

Storage Systems
Dr. K. Gopinath
Department of Computer Science and Engineering
Indian Institute of Science, Bangalore

Storage media, Storage interfaces, Storage access mechanisms, Storage Protocols
Lecture - 05
Storage access mechanisms in Networked/Non-networked/Web storage systems,
Layered architecture in Storage systems, Introduction to Hard Disks, Hard Disk
scheduling algorithms

Today we will look at some more details, but I hinted at last time and go a bit into some more detail.

(Refer Slide Time: 00:30)

Basic Storage API

- GetNew(oid)
 - POSIX: fd=creat(const char *path, mode_t mode)
- Store(oid, data)
 - POSIX: ssize_t write(int fd, const void *buf, size_t count)
- Read(oid, buffer)
 - POSIX: ssize_t read(int fd, void *buf, size_t count)
- Delete(oid)
 - POSIX: int unlink(const char *pathname)
- GetInfo(oid)
 - POSIX: int stat(const char *path, struct stat *buf)

oid: blocknum, filename, filehandle, content hash, key

Also, appl buffered versions.

First let us look at the basic storage API; what are these? Is a basically how you access storage and what are the other additional things that you need to make sure that your system is usable. So, let us say that they are something called an object id which refers to something that you want to access. The object id can be a block number, a file name, a file handle, a content hash or a key or it could be other things also, but let us just think about the following.

So, I want to be able to get an id, I want to be able to store some data correspond that id, I want to be able to given the id, I want to be able to read it into a buffer, sometime later I might want to delete it, but sometimes I want to get the metadata of the object itself. Now, this is at some abstract level, these are some of the important ways to interact ways

to a system. Basically, I needed to create objects, ability to store, read, delete, get information of the object itself.

Now in Unix, there is something called POSIX which is a portable model in Unix systems and you will see that there is a corresponding thing for each of these functionalities. For example, the ability to create an object id is often done through an operation called create and basically saying that I want you to create a file with a particular mode. For example, (Refer Time: 02:26) a read, write etcetera. And it also gives you a handle. Fd is basically a file handle so that you do not have to refer to this path every single time. It turns out that parsing this path is expensive, so the idea is to avoid doing that and create a convenient handle and the handle is used in the future to refer to this object.

Similarly, in POSIX, to store you have something called a write. You give the handle that was created here and because a storing data, you need to specify where you are getting a data from. That is, this part and you also specify how many bytes you really want to store. And this particular thing returns the size of the thing actually that also return. So many case of read you need a specify where that object the contents have to be stored into. That is, a buffer and it also tells you how big you want to read and how much actually succeed in reading.

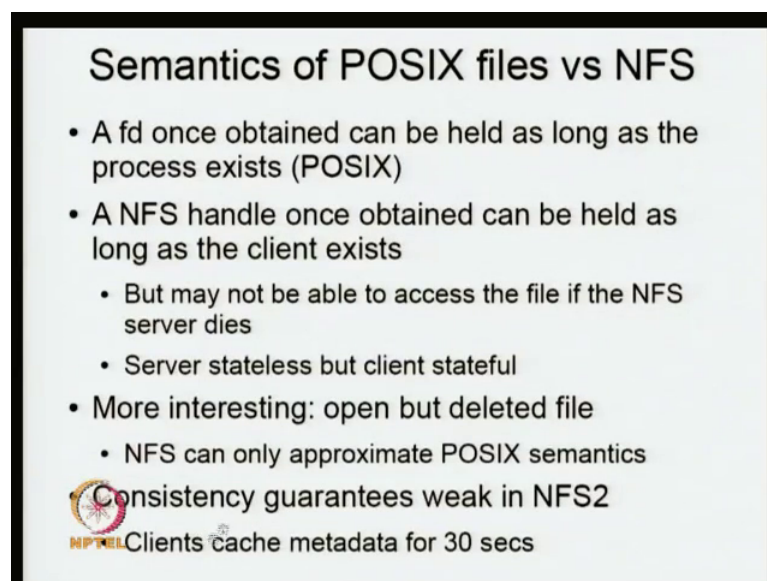
You will again see something interesting here; when it comes to delete, it turns out that you have something called unlink which actually does this deletion part. This is not identical to what delete could be. Basically because, in POSIX it turns out that there could be multiple names which can refer to the same object. So, when you unlink it, you are all only doing is you are doing a reducing, a reference count by one. If the reference count is zero, then you actually have the same function till delete. You can also have something called getting the information about the object itself. This is what is called a stat call, and basically the information that you get about the object is put into the buffer.

Now, this object id as I mentioned to you can be different things: it can be a block number, that means that it does not have a user visible name, it is just got a number. It can be a file name that means, it has got a user visible name. It can be a file handle because it turns out that sometimes for example, in certain storage systems you give it just like a file descriptor. Here you give it a file handle and the file handle is used. I will

come to that soon. In some other systems, you have a content hash. Basically, you retrieve an object by giving a hash of the some parts of the file or the name itself. Similarly, the key also could be used in some (Refer Time: 04:59).


Now these are basic API and this is in different contexts. You will see that this differs in different ways. For example, applications might be very concerned about performance, so they might write their own buffered versions. So, that is something that we have to think about, ok.

(Refer Slide Time: 05:24)



Semantics of POSIX files vs NFS

- A fd once obtained can be held as long as the process exists (POSIX)
- A NFS handle once obtained can be held as long as the client exists
 - But may not be able to access the file if the NFS server dies
 - Server stateless but client stateful
- More interesting: open but deleted file
 - NFS can only approximate POSIX semantics

 Consistency guarantees weak in NFS2
Clients cache metadata for 30 secs

Now, let us just quickly look at how the semantics of these things changes once you introduce something like a network. I think some of might already know NFS stands for network file systems. Basically, this, once I get Ethernet came into picture, then the feasibility of accessing storage in remote nodes became possible and network file systems for device. About early 1980s and they are still use quite widely. And basically, you will see that networking actually has an impact on our semantics. We will just look at how it; how it happens.

Let us see the first one. In POSIX, if you have a file descriptor. If you have a file descriptor you can hold it as long as the process exists. If the process disappears, you lost the handle and you have to again get another handle. So, it is not something which is permanent. It is only as long as the process exists. Whereas, the NFS it is a different semantics because it is now across the storages across multiple nodes. Now there is a

notion of a client and server. Now the file handle is kept by the client. So, the handle is valid as long as a client exists. The client disappears, of course, then you do not have worry about the server having to produce that object when requested, but once a client has a handle, the server by definition is expected to be able to retrieve that object once the client produces that handling. Ok?

So, of course, this is impossible in practice because there is no way to keep a, you know, a server to always respond to any handle because finally, let us say, essentially it is unlimited amount of time. It is just impossible. So, but anyway, the in principle, it is means that in reasoning and reasonably long times for a long time. Given, our handle (Refer Time: 07:35) should be able to produce it. So, for example, in the NFS case, if the server dies the client still has the handle. Then it is expected that once the server comes out it is still able to service that particular request, ok.

So, the basic design in NFS is server is stateless, but the client is state full. So, the server does not remember anything about whether it has let us say any files that were accessed in the past etcetera; does not know anything about these things. It does not keep track of all those things, but given that there is a handle given out, the next time somebody asks for it you able to provide it. So, in a sense that handle should encode something with the server can used to retrieve the object. So, you can see there is a difference in the way the semantics has changed. This a file descriptor in the context of single machine is a file descriptor is valid only as long as the process exists, but in the case NFS the handle exists, a handle is meaningful as long as the client exists.

