current directory
  - So chrooting to a lower directory puts you above your new root (Can re-chroot to real system root)
- **What else can you do as root in a chrooted process?**

# Escaping chroot

- **Re-chroot to a lower directory, then chroot ../..**
  - Each process has one root directory in process structure
  - Implementation special-cases / (always) & .. in root directory
  - chroot does not always change current directory
  - So chrooting to a lower directory puts you above your new root (Can re-chroot to real system root)
- **Create devices that let you access raw disk**
- **Send signals to or ptrace non-chrooted processes**
- **Create setuid program for non-chrooted processes to run**
- **Bind privileged ports, mess with clock, reboot, etc.**
- **Problem: chroot was not originally intended for security**
  - FreeBSD [jail](#) attempts to address the problems
  - Also, Linux [cgroups](#), [namespaces](#) allow containers

# System call interposition

- Why not use *ptrace* or other debugging facilities to control untrusted programs?
- Almost any “damage” must result from system call
  - delete files → unlink
  - overwrite files → open/write
  - attack over network → socket/bind/connect/send/recv
  - leak private data → open/read/socket/connect/write ...
- So enforce policy by allowing/disallowing each syscall
  - Theoretically much more fine-grained than chroot
  - Plus don't need to be root to do it
- **Q: Why is this not a panacea?**

# Limitations of syscall interposition

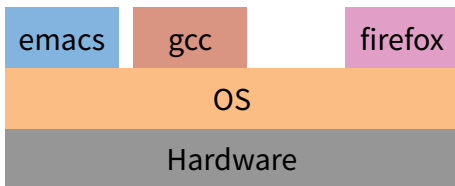
- **Hard to know exact implications of a system call**
  - Too much context not available outside of kernel (e.g., what does this file descriptor number mean?)
  - Context-dependent (e.g., `/proc/self/cwd`)
- **Indirect paths to resources**
  - File descriptor passing, core dumps, “unhelpful processes”
- **Race conditions**
  - Remember difficulty of eliminating TOCCTOU bugs?
  - Now imagine malicious application deliberately doing this
  - Symlinks, directory renames (so “..” changes), ...
- See [[Garfinkel](#)] for a more detailed discussion



# Outline

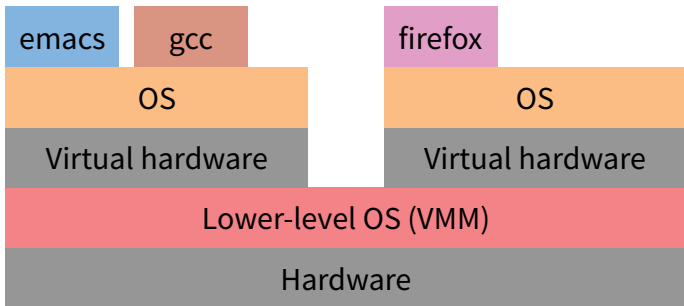
- 1 Confining code with legacy OSes
- 2 **Virtual machines**
- 3 Implementing virtual machines
- 4 Binary translation
- 5 Hardware-assisted virtualization
- 6 Memory management optimizations

# Review: What is an OS



- **OS is software between applications and hardware/external reality**
  - Abstracts hardware to makes applications portable
  - Makes finite resources (memory, # CPU cores) appear much larger
  - Protects processes and users from one another

# What if...



- The process abstraction looked just like hardware?

# How do process abstraction & HW differ?

## Process

Non-privileged registers and instructions

Virtual memory

Errors, signals

File system, directories, files, raw devices

## Hardware

All registers and instructions

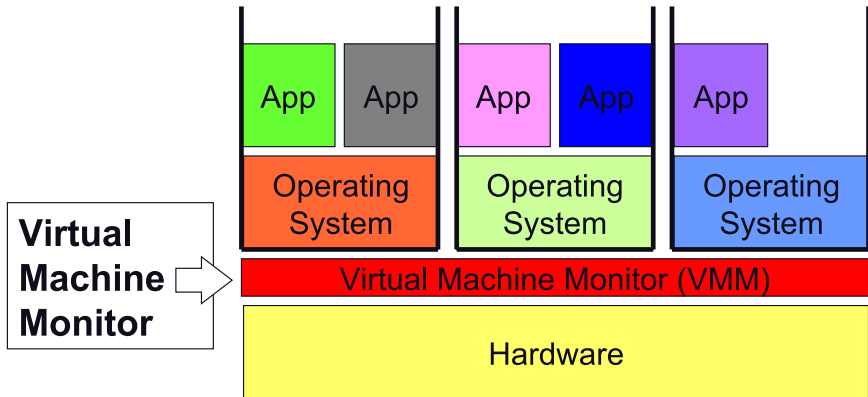
Both virtual and physical memory, MMU functions, TLB/page tables, etc.

