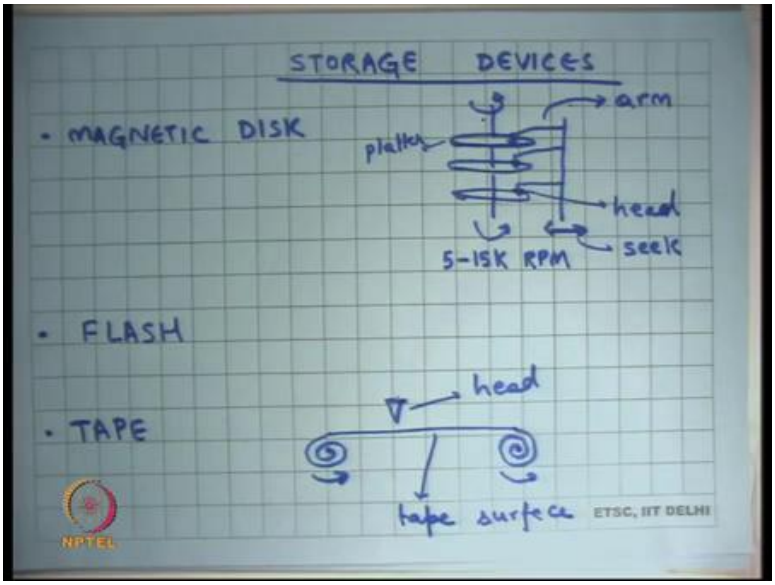
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 31
Storage Devices, Filesystem Interfaces

Today we are going to look in more detail at exactly how a file is implemented using the current set of storage devices, that we have and then what is the; what is a file system interface in general right.

(Refer Slide Time: 00:41)



So, let us look at the storage devices, you know predominantly there are three types of storage devices. In fact, you know the magnetic disk is by far the most popular storage device and we have already seen it. It basically has platters on a spindle, on a spindle and the spindle rotates at some speed typically 5 to 15,000 RPM.

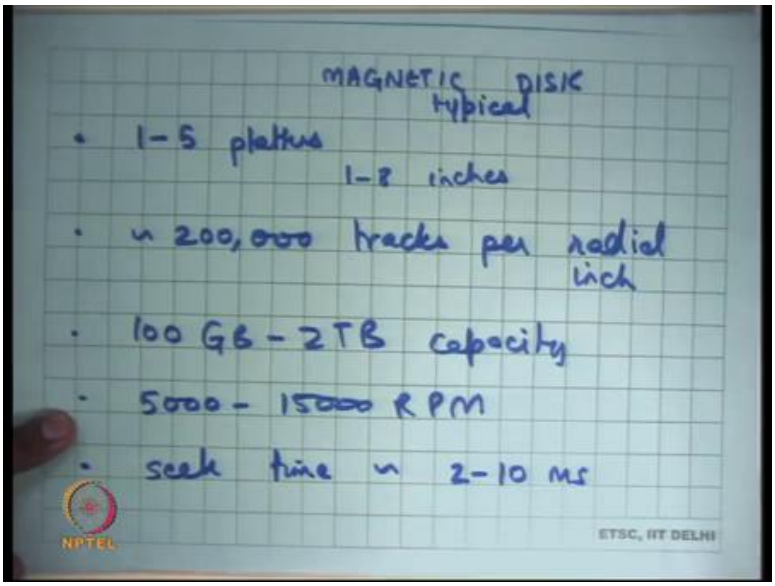
There is a there is an actuator arm which has multiple heads that are reading this data of the spindle of the platters. So, these are called platters, each surface is called a platter. And this these disk arms move readily alright while I mean it is possible to have separate discounts for different platters, you know in practice mostly you will have a single disk arm and multiple heads attached to the same disk arm.

So, all the heads move together in one direction or the other right. And, we also discuss some things about you know how long it takes to seek and in general what is the; what is the throughput of such a system, and we are going to discuss it in more detail.

There is another technology which is relatively more recent, it is called flash, which is available on phones, pen drives etcetera. It does not have any moving parts; it is basically made of solid state which means semiconductor except that its nonvolatile. So, unlike your DRAM which is your main memory this one is nonvolatile which means its state persist across power reboots right.

And, then there is this older storage device that was extremely popular in the earlier days called tape. And, this tape is you know if you seen a cassette player its similar, basically the tape moves. So, you wrap around the tape in the spindles and the spindles rotate in the same direction and those of the tape sort of moves like this and there is a head that reads off the tape ok. So, let us look at these the characteristics of these devices in more detail before we talk about file systems.

(Refer Slide Time: 02:51)

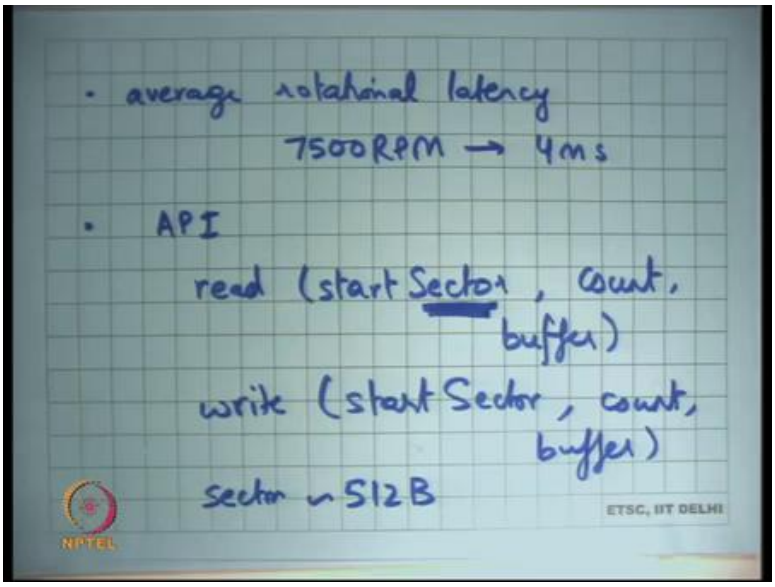


So, magnetic disk well typically 1 to 5 platters alright. So, typical magnetic disk let us say 1 to 5 platters roughly let us say 1 to 8 inches in height right. So, if you have ever seen a hard disk you will basically see that its a package which has you know which is roughly this thick 1 to 8 inches and you have 1 to 5 platters in it; roughly 200,000 tracks per radial inch right.

So, that is the density at which the tracks are packed radially, recall that a track is basically circle on the platter and so, they are concentric circles and each circle is called a track right. And so, the typical density today is 200,000 tracks per radial inch.

Today you can buy disk with 100 GB to 2 TB storage right, capacity right. We have already seen the rotational speeds, they are 5000 to 15000 RPM alright, seek time typical is 2 to 10 milliseconds alright and let us see average rotational latency.

