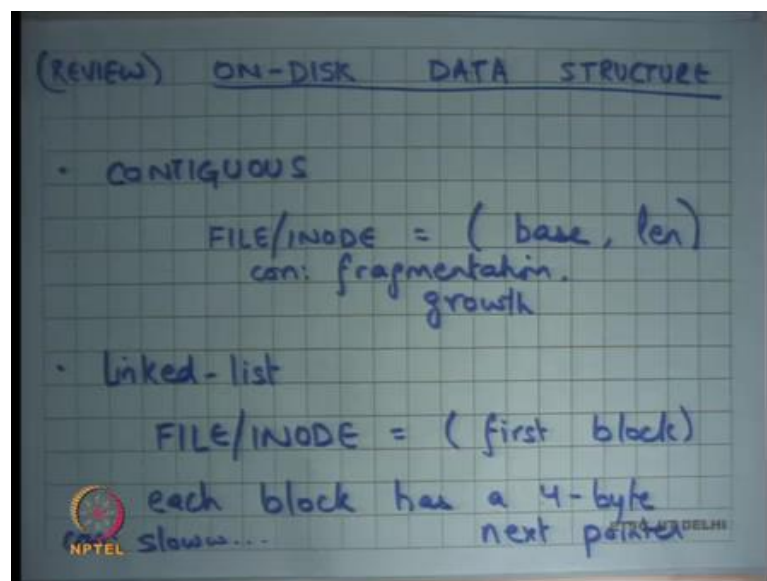


Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 32
Filesystem Implementation

Welcome to Operating Systems lecture 32 and we are talking about Filesystem Implementation.

(Refer Slide Time: 00:33)



And yesterday or in the last lecture we were looking at on disk data structures. We know which is basically the file system is nothing but in on this data structure. And we looked at 2 options one was a contiguous file system where contiguous allocation where a file or an inode is defined by or stores a base and a length and that is where the or file resides. Then advantage of that is that contiguous bytes in a file are also contiguous on the disk, so it makes for fast sequential access of the file.

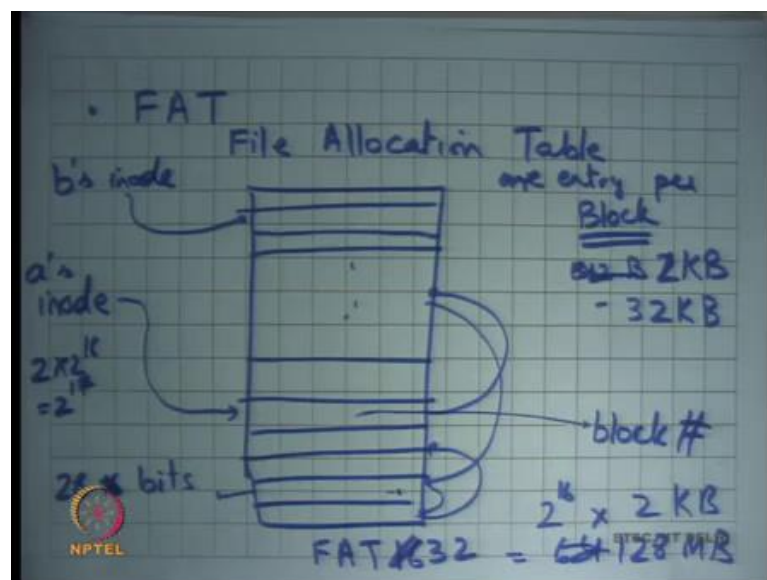
The bad thing about that here is fragmentation and growth right. So, it is similar to what we saw with segmentation in virtual memory it has those problems, the other on the data structure we looked at was a linked list which we said that let us the file or inode contain a pointer to the first block and then each block has a 4 byte next pointer. This gets rid of the problems of fragmentation and also growth, growth is very easy fragmentation is no problem, but it is very slow right.

It is slow not only because of sequential access, because you know for sequential access you have to just keep seeking because the file blocks will be strewn across all around the disk. But it is especially bad for random access you have to actually traverse entire linked list to get to any block in the file.

Moreover, notice that recall that our magnetic disk has this problem, that you know it is better for to give lots of IOS in flight simultaneously right. Recall that the disk controller is doing some scheduling internally and so it is it will be best for the OS to have lots of outstanding IOS for the days. So, that the disk and schedule them in the most optimal way.

But if you have a linked list allocation then and you are traversing the linked list you can have at most one IO in flight at any time right, because you have to wait for the first IO to finish and get the data before you get the next pointer and then you issue the next IO. So, there can only be one IO in flight and that is a really bad thing from a performance standpoint. And the third thing we said was bad about it was basically during reliability, so one link goes bad or one block goes bad the entire file gets corrupted.

(Refer Slide Time: 02:47)



So, let us talk about some more realistic file systems and one successful idea has been what is called FAT or File Allocation Table yes question.

Student: So, far the sigmatic segments if one block gets a corrupted end off if IO corrupted.

Student: So, going to that simple techniques like making that in case of storing both the head and tail can reduce a 12 bits.

So here is a suggestion that instead of using a singly linked list as I said let us have a doubly linked list and that would improve reliability by some degree. But really (Refer Time: 03:20) it I mean if you are just firstly it is it has all those problems of being slow right. So, that is no that is one big corner of this, the other thing is even if you have a doubly linked list if one block goes bad then the next point is gone, and the previous point is gone. So, I do not see how?

Student: I find it should be pointer of both the head and the tail, so in my inode.

Ok.

Student: I can reach.

I see, so you are basically saying that inside my inode I could just told the pointer to my inode. Let us say for each block I could store a pointer to the inode and so I know that these blocks belong to this inode.

Student: So, the what head on the doubly linked list and the (Refer Time: 03:55) of doubly linked list. So, this is a multi method 2 lock in the same file gets too low. So, means?

Student: He is saying like if a blocks gets corrupted; we can travel from page tails.

I see, so if the block gets corrupted then you can travel in the reverse order, means travel from the tail and you can organize your data structures are that 2 block. If only if 2 blocks get corrupted, then the file gets yes, it is a valid thing and you could do that but. So, it you know you could solve the reliability problem by adding more to this linked list idea.

But I mean it is bad anyways right it is bad because of performance reasons ok. So, here is an here is a very nice extension to the linked list idea, which is file allocation table first

used in MS dos, which was basically the idea is that this maintain this linked list not in the blocks themselves. But have a separate file that maintains this linked list right.

So, have a file which you will call the file allocation table right. So, this is the file allocation table or FAT, and this has all these entries for each block right. And now inode just points to so inode points to the head pointer which points somewhere here and within this file you have pointers like these right.