Trap architecture, interrupts

I/O devices accessed using programmed I/O, DMA, interrupts

# Virtual Machine Monitor

- **Thin layer of software that virtualizes the hardware**
  - Exports a virtual machine abstraction that looks like the hardware



# Old idea from the 1960s

- See [\[Goldberg\]](#) from 1974
- **IBM VM/370 – A VMM for IBM mainframe**
  - Multiplex multiple OS environments on expensive hardware
  - Desirable when few machines around
- **Interest died out in the 1980s and 1990s**
  - Hardware got cheap
  - Just put a windows machine on every desktop
- **Today, VMs are used everywhere**
  - Used to solve different problems (software management)
  - But VMM attributes more relevant now than ever

# VMM benefits

- **Software compatibility**
  - VMMs can run pretty much all software
- **Can get low overheads/high performance**
  - Near “raw” machine performance for many workloads
  - With tricks can have direct execution on CPU/MMU
- **Isolation**
  - Seemingly total data isolation between virtual machines
  - Leverage hardware memory protection mechanisms
- **Encapsulation**
  - Virtual machines are not tied to physical machines
  - Checkpoint/migration

# OS backwards compatibility

- **Backward compatibility is bane of new OSES**
  - Huge effort require to innovate but not break
- **Security considerations may make it impossible**
  - Choice: Close security hole and break apps or be insecure
- **Example: Windows XP is end of life**
  - 4.59% machines ran 2001 Windows XP in 2018
  - XP support ended in 2019, eventually XP-capable hardware will die
  - What to do with legacy WinXP applications?
  - Not all applications will run on later Windows
  - Given the number of WinXP applications, practically any OS change will break something
    - if (OS == WinXP) ...
- **Solution: Use a VMM to run both WinXP and Win10**
  - Obvious for OS migration as well: Windows → Linux



# Logical partitioning of servers

- **Run multiple servers on same box (e.g., Amazon EC2)**
  - Modern CPUs more powerful than most services need
  - VMs let you give away less than one machine
  - Server consolidation trend:  $N$  machines  $\rightarrow$  1 real machine
  - 0.10U rack space machine – less power, cooling, space, etc.
- **Isolation of environments**
  - Printer server doesn't take down Exchange server
  - Compromise of one VM can't get at data of others<sup>1</sup>
- **Resource management**
  - Provide service-level agreements
- **Heterogeneous environments**
  - Linux, FreeBSD, Windows, etc.

---

<sup>1</sup>In practice not so simple because of side channels [[Ristenpart](#)] [[Meltdown](#)]

# Outline

- 1 Confining code with legacy OSes
- 2 Virtual machines
- 3 **Implementing virtual machines**
- 4 Binary translation
- 5 Hardware-assisted virtualization
- 6 Memory management optimizations

# Complete Machine Simulation

- **Simplest VMM approach, used by bochs**
- **Build a simulation of all the hardware**
  - CPU – A loop that fetches each instruction, decodes it, simulates its effect on the machine state
  - Memory – Physical memory is just an array, simulate the MMU on all memory accesses
  - I/O – Simulate I/O devices, programmed I/O, DMA, interrupts
- **Problem: Too slow!**
  - CPU/Memory – 100x CPU/MMU simulation
  - I/O Device –  $< 2\times$  slowdown.
  - $100\times$  slowdown makes it not too useful
- **Need faster ways of emulating CPU/MMU**