(Refer Slide Time: 04:25)



So, the seek time basically means the time it takes for the head to reach the track that you want to reach and the rotational latency means the time it takes within the track to reach the sector that you need right.. So, on average the rotational latency will be the time it takes to make half a revolution right.

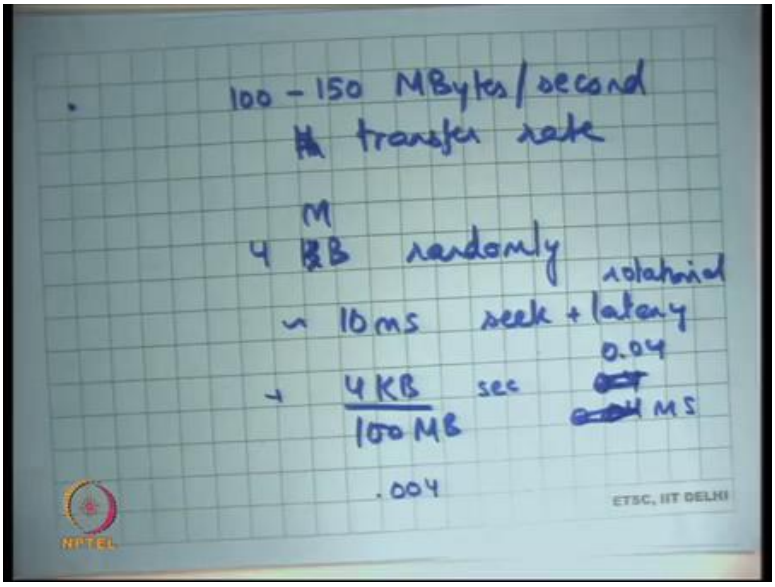
So, with let us say 7500 RPM disk implies roughly 4 milliseconds average rotational latency alright. And, what is the API of a disk or in other words you know how can you access the disk, how the how can the OS access the disk favor to write it in high level?

It basically you can say read, you can specify a start sector, account, and a buffer right. And similarly, you can do write same thing start sector, count, buffer; a sector is basically a unit of storage within a track and typically a sector today is 512 bytes. So, an operating system can ask the disk to read a sector or write a sector and the disk arm; so,

the disk controller will move the arm to the appropriate location and then read data of it alright ok.

Once the arm has been positioned at the sector from where you want to read after that it is just a matter. So, the disk the spindle is constantly rotating right. So, it is just a matter of the head starting to read the data that is stored in magnetic form on the platter. And so, in doing so, with the current speeds you will typically get 100 to 150 megabytes per second of bandwidth transfer rate right.

(Refer Slide Time: 06:33)



So, after you have positioned the head at the right position you know you can now start reading data at roughly 100 to 150 megabytes per second speeds alright. Of course, you know if you let us say so, what that means, is that let us say if I wanted to read 4 kilobytes randomly from you know somewhere I have to actually do seek and latency.

Then I will have roughly 10 milliseconds of seek plus latency where, latency, by latency I mean rotational latency right plus there will be some data transfer time which will be the time it takes for 4 kilobytes to go under the head. And, assuming 100 to 150 megabytes per second 4 kilobytes will probably take 4 KB upon 100 MB seconds right. So, that is.

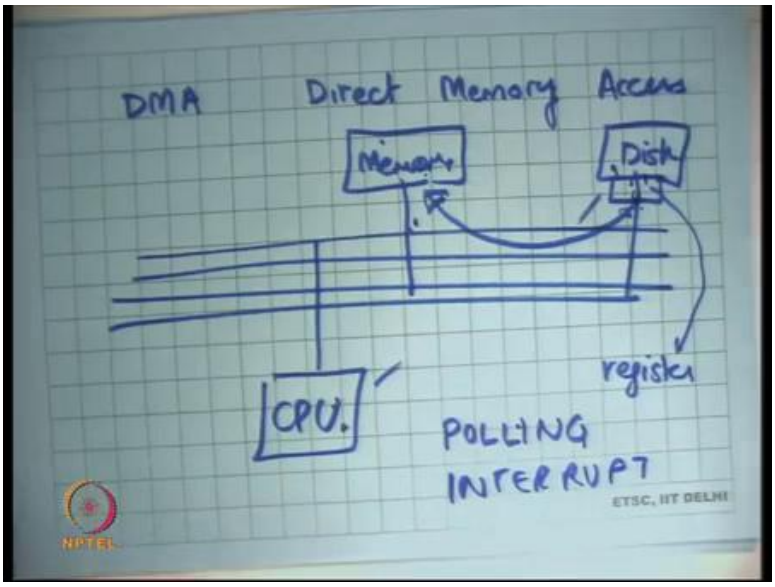
Student: 0.04 milliseconds.

0.04 milliseconds let us see; so, that is a 0.004 0.4 milliseconds right 0.04 milliseconds.

Student: 0.4 milliseconds.

All right its 0.04 milliseconds. So, as you can see its highly dominated by the seek and latency you know a random 4 kilobyte read is highly dominated by the seek and rotational latency. On the other hand, if you read let us say 4 MB of data then the you know then you are basically amortizing the seek and latency over a much larger chunk of data and this time becomes relatively more significant.

(Refer Slide Time: 08:33)



Finally, disks or other devices support what is called DMA or Direct Memory Access alright. What does this mean? Well, how does the CPU initiate transfer? So, direct memory access says that the device can directly access memory without needing the help of CPU to you know do this transfer.

So, let if I just draw the diagram again and let us say this is my bus and this is my CPU, and this is my memory, and this is my disk. Then one way for data transfer is that CPU one by one starts reading data from the disk. So, recall that the disk has some registers here right.

So, the controller will have some registers through which the CPU can program the disk. For example, by writing into the register the CPU can tell the disk I want to read the sector right and then it can read values from those registers to get the answers from the

disk. And, recall that these registers can be accessed either using port mapped space. So, port space like inbe and outbe instruction that we have seen before.

