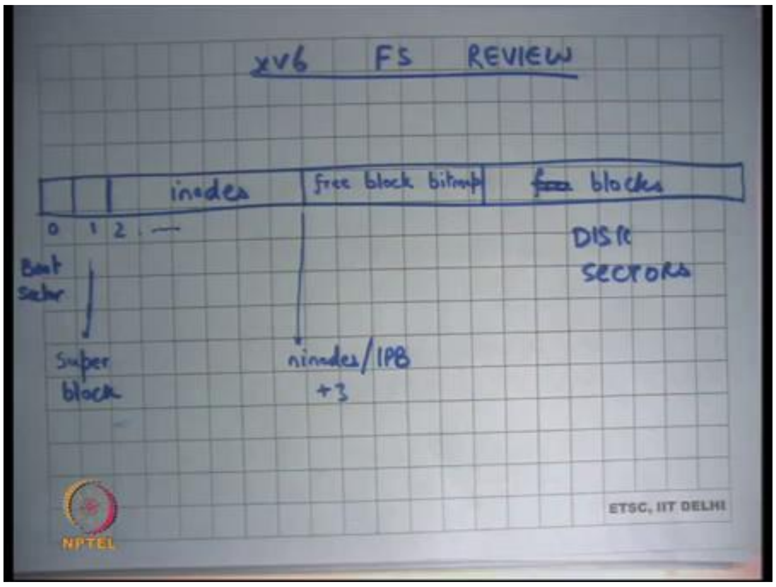
Operating Systems

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

Lecture – 33
Filesystem Operations

Welcome to Operating Systems, lecture 33.

(Refer Slide Time: 00:29)



So, last time we were looking at Filesystems and how file system could be organized; we looked at few different types of file system organizations and we said let us look at the xv6 file system in more detail. Recall that the xv6 file system is an indexed file system in which each file is represented by an inode and an inode contains pointers to the data blocks right.

And, we said that you know this has problems of the maximum this limits the size of the maximum file. So, one simple way to deal with it is to have a 2-level hierarchy but given that most files are small having a 2-level hierarchy for all files is not a good idea.

So, the common optimization is that the first few data pointers or the first few block pointers are direct pointers. So, it is a one level hierarchy for the first few bytes of the file and after certain size of the file you have a 2-level hierarchy and still further you have a 3-level hierarchy. So, that allows you a very large file overall yet having a very small access latency for small files which is the common case. Even for large files the average access latency is small because the cost of index lookup gets amortized over a large amount of data hopefully, alright.

So, we were looking at the xv6 file system and the xv6 file system the let us say this were the disk sectors right or disk blocks. The zeroth sector is the boot sector that is basically completely independent of the file system; file system does not touch it.

The sector number 1 is the super block that contains meta level information about what this file system is. For example, for a fat file system it may contain things like what is the block size right. So, this is sort of written at the format time.

And, then from sector 2 till some value which is hardcoded you have what is inodes right. So, the idea is that you only going to allocate inodes from this area. So, that immediately puts a limit on the maximum number of files you can have in the file system right and then there is the region which is the free block bitmap you can you basically store 1 bit per block to indicate whether the block is used or free.

And, then you actually have the blocks themselves; actually, I should not say free blocks these are basically you know all the blocks free or used blocks right. So, that these this is the place where all the data is stored; this is the place where you basically stored whether the block is valid or not free and this is the place where you store inodes for the files right.

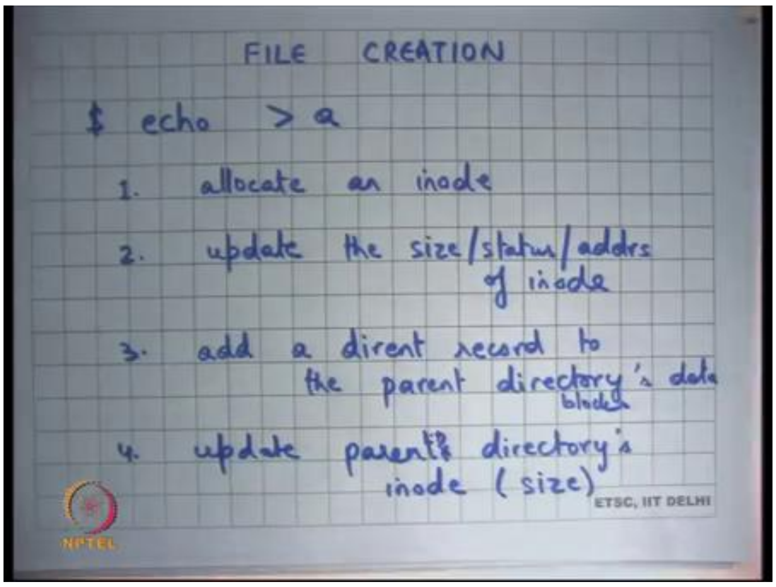
There may multiple organizations you could do here. You could have you know for example, interspersed inodes and blocks that would have had the advantage that you could have had probably better spatial locality between the files index and the files data blocks.

On the other hand, if you want to go through a lot of files for example, you want to do you know a scan of the entire data directory structure and you all you care about is looking at the inode of each file. For example, you just want to count the number of files let us say, then this is a much better structure because all the inodes are located in this region.

So, you can do one sequential read assuming your memory is large enough to read all the inodes and then and process them together right. So, given that we are implementing such file systems on a magnetic disk the most important optimization in general is to reduce the number of random disk accesses, be it reads or writes right because a random access involves a seek kind of rotation. A sequential access is relatively much cheaper right alright.

So, and this constant IPB basically says what is the how many inodes can you store per block. So, inodes per block that depends on size of inode you choose. If I recall correctly the size of an inode on xv6 was 64 bytes and the size for block is 512 bytes right.

(Refer Slide Time: 04:33)


FILE CREATION

$ echo > a

1. allocate an inode
2. update the size/status/addrs of inode
3. add a dirent record to the parent directory's data block
4. update parent's directory's inode (size)

So, we are also looking at what happens if I want to create a file. So, let us look at some operation then look let us look at what are the rights in the disks that need to happen for each operations. So, if I want to create a file let us say I create a file using this command echo greater than a, that just creates an empty file called a in the current working directly or the process right.

So, that current working directory is also the parent directory of this file called a. So, how I am going to do it? I am going to first allocate an inode, I am going to traverse the list of inodes and find one which is not used ah. Update the size, status, and adders of the inode, alright. So, basically just allocate the file and you know say that it is basically size 0, its status is allocated, read – write who owns it for example when it was created and so on whatever else you want to write about it and then you create a directory entry record in the parent directory right.

