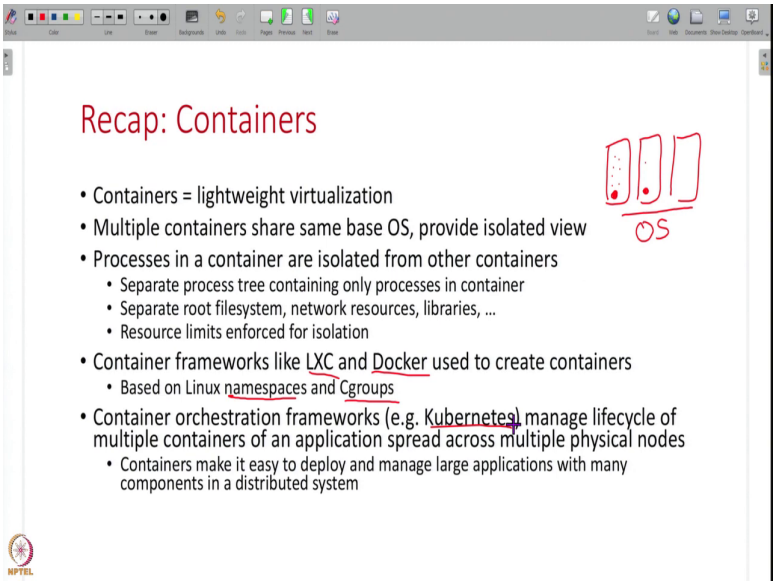
Design and Engineering of Computer System
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 28
Memory and I/O Virtualization

Hello everyone, welcome to the 20th lecture in the course Design and Engineering of Computer Systems. So, in the last couple of weeks we have seen how memory management and IO subsystem work in operating systems. So, in this lecture, I will briefly touch upon how memory and IO are managed inside virtual machines when you have virtualization, and that will wrap up our discussion of memory and IO in operating systems. So, let us get started. So, to recap, we have seen that there are two ways of doing virtualization one is containers and one is VMs.

(Refer Slide Time: 00:52)



So, what are containers? Containers are a way to do lightweight virtualization, where you have multiple containers will share the same underlying OS, but they will still provide different isolated views to processes, processes while in one container will only see the processes in that container it will not see the processes in other containers.

Similarly, you can isolate the file system, network resources. For example, if somebody opens a socket, in this container at a certain port number, another container also can have the same socket at the same port number because these two are isolated from each other. So, all

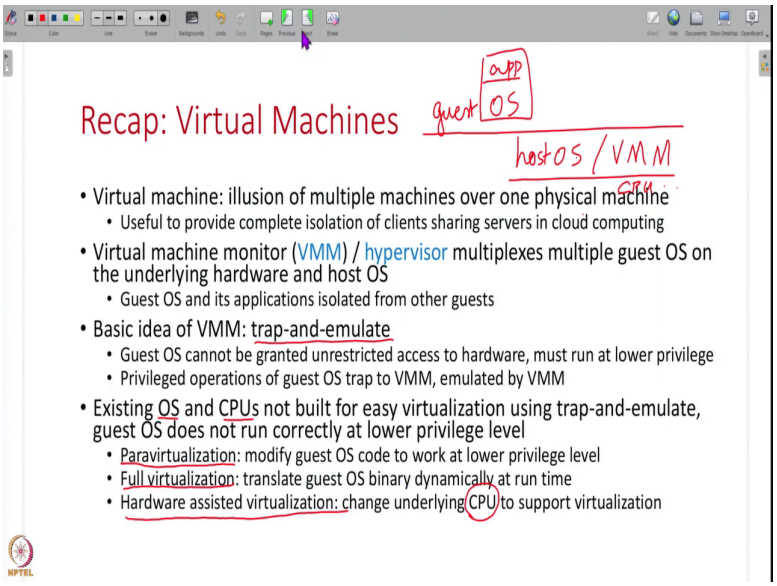
such mechanisms are provided by containers. And also resource limits are enforced, how much CPU, how much memory, how much bandwidth all of that also can be controlled.

