

Virtualization

Virtualization



Traditional Architecture



Virtual Architecture

What is Virtualization?

While we've briefly mentioned that virtualization technology exists and that operating systems run on top of it, we haven't explained why its important or how it works.

Traditionally, when you are talking about Physical Machines you are referring to:

- Real hardware resources
- That a single operating system has exclusive access over
- Through hardware interfaces (I/O, PCI Lanes, Memory Busses, etc)

What is Virtualization?

When we are talking about Virtual Machines we are referring to operating systems that are running inside or “on top of” another operating system layer known as a Virtual Machine Monitor or ‘Hypervisor’

- An operating system running on top of a VMM/Hypervisor is known as a Guest OS

Virtual Machines differ from Physical Machines in a few important ways

- They do not provide the guest OS with exclusive access to the underlying physical machine.
- As such, they do not provide the guest OS with fully-privileged access to the physical machine.

What is Virtualization?

Therefore we can define the Virtual Machine Monitor (VMM) as

- A piece of software running on an operating system (the host OS)
- That can allow another operating system (the guest OS) to be run as an application
- Alongside other applications (which can be other Guest OSs)

This means that for all intents and purposes, we can abstract a Guest OS as just another application running on-top of 'bare metal' or within an OS of its own

So why virtualize?

A few of these reasons we covered during Capacity Planning

- Resource utilization
 - Modern servers are very powerful, want to multiplex their hardware to avoid wasting resources that could be put to use elsewhere
 - The more you can run on one system the less space, power, etc underlying infrastructure you require overall
- Software development and deployment
 - One machines can now run multiple OSs simultaneously and have different workloads associated with each depending on the application requirements

So why virtualize?

- Fault Tolerance

- VMs can be run in a 'High Availability' state where the VMM creates a lockstep copy of a virtual machine on a different host - if the original host suffers a failure, the virtual machine's connections get shifted to the copy without interrupting users

- Strong Isolation between VMs

- Prevents a fault of an application in one VM impacting other VMs running on the same host to protect from bugs or malicious attacks
- Operating systems "leak" a lot of information between processes through the file system and other channels
- Multiple applications may require specific (and conflicting) software packages to run or have very specific operating systems configuration and tuning requirements

So why virtualize?

- Ease of management/deployment
 - Snapshots of VMs can be taken for backup/DR
 - Pre-configured Virtual 'Appliances' can be easily deployed with all the software and configuration ready to go from templates
 - Built-in monitoring and alerting regarding resource usage, unexpected reboots, failures, etc
- Flexible VM relocation
 - Virtual Machines can be 'live migrated' from one physical host to another for maintenance, load balancing, or troubleshooting

(Demo)

So why virtualize?

This technology isn't just limited to Servers and the 'Cloud'

- Virtual Desktop Infrastructure (VDI) is only getting more popular and allows for small cheap terminals to connect users to 'Desktops' running on a server in a datacenter
- Development VMs can be run on laptops to provide pre-configured installations or test out how an application behaves on different operating systems without requiring multiple physical devices
- Mobile Device Virtualization

VMM Requirements

Popek and Goldberg provide three essential requirements in 1974 for a piece of software to be considered a virtual machine monitor (VMM), or to implement a virtual machine

- Fidelity: software on the VMM executes identically to how it would on real hardware
- Performance: to achieve good performance most instructions executed by the guest OS should be run directly on the underlying physical hardware instead of emulated in software
- Safety: the VMM should manage all hardware resources.

The three approaches to Virtualization

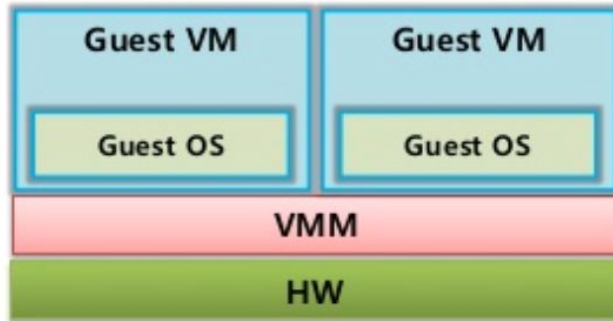
- Full virtualization - Should be able to run guest OS without any necessary modifications.
 - i.e. VMWare, VirtualBox
- Paravirtualization - Includes small changes to the guest operating system to improve interaction with the virtual machine monitor.
 - i.e. VMWare, Xen, Amazon EC2.
- Container virtualization - Namespace and other isolation techniques performed by the operating system to isolate sets of applications from each other. We will be covering this as a separate topic
 - i.e. Docker, CoreOS, Containerd