So, you want to you have created a new inode, but you also want to link it to the parent directory and this case the parent directory is the current working directory. Linking it to the parent directory basically means adding a directory entry record in the parent directories file. Recall that a directory is nothing, but a file with the with the contents of the file being directory entries that indicate the files that are present in the directory right each directory entry is a mapping from a name to an inode number right.

And, you know for simplicity xv6 allows only a 14-character name at most. So, that ensures that a directory entry is the fixed sized directory entry. But, of course, you could make it more complex to allow better you know longer names for example.

So, you create an you add a directory entry record I am going to call it dirent record to the parent directories data blocks right. So, this block is going to be added to the directory entries or directories data blocks and then you going to update the parent directories inode.

So, you have appended a block to the parent directories file and so, you want to also update the parent directories inode to indicate that the size is changed right. So, you know 4 or 5 or some number of writes random writes to the disk to be able to create a file like this right.

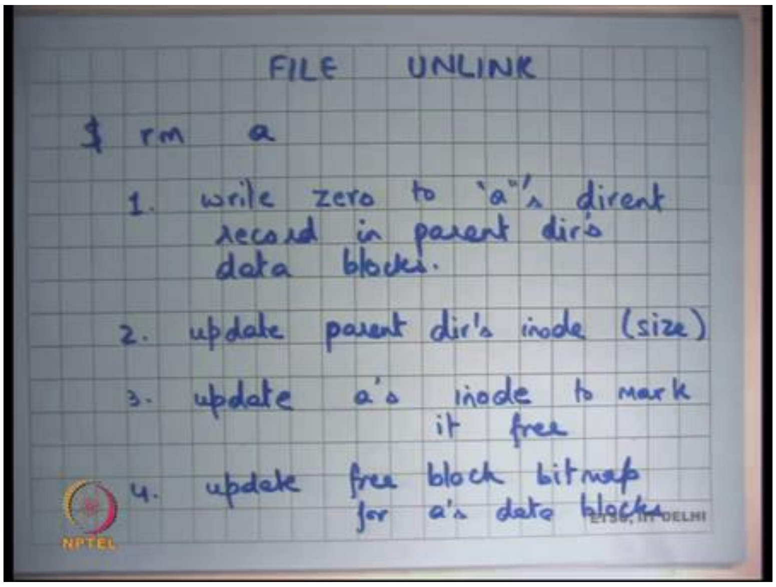
Let us see what happens on a file append. Well, in some sense file creation was similar to saying I want to append to a directory right. Creation of a file is nothing, but saying append to a directory, but let us see you know what happens on a file append. Here I instead of allocating an inode I allocate a block first which basically means write to the bitmap.

Blocks are implemented differently from inodes there you have separate bit map and you have the block themselves as opposed to the inode was having a status filed inside that whether it was free or not right. Then, you zero out the block contents you write X to the block. So, all this is basically being written in the block area.

This write was to the bit map area and these two writes are out to the block area and then finally, there is the write to the inode area that is basically you update the a's inode right. You update the a's inode basically you why do you need to update it? You need to update the size there is one character in it. So, you update the size from 0 to 1 and then you may you also want to update the pointers.

Recall that is an index file system, you are going to update the point inside the inode to point to the first data block right. So, once again all these are random writes they are completely in different parts of the disk and you are going to have to do 4 or 5 random writes to be able to implement file append in a file system like this alright.

(Refer Slide Time: 08:07)



FILE UNLINK

$ rm a

1. write zero to 'a''s dirent record in parent dir's data blocks.
2. update parent dir's inode (size)
3. update a's inode to mark it free
4. update free block bitmap for a's data blocks

Let us look at file unlink. I want to remove a file from the current working directory, or which is which should be of the parent directory of a and let us assume that a exists in this current working directory. So, now one way to do it is you first write you first look at the current working directory and find a file called a right.

So, you find something if you do not find something then you can return an error or if there exists a file called a. So, you find the file called a and you zero it out which basically means that it does not exist anymore right.

So, basically xv6 is implementing directories as a list of as the file that contains a list of dirent records, but the dirent records could themselves have a you know have a zero to indicate that it does not exist. So, write zero to as direct record in parent directories data blocks. So, this is the write to the data block right.

Then, you update the parent directories inode. You may you know in this case if it was a last one, you may want to update the size of the parent directories inode. So, first you wrote to the data block region, then you rewrote to the inode region and then you update a’s inode to mark it free right.

So, this you are deleting a you have marked unlinked it from the parent, but you also want to mark it free, so that it is usable by other files or other operations in future and then you update the free bit bitmap block for a’s data blocks, alright. So, you also mark the data blocks to be free, so that once again a few different writes that are happening to the disk right ok so, alright.

So, we get some sense of you know how costly these operations are to do read and write you know create read and write etcetera. Let us first talk about what how do you deal with concurrency. So, let us say if there are two threads or two processes who simultaneously execute want to append to a file or they simultaneously want to create two files with different names in the same directory.

So, one needs to have concurrency management. One simple concurrency management is have a global lock over the entire file system. Does this global lock need so, you know you can have one global lock that is called the file system lock and only one thread can take this lock at any time by the definition of lock and then each the thread that completes its operation then gets out, but that is obviously, not a very nice design.

Anyways the file system operation is also slow and if only one thread can be inside the file system then if there are multiple threads accessing completely independent parts of the file system you have greatly serialized and made the execution and made execution really slow right. Recall that the most important one of the most important as optimization is to allow the disk controller to have lots of IO’s in flight.

If you have one single serializing block for the entire file system you can only have a few may be even just one IO and flight at any time. So, having one global lock is not a great idea.

But, let us first talk about if I had a global lock where will I store this global lock? Is it enough to store it in memory or do I need to store this global lock on disk? Do I need to implement locks in memory, or do I need to implement locks on disk? What does it mean to implement locks on disk?

It locks implementing locks on disk means you know same thing you assume that something on the disk is atomic. So, recall that we said that there is some instruction that can atomically set locations in memory, bytes in memory right.

Similarly, we need some assumption on the disk. On the disk let us assume that right of a sector on the disk is atomic. So, if you know either the sector gets written or it does not get written. It is not like half of the sector gets written by one thread and then the half gets written by another thread, that is not going to happen. Also, let us assume that if the power goes down in the middle of write into the sector, then the sector write will get completed right.

So, the last sector will either get written or not get written at all, right, that is atomic. So, back to my question do I need the file system lock to be implemented as a disk variable as an on-disk variable or does it is it to just have an in-memory variable to implement mutual exclusion? All I need to do implement is mutual exclusion right. So, can I do implement can I implement mutual exclusion by in memory variable or on disk variable? Answer.

