Hardware Virtualization

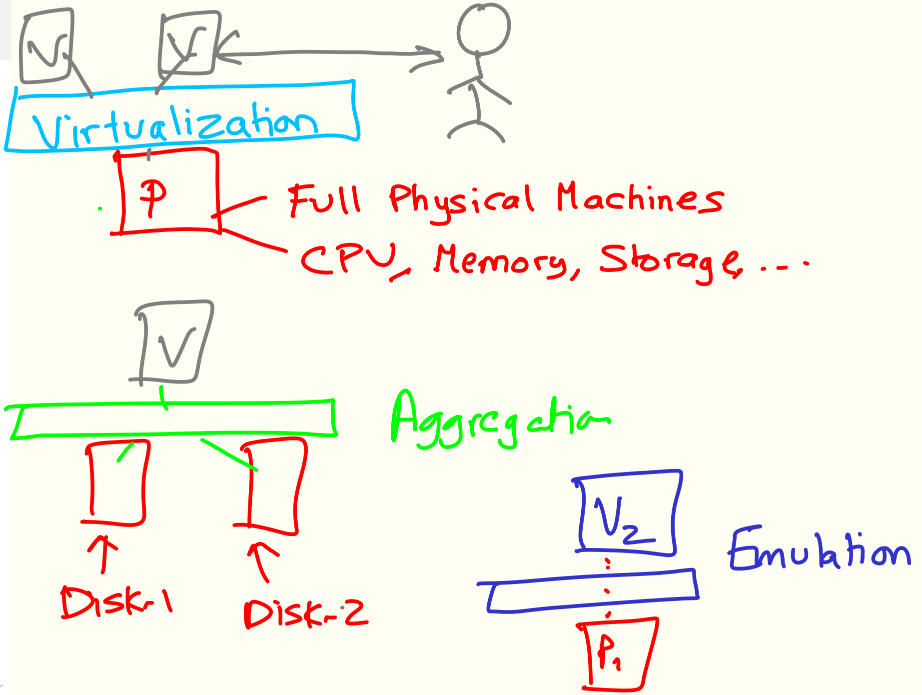
E-516 Cloud Computing

Virtualization

- Virtualization is a vital technique employed throughout the OS
- Given a physical resource, expose a virtual resource through layering and enforced modularity
- Users of the virtual resource (usually) cannot tell the difference

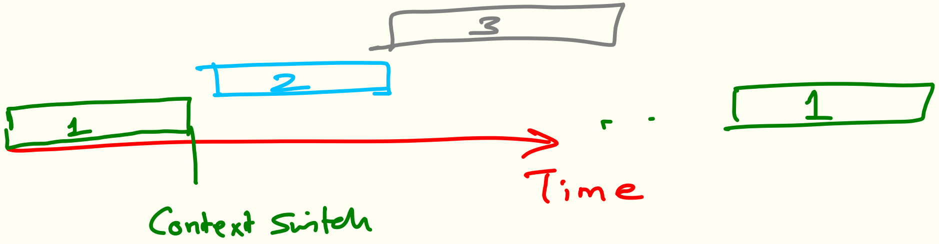
Different forms:

- Multiplexing: Expose many virtual resources
- Aggregation: Combine many physical resources [RAID, Memory]
- Emulation: Provide a *different* virtual resource



