Administivia

• Last project due Thursday
  - Must submit something by midnight even if you get extension
  - Then we’ll have something to grade if you don’t make extension deadline

• Final Exam
  - Wednesday March 18, 12:15-3:15pm
  - Open book, open notes, just like midterm
  - Covers material from all 19 lectures
    ...possibly including topics already on the midterm

• Televised final review session Friday
  - Please come and bring any questions you might have on lecture material

Confining code with legacy OSes

• Often want to confine code on legacy OSes

• Analogy: Firewalls

  Hopelessly Insecure Server

  - Your machine runs hopelessly insecure software
  - Can’t fix it—no source or too complicated
  - Can reason about network traffic

• Similarly block unrusted code within a machine
  - By limiting what it 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 privs 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, so chrooting to a new directory can put you above your new root

• Create devices that let you access raw disk

• Send signals to or ptrace non-chrooted processes

• Create setuid program for non-chrooted proc. to run

• Bind privileged ports, mess with clock, reboot, etc.

• Problem: chroot was not originally intended for security
  - FreeBSD jail, Linux vserver have tried to address problems

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/recev
  - 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), ...
Review: What is an OS

- OS is software between applications and reality
  - Abstracts hardware and makes portable
  - Makes finite into (near) infinite
  - Provides protection

What if...

- The process abstraction looked just like hardware?

How is a process different from HW?

**Process**
- CPU – Non-Privileged registers and instructions.
- Memory – Virtual memory.
- Exceptions – signals, errors.

**Hardware**
- CPU – All registers and instructions.
- Memory – Both virtual and physical memory, memory management, TLB/page tables, etc.
- Exceptions – Trap architecture, interrupts, etc.
- I/O – I/O devices accessed using programmed I/O, DMA, interrupts.

Complete Machine Simulation

- Naïve approach (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!
    - 100x slowdown makes it not too useful
    - CPU/Memory – 100x CPU/MMU simulation
    - I/O Device – < 2× slowdown.
  - 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?
  - Safety – How going to get CPU back? Or stop it from stepping on us? How about cli/halt?
  - Answer: Use protection mechanism
- Run virtual machine's OS directly on CPU at non-privileged level
  - “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 the VM?
  - Example: Page fault, illegal instruction, system call, interrupt
  - Re-start the guest OS simulating the trap
- x86 example:
  - Lookup trap vector in VM’s 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→PA
  - CPU accesses to VA should go to PA (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 VA → VM’s PA → machine address
- **Trick: Use hardware MMU to simulate virtual MMU**
  - Can be folded into page tables: VA—machine address

Tracing

- **VMM must track changes to some memory locations**
  - E.g., when guest OS changes its page tables
  - Must invalidate stale mappings in shadow page tables
- **VMM must track access to some memory locations**
  - E.g., must return appropriate dirty bits in VM PTEs
- **Solution: Tracing – mark pages protected**
  - If guest OS accesses protected page, will trap to VMM
  - Emulate the result of memory access & continue
- **Can allow guest access to VM PTE or HW use of shadow PTE, but not both simultaneously**
  - Never allow direct access to page tables ⇒ lots of tracing faults
  - Allow most access to page tables ⇒ lots of hidden faults
  - Context-switch overhead to pre-compute accessed/dirty bits
  - Result: complex performance trade-off

Virtual Machine Monitor

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

I/O device virtualization

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

MMU Virtualization

- **Trick: Monitor keeps shadow of VM’s page table**
  - Contains mapping from VM’s VA → machine physical memory
- **Can treat shadow page tables as a cache**
  - Have true page faults when a page not in VM’s page table
  - Have hidden page faults when just misses in shadow page table
- **On a page fault, VMM must:**
  - Lookup in VM’s page table mapping from VPN to PPN
  - Determine where PPN is in machine memory (MPN)
  - Insert VPN—MPN mapping in shadow page table
  - Note: Monitor can demand page the virtual machine
- **Uses hardware protection**
  - Monitor never maps itself into VM’s page table
  - never maps other VM’s memory in VM’s page table

Old idea from the 1960s

- IBM VM/370 – A VMM for IBM mainframe
  - Multiplex multiple OS environments on expensive hardware
  - Desirable when few machine around
- **Interest died out in the 1980s and 1990s**
  - Hardware got cheap
  - Compare Windows NT vs. N DOS machines
- **Interesting again today**
  - Different problems today – software management
  - VMM attributes still relevant
Virtual Machine Monitor attributes

- **Software compatibility**
  - Runs pretty much all software
  - Trick: Make virtual hardware match real hardware

- **Low overheads/High performance**
  - Near “raw” machine performance
  - Direct execution of CPU/MMU

- **Complete isolation**
  - Total data isolation between virtual machines
  - Use hardware protection

- **Encapsulation**
  - Virtual machines are not tied to physical machines
  - Checkpoint/Migration