And so, if you want to go to so a file basically points to some entry here and that entry says that the first block it contains the block id right. So, block id or block number of the file and on the disk and so you can read that block so that is the first block. And if you want to get to the second block you change the pointer from here right. So, let us say if you wanted to get to the nth block you need to do pointer chasing within this table, you do not need to actually fetch the real blocks you only need to do pointer chasing within this table.

And the that idea is that if this if the table is small it can be brought into memory in one go and then this point is chasing can be done completely in memory right. And so, if you can do the pointer chasing in memory then you have eliminated much of the problems of the link less allocation that we had before ok. So, let us see how big does this file need to be well for every block we will have one entry.

Student: This block in a sector.

Right, so what does a block mean? A block would mean one sector or some small multiple of sector alright. So, maybe you know 8 sectors or 16 sectors, the MS dos block size ranged between you know 512 bytes to actually or 2 kilobytes for the FAT 32 system to 32 kilobytes right.

So, a block could be anywhere between 2 kilobytes and 32 kilobytes, the block size needs to be decided at the time of format right. So, a disk partition will have a constant block size throughout the life of that partition right. And so, depending on the block size you know you will basically name the disk blocks right.

And so, each entry here needs in so there the first incarnation of this idea was in the FAT 16 file system and the reason was called FAT 16 was each entry was 16 bits right. So,

each entry was 16 bits and the block could be between 2 kilobytes and 32 kilobytes. So, what is the maximum size of a disk that you can support, you know you could have at the most 2^{16} blocks in your disk and each block being let us say 2 kilobytes.

So, if you if the entry size is 16 bits the maximum size of the table can be only 2^{16} right that is the number of unique blocks that you can address and so that is 2^{16} blocks and into the size of the block. So, let us say it is 2 kilobytes then that is roughly 2 megabytes and 64 megabytes was 128 megabytes.

Student: 52.

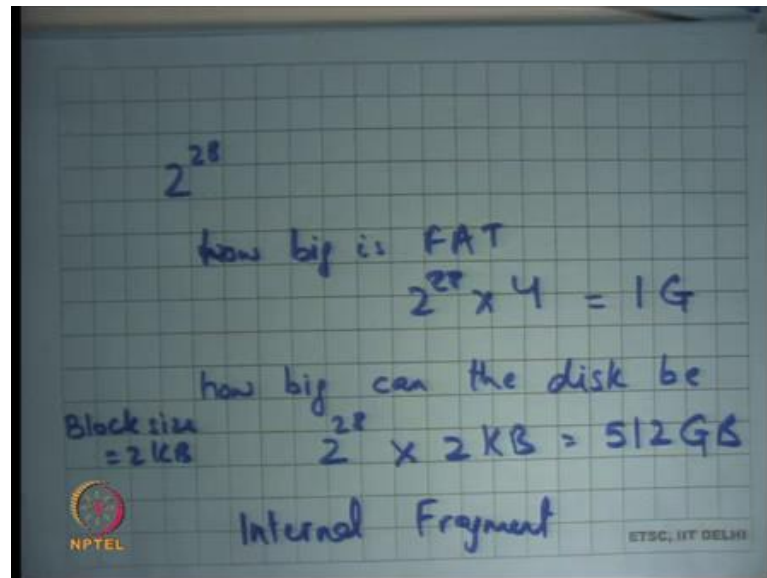
Right let us say the block size was 2 kilobytes and you have a 16-bit entry in your file allocation table you basically have a 128-megabyte disk that you can support with this organization. So, just to make sure that everybody understands this let us say I have another file. So, let us say this is a's inode and this is b's inode then b's inode with point somewhere else and then it will have it is own linked list to follow right and one block will only be part of one linked list.

So, one block can only be part of one file there is no sharing between files of blocks right. So, for a one, so the FAT 16 file system with the 16-bit thing could support a disk of at most 128 MB does not seem large by today's standards. But at that time, it was a reasonable size given the technology of that time right, it is roughly late seventies early eighties.

And what is the size of my FAT 16 table 2^{16} entries each entry being 16 bits. So, that is what $2 * 2^{16}$ right that is 2^{32} or 2^{17} that is a 128 kilobytes right. So, 128 kilobytes at that time memories were typically bigger than 128 kilobytes and so it was a reasonable scheme to have alright.

So, but you know it does not make sense in today's technology, there was a so it was extended to what is called FAT 32 today. And FAT 32 had instead of 16 bits 28 bits for each entry and it supported block sizes between 2 kilobytes and 32 kilobytes and with this you can have a maximum file size of 512 gigabytes.

(Refer Slide Time: 10:20)



So, let us say you have 2^{28} entries, what is the size of how big is FAT $2^{28} * 4$. Let us say each entry is 4 bytes that is 1 gigabyte right and how big can the disk be? 2^{28} . Let us say I was using block size of 2 kilobytes, then I have $2^{28} * 2$ kilobytes that is equal to 512 gigabytes right. So, you can support a maximum size disk of 512 gigabytes with the fact 32 size file system with a block size of 2 kilobytes.

If you want to have a larger disk one straight forward thing to do is increase the block size right. So, you could take it up to till 32 kilobytes and that way you can you know multiply this number by 16. Once again, the block size is decided at format time and just remains constant for the life of that disk partition.

So, what is the advantage of having a small box block size this is a large block size what is the tradeoffs what are the tradeoffs involved, one clear trade off is the larger the block size the larger disk you can support, but what are the other problems?

Student: It will.

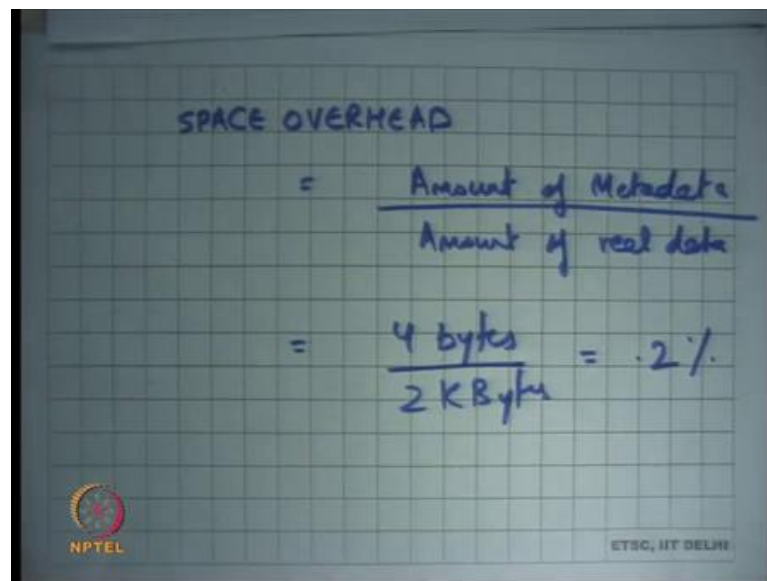
Firstly, there is a you know what is called internal segmentation. So, I want I was interested in let us say 16 bytes you know in the of the disk, but I have to read the entire 32 kilobytes right. So, there is more extra things that you may need to read into the memory right, you need to maintain the buffer cache in that way and you so that is one

thing. So, you need more time for the data transfer also you have your polluter buffer cache.