Types of VMMs

When it comes to VMMs there are two approaches

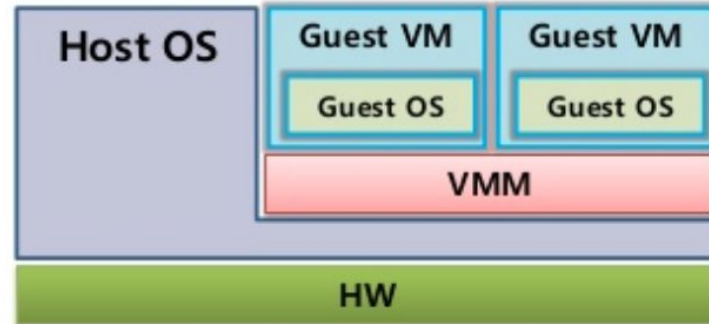
- Type 1 - the VMM runs directly on top of the hardware (Hypervisor)
- Type 2 - the VMM run on top of an existing operating system (Hosted)

Type-1: VMM on HW



- Xen, VMware ESX server, Hyper-V
- Mostly for server, but not limited
- VMM by default
- OS-independent VMM

Type-2: Host OS on HW



- KVM, VMware Workstation, VirtualBox
- Mostly for client devices, but not limited
- VMM on demand
- OS-dependent VMM

So how does Virtualization actually work?

First and foremost, this is not a class Operating Systems (which most of you are probably thankful for)

As such we won't be diving super deep into how hypervisors handle the specifics of syscalls, virtual memory allocation, etc. That said we will take a high level look at how virtualization of CPU, Memory, and I/O devices works

Virtualizing the CPU

The way CPUs are virtualized is inherently similar to applications are allocated CPU resources within an OS (as with VMs, they are treated as applications)

So how does this actually work?

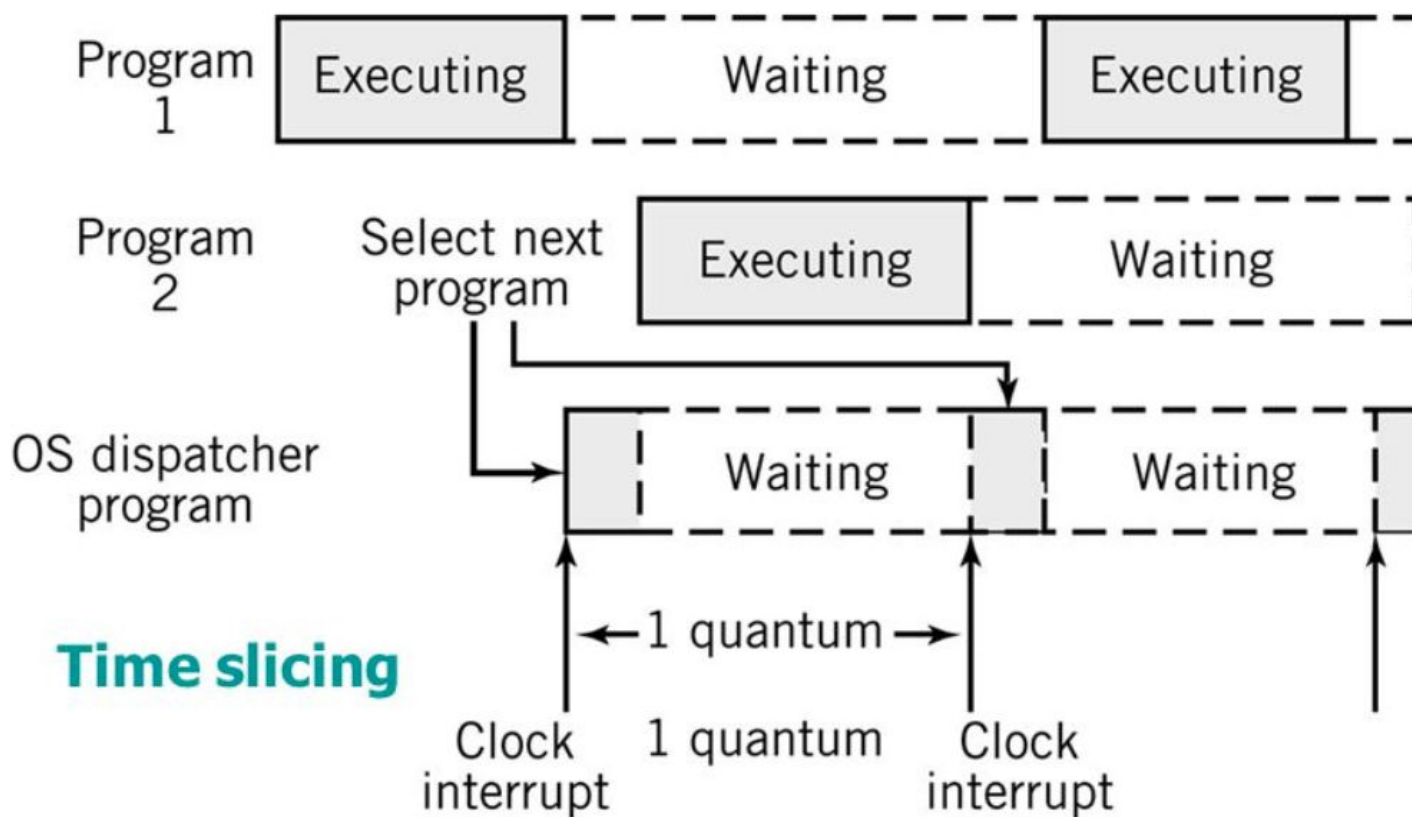
- We could assign CPUs in a 1-1 mapping of physical to virtual
 - Are systems usually running at 100% load all the time?
 - What about all of that wasted 'idle' CPU time?
 - Is this the best we can do to improve resource utilization?
- Does your Operating System need one CPU for each application you're running?