Similar things happen at the case of some other types of usages or idioms in certain file system. For example, in many file systems you have a notion of what is called open, but deleted file. And this kind of idiom exists because it you want to unnecessarily not keep around files that are temporary. So, idea is that you create a temporary file and you open it, but then delete it, and the idea is that when the process exists, that file also is removed automatically. That is the basic idea and it turns out that this one is not feasible to be supported with the same semantics in NFS. It turns out the NFS client has to do certain tricky things to make show that this particular thing happens.

Similarly, in the case of consistency, it turns out that NFS is especially version 2 has very weak consistency guarantees. The reason why this happens is because the clients cache


metadata for some amount of time. Let us say for 30 seconds. So, it assumes that nobody has modified it for 30 seconds, but they of course, is a totally unsustainable argument. When you have lots of multiple parties updating the same file, but this is what is assumed. So NFS later versions, they have notions of what is called a lease by which you can essentially it is not allowed to be modified in multiple parties until the owner actually that guy, that persons lease expires ok.

So, there are, there is a lot of certain difference of that crop up. Once you allow certain different types of access or certain something like networking into the picture.

(Refer Slide Time: 10:38)

Impact of Networking

- Semantics of failure becomes imp
 - NFS uses RPC. What has to be done wrt non-idempotent operations?
- Consistency issues become important
 - NFS semantics different from POSIX
- To support high speed transfers, new kernel infrastructure
 - Parallel to IPC: sockets/TLI
 - For even higher speeds, user-level networking (RDMA)
- Storage spun off in large systems
 - Storage Area Networks (block level protocols)
 - NFS (file level protocols)

 Distributed File Systems/Storage Systems

Again, let us just look at the impact of networking in a slightly different angle. It turns out that the semantics of failure become quite important also. For example, NFS you saw something called remote procedure calls. And once you have this you have to start worrying about what is called non idempotent operators. What are these? For example, normally when the client requests some service, either it can be done any number of times without the result being different.

For example, if I can ask a read to be done on a particular file, I can do it multiple times and still the same result should come out. Whereas, so that will be the example of an idempotent operation, if you have non idempotent operations, it means that you cannot repeat it multiple times and get the same result. A good example is, suppose I delete a file, the first time the file exists. So, when I delete it, it probably says yes I succeeded in

deleting it, but the second time you repeat it, that file it is already gone. Therefore, it should return saying that I did not find the file.

So, there are certain non idempotent operations. Why this is important in the case of RPC or in the case of networking? It is because I ask you to do something, it may be that. That the operation was done, but the response got lost or it might be that my requisition of first place never made it. So, the idea in NFS has been always that you do at least you keep repeating, you retry the operation. So, if you have to retry a delete operation for example, then you will get different results depending on exactly what happened. So, the semantics of failure become very important and so, this is different again from POSIX.


Similarly, the consistency issues become important as I mentioned before. It also turns out that you might need new kernel infrastructure because you are doing high speed transfers. You might need different kinds of infrastructure and again, as we looked at in the first class, it turns out that the (Refer Time: 12:46) of networking you can start doing different types of a larger scale systems. For example, storage area networks, where we use protocols at the block level or we can do it at the file level, which is basically what I mentioned just now. This network file systems or you can think of even larger scale systems distributed file systems or storage systems.

So, the impact of any one particular thing like networking has (Refer Time: 13:10) all over the place ok.

(Refer Slide Time: 13:13)

API changes

- POSIX: Read (fd, buffer, count)
 - Partial writes to a file OK (appends, overwrites, etc)
 - Mmap
- NFS: Read (fd, **offset**, buffer, count)
 - Partial writes and mmap avlbl but no open!
 - Weak consistency model with multiple writers (NFS2)
 - NFS3, NFS4 improve the consistency model
- Amazon S3: "storage" service
 - Key Value store: no features like partial write or mmap
 - Weak consistency ("BASE") model: when no updates occur for a "sufficiently long" period of time, eventually all updates will propagate through the system and all the replicas will be consistent.



Again, you can see how the API it is of changes. In POSIX you say read, file descriptor buffer and count; and POSIX also allows you to do partial writes. You can do appends, we can do overwrites, you can do seeks etcetera; you can do an operation called mmap also which maps a particular file to certain some variation of the shared memory. Whereas, you look at NFS, you will find that because it is a stateless model, it can never remember that implicit pointer where it was supposed to be reading it. In the case of POSIX, there is as there is always an implicit. Let us say pointer from where you are reading things.

Whereas, NFS the server does not keep track of this quote implicit pointer because it is stateless. So, therefore, the client has to explicitly mention the offset. Everything at the center, buffer and count are same, but the client has to supply an additional argument called offset. Again, the NFS has been designed to be as close to POSIX as possible. Therefore, partial right sort of supported and mmap also is supported and this is very critical because without mmap, we cannot support certain virtual memory operations and also exacting files, exacting executables residing in remote file systems ok.

But there are something (Refer Time: 14:34) also missing again from the point of view of stateless design. For example, there is no open because the server is not so much to keep track of what files are open because it is a stateless design and again, NFS as I mentioned before it has a weak consistency model with multiple writers and later models like NFS3 and NFS4 improve the consistency models. For example, NFS4 has a notion of leases; that means that somebody wants to write it. He takes a lease and then that party only has access to that file for updating purposes and only once a lease expires somebody else can get hold of it ok.

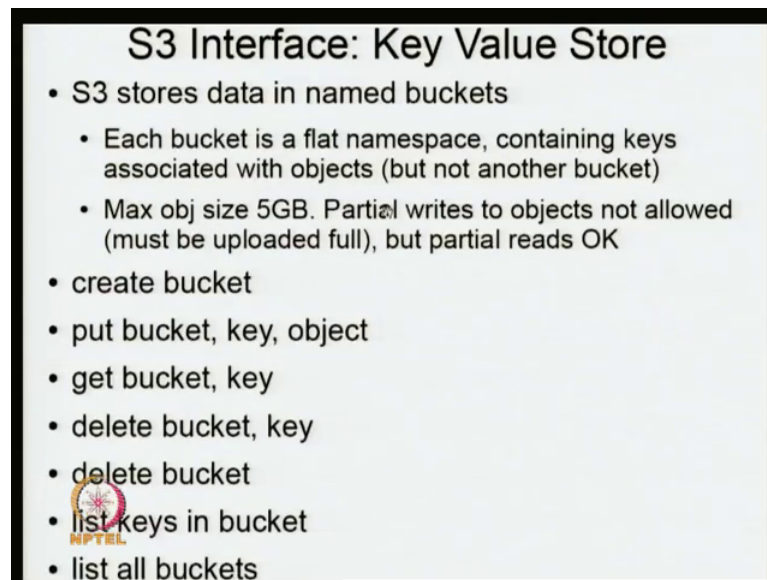
Now let us look at this is one is POSIX one is NFS. Now let us look at the more recent version of how to access storage. Amazon has a S3 storage service. It is what is called a key value store and what it means is that you do not have some other various aspect like in POSIX. All you have is a key value store. What it means is given a key it produces a value and they do not give you any other additional details and it does not have features like partial right nor can it support mmap. This things are make sense because S3 is designed for using what is called the for cloud services; that means, that nobody is going to do let us say using S3 for something like virtual memory operations that does not make sense for this.