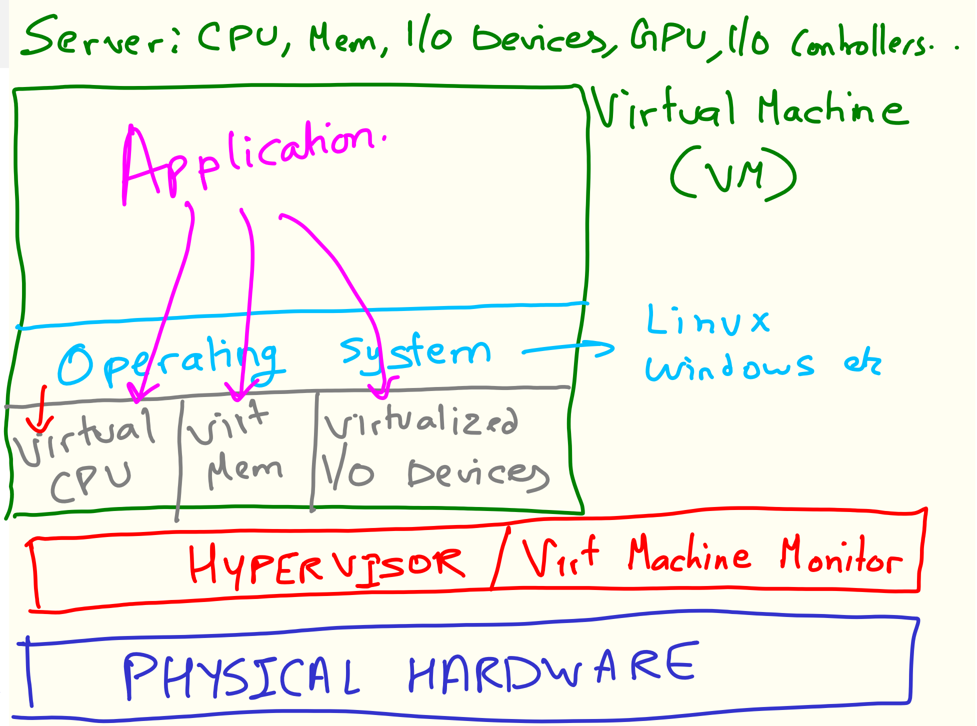
Virtualization in Operating Systems

- Virtualizing CPU enables us to run multiple concurrent processes
  - Mechanism: Time-division multiplexing and context switching
  - Provides multiplexing and isolation
- Similarly, virtualizing memory provides each process the illusion/abstraction of a large, contiguous, and isolated "virtual" memory
- Virtualizing a resource enables safe multiplexing



Virtual Machines: Virtualizing the hardware

- Software abstraction
  - Behaves like hardware
  - Encapsulates all OS and application state
- Virtualization layer (aka Hypervisor)
  - Extra level of indirection
  - Decouples hardware and the OS
  - Enforces isolation
  - Multiplexes physical hardware across VMs



Hardware Virtualization History

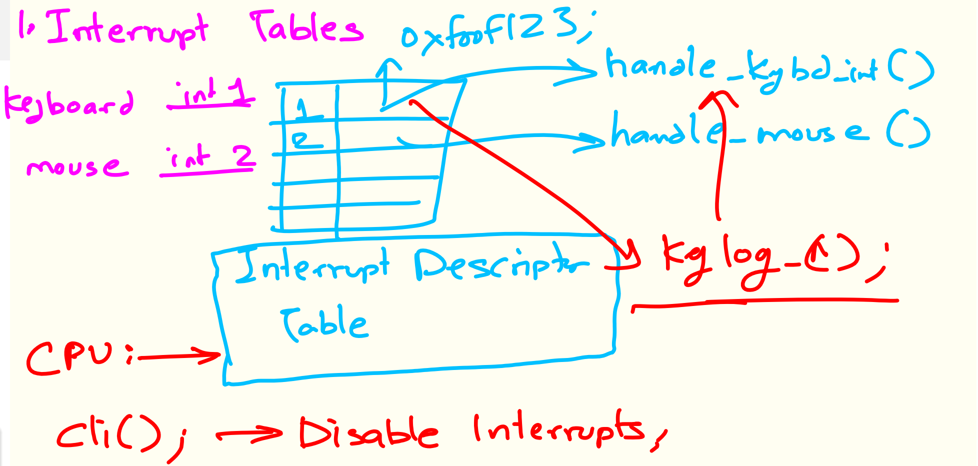
- 1967: IBM System 360/ VM/370 fully virtualizable
- 1980s–1990s: “Forgotten”. x86 had no support
- 1999: VMWare. First x86 virtualization. — Dynamic Binary Translation
- 2003: Xen. Paravirtualization for Linux. Used by Amazon EC2
- 2006: Intel and AMD develop CPU extensions
- 2007: Linux Kernel Virtual Machines (KVM). Used by Google Cloud (and others).

Guest Operating Systems

- VMs run their own operating system (called “guest OS”)
- **Full Virtualization:** run *unmodified* guest OS.
- But, operating systems assume they have full control of actual hardware.
- With virtualization, they only have control over “virtual” hardware.
- **Para Virtualization:** Run virtualization-aware guest OS that participates and helps in the virtualization.

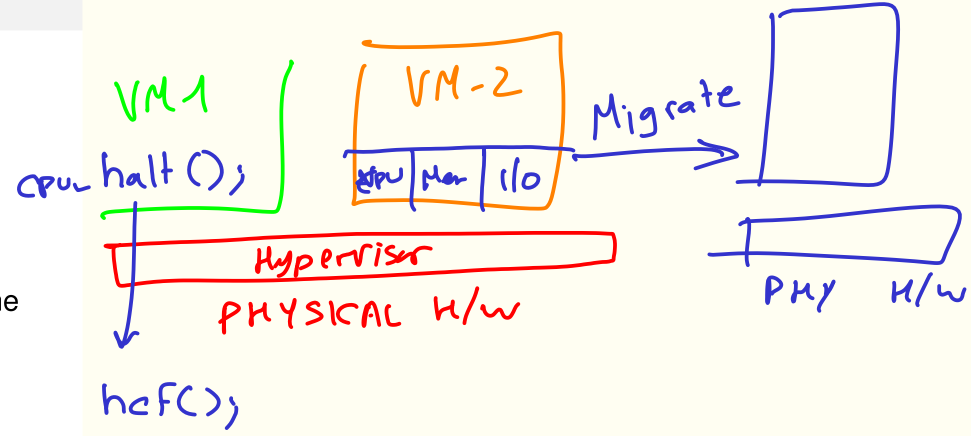
Full machine hardware virtualization is challenging

- What happens when an instruction is executed?
- Memory accesses?
- Control I/O devices?
- Handle interrupts?
- File read/write?