So Linux has mechanisms like namespaces, and C groups to provide this isolation and resource limits. And a lot of container frameworks like LXC and Docker. You may have heard these names use this OS functionalities in order to provide this container abstraction. And we also have a lot of container orchestration frameworks like Kubernetes, for example.

And all of these are very popular when you are running a large application. Because application has multiple components. And these components are put in different containers and spread across multiple machines. And these frameworks these orchestration frameworks like Kubernetes make it easier for us to handle these different components that are spread across multiple machines, they provide you ways to handle the lifecycle when a component crashes, restarted, distribute these components across machines.

All of these functionality is provided by these container orchestration frameworks, which is why a lot of real computer systems large computer systems are typically built on top of these orchestration frameworks, so that you do not have to individually manage each component each container on your own.

(Refer Slide Time: 02:54)



So, the next way to do virtualization is using virtual machines. So, virtual machine is actually more heavyweight than container. You have VM which has its own OS and its own applications. And all of these are running on another host OS or a virtual machine monitor or

a hypervisor. So this is your guest, and this is your host. And then this is running on top of your CPU and other hardware.

So the VMM, basically virtualizes the entire hardware to each of these operating systems. So every operating system kind of things, it is running on the underlying hardware fully on its own, even though there are multiple such guest.

And this is a very important building block for cloud computing, in the cloud when you give multiple users access to the same cloud server, you need this level of isolation that is provided by virtualization. So, we have seen this concept of how do VMMs work they work on the concept of trap and emulate. So just like multiple user space processes cannot access hardware they trapped to the OS and the OS accesses the hardware on their behalf.

Similarly, these multiple guest operating systems also will run at a lower privilege level and whenever they have to access the hardware, they will trap to the VMM and the VMM will access the hardware on their behalf. So that is the basic idea of trap and emulate VMMs just like the OS is virtualizing the CPU for processes similarly, the VMM will virtualize all the hardware for the guest OS.

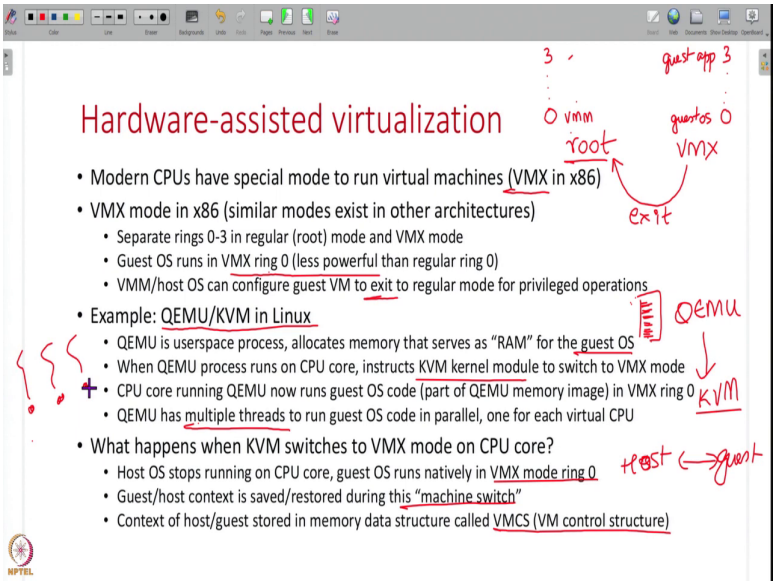
But this is the simple concept is not so easy to implement because modern existing operating systems or CPUs are not easily built for virtualization for example, your guest OS may not run correctly, if you run it at a lower privilege level it may expect to always run at a high privilege level.

So in such cases, we need some extra techniques beyond this simple trap and emulate idea which is there are many different VMMs today based on different ideas. Some VMMs use para virtualization which is they modify the guest OS, so that it works correctly at a lower privilege level.

Some VMMs use full virtualization that is, you cannot modify the OS code, but you will modify the OS binary, the instructions in the OS binary you will translate them so that it works correctly at a lower privilege level.

And finally, you have hardware assisted virtualization, which is the most recent idea, which is where you change the underlying CPU so that no issues come up and existing operating systems can still run without any changes, without changing the code or without translating the binary. So let us look into a little bit more detail on hardware assisted virtualization.

