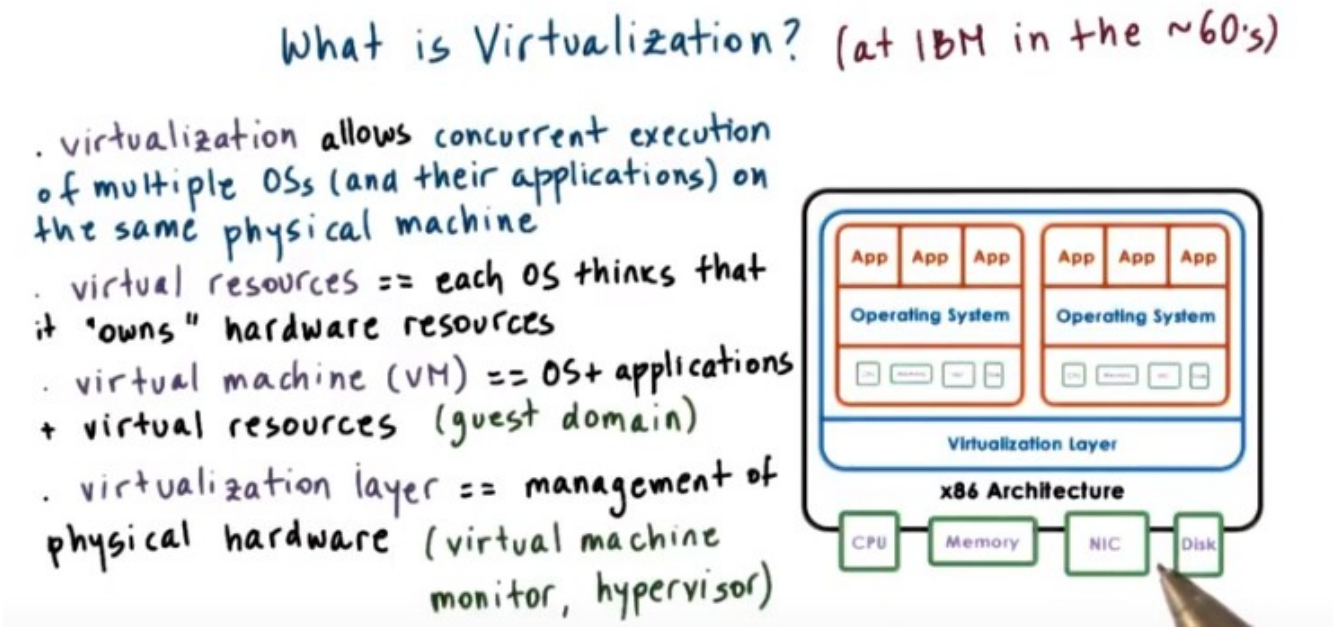
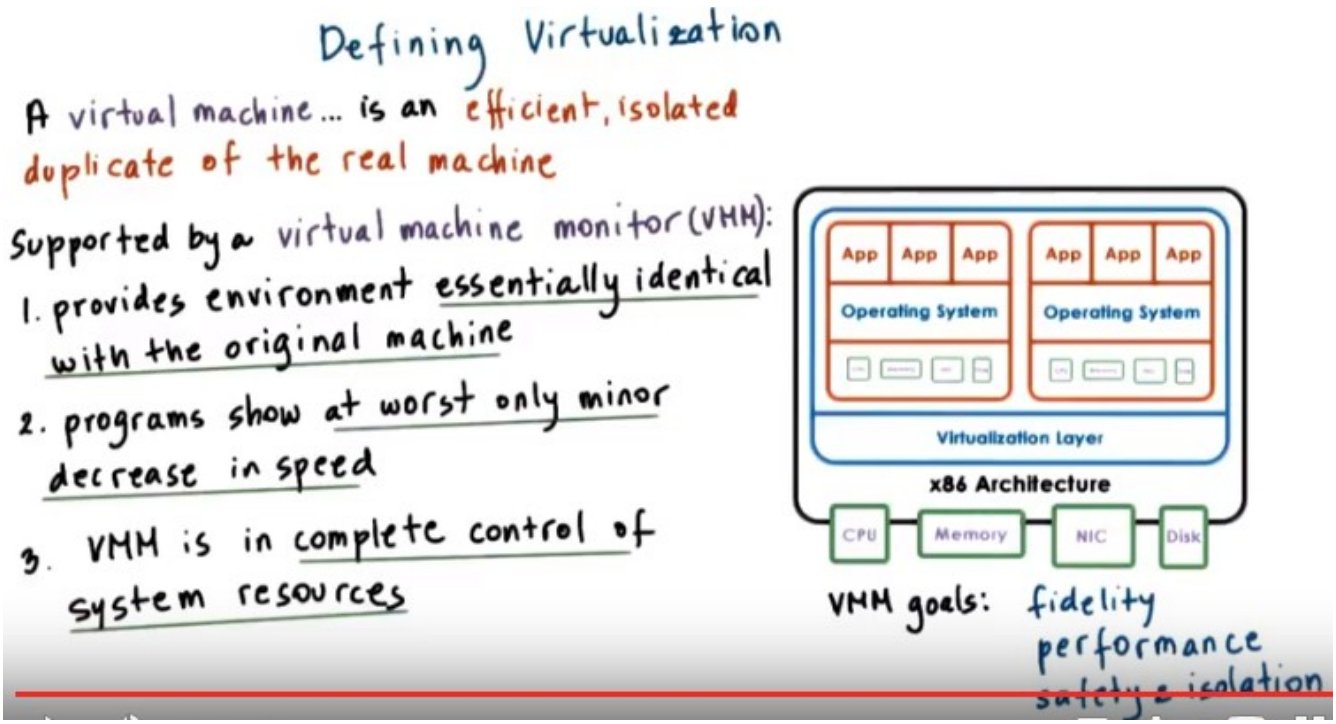
1. In this lesson, we will talk about system virtualization. It's another important component of the system software stack that's involved in the management of the physical resources. We will explain what is virtualization and why it is important. And we will discuss what are some of the key technical approaches that are part of popular virtualization solutions, such Xen and KVM and VMware CSX. In addition, at the end of the lesson we will discuss what are some of the hardware advances that have occurred over the last decade in response to virtualization trends. Specifically in the context of the x86 architecture. A lot of the content in this lecture is also summarized in a paper by Rosenblum and Garfunkel. Virtual Machine Monitors, Current Technologies and Future Trends. Note that this is a paper from 2005, so much more has happened in the virtualization space than what this paper what this paper describes. But it gives, does serve as a useful reference.

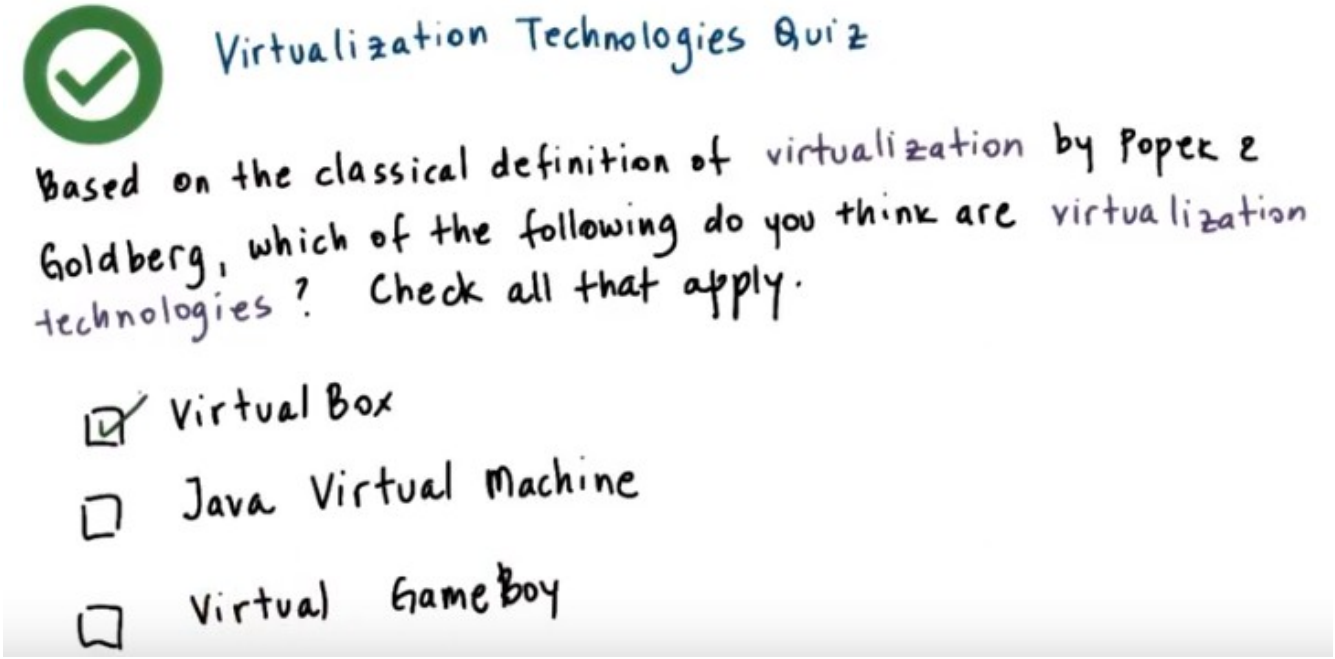


2. Virtualization is an old idea. It originated in the 60s at IBM when the norm of computing was that there were a few large mainframe computers (mainframe computers 意思見第 8 段) that were shared by many users and many business services. In order to concurrently run a very diverse workloads on the same physical hardware, without requiring that a single operating system be used for all of the o- applications for all of the possible purposes, it was necessary to come up with a model where multiple operating systems can concurrently be deployed on the same hardware platform. With virtualization, each of the operating systems that are deployed on the same physical platform has an illusion that it actually owns the underlying hardware resources. Or at least some smaller portion of them. For instance, the operating system may think that it owns some smaller amount of memory compared to the memory that's actually available on the physical machine. Each operating system together with its applications as well as the virtual resources that it pings at us is called a virtual machine, or VM for short. This represents a virtual machine because it is distinct from the physical machine that is natively managed by some software stack. Virtual machines are often referred to as guests such as guest VMs or also referred to as domains, like a guest domain for instance. To support the coexistence of multiple virtual machines on the same physical machine, require some underlying functionality in order to deal with the allocation and the management of the real, physical hardware resources. In addition, it's necessary to provide some isolation guarantees across these VMs. This is provided by the virtualization layer, that is refered to as virtual machine monitor or hypervisor, also. There are multiple ways in which the functionality, that's required by the virtualization layer, can be designed and implemented. And in this lesson we will describe some of the key virtualization related techniques.



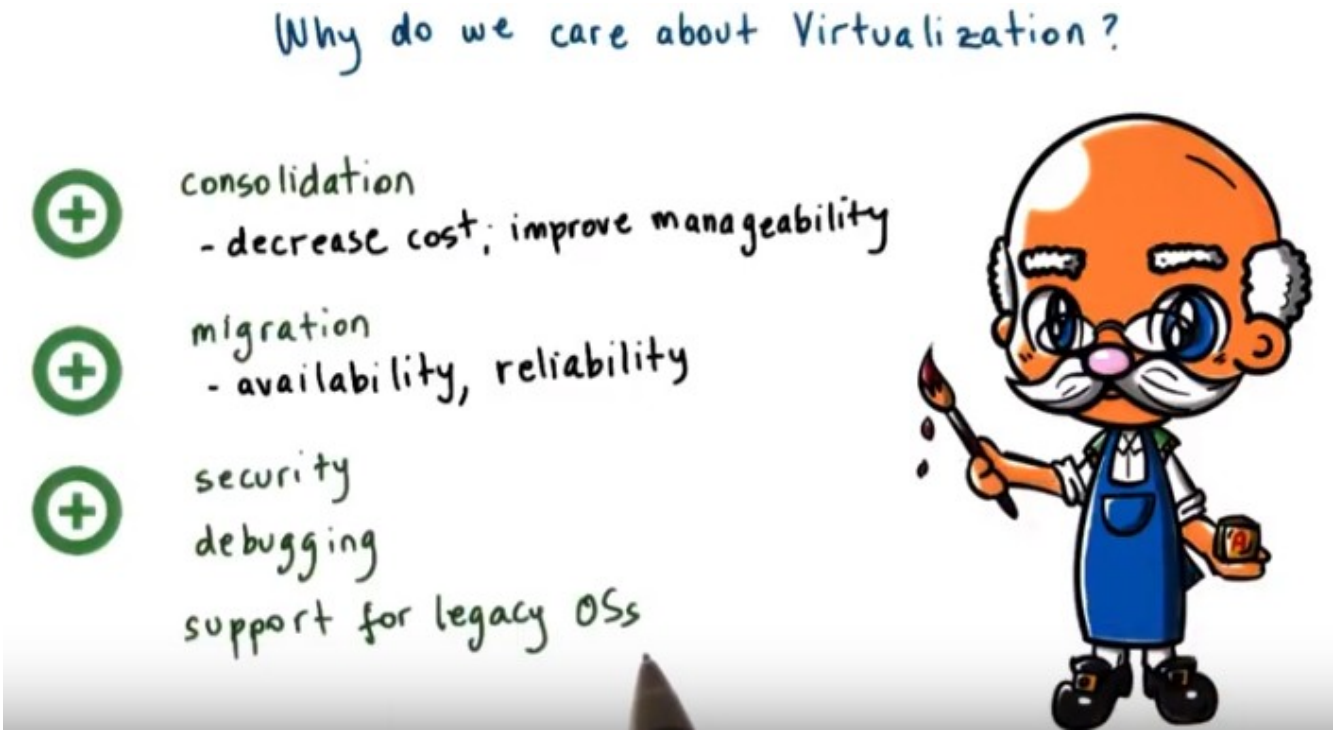
3. A formal explanation of Virtualization is given in an old paper by Povek and Goldberg from 1974. There is a link to that paper in the instructor's notes. We'll go through the ideas that are presented in that paper and we will explain how to define Virtualization. The definition states that a virtual machine is taken to be an efficient isolated duplicate of the real machine. Virtualization is enabled through a virtual machine monitor, or VMM. This is the layer that enables virtual machines to exist. As a piece of software, the virtual machine monitor has three essential characteristics. First: The virtual machine monitor provides an environment for programs that's essentially identical with the original machine. As I said the capacity may differ, however it would be the same kind of CPU that the virtual machine thinks it has. The same types of devices, etc. What this means is that as the, one of the goals of the virtual machine monitor is that it must provide some fidelity(忠實,保真度) that the representation of the hardware that's visible to the, in the virtual machine matches the hardware that's available on the physical platform. Second, programs that run in this environment show at worst only minor decrease in speed. Now, clearly we have a situation where the operating system and its processes in the virtual machine are given only two CPUs as opposed to the four that are available on the physical machine. This won't execute exactly at the same level, it wouldn't be able to complete the programs as fast as if they were in natively. However the goal of the virtual machine monitor is that if the virtual machine were given the exact amount of resources as the physical machine, then the operating system, and the processes, would be able to perform at the same speeds. So another goal of the virtual machine monitor, is to deliver performance to the VMs that's as close to the need of performance as possible. And last, the VMM is in complete control of the system resources. This means that the virtual machine monitor has full control to make decisions, who accesses which resources and when, and it can be relied upon to ensure safety and isolation among the VMs. This doesn't mean that every single hardware access has to be inspected by the VMM layer. Instead, what this means is that the virtual machine monitor determines if a particular VM is to be given direct hardware access. And also that, once those decisions are put in place, a virtual machine can not just change those policies, and potentially hurt other collocated VMs. So one of the goals for the virtual machine monitor is that it has to provide for safety and isolation guarantees.

4. Based on the classical definition of virtualization that's provided by Popec and Goldberg, which of the following do you think are virtualization technologies? The choices are Virtual Box, the Java Virtual Machine, and Virtual GameBoy, which allows you to run Nintendo GameBoy games on your computer. we have included links in the instructor notes in case you want to find out more about these choices. You should check all the answers that applied that are correct for this question.



Instructor Notes Quiz Help :
Classical Definition of Visualization: Visualization (or a virtual machine) is an efficient, isolated duplicate of the machine.

5. The only correct answer is Virtual Box, but let's talk about why. First you must recognize that the goals of the system or platform virtualization are different than what's intended by the Java Virtual Machine or hardware emulator, as in the case of Virtual GameBoy. Remember that according to Popek and Goldberg, a virtual machine is an efficient isolated duplicate of the real machine. JVM is a language run time which provides system services and portability to Java applications, that it is very different than the underlying physical machine. Also, emulators like virtual GameBoy, they emulate some hardware platform, the GameBoy platform, but again, this is very different than the hardware platform where this emulator is running. With system virtualization, this is, in the case of Virtual Box, the physical hardware that's visible to the virtual machine is identical, or at least very, very similar, to the physical platform that actually is supporting the execution of the virtual box, yes.