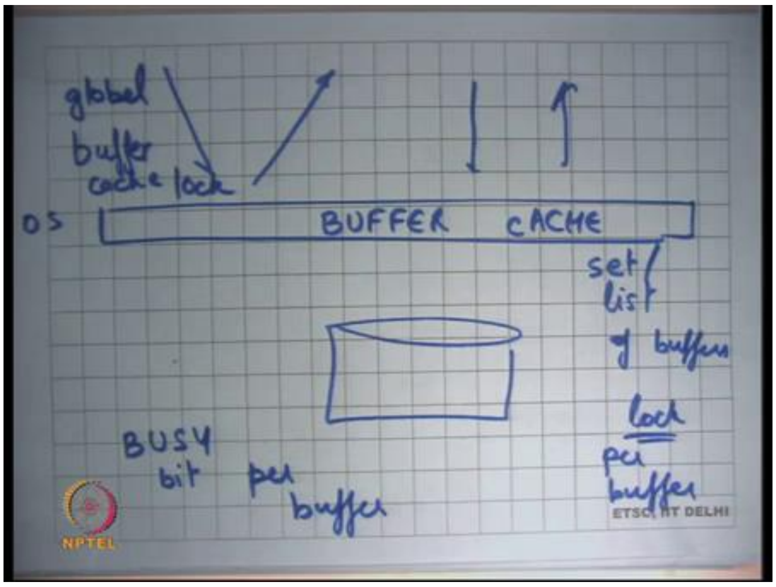
Student: If we want synchronization to proceed even after a power reboot.

If I want synchronization to proceed even after power reboot, well, I mean. So, right now I am not I am just saying that I want mutual exclusion across multiple threads and threads just die after power reboot right. So, threads are basically just in memory constructs. So, they do not; they do not carry over across power reboots.

So, in memory suffices if the only you know so, if you are basically assuming that your disk is only accessed by this operating system that you are using it is not this disk is not shared between multiple machines you know in memory locks are enough right. Assuming that all accesses to the disk are going through this OS layer, the OS is using in-memory locks, two synchronize accesses through the disk. So, in memory locks are enough all right.

So, that is fine. Now, let us look at, but now we are not happy with a global lock. So, we need per file logs, per inode logs, per block per data block logs or something of this of that sort alright ok. So, how can this be done? Well, you will want to have one log per every data block on the disk somehow alright.

(Refer Slide Time: 14:01)



So, one way to think about it is that. Let us say this is my OS and this is my disk. All accesses to the disk go through the OS and the OS inside it maintains what is called a buffer cache you seen this before right. So, a thread cannot read or write to the disk directly. The thread can make a request for a block to get read into the buffer cache and then you at synchronize over the buffer cache right.

So, you can only read or write blocks that are present in the in the buffer cache. So, if you want to read something you will first bring it into the buffer cache and then read data from it, if you want to write something you will first bring data from first bring it into the buffer cache then write to it and you may want to implement write through or write back cache that is a different matter.

But, in any case every you know you have if you have started the disk blocks as a buffer cache and, now you need to synchronize on the buffer cache instead of synchronizing on the data disk blocks themselves because you can nobody can actually touch the data blocks directly. You can only touch the data block by bringing the data into a buffer inside the buffer cache and then you synchronize over the buffer cache alright.

So, basically you have a buffer cache which is you know some set of buffers let us say it is implemented as a list and each buffer of buffers and each buffer has a lock per buffer. So, if I want to let us say create a file into a directory. I need to make four operations.

What I will probably want to do is I will first lock let us say let us look at this let us look at this. I need to do four operations. These are four different sectors, four different blocks. So, I will probably want to lock all of them and then do this operation. Recall fine.

So, basically, we are moving from coarse grained locking to fine grained locking there is an operation that I want to perform. This operation needs to be atomic with respect to all the other threads that may be doing this operation. So, recall that the way to do it is basically if fine grained locking, identify all the data all the data items or datums that need to be touched and get all those locks a priori and then do your operation right.

So, that is what I am going to do get these locks. So, get an get a lock for the buff for the sector or for the buffer that stores the inode, get a lock for the buffer that stores the data block of the directory that contains the directory entry and so on and then release those locks. So, that is fine grained locking. How do I implement these locks? Should these be spin locks?

Student: No.

Do you think it is a good idea to have them as spin locks?

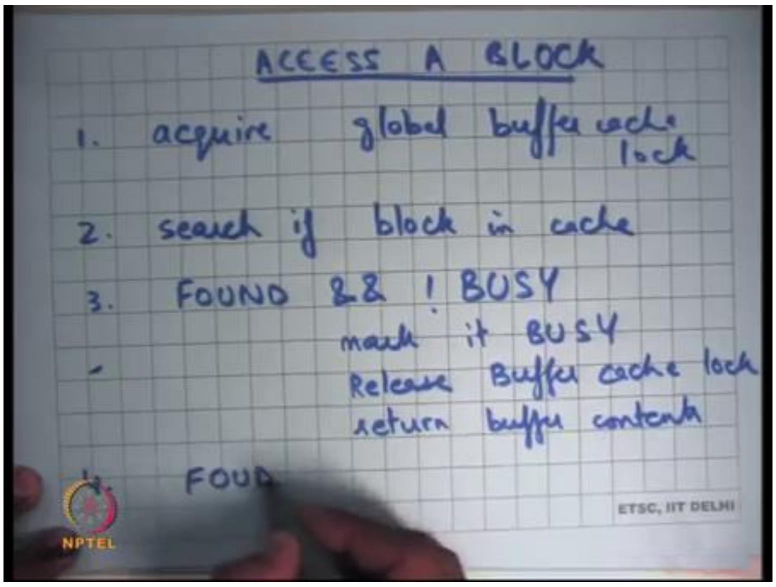
Student: Blocking lock.

They should probably be blocking locks right I mean if you have spin locks then if I am waiting for some buffer to get free or some buffer to get released, then I have to just I want to sleep because these my critical sections are likely to be very large they involved disk accesses milliseconds of time right. So, you want blocking locks alright.

So, one way to implement blocking locks is to just use a bit called busy bit in the buffer structure to indicate whether this bit this buffer is locked or not. So, right and use a buffer cache lock a global buffer cache lock to synchronize accesses to this busy bit per buffer right.

