Administrivia

• **Guest lecture Thursday**
  - Amit Singh (Google): Real-world Operating Systems
  - Please attend lecture if at all possible

• **Last project due Thursday**
  - No extensions unless *all* non-SCPD group members at lecture
  - If staff grants you extension, means only if you attend lecture
  - We will have a more stringent enforcement mechanism

• **Final Exam**
  - Thursday March 18, 12:15-3:15pm
  - Open book, covers all 19 lectures
    (possibly including topics already on the midterm)

• **Televised final review session Friday**
  - Bring questions on lecture material
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

• 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/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
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.
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 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
VMM benefits

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

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

- **Isolation**
  - Seemingly 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
OS backwards compatibility

- **Backward compatibility is bane of new OSes**
  - Huge effort required 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 any change would break something
    
    ```
    if (OS == WinNT) ...
    ```

- **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

• Note: Bad idea w. today’s VMMs, not secure enough
Logical partitioning of servers

- Run multiple servers on same box (e.g., Amazon EC2)
  - Ability to give away less than one machine
    Modern CPUs more powerful than most services need
  - 0.10U rack space machine – less power, cooling, space, etc.
  - Server consolidation trend: $N$ machines $\rightarrow$ 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.

$^a$though in practice not so simple because of side-channel attacks [Ristenpart]
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
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 [Chandra]

- 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 where virtual appliances run

- Target benefits
  - System administration: Centralize and amortize administration of a virtual appliance
  - Mobility: Computing environment follows user around
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!
  - 100x slowdown makes it not too useful
  - CPU/Memory – 100x CPU/MMU simulation
  - I/O Device – < 2x 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 guest OS?**
  - 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**
  - Note “physical memory” no longer mans hardware bits –
    machine memory is hardware bits

- **Trick: Use hardware MMU to simulate virtual MMU**
  - Can be folded into page tables: VA→machine address
Shadow page tables

- Monitor keeps *shadow* of VM’s page table
  - Shadow PT is map from VA → machine address

- 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 VPN → PPN in VM’s (guest OS’s) page table
  - 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 VMs’s memory in VM’s page table
Shadow PT issues

- Hardware only ever sees shadow page page table
  - Guest OS only sees it’s own VM page table, never shadow PT

- Consider the following
  - OS has a page table $T$ mapping $V_U \rightarrow P_U$
  - $T$ itself resides at physical address $P_T$
  - Another page table maps $V_T \rightarrow P_T$
  - VMM stores $P_U$ in machine address $M_U$ and $P_T$ in $M_T$

- What can VMM put in shadow page table?
  - Safe to map $V_U \rightarrow M_U$ or $V_T \rightarrow M_T$

- 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)
Tracing

- VMM needs to get control on some memory accesses
- Guest OS changes VM page table
  - OS should use invlpg instruction, which would trap to VMM –
    but in practice many/most OSes are sloppy about this
  - Must invalidate stale mapping in shadow page table
- Guest OS accesses page when VM PT accessible
  - Accessed/dirty bits in VM PT will no longer be correct
  - Must make VM PT inaccessible in shadow PT
- 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 page tables when switching between VMs

- Makes for complex trade-offs
  - But adaptive binary translation (later) can make this better
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
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: popfl (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 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)
- See [Goldberg] for a discussion
  - 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 runs 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, …

\(^a\) 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 %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  
mov %esi, $2  
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 though 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 $\Rightarrow vcpu.flags.IF = 0$
  - 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 its 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 might get more mem. 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 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)
Final thoughts

- You are all now operating systems experts
- Use this knowledge to build better applications
  - Sometimes need to coax right behavior out of kernel
  - Should be much easier now that you know what’s going on
- Syscall interface can be an innovation barrier
  - Much harder to change kernel than user code
  - Other barriers include standardized net. protocols, servers
  - Get these wrong and many people will suffer
- Some of you will go on to design interfaces that many people are later subjected to
  - Strive to achieve both simplicity and flexibility for users
How to learn more about OSes

• Take CS240 – Advanced Topics in Operating Systems
  - Class will bring you up to speed on OS research
  - Read & discuss 18–25 research papers
  - By the end, should be ready to do OS research

• Get involved in research!

• Lot’s of interesting OS work at Stanford
  - Rosenblum – launched the virtual machine resurgence
  - Lam – collective system, software for mobile devices
  - Levis – seminal work on sensor nets & power management
  - Engler – tools to find OS bugs automatically
  - Boneh/Mitchell – lots of practical OS security work
  - Mazières – done multiple new OSes, FSes, and distributed systems. Applying OS ideas to browser, language security.