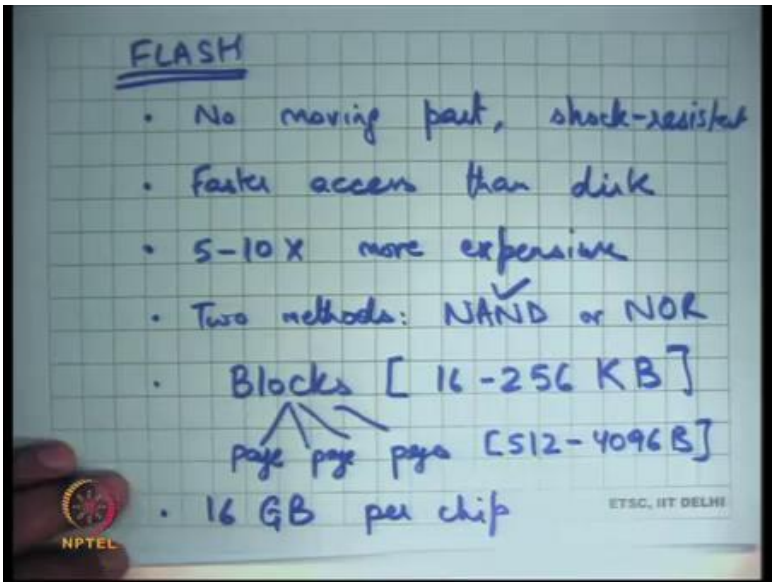
Or, they can be accessed using memory map diode right just load and store on some addresses that are mapped to those registers. So, in either case you are basically accessing the register. So, one way of doing transfer is that the CPU basically keeps reading these registers and then writing keeps writing them to memory.

So, it has a tight loop which basically read from the register and writes to memory, you know that would be a relatively inefficient way of doing it. Most modern disks have controllers that are capable of writing directly to memory.

The way it works is that the CPU registers a request and tells it that this is you know this is the location of the memory address. So, buffer basically points as a physical address that is given to the disk controller. So, and the disk directly writes to memory. So, the CPU just fires a command which says write these many sectors starting at this physical location in memory. And so, the disk controller in can do it without involving the CPU in it and so, that is you know you can imagine that a significant optimization.

And, typically that is how it is done; we have also seen that the CPU when the transfer is finished then the CPU can check whether the transfer is finished either using polling you know periodically keep checking whether the transfer is finished that is polling or interrupts alright. Or, you can configure the disk controller to generate an interrupt when the transfer is finished, and the interrupt handler will proceed accordingly alright ok.

(Refer Slide Time: 11:21)



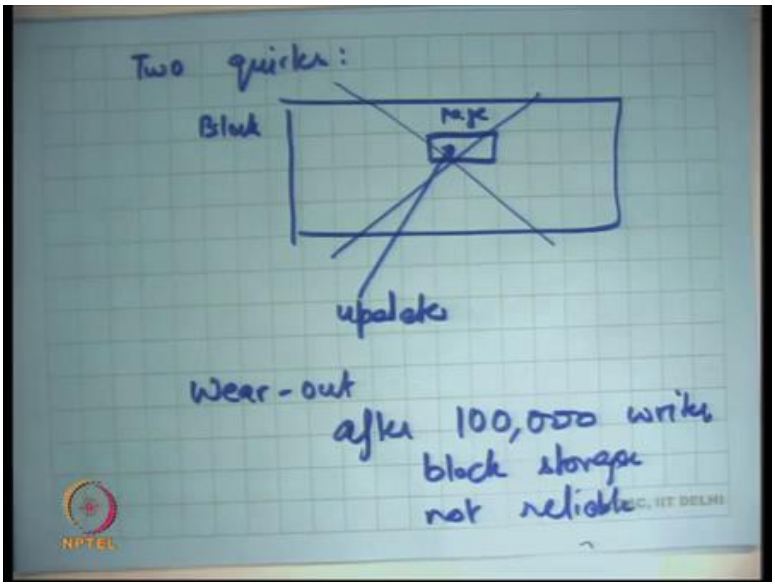
Alright. So, that is the disk, let us briefly look at the more modern technology which is flash and let us look at you know what it has. Firstly, it has no moving parts right. So, that is a significant difference, and which means it is you know shock resistant; you do not have to worry about it you know moving parts breaking down or anything of that sort.

Also, it has faster access than disk right, let us look at you know in what way that fast, but it is also 5 to 10x more expensive today right. So, two methods of implementing flash without going into details let us just say you know, let us just understand that there are two ways NAND or NOR and the NAND one is more popular today.

So, let us see how a flash device is organized, its organized into blocks and blocks contain multiple pages right. So, there are large blocks and the blocks are basically of the size of 16 to 256 kilobytes right and pages are 512 to 4096 bytes.

So, the page is basically is equivalent of a sector on a magnetic disk roughly speaking its between 512 to 4096 bytes. But multiple pages are within a block and a block is 16 to 256 kilobytes on today's flash. And, total capacity today that you can get for a flash device is around 16 gigabytes per chip alright.

(Refer Slide Time: 13:27)



So, there are two significant you know quirks about flash storage which were not there in you know magnetic disks. Firstly, if there is a block let us say this is a block and it has a page in the middle.

So, let us say this is page, this needs updation; updating a page within a block requires erasing the entire block before rewriting the entire block again right. So, just a matter of how its implemented at the physical level you basically need to erase the entire block and then rewrite the entire block to update some random location inside the block right.

So, that is that basically means that if you randomly want to write something on the flash a small write then it is expensive, and the second thing is were out. You can only write to a block let us say around after 100,000 times. So, after 100,000 writes block storage not reliable right. So, after you return to a block more than a 100,000 times you know the disk is no not reliable anymore. So, you know it has a limited self-life, these problems were not with the magnetic disk right.

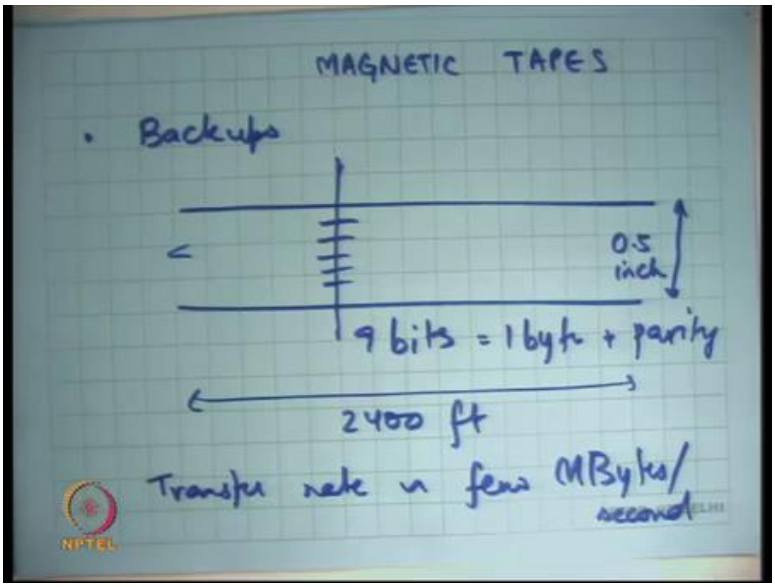
So, there is you know there is a huge amount of research and industry work going on to figure out if flash can be used as a replacement to magnetic disks for all the applications that. And, basically it involves solving these prob you know dealing with these problems in an efficient way.