So, the idea is that I am going to take one global buffer cache lock and then to do mutual exclusion over these busy bits. And, then once I have locked so, these busy bits are going to act as per buffer locks and then once I have taken the lock then I am going to release the global buffer cache lock right. So, the global buffer cache lock is a spin lock that allows you to implement these blocking locks via these busy bits right.

(Refer Slide Time: 18:43)



ACCESS A BLOCK

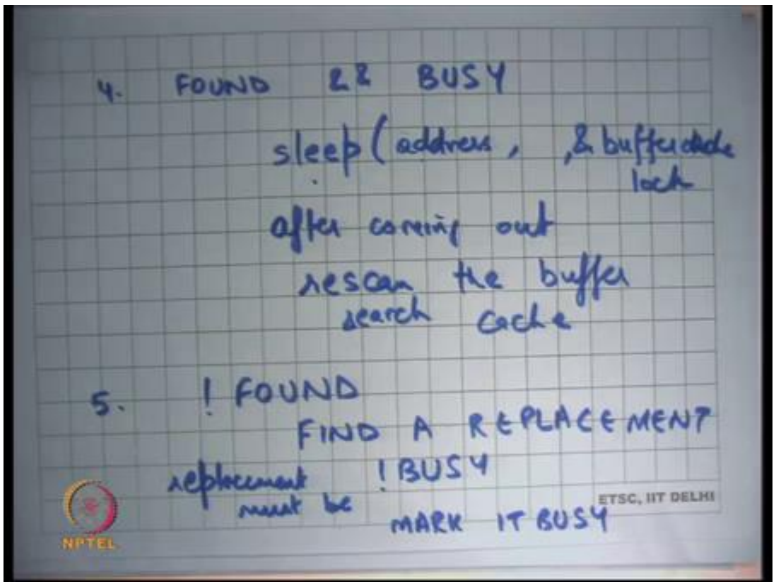
1. acquire global buffer cache lock
2. search if block in cache
3. FOUND && ! BUSY
   - mark it BUSY
   - Release Buffer cache lock
   - return buffer contents
4. FOU

ETSC, IIT DELHI
NPTEL

So, a typical operation would be that let us say I want to access a block. I will first acquire global buffer cache lock alright. Search if block in cache right. So, this is likely to be a fast operation. All you need to do is memory accesses to figure out whether the block that you are looking for is already in cache or not alright.

So, what are the; what are the few options? Let us say found and not busy right. This is the best case I found it in the cache, and it is not currently busy mark it busy right. Release buffer cache lock that is a global buffer cache lock and return the buffer right. So, notice I am using the buffer cache lock to provide mutual exclusion over my index structure and to provide mutual exclusion to accesses to the busy bit alright, that is all alright. So, now, what is the other case?

(Refer Slide Time: 20:25)


4. FOUND && BUSY

sleep(address, ,&buffercache lock

after coming out

rescan the buffer search cache

5. ! FOUND

FIND A REPLACEMENT

replacement must be !BUSY

MARK IT BUSY

Let us say I found right. Let me write it on the different sheet let us say found and busy all right. So, I acquired the buffer cache lock I went through the index and I found it to be busy right. Recall that these busy bits are being mutually exclusive protected against a concurrent accesses by the buffer cache locks.

So, you have been you will either find it busy or you will not find it busy, it is not like their concurrent accesses happening to busy bits. So, it is found and busy and then what you need to do?

Student: Sleep.

You want to sleep right on xv6 you want to sleep that is block or sleep right. So, and you also want to sleep and the release the buffer cache lock right that is the second argument of the sleep. What is the first argument of sleep? You just need to provide some channel on which you are going to sleep.

So, let us say the sleep is let us say buffer address or address right. So, some channel which make sense in this case you basically saying I want to wait for this block to get freed. So, let us just wait on the channel which is then address of this block let us say this block address of this block.

And, and so, at some point, so, while you sleep you also release the buffer cache lock. So, now, other threads can be doing their operations on it and at some points somebody is going to mark this particular buffer as not busy and it is that threads responsibility to call wake up on that particular address and so, I am going to come out of sleep. So, after coming out I can be sure that I have reacquired the buffer cache lock and what should I do this time?

Student: Again, you check.

Again, go to the step one and check if the buffer still exists right. The moment I release the buffer cache lock I have not I cannot assume anything about this state of the buffer cache of on reacquisition of the lock. It is possible that after I release the buffer cache lock somebody came in and not only marked it not busy, but also evicted it from the cache right.

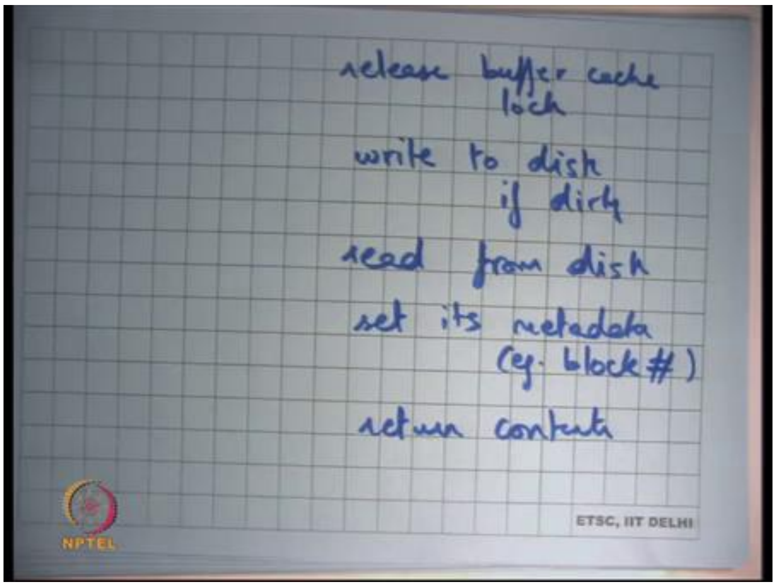
So, after coming out rescan the buffer cache or research alright and do the same thing. And, let us see what is the 5th case or what is the third case, not found in which case you will just read you will find a replacement buffer right.

So, once again you are going to find the replacement with the buffer cache lock held. So, you can be sure that there is not so, that is guaranteeing mutual exclusion. So, recall that all the in-memory data structures are being protected by this global buffer cache lock, but it is the individual buffers that are getting protected by fine grained locks. So, you find a replacement holding the buffer cache lock.

And, what so, firstly, you and make sure that the replacement should be not busy right. So, replacement must be not busy; if it is busy somebody is using that block so, I cannot just replace it right. So, the busy is also guaranteeing making sure that you know while you are doing something on while you are accessing a buffer while some other thread is accessing the buffer nobody else can actually evict it right. So, make sure that is not busy.