(Refer Slide Time: 05:40)



We have looked at it briefly, we have seen how processes run with hardware assisted virtualization in this lecture, we are going to even dig a little bit deeper, understand how memory is managed, how IO happens and all of those things as well. This is a recap of hardware assisted virtualization that we see. So modern CPUs, for example, x86 has this VMX mode, which is a special mode in which to run virtual machines.

So, what is this VMX mode? Normally in x86, if you have your regular mode, which is called the root mode, you have privilege level 0, 1, 2, 3. 0 is the most privileged level where the operating system is running, 3 is where user applications are running. Now, with VMX mode, you will create a separate mode called VMX mode which also has rings 0, 1, 2, 3. In this VMX, ring 0 is where you will run your guest OS.

And in VMX ring 3 will be your guest applications as usual. And, in your root mode. This is where you will run your host OS, your VMM other processes will run in ring 3, this is your regular system. This extra modes of the CPU have been added. In x86, it is called VMX all architectures have the support, this extra modes let you run existing operating systems without any modification.

How does this VMX mode work? This guest OS when it is running in VMX ring 0, it is less powerful than regular ring 0. You do not want to give the guest OS has complete control over the hardware. Why? Because there are many guests, you do not trust them. So, that this guest OS even though it is running in ring 0, this is actually less powerful and the VMM can configure the guest to exit.

If something happens, if some instructions some problematic instructions are running, if hardware is access any such thing happens, you can configure the guest to exit into the root, into the regular mode where the VMM is running and the VMM can handle the exit, can trap and emulate on behalf of the guest OS.

Even though the guest OS is not aware that this is happening. So the popular example of this way of virtualization, hardware assisted virtualization is the QEMU/KVM hypervisor in Linux. So, this hypervisor has two parts QEMU is the user space process. The hypervisor also has a user space process to do a lot of things that can be done at the user level, you do not want to do everything inside the kernel.

So, the QEMU part of it does whatever is possible to be done at the user space level that is it will allocate memory, it will copy the guest OS code, it will create some new kind of RAM for the guest OS all of that will be done and when you have to actually switch to this VMX mode, then it will talk to the KVM kernel module.

So, your hypervisor has two parts, some part is done, which is non privileged is done in user space. So, the QEMU runs as a user space process and this QEMU talks to KVM which is running inside the kernel, which is privileged and this KVM will do the switch into the VMX mode. So, QEMU has created, copied the guest OS into its memory, guest OS code and then, when QEMU says switch to VMX mode to KVM at this point, what is happening, the CPU core has switched to running this OS code, this guest OS code that is there in QEMU. You are no longer running regular user space code of QEMU but you are running this guest OS code in ring 0.

So, when KVM switches to VMX mode, it is like everything at the CPU level is reconfigured. Your host OS stops running all the processes on the host OS stops running and now your guest to OS directly starts running in privileged mode in VMX ring 0. And where is the guest OS code located. It is located in QEMU itself, QEMU has set all of this up it has copied the guest OS code it has, created the guest OS memory image, everything it has done.

And when you switch from this host mode to the guest mode, you have to still save and restore context, now all the host OS context has to be saved the guest OS context has to be restored. So, this is like a context with just that at the machine level it is happening your entire host OS is being saved, and your guest OS is being run, this is called a machine switch.

And when this is done, you have to store all the values of the CPU registers everything that is stored in a special data structure called VMCS, or VM control structure.