# Virtualizing the CPU

- **Observations: Most instructions are the same regardless of processor privileged level**
  - Example: `incl %eax`
- **Why not just give instructions to CPU to execute?**
  - One issue: Safety – How to get the CPU back? Or stop it from stepping on us? How about `cli/halt`?
  - Solution: Use protection mechanisms already in CPU
- **Run virtual machine's OS directly on CPU in unprivileged user mode**
  - “Trap and emulate” approach
  - Most instructions just work
  - Privileged instructions trap into monitor and run simulator on instruction
  - **Makes some assumptions about architecture**

# Virtualizing traps

- **What happens when an interrupt or trap occurs**
  - Like normal kernels: we trap into the monitor
- **What if the interrupt or trap should go to guest OS?**
  - Example: Page fault, illegal instruction, system call, interrupt
  - Re-start the guest OS simulating the trap
- **x86 example:**
  - Give CPU an IDT that vectors back to VMM
  - Look up trap vector in VM's "virtual" IDT
  - Push virtualized %cs, %eip, %eflags, on stack
  - Switch to virtualized privileged mode

# Virtualizing memory

- **Basic MMU functionality:**

- OS manages physical memory (0...MAX\_MEM)
- OS sets up page tables mapping VA  $\rightarrow$  PA
- CPU accesses to VA should go to PA (if paging off, PA = VA)
- Used for every instruction fetch, load, or store

- **Need to implement a virtual “physical memory”**

- Logically need additional level of indirection
- VM's *Guest VA*  $\rightarrow$  VM's *Guest PA*  $\rightarrow$  *Host PA*
- Note “Guest physical” memory no longer means hardware bits
- Hardware is host physical memory (a.k.a. machine memory)

- **Trick: Use hardware MMU to simulate virtual MMU**

- Point hardware at *shadow page table*
- Directly maps Guest VA  $\rightarrow$  Host PA

# Memory mapping summary



physical machine  
-----  
virtual machine



# Shadow page tables

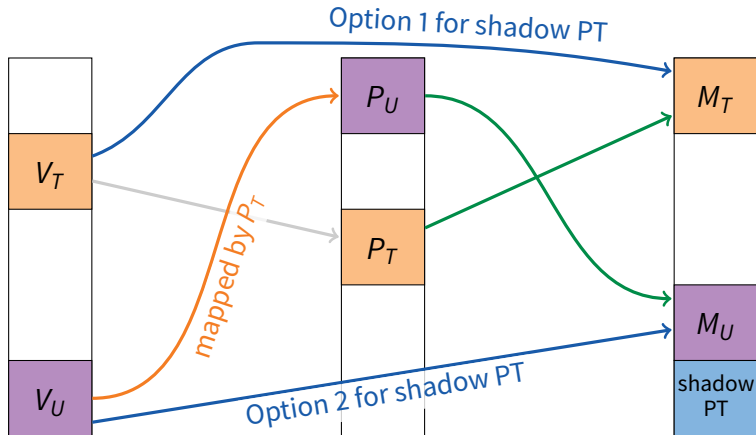
- **VMM responsible for maintaining *shadow* PT**
  - And for maintaining its consistency (including TLB flushes)
- **Shadow page tables are a cache**
  - Have *true page faults* when page not in VM's guest page table
  - Have *hidden page faults* when just misses in shadow page table
- **On a page fault, VMM must:**
  - Lookup guest VPN → guest PPN in guest's page table
  - Determine where guest PPN is in host physical memory
  - Insert guest VPN → host PPN mapping in shadow page table
  - Note: Monitor can demand-page the virtual machine
- **Uses hardware protection**



# Shadow PT issues

- **Hardware only ever sees shadow page table**
  - Guest OS only sees it's own VM page table, never shadow PT
- **Consider the following**
  - Guest OS has a page table  $T$  mapping  $V_U \rightarrow P_U$
  - $T$  itself resides at guest physical address  $P_T$
  - Another guest page table entry maps  $V_T \rightarrow P_T$  (e.g., in Pintos,  $V_T = P_T + \text{PHYS\_BASE}$ )
  - VMM stores  $P_U$  in host physical address  $M_U$  and  $P_T$  in  $M_T$
- **What can VMM put in shadow page table?**
  - Safe to map user page ( $V_U \rightarrow M_U$ ) *or* page table ( $V_T \rightarrow M_T$ )
- **Not safe to map both simultaneously!**
  - If OS writes to  $P_T$ , may make  $V_U \rightarrow M_U$  in shadow PT incorrect
  - If OS reads/writes  $V_U$ , may require accessed/dirty bits to be changed in  $P_T$  (hardware can only change shadow PT)

