Virtual Machines

W4118 Operating Systems I

columbia-os.github.io

Credits to David Mazières

Confining code

- Often want to confine code on legacy OSes.

- Analogy: **Firewalls**
  - Your machine runs potentially insecure software
  - Can’t fix it – no source code or too complicated
  - Can reason about network traffic
- Can we similarly block untrusted code within a machine?
  - Have the OS Limit what the code can interact with

Solution: System call interposition

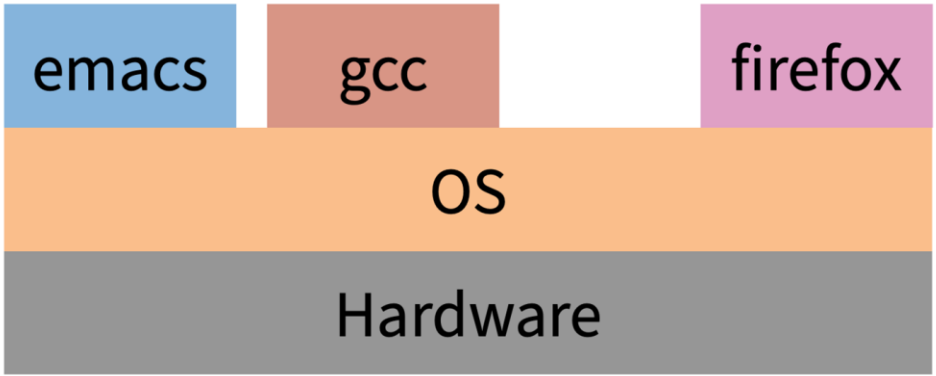
- Why not use ptrace or other debugging facilities to control untrusted programs?
- Almost any “damage” must result from system call
  - delete files → unlink
  - overwrite files → open/write
  - attack over network → socket/bind/connect/send/recv
  - leak private data → open/read/socket/connect/write . . .
- 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

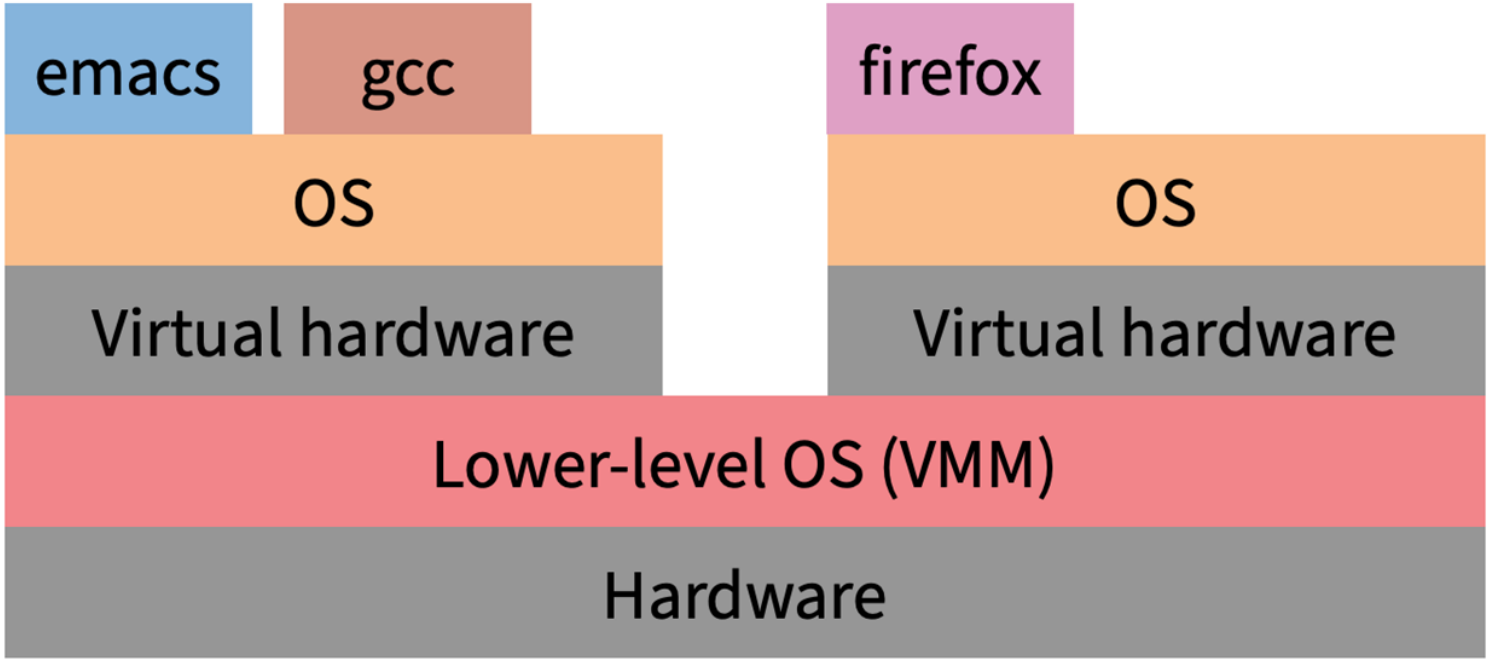
- Hard to know exact implications of a system call
- Indirect paths to resources
- Race conditions
- and many more…