Full Virtualization Requirements

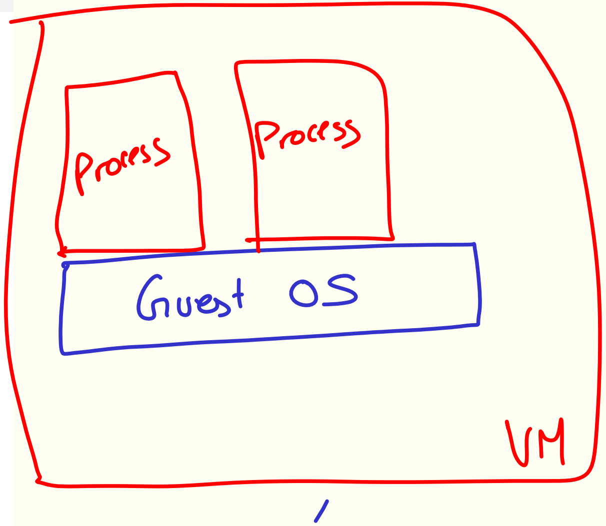
- Isolation. A VM should not interfere with its neighbours.
- Encapsulation. All VM state should be encapsulated by the hypervisor. This can be used to “move” a VM to another machine
- Performance. Applications should not face a high performance when running in a VM. Performance should be “similar” to a bare-metal OS setup.



Virtualization Goals

Popek and Goldberg set out formal requirements in 1974:

- Equivalence. VM should be indistinguishable from underlying hardware
- Resource control. VM (guest OS) should be in control of its own virtualized resources.
- Efficiency. As far as possible, VM instructions should be executed *directly* on the hardware CPU without going through the hypervisor.



Naive Approach: Emulation

- Emulation: reproduce the behavior of hardware in software
- CPU emulaiton: Interpret and translate each CPU instruction
- Device emulation: Interpret and translate device commands
- $10 - 1000\times$ performance penalty
- But, enables cross-platform execution
- x86 Linux emulated using javascript. https://bellard.org/jslinux
- However, emulation breaks the Efficiency requirement—the virutalization software should “get out of the way” as much as possible, instead of emulating every instruction.

CPU:

1. Fetch next instruction (Program Counter)
2. Decode 0101011101011101110

   Opcode → add, rax, rbx
3. Execute

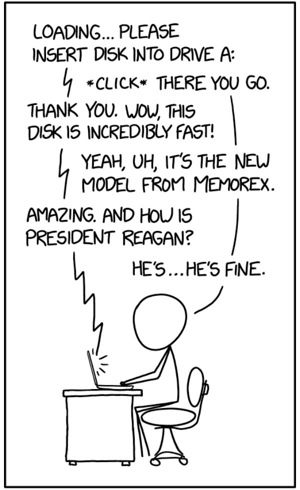
"Software CPU": Registers → Variables

: Memory → Array

: Decoding → Look up table

: Operations add/mul/mov

Emulation is still Useful!



I FEEL WEIRD USING OLD SOFTWARE THAT DOESN'T KNOW IT'S BEING EMULATED.

