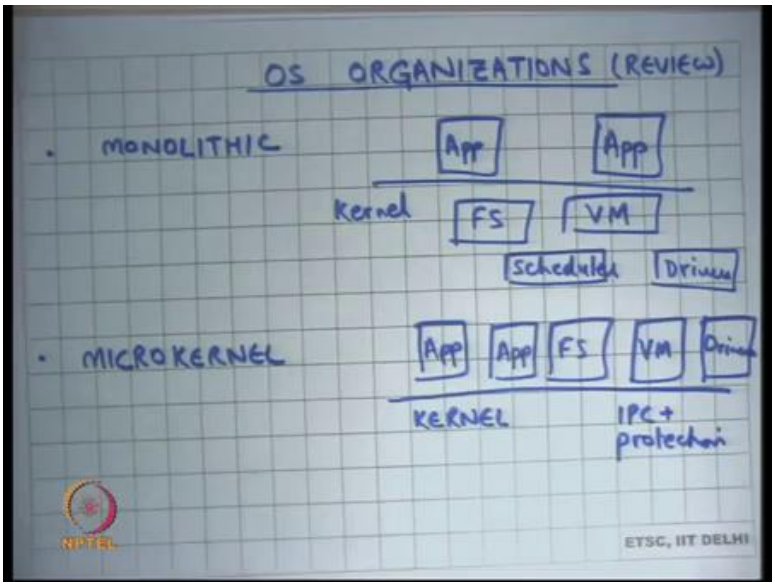
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 40
Virtualization, Cloud Computing, Technology Trends

Welcome to Operating Systems lecture 40 right. So, we were discussing operating system organizations and let us review it.

(Refer Slide Time: 00:33)



So, they are you know there is a monolithic OS organization that we have been used we are used to which is which has you know let us look at the Unix abstractions, the file system, virtual memory, scheduler drivers.

All live in one common address space that is called the kernel and then these different modules exporters system call APIs like read write open close M map, page fault etcetera right. So, these are all sort of attractions that these things provide. Then there was an alternate organization which is called the microkernel, where these different components are moved in separate production domains.

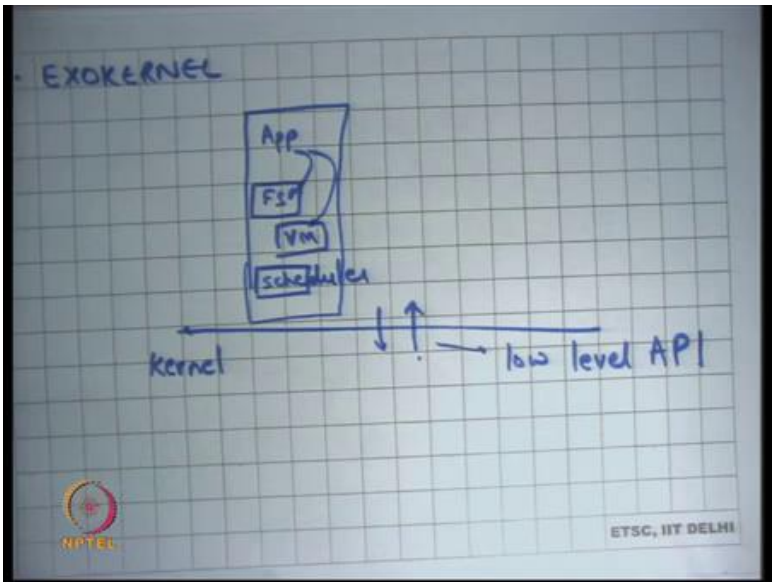
For example, they are run as separate servers; the servers could be running as separate processes in unprivileged mode or they could be running as separate processes in

preparation mode, in either case they are isolated from each other. And, but the kernel is basically it is just an inter process communication IPC and protections.

So, nobody can run away with the resources and it the job of the kernel is really to provide fast inter process communication right. So, this are the advantage that it is very plug and payable pluggable and playable, you can choose your file system, you can choose your virtual memory, subsystem depending on what you want to run, you can actually say here is my virtual memory subsystem why do not you run this one, instead of that one.

Of course, there are some issues there that you know that virtual memory subsystem should be trusted maybe signed by some certificate authority etcetera. But but it is possible to do this right. The other thing is of course, if there is a bug or there is a security flaw in one of these things, let us say there is a security flaw in the driver, it does not affect the security of the full system, it just affects one container and so, faults are not propagated ok.

(Refer Slide Time: 02:25)



And, then we were looking at another type of micro kernel that is called the exokernel. Here the idea is that you expose as much information low level information as possible from the kernel to the application. And, all these different subsystems like the file system virtual memory scheduler, even the drivers are part of the application logic ok.

So, for example, the application can decide what the application knows, that there is a physical address space and there is a virtual address space, and it can decide what should be the footprint of my application or my code on the physical address space, and what should be the footprint in the virtual address space, and what should the mapping be? In other words, it can decide what the replacement algorithm should be when should I take a page fault etcetera and so on right.

Similarly, in the file system case it can choose what file system it likes. So, instead of exposing open rewrite close calls to the application, how about exposing the device interface sort of like the disk interface to the application right. So, tell the application that look you have access to a disk a raw disk and so you can write to sector number 10 and so on.

And, you choose you know what layout you want on this part of the disk. Of course, that has problems you know what about sharing etcetera, but those can be you know built on top of that. So, how do different processes share files etcetera, that can be built on top of that, but at the end of the day the kernel is not involved in implementing the file system. The kernel is only involved in providing an abstraction, that allows the application to build a file system on top of it right.

And, so that makes things very configurable and application can choose what file system it wants, and I was to use the example of a database, you know a database does not like. So, for example, a database may one strong system guarantees. So, if it is written something to a disk block, it needs to be sure that the disk block is actually present on the magnetic platter before it actually you know prints some message or releases the money or something like that right.