Because you are not thinking of supporting virtual memory operations, you know you do not need mmap. Similarly, features like partial write are not supported because it is too costly for them to support this model. Because all they have is a particular key, has a particular value. If you keep writing partial writes; that means, the key might have to keep changing which is a problem. Thus, they do not try to support it. If you look at NFS, it has certainly a weaker consistency model than what we are used to in a Unix system. But it turns out Amazon will go for even weaker models. For example, because this happening on wide area networks where basically networks can fail.

So, or you can get the data from multiple paths. It turns out that the multiple updates can come out of order. Because they can come out of order, they cannot guarantee that consistent views maintained across all clients. So, they have a slightly different model which is called eventually consistent model and although says the following, which is, very weak guarantee. It just basically says that where no updates occur for a sufficiently long period of time eventually, all updates will propagate through the system and all the replicas will be consistent. That is all. You can take the reason why you talk about replicas is because it is a service on this on the cloud and networks etcetera are much more not as reliable as electrical connections within a single desktop system, for example.

Therefore, you often times have multiple replicas to avoid failures. If one of them is inaccessible and so, you have to any particular write has to be propagated to multiple places and if you try to propagate to multiple places then you need to have some way of saying. When those three copies or four copies, they are consistent with each other and that cannot be guaranteed easily in the cloud. So, that is why they have a slightly weaker model and with this weaker model we have to find a way of a accessing your storage.

(Refer Slide Time: 18:18)



S3 Interface: Key Value Store

- S3 stores data in named buckets
 - Each bucket is a flat namespace, containing keys associated with objects (but not another bucket)
 - Max obj size 5GB. Partial writes to objects not allowed (must be uploaded full), but partial reads OK
- create bucket
- put bucket, key, object
- get bucket, key
- delete bucket, key
- delete bucket
- list keys in bucket
- list all buckets

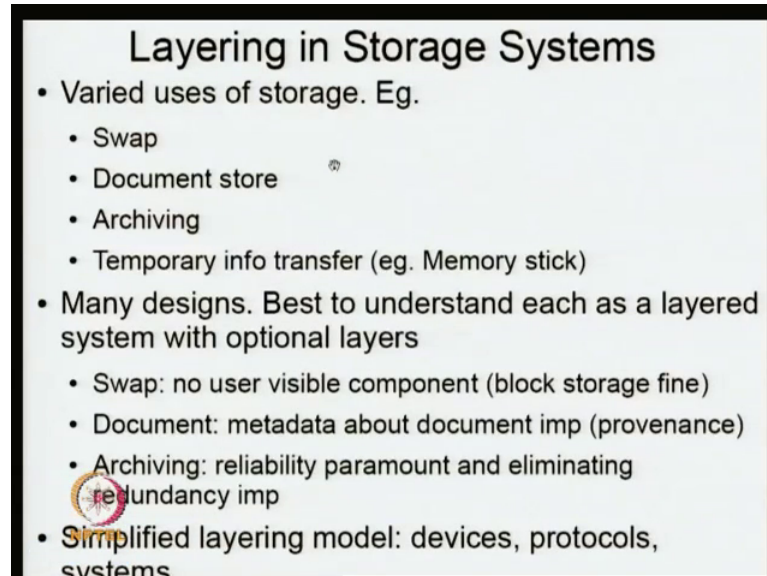
I can, let us just quickly look at the kinds of operations supported by something like the Amazon storage. Say this, this S3 stores data in named buckets, each bucket is the flat name space containing keys ok.

So basically, for each key there is an object and you put multiple objects in a bucket. The maximum object size is 5 gigabytes. It turns out that partial reads are allowed, but as I mentioned earlier partial writes are not allowed; that means, that if you want to update something partially, you need to come up with another object that which you can upload and say. So, the kinds of operations that they support or create bucket again. This similar to the POSIX create put bucket key object; this similar to the write operation although are saying is in this particular bucket. I want you to write this object with a particular key here is basically access is a read call. Basically you are saying that given the key, please give me the in a particular, a bucket. Please give me the object.

I can delete the particular object given the key I can delete the complete bucket. That is, all the objects represent in that again you can think of a bucket as something like a directory or you can list all the keys in a bucket and you can just list all the buckets also. So, this is the type of operation supported. So, usually they do not go beyond this. For example, there is no mmap etcetera as I mentioned before. So, basically this is a slightly more limited model, but if you are storing things like photographs and what not probably this is a good enough model. You do not need anything much more complex in this.

But if you are planning to support the storage has to support something like virtual memory operations. Then this will not be sufficient.

(Refer Slide Time: 20:17)



Layering in Storage Systems

- Varied uses of storage. Eg.
 - Swap
 - Document store
 - Archiving
 - Temporary info transfer (eg. Memory stick)
- Many designs. Best to understand each as a layered system with optional layers
 - Swap: no user visible component (block storage fine)
 - Document: metadata about document imp (provenance)
 - Archiving: reliability paramount and eliminating redundancy imp
- Simplified layering model: devices, protocols, systems

So, this brings us to why there are differences in the storage systems and how they can be let us say made sense off. So first of all, again to reiterate the point, there are varied uses of storage. For example, I can use storage for swap purposes in a desktop system, I can use it as a document store, for example, I store photographs etcetera, I can do archiving that is, I need to keep the storage for a long period of time because it is required by legal or other reasons or I can even do something simpler like, I use some storage for temporary information transfer. For example, I carry a memory stick and I want to transfer it from one machine to another machine.

So, a varieties of uses of storage and so, there are many designs for each of these things. I just listed a few of them, but there can be quite a few more. Now if I want to understand the all. The varieties in which I can use a storage, it turns out it is good to understand the various possibilities by thinking of a storage system as a layered system with many optional layers depending on what layers are produced. I can have different kinds of functionalities. For example, if I take a swap on a desktop system usually the places where I store the pages the user does not have to really know those names etcetera. It is the job of the operating system to keep track of it. There is no user visible component.

So, for this block storage is quite fine, I do not need anything better than this. If I have a document, for example, there could be something more interesting that I need to do here. Basically, because documents are have lot of importance in society, it can for example, if you in India for example, have this something called write information act. So, if some document is produced you may want to know who produced it, when was it produced, how it got modified etcetera; that means, that you need to keep a lot of metadata about the document itself. And basically, but what is often called provenance. It is very important. So, documents stores for example, worry a lot about provenance because legal cases are fought on base of this provenance emails are important. When does end of email is sent that is important issue ok.

So, you need to be able to store it and most important is that in some cases you need to make sure that this provenance does not change. You cannot modify the provenance, you cannot lie about how the some aspect metadata. So, this particular kind of systems will have a slightly different focus because they are worried about legal aspects. Sometimes in archiving, there is a different aspect altogether. You worry about liability because if i, if I am worried about the fact that some data might disappear, I make copies. I archive it let us say.

And if my archiving cannot give me good guarantees, then it is of the useless because the reason why I am archiving it is because to take care of a contingent situation of losing it which may happen very rarely actually. Usually does not happen that often, but if in that real situation if it is not going to work then it is of the useless, at least, that this has to be a rocks on it. Whatever I do, it has to be absolutely trustworthy in terms of ability to produce a document when requested again ok.