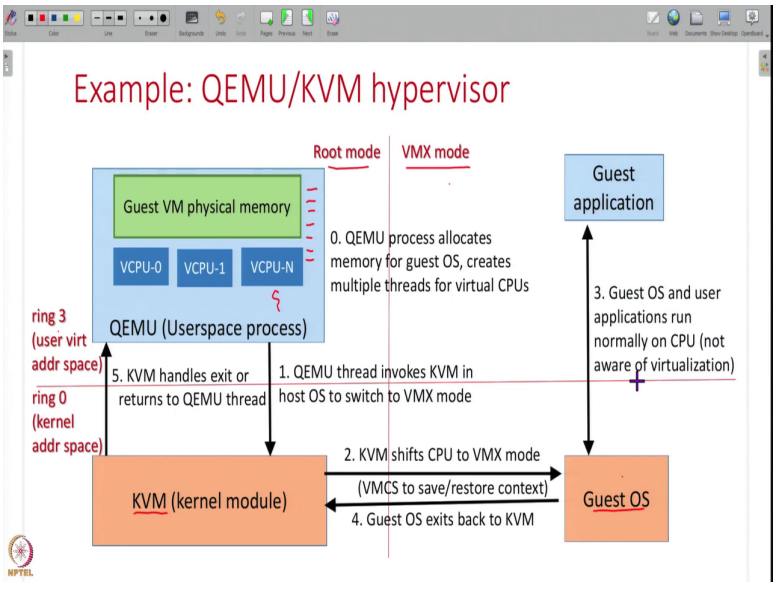
So, to summarize your QEMU creates some sort of a memory for the guest, allocates memory for the guest, copies the guest OS into it just like how OS copied into RAM. Similarly, QEMU copies the guest OS. And then with the help of a kernel module, it switches the CPU to the VMX mode.

And in the VMX mode, the host was stops running, and this guest OS directly starts running. And when you exit back from it, once again, you will restore back the host OS, you will restore back everything that was happening in the host OS, because you have saved the context, and you will restore it back.

And another thing to note is that QEMU can have multiple threads, if you want your guest OS this code to run on multiple CPUs, how will you do it, QEMU will have multiple threads. And on each thread, you will go into VMX mode and run the guest OS code. So, it is almost like now your guest to OS code is running on multiple CPUs in parallel.

If your VM has four CPUs, then QEMU will create four threads and each of these four threads will run on different CPU cores of the host OS. So that the same guest OS code is running four times in parallel, you can do that also in QEMU. So let us just visualize this a little bit.

(Refer Slide Time: 11:52)



So, your CPU can either run in root mode or VMX mode. In root mode, you have the unprivileged ring 3, privileged ring 0, similarly in VMX mode. So, normally user processes are running here, they are running in root mode in your host and QEMU is also one such user process to the host OS, it is nothing different. It just looks like any other process.

What does QEMU do? It allocate a large amount of memory and into that memory it will copy all the guest VM code. As far as the host OS is concerned, this looks like any other code of a process. The host OS does not see this as any different it is any other process. And it also creates multiple threads to execute this code in parallel on multiple CPU cores.

Now, when QEMU has set all of this up, what it will do is, it will tell KVM to switch into VMX mode what that this is a privileged operation, you cannot do it in user space that is why you have split it into a kernel part and a user space part. Whatever is easy, you do it in user space part, the privileged part, you do it inside the kernel module.

So, QEMU one particular QEMU thread that is running this guest OS code will tell KVM; "Hey, switch into VMX mode." At this point what happens, your CPU stops running the host OS it stops running this QEMU process, the host has scheduled QEMU process at this point this QEMU process starts, stops running and you switch to VMX mode and you start running this guest OS code.

KVM shifts the CPU to the VMX mode, you save whatever context whatever the register values here and everything you save into this special area of memory called VMCS. And now your CPU has switched into VMX mode ring 0 and now this guest OS code that QEMU has set up in some memory that starts running and when the guest OS starts to run, of course, it will boot up it will create its own user applications that you create a shell that shell will spawn other processes, we have seen all of this and you keep coming back to the OS running applications all of this will go on, until something happens, some privileged access that the guest OS is not supposed to do until that happens you are in VMX mode.