So, for example, for wear out what you would want to do is you want to build some support either at the operating system level or at the hardware level to ensure that all blocks were out at roughly the same time.

In other words, there should not be any hotspots in your flash storage. So, you should not there should not be a blog that is constantly being written to right because then that. So, the disk is only the flash storage is only as good as last only as long as the block that first goes wrong right. And so, that is one optimization and the other optimization that people work on is basically that you know avoid random updates.

So, organize your file system, structures or your disk drivers in such a way that random updates are avoided ok. And so, you know there are so many ideas there that that can be discussed, but of course, that is not the; that is not the topic of our discussion today.

(Refer Slide Time: 06:15)



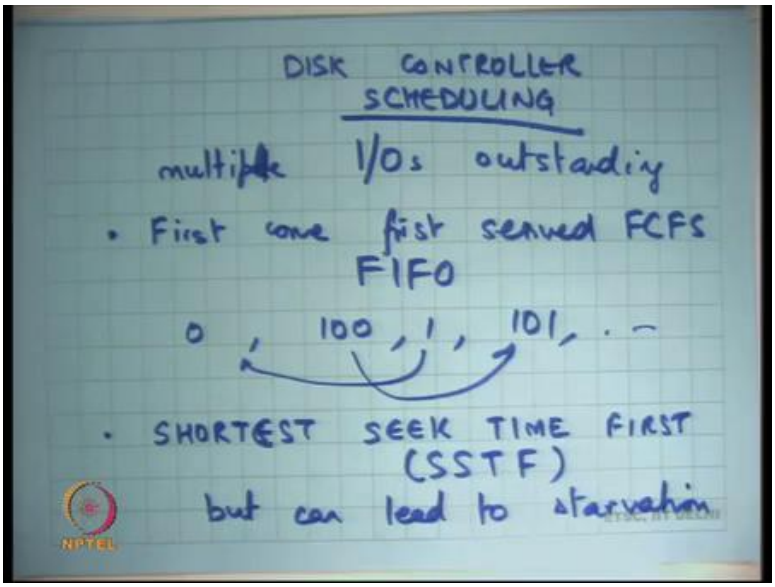
Let us also look at magnetic tapes right. So, magnetic tapes were used a lot for backups, till maybe early 90s or even late 90s ok, but not so much today. Today people use disks for backups instead of using tapes and the reason people use magnetic tapes earlier was basically tapes were much cheaper at that time than disks alright. So, let us see how a tape works. So, let us say this is a tape then it has you know 9 bits.

So, of storage in one cross section; so, its roughly 0.5-inch half a inch wide alright. And, the length can be up to 2 you know large feet 2400 feet let us say ok. And, each cross

section of a tape will have 9 bits which means 1 byte plus parity that is how it typically its used. So, for error detection you basically have a parity bit.

And typical bandwidth transfer rate is roughly few megabytes per second. So, as you can see that today's disks are far much faster than tapes and disks are become in those days disk used to be much more much more expensive. And so, backups which involve lots of storage were done on tapes and so, you would find you know dip lots of tapes in the backup room or in the server room. But today the same thing is instead done using disks alright.

(Refer Slide Time: 18:09)



Finally, let us look at how let us look at the disk controller in a little more detail.

Student: Sir.

And, in particular I am going to look at the scheduling algorithm that goes on within the disk controller. So, in case of a magnetic tape, if you want to access some random byte then you have to actually go through the entire tape to actually restart byte; assuming you have basically have variable sized records in your thing right.

So, that is the; that is the disadvantage; so, that is why it was primarily used for backups and not for you know online transaction processing workloads, alright. So, let us look at the disk controller. So, basically the disk controller has to decide assuming it has multiple IOs outstanding, it has to decide which IO to serve first right.

So, by IO I mean a read or a write request let us decide which IO to service first. And, it seems intuitive that the more flexibility the of the disk the disk controller has which means that the more outstanding requests there are, the better job it can do what scheduling it right.

So, for example, one option is just First Come First Served right, or you know also called FCFS or also called FIFO First in First Out, right same thing. So, this is fair whoever comes first gets there, but what but it completely ignores proximity. So, let us say there were alternate requests one for track 0 and another for track 100 and then there is a 1 then 101 and so on.

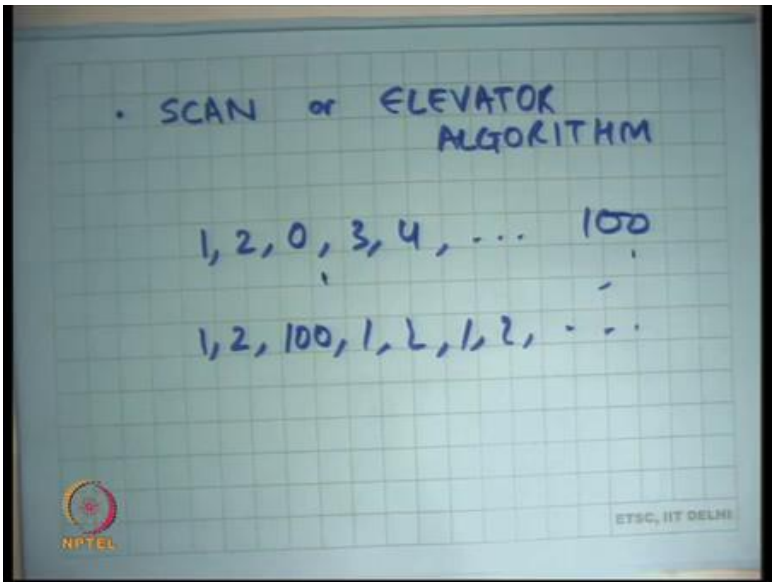
Then you are doing a lot of seeks right, you are spending a lot of time doing seeks; you the better thing would have been to do 0 and 1 together and 100 and 101 together right. So, that is basically a better thing to do FIFO is fair, but it completely ignores location. The other thing to do is Shortest Seek Time First right or also called SSTF that is it.

So, here the idea is let us say I have n requests in my queue, I will compute which of them is my is closest to me is closest to the current location of the head and that is the one that I am going to serve. So, you can imagine that if there are lots of requests outstanding in the queue then you know you can get; you can get significant optimizations there.

So, this is good, but can lead to starvation right what can happen is that you lots of requests are coming for 0 1 0 1 and then there is an outstanding request for 100 which will never get served because the shortlist is basically always between 0 and 1. This is the part of the disk controller so, which means on the hard disk.

So, this scheduling algorithm is within the disk right. So, I am currently discussing you know how the disk controller is optimizing things within that package of the hard disk right. So, there is some logic within the hard disk that is going on to do all this ok.