So, this quite critical provide aspect that is important archiving which is not. So, important on some of these other things is eliminating redundancy. Why because as time progresses I keep on accumulating data or documents. So, I cannot keep on keeping multiple copies or extra copies without control. I need to make sure that I need to keep only controlled amount of redundancy. That not unbounded redundancy I have to make sure that if it is possible for me to store it with less amount of storage it is better. I will introduce redundancy as needed with respect to as I driven from reliability.

But there should be control on how much redundancy is there. It cannot be just arbitrary. So, archiving will have different ideas. So, it turns out that if you think of storage systems you might think of it as multiple layers and it helps us understanding systems by thinking that there is some kind of layering there and we will follow a very simple model. For time being, it is just say that, it is based on devices protocols and systems. Actually system is much more complicated, but we will think of this as call a start.

(Refer Slide Time: 25:04)

Storage Systems Highly Layered

- Multiple layers. Example:
 - Application uses fopen, fread, fwrite, etc.
 - Libc calls open, read, write system calls
 - Kernel calls vop_open, vop_read, vop_write, ...
 - FS implements ufs_open, ufs_read, etc. using virtual memory subsystem
 - Virtual Memory subsystem uses vop_getpage and vop_putpage provided by FS
 - vop_getpage/vop_putpage call pseudo device routines
 - Volume Manager or NFS client code
 - Device Driver (SCSI) or Network driver
 - HBA or NIC
 - Disk or Remote Disk

And then to give an example how deep it can be we will just take an example ok.

As I mentioned we will just think of it as some simpler model, but actual system is fairly complex. Let us take a simple access to your storage system. Again, we will use a traditional desktop system. An application will access some piece of a storage using something called fopen, fread or fwrite. These are calls that are the libc level and libc actually in turn translates this into some system calls open, read and write system calls a system calls. Fine they finally, are executed in the Kernel and the Kernel finally, actually calls something is called VOP, underscore, open etcetera. This is way to ensure that the Kernel is flexible with respect to the different as a file system that can be under on the different type of file system that can be on the desktop system. So, so this is the file system independent way of calling those routines by the Kernel again. This gets translated into a file system specific call.

For example, there is something called Unix file system that is, there in BSD file systems and some other operating systems and so this file system independent open call is translated through certain function pointers into a file system. Specific call UFS, open UFS, read etcetera and these things are implemented by the file system using the virtual memory subsystem and virtual memory subsystem intern uses the file system to actually access the disk, actually access the file system. It uses something called getpage and putpage and that the lower level of recursion. These things actually call the device routines and again there is a huge amount of layering going on here. Thus I am calling it a pseudo device routine.

We actually does not even touch the actual device. You get a disk, it is going through layers of software before it finally, touches that the disk. So, for example, there could be something called a volume manager. A volume manager is often used to aggregate multiple disks. So, you might want to have a disk which is let us say 100 gigabytes whereas, the biggest that is available is right now 3. Made a mistake, I am sorry. The, you may want to have a storage device which is that is a 100 terabytes, but the biggest disk available now is 3 terabytes.

So, is it possible to construct a 100 terabyte virtual disk out of multiple 3 terabyte disks. That is what a volume manager does. So, this again is a piece of software. It is not hardware. This is basically piece of software. Again this volume manager written talks to a device driver again is a piece of software which talks to for example, varieties of disks. We will talk about SCSI disks mostly because this is the most flexible type of disk and this particular SCSI disk will talk to an agent called HBA. So, that it insulates itself from the many interrupts the device will send to the CPU otherwise, host bus adapter. This host bus adapter actually intern final talks to disk. This is one type of layering that is going on if you are talking about networked file system from VOP getpage. You can actually go to the NFS client code and that in turn will talk to a network driver.

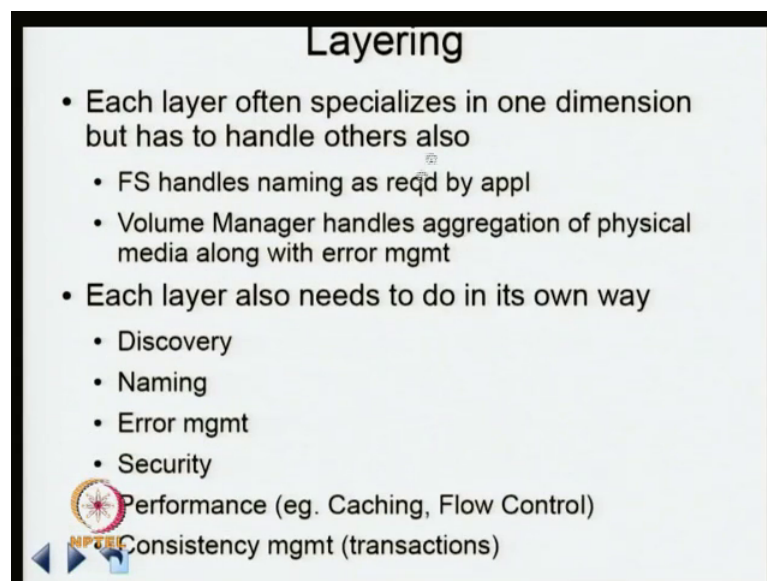
A network driver in turn will talk to the network interface card and that in turn goes to the other side to the remote disk and it has to traverse another. It is own set of these limits on the other remote disk. On the remotes machine it the form nic. It might go to the other remote machine and there it might have to go through a volume manager and device driver HBA and disk. You can see that from this there is a fairly large sequence. Something that happens and you will actually so, see that this network stack achieves a

part of your storage stack because this network stack is on here and all those stack is here.

So, the 7 layer network layering model that you might have come across in ISO model in whatever form; it is also present somewhere here. Storage systems have not been formalized to that extent like the network systems because by definition, the network systems have to interoperate. They have to interoperate. Without that, there is no point networking. So, if you want to interoperate you have to really codify exactly how you can talk to each other. That is why there has been serious attempts that are coming up with layers of a which can be mapped through each other in spite of different systems.

Whereas, in storage systems, it does not happen to the same extent, but is beginning to happen even though, it has a similar kind of multiple layers.

(Refer Slide Time: 30:16)



Layering

- Each layer often specializes in one dimension but has to handle others also
 - FS handles naming as reqd by appl
 - Volume Manager handles aggregation of physical media along with error mgmt
- Each layer also needs to do in its own way
 - Discovery
 - Naming
 - Error mgmt
 - Security
 - Performance (eg. Caching, Flow Control)
 - Consistency mgmt (transactions)

So again, so we will just look at layering a bit. So, what do you mean? What do you mean by layers? Basically these are either different software layers along with certain a possibly; certain layers being in hardware itself and each of them specializes in one dimension, but each layer also typically handles some other dimensions also. A good example let us take the case of file system.

Typically the file system is available or used because it provides a convenient naming as required by applications and end users. There is also a notion what is called the raw disk.

For example, where no naming is provided and it is very difficult to use database. For example, can use or disks, but managing those disks becomes difficult and therefore, often times many files many databases actually run on top of a file system because a file system takes care of some of this issues about managing the so, many raw disk blocks. So, file system has a particular functionality as I mentioned. One important thing is naming.

Similarly, volume manager handles aggregation of physical media along with error management. It is a different kinds of functionally. Now it turns out that if you separate this functionality in these two layers. File system doing once in a volume manager sometimes it simplifies the software design. So, that there is some kind of a division of labor, but the same time it can also create complications basically because if each of them is designed independently then certain types of performance guarantees that you want to provide may not be possible. It might become more difficult ok.