So, the more data you bring in the more data you need to evict and if that extra data is not really useful then you are polluting your buffer cache. The flip side of it is the larger the block size the more spatial locality can be exploited right. So, it is really a tradeoff between spatial locality and internal fragmentation.

So, the larger the block size assuming your program has a lot of spatial locality, a better thing to do is to have a larger block size. On the other hand, if it does not have that much special locality you know the smaller block size would have performed better alright. What is the space overhead of a FAT file system?

(Refer Slide Time: 12:58)



The slide is titled "SPACE OVERHEAD" and shows the following calculation:

$$\text{SPACE OVERHEAD} = \frac{\text{Amount of Metadata}}{\text{Amount of real data}}$$
$$= \frac{4 \text{ bytes}}{2 \text{ KBytes}} = .2\%$$

At the bottom left is the NPTEL logo and at the bottom right is the text "ETSC, IIT DELHI".

So, if I were to say you know space overhead of a file system, I basically say amount of metadata divided by amount of real data right. So, how much metadata is stored per unit of real data in this file system, well I am storing let us say 4 bytes per 2 kilobytes. Let us say I am using a 2-kilobyte block then that is roughly 0.2 percent right. So, that is not too bad that the space overhead of this file system.

(Refer Slide Time: 13:50)



What about reliability? So, let us first talk about Efficiency right. So, what were the few things that we were concerned about when we talked about efficiency number one, we wanted to say how good is sequential access. So, let us say I wanted to read the file sequentially from you know some x to $x + 100$ or whatever.

And so what should be my what should the speed be well in contiguous app it was great, what do you think in link less it was very bad what do you what do you think about FAT it is better than ling list.

Student: (Refer Time: 14:25).

Better than linked list because, you can compute in memory the list of blocks that need to be read a priori, you do not need to wait for blocks one by one right. So, you get you know the exact 100 blocks, these 100 blocks may be strewn across the disk. So, they may not be a contiguous necessarily, but you can give a hundred blocks on in flight to the disk.

So, you can have 100 outstanding IO simultaneously and the disk scheduler can schedule those 100 blocks efficiently and give you much better performance than the linked list allocation. But it is where it is not as good as the contiguous allocation of course, because the contiguous allocation would have take just you used one seek and one locate latency and the rest would have been the data transfer time.

Here you will have multiple seeks and multiple (Refer Time: 15:06) but at the same time you know because you have so many outstanding IOS the effect is amortized alright. The disk scheduler can schedule things properly. What about random IO? Let us say I just want to say get to the offset x in a file is it any is it fast or is it slow.

Student: Fast.

It is as fast as it can be right assuming that the file allocation table is in the memory you just chase the pointers inside memory, you figure out what this block you want to read and then you should that that requires just and now that is the minimum you need to do anyways yes right.

Student: Sir we need said about corruption of a block.

Right.

Student: (Refer Slide Time: 15:45).

Good.

(Refer Slide Time: 15:48)



So, that is efficient. So, we have talked about efficiency let us talk about. So, this is time efficiency of course and then let us talk about Reliability right. So, these are some parameters we are sort of evaluating our file system design on. So, is it reliable?

Student: Yes, provided memory is reliable.

The his has an answer provided memory is reliable, no we are not talking about reliability in terms of we are talking about reliability in terms of durability of data. So, you know it is we expect disks to live for multiple tens of years if not you know more. And so, if this cliff for that long the probability that one sector in the disk gets corrupted is extremely high, relatively high right.

And but we also want and the idea of using disk is that a data remains durable across tens of years let us say right. So, it is it possible that one disk block gets corrupted and that causes a lots of data loss in this organization.

Student: FAT.

Student: Sir.

Well if the FAT block gets corrupted. So, after the FAT is also getting stored on the disk right. So, that if the FAT block gets corrupted then yes you will basically suffer a lot of data loss. So, what is what is one easy way to fix it?

Student: Multiple.

Have multiple copies of the FAT that is alright. So, I have multiple copies of the FAT. I said 0.2 percent overhead now it will become 4 percent overhead space overhead.

But you know you will have you know significantly higher reliability right. So, that is easy so typically you will have multiple copies of 2 copies of the FAT in your on your disk to ensure the reliability question.

Student: Sir when we copy something or write something we will have to update on that table.

When we copy something or write something, we must update the table yes.

Student: But it can get the require right.

So, you know when I talk about reliability, I am talking about of a stationary disk all right. So, what happens if you know what happens while the program is running and some crash happens, you know that is not going I am not going to call it reliability I am

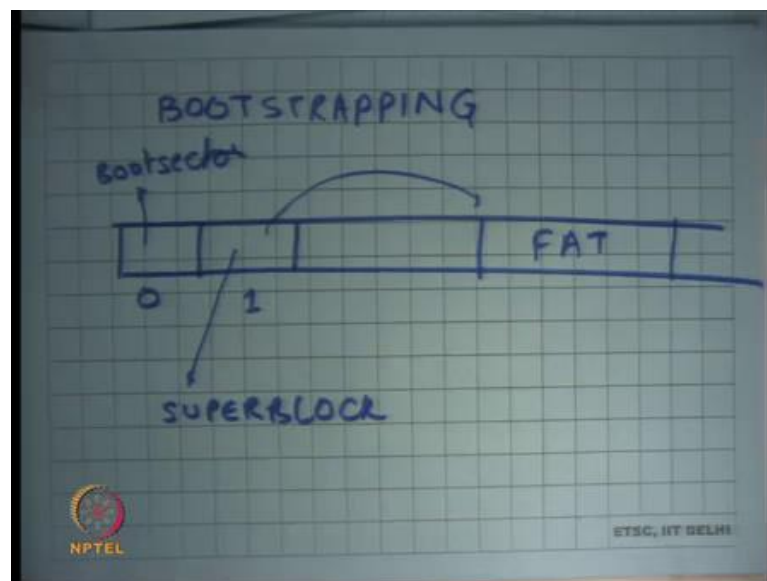
going to call it durability guarantees of the file system and we want to discuss that very soon ok.