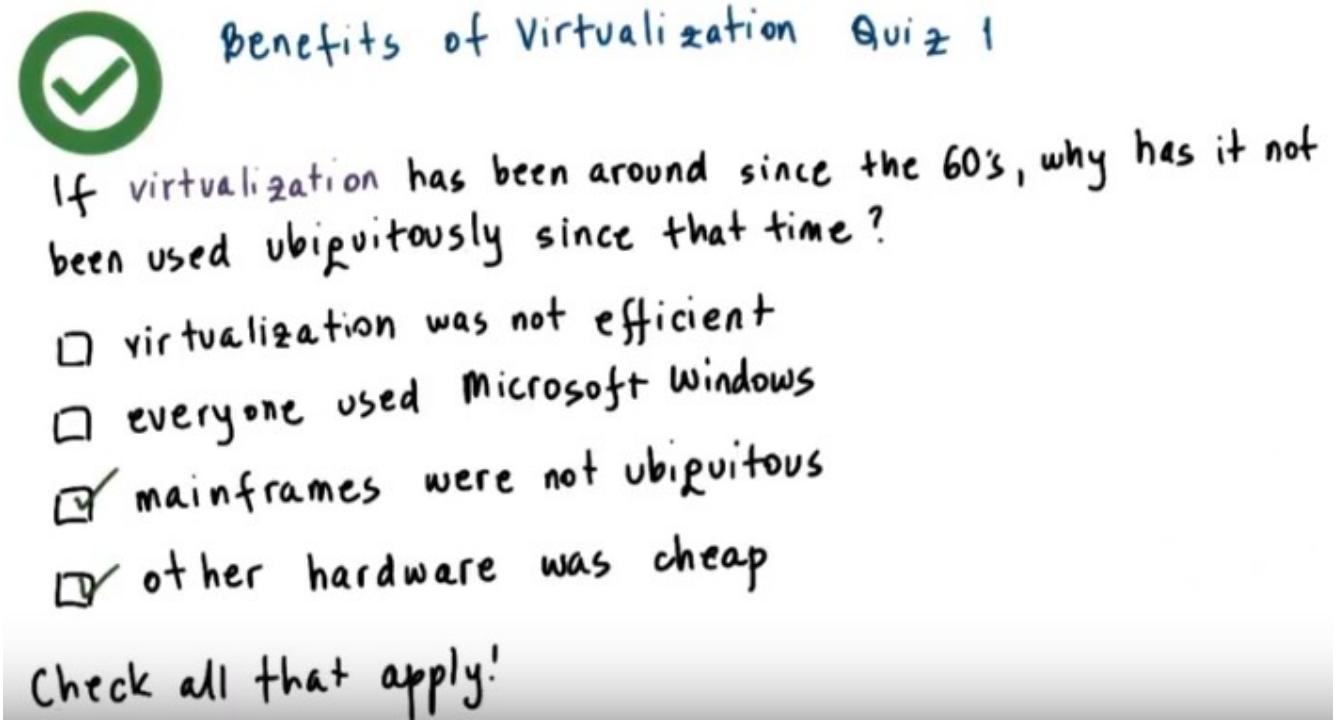


Consolidation: 團結, 合併, 鞏固

6. So why do we care about Virtualization? Well, first it enables consolidation. And consolidation is this ability to run multiple virtual machines, with their operating systems and applications, on a single physical platform. Consolidation then leads to improved cost efficiency. With fewer machines, with less space, potentially with fewer admins, and with fewer electrical bills, we will be able to run the same kind of workload. So consolidation delivers benefits because it allows us to decrease costs, and also to improve the manageability of the system. In addition with Virtualization, once the operating system in its applications are nicely encapsulated in a VM, and decoupled from the actual physical hardware. It becomes more easy to migrate the OS in the applications from one physical machine to another physical machine. Or even to copy and clone them onto multiple physical machines at the same time. As a result, Virtualization leads to mechanisms that are useful in order to provide greater availability of the services. Like, for instance, if we have an increase in the load, we can just, create multiple VMs and address that issue. So, increase the availability of the system. And also to provide solutions that improve the reliability. For instance, if we detect that a particular hardware node is getting hot and likely will fail, we can just migrate those VMs onto some other physical nodes. There are other benefits to Virtualization. For instance because the OS and the applications are nicely encapsulated in a virtual machine. It becomes more easy to contain any kinds of bugs, or any kinds of malicious behavior, to those resources that are available to the virtual machine only, and not to potentially affect the entire hardware system. Speaking of debugging, in particular, Virtualization has delivered some other benefits, that it has become a very important platform for operating systems research. It lets systems researchers quickly introduce new operating system feature and has them in the OS that's encapsulated in the VM. And then they have ability to more quickly view the effects of that and debug it as opposed to a more traditional cycle, which would have included hardware restarts of the machines and then searches through the log files or the error files, etc. Virtualization is also useful because it provides affordable support for legacy operating systems. With Virtualization it's no longer necessary to designate some hardware resources for some older operating system just because

it's needed to run one particular application, for instance. Instead, that legacy OS and applications can be packaged in a virtual machine and then can be co-allocated, consolidated, on the same hardware resources that support other VMs and other applications. And there are many other benefits to Virtualization. These are some of the key ones, though.

7. Let me ask a question now, if virtualization has been around since the 60s, why do you think it hasn't been used ubiquitously since then. Here are the possible choices. Virtualization was not efficient. Everyone used Microsoft Windows. Mainframes were not ubiquitous. Or other hardware was cheap. You should check all that apply.



ubiquitously: 到處存在地

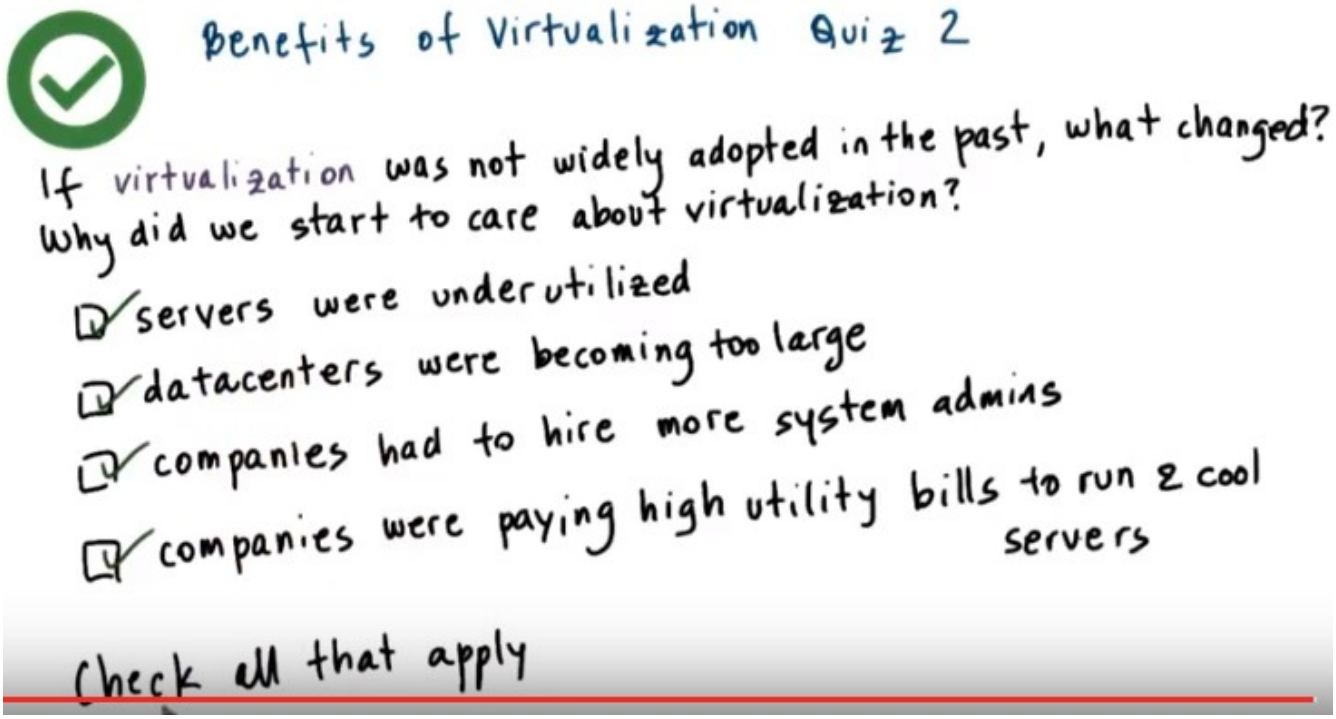
8. The correct answers to this question are that, the fact that virtualization wasn't dominant technology all these years is that first mainframes weren't ubiquitous, and then other hardware was cheap. Majority of the companies did not necessarily run mainframe computers. They ran servers that were based on x86 architecture mostly. This was affordable, and it was always much simpler to just add new pieces of hardware than to try to figure out how to make multiple applications and multiple operating systems coexist on that same hardware platform. This trend of just buying more machines if you need to run a different kind of operating system to support different applications continued for a few decades actually.

From wiki:
大型電腦（英語：mainframe computer），又稱大型電腦、大型主機、主機等，是從 IBM System/360 開始的一系列電腦及與其相容或同等級的電腦，主要用於大量資料和關鍵專案的計算，例如銀行金融交易及資料處理、人口普查、企業資源規劃等等。有些大型電腦可以同時執行多作業系統，因此不像是一台電腦而更像是多台虛擬機器，因此一台主機可以替代多台普通的伺服器，是虛擬化的先驅。同時主機還擁有強大的容錯能力。IBM 目前控制主機市場超過 90%的市場份額。

與超級電腦的區別
超級電腦有極強的計算速度，通常由於科學與工程上的計算，這些計算的速度受運算速度與記憶體大小所限制；而主機運算任務主要受資料傳輸與轉移、可靠性及並行處理效能所限制。

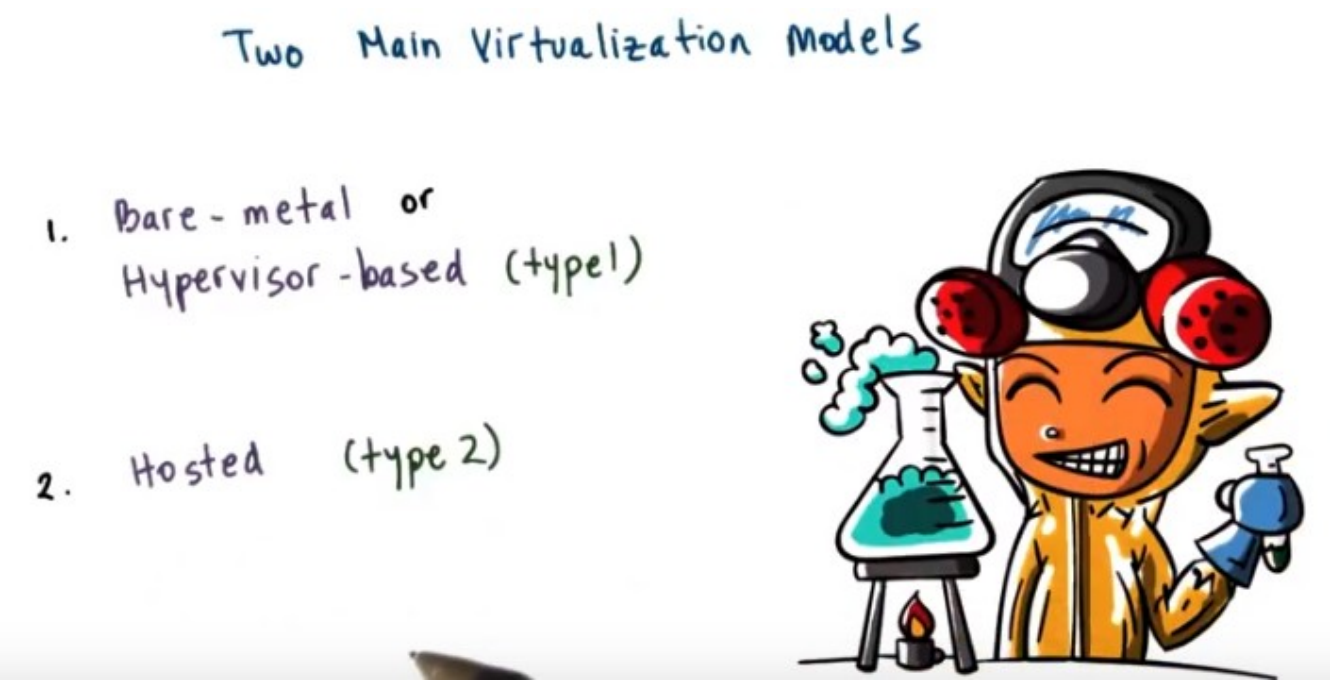
主機更傾向於整數運算，如訂單資料、銀行資料等，同時在安全性、可靠性和穩定性方面優於超級電腦。而超級電腦更強調浮點運算效能，如天氣預報。主機在處理資料的同時需要讀寫或傳輸大量資訊，如海量的交易資訊、航班資訊等等。

9. Let me ask a second question now. If virtualization was not widely adopted in the past, what changed? Why did we start to care about virtualization all of a sudden? The choices are, servers were under utilized, data centers were becoming too large, companies had to hire more system admins, or companies were paying high utility bills to run and cool the servers. You should check again all of the choices that apply.

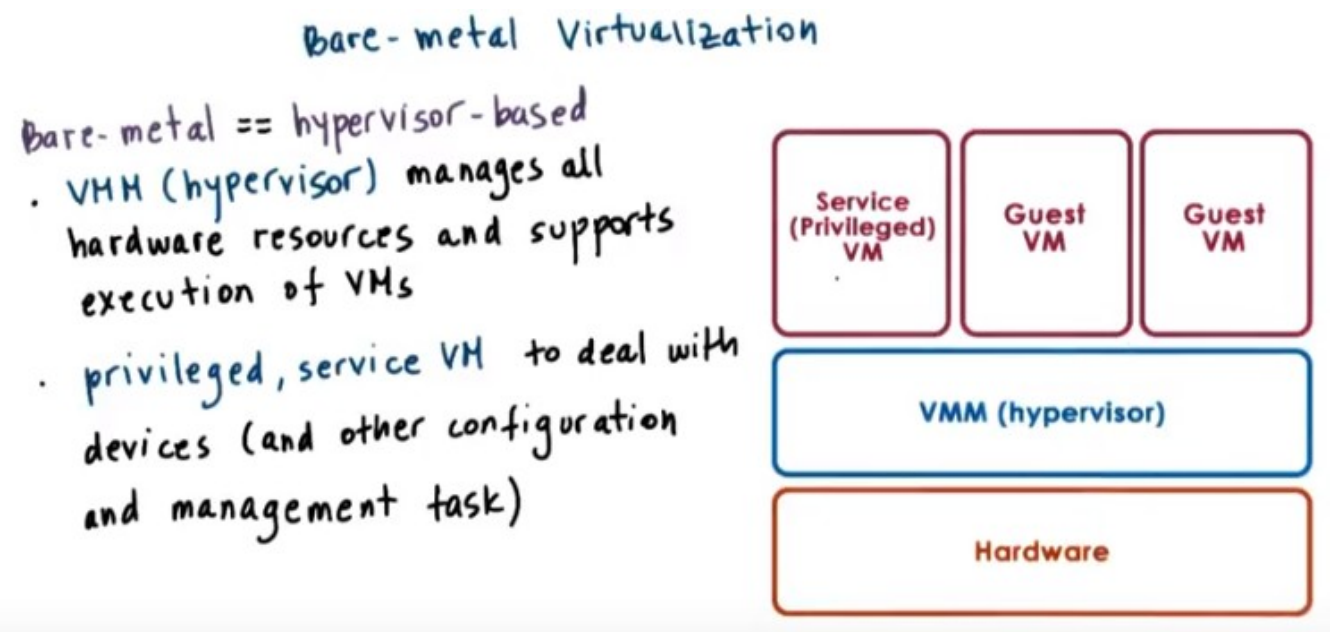


10. Using the model of just buying new hardware whenever there is a need to run a slightly different operating system or to support slightly different applications, in that process, data centers (可能是指處理 data 的機房) became too large. At the same time, some of these servers were really very underutilized (因為一個不常用的係統要佔一台 server). In fact, on average, the utilization rates in data centers were around 10, 20% tops. So as a result, companies, now that they had to manage these large data centers with lots of machines, they had to hire more system admins. So this is a correct choice. And at the same time, they had to spend more money to host all of those machines, to power them, so they have higher electric bills, to cool the data centers since the machines need to operate within certain temperature variability. So this was basically burning through their operating budget. The fact that all of these choices are correct also translated at that time of something like companies spending 70% of their IT budget on operating expenses (如管理這些電腦) versus on capital expenses (如買新東西), like actually buying new hardware or new software services. So then it became apparent that it was important to revisit virtualization technology as a mechanism for consolidating some of these workloads on fewer hardware resources that will be easier to manage them and more cost-effective to

actually run them. And this is why the industry and the community overall revisited these solutions that were in existence for certain types of platforms for decades at the time.



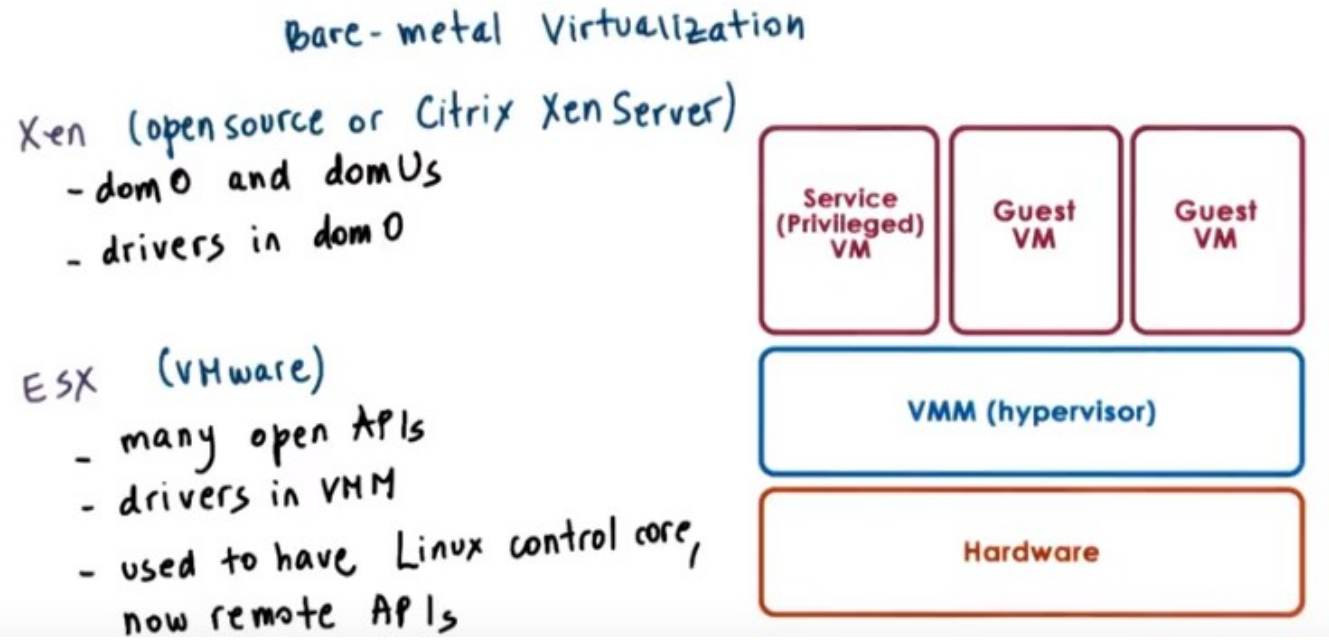
11. Before describing the technical requirements for virtualization, let's take a look at the Two Main Virtualization Models. The two popular models for virtualization are called 'Bare-metal or Hypervisor-based' and 'Hosted'. They are also often referred to as (type 1) for the Hypervisor-based model and (type 2) for the Hosted model for virtualization solutions.