Floppy Drive Emulation: → 1.4 MB File on my FS

1. Initializ...
2. Read sector/block (100bytes) ⟷ read file offset (100bytes)
3. Write sector
4. Control

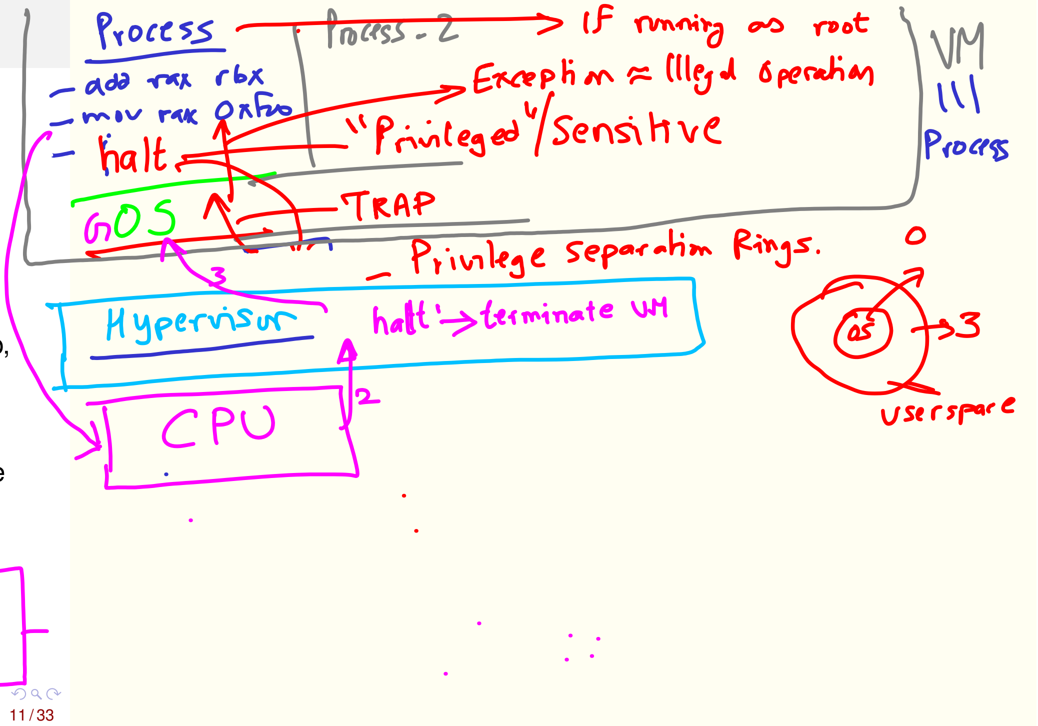
⋮

~100s

Direct execution Challenges

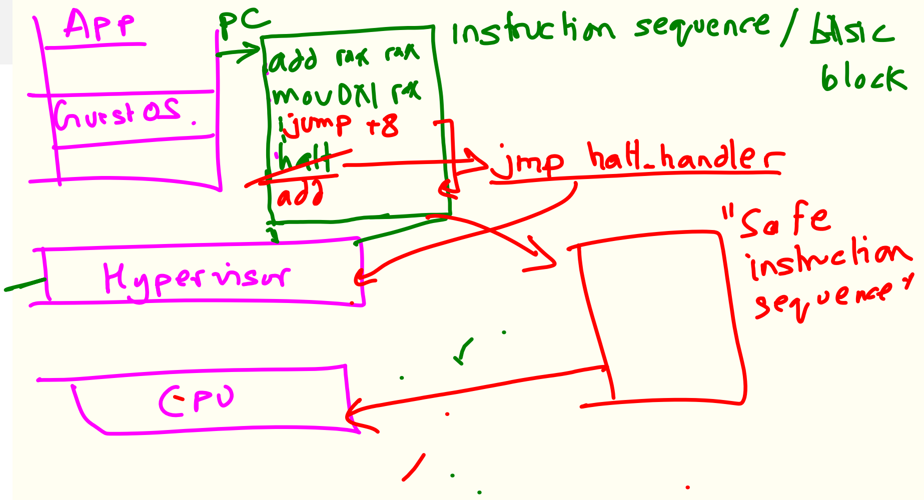
Why not just run the VM as another user-space process?

- Guest OS will want to run in a "privileged CPU mode"
- If VM runs as a userspace process, these instructions will not be allowed
- Ideal case (and Popek-Goldberg requirement): every privileged instruction should result in a *trap*
  - Control then transfers to the hypervisor, which can handle the trap, just like conventional OS.
  - Hypervisor can *emulate* these privileged instructions
  - **Trap-and-emulate** approach.
  - Example: guest OS calls `halt`. Hypervisor traps and emulates the guest OS intent, and turns off the Virtual Machine by killing the userspace process that the VM was running as.
- x86 : Nah.
  - Some instructions behave differently when executed with different privilege levels.
  - Traps are not always generated. Instructions thus fail silently, and guest OS crashes.



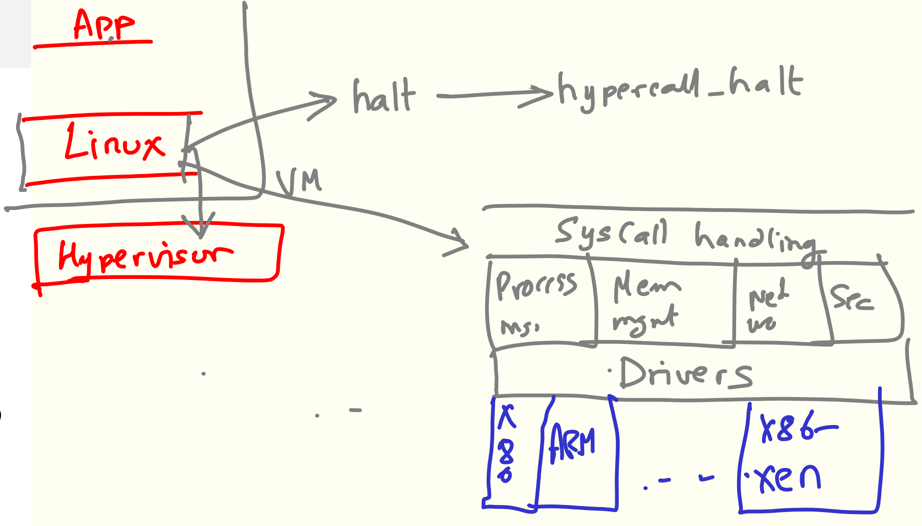
Dynamic Binary Translation

- Application code inside VM is generally “safe” and can be directly executed (there are no sensitive instructions)
- Guest OS issues sensitive instructions.
- **Key idea: Rewrite the instructions that are executed by the guest OS**
- Also refered to as “Just in time” translation
- Before some VM (guest OS) code is executed, hypervisor “scans” it, and rewrites the sensitive instructions to emulate them.
- Typically done at basic-block level.
- Approach pioneered by VMware to make x86 virtualizable
- Performance overhead can be reduced with engineering optimizations:
  - Keep a cache of translated blocks
  - Offset memory accesses and jumps become tricky when mixing translated and vanilla basic blocks.