# Illustration



- **Option 1:** Page table accessible at  $V_T$ , but changes won't be reflected in shadow PT or TLB; access to  $V_U$  dangerous
- **Option 2:**  $V_U$  accessible, but hardware sets accessed/dirty bits only in shadow PT, not in guest PT at  $P_T/M_T$

# Tracing

- **VMM needs to get control on some memory accesses**
- **Guest OS changes previously used mapping in its PT**
  - Must intercept to invalidate stale mappings in shadow PT, TLB
  - Note: OS *should* use `invlpg` instruction, which would trap to VMM – but in practice many/most Oses are sloppy about this
- **Guest OS accesses page when its VM PT is accessible**
  - Accessed/dirty bits in VM PT may no longer be correct
  - Must intercept to fix up VM PT (or make VM PT inaccessible)
- **Solution: *Tracing***
  - To track page access, make VPN(s) invalid in shadow PT
  - If guest OS accesses page, will trap to VMM w. page fault
  - VMM can emulate the result of memory access & restart guest OS, just as an OS restarts a process after a page fault

# Tracing vs. hidden faults

- **Suppose VMM never allowed access to VM PTs?**
  - Every PTE access would incur the cost of a tracing fault
  - Very expensive when OS changes lots of PTEs
- **Suppose OS allowed access to *most* page tables (except very recently accessed regions)**
  - Now lots of hidden faults when accessing new region
  - Plus overhead to pre-compute accessed/dirty bits from shadow PT as page tables preemptively made valid in shadow PT
- **Makes for complex trade-offs**
  - But adaptive binary translation (later) can make this better

- **Types of communication**
  - Special instruction – `in/out`
  - Memory-mapped I/O (PIO)
  - Interrupts
  - DMA
- **Make `in/out` and PIO trap into monitor**
- **Use tracing for memory-mapped I/O**
- **Run simulation of I/O device**
  - Interrupt – Tell CPU simulator to generate interrupt
  - DMA – Copy data to/from physical memory of virtual machine

# CPU virtualization requirements

- **Need protection levels to run VMs and monitors**
- **All unsafe/privileged operations should trap**
  - Example: disable interrupt, access I/O dev, ...
  - x86 problem: `popf1` (different semantics in different rings)
- **Privilege level should not be visible to software**
  - Software shouldn't be able to query and find out it's in a VM
  - x86 problem: `movw %cs, %ax`
- **Trap should be transparent to software in VM**
  - Software in VM shouldn't be able to tell if instruction trapped
  - x86 problem: traps can destroy machine state (E.g., if internal segment register was out of sync with GDT)
- **See [\[Goldberg\]](#) for a discussion**

# Outline

- 1 Confining code with legacy OSes
- 2 Virtual machines
- 3 Implementing virtual machines
- 4 Binary translation
- 5 Hardware-assisted virtualization
- 6 Memory management optimizations

# Binary translation

- **Cannot directly execute guest OS kernel code on x86**
  - Can maybe execute most user code directly
  - But how to get good performance on kernel code?
- **Original VMware solution: binary translation**
  - Don't run slow instruction-by-instruction emulator
  - Instead, translate guest kernel code into code that runs in fully-privileged kernel mode, but acts safely<sup>2</sup>
- **Challenges:**
  - Don't know the difference between code and data (guest OS might include self-modifying code)
  - Translated code may not be the same size as original
  - Prevent translated code from messing with VMM memory
  - Performance, performance, performance, ...

---

<sup>2</sup>actually CPL 1, so that the VMM has its own exception stack



# VMware binary translator

- **VMware translates kernel dynamically (like a JIT)**
  - Start at guest `eip`
  - Accumulate up to 12 instructions until next control transfer
  - Translate into binary code that can run in VMM context
- **Most instructions translated identically**
  - E.g., regular `movl` instructions
- **Use segmentation to protect VMM memory**
  - VMM located in high virtual addresses
  - Segment registers “truncated” to block access to high VAs
  - `gs` segment not truncated; use it to access VMM data
  - Any guest use of `gs` (rare) can't be identically translated

Details/examples from [\[Adams & Agesen\]](#)