記住: VMM = hypervisor

The Bare-metal model is like what we illustrated before. A virtual machine monitor or hypervisor is

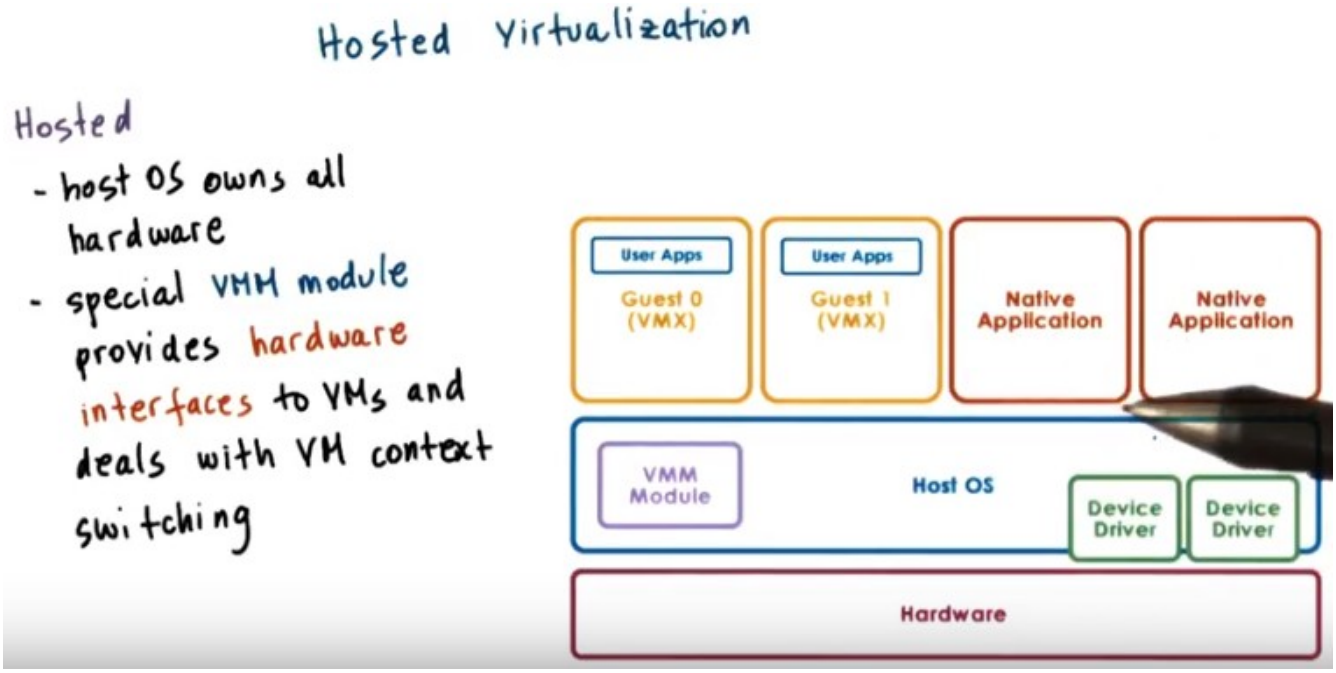
responsible for the management of the physical resources, and it supports execution of entire virtual machine. One issue with this model are devices. According to the model, the hypervisor must manage all possible devices, or stated differently, device manufacturers now have to provide the device drivers, not just for the different operating systems, but also for the different hypervisors that are out there. To illuminate this, the hypervisor model typically integrates a special virtual machine. Like a service VM that runs a standard operating system and has full hardware privileges to access and perform any kind of hardware manipulation just like if you were in need of BMV~ hardware. It is this privilege VM then that would run all of the device drivers and would have control over how the devices on the platform are used. The service VM also runs some other management tasks and configuration tasks that specifies exactly how the hypervisor would share the resources across the guest VMs, for instance.



上圖中的 Xen 和 ESX 都是 hypervisor
注意 Xen 和 ESX 中 driver 所在的位置之不同

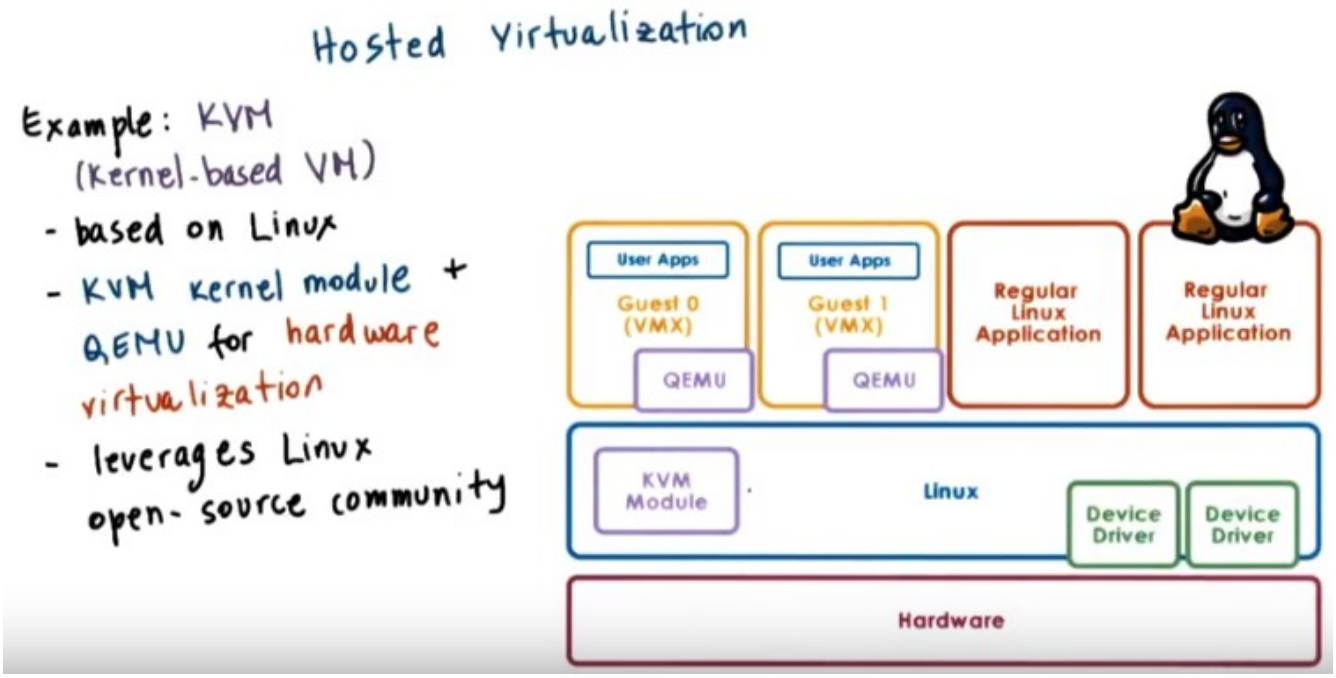
This model is adapted by the Xen virtualization solution, as, and also by the VMware's hypervisor, the ESX hypervisor. Regarding Xen, both when it comes to the open source version, as well as the version that's supported by Citrix, the Xen server. The VMs that are run in the virtualized environment are referred to as domains. The privileged domain is called dom 0, and the guest VMs are referred to as domUs. Xen is the actual hypervisor and all of the drivers are running in the privileged domain, in dom 0. So the management of all of the devices has to involve the execution of the drivers that are part of dom 0. ESX: Given that VMware and its hypervisors were first to market, VMware still owns the largest percentage of virtualized server cores. So these server cores run the ESX hypervisor. Even this fact, viewers in a position to actually mandate(命令) from the vendors that they do provide the drivers for the different devices. That are going to be part of the hypervisor (即 VMM). Now, this is not as bad because this is really targeting the server. Part of the market space and in servers in data centers there not going to be that many devices or there are going to be relatively a fewer devices compared to what you would see on your laptop or desktops or in general the client platforms. To support a third party community of developers VMware actually also exports a number of APIs. So this not just for the sake of the developers, but also for users when they want to configure exactly the kinds of policies that will be enforced by the hypervisor. And in the past the ESX architecture was such that there was a control

core, a core domain, if you will, that was based on a regular operating system, it was based on Linux. But the right now, all of the configuration related tasks are configured via remote APIs.



Host OS: manage hardware
VMM module: provide hardware interfaces

12. The second model is called the Hosted model. In this model, at the lowest level, there is a full fledged(喂養, 長羽毛) host operating system that manages all of the hardware resources. The Host OS integrates a virtual machine monitor module (即中間那層左邊那個), that's responsible for providing the virtual machines with their virtual platform interface and for managing all of the context switching scheduling, etc. As necessary, this VMM module will invoke drivers or other components of the host operating system as needed. This one benefit of this model is that it can leverage all of the services and mechanisms that are already developed for the host operating system. Much less functionality needs to be redeveloped for the VMM Module in this manner. Also, note that on this host operating system, you may run Guest VM's through the Virtual Machine Module, but you can also run Native Applications directly on the host operating system as you would in general.

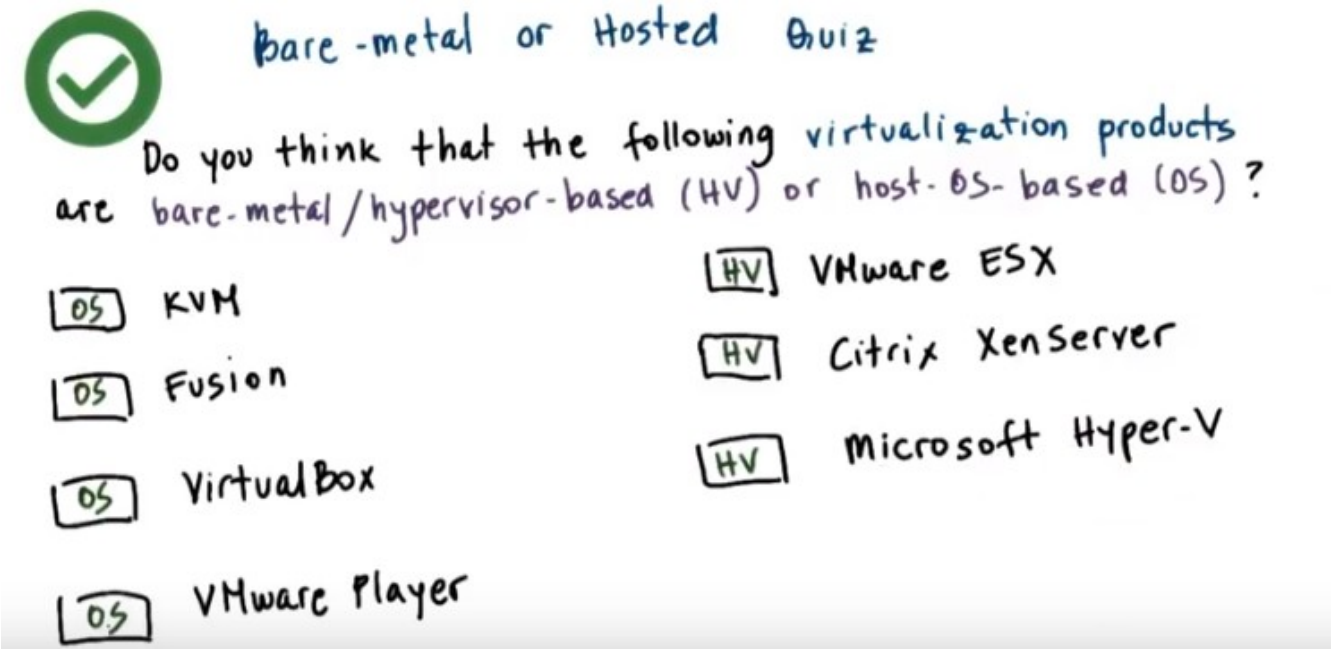


KVM:
Host OS: Linux
VMM module: KVM Module

One example of the Hosted model is KVM, which stands for kernel-based VM. That's based on the Linux operating system. The Linux host provides all aspects of the physical hardware management and just like any regular OS, it can run directly regular Linux applications. The support for running Guest Virtual Machines is through a combination of our kernel module, that's the KVM module and a hardware emulator called QEMU. We said that the goal of virtualization is to provide identical hardware. So here, this emulator is used as a virtualizer. It really matches the underlying hardware resources, the X86 Intel or AMD. The support for running SVMs in KVM is through a combination of a kernel module KVM and a hardware emulator, QEMU. We said that virtualization the intent is to provide identical hardware, so this QEMU emulator isn't emulating some bizarre hardware platform. Instead, it's used in what's called a virtualizer mode. So the resources that are available to the Guest VM are actually the exact hardware resources from the physical platform, except that this virtualizer intervenes during certain types of critical operations or specific instructions or re, relative to pass control to the KVM Module and the Host OS. One example of that would be any aspect of IO management, because all of the support for devices, the device drivers are handled as part of the Linux operating system the Host OS. A huge benefit for KVM has been that it's able to really leverage all of the advances that are continuously being contributed to the large Linux open-source community. Because of this KVM can quickly adapt to new hardware features, new devices, new security, bugs or similar things. In fact, the KVM Module was originally developed as a Linux module in order to allow regular use of Linux applications to take advantage of some of the virtualization related hardware that started appearing in commodity platforms. All of the sudden, users realized that this can be useful to actually run guest operating system and regular virtual machines. And so three months later, KVM was

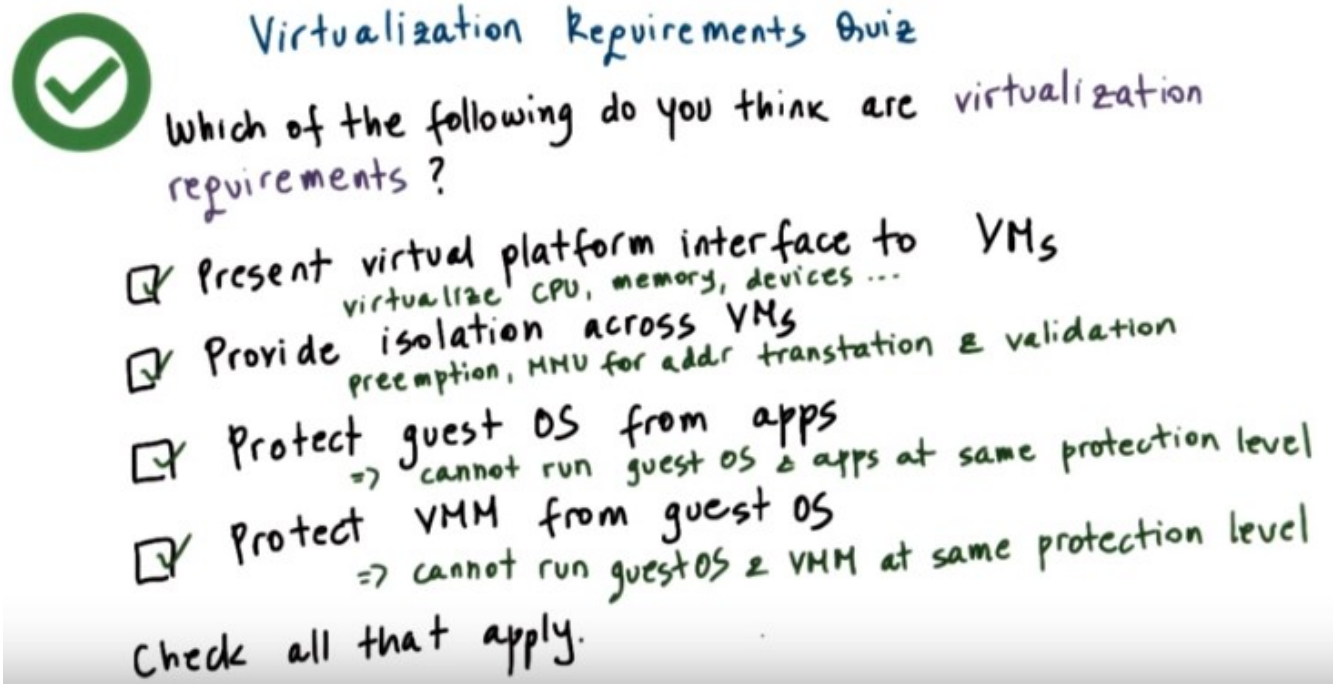
an actual virtualization solution that was part of the mainstream Linux kernel.