Paravirtualization

- Pioneered by Xen in 2003. (Research project from Cambridge University)
- First open-source x86 virtualization
- **Key-idea: Modify the guest OS to never issue sensitive instructions directly.**
- Instead, guest makes "hypercalls" to the hypervisor when it wants to do something privileged.
- Surprisingly, the amount of modifications required are small, and relatively easy to make.



Hardware assisted Virtualization

- In 2006, Intel and AMD, *finally* fixed x86
- New privileged ring level added : -1
- *Hardware-assisted trap and emulate*
- All sensitive instructions now trap. Yay!
- When guest OS executes these instructions, they cause a `VM-exit`
- Hypervisor handles the VM-exit, and resumes the VM through the `VM-enter` instruction.
- Hardware assigns each VM a VMCB/VMCS (VM control block/structure) which maintains trap information.
- Used by all hypervisors today.
- First used by KVM (Linux's kernel virtual machine module)

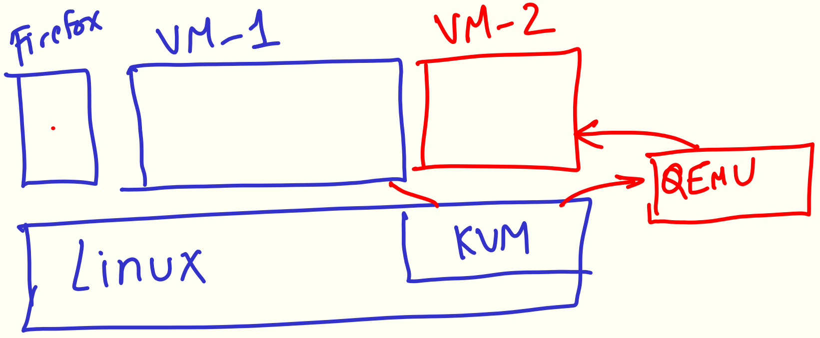
Trap & Emulate

KVM

- Key idea: VMs are just Linux processes!
- Hardware extensions make hypervisors easy to write
- A lot of what the hypervisor does (resource management and control) is done by the OS (Linux) anyway.
- Why write a new OS, just use Linux as the hypervisor!

QEMU

- Quick Emulator
- Emulates all kinds of devices (bios, cdrom, network cards,...)
- KVM uses QEMU for device emulation and handling all userspace VM management operations
- QEMU handles launching and stopping VMs, monitoring, debugging, etc.

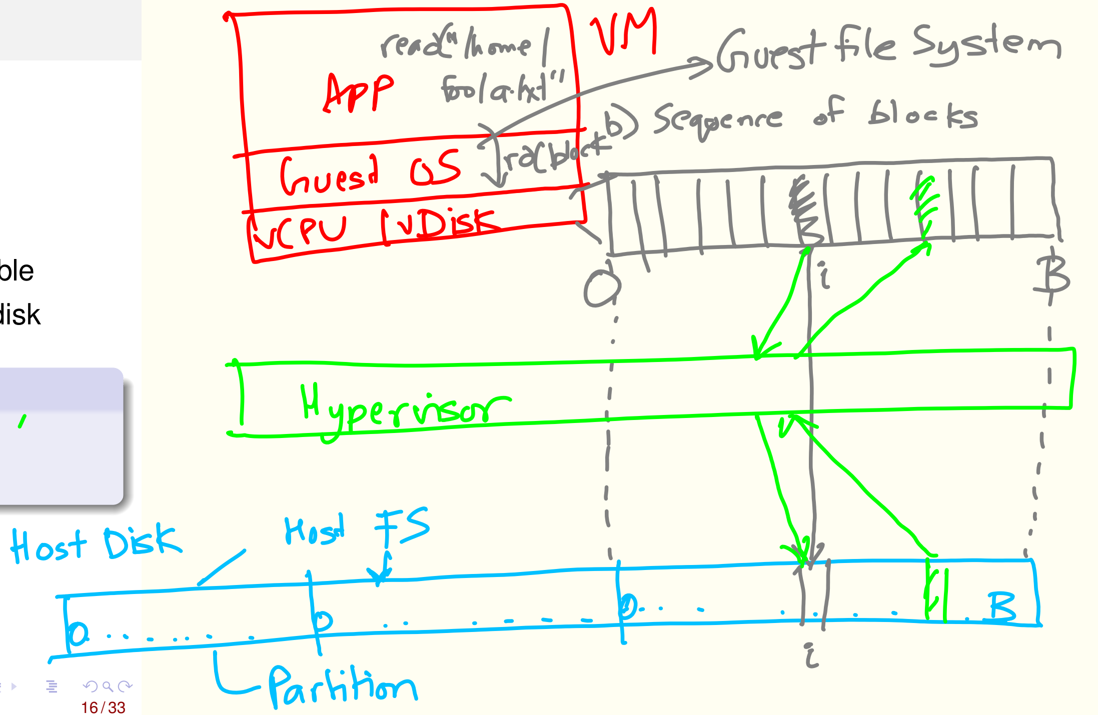


Storage Virtualization

- Bare-metal OS: Filesystem writes to physical disk partition
- Guest OS: Allowing direct writes to disk is not usually suitable
- **Note:** Guest OS runs its own file system on its virtualized disk