Virtualizing the CPU

Of course not - we run 100s of applications on just a few CPU Cores

- The way they handle this is by sharing a single CPU between multiple applications (or in the case, Virtual Machines)
- That said, no two VMs can be using the same CPU at the same time (even if it isn't being used to 100% efficiency) as a CPU core can only work on one workload at a time
- Therefore, sharing needs to be done by time slicing
 - Each VM gets a CPU resource for a certain amount of time (based upon a variety of factors) and switches back and forth with other VMs and then each VM will then timeslice its OS/Applications within its allocated quantum.

Virtualizing the CPU



Virtualizing the CPU

So by sharing CPU Cores with multiple VMs, does this mean we can allocate more vCPUs than we have pCPUs?

- The short answer is yes, absolutely - this is known as Overcommitment
- The longer answer here is a bit more nuanced as the exact amount of CPU overcommitment a VMware host can accommodate will depend on the VMs and the applications they are running
- A general guide for performance of {allocated vCPUs}:{total pCPU} from the Best Practices recommendations is:
 - 1:1 to 3:1 is no problem
 - 3:1 to 5:1 may begin to cause performance degradation
 - 6:1 or greater is often going to cause a problem

Virtualizing the CPU

The caveat here is do not allocate more vCPUs than needed to a VM as this can unnecessarily limit resource availability for other VMs and increase CPU Ready wait time

- When allocating resources for a new VM, start with one vCPU per VM (or whatever the minimum requirements for the workload is and increase as needed
- This is due to the fact that when scheduling CPU time for a VM, the hypervisor is required to schedule all of the vCPUs for a VM to pCPUs at the same time