Review: What is an OS



- OS is the software between applications and hardware/the world
  - Abstract the hardware to make the 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?

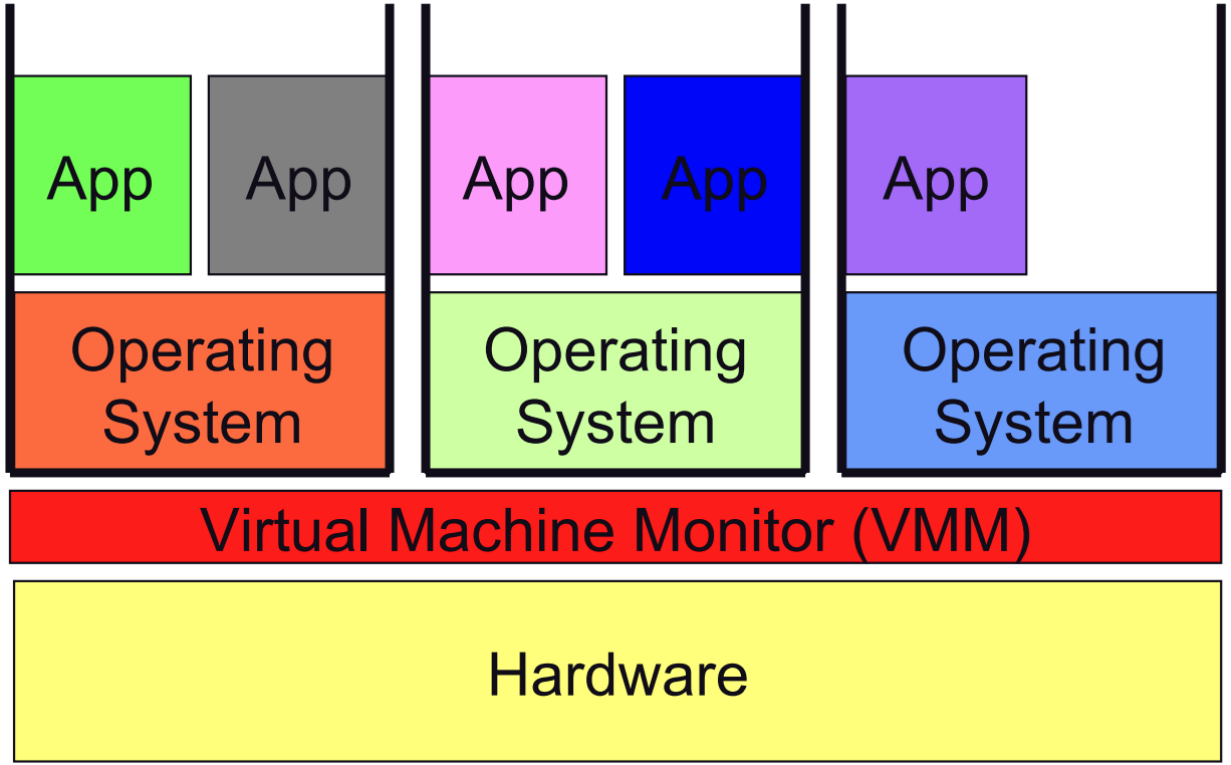


How do they differ?