Solution 1: Assign a physical disk partition to VM

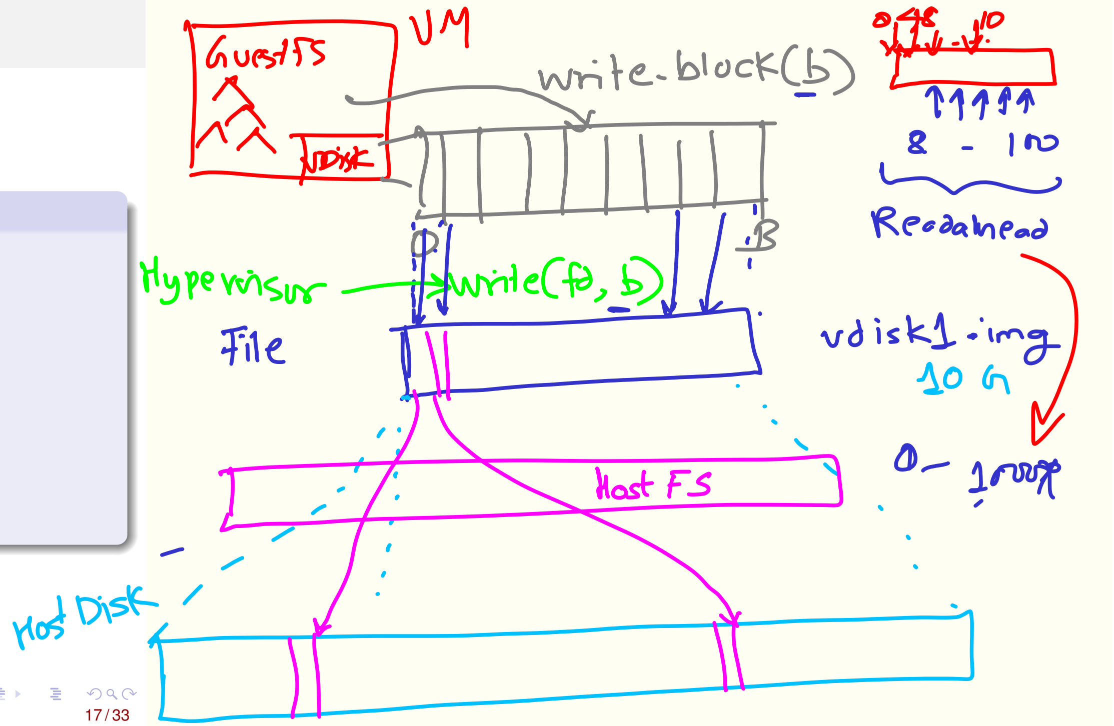
- Physical disk formats limit the number of partitions
- VMs get partitions of pre-allocated sizes



Virtual disks

Solution 2: Virtual disks

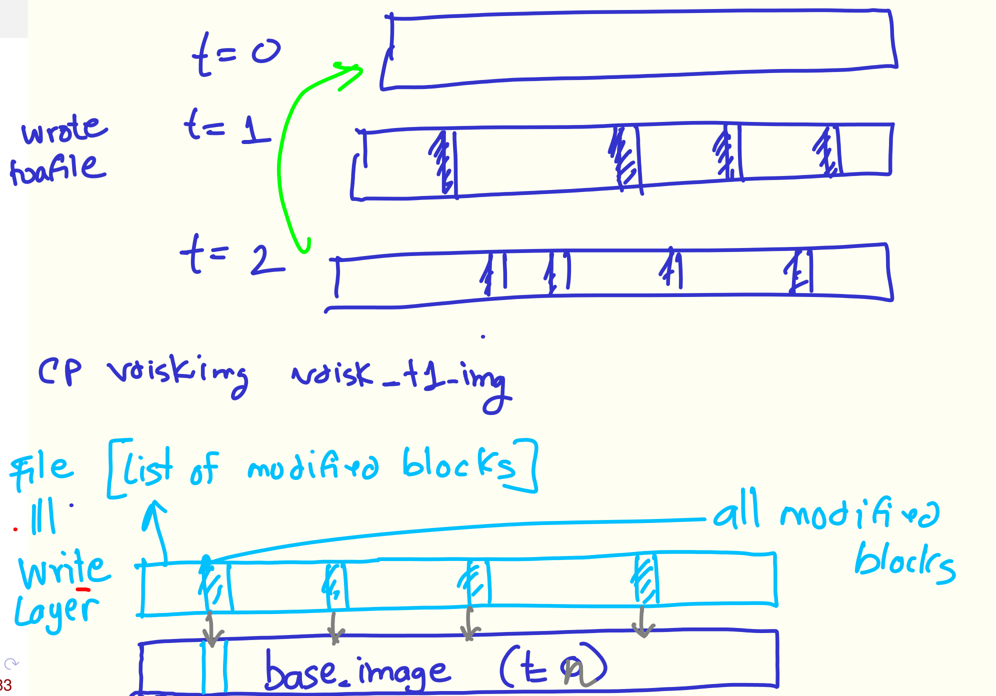
- Hypervisor intercepts and translates disk writes
- Guest OS writes to `guest-block-i`
- Hypervisor maintains a `guest-block` to `host-block` table
- Usually, a virtual disk is a **file** on the host file system
- Example, VM may be assigned `/home/VMs/vdisk1.img`
- `guest-block-i` is simply `offset-i` in the `vdisk1.img` file
- Two filesystems in play: Guest FS and Host FS