# Control transfer

- All branches/jumps require indirection

- **Original:**

```
isPrime: mov %edi, %ecx # %ecx = %edi (a)
         mov $2, %esi  # i = 2
         cmp %ecx, %esi # is i >= a?
         jge prime     # jump if yes
         ...
```

- **C source:**

```
int
isPrime (int a)
{
    for (int i = 2; i < a; i++) {
        if (a % i == 0)
            return 0;
    }
    return 1;
}
```

# Control transfer

- All branches/jumps require indirection

- **Original:**

```
isPrime: mov %edi, %ecx # %ecx = %edi (a)
        mov $2, %esi   # i = 2
        cmp %ecx, %esi # is i >= a?
        jge prime     # jump if yes
        ...
```

- **Translated:**

```
isPrime': mov %edi, %ecx # IDENT
          mov $2, %esi
          cmp %ecx, %esi
          jge [takenAddr] # JCC
          jmp [fallthrAddr]
```

- Brackets ([...]) indicate *continuations*

- First time jumped to, target untranslated; translate on demand
- Then fix up continuation to branch to translated code
- Can elide [fallthrAddr] if fallthrough next translated

# Non-identically translated code

- **PC-relative branches & Direct control flow**
  - Just compensate for output address of translator on target
  - Insignificant overhead
- **Indirect control flow**
  - E.g., jump through register (function pointer) or `ret`
  - Can't assume code is "normal" (e.g., must faithfully `ret` even if stack doesn't have return address)
  - Look up target address in hash table to see if already translated
  - "Single-digit percentage" overhead
- **Privileged instructions**
  - Appropriately modify VMM state
  - E.g., `cli`  $\implies$  `vcpu.flags.IF = 0`
  - Can be faster than original!

# Adaptive binary translation

- **One remaining source of overhead is tracing faults**
  - E.g., when modifying page table or descriptor table
- **Idea: Use binary translation to speed up**
  - E.g., translate write of PTE into write of guest & shadow PTE
  - Translate read of PTE to get accessed & dirty bits from shadow
- **Problem: Which instructions to translate?**
- **Solution: “innocent until proven guilty” model**
  - Initially always translate as much code identically as possible
  - Track number of tracing faults caused by an instruction
  - If high number, re-translate to non-identical code
  - May call out to interpreter, or just jump to new code

# Outline

- 1 Confining code with legacy OSes
- 2 Virtual machines
- 3 Implementing virtual machines
- 4 Binary translation
- 5 **Hardware-assisted virtualization**
- 6 Memory management optimizations

# Hardware-assisted virtualization

- **Both Intel and AMD now have hardware support**
  - Different mechanisms, similar concepts
  - This lecture covers AMD (see [\[AMD Vol 2\]](#), Ch. 15)
  - For Intel details, see [\[Intel Vol 3c\]](#)
- **VM-enabled CPUs support new *guest* mode**
  - This is separate from kernel/user modes in bits 0–1 of `%cs`
  - Less privileged than *host* mode (where VMM runs)
  - Some sensitive instructions trap in guest mode (e.g., `load %cr3`)
  - Hardware keeps shadow state for many things (e.g., `%eflags`)
- **Enter guest mode with `vmrun` instruction**
  - Loads state from hardware-defined 1-KiB VMCB data structure
- **Various events cause EXIT back to host mode**
  - On EXIT, hardware saves state back to VMCB

# VMCB control bits

- ***Intercept vector specifies what ops should cause EXIT***
  - One bit for each of %cr0-%cr15 to say trap on read
  - One bit for each of %cr0-%cr15 to say trap on write
  - 32 analogous bits for the debug registers (%dr0-%dr15)
  - 32 bits for whether to intercept exception vectors 0-31
  - Bits for various other events (e.g., NMI, SMI, ...)
  - Bit to intercept writes to sensitive bits of %cr0
  - 8 bits to intercept reads and writes of IDTR, GDTR, LDTR, TR
  - Bits to intercept `rdtsc`, `rdpmc`, `pushf`, `popf`, `vmrun`, `hlt`, `invlpg`, `int`, `iret`, `in/out` (to selected ports), ...
- **EXIT code and reason (e.g., which inst. caused EXIT)**
- **Other control values**
  - Pending virtual interrupt, event/exception injection