Once you have found the replacement you mark it busy. You mark it busy because now you have decided that you are going to operate on this particular block and what you are going to do.

(Refer Slide Time: 24:33)



release buffer cache lock

write to disk if dirty

read from disk

set its metadata (eg. block #)

return contents

NPTEL
ETSC, IIT DELHI

So, once you have marked it busy at this point you can release the buffer cache lock right. So, basically it is you have to use the global lock to get the fine grained lock and then once you got the fine grained lock which basically means you marked it busy you release the global lock right. Now, you have the busy bit set so, nobody else can touch it till you make it not busy right.

And, now while you are holding the busy bit on that block you can write it to disk assuming it is dirty; so, if dirty read from disk the new block that you wanted to read you know set its meta data, example block number alright and return the contents right. So, I am using so, here is a very good example where fine-grained locking is absolutely necessary to have any kind of performance.

And, you are using a global lock to get these fine-grained locks. You do not want these fine grained locks to be spin locks you want these fine grained locks to be blocking locks, and so, to ensure to do that you need a global spin lock to synchronize accesses to these blocking locks and these fine grained locks are basically making sure that while you are working on something it does not get evicted nobody else touches it while you are working on it alright. So, but what is the problem with fine grained locks?

Student: Order.

There is one problem right we all know.

Student: Order.

Deadlocks, ordering. So, deadlocks. So, anytime you have fine grained locks there should be an alarm bell that you know I have to worry about deadlocks. So, what can what are some bad things that can happen? Now, let us just take this example. Let us say somebody is creating a file he allocates an inode.

So, he takes a lock on the block that contains the inode right and then he adds a directory entry record then he tries to take a lock on the directory entry record. So, first it takes a lock on the inode, then he takes a lock on the directory entry lock. If there is some other thread let us say unlink that first takes a lock on the directory entry record and then takes a lock on the inode then there is a possible deadlock right.

Student: (Refer Time: 27:12).

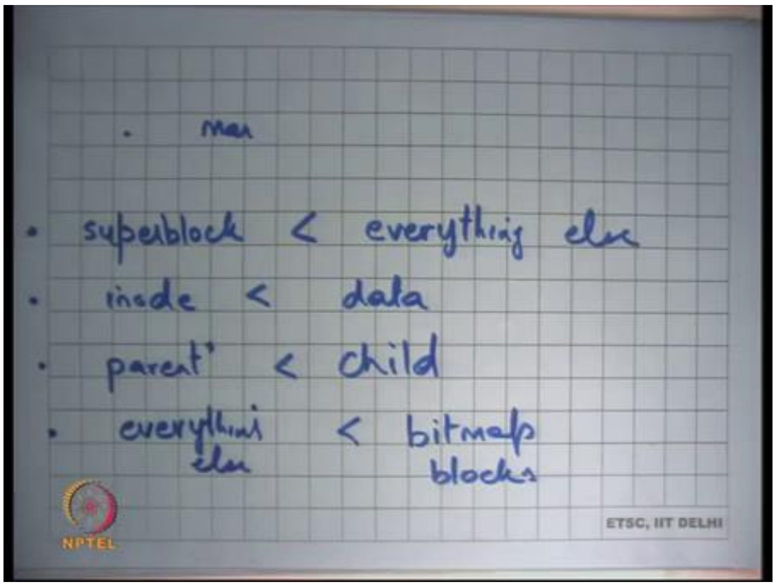
Well, I mean if you want this entire creation to be atomic you would want to hold both locks at the same time right. Otherwise what will happen is if you release the lock at this point; so, you allocate an inode and then you release the lock in the middle some other thread comes in so, it is going to see some inconsistency in your data structure, in your file structure right.

So, if you want that you know this entire creation. So, it is possible that the entire creation need not be atomic to ensure that inconsistency do not people no thread can see inconsistent data structure. So, there are some invariances are having to be maintained right. So, in general when you are talking about concurrent access and synchronization you basically first want to think about what are the invariants that need to be maintained.

And, the invariant typically looks like this: if I have been able to acquire the lock then I can assume that the data structure looks like this right. For example, if I have been able to acquire a lock on the tree then the tree must be a tree it should not have any cycles right or there should not be any dangling pointers and things like that right.

So, if you release the lock somewhere here then you are releasing the lock at an inconsistent state. So, the idea should be that you take a lock, and then you perform some operations and only release the lock when you leave the system in consistent state all right. So, you take two locks here let us say to make sure that and so, they can be deadlocks right.

(Refer Slide Time: 28:55)


- max
- superblock < everything else
- inode < data
- parent' < child
- everything else < bitmap blocks

ETSC, IIT DELHI
NPTEL

So, you need some kind of global orderings right. So, some orderings that that xv6 follows and the this is there is something that I was just looking through is that you know. If there is some operation that needs to touch the super block, then mark super block first right. So, let me just do ordering.

So, let us say super block before anything else before everything else right. So, this the less than sign basically says if you want this operation to do something on the super block sector then it should be taken first right ok, then inode before data right.

So, if you want to do something on the inode block and the corresponding data blocks, we will first take a block lock on the inode and then take a lock on the data and you the programmer needs to be careful about doing this right.

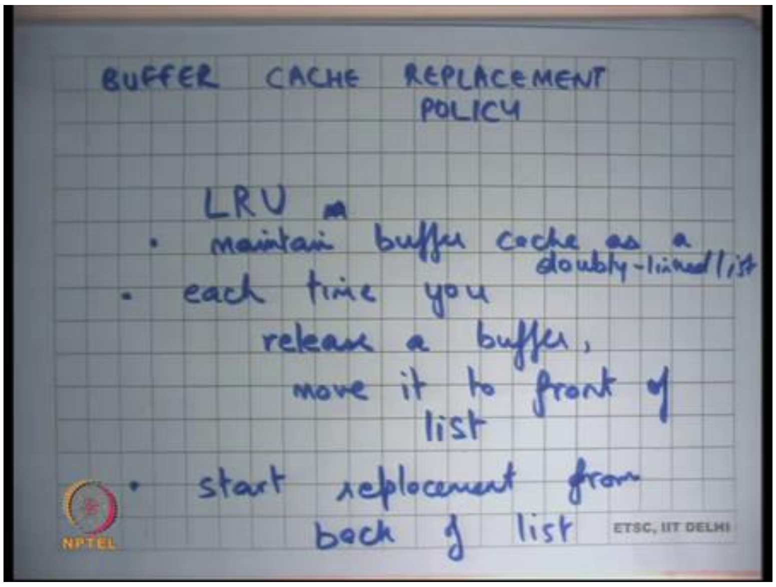
Let me look at some example. For example, here file unlink is an example where you first write to the data blocks and then you write to the inode, but locking should be in the opposite order alright otherwise you know you have deadlocks. So, programmer needs to be careful all right.