But when I am talking about reliability I am talking about stationary disk and what is it is reliability. For FAT 32 we cannot keep the whole FAT table in memory, but yes you I mean you. So, of course, you don't need to keep the whole FAT table in memory, there is some caching that you can do there also. So, you can treat the physical memory as a cache for the FAT table itself alright and assuming there is locality of access in the FAT table the cache should have a very high hit rate.

Student: (Refer Time: 18:21).

Finally, how do you bootstrap the system?

(Refer Slide Time: 18:30)

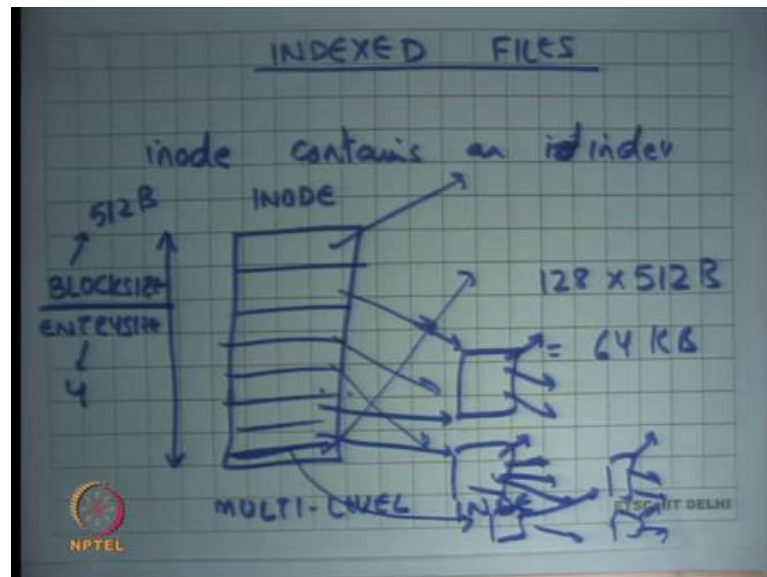


So, in general there for every file system. So, let us say bootstrapping so by bootstrapping I mean how does the OS know where the FAT lives right, once I gets to the FAT then it knows where all the files live that is no problem. But how does the OS is know where the FAT files well one you know one typical way to do that is let us say this is the disk and the 0th sector recall is the boot sector.

So, we do not use that for the file system at all ok. The boot sector is completely independent of the file system, it has nothing to do with the file system. Recall that the boot sector actually contains instructions that need to be executed at a boot time, so it has

nothing to do with a file system. And you know the first block is assumed to have some it is called a super block and this block contains information about the FAT let us. So, here is the FAT let us say right and then the FAT has pointers has to everything else alright ok.

(Refer Slide Time: 19:44)



So, FAT is a real file system that you use. The other type another type of file system is called what is called Indexed file system right. And the idea here is simple in the FAT file system we had the inode point to the head or the first block of the file. In an index file inode contains an index right.

So, here is an inode and it contains many rows and it has an index and so on right. Basically, saying that the first block of the file is at this location the second block is at this location and so on right. So, this is an index file. So, you have more information in the inode and so once you read the inode into memory then you can get to all the blocks of that memory of that inode ok. So, this is an alternate organization you just have an index over the thing. So, let us see what are the pros and cons of this.

Student: This index lives on disk.

This index lives on disk yes definitely.

Student: But these.

These are all on disk data structures.

Student: So, what then we are accessing the file give for the whole data structure into the memory and then.

Right, so now if I want to access a file and I know. So, this is called the inode right this index is also this index is stored in the inode. So, the inode needs to be brought into the memory and then you will look at the index to get the block right. Assuming that the inode was previously cached you know you get you take only one disk access. If the inode it was not previously cached, then you take 2 disk accesses one for the inode and one for the disk block right.

It is not very different from the FAT file system also you know you cannot you know it is not possible to store the entire FAT in memory or you may not want to store the entire FAT in memory all the time. So, sometimes you may take a miss on the fact itself right, so that it is similar to that ok. So, this is an index file system. So, what is the problem? Firstly, how big can the ion index b right.

So, how big can the index b you would want that the index is not bigger than the block right or the inode is not bigger than the block. And so, you know let us say your block size is block size let us say then block size divided by entry size and let us say entry size was. So, divided by entry size is the size of your index right. And let us say typical block size is let us say you know just take an example let us say 512 bytes and entry size is 4 then you have you know $512/4$ that is 128 entries.

And so, what does it mean? It basically means that there s a limit on the maximum size of a file that you can have right. So, what is the maximum size of a file you can have you have 128 entries into a block size of 512 bytes you know that is roughly $128 * 512$ that is 64 kilobytes. So, you know index files look good, but you know for example, for a block size of 512 bytes I can only have a file size of 64 kilobytes a block size of 2 kilobytes will increase the number by a factor of 4, but that is not good enough right.

So, it is not great because unless I you know increase the size of my inodes so, but so, it is a tradeoff between the size of the index and the size of the file. So, you know the larger in the index you can support, but that also adds to space over it. So, what is a

better thing to do, you know this is a one level index one level index can only grow to that you know one level index can only point to that much data.

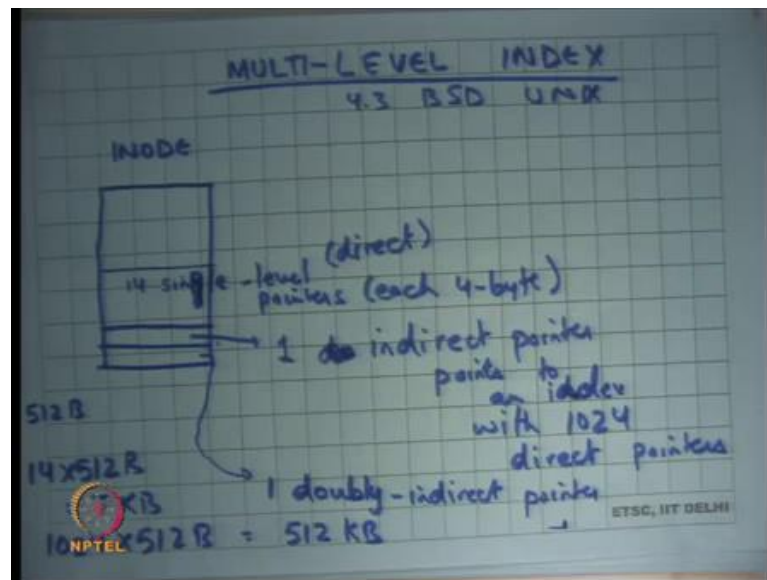
So, you want to have a 2-level index that is one way of you know we have all studied things like trees in our data structures goes. But if you have a 2-level index then the trade off is that I have to make 2 disk accesses before I actually you know actually get to the block, so actually 3 total disk accesses.

