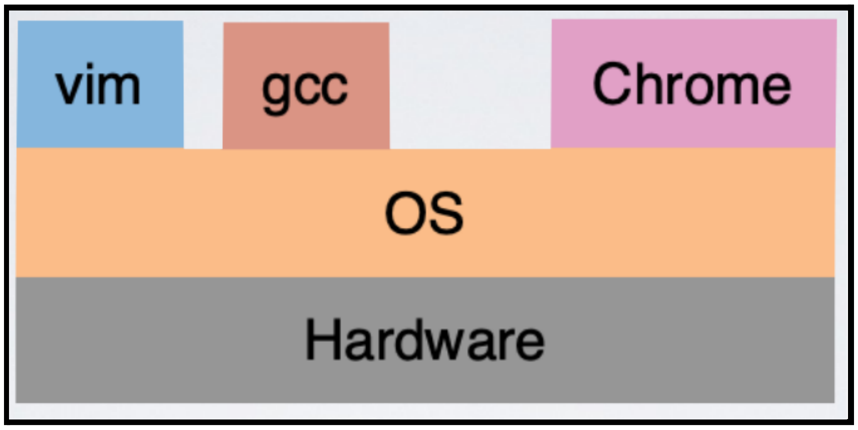
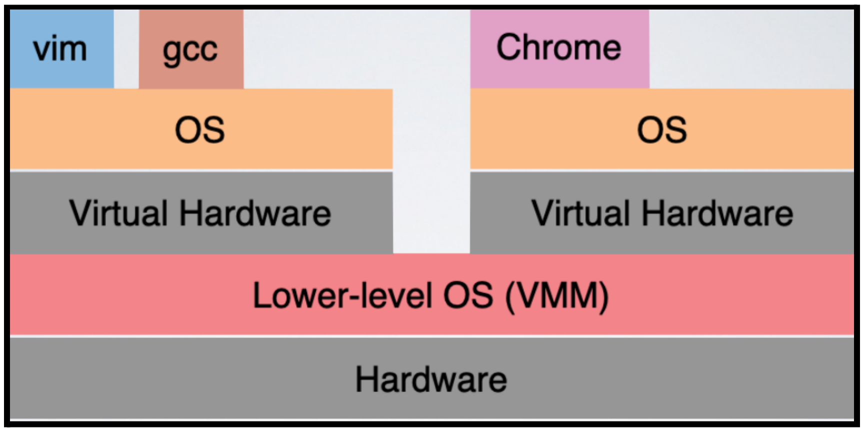
Lecture Notes:

- **Virtualization:**
- What is an OS:
    - An OS is software between applications and hardware.
    - It abstracts the hardware to make applications portable.
    - It makes finite resources (such as memory, number of CPU cores) appear much larger and fully dedicated to running one application.
    - It protects processes and users from one another.
    - I.e.



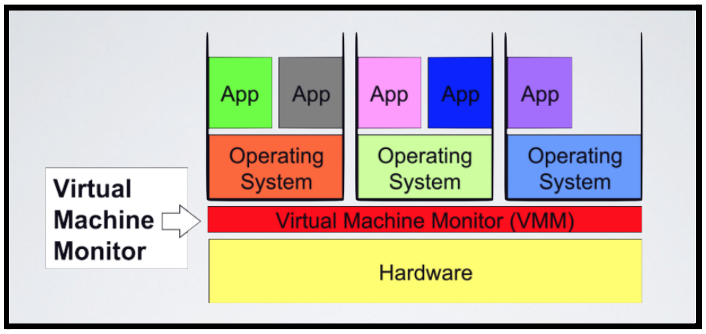
    - However, what if the process abstraction looked just like hardware? I.e.



- How do process abstraction & hardware 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 and signals	Trap, interrupts
File systems, directories, files, raw devices	I/O devices accessed through programmed I/O, DMA, interrupts

- VMM - Virtual Machine Monitor:
    - The VMM is a thin layer of software that virtualizes the hardware.
    - It exports a virtual machine abstraction that looks like the hardware.
    - It provides the illusion that software has full control over the hardware.
    - The **Virtual Machine Monitor/Hypervisor** runs multiple OSes simultaneously on the same physical machine.
    - I.e.