13. For a quiz, I would like you to do a quick survey of some virtualization products. The question that you need to answer is, do you think that the following virtualization products are based on bare-metal or hypervisor-based virtualization, or host-OS-based virtualization? The product you need to look at are KVM, Fustion, VirtualBox, VMware Player, VMware ESX, Citrix XenServer, and Microsoft Hyper-V. And you should mark HV for hypervisor-based solutions or OS if you think the solution is host-OS-based.

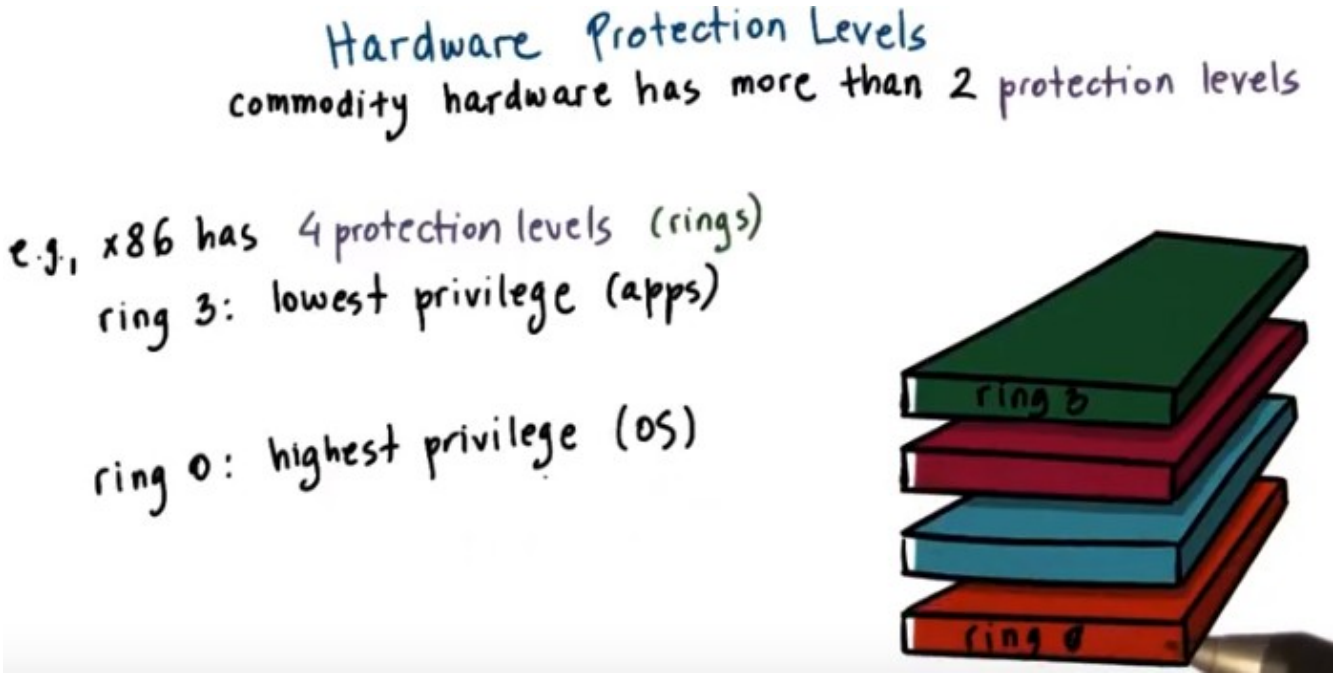


14. As we stated earlier, VMware ESX and the Citrix XenServer are both hypervisor- based solutions. So is Microsoft's Hyper-V. All of the other products are hosted. However, it's important to note that at KVM, the host OS switches to a, a mode, a module in order to assume a hypervisor-like role. So, the rest of the operating system really plays a secondary supporting role like, like a privileged partition.

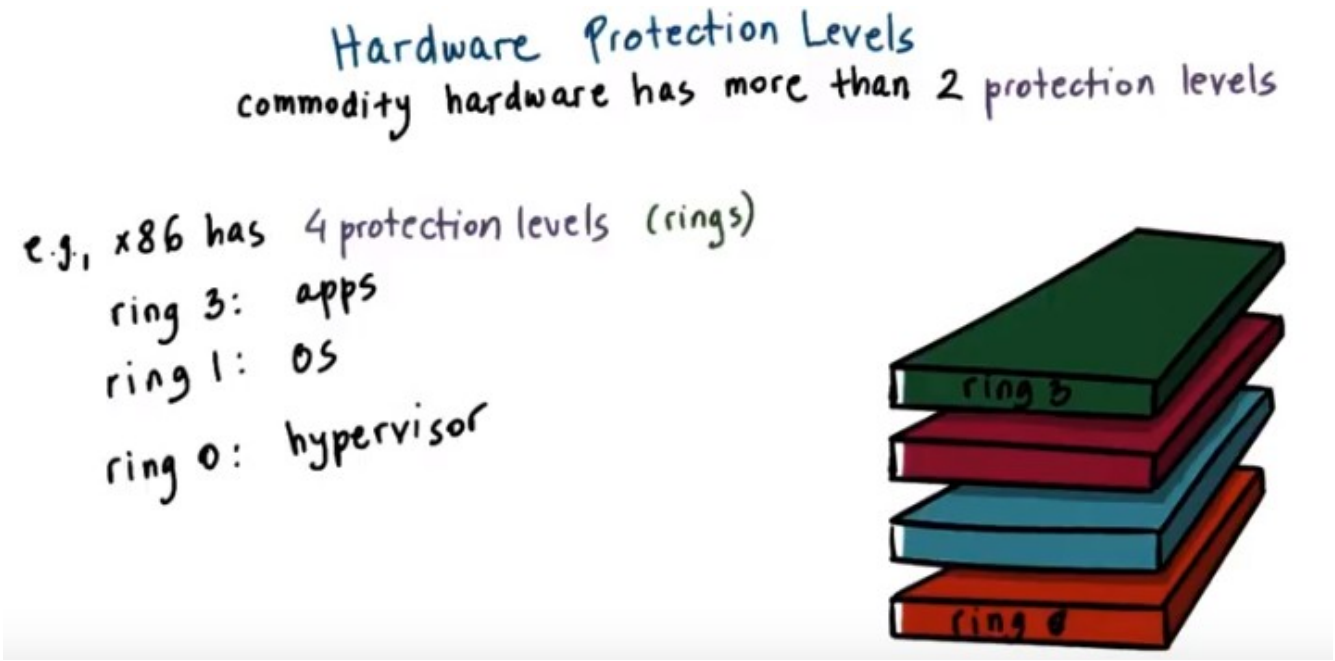
15. So before we go any further, let me ask you one question based on what we've learned so far. Which of the following do you think are virtualization requirements? Here are some possible choices. Present the virtual platform interface to the guest VMs. Provide isolation across the guest VMs. Protect the guest operating system from the applications that are running in the VM. Protect the hypervisor or the virtual machine monitor from the guest operating system. Among these choices, you should check all of the ones that apply.



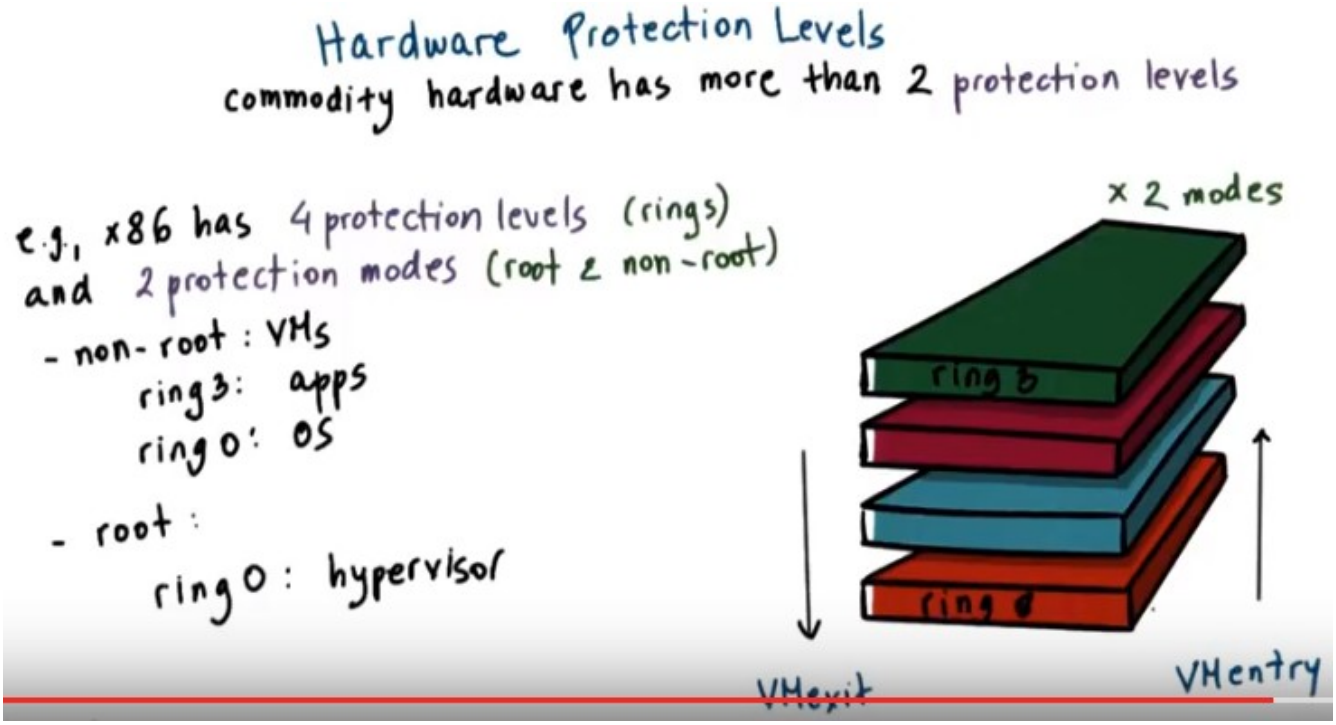
16. If you've marked all of these as correct answers, then you're correct, and I'll explain why. First, at the lowest level, we said that the virtual machine monitor must provide guest VMs with a virtual platform interface to all of the hardware resources, the CPU, the memory, the devices. So this clearly is a requirement. Obviously, the virtual machine monitor will have to isolate guest VMs from one another. And this actually can be pretty easily achieved using the similar kinds of mechanisms that are used by operating systems to provide isolation across the guest VMs. So, the hypervisor will use techniques or the virtual machine monitor will use techniques like preemption, will take advantage of hardware support in the memory management unit so that it can perform validations and translations of memory references pretty quickly. So, there are opportunities to achieve this requirement efficiently using the existing methods and the existing hardware support. Also, within the virtual machine at the topmost level of the stack, the virtualization solution must continue to provide the ability to protect the guest operating system from faulty or malicious applications. We don't want a single application, when it crashes, to take the entire guest OS down. What this means is that somehow we have to have separate protection levels for the applications and for the guest OS. So these expectations that exist when the guest OS is executing natively on the physical platform, they must continue to be valid in the virtualized environment. At the same time, the virtualization solution has to have mechanisms to protect the virtual machine monitor from the guest operating systems. We don't want a single faulty or malicious guest OS to bring down the hypervisor in the entire machine. What this means is that we cannot have a solution in which the guest operating system and the virtual machine monitor run at the same protection level. They have to be separated.



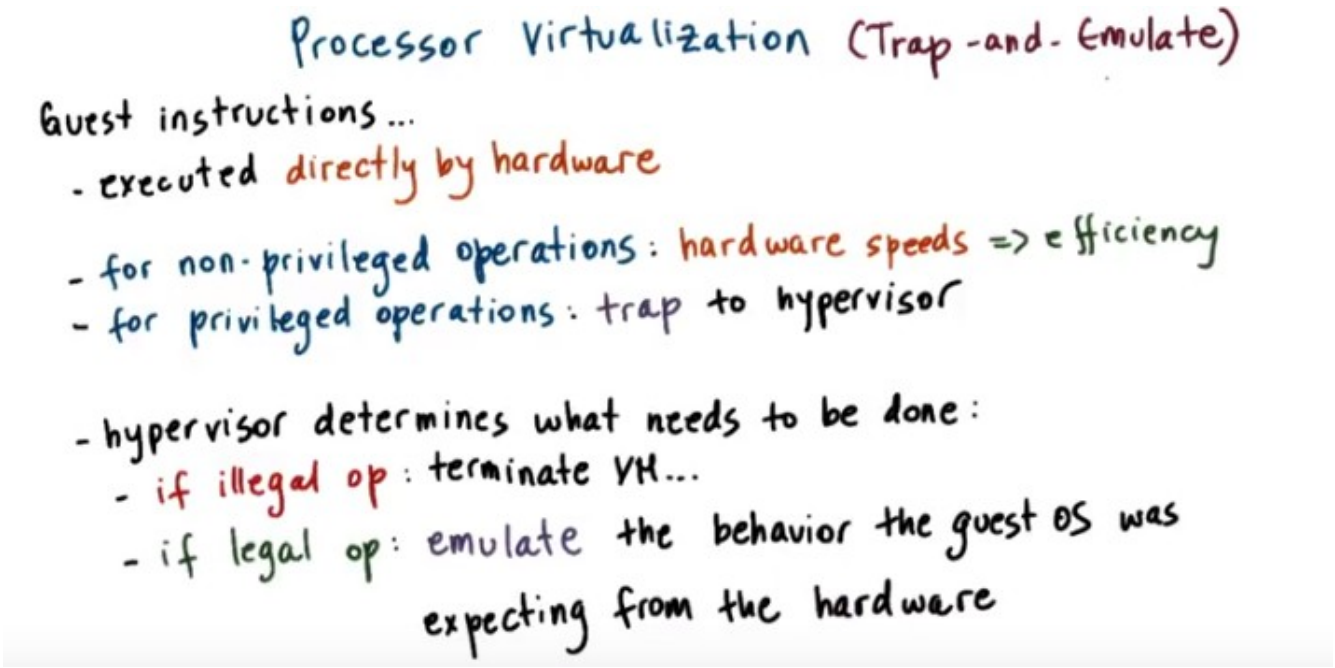
17. When thinking about how to address the virtualization requirements that we just mentioned in the previous quiz, it is fortunate to observe that commodity hardware actually has more than 2 protection levels. Looking at the architecture that's, at least in the server space, most dominant, the x86 architecture, there are four protection levels called rings. Ring 0 has the highest privilege and can access all of the resources and execute all hardware-supported instructions. And this is where in a, a native model, the operating system would reside. So, when the OS is in control of all the hardware resources, it sits in ring 0. In contrast, ring 3 has the least level of privilege, so this is where the applications would reside. And whenever the applications try to perform something, some operation for which they don't have the appropriate privileges, then a trap would be caused, and control would be switched to the ring 0, to the lowest privileged level.



One way in which these protection levels can be used is to put the hypervisor now in ring 0, so that's the one that has full control over the hardware, to leave the applications to execute at ring 3 level, and then the operating system would execute at ring 1 level. We'll explain how this actually works in the following video.

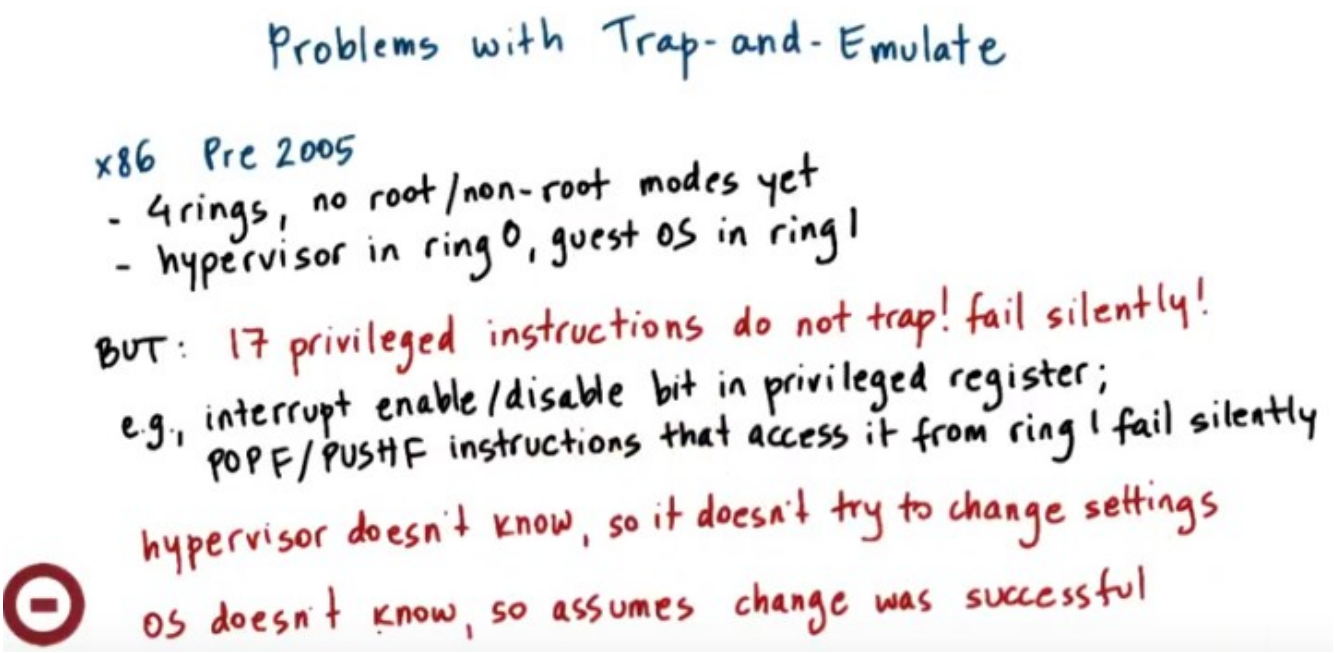


More recent x86 architectures also introduce two different protection modes called root and non-root. Within each of these modes, the four protection levels exist, so there are like two times these protection rings. Now, when running in root mode, all of the operations are permitted, all hardware is accessible, all instructions can be executed. So, this is the highest privilege level. And this is, the ring 0 of the root mode, is where we would run the hypervisor. In contrast, in non-root mode, certain types of operations are not permitted. So then, the guest VMs would execute in this non-root mode. And they would have, as they did in the native execution, their applications running in ring 3 and the operating system running at ring 0 privilege level. Attempts by the guest operating system to perform privileged operations cause traps that are called VMexit. And these trigger a switch to this root mode and pass control to the hypervisor. When the hypervisor completes its operation, it passes control back to the virtual machine by performing a VMentry which switches the mode into non-root mode, to ring 0, so that the execution continues