Virtualizing the CPU

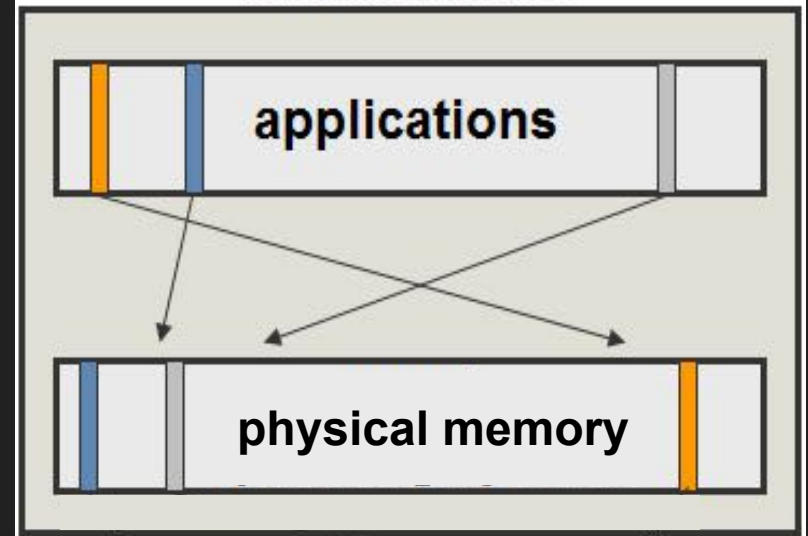
This means if we have

- A physical host with 8 pCPU available (after overhead)
- We want to run two VMs on it
- If we allocate 6 vCPUs to each of them then only one of the VMs can be running at any given time
- Whereas if we can cut them back to 4 vCPU each, they can both run simultaneously

This also means you cannot allocate more vCPUs to a single VM than you have pCPUs on a physical host

Virtualizing Memory

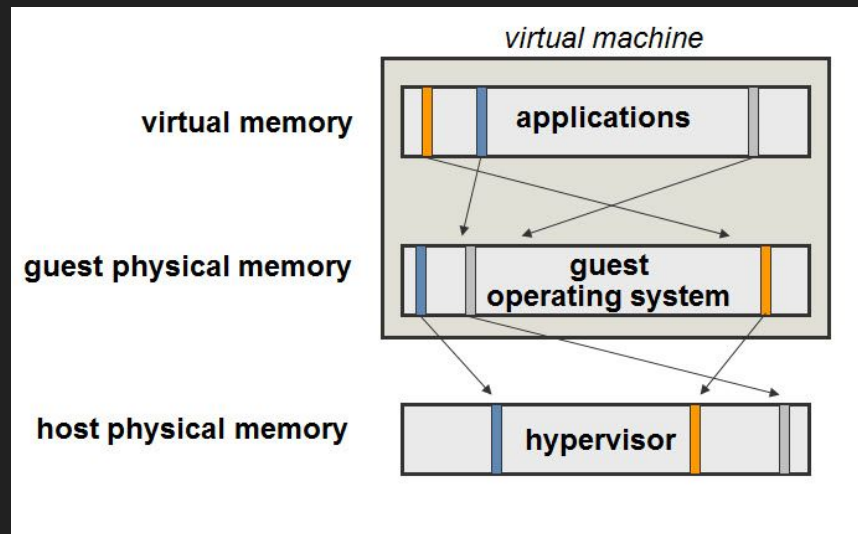
Memory Management is a large and complex topic of Operating System design - for this class we are going to focus on some high level techniques regarding special features when it comes to Virtual Machine RAM and not focus too much on the underlying implementation.



Virtualizing Memory

- Let's start with basic RAM allocation - when a VM uses RAM, its OS (the hypervisor) handles allocating a place on the physical RAM for it to be stored
- When you create a machine with 2GB of RAM, the hypervisor does not need to allocate that much physical memory all at once (and often doesn't)
- Why might this be the case?

Because machines that are allocated X amount of RAM often do not use it all immediately, and may never use all of it



Virtualizing Memory

There are a several metrics when it comes to VM Memory - we are going to cover 3 for now



- Provisioned Memory - The amount of memory allocated to a VM plus the overhead needed to manage the VM (Static)
- Consumed Memory - Current level of host memory consumed for a specified VM
- Active Guest Memory - Hypervisor estimate of memory actively being used in the VM's guest OS. The hypervisor does not communicate with the guest OS, so it does not know if any memory allocated to the VM is no longer needed



Resources

Consumed Host CPU:	406 MHz
Consumed Host Memory:	8261.00 MB
Active Guest Memory:	1474.00 MB

[Refresh Storage Usage](#)

Provisioned Storage:	208.05 GB
Not-shared Storage:	208.05 GB
Used Storage:	208.05 GB

Storage	Status	Drive Type
 SANArray2	 Normal	Non-SSD

Network	Type	Sta
 VM Network	Standard port group	

Virtualizing Memory - Overprovisioning

If the hypervisor doesn't allocate physical memory until the VM needs it, does this mean we can overcommit RAM similarly to how we overcommit vCPU?