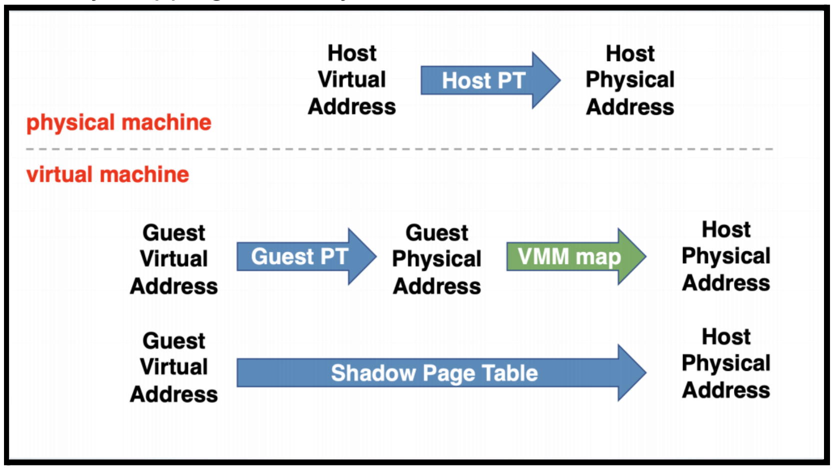


- **Motivations:**
- Virtualization was an old idea, starting from the 70's. At that time, computers were very big (they took up a room). However, in the 80's computers began getting cheaper, so the idea died out. The 80's were the era of personal computers. However, the idea was revived by the Disco work led by Mendel Rosenblum, who later led the foundation of VMware.
- Nowadays, VMs are used everywhere.
- They got popularized by cloud computing and are used to solve different problems.
- VMMs are a hot topic in industry and academia.
- Why we use virtualization:
    1. **Software compatibility:** VMMs can run pretty much all software.
    2. **Resource utilization:** Machines today are powerful, and we want to multiplex their hardware.
    3. **Isolation:** Seemingly total data isolation between virtual machines.
    4. **Encapsulation:** Virtual machines are not tied to physical machines.
    5. **Many other cool applications:** Debugging, emulation, security, speculation, fault tolerance, etc.
- Backward compatibility is the bane of new OSes as it requires huge efforts to innovate but not break. Security considerations may make it impossible in practice.
- Logical partitioning of servers:
    - **Run multiple servers on same box (e.g. Amazon EC2):**
        - Modern CPUs are much more powerful than most services need.
        - VMs let you give away less than one machine for running a service.
        - Server consolidation: N machines → 1 real machine.
        - Consolidation leads to cost savings (less power, cooling, management, etc).
    - **Isolation of environments:**
        - Safety - A printer server failure doesn't take down Exchange server.
        - Security - The compromise of one VM cannot get the data of others.
    - **Resource management:**
        - Provide service-level agreements.
    - **Heterogeneous environments:**
        - Linux, FreeBSD, Windows, etc.