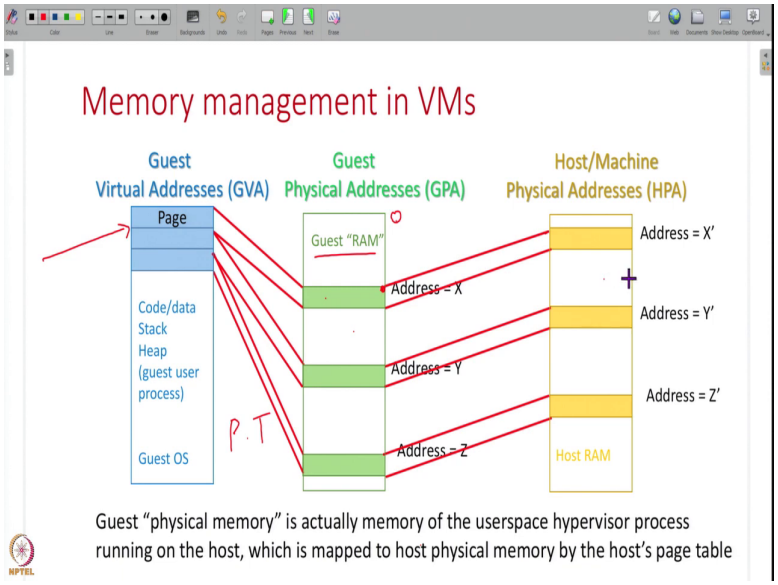
Note that this ring 0 is not all powerful like regular ring 0, if you do something stupid, the VMM will say stop and the guest was will exit back into KVM. Now once again, whatever context you have saved of the host, you will restore all of that context, the guest is stopped, the host OS starts running.

Now this KVM will see this guest has exited. Why has it exited? Maybe it needs something. It will go back to QEMU, QEMU, will see what to do about the guest. It will handle the

event. And then once again, for example, the guest needed some IO, you will come here QEMU will do the IO and then you might go back to the guest again. In this way, you will keep switching between. On every CPU core this will happen independently.

And, every CPU core wherever a QEMU thread is running. It can pause the host OS, switch to the guest OS run it and come back after some time. Note that the host was is not aware of it at all, the host OS context is saved and restored. So, the host OS does not realize it was even paused and some other OS run. All the host OS sees, I am running this QEMU process, that is all.

(Refer Slide Time: 15:17)



So, the next thing that comes up is how is memory managed inside a VM. So, now you have, say your guest virtual machine is running some processes, which have its own virtual addresses, we have seen this, and the guest has a lot of RAM. And from this RAM, it will assign physical frames and it will give to the various guest processes for their code data, stack, heap, whatever it is, the guest OS is doing this.

But whatever RAM that the guest OS thinks it has is not actually physical RAM the guest OS, when it sees a physical address of 0, it is not actual physical address of 0, why, because there are multiple such guests running, it is not just one guest that is running, this is not the only OS running that has access to all the RAM. This is in fact multiple such guests are running and this guest's memory is in fact part of a process, part of the QEMU user space process.

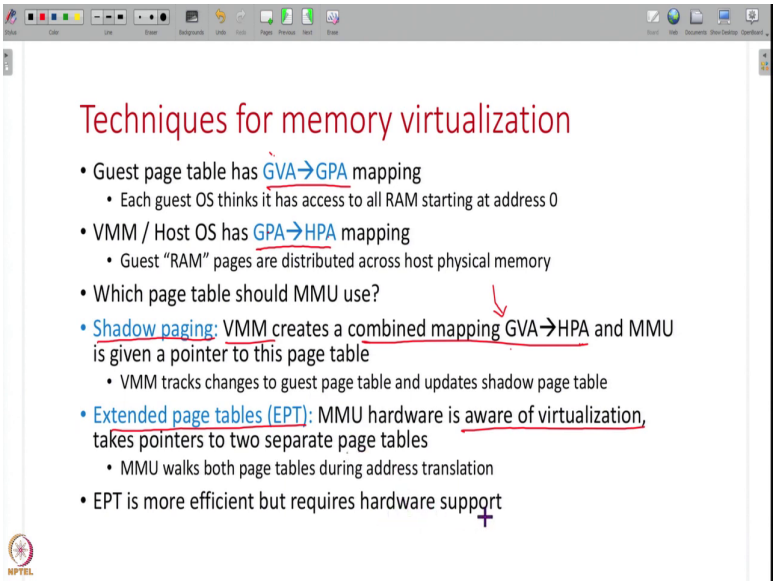
So therefore, what the guest thinks is a certain physical address is actually not the actual physical address in RAM, there is another layer of indirection, mapping from this guest physical to the host physical addresses.