So, these kinds of guarantees if you provide on top of a regular operating system like Unix, you will need to do system calls like epsink. And, those because there is multiple layers, then all those layers do not interact very well with each other that overall performance of the system is not optimal, but if you have this kind of a system, then an application can actually tune you know there is no you have gotten rid of the layers, the multiple layers you were saying that the application can choose.

Student: (Refer Time: 05:04) chalk (Refer Time: 05:05).

Chalk.

Student: Yeah.

No, I do not have it I do not think so.

Student: Here is. Here is alright sorry. So, the application can choose what file system abstraction it wants. So, there is only one file system abstraction ok. So, let us look at. So, you may say, but you know that is too much headache for the application developer. Does that mean if I want to implement a small application, I need to implement a full file system and a virtual memory subsystem, before I can get hello world to run, well the answer to that is very simple you just use libraries right.

So, you have standard libraries for example, you have you use you know you have a library that implements the ext3 file system it is a user level library right. That implements the ext3 file system using the obstructions provide by the kernel. Most applications we want to just use the ex 3 user level library, but let us say your application is a database application, it says I do not want to use the ex 3 library, I want to use my own sort of a library.

So, this flexibility is possible, because you are giving lower level interfaces to the application as opposed to higher level interfaces for application, yes.

Student: (Refer Time: 06:23) too much over head like (Refer Time: 06:24) will be having some of the information about.

Would not there will be too much overhead interesting, would not there will be too much overhead why do you think there will be too much overhead, you say why well every App will have it is private copy of the file system code. Every process will have a private copy of the file system data or similarly VM data, driver data.

So, is not that too much overhead. Well you know you could argue there is some overhead yes because, but at the same time you could share these things across process is right. So, processes can actually share address spaces. So, libraries that are common to multiple processes, can actually live in shared space, can actually have one physical address copy. And, multiple virtual addresses spaces can point to the same library copy.

And, these kinds of optimizations are well known, and you already know about them right. So, multiple processes sharing libraries, sharing data are actually pointing to the same physical memory area and so, this space overhead is not much very negligible in fact.

Student: Sir, one more kernel has to keep track of like you said last time for every Apps for bad (Refer Time: 07:34) going running away with the.

Right this is what the kernel has to do is firstly, it has to design a low level API, that allows all these applications to actually control these low level interfaces, like low level devices for example, or low level disk or network or whatever or CPU or memory.

But the other thing it needs to do is basically implement protection. These applications are not trusted, they are not trustworthy and so an application should not be able to for example, take to run away with the resources, or it should not be able to create mappings that are it is not allowed to create, or should not be able to write 2 blocks disk blocks that it is not supposed to write to.

Right. So, those are small things that the kernel will ensure.

So, kernel job is protection and that is poor protection and low-level implementation of low-level API that is all. So, that way the kernel becomes much thinner, than the original thing any way protection was there right. If, you whether you put it a higher level or you put it a lower layer, that will remain in protection or you implement protection at a lower layer.

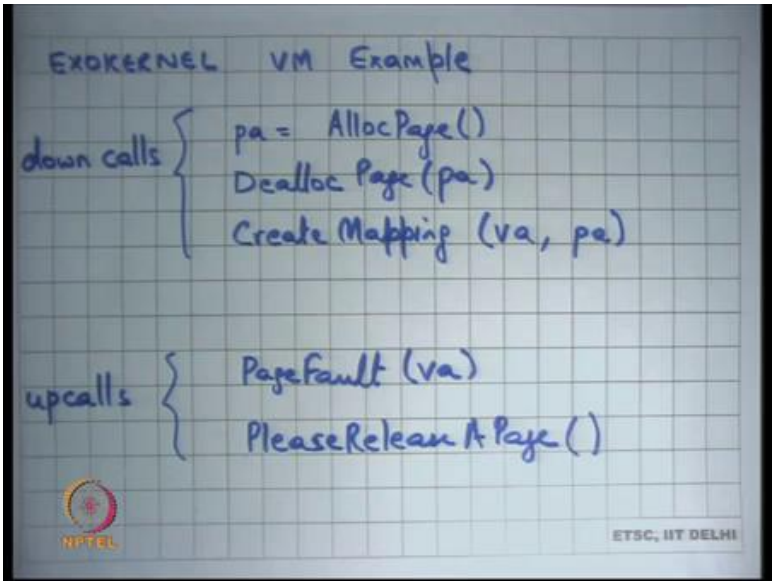
Student: Sir, but in previous case we had like just simpler abstraction that kind of protection for which was same for every App right. In this case we have keep records of which have are (Refer Time: 08:51)

No so, question is in the previous case in the monolithic kernel, the abstractions were identical for different processes. Here the abstractions are not identical for different processes that is not true; the abstractions are still identical for different processes; the abstractions are these low-level APIs. What is happening between the App and the these layers kernel does not care?

Student: So, sir I am saying like kernel has to keep track of whether the app is running for too long (Refer Time: 09:18) did it yield when I requested it to (Refer Time: 09:21)

So, you know kernel has to keep track of how long the App has been running, but that is true even for monolithic right. That is just slightly more complex we are going to discuss it with more examples let us look at this.

(Refer Slide Time: 09:33)



So, we were looking at the exokernels VM example and see said ok. So, because unlike Unix, where the application has no idea that there is something called a physical address space. The application only knows about a virtual address space right. And, it is the operating system that is basically create managing this mapping between the virtual address space and the physical address space. In the exokernel you actually expose this entire mapping to the application, and you can and their interface that you gave is.

So, the that down calls which are you know system calls basically that application can make. So, you can allocate a physical address page and you can deallocate a physical page. So, notice that I am saying pa which basically means physical address. So, the application knows that it is a physical address space. Application has full knowledge about what is the footprint in the physical memory.