(Refer Slide Time: 21:57)



And so, and the third one which is what is used is called the scan algorithm or elevator algorithm. And, it takes its name from exactly how typically an elevator works which is that you first you choose a direction. So, the two directions right when you are moving readily either you will move inwards, or you will move outwards. So, you choose a direction in FIFO order.

So, whoever came first you choose that direction just like in an elevator, but if you have chosen a direction you will service all requests in that direction before changing the direction ok. So, it is very much like an elevator right. So, let us say somebody on the top floor presses a button and after that somebody in the middle floor presses a button then you know it will serve all everybody all the way to the top floor.

In fact, let us say there was somebody on top + 1 who presses the button with by the time it reaches top then it will also serve top + 1 before it changes direction right so, that is the idea. And so, that takes care of both efficiency and is also starvation free right. So, if you have chosen a direction after that you are going to go through and serve all those people in that direction. Of course, you are going to come you are going to change direction in some time and that is when you are going to. So, there is no request that can get starved in this case alright.

So, one way to think about the elevator algorithm is you choose the direction in FIFO order and within a direction you use shortest you use SSTF right. So, within a direction

you will choose whoever came comes on the way first right you want. So, with if you have chosen a direction whoever comes first you are going to give him. So, that is now within a direction is it SSTF and choosing a direction is FIFO.

Student: Sir.

Yes.

Student: Sir, would it be better to just keep continuing in one direction itself, since it is a best, we could eventually come down to 0 once again.

I am talking about the radial movement here.

Student: Ok.

Right, I am talking about the radial moment of course, the rotation just keeps moving, I am talking about the radial movement.

Student: Sir.

Yes.

Student: Even in this case starvation can happen, even in this case starvation can happen.

Student: So, these suppose a request comes from 1.

Ok.

Student: Then request comes for 2, then a request comes for 0 and then 3 4 5 6 7.

Let me write it. So, 1 2.

Student: 1 2 then 0.

1 2 0.

Student: 0 then 3 4.

3 4.

Student: So, on up to 100 maybe.

100. So, after 100 it is going to come back to 0 right.

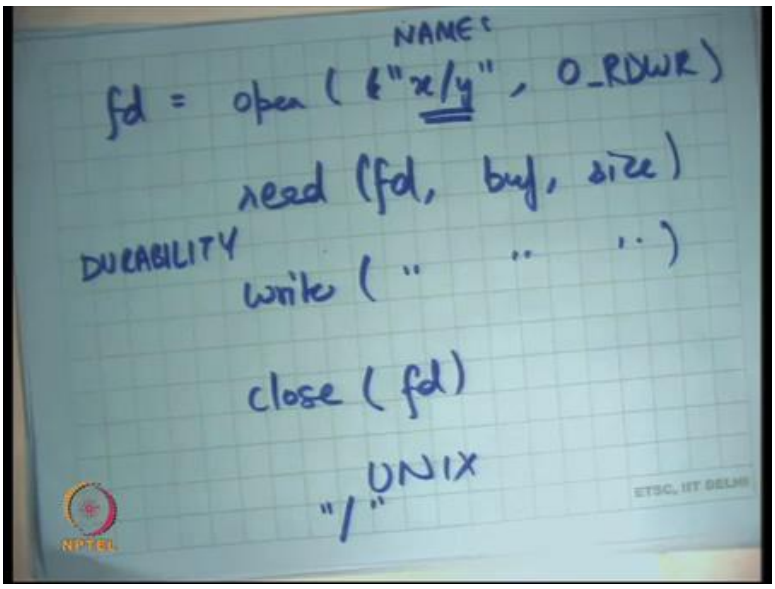
Student: 0.

I mean its it cannot go on for forever basically right; in the case of SSTF it could go on forever for example, I had 1 2 100 1 2 1 2 and so on right. So, 100 gets stuck forever basically, but the here there is a bounded time within which it is going to turn back; it by starvation I mean that it is never you know there is a possibility that there is a non-zero probability that it will never get served. There is an unbounded time, you cannot say when alright so good.

So, those are the storage devices, but by far you know in the last 20, 30 years the magnetic disk has been the primary way of doing storage, secondary storage. And, for that reason most of our operating system instructions have been designed and optimized for magnetic disk right. Tomorrow if you know with some inventions in physics you get some better storage, it also means that many layers above this in software and then hardware needs to be changed to better deal with these new devices.

So, let us now completely think about magnetic disk from now on and let us look at how what kind of abstractions that the operating system provides us.

(Refer Slide Time: 26:01)



So, recall that we have open some file name which is let us say a string x/y and some more which says you know read write right, that gives me a file descriptor. Then you

have read on file descriptor and some buffer and some size and you have write same thing and you can have close right.

So, open read write close is the interface that we are very much familiar with. This interface you know let us let us say this was you know Unix. So, this interface treats a file as a stream of bytes starting from 0 till you know some maximum length and that is it. So, there is no structure on the file so, it is completely unstructured.

On the other hand, you know the other option could have been that files are some have some structure. So, for example, you could have said my operating system provides files which are a sequence of records where each record is of size you know 64 bytes. And, the first field of that record is a natural number, just some 4-bit 4-byte number ok.

So, databases do this right; so, databases represent data as some structured data is to the some structure it is a collection of records and there is a structure on the records. As opposed to that an operating system has decided that we will not use any structure on the files, we will just read the files as a sequence of bytes that is all. If somebody wants to implement a structure, he can build implement a structure on top of this interface right. So, you can always say that this file you know you can always see the sequence of bytes, you can always interpret a structure on the sequence of bytes right.

The disadvantage of doing that, the only disadvantage of doing that is that you have two layers of reaction right. So, you are implementing a files as a in a certain way and then you have implementing a records on top of this virtual stream of bytes right. So, the file system has not so, the disk number of disk accesses and seek time, the notations have not been optimized for the record structure right.

On the other hand, your application may want to optimize it for the record structure and that is the reason that a database will not use a file system to implement its logic right. So, the database tries to bypass the file system the database wants direct access to the disk. And so, and it specifies what is this; what is this structured in variant that you want right. So, the only difference between so, that is the reason so, for performance reasons the database will not use a file system beneath it.