So, the guest OS whatever page tables it has, they keep track of this mapping from guest virtual address space, guest process virtual addresses to guest physical addresses that the guest knows about. But whatever memory you have given to a particular guest, where is it actually in RAM that only the host or the VMM knows about.

Because this guest physical memory is not actually, you have not given any guests full access to the RAM, it is just part of the memory of a regular, QEMU process, that is all. And therefore, that also is mapped into some other host physical address is assigned to it and it is mapped to a different set of addresses.

So, when you translate addresses, you have to go from guest virtual address using this page table, you will translate into guest physical address. And then this guest physical address, the VMM will translate into host physical address, only then you can access actual RAM that is there on your machine.

(Refer Slide Time: 17:32)



So, the problem of memory virtualization how you manage memory inside a VM has gotten more complicated due to one more layer of virtualization here. The guest page table has the guest virtual to guest physical mapping and the VMM has this guest physical to host physical mapping.

Why? Because the VMM knows I have given this memory to this guest, I have given that memory to that guest it knows where it has put the guest's memory, where it has put whatever the guest thinks as its RAM, where it has put in actual host physical memory only the host OS and the VMM know.

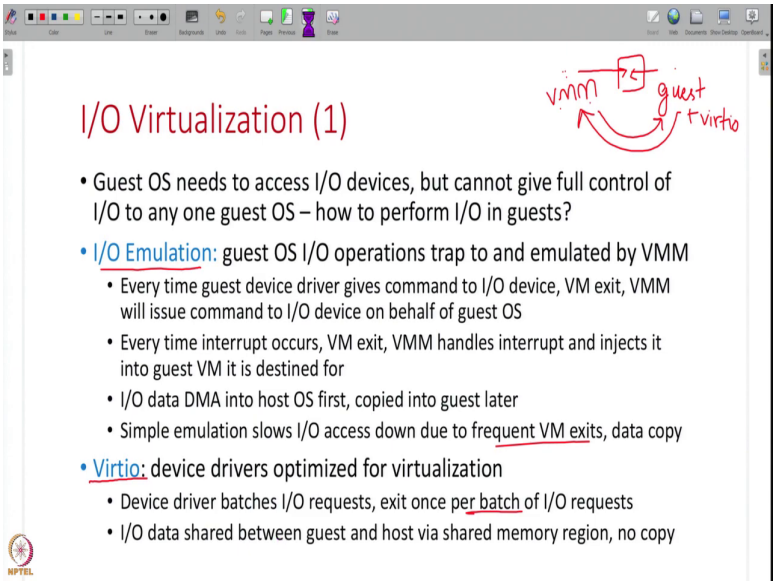
So, there will be one more set of translations like this. So now, you have two different page tables. So, which page table will the MMU use? There are two ways. One thing that the VMM can do is, it can create a combined mapping. This guest virtual address is this guest physical address, you look this up in this table. Finally, from a guest virtual address to the final host physical address, you can create a mapping, combined mappings you can create and put them in what are called Shadow page tables, that is extra page tables, not the guest page table but an extra page table the VMM can create.

And keep updating this page table as the guest updates its page table you keep updating the shadow page table and this shadow page table will be used by the MMU. So whenever some virtual address is being used, you will use this translation to translate it into actual RAM locations. This is one idea.

The other idea is instead of the VMM combining these page tables, you can once again tell the MMU only look here are two different page tables, you combine them yourselves. That is the MMU hardware can be made aware of virtualization, you can say here are two different page tables. That concept is called extended page tables.

And it can take pointers to two separate page tables and walk both page tables during address translation. Of course this is more efficient because the hardware is doing the work but it also requires some hardware support. So next we are going to understand how IO virtualization happens that is when a guest does IO what exactly happens.

(Refer Slide Time: 19:48)



So, we have seen that once again just like how guest cannot be given full access to RAM. Any guest OS cannot be given full access to IO devices also because multiple VMs are sharing the same server and there are security issues involved. So when the guest needs to access IO, it has to go through the VMM.

And how do we do this? There are many ways of doing it. The simplest is what is called IO emulation. That is, the guest OS tries to do any IO operation, for example, in VMX ring 0, then this will trap into the VMM. There will be a VM exit; you will trap to the VMM. And then the VMM, the QEMU user process, for example, if the guest wants to open a file, the QEMU user process can open a file, if the guest wants to read the QEMU user process can read a file. You can just emulate whatever the guest wants to do inside the VMM.