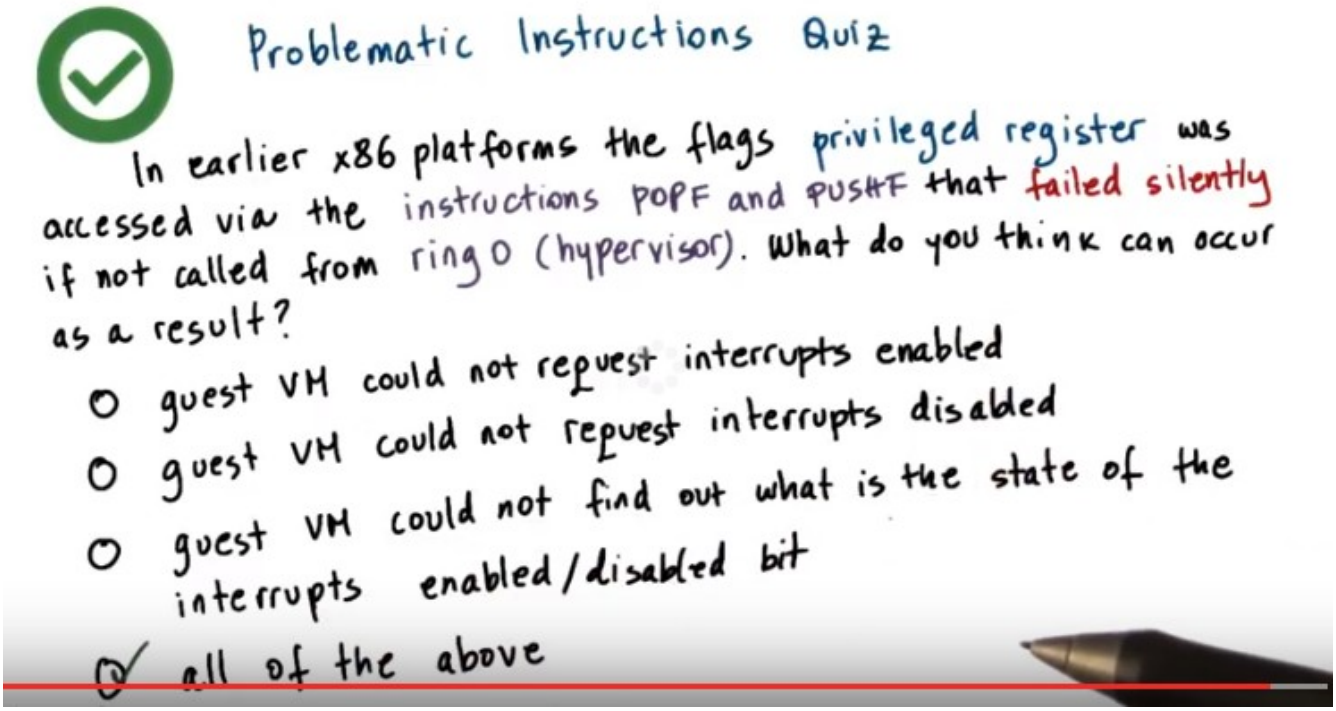
emulate: 模仿

18. Now that we understand how hardware-supported protection levels can be used, we can start explaining how virtualization techniques can be developed that achieved our goal to efficiently at near native speeds allow execution of virtual machines on top of these basically identical virtual platforms. First, guest instructions are executed directly by the hardware. That's important thing to know. The virtual machine monitor does not interfere with every single instruction that is issued by the guest operating system, or its applications for that matter. What this means is that just like the OS doesn't interfere on every single instruction and memory access, here the hypervisor does not interpose itself on every single operation and every single memory access performed by the guest. As long as the guest operating system and its applications operate within the resources that were allocated to them by the hypervisor, then everything is safe. The instructions in those cases will operate at hardware speeds and this will lead to efficiency of the virtualization solution. Whenever a privileged instruction gets accessed, then the processor causes a trap, and control is automatically switched to the most privileged level, so to the hypervisor. At this point the hypervisor can determine whether the operation is to be an allowed or not. If the operation is an illegal operation and it shouldn't be allowed, then the hypervisor can perform some action like to terminate the VM. If the operation is, should be allowed, however, it's a legal operation, in that case, the hypervisor should perform the necessary emulation so that the guest operating system is under the impression that it actually does have control over the hardware. So from the guest perspective, it should seem as if the hardware did exactly what it was expected to do given the instruction. In reality, however, it was the hypervisor that intervened, that potentially executed slightly different set of operations in order to achieve that emulation. This trap-and-emulate mechanism is a key method in which virtualization solutions rely in order to achieve efficient CPU virtualization.

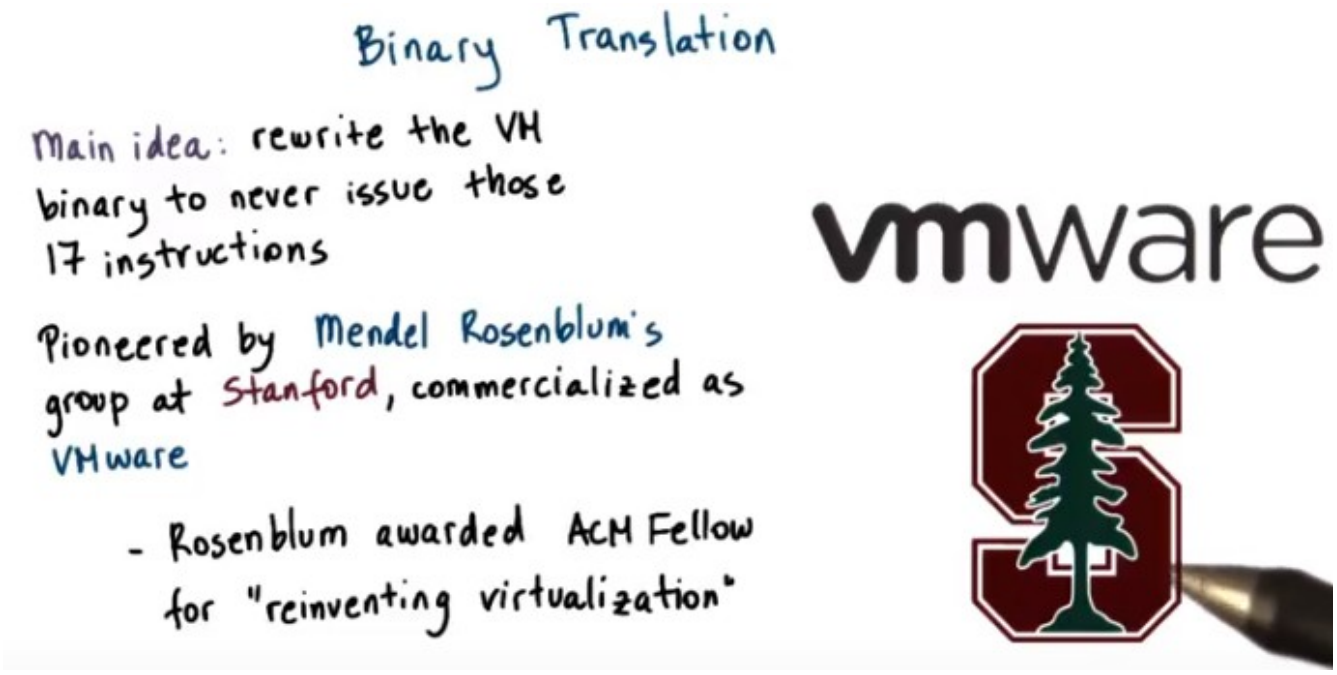


19. Although the trap-and-emulate model seems like it will solve all problems and it worked beautifully on the mainframes, when in the 90s, the need to reapply virtualization solutions to the prevalent x86 architecture came up, it turned out that there were certain problems with this model. At the time, x86 platforms had just the 4 rings. There wasn't any support for root or non-root mode. And so the way to virtualize them would be to run the hypervisor in ring 0 and the guest OS in ring 1. However, it turned out that there were exactly 17 instructions, that were privileged in that hardware would not allow them to be executed if they're not issued from the most privileged ring 0. However, they did not cause a trap. Issuing them from another protection level from ring 1 or above wouldn't pass control to the hypervisor and instead would just fail silently. For instance enabling or disabling interrupts requires manipulating of a bit in a privileged flags register and this can be done via the POPF/PUSHF instructions. However, when these instructions are issued from ring1 in the Pre 2005 architecture, they just fail and the instructions pipeline is allowed to continue to the next instruction. The problem with the situation that there is no trap is that since control isn't passed to the hypervisor, the hypervisor has no idea that the OS wanted to change the interrupt status. So, the hypervisor will not do anything to change these settings, will not emulate the behavior that was required, that was intended with this instruction. At the same time because the failure of the instruction was silent, the operating system, the guest OS, doesn't know that anything wrong happened, so the OS will continue its execution, assuming that correctly the interrupts were enabled or disabled as intend. So the OS will then go ahead and perform operations that, for instance, if interrupted can leave it in corrupt or in deadlock state. Which was intended to be avoided by some manipulation of this flags register to disable interrupt. So clearly this is a major problem and makes this trap-and-emulate approach not applicable for these architectures.

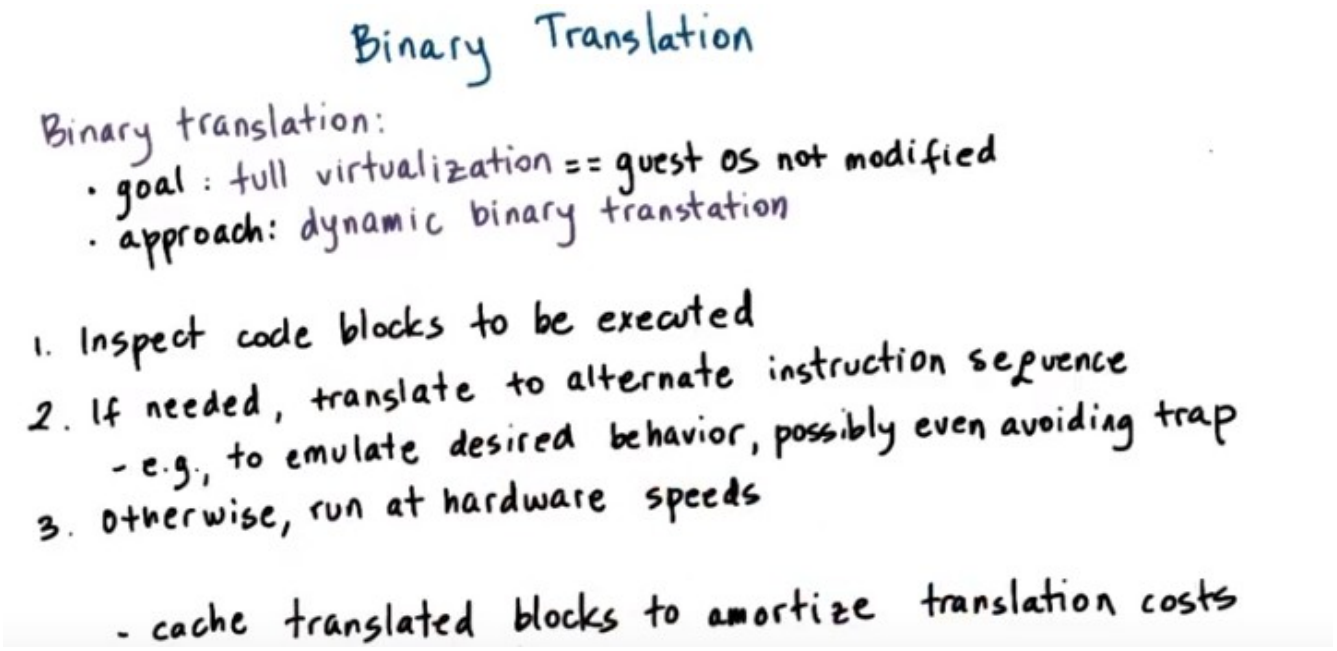
20. Let's explain the problem with some of these problematic instructions a little more via quiz. So we said that in the earlier x86 architectures, the CPU flags privileged register was accessed via instructions POPF and PUSHF. And these instructions failed silently if they're not accessed from the most privileged ring, ring zero. This is where the hypervisor would reside. What do you think can occur as a result of this situation? The options are, the guest VM could not request interrupts to be enabled, the guest VM could not request interrupts to be disabled, the guest VM could not find out what is the state of the interrupts enabled/disabled bit, or all of the above.



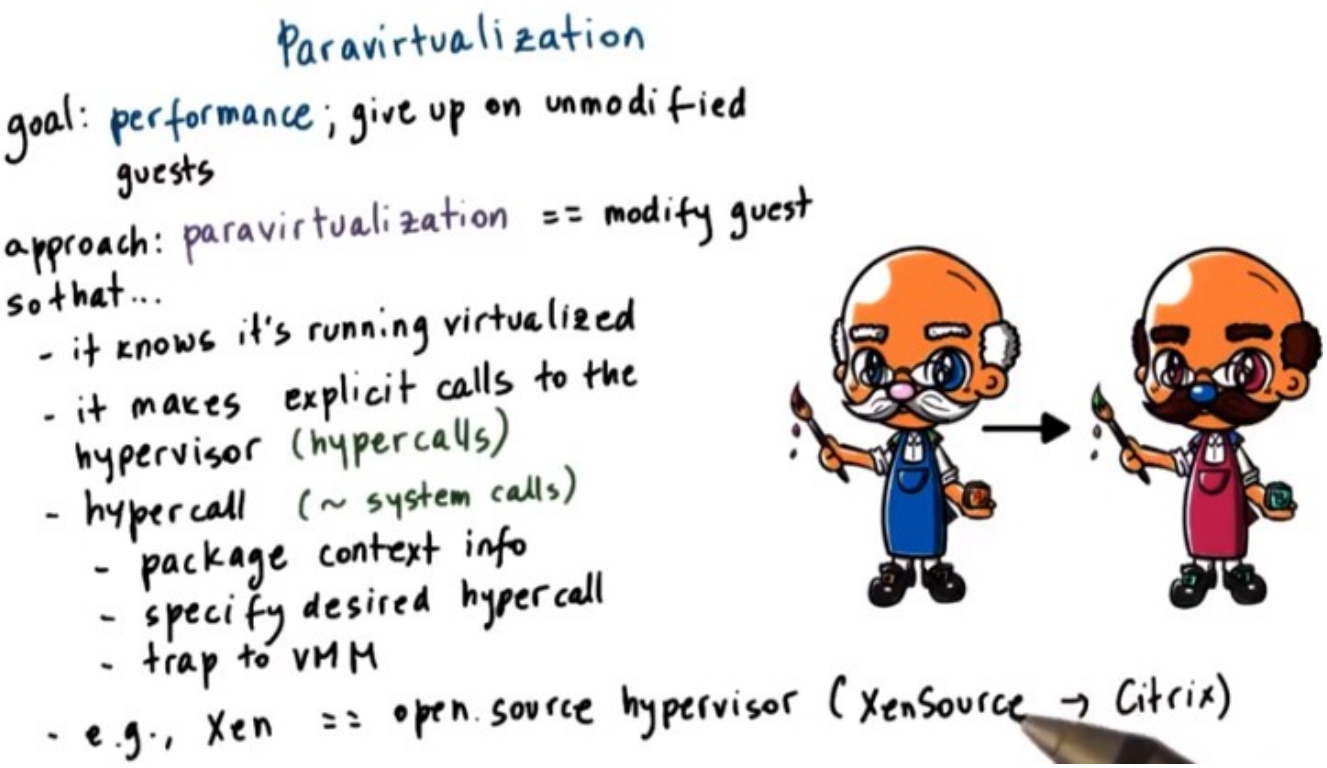
21. The correct answer is all of the above. To perform any of these operations requires access to this privileged register and requires execution of these instructions. When these fail silently, the guest will assume that the request completed, and may end up interpreting some other information that's on the stack incorrectly, as if that's information that's provided by that register. So none of these will be successful.



22. <u>One approach that was taken to solve the problem with the 17 instructions, was to rewrite the binary of the guest VM so that it never really issues any one of these operations. This process is called binary translation.</u> This approach was pioneered by research at Stanford University, by a group led by Professor Mendel Rosenblum. And subsequently, this was commercialized as VMware. Now some 15 plus years and 30, $40 billion later, VMware still owns by far the largest share of the virtualized cores in the server market. Rosenblum later received the ACM Fellow reward, and in the recognition he was specifically credited for reinventing virtualization. He served as VMware's chief scientist for about ten years and now is back full time at Stanford.