Different thought about OSes

- **Installing software on hardware is broken**
  - Tight coupling of OS and applications to hardware creates management problems

- **Want to subdivide OS:**
  - Hardware drivers
  - Hardware management
  - System support software

- **Turn OSes into normal software that can be managed**

Backward compatibility with VMMs

- **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: Not all WinNT applications run on WinXP or XP on Vista**
  - In spite of a huge compatibility effort
  - Given the number of applications that ran on WinNT, practically anything would break something
  
  \[
  \text{if (OS} \not= \text{WinNT)} \ldots
  \]

- **Solution: Use a VMM to run both WinNT and WinXP**
  - Obvious for OS migration as well: Windows → Linux

Isolation & Multi-level security

- **Traditional tension: Security vs. Usability**
  - Secure systems tend not to be that usable
  - Flexible systems are not that secure

- **Additional information assurance requirement**
  - Data cannot flow between networks of different classification

- **Solution: Run two VMs:**
  - Classified VM
  - Internet VM

- **Use isolation property to isolate two VMs**
  - VMM has control of the information flow between machines
  - Declassifier mechanism

Logical partitioning of servers

- **Run multiple servers on same box**
  - Ability to give away less than one machine
  - Modern CPUs more powerful than most services need
  - 0.10U rack space machine – Better power, cooling, floor space, etc.
  - Server consolidation trend: \( N \) machines → 1 real machine

- **Isolation of environments**
  - Printer server doesn’t take down Exchange server
  - Compromise of one VM can’t get at data of others\(^a\)

- **Resource management**
  - Provide service-level agreements

- **Heterogeneous environments**
  - Linux, FreeBSD, Windows, etc.

Example: VMMs for IDS

- **Problem Area: Intrusion Detection Systems (IDS)**

- **Trade-offs**
  - Host-based IDS (HIDS):
    - Good visibility to catch intruder
    - Weak isolation from intruder disabling/masking IDS

  - Network-based IDS (NIDS):
    - Good isolation from attack from intruder
    - Weak visibility can allow intruder to slip by unnoticed

- **Would like visibility of HIDS with isolation of NIDS**
  - Idea: Do it in the virtual machine monitor

---
\(^a\)though in practice not so simple because of side-channel attacks
VMM-based IDS

- **Strong isolation**
  - VMM isolate software in VM from VMM.
  - Comprise OS in VM can’t disable IDS in VMM.
- **Introspection – Peer inside at software running in VM**
  - VMM can see: Physical memory, registers, I/O device state, etc.
  - Signature scan of memory
    Look through physical memory for patterns or signs of break-in
- **Interposition – Modify VM abstraction to enhance security**
  - Memory Access Enforcer (Interpose on page protection)
  - NIC Access Enforcer (Interpose on virtual network device)

Collective Project: A Compute Utility

- **Distributed system where all software runs in VMs**
  - Research with Prof. Monica Lam and students
  - Technology transfer to moka5.com
- **Virtual Appliance abstraction**
  - x86 virtual machine
  - Target specialized environment (e.g. program development)
  - Store in a centralized persistent storage repository
  - Cached on the machine were virtual appliances run
- **Target benefits**
  - System administration: Centralize and amortize administration of a virtual appliance
  - Mobility: Computing environment follows user around

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: mov %cs, %ax
- **Trap should be transparent to software in VM**
  - Software in VM should 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)
- **Lost art with modern hardware**

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?
- **VMware solution: binary translation**
  - Don’t run slow instruction-by-instruction emulator
  - Instead, translate guest kernel code into code that run in fully-privileged monitor mode
- **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, …

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 %ecx, %edi ; %ecx = %edi (a)
  - mov %esi, $2 ; i = 2
  - cmp %esi, %ecx ; is i >= a?
  - jge prime ; jump if yes
- **Translated:**
  - isPrime': mov %ecx, %edi ; IDENT
  - cmp %esi, %ecx
  - 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 =⇒ 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 PTE read 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

ESX mem. mgmt. [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 “physical” memory VM OS thinks exists
  - share – How much mem. to give VM relative to other VMs
- **Straw man: Allocate based on share, use LRU paging**
  - OS already uses LRU ⇒ 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

- **Idea: Have guest OS return memory to VMM**
  - Then VMM doesn’t have to page memory to disk
- **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
- **ESX trick: Baloon 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
  - Baloon driver tells VMM to re-cycle it’s private pages

Sharing pages across VMs

- **Often run many VMs with same OS, programs**
  - Will result in many machine pages containing same data
- **Idea: Use 1 machine page for all copies of phys. page**
- **Keep big hash table mapping:** Hash(contents)→info
  - If machine 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 may consume memory than needed**
- **Solution: Idle-memory tax**
  - Use statistical sampling to determine a VM’s % idle memory
  - Reclaim from VM w. lowest S/P, reclaim from larger VM
  - Instead of S/P, reclaim from VM w. 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)