So, layering is good. Simplifies your ability to come up with software, but it can also prevents anything from happening. A good example is, let us take real time aspects with respect to access. Now, we have to ensure that if there is somebody doing real time access for example, you have what is called multimedia file systems and you might want to guarantee that each access happens within particular amount of time. Now if you have split into 2 parts, file system and volume manager then it means that if you want to give real time guarantees then you have to work both of them have to work together.

But if they are independently designed it might be not so, easy to do it. So, that is reason why oftentimes if you are thinking about real time aspects or security or any other aspects a integrated design is often better and often times you will find that this kind of layering sometimes can be problematic and so depending on the seriousness. Seriousness of which you have to support certain a property you may decide to go with a layering or you might want to club the different layers in different ways and or you might want to do a with layering that interface with the property that you want to support.

For example, there are file systems like XFS which club file system in the naming part, as well as the volume manager together. There are some other designs which do with the different things. For example, Linux typically has this 2 different file system and volume managers, that 2 different things now, one aspect over layering is that each layer also

needs to do in its own way some various functionalities. For example, discovery, naming, error management, security, performance.

Let me just mention one or two things about each of these things. What is discovery? Discovery means, if there is a layer, it has to figure out what is out there. Let us say we are at the lowest level, the device driver needs to know how many devices are there. So, the devices may be attached to a bus and at the booting time it has to figure out what are all the disks that out there or suppose, at I am at the NFS level, I want to find out which network file systems are available that I can mount I should be able to access.

So, there are remote. Many remote machines exporting many different remote file systems and I need to figure out where they are. Let some kind of discovery how is there a when I can discover where things are and notice that this has to be there at every layer. It is not just has to be constant in one layer. It has to be there at every layer naming also same story. We already looked at file system that it provides naming as required by application, sort or users. You will find that this naming also is required for at multiple layers. Again to look at, the most simple case list, in case of disks in Linux for example, we have something is called slash dev SDA slash dev SDB etcetera ok.

So you have 2 disks, one of them might be called a slash dev SDA another thing can be called slash dev SDB. Now Linux has used. This simple model for quite some time and this turns out to be somewhat problematic when you go to very large scale systems. Suppose you are running a big database system and require about 500 disks. Now having all this slash dev SDA, SDB etcetera; it becomes a big headache. You actually want to do it slightly more differently and typically what you have to do is you have to do it based on the structure of the system. So, structure by which you are actually connecting all the disk themselves.

So, there are different more sophisticated models by which you name these disks and so, for large installations which are becomes very important and it is very crucial because in when a system is running, when a disk fails, you might have to replace it. When you want to replace it you may have to not make you should of course, ideally not make any mistakes. So, for that you know this kind of things are very critical; how to name them, how to ensure that the right things happen when some error management has to take place, ok similarly error management. So, the error management that happens at a file

system is different from that happens from a volume manager and that is different from what happens in the disk flavor for example, if you are at NFS level network file system level if you try to access a file on the remote file on the remote machine then you retry ok.

So, you keep retrying till you get it because you have a handle and NFS guarantees that if you have handled, you are supposed to be able to access the access the file or the object. So, all it does is retry and there are various models. Something called hard retries sorry hard mounting and soft mounting. In soft mounting you do it a few number of times and then give up. In hard mounting you keep on doing it till you get it now. So, many thing happens in the case of SCSI devices also. In a SCSI device you, if they are sitting on a bus and some device driver tries to access a particular disk or a particular offset.

Now, that can be any anything that can multiple thing that can happen. It because let us assume that is this are on a electrical bus. Now electrical buses can mole function. If the mole function then you also again have to retry till somebody has done what is called a bus reset. So, do a bus reset then whatever electrical malfunction is there. That might be taken care of and again it can you can again retry and probably it will go through. So, every single layer incorporate some aspect of error management same thing on security.

So, if you are looking at let us say the security of data then the file system will probably handle it one way. The volume manager will do it in probably 3 different ways depending on the context and so, for example, a file system might encrypt it and it will do its own key management. Volume manager might decide that so the file system is doing it. I do not have to worry about it. That is not my business or nowadays you can get disks with encryption possible. C gate for example, sells some disks in which the whole disk can be encrypted ok.

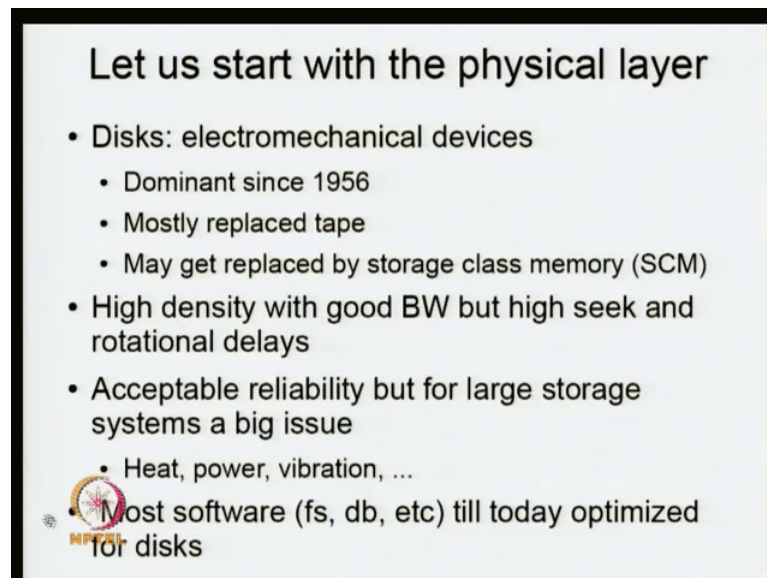
So, this is not at the file system level not at the volume manager level it is much lower at the device level itself that also is possible again performance also has to be managed at each of these layers in its own ways for example, some layers will be caching some layers have to do flow control for example, if I am doing NFS then usually NFS uses UDP and so, I might have to actually manage how I pump the network with UDP packets. That is some part of at something that NFS client has to manage ok.

Similarly, consistency management we will discuss this in some detail. Each party might have to worry about certain aspects relating to consistency for example, volume manager as I mentioned, handles multiple disks. So, from if a particular disk dies then it has to update its data about what disks are available and it has to have a consistent model of what disks are available, what disks are suffering, transient errors that are actually etcetera ok.

Similarly, consistency management of network file system you have to ensure that a multiple guys are writing to a file. Do you guarantee that everybody sees the same? Let us say view of the file or or do you leave it to NFS which basically says I do not guarantee those guys except at the consistency except at 30 second intervals or something else something similar; and if you are talking about web scaled systems like S3 you have to worry about consistency as I mentioned earlier. You worry about things like eventual consistency models.

So, every layer also has to do its own way ok.