What is it is footprint in the physical memory in Unix an application has no idea right, it is OS who is managing how much physical footprint you have and then there is this

down called create mapping run virtual address physical address. Of course, you know there are some production mechanisms and the kernel must ensure, to make sure that it can only create mappings to physical addresses that it owns right, or it has allot right.

So, it is also that there is no security problem of that is all. And, then there are up calls; up calls are similar to signals in Unix, where the kernel can actually ask the application to execute some code, and there is some entry points at the kernel the application must have preregistered and so some up calls are page fault. So, the kernel tells the application that is a page fault on this virtual address and the application can decide that it wants to allot one more physical address physical page and then create this mapping and then resume the execution.

In doing so it may want to first de alloc some other physical address space, because you know the application the kernel is not allowing you to alloc more than certain number of physical address space etcetera. And, then so that is page fault, and the other thing that the kernel will want to make sure ensure is that the application cannot just run away with this memory.

And, so in this case let us say there are other applications that are running, and you want to do some kind of fear allocation of a memory, then the kernel can make an up calls please leave the page. So, notice that this is unlike Unix, where the kernel would just take away the page, it will run some algorithm internally to figure out which page to take let us just take it away right.

Without that without consulting the application at all and that was a problem right. The problem was that I was not consulting the application; the application was perhaps the best judge of which page to actually throw away right and, whether it needs to be written to the swap device or not.

In this case you just tell the application I want you to release the page. And, so, the application then you know chooses which page to release and then calls Dealloc Page on that particular page. And, then you can you know to make sure that this is safe, you can have another cisco up call which says, you know first release the page and you give a particular physical address that I am releasing this. And, if and he should he better honor that, if he does not honoring that, he will probably get a seg fault very soon right yes.

Student: But there might be some algorithm, which are better than at global level. For example, disk scheduler. Now, for example, you have said that in elevator kind of thing in the elevator algorithm, if you have free knowledge of the sector which we need to leave we can persuade a more efficient way.

Ok.

Student: Now in this case every, now in this case every application will be running it is own disk scheduling algorithm.

Ok.

Student: So, it would just be a problem.

So, fair point question is the some algorithm or some policies that are best implemented at global level and not at a local level right. For example, may I do not want to run my page placement algorithm at a global level right, that may be the most efficient thing to do. As opposed to say that, I want to give a take a page from you and you decide which page to give right.

I may want to consider all the pages across all the processes at the global pool and then choose the one that is least recently used let us say. In this case I am you know I am basically pre committing to a process, that this is the process I want to take a page from and then I make an up call saying please release the page.

So, is not there, is not there an inefficiency there well yes, I mean. So, there is a tradeoff. Firstly, notice that there is there are two kinds of policies here; one is the policy that the application will implement internally, and then there is another policy at the OS level. Anyways, which is choosing, which process to ask right.

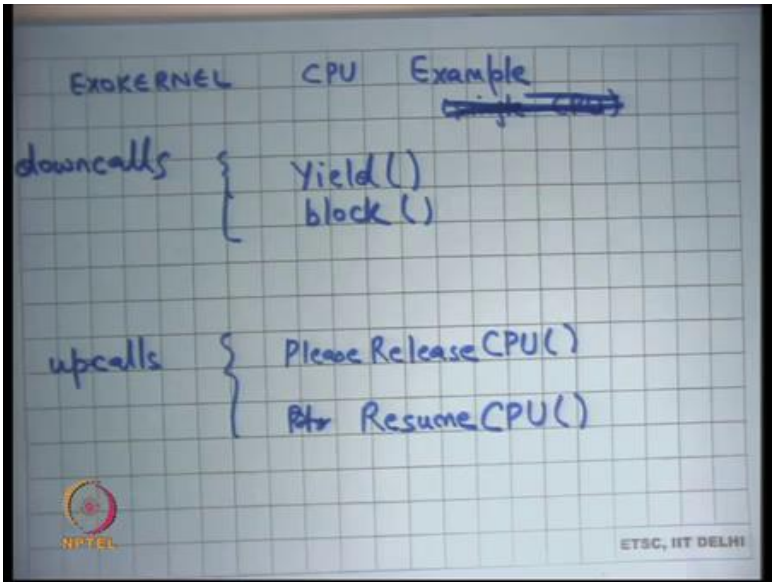
So, it already the kernel knows that each process has these many pages. So, there is there are two levels of policies. And, so there is still a policy inside the kernel that cannot be overridden, which basically saying, which is a global policy right.

So, what the exokernel is doing is separating the global policy from the local policy. Is saying let us let the application have the flexibility to at least decide the local policy,

global policy is still in the hands of the kernel. For example, it will choose it will save which process to give this up call to make, so, that is the global policy.

Similarly, you know any other shared device like the disks you took the example of disk scheduling same thing right. So, there is a global policy engine running in the kernel level, but you are giving the application the flexibility to at least decide the local policy, you are right. The application can still have no control over the global policy perfectly fine, perfectly valid point.

(Refer Slide Time: 15:05)



Let us take another example; exokernel CPU. So, you know you have down calls like yield and block; these are similar to a Unix they basically say I want to yield this CPU. By yield you mean I want to stop running on the CPU somebody else can run, at the same time I am ready to run. So, if there is nobody else to run I can you know I can be rescheduled back and block basically means that I am blocked so, you know you till I am woken up by somebody I am not available to run.

These are similar, but the interesting thing is up calls unlike Unix where, where the kernel would just take away the CPU from the application and we saw some problems with that right. So, the kernel can just say, I need a CPU back just take it away right.

So, irrespective of where the application was in it is execution, you could just take away the CPU from the application. And, there was a problem with that let us say the

application was actually holding a spinlock right. And, now within the critical section the CPU was taken away from the application, then all the other processes which were know depending on the spin lock will just waste their time spinning on this spin lock for the time quantum right.