Then, parents parent before child, right. So, not only does do you have a hierarchy in the file system metadata, there are inodes then there are data blocks you also have a hierarchy in the directory structure right. So, there are parents inodes and then there are child's inodes. So, parent directory inode and the child directories inodes; parent directories blocks, child directories blocks and so on right.

So, there needs to be some order on that as well. And, let us say everything so, bitmap blocks. So, nothing after bitmap blocks, so, everything else before bitmap blocks right. So, you should not hold a bitmap block and then try to hold a lock on something else right.

So, just a very good example of you know of a relatively complex system where you have fine grained locks and then you need to have some global order and there are lots of different types of entities and you need to have some global order on these different types of entities I mean. So, this is something I mean these are some invariants that xv6 could follows. You can check it for your for yourself you know if there are any if they are more, they are less let us find out let us know alright.

(Refer Slide Time: 31:41)


BUFFER CACHE REPLACEMENT POLICY

LRU

- maintain buffer cache as a doubly-linked list
- each time you release a buffer, move it to front of list
- start replacement from back of list

ETSC, IIT DELHI
NPTEL

Finally, let us look at the cache replacement policy. So, recall that unlike the virtual memory subsystem, where you were really cared about the speed of the hit path right; the hit path was a memory access and you did not want anything any overhead on the hit path. The maximum you tolerated was setting up of a setting a bit, the access bit, and that too was done by the hardware. In case of file system, you have more flexibility.

Assuming that the file system is being accessed by a system calls like read and write, there is anyways a lot of overhead of taking a trap, making function calls, and potentially even making a disk access. So, the whole path of the hit path itself is quite expensive. So, you can do much more than just one bit for the access bit right.

So, in fact, the xv6 file system implements LRU for the buffer cache and the this is the way it works. Each time you release a buffer so, maintain how is it implemented maintain buffer cache as a list as a doubly linked list all right and each time you instead of access I am going to say release a buffer.

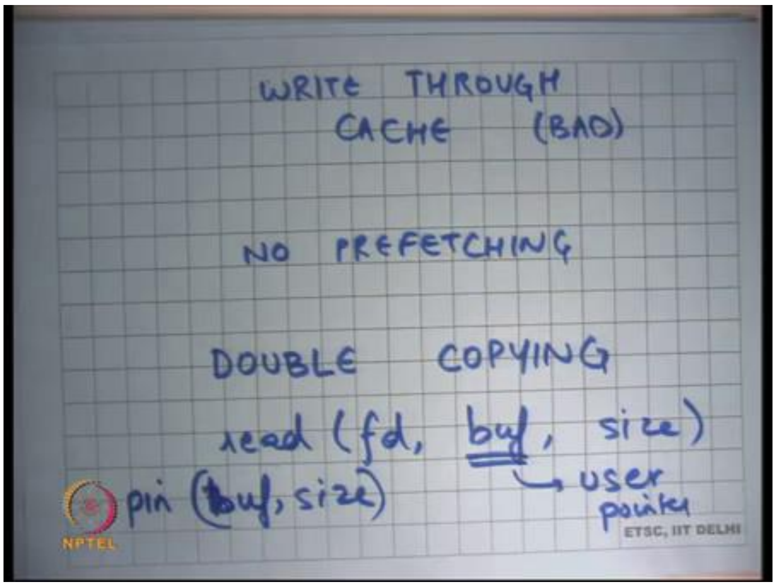
So, at the time of access you do you know when you first access the buffer you mark it busy. As long as it is busy it cannot be replaced, so, you do not need to do anything at the time of access. At the time of release that is the; that is the time you can basically say that I want to do something about this buffer to make sure that it gets released, it gets evicted the last because I have just accessed it. So, it is most recent access. So, it should be released the last.

So, each time you release a buffer move it to front of list all these doubly-linked list right and start anytime you want to replace you start replacement from back of list right and that is LRU for you right.

So, each time you access something you just move it to the front of list, you want to start replacement you start from the back of list. Recall that when you start replacement you look at the last you will look at the first you look at the last element and you get the first one starting at from the last that is not busy that is the one you will going to release. Anything that is busy is currently getting accessed.

So, the you know that is not a good candid for replacement any ways alright. So, that is LRU cache replacement for the buffer cache in xv6 alright.

(Refer Slide Time: 34:53)



WRITE THROUGH
CACHE (BAD)

NO PREFETCHING

DOUBLE COPYING

read (fd, buf, size)

pin (buf, size)

user pointer
ETSC, IIT DELHI
NPTEL

Rather write through cache ok. So, write through cache is there for simplicity reasons, but let us look at is this the great policy. It is a very it is actually a poor policy from a performance standpoint each time you write something to a block it goes all the way to disk. Even if you overwrite some things a 100 times, there are 100 write through the disk. If it was write back policy, then you could have absorbed 99 of those writes in memory and just given 1 write to the disk right.

So, we all know that write back has this nice thing that it absorbs multiple writes to the same data block. More importantly given the physical characteristics of a magnetic disk write back cache is a much better option than write through cache right.

Write through cache basically means that the disk write is on the head path or on the critical path on the and so and you can only write you basically have only one outstanding IO or very small number of outstanding IOs under time. With write back cache you can batch many writes together and that way you can get much more efficient utilization of the disk bandwidth right.

Recall that elevator scheduler inside the disk controller so, if you give lots of things together that is the much better option right. So, disk a write through write back cache is better not just is even more relevant in the case of a magnetic disk because of because right batches are batching rights is a good optimization right.

Secondly, it has no prefetching let us see what it means. So, let us just say this is bad right a real file system would probably not want to write through cache it will want to write back cache and we will look at some issues there.

Secondly, the xv6 has no prefetching, what does it mean? Quite likely most many workloads have a very sequential behavior sequential varies. So, let us say I do I execute a grep command. Grep commands just going to go through the file in sequential order from byte 0 to byte last right.

And, so there is a very sequential thing let us say I implement I execute grep on top of the xv6 file system, what is going to happen is it is going to ask for block 0. The block 0 is going to get red and the by the time I actually start processing the block 0 the disk is already rotating. So, block one assuming that the disk was contiguously laid out on disk or the file was contiguously laid out on disk you are basically wasting rotations right.