(Refer Slide Time: 40:52)



Let us start with the physical layer

- Disks: electromechanical devices
 - Dominant since 1956
 - Mostly replaced tape
 - May get replaced by storage class memory (SCM)
- High density with good BW but high seek and rotational delays
- Acceptable reliability but for large storage systems a big issue
 - Heat, power, vibration, ...
- Most software (fs, db, etc) till today optimized for disks

So, what we will do right now is to quickly look at I would like to go through the whole of the stack up and down as a part of this course. But what we will do is we will start with the simplest possible time we will look at the disks I say the simplest, but actually it is not a simple thing at all it is actually a very complicated device and these are been

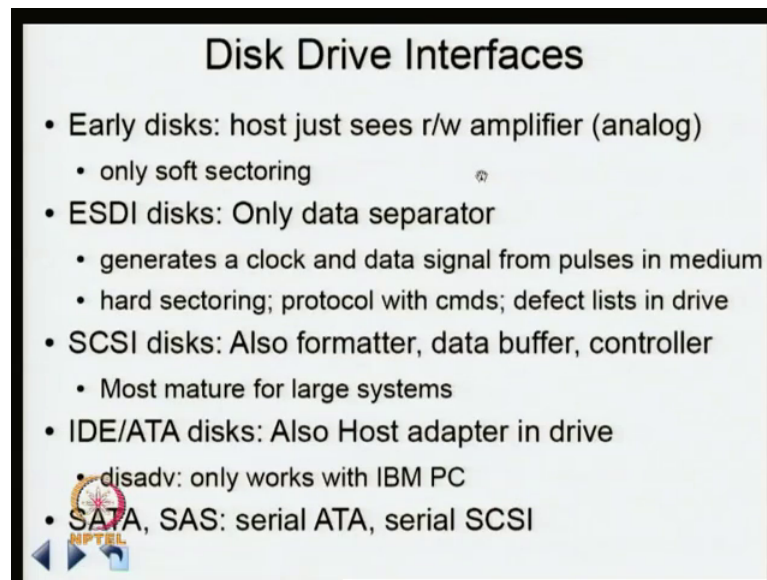
dominant since 1956 and they essentially replace tape. Before 1956, tape was the primary medium for storing things.

But nowadays tape is a some kind of it is used only in special archival kind of situations and disk is quite dominant even today, but it is possible that in the next 20 years or so, it can get replaced by something called storage class memories flash being one recent example of it. So, disks are still let us say not optional. We still have to go through using this, but I think some date might get replaced. What is good about disk is that it is got high density with good bandwidth the biggest problems are that it has got high seek and rotational delays ok.

It has got acceptable reliability, but for large storage systems it is a big issue. That is why you will notice that large scale file systems like Google file system etcetera they worry about this and their design takes care of liability as an important aspect. Some of the reasons why worry about them is because disk can let us say they consume power, they produce lot of heat, they vibrate and because of vibration it may turn out that you might instead of reading a particular track on the disk you might read something else that also is possible. So, a disk is really on the if you think about the whole thing it is quite a good device, that is why it has survived for 5 decades or more. It is still survival we still go on or laptops most of them still come with disk. It also turns out that in most software as of now, all assume that disk are sitting out there.

For example, file system with databases we are all optimize thinking that there are disruptive. Of course, this has to change once we move to storage class memories this is going to be a big major change as things happened.

(Refer Slide Time: 43:24)



Disk Drive Interfaces

- Early disks: host just sees r/w amplifier (analog)
 - only soft sectoring
- ESDI disks: Only data separator
 - generates a clock and data signal from pulses in medium
 - hard sectoring; protocol with cmds; defect lists in drive
- SCSI disks: Also formatter, data buffer, controller
 - Most mature for large systems
- IDE/ATA disks: Also Host adapter in drive
 - disadv: only works with IBM PC
- SATA, SAS: serial ATA, serial SCSI

I can next to give a high level idea about disks. You will see that disks have changed over period of time. In the beginning they prove they gave very little functionality or you saw was a read write amplifier analog it had only what is called soft sectoring; that means, there was nothing on the disk itself from hardware point of view which could say that something it use a sector. A sector being unit of transfer it has to be done by software only.

Now, once we have hard sectoring then you can possibly do certain things independent the disk can do something independently. It does not have depend the software doing it and it may be that hardware is going to can be more effective at this low levels things, low level operations if you do it at a high level from the CPU side it is a lot of work. So, in a sense, if you have a hard sectoring, lot of the housekeeping jobs can be done by the device itself does not have to go to all the way it has does not go to travel all the way up to the CPUs to or the software to handle it.

Similarly, there is a second generation of disk called ESDI disk scheming. It provided hard sectoring. It provided what is called defect list and drive what it means is that if you tried writing to some place and it was proved to be difficult for whatever reasons then it will say that that particular sector is defective and it will keep note of it on the disk itself. So, that it will not try to write to that one. It will actually vector to the some other area, specially kept for handling defects of this kind ok.

So, so there are various things that the next generation did if you look at SCSI disk. They provide lot more additional functionality. They provide some data buffers some more sophisticated controllers and these are quite suitable for large scale systems and these are currently the best disk that you can get and in all high installations. We will see on this SCSI disk now it is possible for you for us to use SCSI disk, but it turns out that you need a agent as I mentioned HBAs on the a CPU side. So, you need a HBA on the CPU side and then you can access SCSI disk.

Now, for local systems like IBM pc etcetera, it turned out having HBA was a cost and a costly affair. So, they integrated the HBA with this disk and they produce what is called IDE or a ATA kind of disks. So, basically the host adapter is in the drive itself and basically, it is a electrical interface is directly attached to the mother board of the system. So, from our certain costs point of view because it is most electronics the more you do large scale integration the less explains you have to suffer and so, IDE, ATA disks are basically similar to SCSI disk except that the host bus adapter is sitting on a motherboard itself directly. The only problem is that it works only with IBM pc and; that means, that whatever these SCSI are doing assumes that some IBM pc sitting out there.

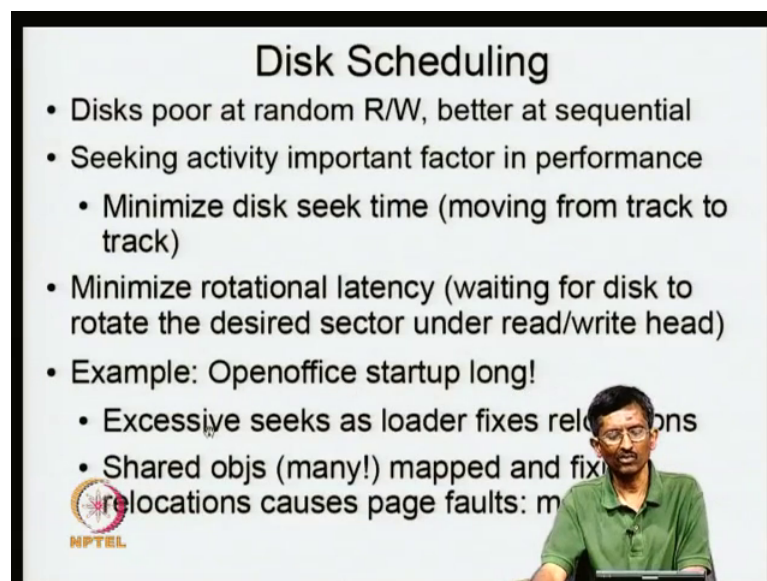
That means that it assumes that are only 2 disks out there possible. Maximum the bus can support is 2 disks or 4 disks or whatever it is and so, everything is geared with those kind of assumptions. Now this is not good. If I am talking about a high end system where I can have something like as many as 30 or 40 disks and making this assumption, there are only 2 disc like IDE or ATA disk. A ATA buses assume it is not a good thing ok.