- Well, yes and no - overcommitting RAM is a bit more nuanced than allocating vCPU time
 - Why do you think this is the case?
- Where a VM can 'share' a CPU with another VM via time slicing, RAM doesn't quite work the same way as it is storing data
 - For a VM to 'share' RAM would require putting the existing memory somewhere else
 - Standard operating systems have a concept for this called Swapping where it writes the contents of RAM to disk to make more room in memory
 - Hypervisors are also capable of swapping, but like most OSs they use it as a last resort due to the huge performance losses

Virtualizing Memory - Overprovisioning

So if we can't time slice to overcommit memory how is it possible?

- As we mentioned earlier, when a VM is configured with 2GB of RAM for example, the hypervisor does not allocate it all at once for physical RAM - it waits until the VM asks for it and then allocates it.
- This means that if we have a physical host with 10GB of ram, we could support 5 VMs (assuming they used all of their 2GB configuration), that said - it also means we could support 40 VMs with the same configuration, assuming they never actually use more than 512M of their configured memory.

So does this mean we should overprovision all of our VMs with extra RAM?

Virtualizing Memory - Overprovisioning

Absolutely not - because the trouble is when a VM is allocated physical memory by the hypervisor it can not be easily reclaimed

- So what happens if we overcommit memory and the VMs actually try to use it?
- Well, the hypervisor would start to undergo memory pressure as the remaining physical memory is used up and as a last resort it will start swapping memory to disk

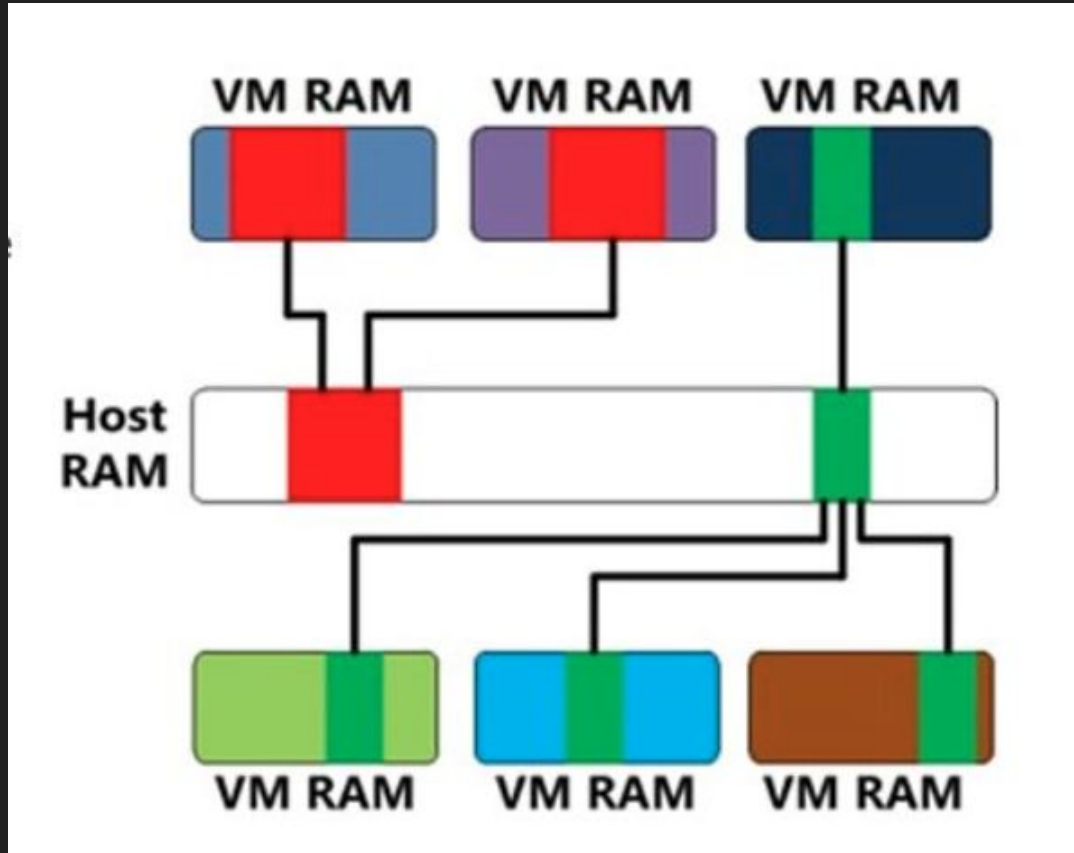
However, before it starts swapping, the hypervisor may have a few tricks up its sleeve to reclaim memory