And similarly, every time an interrupt occurs, the VMM can first look at the interrupt and if it is for the guest, it can then inject the interrupt when the guest starts it can tell the guests that look you have an interrupt. So, every time there is an IO command given to the device or an interrupt occurs, you will exit to the VMM, VMM will handle it.

Similarly, when IO data comes from the device also, you will first DMA it into the VMM. And then the VMM will give it to the guest, everything goes through the VMM. So, this is a simple idea, It works reasonably fine for slow IO devices. But once you have a very fast IO devices, there will be so many VM exits that it might slow down your VM significantly, and you are also copying data DMAing it here, then copying it here, this overhead might get a little too much for high speed IO devices.

So, what you do for high speed IO devices is, today you have special device drivers available that are called virtio device drivers. so these are device drivers that are optimized for virtualization inside your guests, you do not use the regular device drivers that come with your operating system, but you will install new device drivers.

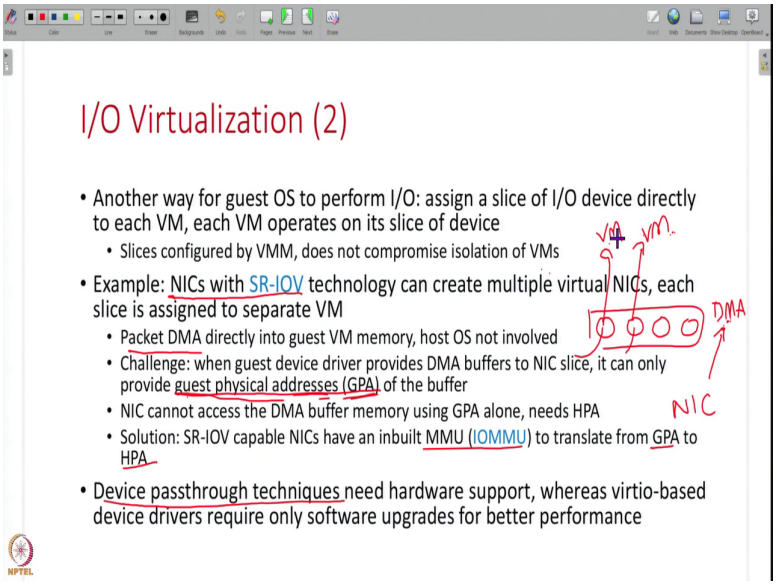
And what do these virtio device drivers do instead of exiting for every IO operation, what they will do is they will batch IO requests, you will collect multiple IO requests instead of regular device driver anytime you have to give a command to the disk it will just give the command to IO device disk or whatever.

But these virtio device drivers know that they are not accessing the real hardware. So therefore, they will collect all these IO requests and once per batch, they will exit into the VMM, so that you avoid these frequent back and forth between the VMX mode than the regular mode.

Similarly, when you have to share IO data between the guest and the host, you will set up a separate shared memory region into which VMM and the guest both have access to and they can read the shared data. So, these are all extra changes needed to device drivers which are optimized for virtualization.

So the other technique is instead of going through the VMM and the host even in virtio you are going through the host is that it is optimized. Another way you can give a guest VM its own slice of the IO device. If the IO device supports this, you can give each VM a slice of the IO device, so that they are not interfering with each other.

(Refer Slide Time: 23:12)



So, an example is modern network cards come with a feature called SR-IOV for single root IO virtualization. In simple terms, what this means is that you can make one network card look like multiple different network cards, which are isolated from each other. And you can give each network card to a separate VM.

So, traffic coming to this network card goes only to this VM, traffic to this network card goes only to this VM in this way they are not interfering with each other, at the same time you do not have to give data to the host VMM first and then copy to the VM. So, what these SR-IOV network cards do is they have separate NICs, which are of course configured these slices are configured by the VMM. But once you configure them a VM has exclusive access to its slice.