So, these are suitable for larger systems, but you have that expense of a host bus adapter. Here the host bus adapters are integrated, but they can only be used in limited situations and this IDE ATA has changed into SATA and devices again. The difference being that this is mostly to electrical reasons. It turns out that this disk started with 8 bit interfaces, electrical interfaces then it becomes 16 bit interfaces then it become 32 bit interfaces. Now it turns out that if you keep going to larger number of bits to be sent at the same time there are some issues called clock skew. The signals do not propagate at high when that a higher megahertz. At the same way, there is some clock skew which create problems.

So, what people do is they provide what is called a serial motions of disk; that means, that the instead of providing 32 bits parallel which and clocking them is somewhat difficult makes it little more intricate that the electrical design becomes more difficult. You can provide what is called a serial type of thing at much higher frequencies. So, for example, we can have a currently you get 3 gigabit per second SATA devices; that means, you send only 1 bit at a time, but at 3 gigabit per second those busses can handle those. So, the bus can handle 3 gigabit per second, but the devices might be able to pump it at something like 600 megabit per second this is serial ok.

So, similarly for SCSI you have something called SAS serial SCSI and a same problems. I cannot electrically draw a 32 bits with our clock skew because that is difficult thing. I do a SAS. This is just a basic introduction I, I am pretty sure there is lot more detail to be understood. If you want to study, but this is some high level idea.

(Refer Slide Time: 49:12)



Disk Scheduling

- Disks poor at random R/W, better at sequential
- Seeking activity important factor in performance
 - Minimize disk seek time (moving from track to track)
- Minimize rotational latency (waiting for disk to rotate the desired sector under read/write head)
- Example: Openoffice startup long!
 - Excessive seeks as loader fixes relocations
 - Shared objs (many!) mapped and fixed relocations causes page faults: m

NPTEL

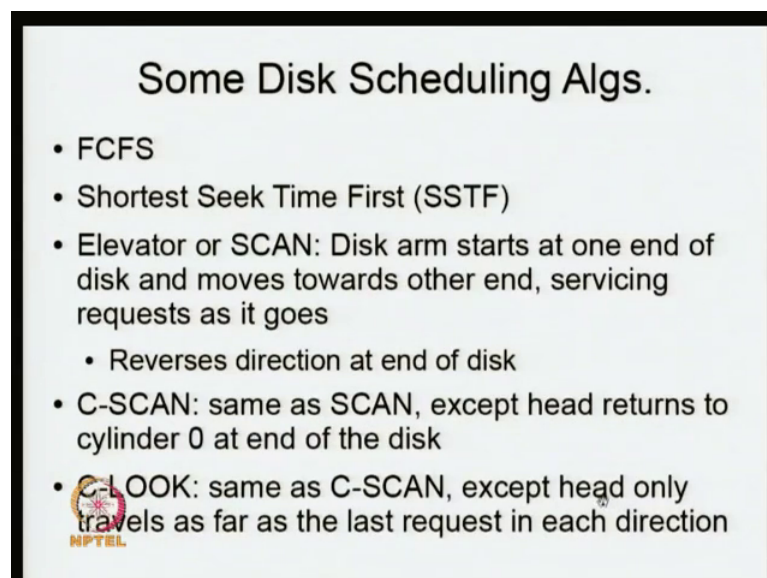
Now, let us just look at some aspects of the disk the disks are poor at random read write, but better at sequential and seeking activity important factor in performance.

So, I have to minimize this. We kept basic reason is that each seek takes about 2 to 10 milliseconds and that is the quite a long time and so, I need to figure out ways of reducing the seek time and that is why as I mentioned a lot of software is designed to make sure this does not this minimizes as far as possible.

Similarly, you want to minimize rotational latency waiting for the disk to rotate the desired sector under read, write head and you can see this effects of this seek time and rotational latency in many applications. You might use for example, open office many people who have used it will might have noticed that the startup takes too long and basically, the reason why that happens is because you need lot of excessive seeks and this excessive seeks. It is about as many as 100 of them that basically what causes open office to becomes quite slow.

The exact reasons are little more complicated which I am not complicate into.

(Refer Slide Time: 50:30)



Some Disk Scheduling Algs.

- FCFS
- Shortest Seek Time First (SSTF)
- Elevator or SCAN: Disk arm starts at one end of disk and moves towards other end, servicing requests as it goes
 - Reverses direction at end of disk
- C-SCAN: same as SCAN, except head returns to cylinder 0 at end of the disk
- C-LOOK: same as C-SCAN, except head only travels as far as the last request in each direction

NPTEL

So, again we need to look at the kinds of given that you have a disk you may want to think about how to schedule operations on it. So that we can do better than what is happening. Right now for example, you can have what is called first come first serve. That is as a request coming, we do it in the same order. This is not good because that could be on one part of the disk and then if I get an access on the other side of a disk or in the center of the disk that is from one enter the disk to other end in the center of the disk. For example, then I will be seeking a lot.


So, that is why this first come first serve could be a problem. There could be some other types like called shortest seek time first because the seek is so important then why not try to find the seek that is closest next. So, for example, somewhere I am at a particular point in the disk. I find the seek which is to be service which is closest to the ram and this also

has been attempted, but the one which has been widely used is what is called elevator or scan. It is similar to the where we use elevators. So, disk arm starts at one end of the disk and moves towards other end just like elevator which start from the one end of the thing all the way goes all the way goes up and service and request as it goes and reverses direction at the end of the disk.

This turns out to be a fairly good model similar thing is c scan. Same as this scan model except head returns to cylinder zero. In a it is like an elevator situation. You go all the way out and elevator you drop all the way down to the lowest flow and this is it. Might seems very strange why people are doing it, but it is gives you some let us say, a more real time guarantees its better at handling. The latency is seen by is not that high for some more basically variance is not very high with this c look is similar to c scan except that head only travels as for as the last request in each direction. There is it do not go all the way to the under the disk. You only go as much as what is available requests are there.

(Refer Slide Time: 52:35)

Linux Disk Scheduling

- (Linus) Elevator (default till '03)
- Deadline
 - Imposes a deadline on all I/O operations to prevent resource starvation.
- Anticipatory (default '04 - '06; now removed)
 - pauses for a short time (a few ms) after a read operation in anticipation of other close-by read reqs
- Completely Fair Queuing (CFQ) (default from '06)
 -  allocates timeslices for each of the per-process queues (synch/asynch) for access to the disk
- Null

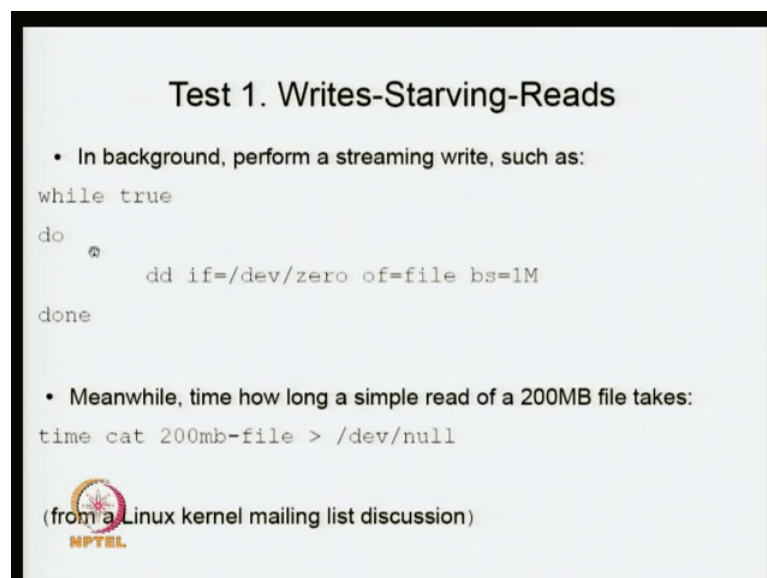
Again we just quickly look at the disk scheduling again. These are the once what I said earlier was some other classical disk scheduling algorithms and in the recent pass again there should be a lot of work and for example, we have the elevator algorithm which Linus implemented one time back which was default till 2003. There is also a deadline version imposes a deadline all IO operations to prevent resource starvation. We can have

what is called anticipatory which was used quite a bit in Linux systems, but now it is no longer the default. Actually it has been removed. This one, what it does is the idea is that if you are doing a disk operation.