So, it is not and is given that most files are small right. So, most of the files in your file system if you think about it are executables which are roughly few kilobytes in size or configuration files which are just small text files or code files dot c or dot h files or java files these are also you know small relatively small files few kilobytes. And so, I want to optimize for the common case, which is lots of small files, but then I have some files that are really large right, like log files which run into megabytes hundreds of megabytes or even gigabytes right.

So, my typical workload or you know file size file distribution size this version is lots of small files some very large files and I want to optimize for that. So, instead of having a uniform 2 level structure what is done is you have a multi level index. What does this means is that the first few entries are single level, then few entries are actually double level which means they point to other indexes that point to more blocks and then the last entry or last few entries are actually triple level or actually this is called double level. So, then there is triple level which means you know it is a three-level hierarchy.

So, what you have done is that for the common case that you have small file, you have only one access in one meta data access and for the other common case which is very large file small number of large files you will have to make 2 or three meta data accesses. But those accesses are likely to be get degrade or mortised over a large number of blocks right after all there was a large file. So, you are going to probably access a lot of blocks in that file if you are accessing it right.

(Refer Slide Time: 25:46)



So, that is let me draw this again this is a multi level indexed file all right. So, let me take the example of a real operating system 4.3 BSD UNIX and let us look at x file system structure and users multi level index file inode contains. So, this is the inode it contains 14 single level pointers each 4 byte. Next entry contains 1 doubly index pointer or indirect block one in you know the other way to call it.

So, instead of calling it double index double level I will call it one indirect. So, these are direct pointers and one indirect pointer that points to an index with 1024 direct pointers all right. And last index so this is the this one entry and then last entry points to 1 doubly indirect pointer alright. So, by doubly indirect I mean it points to a block that contains pointers to more indirect.

So, it is a three-level hierarchy right. So, it is a 2-level hierarchy from here on, here there is a one level hierarchy and here these are direct pointers. So, with something like this you have optimized for the common case which is that small files. So, no small files of size less than let us say my block size was 512 bytes then files of size 14 into 512 bytes that is 7 kilobytes.

So, files of up to size 7 kilobytes will need only 1 metadata access to be referenced right. Files of size you know so 1 indirect pointers with 1 0 to fold direct pointer. So, that is 1024 into 512 bytes. So, that is 512 kilobytes. So, files of size up to 512 kilobytes will need 2 metadata accesses to be able to you know dereference them.

And then file is bigger than that which can you know which will be 512 megabytes or larger or can be done using 3 metadata accesses. So, that is basically it is done, still you have a size maximum size limit on the maximum size file you can have. So, you know every file system will have some limit and what is the maximum size of a file that you can have and, but all you are doing is just making that limit really large right and depending on the current technology that may suffice and or may not suffice depending on that right for example, you know for a long time Linux did not support files larger than 2 gigabytes right you know (Refer Time: 29:18).

For example, I remember till 2 early 2000s you know Linux was not supporting files greater than 2 kilobytes, but then now future versions of the file system basically allowed bigger files and so on right. So, depending on the hardware technology you will basically revise this as it goes.

So, that is basically the file system structure we have looked at the fact file system which is an in memory a linked list and then there is an index file system and both are widely used index file system is used in a Unix and also the NTFS file system is an index file system and the FAT that file system is also you know by it is name called FAT file system.

Now, one of the things that you do worry about in a file system is what happens on a power failure or a crash you will you are in the middle of doing something and a crash happens and then you come back again your things may not be an inconsistent state, how do you deal with that all right. So, one way to deal with that is that you do not worry about it at all right, you just say if a crash happens in the middle of something.

(Refer Slide Time: 30:36)



Then I have add my at a reboot time I will run a program for example, you know in the Linux for Unix systems have this function program called fsck. So, you run the fsck program gets to run on a reboot and it will you know scan a global scan over your file system and see if it finds something wrong. And sometimes it may not be able to automatically decide whether what should the fix be.

So, if it finds something wrong what should the fix be it may not be able to automatically decide. So, it may need to actually ask the user and you know it is a very painful positive decision procedure for the user to figure out you know what happened and what could have what should the right thing to do.

And we are going to look at what kind of questions can come up and what kind of things that can go wrong alright. The problem with this the nice thing about this is that in the common case when there is no crash, things can work at full speed and very simply no not problem the bad thing about this is that the fsck program can take a long time. In the early days 90s disk used to be small so, a global scan was possible in a few seconds maybe you know 10s of seconds or minutes.

But will this sizes of 512 gigabytes to 2 terabytes you know distant and easily take up to know half an hour or 1 hour or maybe more ok. So, that is a cause of concern and so this technique while was used for a long time in real operating systems is no longer used and some other things are used the other. So, that is one thing, the other thing to do is

basically say that I will ensure that you know so there is something called Journaling and we are going to discuss what it means.

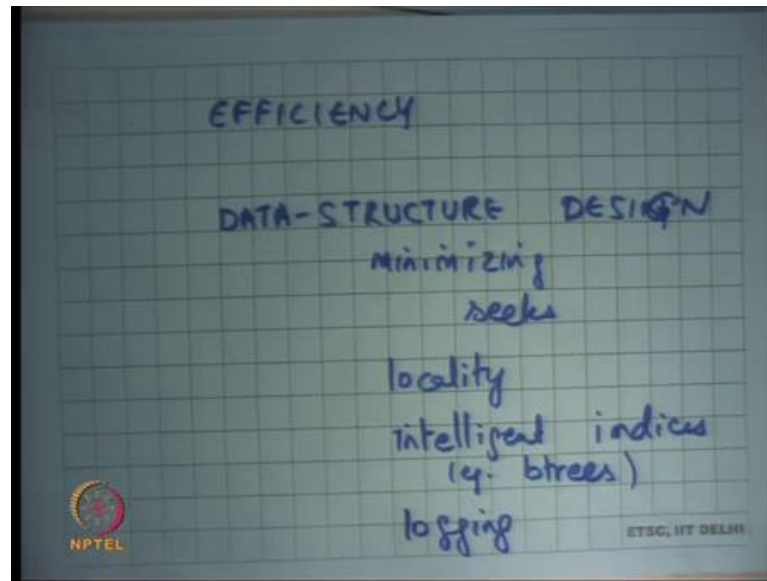
So, journaling basically ensures that your updates to the file system are atomic with respect to the power failure all right. So, there is a way to; there is a way to ensure that updates to the file system are atomic with respect to file power failure. So, if a power failure happened in the middle of an operation, then it is as though the operation never took place alright.