More Virtual disks

- Virtual disks make full-disk snapshots easy
- Hypervisor can record all blocks written by the VM
- Common technique: **copy-on-write**
- Enabled by more complex disk formats (qcow2, etc)
- Guest writes to **guest-block-i**
- Original mapping is **virtual-disk-block-i**
- Hypervisor *copies* the original vdisk-block-i to vdisk-block-j.
- Write operation is applied to vdisk-block-j.
- Hypervisor updates the mapping : guest-block-i is now vdisk-block-j
- Old block (vdisk-block-i) remains unmodified.

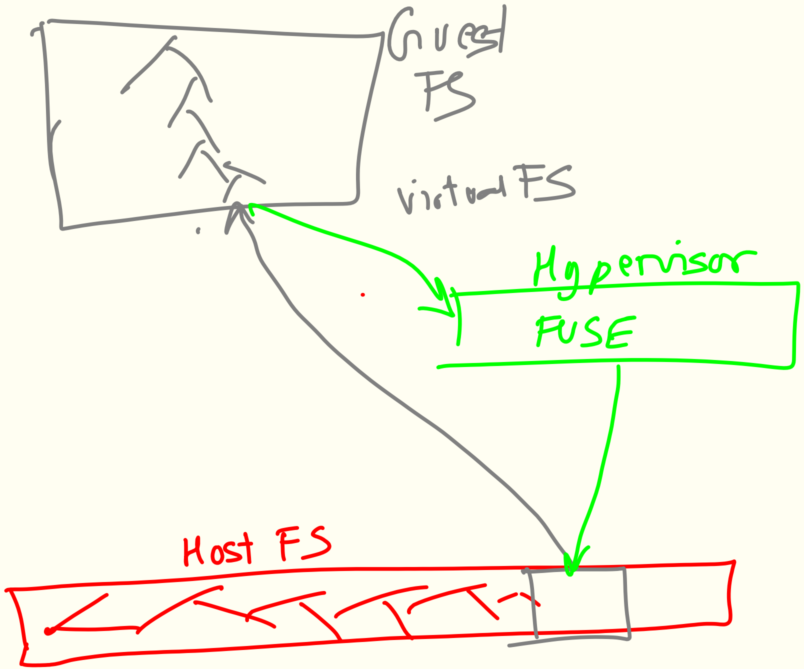
- Copy on write allows disk snapshots : Copy all modified blocks.
- Notion of layered storage: Snapshot contains only modified blocks, and uses the original VM disk for unmodified blocks.



Remote Virtual Disks

Remote Disks

- In many cases, the virtual disk can also be *remote*
- Simple approach: Virtual disk is on an NFS file system
- Or use vdisks on Storage Area Networks (SANs)



Using KVM

- Launch VM: `sudo qemu-system-x86_64 -enable-kvm vdisk1.img`
- Install OS: `qemu -cdrom ubuntu.iso -boot d vdisk1.img`
- Create raw/plain vdisk: `qemu-img create vdisk1.img 10g`
- Copy-on-write: `qemu-img create vdisk2.qcow2 10g`
- Create snapshot `qemu-img create snap1.img -b vdisk2.qcow2`
- VM memory: `-m 4g`
- Number of vCPUs, SMP and NUMA configuration, ...
- Networking options : bridge (tun/tap interface), userspace (NAT), ...

All VMs are Processes

Memory Virtualization

Conventional bare-metal OS

- Process Virtual Address $\rightarrow$ Physical Address
- OS sets up **page-table** for this translation
- Higher-order bits of addresses used to determine *page-number*
- Address = Page-number + Offset within page
- Virtual to physical translation done by CPU's MMU (memory management unit) for *every* read/write access
- CPU maintains a cache of translations: Translation Lookaside Buffer

With Hardware Virtualization

- Guest OS maintains Guest Virtual Address $\rightarrow$ Guest Physical Address page tables
- Another layer of indirection is needed:
- Hypervisor does the Guest Physical Address $\rightarrow$ Machine Physical Address mapping

Approaches to Memory Virtualization

Demand-filled Software MMU

- Hypervisors can maintain guest-physical to machine-physical mappings
- On-demand translation: For every guest-physical page access, hypervisor looks up the machine-physical page number and inserts that into the page-table that the CPU MMU "walks".
- This is effectively a "software managed TLB"
- Hypervisor marks all page table entries as "invalid" initially, and fills them on page-faults
- Essentially trap-and-emulate (more like trap and translate)

Hardware assisted paging

- Virtualization-enabled CPUs support *multi-dimensional* paging
- CPU MMU can walk Guest and Host page tables

Shadow Paging

- Xen introduced shadow paging, a different approach to memory virtualization
- Key idea: When the guest OS wants modifies its own page tables, it makes a hypercall instead
- Hypervisor then modifies the page table on the Guest OS's behalf
- CPU points to the modified page-table created by the Hypervisor
- Thus, hypervisor provides a "Shadow page table" for every guest page table
- Advantage: Minimal address translation overheads, since there is no extra translation required during regular execution.
- Even with hardware assisted double-paging, CPU has to access multiple pages for translation for every VM memory access.