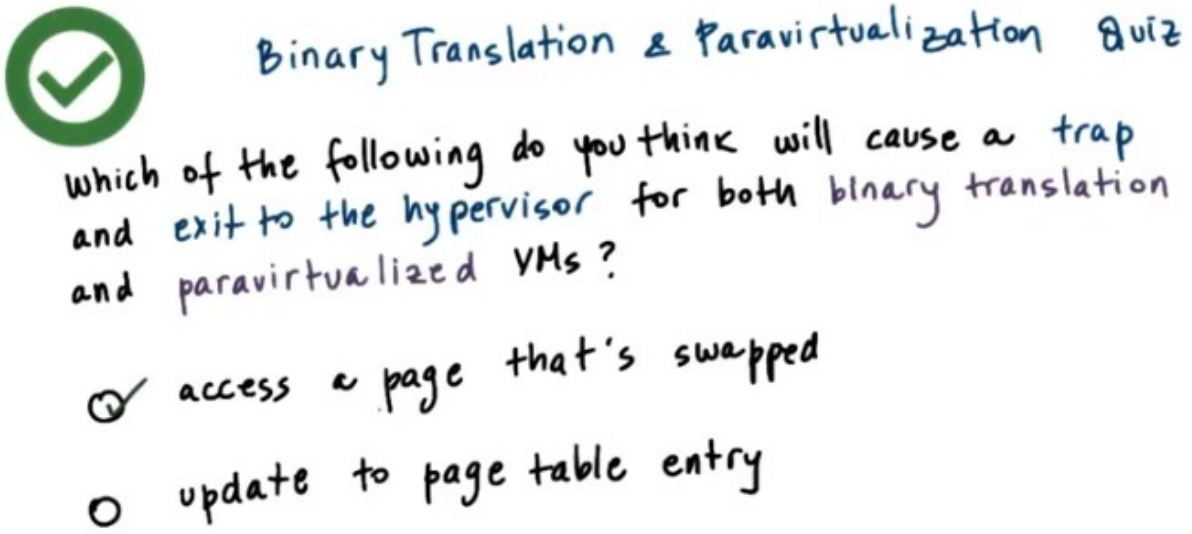


Let me give you now a brief description of what binary translation actually is. A key thing to note is that the goal that's pursued by VMware is to run unmodified guest operating systems. Meaning that we don't need to install any special drivers, or policies or otherwise to change the guest OS in order to run in a virtualized environment. As a startup, they clearly couldn't tell Microsoft to modify Windows so that VMware can improve it's, it's success rate. So, this type of virtualization where the guest OS is not modified is called full virtualization (後面會多次提到 full virtualization). The basic approach consists of the following. Instruction sequences that are about to be executed are dynamically captured from the VM binary, and this is typically done at some meaningful granularity like a basic block such as a loop or a function. Now the reason that this is done dynamically versus statically, so up front before any code is actually run, is because the exact execution sequence may depend on the parameters that are available at runtime. So it's input dependent. So you cannot really do all of this in an efficient way statically up front. Or in some cases you just cannot do it all because you don't have the input parameters. So then you dynamically capture these code blocks and then inspect them to see whether any of these 17 infamous instructions is about to be issued. If it turns out that the code block doesn't have any of these bad instructions, it's marked as safe and allowed to execute natively at hardware speeds. However, if one of the bad instructions is found in the code block, then that particular instruction is translated into some other instruction sequence that avoids the undesired instruction and in some way, emulates the desired behavior. This can possibly be achieved, even by bypassing a trap to the hypervisor. Certainly, binary translation adds overheads, and the number of mechanisms are incorporated specifically in the viewer solutions, in order to improve the efficiency of the process. These things include mechanisms such as, caching code fragments that correspond to the translated basic blocks. So that the translation process can be avoided in the future. Also, the steps like distinguishing which portions of the binary should be analyzed. For instance, distinguishing between the kernel and the application code and making sure that the kernel code is the one that's analyzed and various other optimizations.

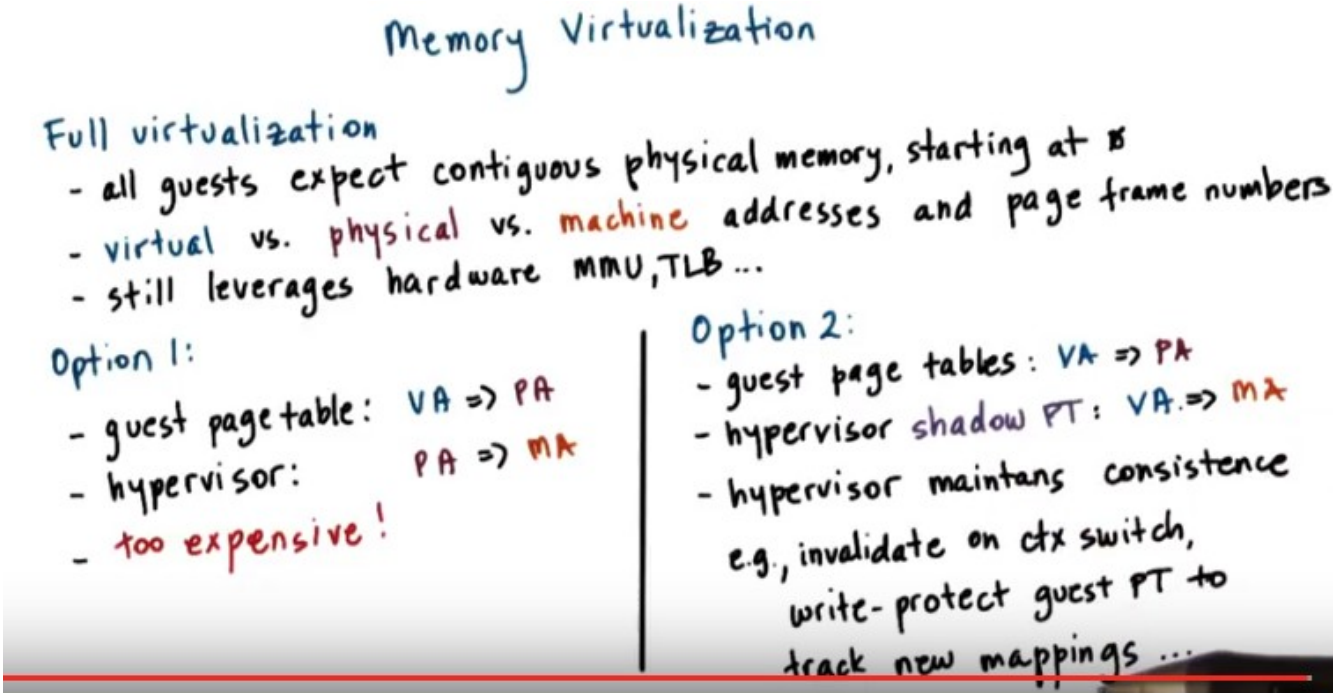


23. A very different approach is to give up on the goal of running unmodified operating systems. Instead, the primary goal is to offer a virtualization solution that offers performance and avoids some of the overheads that may be associated with any of the complexities that are necessary to support unmodified guests. In contrast to full virtualization, this is called paravirtualization. With paravirtualization, the guest OS is modified so that it now knows that it's running in a virtualized environment on top of a hypervisor as opposed to on top of native physical resources. A paravirtualized guest OS will not necessarily try to directly perform operations, which it knows that they will fail. And instead, it will make explicit calls to the hypervisor to request the desired behavior. Or specifically, the desired hardware manipulations. These calls to the hypervisor are called hypercalls. And they behave in a way that's similar to the way system calls behave in an operating system. So the unprivileged guest OS here that's modified will package all relevant information about its context, its current state. And it will specify the desired hypercall. And at that point, it will issue the hypercall and that will trap to the virtual machine monitor. When the hypervisor completes the operation, control will be pass back to the virtual machine, to the guest, and any data, if appropriate, will be made available to it. This approach of paravirtualization was originally adapted and popularized by the Xen hypervisor. This was a popular virtualization solution and originally was an open source hypervisor that started as a research project at University of Cambridge in the UK. This was later commercialized as XenSource and XenSource is now owned by Citrix. But there still remains a lot of activity in the open source Xen project, including at our own research group here. One thing to note, however, is that the open source Xen version as, and the Citrix Xen version have diverged perhaps substantially over time

24. Let me ask a question now, which of the following do you think will cause a trap and will exit the VM and pass control to the hypervisor for both binary translation and for paravirtualized VM's? The options are, access to a page that's been swapped, or update to a page table entry.



25. The first option is correct. If a page is not present, it will be the hardware MMU (下段提到了MMU, 即 memory management unit) that will fall, and it will pass control to the hypervisor regardless of the virtualization approach. For the second option, update to a page table entry, this is not always true. It really depends on whether the OS has write permissions for the page tables that it uses or not. We'll see in the next videos how this can be handle.



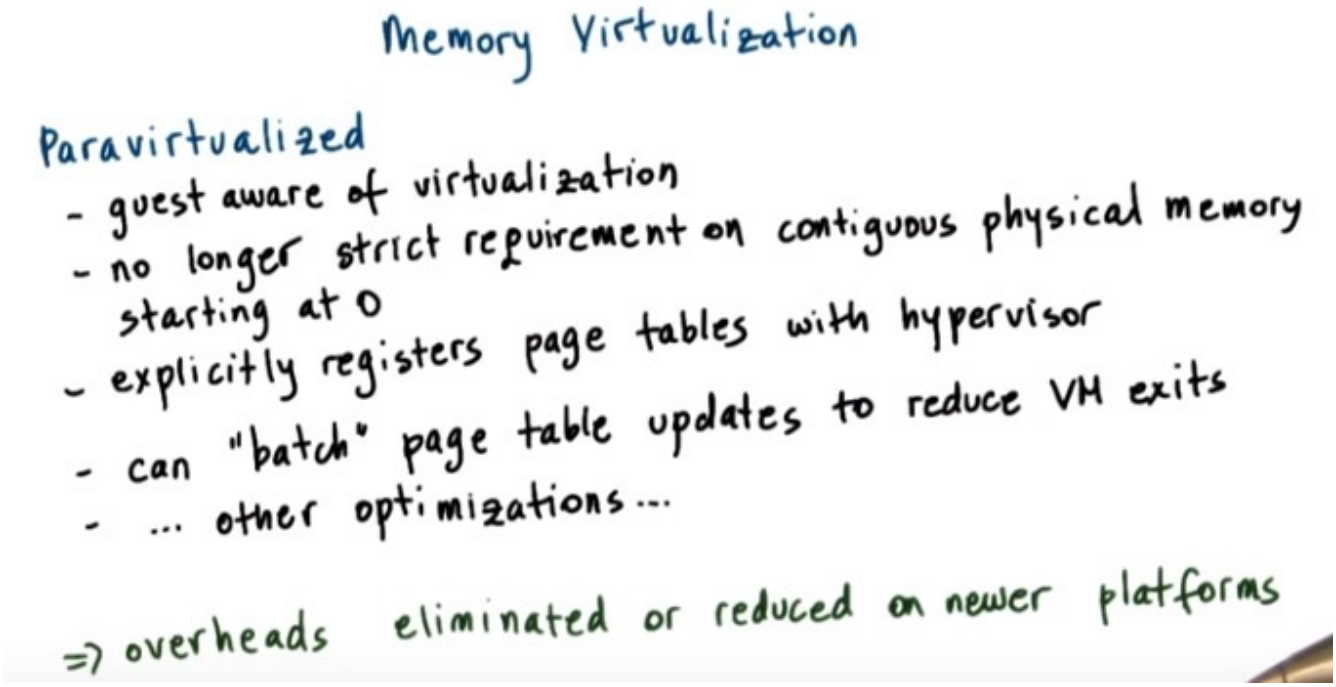
Shadow PT 即 shadow page table

virtual address: application 看到的
physical address: guest OS 看到的
machine address: 實際 memory 中的 address, 即之前的 physical address

VA => PA 即將 virtual address map 到 physical address

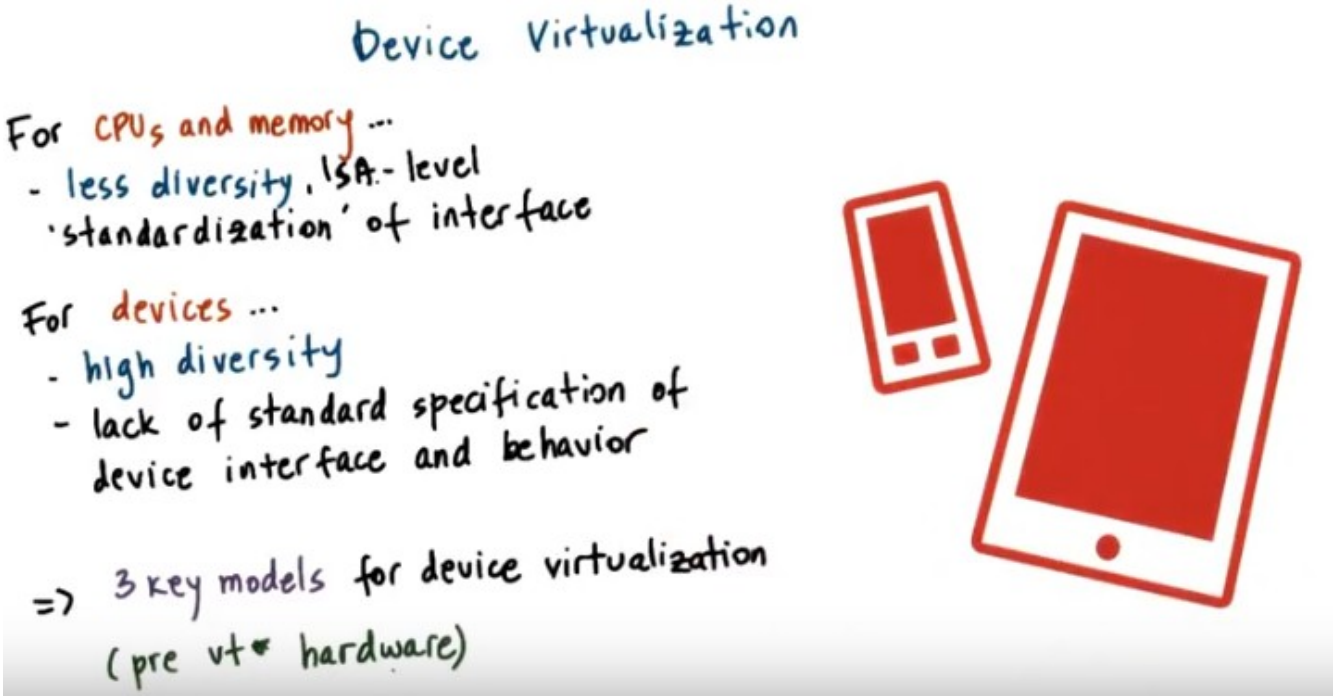
26. So far we've focused on explaining the basics of how to virtualize efficiently the CPU, but let's now look at the other types of resources looking at memory first. We will explain how memory virtualization can be achieved for the two basic virtualization approaches, whether it's based on full virtualization, or requires guest modification and we will talk about full virtualization first. For full virtualization a key requirement is that the guest operating system continues to observe a contiguous linear physical address space that starts from physical address zero, this is what an operating system will see if it actually own the physical memory and run natively on physical hardware. To achieve this we distinguish among three types of addresses, **virtual addresses**, so these are the ones that are used by the applications in the guest. **Physical addresses**, these are the ones that the guest thinks are the addresses of the physical resource and the **machine addresses**, these are the actual machine addresses with the actual physical addresses on the underlying platform. The similar distinction of virtual verses physical verses machine will also apply to the page numbers and the page frame numbers. So given this, the guest operating system can continue make mappings of virtual addresses to of the physical addresses that it thinks it owns, and then underneath that the hypervisor will then pick these physical addresses that the guests believes are the real ones and map them to the real machine addresses. So in a sense they're two page tables, one that's maintained by the guest operating system, and another one that's maintained by the hypervisor. Now remember that at the hardware level, we have a number of mechanisms, the memory management unit (MMU), the TLB caching of the address translations, that these mechanisms help with the address translation process, make it much more efficient, and don't

require us, in software, to repeatedly perform address translations and validations. Now this option that we discussed so far will require that every single memory access goes through two separate translation, the first one which will be done in software, and then the second one potentially can take advantage of hardware resources like TLB because the hardware will understand only this page table. Clearly this will be too expensive since this will add overheads on every single memory reference, it will slow down the ability to run at near native hardware speeds. The second option is for the hypervisor to maintain a shadow page table, in which it actually looks at what are the virtual addresses that the guests has mapped to these physical addresses. And then in the shadow page table it directly establishes a mapping between the **virtual** addresses that are used by the guest, and the **machine** addresses that are used by the hypervisor by the physical hardware. Then if the hardware MMU uses this page table, the guest operating system is allowed to execute natively using the applications will use virtual addresses, and these will be directly translated to the machine addresses that are, used by the physical hardware. The hypervisor will clearly have to be responsible to maintain consistence between these two page tables and it will have to employ mechanism that for instance invalidate, what is the currently valid page tables, shadow page table whenever there is a, context switch or to write protect the guest page table, in order to keep track of new mappings that the guest operating system install since similar mechanism. This write protection is necessary so that whenever the guest OS tries to install new **virtual** to **physical** address mapping in the page tables that are used by the guest, this will cause a trap to the hypervisor, and then the hypervisor will be able to pick up that **virtual** address and then associate the corresponding **machine** address and insert this mapping into the page table that is used by the hardware MMU. This can be done completely transparently to the guest operating system.