So, that is you know you maintain a data structure that may ensures that a non base data structure that ensures that the it does not take place or and if it (Refer Time: 32:40) happens after the end of the operation then you know the operation and so either the operation has taken place in full or not at all, so that is atomic. So, one thing is journaling everything, which means you general journal both the meta data which is you know these inode structures that you have and the data of the user that is that will provide you very good guarantees over atomicity and the user can be sure that whatever is doing is correct.

And the other option is to journal only meta data right. Here you are saying that I will make basically make sure that my metadata or the inode structures and other things that I am using for my file system are remain consistent across power reboots. But the users data may become an inconsistent right, it is something like you know killing a process. When you kill a process you make sure that the kernel side of state is cleanly freed right, recall that you do not just free the process you just wait for the process to reach some clean boundary before you actually free it.

But at the same time there is no guarantee for the user right the user could be in the middle of something and he can immediately get vanished right. So, it is a you and the user should expect that right. So, that is the other approach which is journal only the metadata and we are going to discuss this again.

(Refer Slide Time: 34:07)



The other thing you have to worry about is of course Efficiency. So, how do you provide efficiency? Well firstly, you know data structure design that we have already discussed for like them and this is basically about minimizing seeks right.

So, some things that can be done here is locality right. So, by locality I mean play try to place blocks consecutive blocks of the file consecutively on the disk, because assuming that the file has spatial locality it will automatically it will directly translate into spatial locality on disk and so you will reduce the number of disks this seeks right.

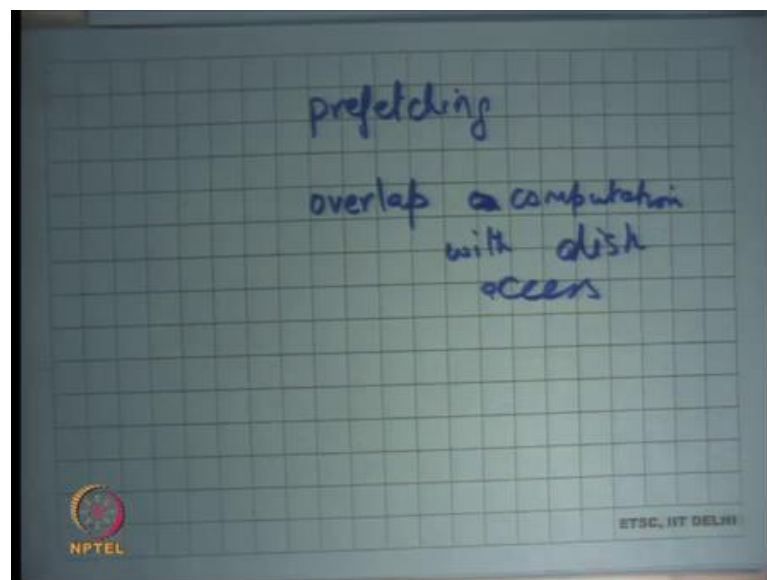
So, for example, your disk defragmenter program that you may have seen in some operating systems, will do we will try to improve locality to give better efficiency for a file system right. Then you know intelligent indexes intelligent indices right and examples you know multi level you have seen there could be you know you could use be trees and choose.

So, be trees with carefully chosen values of b can basically optimize performance because, your b can be tuned to the size of your block on the file system right. So, if you carefully choose b trees with the for the chosen b is then you can optimize these things and you know databases use this a lot right. And the other way to minimize seeks is what is called logging. So, here the idea is that try to make writes to the disk as sequential as possible right.

So, instead of right let us say I wanted to write ten different files at the same time and these files are completely strewn across the disk. So, instead of going to each file separately and writing to them could I you know organize my data structure such that all these rights appear as a log and so it becomes one sequential right and that is what is gets written to the disk and asynchronously it gets updated on the real file system ok.

And so, you and using that you can basically minimize the seeks required on the fast path which is you know when you are waiting for the district to finish. So, you can just log the data on to the disk in one go and asynchronously transfer it to the file and then we go to look at examples on how to do this.

(Refer Slide Time: 36:46)

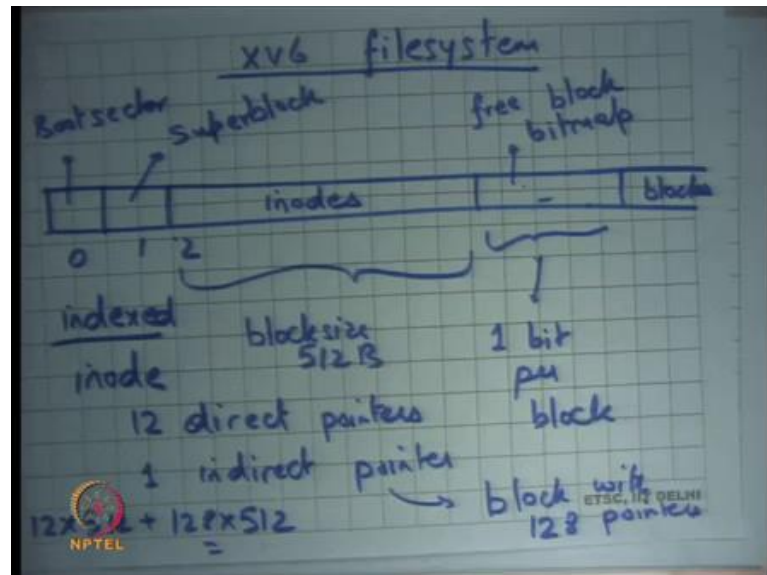


The other thing you can do is Prefetching of course right. So, you read some data you have some algorithm which basically figures out that there is a lot of spatial locality. So, you prefetch data of course there is a; there is a; there is a trade off trade off about how much to prefetch and how much you cannot prefetch and overlap computation with disk access.

It is a good idea to you know while you are doing the disk access to also keep computing, so that firstly you will keep you of CPU busy and secondly your computation may lead to more disk accesses and that may lead to better this scheduling overall all right.

So, we are going to look at you know. So, these are the basic sort of design principles on which we are going to decide what is a good file system and it is not a bad file system.

(Refer Slide Time: 37:47)



But let us first start with a simple file system which is the Xv6 file system and let us look at a realistic implementation of a quite simple way of bone file system all right ok. So, so here is the Xv6 file system, let us say this is my disk and this is block 0 that is the boot sector this is block 1 that is the super block right and then from block 2 to some number they have inodes ok.