So, that would up that was we used that example at the schedule when we were discussing scheduling right. And, that problem was basically because the application had no control over when there was a context switch. Here you are going to make an up call instead asking the application to please release the CPU.

So, the nice thing about this is when you ask the application to say please release the CPU, the application knows that I am in the middle of a critical section, it can actually execute the minimum number of instructions required to come out of the critical section and then call yield right. That is a much better scheduling paradigm than what we saw in Unix right.

So, it does not have this problem of getting interrupted in the middle of a critical section and so everybody else just keeps wasting the CPU, because it is just spins on the lock. Also, you can you know the application knows exactly which registers to save. So, let us say you know the application is just using 3 registers, so, the when it actually is called yield it just needs to save those 3 registers, before it actually says, I want to yield the CPU.

Similarly, there is an up call called resume CPU. So, unlike Unix where when you resume just start from where you preempted the process last time right. In case of in case in the case of exokernel, you will instead make an up call saying resume CPU. And, so up call is going to go to a certain point in the application, and the resume CPU up call will know exactly which registers to reload.

Depending on which registers were saved in my previous yield call right. So, you have saved on the amount of data that you need to save and restore also.

Student: But the data is being stored by the user program itself (Refer Time: 17:50).

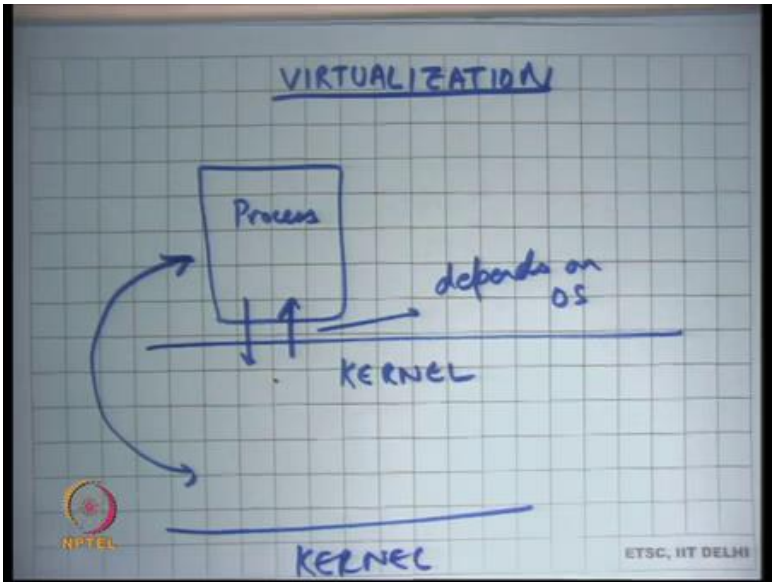
That data is being stored by the user program itself yes. So, the user it is a user programs responsibility to save and restore it is own data right. As opposed to and it has this flex,

you know you can give it that flexibility because you were making, you know you are requesting and resuming through you are giving him control over it. As opposed to Unix where you would just take it away and give it back, in which case it becomes a kernel's responsibility to save everything.

Student: Could it is that much of an advantages, because anyways only a small overhead.

Is that much of an advantage well you know registers are very small it is not a big deal really, but no in certain cases it may be a significant advantage. But, I think the main the most important thing is the scheduling a point that I discussed which is basically, that you do not have problems like the convoy effect where basically you if one process gets interrupted in the middle of a critical section, then there is a very bad scheduling behavior that you get good all right. So, with that I am going to move to my next topic, which is virtualization ok.

(Refer Slide Time: 19:07)



This a somewhat relatively modern topic. Are there practical examples of exokernels? Well so, exokernels were first proposed in late 90s as a research paper. And, you know many ideas from the exokernel paper have been used in modern kernels, especially in the modern kernels where we are basically looking at kernels that scale across lot lots of CPUs, like 60 to 80 CPUs you know these kinds of ideas are actually proving very useful.

And, so many ideas from the exokernel paper are being used in modern kernels. Especially research kernels today which are targeting a large multi core CPU machines all right. So, let us talk about virtualization. So far, we saw that there was a kernel and then there was a process right.

And, there was an abstraction between the process and the kernel. And, this abstraction was designed such that the process can do what it wants to do and also it is safe. This abstraction dependent on the kernel, it depends on the OS let us say. For example, windows will have a different abstraction from Linux; Linux version 2.2 will have a different abstraction from Linux version 3.0 and so on right.

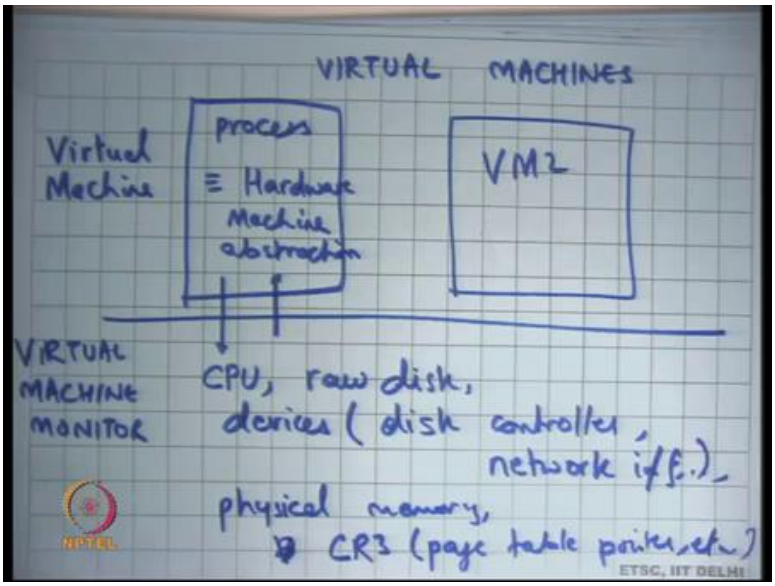
So, these abstractions are relatively very fluid it is very difficult to take a process that would run on a Linux 3.0 and make it run on Linux 2.2. Even more difficult to take a process that runs on windows and make it run on some on Linux let us say or vice versa right. So, all these things are relatively difficult.