27. In contrast, in paravirtualized systems, the operating system knows that it's executing in a virtualized environment. Because of this, there is no longer a strict requirement for the guest OS to use contiguous physical memory that starts at zero. And the guest OS can explicitly register the page tables that it uses with the hypervisor so there is no need for maintaining dual page tables, one of the guest and then another shadow one at the hypervisor level. Now, the guest still doesn't have write permissions to this page table that's now used by the hardware because otherwise the guest potentially can establish any mapping and corrupt other VMs that are running on the same system. So, because of

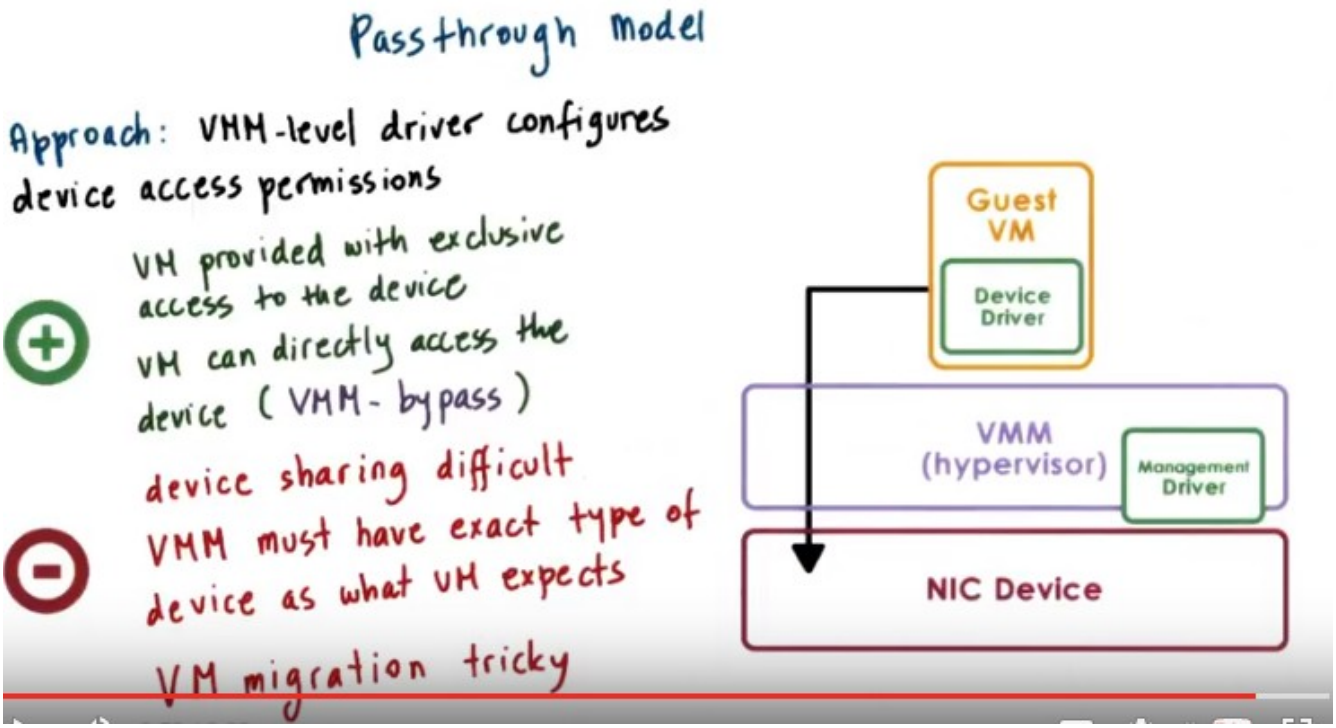
that, every update to the page table would cause a trap and pass control to the hypervisor. But because the guest is paravirtualized, and we can modify the guest and do tricks like batch a number of page table updates and then issue a single hypercall in this case to tell the hypervisor to install all of these mappings. So this can amortize the cost of the exit across multiple operations. There can be other optimizations that are useful. For instance, optimizations related to how the memory's managed so that it's more friendly to execution in a virtualized environment or so that it's more cooperative with respect to other VMs in the system and other things. One thing to note that the two mechanisms that I described with respect to memory virtualization for both full as well as paravirtualized VMs have substantially been improved given advances in the new hardware architectures. So, some of these overheads have completely been eliminated or at least substantially reduced if we take a look at what's happening at the newer generation of x86 platforms. And we will talk about that shortly in this lesson.



ISA: instruction set architecture

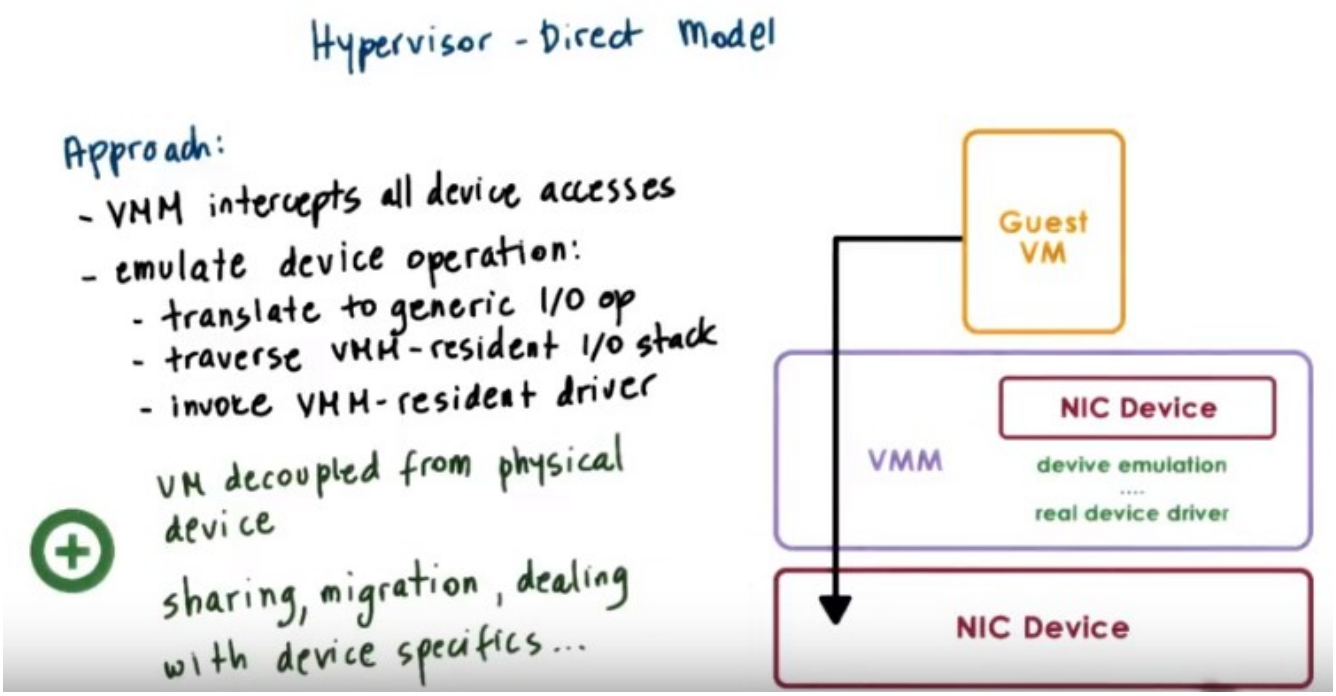
28. When we talk about Virtualization. When we look at CPUs in memory. Certain things are relatively less complicated, in spite of everything that we said, so far. Because there is a significant level of standardization at the instruction set architecture (ISA), across different platforms. So then from a Virtualization standpoint, we know that we have to support a specific ISA. And then we don't care if there are lower level differences between the hardware because it's up to the hardware manufacturers to be standardized at the ISA level. This clearly is the case for a particular ISA for instance, for x86 things will be different across x86 and for instance MIPS platforms. When we look at devices however, there is a much greater diversity in the types of devices, if we compare that to the types of CPU instruction sets. And also there is lack of standardization when it comes to the specifics of the device interface and the semantics of that interface: So, what is the behaviour when a particular call is invoked, how a particular device should respond to a call to send the packet, for instance. To deal with this diversity, Virtualization solutions adopt one of three key models, to virtualize devices. We will describe these models, next. And note that they were developed before any Virtualization friendly hardware extensions were made to the, CP architectures and the chip sets. And these new modifications of the hardware make some aspects of device Virtualization much simpler than than what

originally it was when these models were first introduced.

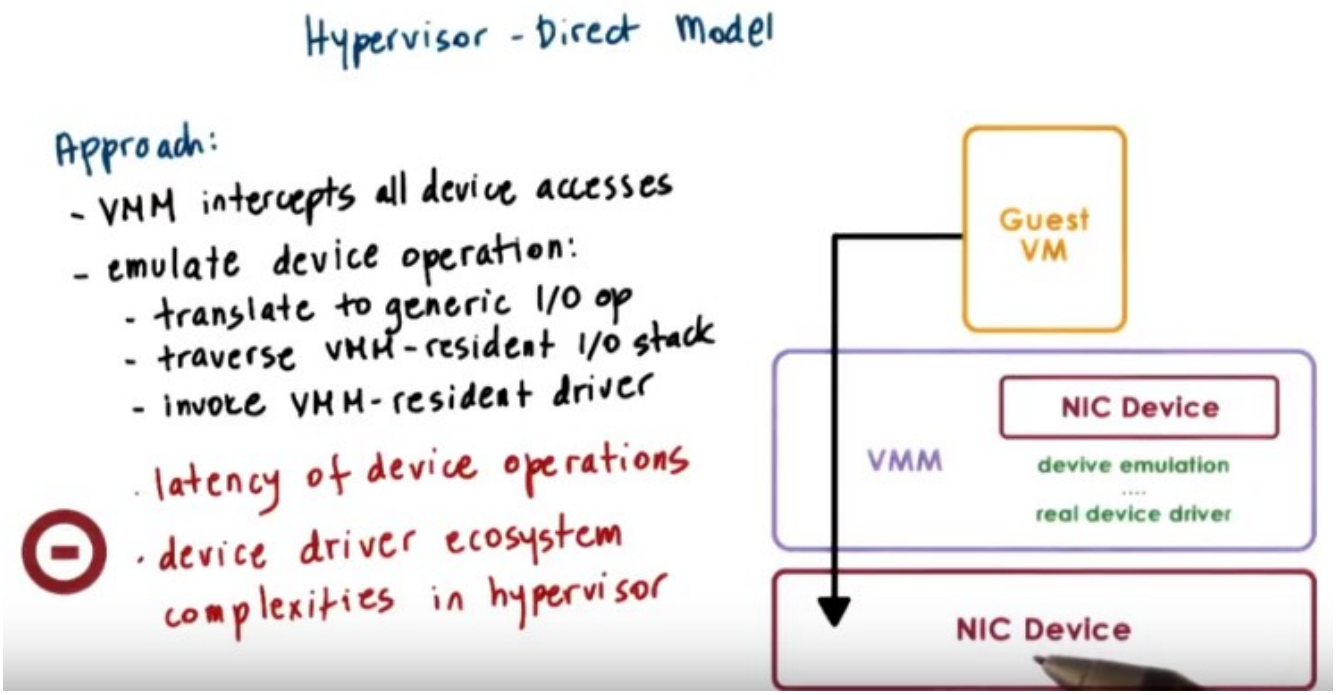


29. One model is the so-called Passthrough Model. The way the pass through model works is that the VMM level driver is responsible for configuring the access permissions for device. For instance, it will allow a Guest VM to have access to the memory where the control registers for the device are. (configure 完了後, VMM 就不管了) There are clearly benefits to this approach, one is for instance, that the guest VM (不是 VMM) has exclusive access to a particular device. It's the only one that can manipulate its state, it can control it and it's the only one that will use it. Also, the VM's accesses to the device completely bypass the hypervisor(即 VMM), so there are direct accesses to the device. This model is also called the VMM bypass model. 缺點: Now clearly, once we start providing VMs with exclusive access, figuring out a way to share devices will become difficult. Well, basically have to continuously reassign, which particular VM it can access a particular device over time. But the sharing will not happen simultaneously, concurrently. That in some cases is, is really not doable, because of the limitations of the device. In other cases, it can be done, but it will be very high overhead operations. So, in practice really device sharing with this model is not feasible. Now because the hypervisor is completely out of the way, it means that the Guest VM and the Device Driver that's in the Guest VM, directly operates and controls the particular physical device. So that means that when we're launching this Guest VM, there better be a device of the exact same type as expected by the Guest OS on the physical machine (即 device 要讓 Guest OS 覺得自己就像在 physical machine 上一樣). In some cases, maybe in the server space that's not as critical of a requirement just because there are fewer types of devices that are commonly present there. But in other environments, this is really not a practical constraint. Remember, we're not talking about the fact that there needs to be a network interface, or a disk device or hard disk device. We're talking about the exact same, particular type of network card or hard disk drive that the Guest VM expects depending on the device drivers that it has. Also, we

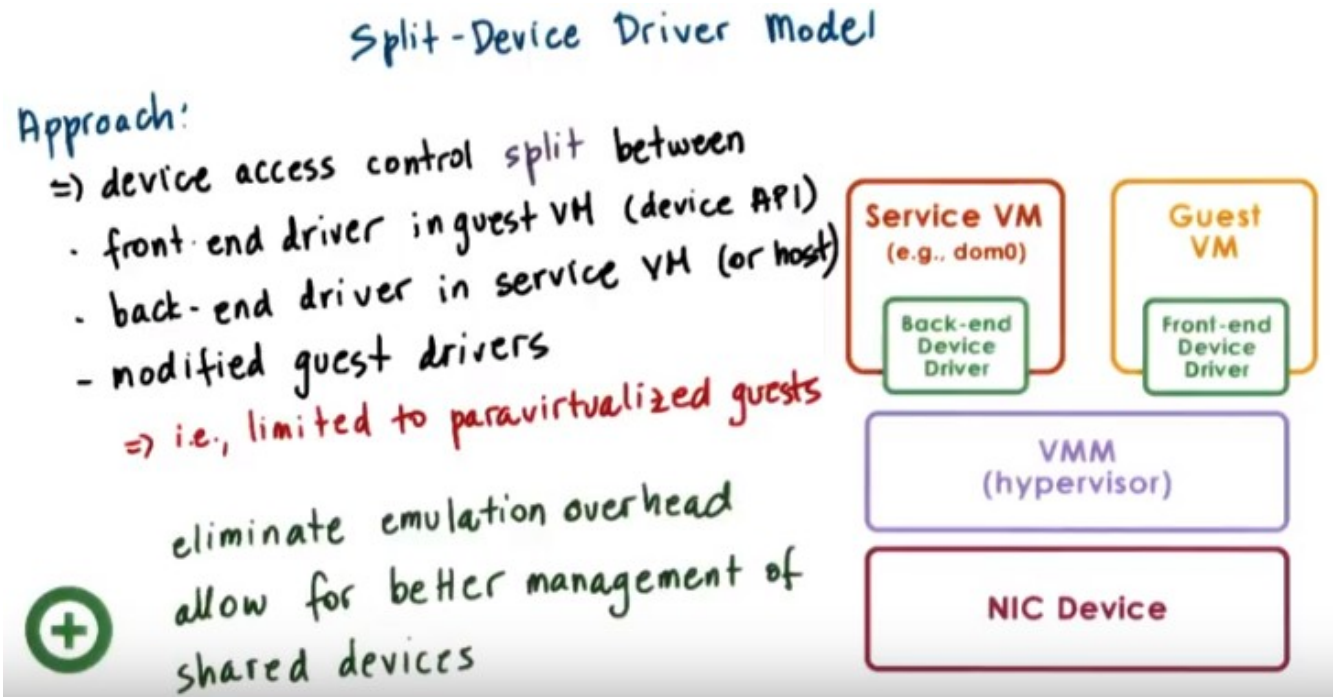
mentioned that one of the benefits of virtualization is that the Guest VM started decouples from the physical hardware. And therefore, we can migrate them easily to other nodes in the system. Well, this pass through really breaks that decoupling, because it directly binds a device to a VM. This makes migration difficult in particular, because there may be some device specific state and 'potentially even device resident state' that would also need to be copied and migrated and then properly configured at the destination mount and then basically that turns VM migration not in a hyperviser and VM specific operation. But it needs to be implemented in a way that knows how to deal with the device specifics of all of the particular devices that are of interest.



30. The second model for providing virtualization of physical devices is to allow the hypervisor to first intercept(截取) every single possible device access that's performed by a guest VM. Once the hypervisor has the information about the device operation that the guest VM wanted to perform, it doesn't have a requirement that the device that the guest VM wants and the physical device match. So the hypervisor would first translate that device access to some generic representation of an I/O operation for that particular family of devices, whether they are networked devices or disk devices. And then it (hypervisor) will traverse the hypervisor resident, the VMM resident I/O stack. So the bottom of that stack is the actual real device driver, and the hypervisor will invoke that device driver and perform the I/O operation on behalf of the guest VM. Clearly, a key benefit of this approach is that now the virtual machine is again decoupled from the physical device. Any translation, any emulation, will be performed by the hypervisor layer. And because of that, operations such as sharing and migration, or the requirements of how we need to deal with device specifics, all of that becomes simpler in a sense. This is a model that's originally adapted by the VMware ESX hypervisor.