Then to some point they have what is called a free block bitmap and then you have the block stem cells right, so let us look at it. So, the first 2 sectors are clear good sector and super block, the next few sectors are reserved for inodes. It basically means that all inodes will be allocated from here right. So, all the so that basically immediately puts restriction on the maximum number of files that you can have in a system all right.

One advantage of putting all the inodes together like this is that often you may want lots of locality between your inodes right. So, let us say you are doing ls right. So, ls basically wants to look at all the files in a directory and all it cares about is you know things like last modified time and some characteristics that can be just go you know you do not need to look at the data blocks you only need to look at some statistics about that inode right. And so, if lots of inodes are being read there is a lot of locality between the inodes and so you can service a lot of requires from here all right.

But on the other hand conversely you also expect there to be a lot of locality between inodes and data blocks right it is quite also quite likely that when you access an inode then you are going to access the data block corresponding to that inode. But this organization does not exploit that locality right so there is a trade off. Then there is a free block bitmap it basically says which all these blocks are free.

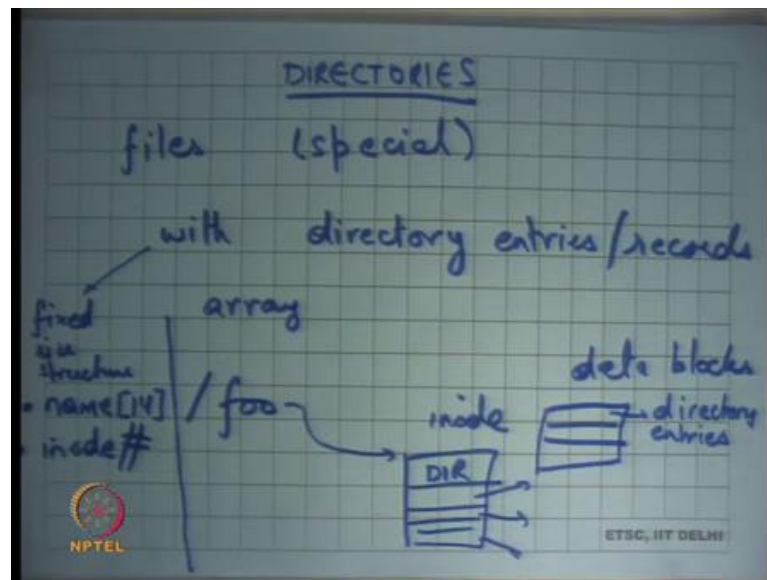
So, it is basically maintaining a list of free blocks right and basically using one bit per block, indicating whether that block is free or not. Once again you expect that this free block bitmap is a very sort of hot region because, if you are creating or deleting a lot of files or and so on or writing to a lot of files then you probably need to do that have this and so it is good to have this in a very local way. So, there is a locality in free block bit right.

Let us look at the inode structure. So, they use index files right and the inode structure has 12 direct pointers all right and 1 indirect pointer block size is 512 bytes. Indirect pointer points to a block with $512/4$ that is 128 pointers ok. Notice that the inode itself has only 12 pointers $12 + 1$ 13 pointers.

Where the indirect block and have more pointers why? Well the inode has also extra information in store likes you know things like what is what the permissions are, who owns it what is the last modified time stamp and other things right.

So, there is more information in the inode. So, that takes up some space, so you have only space left for 13 pointers in the inode, but in the indirect point block you have the whole space available for pointers all right. So, you can calculate what is the maximum size of a file that you can have, you can have $12 * 512$ bytes + $128 * 512$ right, that is the maximum size of a file you can have roughly 64 kilobytes all right ok.

(Refer Slide Time: 42:33)



So, what about Directories, how are directories implemented? Directories are nothing but files but special, they are special in that the user cannot read them directly it is the operating system that can read it all right and so and it is only the operating system that can interpret it right. So, directory is a file with records. So, directory entries right and on XV6 it is represented as an array of directory entries all right.

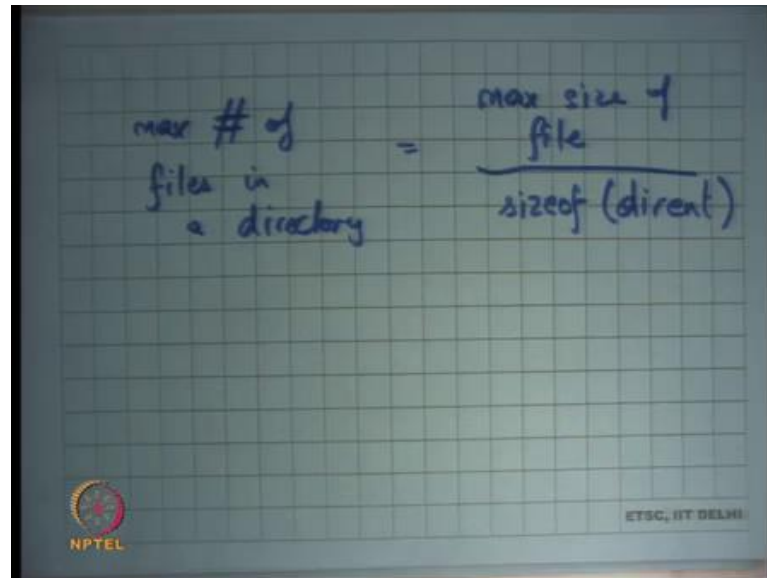
So, for example, if you have a directory let us say you have a directory called slash foo, then slash foo will point to an inode. There will be some status file status flag in the inode which says I am a directory right, all other files will have this status flag saying I am a file, but a directory. So, that so there will be a bit which says distinguish between files and directories.

And then just like everything else it will have pointers direct pointers. So, these are, and these are data blocks, but these data blocks will contain directory entries right. So, basically of a directory is nothing but a file which stores directory entries in them all right and directory entries and XV6 are represented by a structure which is fixed size structure which contains 2 fields name and inode number all right.