- **Implementation:**
- Implementing VMMs - requirements:
    - **Fidelity:** OSes and applications work the same without modification (although we may modify the OS a bit).
    - **Isolation:** VMM protects resources and VMs from each other.
    - **Performance:** VMM is another layer of software and therefore there is some overhead. We want to minimize the overhead.
- What needs to be virtualized:
    - Exactly what you would expect:
        - CPU
        - Events (hardware and software interrupts)
        - Memory
        - I/O devices
    - Isn't this just duplicating OS functionality in a VMM?:
        - Yes: Approaches will be similar to what we do with OSes. However, they will be simpler in functionality, since VMMs are much smaller than OSes.
        - No: It implements a different abstraction. Hardware interface vs. OS interface
- Approach 1 - Complete machine simulation:
    - Simplest VMM approach.
    - Used by Bochs.
    - The idea is to build a simulation of all the hardware:
        - CPU – A loop that fetches each instruction, decodes it, and simulates its effect on the machine state (no direct execution).
        - 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, etc.
    - However, this is too slow.
      CPU/Memory – 100x slowdown.
      I/O Device – 2x slowdown.
    - We need faster ways of emulating the CPU/MMU.
- Approach 2 - Virtualizing the CPU/MMU:
    - Observations - Most instructions are the same regardless of processor privileged level.
      E.g. incl %eax.
    - Why not just give instructions to the CPU to execute?
    - The problem is safety. How can we prevent privileged instructions from interfering with the hypervisor and other OSes?
    - The solution is to use the protection mechanisms already in the CPU.
    - I.e. "Trap and emulate" approach:
        - Run the virtual machine's OS directly on the CPU in unprivileged user mode.
        - Privileged instructions trap into monitor and run simulator on instruction.
- Virtualizing interrupts:
    - The OS assumes to be in control of interrupts via the interrupt table.
    - So what happens when an interrupt or trap occurs in a virtual environment? ➡ The VMM handles the interrupt (in kernel mode) using the "virtual" interrupt handler table of the running OS.
    - Some interrupts can be shadowed.

- Virtualizing memory:
    - The OS assumes to be in full control over memory via the page table.
    - But VMM partitions memory among VMs:
        - VMM needs to assign hardware pages to VMs.
        - VMM needs to control mappings for isolation.
          Cannot allow an OS to map a virtual page to any hardware page.
          The OS can only map to a hardware page given to it by the VMM.
    - Hardware-managed TLBs make this difficult. When the TLB misses, the hardware automatically walks the page tables in memory. As a result, the VMM needs to control access by OS to page tables.
    - One way - direct mapping:
        - The VMM uses the page tables that a guest OS creates (direct mapping by MMU).
        - The VMM validates all updates to page tables by guest OS. The OS can read page tables without modification but the VMM needs to check all page table entry writes to ensure that the virtual-to-physical mapping is valid.
        - This requires the OS to patch updates to the page table.
    - Another way - level of indirection:
        - Three abstractions of memory:
            1. Machine - Actual hardware memory (16 GB of DRAM).
            2. Physical - Abstraction of hardware memory managed by OS. If a VMM allocates 512 MB to a VM, the OS thinks the computer has 512 MB of contiguous physical memory (underlying machine memory may be discontiguous).
            3. Virtual - Virtual address spaces (similar to virtual memory). The standard is 2^{32} or 2^{64} address space.
    - Shadow page tables:
        - The VMM creates and manages page tables that map virtual pages directly to machine pages. These tables are loaded into the MMU on a context switch.
        - The VMM page tables are the shadow page tables.
        - The VMM needs to keep its virtual to machine tables consistent with changes made by the OS to its virtual to physical tables.
            - VMM maps OS page tables as read-only (i.e., write-protected).
            - When the OS writes to page tables, trap to VMM.
            - Memory tracing: VMM applies write to shadow table and OS table and returns.
            - Memory-mapped devices must be protected for both read-protected and write-protected.

- Memory mapping summary:



- Memory Allocation:
  - VMMs tend to have simple hardware memory allocation policies:
    - Static - VM gets 512 MB of hardware memory for life.
    - No dynamic adjustment based on load. OSes are not designed to handle changes in physical memory.
    - No swapping to disk.
  - More sophistication - overcommit with balloon driver:
    - Balloon driver runs inside the OS to consume hardware pages and steals from virtual memory and file buffer cache (balloon grows).
    - Gives hardware pages to other VMs (those balloons shrink).
- Virtualizing I/O:
  - OSes can no longer interact directly with I/O devices.
    1. Make an in/out trap into VMM and use tracing for memory-mapped I/O.
    2. Run simulation of I/O device.
       Interrupt – tell CPU simulator to generate interrupt.
       DMA – copy data to/from physical memory of virtual machine.