Also, it is very difficult to let us say I have 2 machines a process has lots of bindings with the kernel. For example, it has a file descriptor table it has virtual memory etcetera. So, it is very difficult to migrate processes between life between two different machines. So, you have two different machines running, two different kernels; let us say they are even running the same kernel.

You know I cannot just take a process here and just say you know let us run it let us start running it here, you know I cannot take a running process here and then start resume and resuming it on the other side, because a lot of the state of the process is actually inside the kernel. And, there is a lot of state that a process can have.

And, you know as so far, we have seen given that it is a monolithic kernel, a monolithic kernel has a file system, a virtual memory of system and so there are all kinds of data structures that are living inside the kernel. And, so all those data structures need to be migrated as well and that seems like a very hard problem right.

(Refer Slide Time: 22:21)



So virtual so, virtual machines are a concept which says, let this abstraction that I am calling a process be somewhat equal to the hardware abstraction, machine abstraction. If, I could do this efficiently then you know the abstraction that I have with the with whatever is running beneath it let us call it the kernel for now. It is going to be simply that our CPU, a raw disk devices like the disk controller right, network card, network interface and so on.

And, physical memory and virtual memory subsystem, which are whatever the hardware supports. For example, in order all the registers like CR3, which is the which is pointed to the page table right. So, what if the abstraction that the kernel provides to the process is identical to the abstraction now the kernel sees itself on the hardware right.

So, the kernel is returned to assume a certain obstruction which is a hardware abstraction. The hardware abstraction is there is a CPU that runs one instruction after another, it has a certain instruction set jump allows you to change the program counter and so on the memory accesses.

And, then there is a virtual memory subsystem, which can be controlled by going through CR3 and you can configure page tables, there are devices hardware devices that can be accessed using in and out instructions and so on right. There is an interface of the hardware provides and is specified in the hardware manual of the manufacturer.

Can the same interface be the one that the kernel provides to the process? And, if it is able to do this efficiently, then the interface between the process and the application and the kernel becomes much more solid or much more compact, and much more much less fluid.

In the sense that hardware abstractions seldom change or change at a much slower pace than the kernel abstraction change right. You get a kernel version every few months, but the hardware version changes at a much slower rate. Moreover, hardware, preserves, what is called backward compatibility, you know anything that you could have run on a previous generation of a processor, or a previous generation of disk controller will run on the new generation of the disk controller and so on right.

So, if you could do that, then this abstraction becomes relatively solid. And, so it will be possible to take a process and you know run it on a different kernel. And, so you basically know that this state that this process has is very compact, it consists of the physical memory footprint, the page tables, the disk contents and the network interface state and other registers right.

So, these this is basically you can draw a line I mean this is exactly the state of this process and if I can save it and restore it somewhere else, I can migrate this process from one machine to another right. And, I can also migrate it across different versions of the kernel ok.

So, this abstraction is called the virtual machine abstraction ok, because it is just a machine that is implemented in a virtual manner in a process container. And, so you could run multiple virtual machines on one physical machine right, it is just like running multiple processes on one kernel except that the abstraction is has changed ok. The abstraction is not no longer of that of a Unix process, the abstraction is that of a machine of hardware that you are providing it. Does it mean that you are exposing all the hardware to the process directly or not?

So, well I mean; firstly, just like Unix could not trust it is processes, we do not want to trust our virtual machines right. So, you cannot just expose the hardware directly. The question is how does one implement these virtual machines efficiently? So, let me just also say that these containers are called virtual machines and the kernel is called the virtual machine monitor.

If, I am doing something like this; firstly, let us understand, what are the advantages of doing something like this? The advantages of doing something like this they are firstly, because the abstraction of the container is very solid, it is very well defined, it is written in the hardware manual. And, it is basically something that has a guarantee of backward called compatibility, you can actually take a virtual machine container and move it at will to some other virtual machine monitor and still expect it to run it exactly like it was running in the previous one machine right.

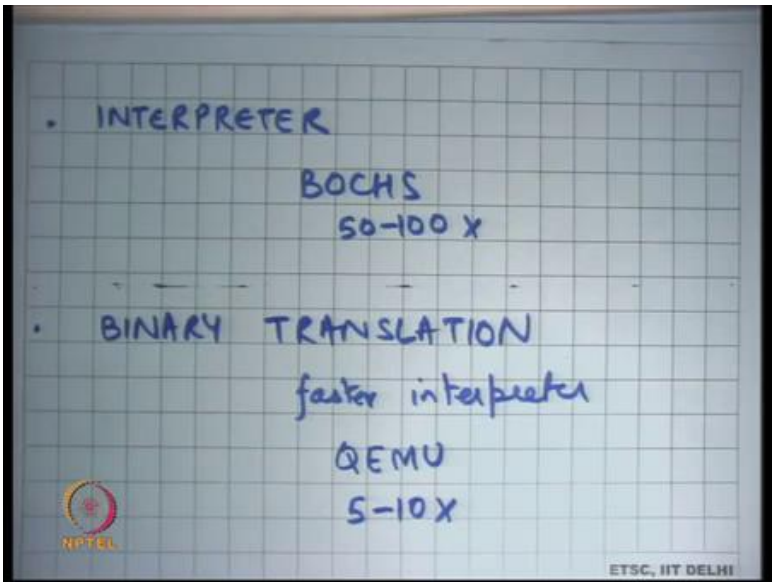
So, you can actually do migration across different machines, which would be a harder thing to do for something as amorphous as a Unix process all right. So, it can it is actually a very good fit for a distributed system where you can imagine that you have different lots of different machines, and you can take one virtual machine and just decide where to schedule this process right.