Process	Hardware
Non-privileged registers and instructions	All registers and instructions
Virtual memory	Both virtual and physical memory, MMU functions, TLB/page tables, etc.
Errors, signals	Trap architecture, interrupts
File system, directories, files	I/O devices accessed using programmed I/O, DMA, interrupts

Virtual Machine Monitor

Thin layer of software that virtualizes the hardware



Old idea from the 1960s

- **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 buy a desktop and put windows

- **Today, VMs are used everywhere**
  - Used to solve different problems
  - VMM attributes more relevant than ever

VMM benefits

- **Software compatibility**
  - VMMs can run pretty much all software
- **Can get low overheads/high performance**
  - New raw machine performance for many workloads
  - With tricks and/or right hardware 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

Logical partitioning of servers

- **Run multiple servers on same box (e.g., AWS EC2)**
  - Modern CPUs more powerful than most services need
  - VMs let you give away less than one machine
- **Isolation of environments**
  - Printer server doesn’t take down email server
  - Compromise of VMs can’t get at data of other
- **Resource management**
  - Provide service-level agreements
- **Heterogeneous environments**
  - Linux, FreeBSD, Windows, etc.

Naive Approach: Complete Machine Simulation

- **Simplest VMM approach**
- **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 - < 2x slowdown
  - 100x slowdown makes it not too useful
- **Need faster ways of emulating CPU/MMU**

Virtualizing the CPU

- **Observation: Most instruction are the same regardless of processor privilege level**

- **Why not just give instructions to CPU to execute?**
  - One issue: Safety - How to get the CPU back? Or stop if from stepping on us?
  - Solution: Use protection mechanisms already in CPU

- **Run virtual machine's OS directly on CPU in unprivileged user mode**
  - Trap and emulate
  - 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 guest OS?**
  - Example: Page fault, illegal instruction, system call, interrupt
  - Jump to the guest OS simulating the trap

- **x86 example:**
  - Give the CPU an IDT that vectors back to VMM
  - Look up trap vector in VM's "virtual" IDT
  - Push virtualized registers/arguments , on stack
  - Switch to virtualized privileged mode