I/O Devices

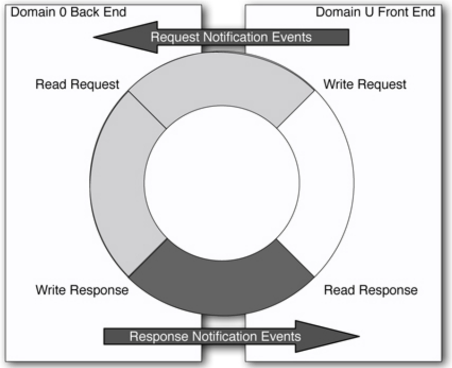
- QEMU emulates I/O devices such as keyboard, mouse, disk controllers, network interface cards (NICs), . . .
- Guest OS sees generic virtual hardware devices
- Guest OS device drivers interact with virtual devices
- QEMU faithfully implements and emulates hardware functionality
- Example: Guest sends packet through vNIC $\rightarrow$ QEMU will send the packet through real NIC

Hardware assisted I/O Virtualization

- Emulating hardware devices allows flexibility and resource management
- But biggest drawback: performance, especially for latency-sensitive devices and operations (fast NICs > 10 Gbps)
- Some I/O devices have virtualization capability
- I/O device exposes multiple “virtual” devices
- Each virtual device can be assigned directly to the VM
- Often accomplished with SR-IOV (Single Root I/O Virtualization).
- Guest OS interacts with hardware directly, instead of going through an emulated device.
- Especially popular in network cards

Xen Paravirtualized I/O

- Expose a simplified device to the guest
- No need to emulate different types of the same I/O device
- Set of ring-buffers for reading and writing from/to device
- Guest OS must have drivers for paravirtualized devices
- Hypervisor does actual device transfers



Virtual Machine Snapshots

VM State is comprised of:

- vCPU state (registers etc.)
- I/O device state (registers, other device state)
- Virtual disk state. “Easy” with copy on write virtual disks.
- All **Memory** state. (The most interesting)

- Offline snapshot: Stop or pause VM and take full snapshot
- Online/Live snapshot: Take snapshot of running VM.

Snapshot uses:

- Full-system backups
- Debugging
- **Migrating VMs between physical servers.**

VM Live migration

- Move entire VM from one server to another
- *Without affecting application*
- **Live** $\rightarrow$ VM cannot be stopped during the migration

Usecases:

- If physical server needs to be shut-down for maintenance
- Server is overloaded with VMs, so move some to other servers
- Move VM closer to the end-user if user moves (“follow the user”)

- Typically, virtual disk is remote (mounted over the network)
- The challenge is therefore how to move all memory state without application downtime

Migration Flow

Live Migration flow:

- VM is running on `source` till time t
- VM runs on `destination` from tim $t + \delta$.
- δ is the *downtime* when the VM is not running
- Goal: make downtime as small as possible
- For *offline* migration, downtime is the time it takes to copy all VM state over the network (can be ~minutes).

Live Memory Snapshot

Central problem:

- As a VM executes, it writes to its memory pages
- Saving a memory snapshot entails copying memory pages and storing them (typically on disk).
  - Thus, memory snapshots take non-zero time
- How to save something that is constantly changing?

Live Memory Migration

1. Copy entire memory to remote server
   - Takes time $t_0 = t(M)$, where M is the memory size
   - During this time, some pages have changed
2. Copy **only the pages that have changed** to remote server
   - Hypervisor write-protects all pages. If page is written to by the VM, CPU marks it as “dirty”.
   - Hypervisor copies all dirty pages into a buffer, and sends them over the network.
   - Because of locality of reference, number of pages dirtied is small, and is $D_{t(M)}$
   - Time required to send these pages is $t_i = t(D_{t_{i-1}}) \leq t_{i-1}$.
3. Repeat step 2 until dirty pages D_{t_i} is small enough.
   - Can be determined based on acceptable downtime
   - Transferring ~10 megabytes of pages will result in downtime of only few milliseconds!
4. If dirty page threshold is reached, **stop** the VM, and copy the remaining dirty pages and vCPU and I/O state.
5. Resume VM on destination server

More VM Live migration

- Iteratively copying smaller and smaller number of dirty pages
- Large VMs (several GB of memory) can still be migrated with very small downtimes
- Usually applications are not affected, if downtime is smaller than the network timeouts their clients set (which is usually 10s of seconds).

Post-copy VM migration

- So far, have seen a “pre-copy” approach
- Copy all VM memory state to destination before running on destination

Alternative approach: Post-copy

- Move vCPU state to destination first
- VM execution will cause page-faults
- Copy pages from source upon first access
- Advantage: VM can start running on destination immediately
- Downside: Residual state (pages) can exist on source server for a long time