You can schedule it on this physical machine, or you can schedule it on that physical machine and so on. It also allows you to do other interesting things for example, you could run a full operating system within this container right. So, you could install let us say a Linux system, and spawn, virtual machine, and install a windows operating system inside that virtual machine.

Because, the abstraction is exactly that of the hardware, the windows operating system running inside this container well never actually know that is actually running inside a virtual machine, it will see exactly the same abstractions that would I have seen at the hardware level right.

So, there are there are lots of advantages to doing this. And, this virtualization forms the building blocks of what is also called as cloud computing and I want to discuss that more a little bit in a little bit, but let us first discuss how virtual machines are implemented it is if it comes to your question. Firstly, I need to implement them in some efficient manner; otherwise the whole thing is not practical. So, what is the simplest way to implement virtual machines, well one way to implement virtual machines is interpretation.

(Refer Slide Time: 29:41)



The interpreter based right. We just write an interpreter, which basically a (Refer Time: 29:53), which basically behaves exactly like what your hardware does and just takes one instruction executes it takes, the next instruction executes it and so on right. And, so you basically have an interpreter and you run this the code of the machine, which is executable now inside this interpreter and this the code should run right.

So, for example, a software like Bochs is a virtual machine monitor, could be called a virtual machine monitor, at least in the definition that we have so, far except that it is too slow right. It is going to take one instruction from the process, it is going to decode, it is going to see what registers it accesses, it is going to emulate those registers in memory, it is going to perform this operation and then go to the next instruction and so on right.

So, there is an interpreter loop each instruction has to internally execute 50 to a 100 more instructions and so this has an overhead of 50 to a 100x. Notice that this interpretation based approach is completely safe, the you do not need to expose raw hardware to the executable the executable sees virtual hardware. And, this virtual hardware is a virtual CPU, you could emulate a virtual device like a virtual network card and so on. And they are basically just talking to this virtual network card with the same interface that you would have had for a physical network card right.

And, the virtual network card internally is using let us say system calls exported by the virtual machine monitor to actually send packets on the physical wire ok. So, that is an

interpretation based virtual machine monitor it is really slow, but I it is all completely safe, I do not need to trust anything.

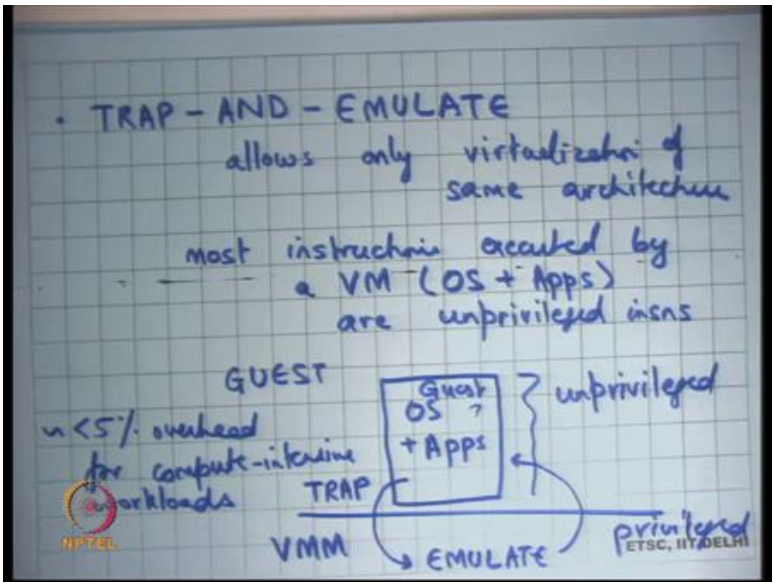
The other approach is basically, what is called binary translation, here the idea is that if there is a piece of code that gets executed a 100 times or a million times that is more likely. Then you do not need to run the interpreter loop every time you see that instruction, you can take those let us say you can take those a 100 instructions, and translate them to some safe counterpart, and then execute those 100 you know execute the translated counterpart a million times right.

So, it is a fast way of doing interpretation let us say it is a fast interpreter, a faster interpreter where basically instead of taking an instruction every time decoding it and x performing it is operations, you translate that instruction into the operations that need to be performed each time that instruction gets executed and you just jump to the translation right, that is basically.

And, an example of this kind of a system is QEMU, which is basically which you have used in your assignments they give roughly 5 to 10x overheads ok. So, it is an improvement over something like Bochs, but still not good enough right. I would not use something like QEMU to run a virtual machines for which I care about performance.

So, because they have such high overheads these software systems are not called virtual machine monitors. So, the additional requirement on a virtual machine monitor is that, the performance overhead of virtualization should be very small, it cannot be it should be in some percentage points it cannot be a 5 to 10x for example. So, the third way of and the most common way of doing virtualization is what is called trap and emulate ok.

(Refer Slide Time: 33:51)



So, what is done here is here you allow so, before I go forward let me also point out that in both these approaches interpreter and binary translation. I could have run a virtual machine for one architecture on the physical machine of another architecture nothing stops you from doing that right. I could run the virtual machine for the x86 architecture on a physical machine for power PC architecture right because I am just completely emulating the x86 architecture on power PC, it does not matter in over the underlying machine is.

Student: So, if the monitor would be modified accordingly.

Well the monitor you know will be modified accordingly in the sense that let us say if you are writing it in a higher level language like C, then it should be compiled for power PC that is all right. If, the underlying machine is power PC then you will compile it for power PC that is all otherwise the monitor's remains the same, but something like a trap and emulate allows only virtualization of same architecture.

So, which basically means that you can only run the virtual machine of the same architecture as the underlying physical architecture ok. And, the way it works is the key observation is that most instructions executed by VM and VM basically I mean the operating system that is running, and the applications are unprivileged instructions.