Transparent Page Sharing (TPS)

- The VMware hypervisor is capable of tracking identical memory pages within VMs with the same OS
- So when the hypervisor finds identical pages on multiple VMs on a host, it can share them among VMs via pointers to the physical memory page
- Let's say, there are 30 copies of the same memory segment among different VMs with the same OS. ESXi keeps just one and the rest are just pointers

As a result, the total virtual machine host memory consumption is lowered and we can allocate more memory to our VMs (to overcommit even more). TPS works in the background, from time to time as memory pressure builds.

Transparent Page Sharing (TPS)



Ballooning

Ballooning is a process where the hypervisor reclaims memory back from the virtual machine by causing a process within the Guest OS to intentionally eat up memory.

(I know - it seems counter intuitive)

- Let's take a high level example:
 - Inside a virtual machine you start an application like Google Chrome
 - Chrome as an application will ask the Guest OS for memory. The Guest OS will give it memory and map it from the virtual memory -> guest physical memory
 - The hypervisor then sees the request for memory and the hypervisor maps guest physical memory -> host physical memory
 - Now everything is perfect. You surf the web for a few hours. And then you close it down.
 - When you close Chrome, the guest operating system will mark the memory as "free" and make it available for other applications
 - BUT since the hypervisor does not have access to see the Guest OS's "free memory" list, the memory will still be mapped in "host physical memory" and still puts memory load on the ESXi host

Ballooning

This is where ballooning comes into play.

In case of an ESXi host running low on memory the hypervisor will ask the “balloon” driver installed inside the virtual machine to “inflate”

- The balloon driver will inflate and because it is “inside” the operating system it will start by getting memory from the “free list”. The hypervisor will detect what memory the balloon driver has reclaimed and will free it up on the “host physical memory” layer
- The balloon driver can inflate up to a maximum of 65%. For instance a VM with 1000MB memory the balloon can inflate to 650MB.

If a host is under memory pressure, the hypervisor will try to use TPS and Ballooning to free up memory but failing that it'll resort to swapping to disk.

Memory Compression & Swap

If TPS and Ballooning are unable to reduce memory pressure above normal thresholds the hypervisor then resorts to Compression and Swapping

- For compression, the hypervisor looks for memory pages that it can compress and reduce by at least 50%
 - This requires scanning through memory, executing compression operations, and storing it back into memory
 - Not exactly an 'inexpensive' operation
- As a last resort, the hypervisor will implement swapping memory to disk - this can be a huge performance impact as it means the speed of RAM is now limited to the speed of the swap disk

I/O Virtualization








In addition to CPU/Memory, a number of I/O devices need to be taken into consideration for Guest OSs.

These can range from Storage and Networking to arbitrary add-in cards, PCI devices, custom hardware, and more. So how does the hypervisor take all of these devices into consideration and properly hand them off to the Guest OS?