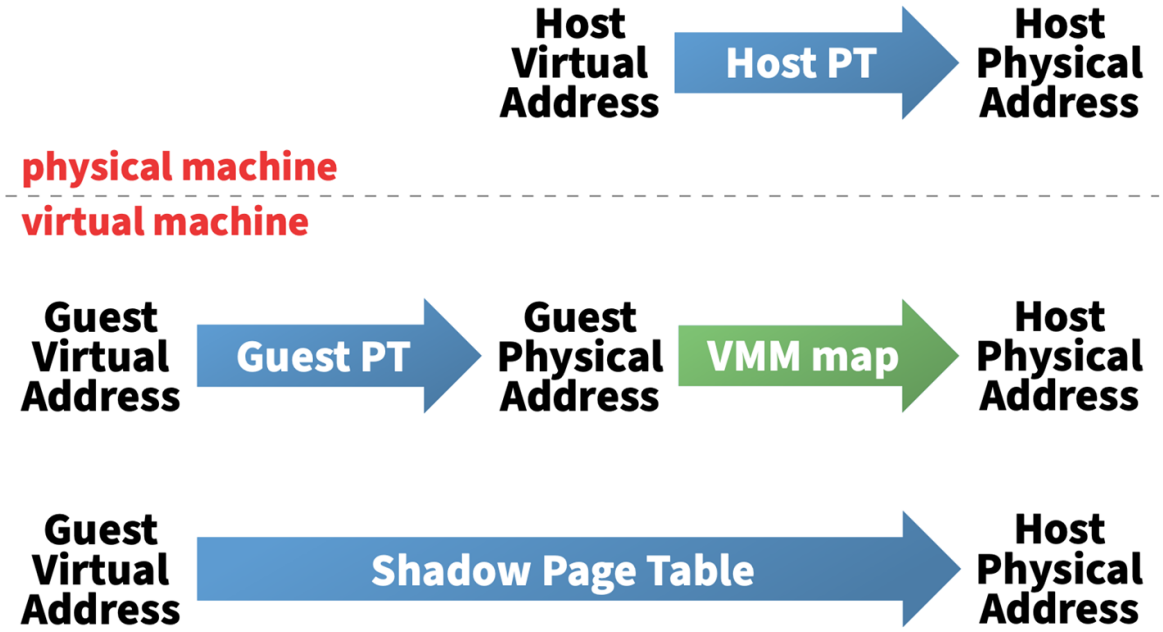
Virtualizing memory

- **Basic MMU functionality:**
  - OS managers physical memory (0…MAX_MEM)
  - OS sets up pages tables mapping VA→PA
  - CPU accesses to VA should go to PA
  - Used for every instruction fetch,load,store

- **Need to implement a virtual "physical memory"**
  - Logically: another level of indirection
  - VM's Guest VA → VM's Guest PA → Host PA
  - "Guest physical" memory does not mean hardware bits
  - Hardware is the host physical memory

- **Trick: Use hardware MMU to simulate virtual MMU**
  - Point hardware at *shadow page table*
  - Directly maps Guest VA → Host PA

Memory mapping summary



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 fault when page not in VM's guest page table
  - Have hidden page fault when just misses in shadow page table

- **On a page fault, VMM must:**
  - Lookup guest VPN→guest PPN in guest's page table
  - If true page fault, emulate page fault in guest OS
  - Otherwise, determine where guest PPN is in host physical memory
  - Insert guest VPN→host PPN mapping in shadow page table

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 Vu→Pu
  - T itself resides at guest physical address Pt
  - Another guest page table entry maps Vt→Pt
  - VMM stores Pu in host physical address Mu and Pt in Mt
- **What can VMM put in the shadow page table?**
  - Safe to map user page (Vu→Mu) **or** page table (Vt→Mt)
- **Not safe to map both simultaneously**
  - If OS writes to Pt, may make Vu→Mu in shadow PT incorrect
  - If OS reads/writes Vu, may require accessed/dirty bits to be changed in Pt (hardware can only change shadow PT)

Hardware support for virtualization

- **Intel/AMD/ARM now have hardware support**
  - Different mechanisms, similar concepts except for ARM
- **VM-enabled CPUs support new guest mode**
- **Enter guest mode with `vmrun` instruction**
  - Loads state from hardware-defined 1KB VMCB data structure
- **Various events cause EXIT back to host mode**
  - On EXIT, hardware save state back to VMCB
- **Entering/exiting VMM more expensive than syscall**
  - Have to save and restore large VM-state structure
- **CPUs support nested paging**
  - Eliminates shadow PT, simplifies VMM
  - Guests can manipulate base of page table w/o VM exit
  - Dramatically increases cost of TLB misses

Hardware vs. Software virtualization

- **HW make implementing VMM much easier**

- **Hardware VM is better at entering/exiting kernel**
  - Apache on Windows benchmark: one address space, lots of syscalls, hardware VM does better
  - Apache on Linux w. many address spaces: lots of context switches, fault, etc., software faster

- **Fork with copy-on-write bad**
  - 106x slower than native!

QEMU and KVM

- QEMU (Quick Emulator): emulates OR virtualizes a machine
- KVM allows linux to function as a hypervisor and acts as CPU accelerator
  - Runs guest until it has an vm_exit (trap, device access)
  - Provides fast device emulation via virtio