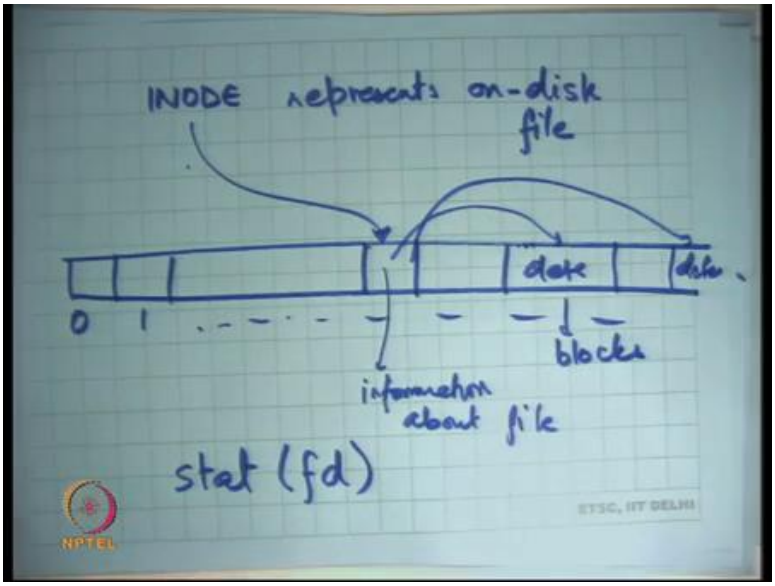
The database will directly access the disk and use structured representation right at the disk block level to get better efficiency. But, in either case you can always you know for

compatibility purposes run a database or a file system and in which case the file will be treated as just the disk basically alright.

So, let us look at this interface once more so firstly, there are names. So, the file system needs to implement naming right. So, on Unix you basically have the special name for it slash which basically means the root directory and then there is a concept of a directory which contains mappings from names to files alright and so, and you can do this in a hierarchical way.

So, the root directory so, there will be a special place on the disk where the root directory stored. And, the root directory will have a mapping from names to files and then those files themselves could be directories and so on right. So, it can be done in a recursive manner alright. What do I; what do I mean by names to file? So, what is the way what is the what is the file?

(Refer Slide Time: 29:49)



Well, a file on a disk typically is separate from its name; so, it is also called an inode represents on disk file, it is a handle to name a file ok. So, let us say this is my disk, I am representing it as logical namespace sector 0, sector 1 and so on right; till some value and a file is represented by an inode. So, inode will point to some location here which will contain information about the file, this is a one disk inode right.

So, Unix refers to a file by its inode and not by its path name, a path name translates the name into its inode and from there on its an inode that is important right. Inode is the on-disk block that contains information about that file. So, for example, when you do fd is equal to open some name and some permissions, it will obtained the mapping from this name to this inode.

And, then store a mapping internally between fd and the corresponding inode number right. And, then when you call read then it is just going to look at that inode and do the operation is based on that. So, there is something called an inode which basically represents represent the file and the user cannot see that inode except you know there is a system call called stat which allows you to. So, let us say stat on fd which allows you to inspect certain fields of the inode for example, what is the size of the inode and what is the status of the inode whether it is read only write read write etcetera and so on ok.

So, you can this structure is completely hidden from the user except that he can inspect it using stack, which is basically to just read write access, it cannot you know directly write to it. Writes to the inode by the user can only happen through this interface which is read write. So, for example, when you write something then you know something needs to be changed on the disk. So, the inode will get over written, but there is no other way to directly touch the inode by the user.

The file itself has you know the data of the file will be stored as disk blocks on the disks right. So, let us say this is data, this is data 2 and so on and the inode should say where this data is. So, it should have some mapping between offset in a file and the data on the disk right. So, there is data are called you know disk blocks and when you say read you basically implicitly there is an offset which basically says this is the offset value.

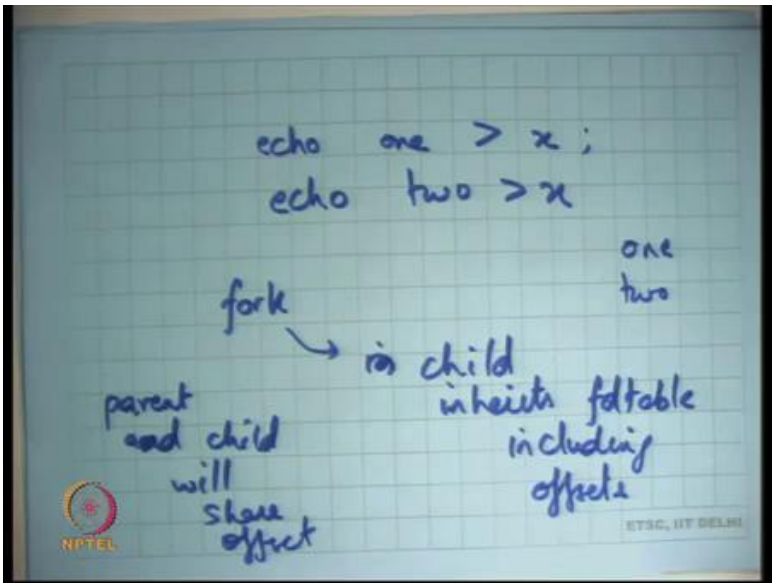
And, so the operating system should convert the offset into the corresponding disk block number using the information from the inode to get that data for you right; so, that is basically the idea. So, recall that in this interface when I say read fd, buf and size no where do I say a given offset right, the offset is implicit. When you open a file, you know depending on what mode you opened it, let us say if you open it in read write mode the offset is initialized to 0.

And, then as you read or write to the file the offset is automatically incremented by the data items that you read or write wrote right. This actually turns out to be an interesting

design decision in at the interface level. The other option could have been that you, you know having a fourth parameter here which says what offset you want to read.

And so, the user explicitly says this is the offset I want to maintain read and so, you know if he wants to read sequentially then it is his responsibility to maintain the offset in its local variable order ok. So, let us look at the tradeoff between having an implicit offset as it is done and Unix and having an explicit offset right. So, if you have an implicit offset then let us say you wanted to implement something like echo one to x and echo two to x right.

(Refer Slide Time: 33:51)



So, basically want to write to file to write one and then you want to write two right. If you had explicit offsets then they would end up overwriting each other right, because each program will not know. So, recall that when you fork you also inherit when you fork child inherits the fd table which includes the offsets right.

So, in this case the parent is going to fork a process which is going to execute the command echo, echo is an executable and its going to write something to the file. When it is going to write something to file because the offset was implicit and was shared between the parent and the child.

So, what will happen is that the parent and child will share the offset right. Because, the parent and the child will share the offset; if the child writes anything to the file the offset

of the parent automatically increases right. And so, you do not end up overwriting each other you end up appending to each other's output.

The output the appending append may have the non-deterministic, but in any case, it's you know it's not overwriting. If you can do some synchronization of this time that you know first you are going to write one, then you by semicolon you basically mean that you going to wait for the first process to exit, before you start on the second process.