So, a huge majority 99.9 percent or more of the instructions that get executed inside a virtual machine, instructions that are very regular in nature like move, add, subtract, load store etcetera right. So, these are very regular instructions, these instructions would have had the same effect whether they are run in the user space or whether they have they run in the kernel space.

And, and so, these instructions the emulation of these instructions can, or the translation of these instructions can be made identity. So, if the guests are so, if the process which I am also going to call the guest right. So, the guest executes an instruction called an unprivileged instruction, I can just execute the same unprivileged instruction in the host and get the same behavior right.

And, so what you do is you take the operating system and the applications. So, the guest operating system and applications and run it in unprivileged mode right and the virtual machine monitor runs in privileged mode. Because most of the instructions are unprivileged in nature when they execute, they just execute as though they are running natively, if they execute any privileged instruction. For example, if they try to access CR3 right or if they try to execute in instruction or an out instruction to access the device you trap ok.

Just like and then you emulate just like you were doing in Bochs and QEMU you emulate inside the VM and then you return. So, what you do is you take. So, consider this disk image of a virtual machine as your executable right, that is the; that is the disk image, that is basically that basically lives on your hard disk for a physical machine, that is a hard disk (Refer Time: 38:05) the hard disk is basically the executable.

And, so now this hard disk in the virtual world is going to live in a file that is called we are going to call your virtual disk, and I am going to run this virtual disk inside this container called the virtual machine. And, this virtual disk is going to start running some of these instructions one after another, what you are going to do is you are going to run these instructions all these instructions in unprivileged mode. Even though these instructions were meant to some of these instructions, were meant to be run in privileged mode.

For example, when you boot a machine the first few instructions the assumes there is a privileged mode, anytime you transition to the kernel you assume that these instructions

are running the privileged mode, but the change you make as you run all these instructions in the unprivileged mode. Most of these instructions execute without any problem, because most of these instructions are unprivileged instructions. Any instruction that is a privileged instruction that must have been an instruction inside the guest who is kernel will cause a trap right.

So, you rely on some hardware properties that any which basically say that if you have tried to execute a privilege instruction in an unprivileged mode you always get a trap all right. So, it causes a trap and you handle the trap by emulating that instruction in software, just like you had done for Bochs or QEMU and then you return back to your guest right.

So, this kind of a virtualization is called trap and emulate virtualization and significantly faster, then either interpretation or binary translation and you will for something which is compute intensive you will have less than 5 percent overhead for compute intensive applications ok.

On the other hand, something that is very IO intensive, it is very likely that you are going to be executing lots of privilege instructions. If the guest is exec trying to access lots of you know virtual devices, then it is probably executing lots of in and out instructions or something that is privileged. And, anything that is privileged is going to cause a trap, it is going to execute some emulation logic inside the virtual machine monitor and then return back and so, there is extra overhead of doing this trap and emulation is going to become visible on IO intensive application.

So, on in IO intensive application, you can have overheads of up to let us say 2 to 3x or 4x, you know you have got go back get back to the QEMU kind of performance for IO intensive application. And, this is just a basic trap and emulate style virtualization there are many optimizations that can be done over it, and today you know with hardware support, and all that you can do virtual machine monitor virtual machines at almost negligible overhead.

So, what have we achieved we basically have a process abstraction that looks very much like the hardware abstraction and it runs with almost 0 percent over head right? That was you know that is the whole idea of a process attraction in the first place right. The

process should not be running too much lower than when you are running it over kernel as opposed to not running it over the kernel.

Is the VMM running on top of the kernel well consider VMM as a part of the kernels. So, VMM is yet another subsystem inside the kernel just like fs, virtual memory, drivers, there is yet another type system inside the kernel called the virtual machine monitor, great. It is a relatively modern subsystem, because you know virtualization is a relatively modern thing, it was very popular in the early days of computing in the 1960s, when machines used to be very large like the IBM mainframes and used to be really expensive.

So, the only way to actually make full use of these machines was basically lots of people sharing these machines and at that time virtual machines was a very popular concept and lots of research was done on you know using implementing fast virtual machines.

Over the years computers became smaller, they became personal, things did not have to be shared, computers did not have to be shared and so, virtualized it the technology of virtualization was lost. What do I mean by technology of virtualization was lost the hardware interfaces were not virtualization compatible?

For example, recall that one of the requirements to implement this trap and emulate style virtualization is that if I run a privilege instruction in an unprivileged mode it must trap right. So, but the hardware designers just did not care about it, because they thought these are personal computers. And, similarly the software stacks were not tuned for virtualization etcetera.

But, it was rediscovered because now today even though our hardware is relatively much cheaper the cost of actually managing this hardware, which includes power, store, power space, people who know how to manage these things, software maintenance up gradation, security viruses etcetera. All these problems basically make you know have moved us back to a and you know made the case for again a centralized computing infrastructure that lots of people are sharing and that is what we are calling cloud computing today right.

And, so one of the building blocks of cloud computing is virtualization. So, let us see why virtualization is helpful or why you know why cloud computing is a useful thing.

(Refer Slide Time: 43:47)

If, you look at the utilization levels. So, let us say this was time and let us say this was utilization right. So, what do I mean let me take one any one machine, let us say let us take the example of a web server, let us take the example of the web server of IIT Delhi right?

So, if I look at the CPU utilization of IIT Delhi's web server, a probably see in a let say this is 100 percent and this is 0, then it will probably we somewhere like this. Sometimes, there will be a surge some results are getting announced and then there is like this and then there is a surge again let us say etcetera right.

So, this will be the typical curve for CPU utilization of a server, same things for let say memory or disk, band width, or network right. So, usually we will get some kind of behavior, where on average the utilization levels of the machine are very low, but sometimes there is a very high serge in the demand, and you get very relatively higher utilization there was. And, so what typically do is we provision for the maximum we tried to provision for the maximum. So, the when there is a very high load, I do not see problems.