For example, things like packet DMA, they will be directly DMA into the VM memory you no longer DMA to the host memory first and then copy to the VM memory. But this is not straightforward. Why? Because the guest OS can only provide addresses of the DMA buffers in the form of guest physical addresses.

So in order for the NIC to do a DMA into memory, you need to know what is the physical address of this DMA buffer only then can the NIC copy into it. But if the guest OS is giving it dummy physical addresses guest physical addresses which are not correct, how will the DMA happen into RAM it cannot happen.

Therefore, the solution for this is that SR-IOV capable NICs also have something like an MMU built into them, which will translate from this guest physical to actual host physical

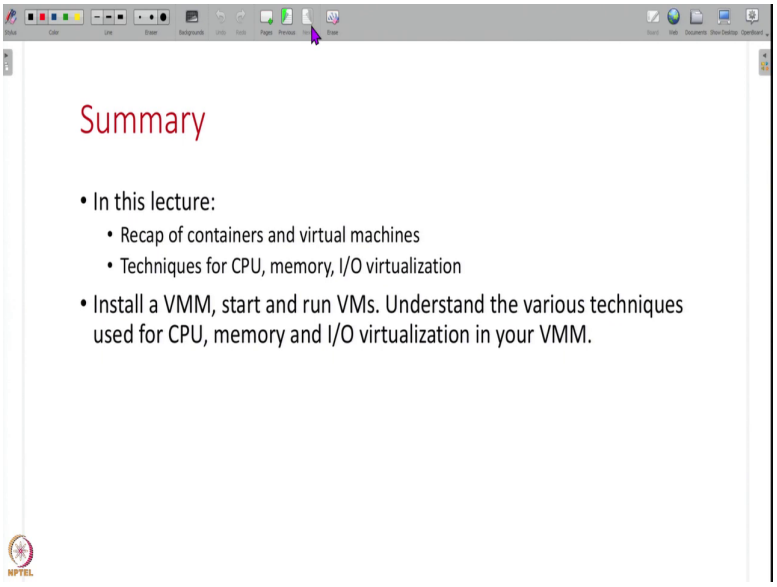
addresses that is they are doing the job of the MMU also, in addition to doing DMA, you are also doing the job of the MMU you are getting some pointer to a page table and you are translating addresses so that you can correctly do DMA into physical memory. So that with this IO MMU, you can directly DMA data into the guest OS DMA buffers, so that you do not have to go through the host OS for DMA.

So this is a very advanced technology that requires a lot of hardware support. So, these kind of ideas are called device pass through techniques, that is, you are bypassing the VMM. And directly the devices being assigned to the VM. So, such techniques while they are efficient, they require more hardware support.

On the other hand, techniques like virtio are just device driver upgrades, in your VM, you do not change anything in the hardware, you just install virtio driver, you get better performance. So there is a trade off. Are you willing to change the hardware? Or are you only sticking to software level changes?

So based on these different parameters, you can decide for your application if you are running inside a VM, then which techniques should you use for IO virtualization. So in this lecture, we have just done a recap of containers and VMs that we have studied in an earlier lecture. And we have added a few more concepts to what we have studied before.

(Refer Slide Time: 26:28)



Specifically, we have seen how memory and IO are virtualized. So now you have all the techniques that are needed to virtualize the CPU, memory and IO, all the techniques you have

understood and different VMMs use different combinations of these techniques. Some may use full virtualization, paravirtualization or hardware assisted, some may use extended page tables or shadow page tables or SR-IOV or virtio. It really depends different applications depending on the need of the application and the technology in the VMM different techniques can be used.

So, as a small exercise, install a VMM, run a few VMs and try to understand by reading the documentation of the VMM what techniques it is using for CPU, memory and IO virtualization. For example, you may notice some special virtio device drivers that are running. Are they running? Are you using? Does your hardware have SR-IOV capability? Are you using hardware assisted virtualization? So understand these things for any VMM that you are using in order to see an application of these concepts in your real life? So that is all I have for this lecture. Thank you, everyone.

And we will see you in the next week when we cover a completely new topic of how networking works, how computer networks specifically the internet works. That is what we're going to see starting next week onwards. So, the past three weeks have completed our discussion of operating systems and we will move on to the next set of topics in the course. Thank you all and see you next week.