If the disk if the file system could instead say I want block 0 through 10, then all those 0 through 10 could have been red in one go. But, now if you are going to say is 0, then 1, then 2, then each time you read a block you waste a full rotation before you get to one than you waste the full rotation you get to 2 and so on right. So, it is much more it is milli seconds more expensive than what could have been done if you had prefetching right. So, it is a so, that is the; that is the thing.

And, let us see there are actually double copies right. So, let us there is another fact about the file system that we have as we have seen it so far, that is double copying. What do I mean? If an application says I want to read sector x, then first the sector x gets copied from the disk device to the buffer cache.

The buffer cache is in kernel data structure, the user cannot see it directly and then from the buffer cache to the users address space right. So, recall that as user will probably want to say something like read fd, buf, size, right? And this buff will be a user pointer right.

To implement this basically the file system has to first read from disk to the buffer cache and then from buffer cache to the user address space. Is that a big problem? Well, assuming for something like a magnetic disk it is not a problem because the disk access itself is the bigger bottleneck right the memory you are copying is not that big a bottleneck.

But, let us say you were running on a very fast network device right then these kind of double copying cause actually do become bottlenecks right. So, you know think about giga 10 gigabyte networking. So, if this file descriptor was not pointing to a file on disk it as pointing to some network device and you wanted to read data from the network device at a very fast speed, then your first copying it to the buffer cache and then copying it to the user space.

Could you have done better? Well, one way to do this is let us say you can say that when the user says read fd, buff I take this pointer and I pass it all the way to the disk driver right. Recall that the pointer that I give to the disk driver is that of the buffer cache and that is having this problem of double copying.

The other option could have been let us take the pointer from the user converted it into its physical address and then pass it all the way to the disk driver and so, the disk driver directly writes to the user mapped memory and then I just written and so, there is just one copy. The disk driver directly writes to the user memory. What are some things I need to be careful about? Recall that the virtual memories of system do swapping right. So, it can actually swap out data or swap out address space regions from the user space. So, I need we need to make sure that this region pointed to by buff till size cannot be swapped out.

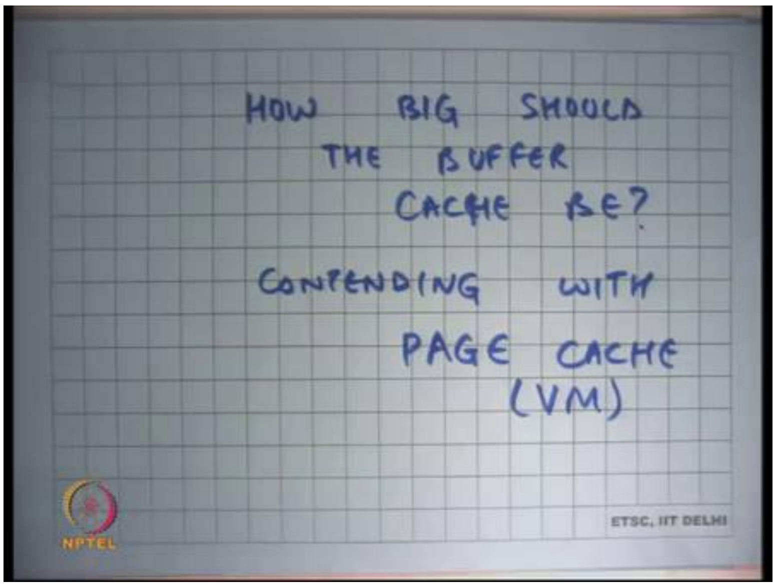
So, I need to lock it in memory right in all you know the term used is basically pin it in memory right. So, you pin them in memory; you basically do not allow it to go through disk swap do not allow that to get to disk number 1.

But, more importantly you disallow so, this read if there was locality of accesses by multiple processes then you also getting rid of you know you know basically also hampering locality. So, if this one process temporal locality is not getting properly address because buffer cache has it is nice property that it is a sheared cache between multiple processes.

Now, if you bypass the buffer cache and directly start reading to user address spaces if some other process also wants to read the same data then he will have to make another disk access. So, you cannot reuse the cache right. So, in some situations you want to avoid double copying and these situations are situations where you are you care about very high IO bandwidths.

Think about network switches, network routers etcetera and you do not expect that much temporal locality of access across multiple processes right. On the other hands, there is other cases where you know double copying may be a reasonable overhead to take given its advantages alright.

(Refer Slide Time: 42:33)



HOW BIG SHOULD THE BUFFER CACHE BE?

CONTENDING WITH PAGE CACHE (VM)

And, finally how big should the buffer cache be? What is the tradeoff? Can I if I give the entire memory to the buffer cache what happens? I will have no memory for my virtual memory subsystem you know the process. So, recall that the memory is being used for two purposes as you have discussed it so far. One was to act as a cache for the virtual address space and another as a cache for the file system right.

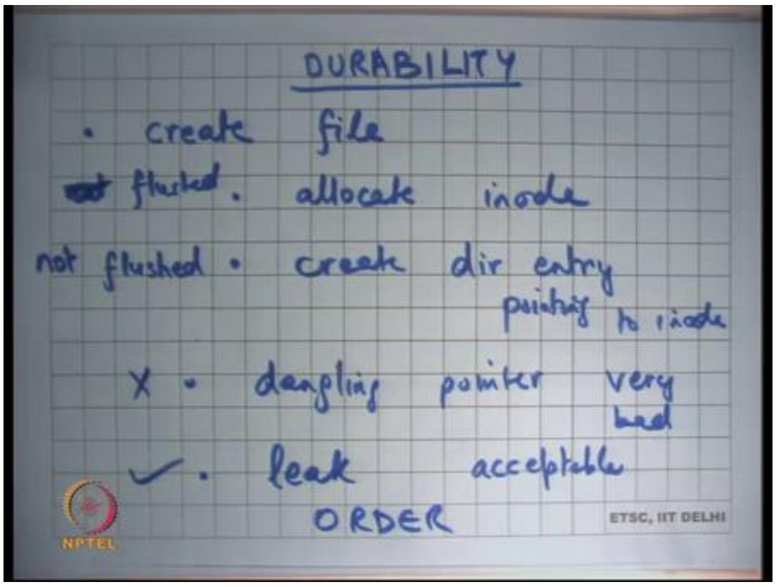
So, there are two caches in the memory one as a cache of the virtual memory address space and another as a cache to the file system and these are completely disjoint right. One is caching swap space data and memory mapped file data etcetera, and another is caching file data. And, both are completely different semantics also, and also different access patterns and replacement policies right and performance tradeoffs more importantly.

