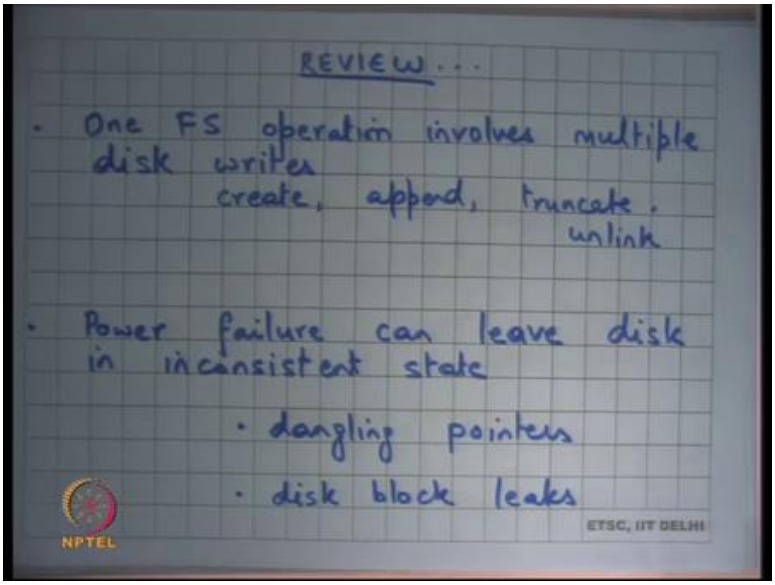
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 34
Crash Recovery and Logging

Welcome to Operating Systems lecture 34.

(Refer Slide Time: 00:28)



So, we were discussing file systems and we saw that one file system operation typically involves multiple disk writes right. So, examples being create of a file you need to write to the parent directory, you need to write to the inode, you need to write to the data blocks right. Similarly append to a file you need to write to the data blocks, you need to write to the index, you need to write to the inode.

Truncate say same thing, but this time you are removing blocks from the file. Appending was adding files to the blocks to the file; truncate is just removing files blocks from the file. So, let us you truncate just removes the last few blocks from the file and unlink similar thing you just removing a file from a directory and once again all these operations will involve multiple disk writes typically 4 to 8 disk writes per operation.

And, the problem we were looking at last time was what happens if there is a power failure in the middle of one operation right and so the problem is that leaves the disk in

an inconsistent state and that can result in many bad things to happen. So, 2 and you can sort of categorize them into 2 types; One is dangling pointers where you have a directory pointing to a file which has not which did not get committed to disk which did not get read into disk and so the directory is pointing to some free disk block right.

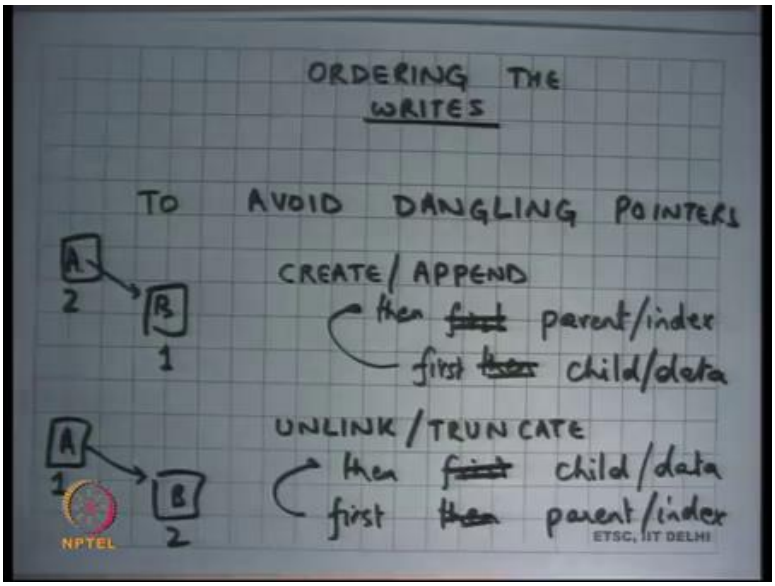
So, that is a possibility and that is a very dangerous thing. Firstly, because you know when the when power comes back on the there is no way to figure out whether this is a dangling pointer or whether this a real pointer right. So, if that dangling pointer is actually pointing to some block that looks like an inode, suddenly the user will have access to this particular file, that it was not supposed to have access to right. Any logic that dependent on the contents of that file for the users program it is going to cause crashes more you know what worse that can happen is the user can gain access into somebody else's file right.

So, dangling pointers are a very bad thing and you would ideally want to avoid dangling pointers. The other problem that can happen is disk block leaks right. Here what can happen is that you basically initialize some data and you are about to create a pointer to this data from the index. Let us say you appending to a file and you created some blocks at the end and now you wanted to change the index in the inode and make point to those blocks and there was power failure in the middle.

So, what is happened is you have initialized some blocks you have removed those blocks from the free list, but you have not really created pointers to them before the power failure happened right. And, so when the power comes back again what you want to find out is that there are some blocks that are neither present in the free list and nor are they part of your directory tree or your file tree right.

And, so that is a bad thing also because there these disk blocks that will never be used and if you keep doing this, if you just keep you know having power failures unannounced power failures then eventually you will you can have lots of leaks lots of space in the disk that can remain unused forever. But, in any case disk block leaks are less dangerous than dangling pointers it is not a correctness problem it is a performance problem right. So, the solution you are looking at last time was let the file system order the rights in a way such that it avoids dangling pointers.