# Guest state saved in VMCB

- **Saved guest state**
  - Full segment registers (i.e., base, lim, attr, not just selectors)
  - Full GDTR, LDTR, IDTR, TR
  - Guest %cr3, %cr2, and other cr/dr registers
  - Guest %eip and %eflags (%rip & %rflags for 64-bit processors)
  - Guest %rax register
- **Entering/exiting VMM more expensive than syscall**
  - Have to save and restore large VM-state structure

# Hardware vs. Software virtualization

- **HW VM makes implementing VMM much easier**
  - Avoids implementing binary translation (BT)
- **Hardware VM is better at entering/exiting kernel**
  - E.g., Apache on Windows benchmark: one address space, lots of syscalls, hardware VM does better [\[Adams\]](#)
  - Apache on Linux w. many address spaces: lots of context switches, tracing faults, etc., Software faster [\[Adams\]](#)
- **Fork with copy-on-write bad for both HW & BT**
  - [\[Adams\]](#) reports fork benchmark where BT-based virtualization 37× and HW-based 106× slower than native!
- **Today, CPUs support *nested paging* (a.k.a. EPT on intel)**
  - Eliminates shadow PT & tracing faults, simplifies VMM
  - Guests can now manipulate %cr3 w/o VM EXIT
  - But dramatically increases cost of TLB misses

# Outline

- 1 Confining code with legacy OSes
- 2 Virtual machines
- 3 Implementing virtual machines
- 4 Binary translation
- 5 Hardware-assisted virtualization
- 6 **Memory management optimizations**

# ESX memory management [Waldspurger]

- **Virtual machines see virtualized physical memory**
  - Can let VMs use more “physical” memory than in machine
- **How to apportion memory between machines?**
- **VMware ESX has three parameters per VM:**
  - **min** – Don’t bother running w/o this much machine memory
  - **max** – Amount of guest physical memory VM OS thinks exists
  - **share** – How much memory to give VM relative to other VMs
- **Straw man: Allocate based on share, use LRU paging**
  - OS already uses LRU  $\implies$  double paging
  - OS will re-cycle whatever “physical” page VMM just paged out
  - So better to do random eviction
- **Next: 3 cool memory management tricks**

# Reclaiming pages

- **Normally OS just uses all available memory**
  - But some memory much more important than other memory
  - E.g., buffer cache may contain old, clean buffers; OS won't discard if doesn't need memory... but VMM may need memory
- **Idea: Have guest OS return memory to VMM**
  - Then VMM doesn't have to page memory to disk
- **ESX trick: Balloon driver**
  - Special pseudo-device driver in supported guest OS kernels
  - Communicates with VMM through special interface
  - When VMM needs memory, allocates many pages in guest OS
  - Balloon driver tells VMM to re-cycle its private pages

# Sharing pages across VMs

- **Often run many VMs with same OS, programs**
  - Will result in many host physical pages containing same data
- **Idea: Use 1 host physical page for all copies of guest physical page (in any virtual machine)**
- **Keep big hash table mapping: Hash(contents)→info**
  - If host physical page mapped once, info is VM/PPN where mapped. In that case, Hash is only a hint, as page may have changed
  - If machine page mapped copy-on-write as multiple physical pages, info is just reference count
- **Scan OS pages randomly to populate hash table**
- **Always try sharing a page before paging it out**

# Idle memory tax

- **Need machine page? What VM to take it from?**
- **Normal proportional share scheme**
  - Reclaim from VM with lowest “shares-to-pages” ( $S/P$ ) ratio
  - If  $A$  &  $B$  both have  $S = 1$ , reclaim from larger VM
  - If  $A$  has twice  $B$ 's share, can use twice the machine memory
- **High-priority VMs might get more memory than needed**
- **Solution: Idle-memory tax**
  - Use statistical sampling to determine a VM's % idle memory (randomly invalidate pages & count the number faulted back)
  - Instead of  $S/P$ , reclaim from VM with lowest  $S / (P(f + k(1 - f)))$ .  
 $f$  = fraction of non-idle pages;  $k$  = “idle page cost” parameter.
  - Be conservative & overestimate  $f$  to respect priorities  
( $f$  is max of slow, fast, and recent memory usage samples)