So, a directory entry contains 2 fields the name of the file or the directory right. So, this name could be the name of file or a nested directory and the corresponding inode number and XV6 says you know I will only allow names up to 14 characters long. So, you have a fixed size on how big the name can be and inode number is fixed and so there is a fixed

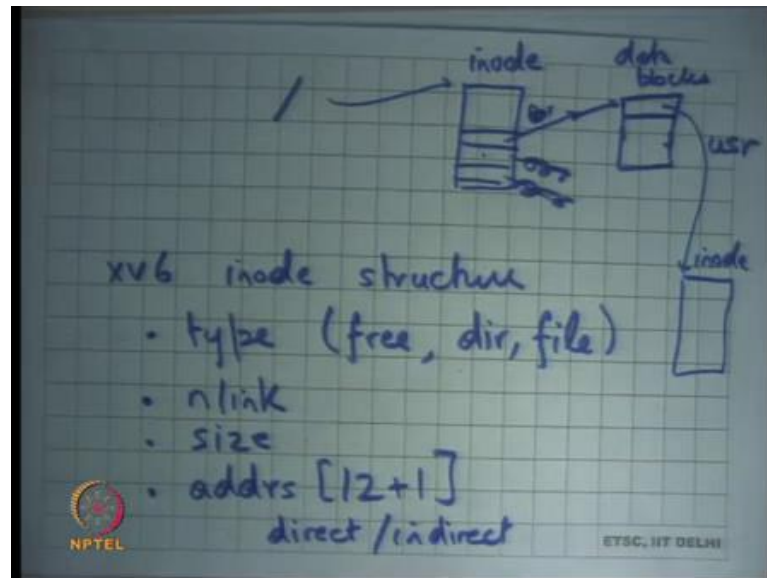
size structure of directory entries and that is basically you know used to implement directories. So, what is the maximum size of what is the maximum number of files you can have in a directory.

(Refer Slide Time: 45:30)


$$\text{max \# of files in a directory} = \frac{\text{max size of file}}{\text{sizeof(dirent)}}$$

Let us say my directory, so the maximum number of files I can have in a directory is basically is what? Max size of file which we computed earlier which was let us say 64 kilobytes on XV6 divided by size of directory entry right let me call it dirent. So, that is basically the maximum number of files you can have in a directory right. Whatever the maximum size of file you can support that is the amount of space you can have for storing directory contents and so you can have that many entries inside a directory.

(Refer Slide Time: 46:23)



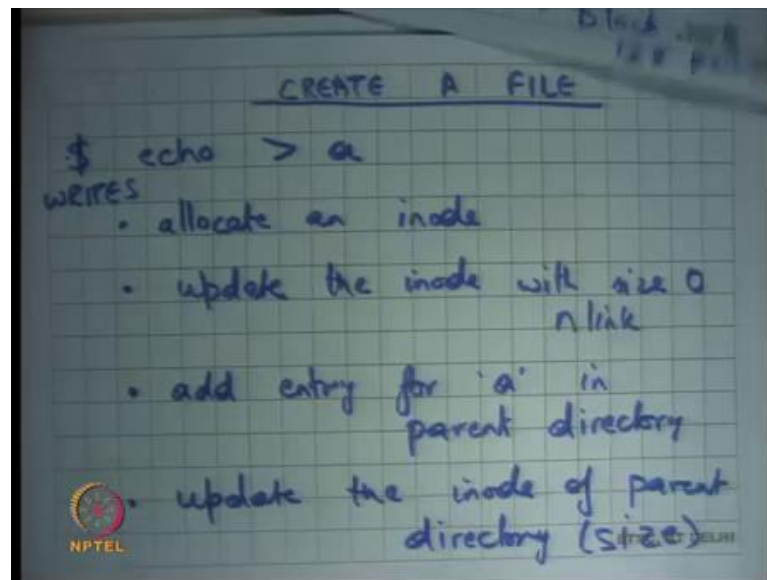
So, if I were to draw a diagram let us say this is slash. So, that is the slash directory the root directory that points to an inode right and these point to different. So, these have you know the name can be let us say use slash user it is pointing to another inode sorry. So, these point of data blocks right. So, the inode themselves point to the other box. So, let us say the slash directory has only 10 files and 10 files can fit in 1 data block.

So, you only have 1 data block all the others are 0 right and this data block will have directory entries which will be mapping names like slash user to inode numbers right. And they themselves may be directories which mean they themselves may now have this structure within them or they may files, in which case these data blocks do not have any special meaning they are something that the user has defined.

So, let us look at the XV6 inode structure once in detail a little bit, you have a type which says whether it is free or directory or file right. You have nlink which says how many inward links do I have right, what is the size? The size basically means what the size of this file for a file. And for the directory it basically means the number of directory entries into the size of the directory entry right.

So, in both cases size basically means the size of the data blocks or the size of the useful content in this file or directory and then you have the addresses or pointers you recall the 12 plus 1 thing right direct and indirect.

(Refer Slide Time: 48:47)



So, let us say I wanted to create a file. So, let us say you want to create a file on XV6 and let us say you did that using this command called echo greater than a. So, what happens now just as a recap you are running the shell process you type this command it calls fork it calls exec echo, echo program gets to run with its output redirected to the file a right.

Now before, but before the file a let us say does not exist earlier. So, the first thing the program needs to do the shell program use to do is to create the file a right. So, let us look at from a disk perspective what all needs to happen for the file to get created ok.

So, let us look at this figure I want to create a file a. So, first I will allocate an inode how will I allocate an inode? I will go through these lists of I nodes and find the first one that is free, so that is allocating an inode. So, that is one disk access and once when I allocate it, I will also mark it as a file instead of a free flag I will mark it as a file flag.

Then I will update the inode with size 0 right. So, you will allocate an inode and you will update the inode with size is equal to 0 and you may want to update other things like nlink etc. For example, now there is a directory that is pointing to me. So, you would not want to increment the end link of this file ok.

And you may also so you may also want to 0 out all the direct and indirect pointers. So, you know so that is another disk access you could have potentially combined the first 2 disk accesses if you are; if you are; if you are being smart about it, then you add entry for

a in parent directory right. So, how do you do this, if you know you basically look up the parents directory to a directory is file and so the file contents are going to live somewhere here.

And there you basically append to the directories file and another directory entry which has name equal to a and inode number equal to whatever you allocated, what do you found earlier right. So, that is another disk write. So, I am actually just counting the number of disk writes, I m not counting the disk reads at all right. You may actually need to go through the directory that is inode and the directories data blocks to basically get to that point and then you update the directories data block.

So, in this case the updation of the directories data block means adding or appending to it is contents. So, what is the last thing you need to do update the size of the directory right this you have appended to the file of the directory. So, you need to update the inode of parent directory particular size.

So, at this point you have created the file, you have allocated an inode initialized it created in mapping using by appending a directory entry to the parent directory and updated the size of the parent inode ok. So, you know many disk accesses for one operation all right. So, right so let us stop here and we are going to continue this a this method of looking at what happens on a file operation and in particular we are going to look at what happens if people can if multiple processes come concurrently try to write to the same file.

For example, or how does data get appended to a file and how do files get removed from a directory. So, you know what are all the operations there need to be done to do this ok.

Thanks.