Automatically will happen is the first the first child process incremented the offset and so, the parents offset also got implemented and then the second child process saw the increment saw the updated offset. So, you basically see one and two in the file right. So, it is possible to do this, if you had explicit offsets then you know to do this you will have a more complicated interface, will have to explicitly say, will have to you know pass another argument here that at what offset you want to write right. And, that offset needs to be incremented between two consecutive calls to echo right.

In general, so you may say, but you know it sounds it sounds dangerous if I spawn a child and its writing then my offset is getting incremented. And so, and I may not be expecting it, well I mean if you care about different offsets then one option is to just reopen the file right.

So, when you fork you basically copy the file descriptor table which means you have shared offsets, but if you want separate offsets all you need to do is reopen the file. So, implicit of you know if you want to explicit offsets it is easy to do it by just reopening, but the other way around is very hard.

So, its implicit offsets seem to be a better interface. So, it is a good; its a good example of you know how interface design can greatly impact your program structure and program elegance which I should say. So, this file API open read write close actually has turned out to be quite useful, it is not just used for accessing on disk files, it is also used for accessing things like you know pipes.

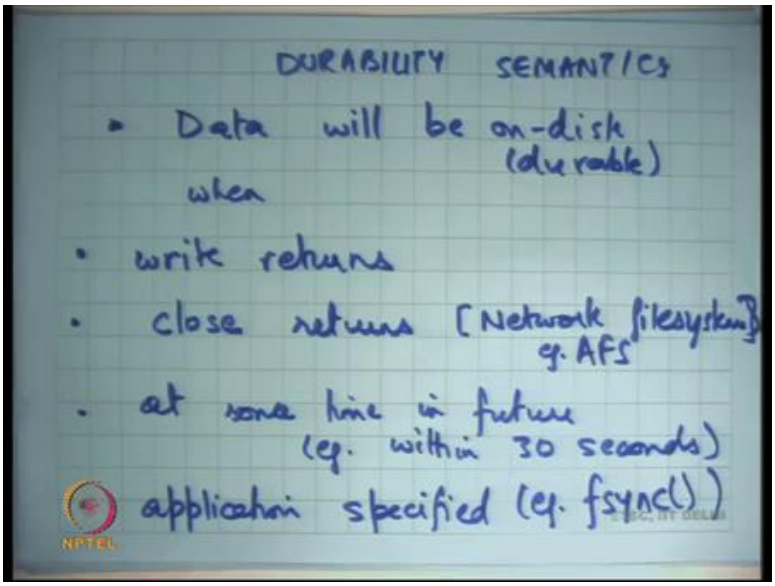
For example, you can call read and write on pipes, we have we have seen that before. It can be used to access devices; I can call read and write on the console device and it basically means the same thing alright or any other device for that matter.

And, also its it can be overloaded to do things like the slash block file system right, that you have seen before which allows you to look at the kernel state right. So, it's a very it's a useful abstraction that took some time to you know get concretized but has been used in Unix later plan 9 and then Linux.

And of course, you know other operating systems today. Now, let us look at; let us look at durability semantics of these calls. By durability I mean if I say right can I be sure that the data is on the disk which means if there was a power failure after I called right after the right you turn.

And, then you know there is a reboot can I be sure that whatever I wrote is going to be present on the disk right. So, these are you know these are the this is part of the semantics that the operating system provides to you and these are flexible. Unix does not specify what semantics you want, it is up to the operating system designer to decide what semantics is one. And, let us look at what are the different semantics that can have that you can have. So, let us say durability semantics right.

(Refer Slide Time: 38:39)



So, one is data. So, by durability I mean data will be on disk or you know durable when one option is when write returns right. So now, operating system may provide a guarantee that whenever the write returns you can be sure that the data is on the disk ok. This is great, its most conservative; basically, saying that I will always you know on

every write system call I am going to access a disk which means your buffer cache is basically write through right.

And so, if you are doing 1 byte writes then you are basically its very slow. So, it is very slow, it is most conservative. So, it is not; it is not; it does not look all that great, the other option is before close returns. So, you can say you know I do not care if there are multiple writes, but if you close the file that is the point I will say that you know definitely all the data for this particular file has been made durable right.

So, that is another option and some network file systems use this alright example: AFS. So, AFS is a network file system that uses. The other option is at some point in future; so, no guarantees exam. So, let us say example within 30 seconds right. So, the write may return, the close may return, but there is no guarantee that the data is actually on disk. The only guarantee is that within some bounded time, let us say 30 seconds your data will be on disk. But if there is a power failure within 30 seconds there is no guarantee right.

So, this has the highest performance, it is a right by cash; the operating systems cache replacement you know Daemon is going to run every let us say 30 seconds and it is going to flush things to disks. But there is no guarantee for the user that you know if you write something then its durable. But this does not seem like a very nice option right because after all you know imagine that you know your operating system was running inside an ATM machine and you wanted to withdraw some money.

And so, it deducts it deducted some money out of your account and it said write in your account. So, it's made the write system call and it return at its also made the close system call you got the money and you went away. And, what you did was you pulled out the power club power plug and so, you know that your update is lost basically.

And so, you plug it back again and you have the money back in your account. So, there has to be some way for the user or some way to for the application to say you know before I before I give this money, I should be 100 percent sure that the data is durable right basically.

And so, if you are doing this then you need some application level interface and this interface is called fsync on Unix. It basically says the application can call the fsync

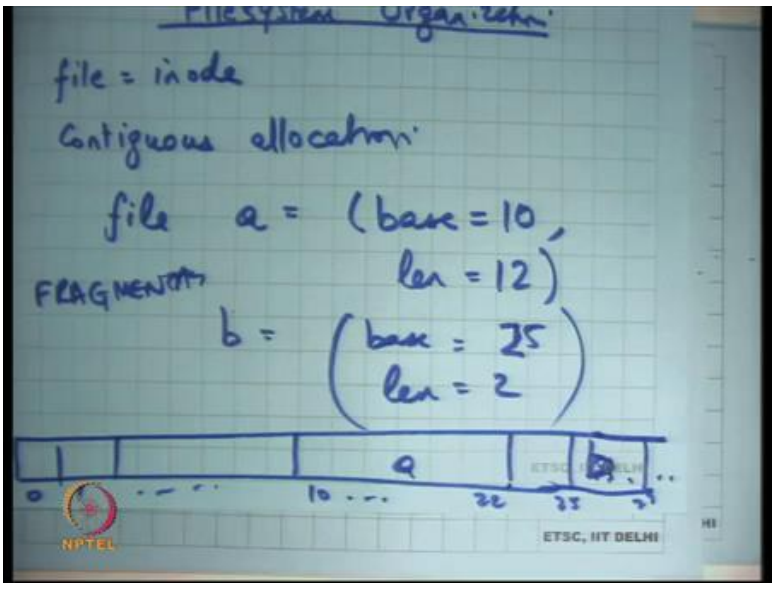
system call; so, this is again a system called on a file descriptor let us say or and then you say you know by the time fsync returns I will be 100 percent sure that the data has been made durable.