So, you are basically the so, the OS needs to decide how big should the buffer cache be and if it makes it big then there is little. So, it is contending with page cache right from the VM subsystem. And, so this decision can be made statistically. You can say I will have half of my memory for buffer cache and rest half for the VM page cache or you know, or you could do dynamic thing based on usage patterns.

So, basically you want to see if there is a lot of file system operations happening then you want to make the buffer cache larger. On the other hand, if there are not that many page faults happening so, you want to make buffer cache larger. On the other hand, if there are lots of page fault happening not that many buffer cache miss is happening then you want to make a VM cache larger.

And, you could do some coarse-grained dynamic tuning to figure this out right, but of course, xv6 just uses the static size (Refer Time: 44:53) ok.

(Refer Slide Time: 44:59)



DURABILITY

- create file

~~not~~ flushed • allocate inode

not flushed • create dir entry pointing to inode

X • dangling pointer very bad

✓ • leak acceptable

ORDER

ETSC, IIT DELHI

Now, let us look at durability. So, one of the most important and interesting aspects of file system design is that state should not become inconsistent across power failures alright. So, let us see what can happen. Let us say I wanted to create a file and let us say I wanted to and to create a file I needed to make at least four disk writes and these are the four disk writes and at this point after I made the first disk write power caches, this is the power caches right.

Recall that all your busy bits etcetera was only in memory data structure and they were only there to protect against concurrent accesses by other threads. But now if there is a power failure then you have left the file system in an inconsistent state irrespective whether you use locks or not whether you use global locks or not, it does not matter.

The file system is still in an in inconsistent state because then it comes back again you see that an inode has been allocated, but it is not mark pointing to anything right. worst ok. So, let us see what are some bad things that can happen. Let us say I am creating a file. There are basically you know at the coarse level there are two things I need to do. I need to allocate inode and I need to create directory entry right which points to inode.

Well, let us say a power failure happens and I have created the directory entry pointing to inode and I. So, I have let us say this has been flushed to disk and this has not been flushed to disk. So, I did I did both these operations, but I did this let us say first from a disk standpoint. So, I first created the directory entry and, now I was going to create allocate an inode.

But before I could allocate an inode the power went off. Power comes back again, what do you have? You have a directory entry pointing to some dangling inode, dangling pointer right. This is the bad thing. Why is it bad?

For example, let us say the inode that it is pointing to belong to some other user. So, now, I can potentially read the contents of the other users file. So, it is a security flaw from that standpoint right or it contains some garbage data which makes me feel that the file is actually really large right.

So, basically, I have a dangling pointer and the dangling pointer is pointing to some garbage and this garbage could be security sensitive data it could be something that come very badly confuse me right.

Let us see if the other thing happens. Let us say this gets flushed and this does not get flushed. So, firstly, you know. So, either of these things can happen. So, let us say I have allocated an inode all right, but I have not yet created an entry for it in the directory and the power goes down. Power comes back again what is the inconsistency in my system?

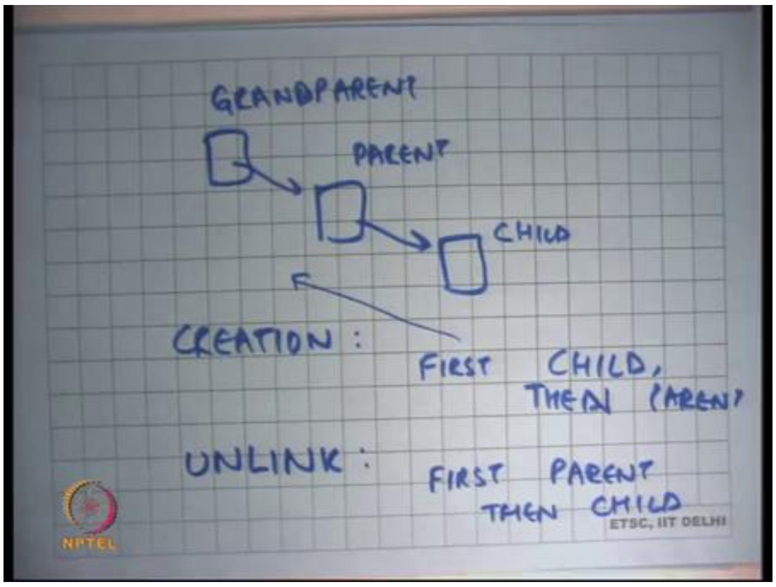
The inconsistency is that I have some node that is not free that is allocated, yet I have nobody pointing to it right. So, that is not as bad as a dangling pointer right. In the first case I had a dangling pointer and this we said was very bad and the in the other case we have a leak.

Why do I call it a leak? Basically, there is some data that is not usable anywhere or some blocks. So, the inode block is actually marked allocated, but it is not being actually be used in the file system and so, it is a leak. It is a space leak right. I will not be able to use it for anything else any longer right. So, this is acceptable.

Now, you have to so, one of these; one of these two things have to happen. So, to make this acceptable, one way to do this is to order the writes right. You say that whenever you create a file, I am going to first allocate an inode, flush it to disk and then create a directory entry.

If I follow obey this ordering then I can be sure then there will be never ending you know this is disallowed and this is possible, but this is acceptable. So, only acceptable things are possible alright. So, you order the writes to disk. Assuming let us assume a right through cache for the time beings. So, you would basically say that I want to create something I am going to first.

(Refer Slide Time: 50:13)



So, in general if there is a data structure that has something like this then at creation time. So, let us say this is you know all these are parent child relationships. So, this is let us say child, this is parent, and this is grandparent right. So, there is parent-child relationship between them.

In case of a directory and subdirectory it is a parent-child relationship between parent subdirectory and directory and subdirectory. In case of a file it is a there is a parent-child relationship between the inode and the data block; inode being the parent, data block being the child right. In the case of doubly indirect blocks, there is a 3-level hierarchy.

The top level inode, the indirect block and the data block and so on right. So, there is some parent-child hierarchy and at the creation at the creation time you are going to move in this direction right; first child, then parent right. So, that is going to ensure that you can only have leaks, will never have dangling pointers.

At the time of unlink or remove, what? In the opposite direction, first parent, then child right. So, if I want to remove something, let us say I wanted to remove this. So, I am going to first update the parent to say that you know there is no pointer here. So, I am going to first remove this pointer and at this point if there is a cache then I have a leak that child's data blocks are not getting are not accessible anymore, but there is no dangling pointer right. On the other hand, if I had first then a I had first an allocated a child then and there is a power cache after that then the parent would have had a dangling pointer alright.

Let us continue this discussion on durability further in the next lecture.