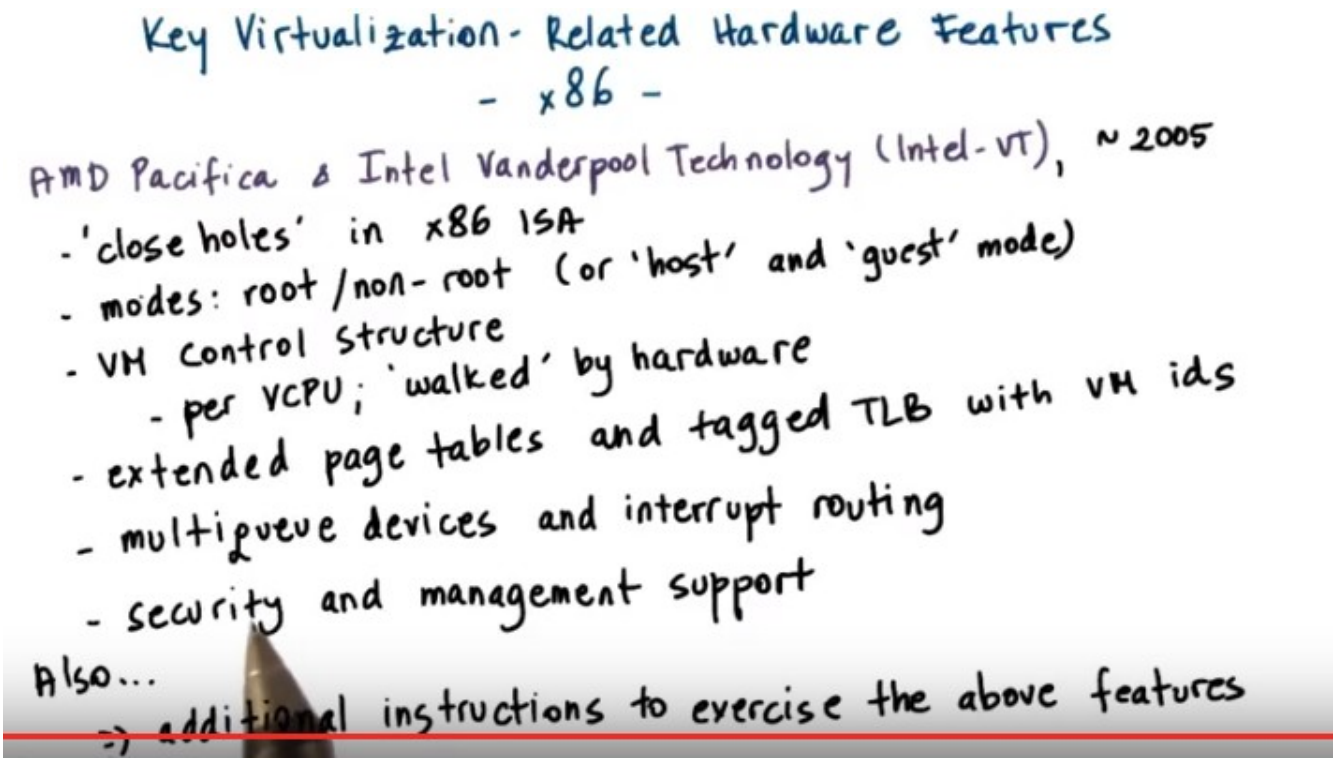


The downside of the model is that it clearly adds latency on device accesses because of this **emulation** step. And then also it requires that the device driver ecosystem is in certain ways integrated with the hypervisor. Now the hypervisor need to support all of the drivers so that it can perform the necessary operations. And the hypervisor is then exposed to all of the complexities of, and bugs of various device drivers. Again, we said earlier in the case of VMware, because of its market share, this model was a reasonable model, and it made sense and it's been sustained because of that.



31. A third model to device virtualization is the so-called Split-Device Driver Model. This model is called split, because all of the device accesses are controlled in a way that involves both a component that resides in the Guest Virtual Machine and also a component that resides in the hypervisor layer. Specifically, devices **accesses** are controlled using a device driver that sits in the Guest VM called the front-end device driver. And then the actual driver for the physical device that is the regular device driver that's used by operating systems when they run natively, in this back-end driver, referred to as back-end resides in the service VM in the case where we have a virtualization model that involves a service VM or the host operating system in the case of type two virtualization. The point is that this back-end should really be just the regular device driver that the service, the Linux OS for instance and the Service VM would just be able to install and use, even if it's not running in a virtualized environment. Now although this back-end driver does not necessarily have to be modified, the front-end driver has to be modified, because it explicitly needs to take the device operations that are made by the applications in the guest and then put them together in something that looks like a, a special message that will directly be passed to this back-end component that's in the service VM. So this approach, essentially applies only to paravirtualized guests that will be able to explicitly install these special front-end device drivers. What these (視頻中指到 Front-end Device Driver 的) really are, they look like wrappers for the actual device API (見圖中左邊). So the applications potentially, they don't have to be modified. They will continue making the same kinds of requests for device operations. But this front-end device driver will treat these operations, especially will not attempt to actually shoot off access to the physical device, instead will create messages that will get passed to the service VM. One benefit of the approach is that it can eliminate the overheads that are associated with device simulation that the previous model required. Now, we don't have to reverse engineer exactly what the Guest OS is trying to do. The Guest OS via its front end device driver explicitly tells the virtualization layer, so overall these two components. What exactly is it that the guest VM requires. Another more subtle benefit of the approach is that now that we have this back-end component that's centralized in that it will re, accept requests from all the Guest VM's in the system and then try to decide which one of those gets to execute the physical device access. There's some opportunities for some better management of the shared devices accesses. So to enforce some finer grained policies regarding fairness, regarding

priorities and those sorts of things.



32. The solutions that we described on how to virtualize memory and I/O clearly indicate that there is some degree of complexity and overhead that have to be incurred by the due to virtualization. Given the wide recognition that virtualization delivers important benefits and it's been pointed out earlier in this lesson in that it presented an important path to address some of the issues related to rising operating costs in the IT industry. The hardware companies responded and they modified their architectures in a way that makes them more appropriate for virtualization. In the x86 world these virtualization friendly architectures started appearing around 2005. Read AMD Pacifica and Intel Vanderpool Technology or LIntel-VT for short.

'close holes' in x86 ISA:
With respect to x86, so one of the first things that was fixed was to close the holes with respect to those 17 non-virtualizable instructions, so that they will cause a trap and pass control over in a privileged mode.

modes: root/non-root (or 'host' and 'guest' mode):
Also, the new protection mode was introduced. So, as opposed to having just one protection mode with four ranks, now there are two protection modes, so root and non-root. Also referred to as host because this is the route is the mode in which the host operating system, the hypervisor would run. And the non-root, that's also referred to as guest which is where the guest VM would run.

VM control structure (per VCPU; 'walked' by hardware):
Also, a support was added for the processor, the hardware processor to understand and to be able to interpret information that describes the state of the virtual processors called VCPUs. This information is captured in a VM Control Structure or also called a VM control block in the AMD x86 architectures.

The fact that the hypervisor understands how to interpret this data, so it can walk this data structure is the term that's commonly used. Means that it can specify whether or not a system call should trap. So, it's easy for the hypervisor to know that a particular type of operation should not cause a trap into root mode and instead should be handled by the privilege layer in the non-root mode, so the privilege layer in the non-root mode is the operating system. Then other pieces of information, then that in a certain way can help reduce the virtualization overheads.

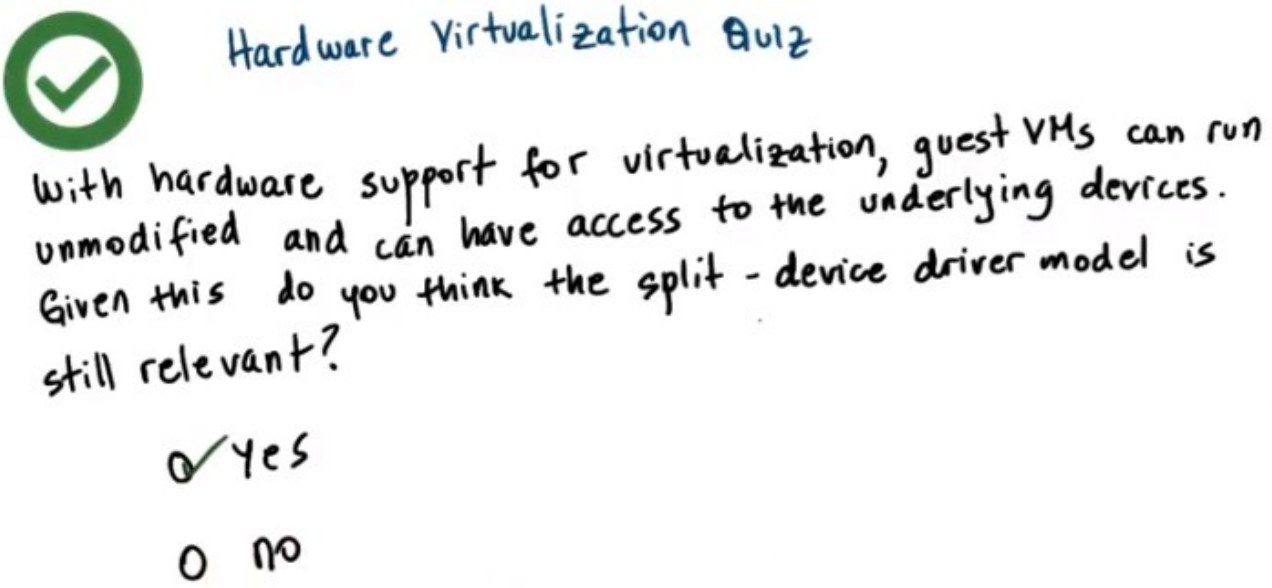
extended page tables and tagged TLB with VM ids:
The next step in terms of virtualization related advances was to make it easier to manage memory. Since hardware was already able to understand the presence of different VMs, the next step here involved tagging the memory structures used by the hypervisor with the corresponding VM identifiers. So this led to support for extended page tables where the page table entries now include information about the VM id and also tagged TLBs. What this means is that if there is a context switch among Vms, that's also called the world switch, when we're switching from one VM to another, we don't have to flush or invalidate those entries that are in the TLB that belong to the previous VM. This is because the MMU, when it performs a check against the TLB entries will try to match both the virtual address that is causing the access request as well as the VM identifier. And if they both match, then it will proceed with the address that's specified in the TLB entry. Otherwise, it will deal with the page fault failures. As a result, context switches are now much more efficient.

multiqueue devices and interrupt routing:
Hardware was also extended to add better support for I/O virtualization and this included modifications both to the processor and the chipset side. And also device and system interconnect capabilities that were introduced in order to support this. Some examples of these features include things like multiqueue capabilities on the device, and you can think of this as the device having multiple logical interfaces where each interface can be used by a separate VM. And also better support of interrupt routing, so that when a device needs to deliver an interrupt to a specific VM, it actually interrupts the core where that VM is executing and not some other CPUs.

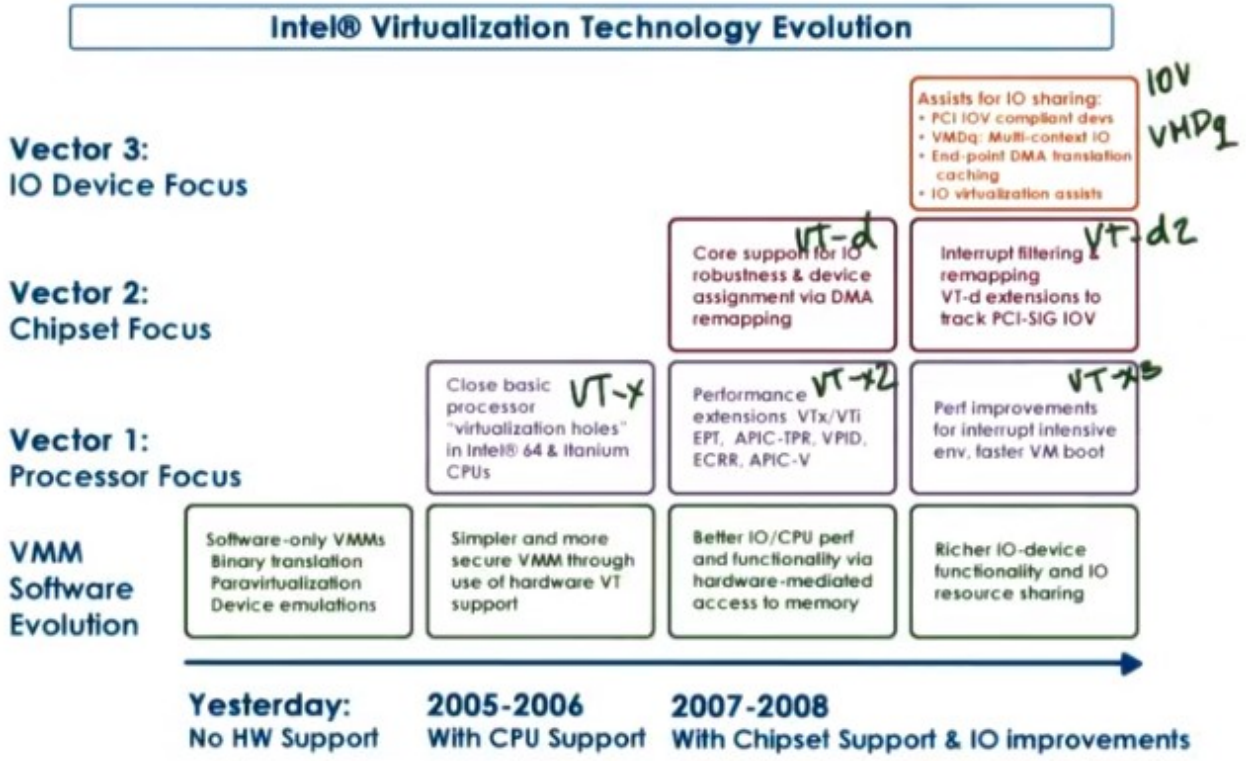
security and management support:
Additional virtualization related hardware features were also included for stronger security guarantees that now can be made to the VMs and also to protect VMs from one another as well as from the hypervisor. And also, features for better management support or for more efficiently to be able to perform various management operations in virtualized environments. You can think of this as more virtualization friendly management interfaces.

additional instructions to exercise the above features:
Also, a number of new instructions were added to x86 in order to actually exercise all of these new features. For instance, a new instruction was introduced to transition from one mode to another. Basically, to transition from root mode or to return control to non-root mode. Or a new instructions to manipulate in certain ways state that's in the per VM control data structure, etc.

33. Let me ask a question now. With hardware support for virtualization, guest VMs can now run unmodified and can have access to the underlying devices. Given this what do you think is the split device driver model still relevant? Answer either yes or no.

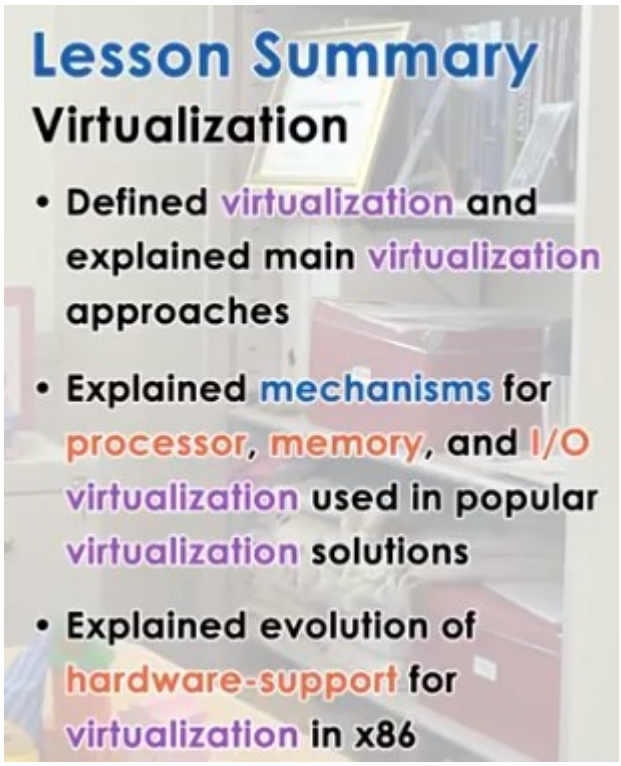


34. The answer to this is yes, because with the split device driver model we are consolidating all of the request for device access to the surface VM, where we can make better decisions and enforce stronger policies in terms of how the device is share. Without perhaps relying to specific support for desired behavior, for desired sharing behavior on the actual physical device.



35. And finally I'd like to show you this illustration from Intel that summarizes the advances in the Intel

VT architecture over the last decade or so. The lowest set of boxes summarize the impact of the virtualization related hardware modifications that have been added along the three different dimensions (即左邊那三個 vector). The three dimensions are as follows, vector 1, refers to any CPU related modifications and this includes things like fixing issues with non-virtualizable instructions or adding support to the CPU for extended page tables. Vector 2, refers to modifications pertaining to the entire chipset, such as chipset site support for technologies like what's called SR-IOV, that technologies that help with IO virtualization, or for IO routing and mechanisms for direct device access, or support rather for direct device access. And then the third vector, the third dimension in which Intel has contributed to the evolution of virtualization friendly technology, is to actually push advances in the device level technology. So this is what's happening on the device as opposed to on the chip you, on the chipset or the CPU side. And with this draws devices are more easily integrated in a virtualized environment. This includes things like support on the device for DMA remapping so that it can appropriately DMA into the correct set of memory depending on the virtual machine that it targets. And to also support the multiple queue, the multiple logical interfaces such that different logical interface can be given to a different VM. The architecture versions that encapsulate these modifications are referred to as VT-x for the modifications along the first vector. With a VT-x2, VT-x3 occurring in subsequent generations of processors. Then VT-d for the advances along the second dimension with VT-d following and so forth, and then the architecture modifications along the third vector are encapsulated into what's referred to as IOVs or IO virtualization technology. And VMDq what stands for virtual machine device queue's, so this is device residence support.



36. In this lesson we explained what is virtualization, and what are some of the main approaches that are used to virtualize physical platforms. We specifically described mechanisms for processor and memory, or device virtualization that either were or still are part of the dominant virtualization solutions, such as Zen and KVM or the VM Ware products. In addition, we discussed some of the hardware advances that have occurred as a result of the need to efficiently virtualize the popular x86

platform.

37. As the final quiz, please tell us what you learned in this lesson. Also, we would love to hear your feedback on how we might improve this lesson in the future.