But, most of the times this maximum is actually getting wasted it may not it is getting wasted because I am probably consuming lot of power is still, I am consuming less power at 10 percent utilization and then when what I consuming 100 percent, but still I am consuming power and I am consuming the equal amount of the space definitely and

so on right. What would having better was if I had if I was able to run multiple machines on the same physical machine.

So, let say I have you know I have the IITD's web server and let say I have mail sever, which is let say my departments mail server the computer science, department mail server. And let say it has behavior like this, then these can easily we multiplexed one the same physical machine without seeing performance degradation on either application or yet having better utilization over for my physical machine over all right.

So, virtual machines allow this ok. And, they allow this without having to worry about software compare ability, you may say what is a big deal why do not I just have one Linux machine and run the mail server as one process, my departments mail server is one process, and you are your institutes web server as another process, and let them you know co-exist, and it is going to do exactly what you wanted to do.

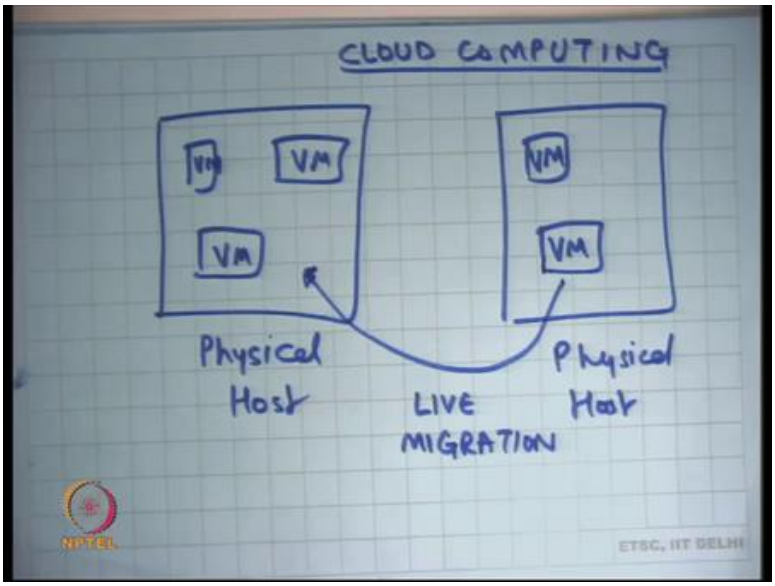
But very difficult to because the abstraction are so, fragile it is very difficult to ensure that the departments web server can be run in to one the same operating system as the depart institutes mail server without causing any problems with each other and without having any security implications right. So, this department web server may be less critical than the institutes web server and so, you may want that you know that it should be completely isolated from each other.

It something may not be that carefully written other thing may be carefully written. And, because the abstraction of the Unix abstraction let say, are very fragile you know for example, the configuration files etcetera, you know these processes will depend on the configuration files this slash etc folder etcetera.

It becomes a lot of it becomes a mess after less time to manage lots of different processes running on a single operating systems. On the other hand, the virtual machine abstraction is very clean, it is the hardware abstraction. So, I can just take one hardware abstraction and run it and I can I cannot have to I do not want to worry about security problems or interference problems between these two virtual machines.

So, what this allows me to do is basically let say, I have this is a physical machine let me call it a physical host and these are virtual machines, VM, VM, VM.

(Refer Slide Time: 48:07)



This is another host this is VM right. So, I am showing you cloud computing scenarios what is called cloud computing in the modern world. So, this is what a cloud computing environment will look like, you will have multiple physical hosts and you would be running typically multiple virtual machines within a physical host. Typically, you would be you could depending on the utilization levels of each VM, you know you could pack multiple VMs on the single host it is very common to see tens of VMs packed on a single physical host. Because, the utilization levels of each of these VMs is very low right.

And, so you could be running these things together the nice thing is because is abstractions are so, tight I can at any time decide to move this virtual machine from here to here right. So, this capability is what is called live migration, this can be done in a way such that the virtual machine has near 0 down time right. So, it appears that the virtual machine never switched off right.

So, there is a you know one way to show this is basically this is interesting demo where let say you are watching movie on a virtual machine, a live migration happens which means the virtual machine shifts from one physical host to another. There is some amount of copying that is going on for all the memory state etcetera, it is all happening in the background and at some point the entire virtual machine shifts from one host to the other and you do not want to see of blip in your movie right.

So, that is basically that is what I mean by a near 0 down time live migration. So, that is the very wave power capability what allows me to do is it basically allows me to dynamically schedule my work across my physical host, at will and that is one of the main strength of cloud computing. Let say I have you know let say these 3 VMs become very highly utilized certainly.

So, I can choose to move this VM from here to here right, it is all let say it is in the middle of the night and none of these VMs are actually using any resources at all. So, I can pack all these VMs on a single physical host and actually switch of this particular physical host and save power right. Also, I can do fault tolerance which basically means now your virtual machine state can be encapsulated as a files.

So, running virtual machine can be snapshotted and it is state can be preserved, it is live state can be preserved as a file, and it can be loaded from where it was at will to basically get exactly what how it was when it was snapshotted right. That allows me to do fault tolerance; fault tolerance basically means, if there was an error in either hardware error, or network error, or power failure I it can revert to this snapshot of a virtual machine and basically get the same kind of behavior.

So, cloud computing has its advantages basically, they have talked about for the more interesting thing from operating system standpoint is that it brings open all the issues that we thought were dead for a long time. For example, scheduling now becomes a very important issue right. If, you are running a cloud computing facility if you running a data centre and becoming very important that you managed the data centre with maximum utilization and the maximum utilization depends on what scheduling you will got them you using right.

And, like the personal computer world were generally the hardware resources are plenty in plenty, in cloud computing environment you basically sharing these resources across lots of people and you have minimize trying to minimize your costs. So, resources are never plenty you are basically always trying to optimize your resources in some sense all right ok.

So, with that we finish this course.