It is very likely that you will be doing another operation nearby. So, instead of trying to do some other a unrelated access, you see if you one more access or some nearby I think comes in so, that we can get better. So, what is the anticipatory? This turned out to be quite interesting once upon a time, but people are finding that it is not as good in many other situations. So, there is something called completely fair queuing has come in and basically, what you do is, you allocate time slices for each of the per process queues for both asynchronous requests and asynchronous synchronous and asynchronous requests for access to the disks.

So basically, there is a key for there are multiple queues for there are per process queues and they are maintained for both synchronous requests as well as asynchronous request and you allocate time slices for them and then based on that, you do the scheduling. There is also something called null; that means, you do not do any a modification of the request. As a come in, they order you do it in the there where (Refer Time: 54:25) model as a first come first serve.

(Refer Slide Time: 54:26)



Test 1. Writes-Starving-Reads


- In background, perform a streaming write, such as:

```
while true
do
    dd if=/dev/zero of=file bs=1M
done
```

- Meanwhile, time how long a simple read of a 200MB file takes:

```
time cat 200mb-file > /dev/null
```

(from a Linux kernel mailing list discussion)

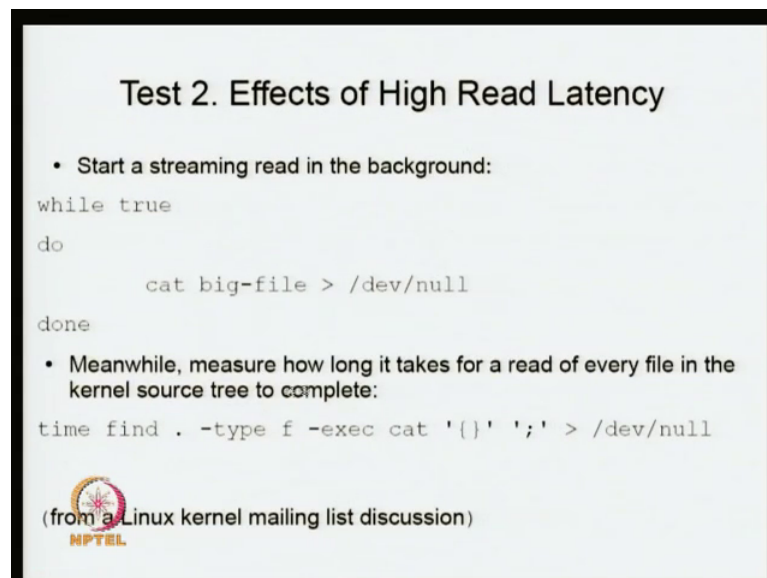


So, I will just briefly look at how critical it is to do these things well. Suppose you have a streaming write; that means, you are doing very very long big writes basically, keep on

writing it. What is this guy is doing is he is writing zeroes to a file with a block size of 1 megabyte.

So, this is the background thing and then you do a foreground read of a large file. So, basically, you are the 200 megabyte file, you are cating it to null dev slash dev null. So, basically I want to find out how much it takes this is one example.

(Refer Slide Time: 55:04)



Test 2. Effects of High Read Latency

- Start a streaming read in the background:

```
while true  
do  
    cat big-file > /dev/null  
done
```
- Meanwhile, measure how long it takes for a read of every file in the kernel source tree to complete:

```
time find . -type f -exec cat '{}' ';' > /dev/null
```


(from a Linux kernel mailing list discussion)
NPTEL

The similar thing about a streaming read in the background and you are trying to traverse a deep directory and cat every file read that is list every file in it. Now I just want to show you how differently the performance of a based on the different types of scheduling that can be attempted.

(Refer Slide Time: 55:25)

Performance Results		
I/O Scheduler and Kernel	Test 1	Test 2
Linux Elevator on 2.4	45.0 secs	30 mins, 28 secs
Deadline I/O Scheduler on 2.6	40.0 secs	3 mins, 30 secs
Anticipatory I/O Scheduler on 2.6	4.6 secs	15 secs

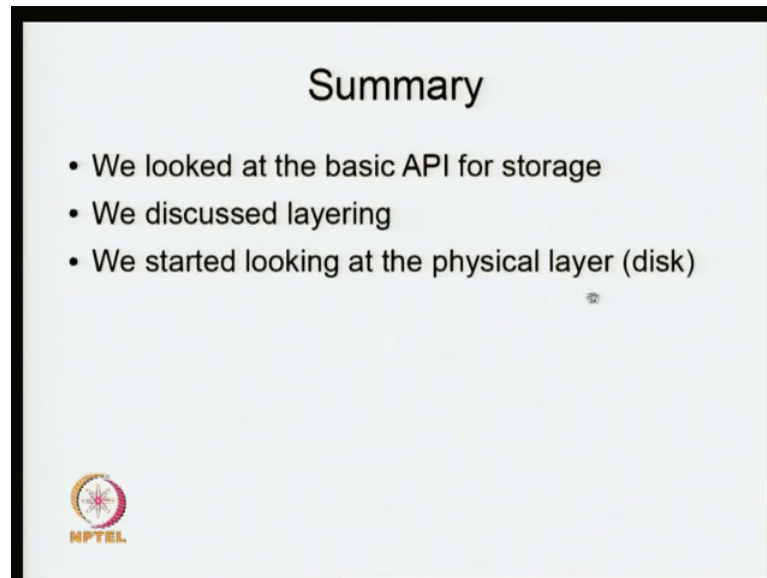
(from a Linux kernel mailing list discussion)



So, if you look at it for example, the results can be quite dramatically different. For example, in the case of the elevator algorithm if we take 45 seconds anticipatory IO scheduler takes about one-tenth of it. In the case of the previous other one, basically, this looking into this high read latency because you are basically having streaming read right we can see the variety of results that you get the difference being as much as 30 minutes to 15 seconds ok.

.So, it is very important that you your scheduling is write because if you do not get this write you can see a shoe difference in performance and that is why there is been a lot of, a lot of software tries to figure out for a particular application. How best to use the disk. So, that is a lot of work is done in this area.


(Refer Slide Time: 56:21)



The slide is titled "Summary" and contains three bullet points. The NPTEL logo is located in the bottom left corner of the slide.

Summary

- We looked at the basic API for storage
- We discussed layering
- We started looking at the physical layer (disk)



I think we like to stop here at this point. So, let us we basically mention that we looked at the basic API for storage. We discussed briefly some aspect of layering and we started looking at the physical layer. I will continue in the next class about the physical layer aspect of it a bit more and then you look at SCSI disks a bit and then afterwards we will proceed on to some other layers.

Thank you.