And so, for an ATM you will first call fsync, wait for fsync to return before you give out the money alright right. So, these are all sort of options with which the operating system with which the file system can work. And, something like Linux uses this right, at sometime in future with fsync right because of because that has the maximum ok. But, when you are doing something like this you have to be very careful because if there is a crash you should be in sure, that it does not corrupt the file system state itself right.

So, if you are doing some basically you are playing fast and dirty right. You basically saying I want to be as fast as possible, but that you know that should not mean that if there is a power failure at an inopportune moment; then you know your all your data has been erased or your file system has become corrupt.

So, it needs other ways to make sure that the file system remains intact and that is one of the most interesting design challenges of a file system. And, we are going to look at a few different ways of doing that and how modern file systems are doing it ok.

(Refer Slide Time: 43:35)



So, let us look at how a file is; so, file system organization right. So, before we talk about durability, we just got a sense of you know the here is this problem of durability and

semantics that the file system needs to solve. First let us understand how this how should the file system get organized right. So, by organized I mean I said that the file is represented by an inode, but then how should the data blocks be laid out. So, what should be the data structure that you use on the disk to basically store these files right?

So, one option is what is called contiguous allocation right. So, were file a is equal to some base, let us say base block is 10 and length which is you know. So, this here is a file which has which starts at base 10 and has 10 blocks right. And so, and similarly b is some base is equal to let us say 15 and length is equal to 2 whatever right. And so if I draw the disk then you basically have 10 …. 22 is a and this cannot be 15 right.

So, it must be let us say 25 they cannot overlap right and then you have 25 …. 27 which is b right. So, that is contiguous allocation right, this sounds similar to how we did segmentation in virtual memory right. So, basically you know there is a duel between files and processes and memory and disk. So, you basically say that entire disk is one big segment. This entire file is one big segment and you find space for. What is the problem?

Student: Fragmentation.

Fragmentation right.

So, for example, this space is cannot be used for a file which needs 5 blocks or something right. So, one problem is fragmentation. What is the other problem?

Student: File growth.

Growth right, if I want to grow the file a by 10 blocks, I have to copy the entire file to some other location and that is a global operation, that is a very very expensive operation. Because, to you know let us say by file of the gigabytes long and I want just wanted to append a few bytes to it and I cannot find space and I had to actually do you know a gigabyte operation to append a few bytes. So, that is not very good. What is some good things about these contiguous files allocation?

Student: Fast access.

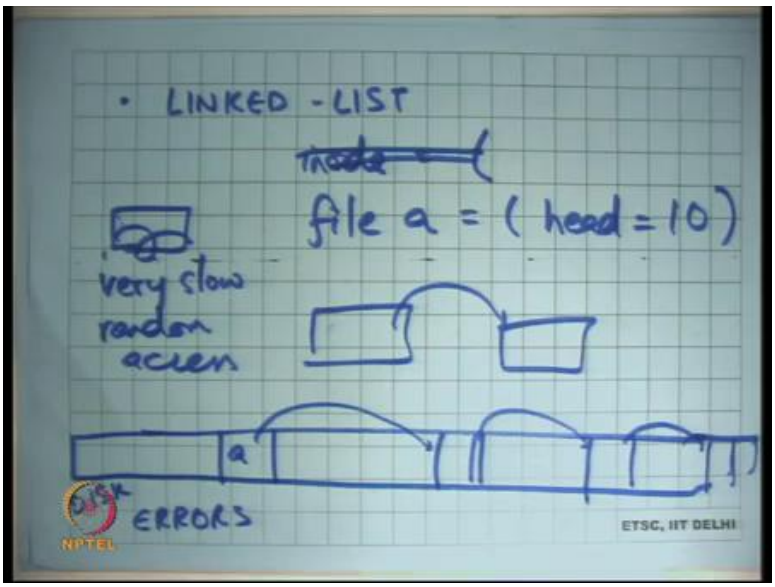
Very fast access if you want to know; so, the read system call, for example, only need to first get the inode. So, this information base and length it will be stored on the inode. So,

the read system call will get to need get to the inode, get the base pointer, add the offset to it and then that is it then.

So, it is basically just 1 metadata access to get to any random location. More importantly sequential reads which is also a common case which means you know a an application is just reading through the file sequentially, that is fast right.

A sequential file read directly translates to a sequential disk read and recalls that the sequential disk read is great right. So, sequential reads are fast here and so, this is great for any workload that requires sequential reads ok. So, we are not happy with fragmentation and growth. So, the other option is what is called linked list.

(Refer Slide Time: 47:05)



So, you know instead of an if it is an contiguous array let us have a linked list where inode; so, inode will point to let us say file a which basically means the inode storage is going to have a head pointer, head is equal to 10 and that is it right.

That is basically head is equal to 10 and then each block will basically say what is my next pointer right. So, if I have a disk then let us say this is file a its pointing to this location and this will have some next pointer which will point to b and so on right. And so, the storage that is been used for the next pointer will not be exposed to the user of course right. So, that is something hidden from the users.

The user cannot see the next pointer, it can only see the data which is stored in the rest 512 minus 4 bytes ok. So, what is what do you think about this? So firstly, access for a random offset is terribly slow, if I want to say I want to access offset 100 of let us say offset 10,000 of file a.

Then I have to actually do pointer chasing and at each pointer chase I have to actually get the disk in buffer cache and not just I have to get the disk in buffer cache and I have to wait for it to finish. I cannot overlap computation with disk access right, I have to wait for it to come before I can actually read its pointer.

So, basically, I have to block I have to keep blocking. So, random access is very slow, terribly slow; also, sequential access is also slow right. Because, now sequential access basically means that you have to actually do a lots of seeks because you have distributed all over that is fine. And, another important thing that; so, apart from efficiency the other thing you have to worry about in file systems is errors ok so, disk errors. So, for example, some blog goes bad, one blog goes bad in your file. What does it mean?

Right. So, in case of linked list if there is one blog that goes bad, the entire file gets corrupted right because you lose all the data. In the case of contiguous allocation one blog goes bad only that blogs worth of data has been lost, all the other data can be recovered right.

But, in the linked list allocation one block goes bad, the whole potentially the whole file goes bad right. So, you have to worry about that and also when you are doing this you know in ok. So, basically so that is another problem right.

So, we have looked at contiguous file allocation and linked list file allocation and both seem rather impractical. Contiguous has fragmentation problems and growth problems, linked list has access efficiency problems. So, we are going to look at some more realistic file systems which are some combination of these next time.