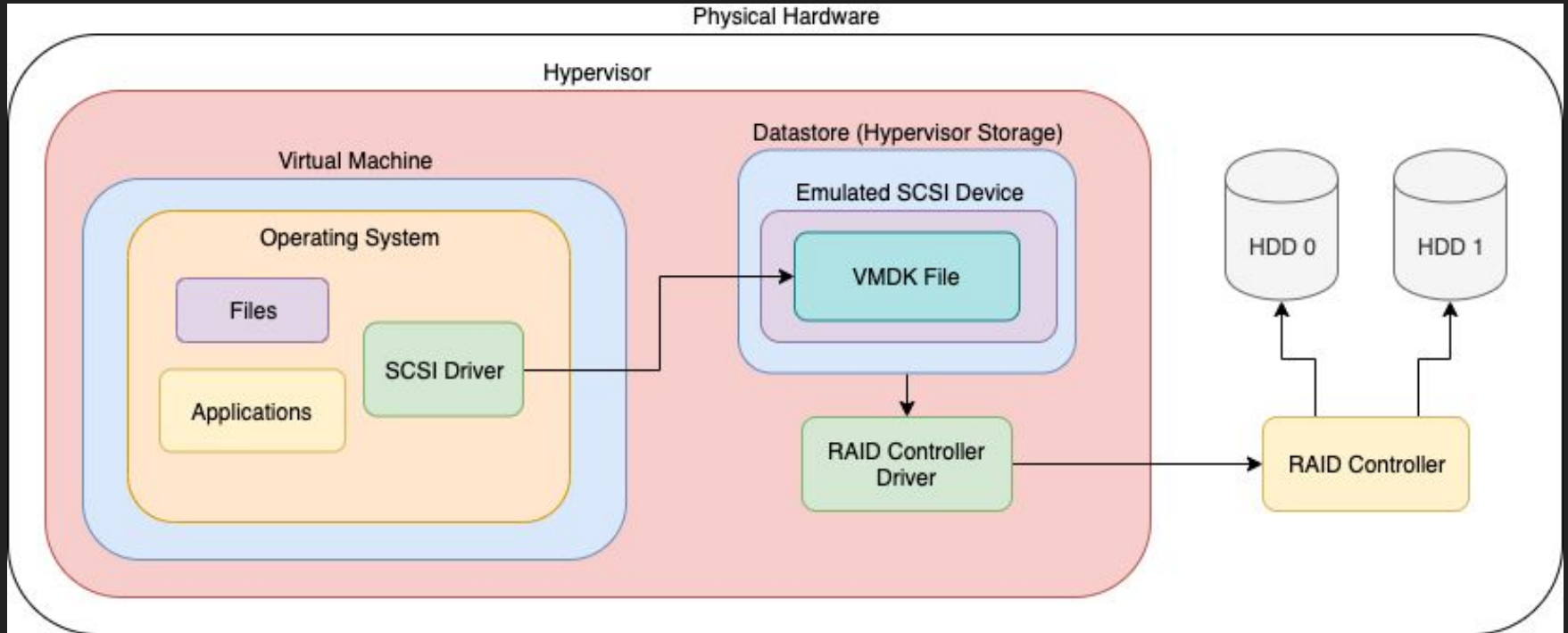
It can achieve this via 'pass-through' where the hypervisor effectively hands over the physical device directly to the Guest OS, or via an 'emulated device' where the hypervisor handles multiplexing access to the device for potentially multiple VMs

Storage Virtualization

Let's start with storage - the Guest OS expects to see a storage device and appear to have full control over it. Similar to how a RAID controller abstracts away the underlying physical devices and presents a 'Virtual Disk', the hypervisor introduces another layer of abstraction where the disk presented to the VM is actually just a file (.vmdk or Virtual Machine Disk) - however to the Guest OS it acts just like a SCSI HDD/SSD.

 env.iso	52 KB
 env.json	9.07 KB
 ephemeral_disk.vmdk	2,012,160 KB
 vm-0ce30a62-bce5-455a-85c0-1c11f93e0aaa-081bc8cc.hlog	0.47 KB
 vm-0ce30a62-bce5-455a-85c0-1c11f93e0aaa-10cea976.vswp	2,097,152 KB
 vm-0ce30a62-bce5-455a-85c0-1c11f93e0aaa.nvram	8.48 KB
 vm-0ce30a62-bce5-455a-85c0-1c11f93e0aaa.vmdk	171,008 KB

Storage Virtualization



Types of Storage Provisioning

There are usually 3 types of storage provisioning

Thick (Lazy Zeroed) - Allocates all storage up front but waits to zero out the blocks until they need to be written to

Thick (Eager Zeroed) - Allocates all storage up front and zeros out the entire disk up front as well

Thin - Does not allocate any storage up front, only provisions what the VM has used thusfar - need to be careful with overprovisioning!

▼ New Hard disk *	40	GB ▼
Maximum Size	168.79 GB	
VM storage policy	Datastore Default ▼	
Location	Store with the virtual machine ▼	
Disk Provisioning	<div>✓ Thick Provision Lazy Zeroed Thick Provision Eager Zeroed Thin Provision</div>	
Sharing		

Questions?