(Refer Slide Time: 04:12)



So, we say you know one of these has to happen you know after all there are multiple disk writes are happening and you know I cannot make them atomic with respect to power failures power can go get out any time. So, I have choice between dangling pointers and disk block leaks, and I will choose disk block leaks over dangling pointers.

And so let so basically, the idea is that I will order the rights to the disk such as the I will avoid the dangling pointers and which basically means that when you are adding something for example, I am creating a file I will first create the file or I will first create that initialize the data blocks and then create a pointer into the index right.

So, this way there will never be a dangling pointer because the index pointer will only be created after the disk data block has been initialized. In general, whenever there is a there is a data structure like this where block A is pointing to block B then you will first do this and then do this.

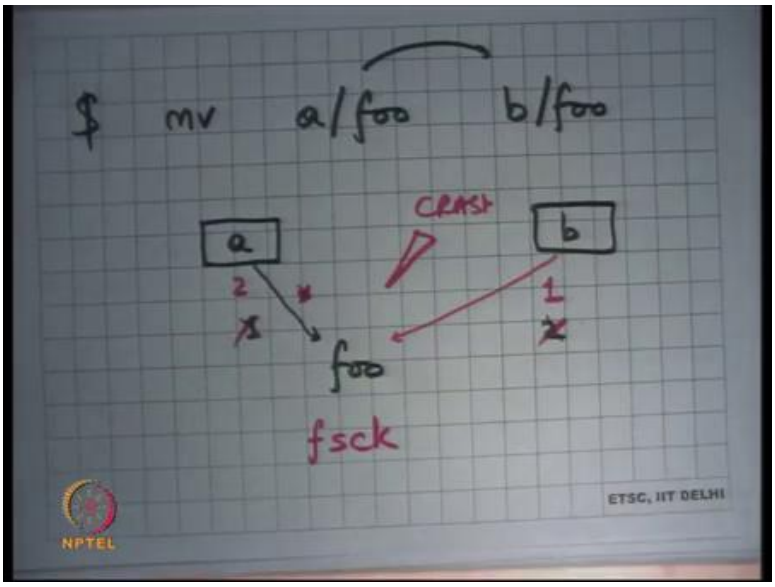
Student: Create an opportunity.

Because I am talking about create. So, you will first create the block B and then add an entry for it in the index A. So, you will first do this and then do this for create. On the other hand, when you were doing unlink for truncate when you are removing things you first want to remove it from the index and then deallocate or you know free or

uninitialized these blocks right. So, if I am doing unlink or truncate and I have a structure like this if I deallocate this first then I have a dangling pointer.

So, it is better to first write to A to remove that pointer and then deallocate B right. So, you will first do A and then B in this case right. So, first I have written it in the other way. So, yeah this is the mistake. So, first child or data and then parent or index and first parent and then child or data right; so, that is alright ok.

(Refer Slide Time: 06:25)



So, that is you know, but in all these discussion I am assuming that all operations have to happen synchronously which means I am not I am assuming the write through behavior right. I am saying that if I make an operation it should go to the disk because after all immediately because if I am writing my code I am saying update the index then update the data the file then and I wanted to happen synchronously because if it is not happening synchronously and if I am using a write back cache then even though I did these operations in a certain order what will really matter is in which order was the were these data blocks flushed to the disk right.

So, it can happen that in the cache I did this first and then this later, but when it was actually flushed to disk this happened before this right. So, all you know there is no use for ordering if there is a cache. So, in general you know doing sync writes is not very performant, but it has the nice safety property that you can order things too avoid

dangling pointers and this was indeed used for some years in many operating systems in the early days right.

But we know that this has a big performance penalty but let us also look at another thing is it always possible to do this ordering. So, let us say I had an operation which said move a file foo from directory a to directory b right. So, let us say I wanted to do this and so this is let us say the rename operation and I wanted this to be atomic with respect to power failures or I wanted to make sure that things are safe right.

So, the here in this case there are two things that need to happen. Let us say this is a and this is b these are directories earlier a was pointing to foo and now you want to remove the pointer from foo and create a pointer to foo from here B right. So, now, in this case you may you know you may want to say what should be done first should I remove this pointer first and then create this pointer or should I first create this pointer and then remove this pointer right. So, there are two options here I can either. So, I can either do this first and this second or I can do this first and this second.

In both cases my file system is on a reboot can appear in a very inconsistent state. So, it is no longer the case that one of them will only result in a leak if I do this first. So, let us say I do this first and the power failure happens before I had deleted the second pointer I have situation where one file is being pointed to by multiple directories right and this may not be acceptable and there is no way an operating. So, depending on the file system semantics they may or may not be acceptable. So, let us assume that it is not acceptable and there is no way that an the file system can automatically figure out what to do right.

So, when it comes back again and it figures out there is something wrong there it will probably want to say I either want to remove this or I want to remove this, but has no way to figure out which one to remove. So, at that point you know there are couple of options either you ask the user look I see some inconsistency a file seems to be part of 2 directories or a data block seems to be part of 2 files you know it is similar then what you want to do.

So, those are options, and these are the things that you may have seen in programs like fs check right. So, there are these programs that run on you know if you have not mounted your disk cleanly then when you bring it bring it back again there is a program that will do a global file system traversal and figure out if there are any inconsistencies and the

advantage that ordering gives you is that you have limited the inconsistency to only certain types.

In this case perhaps it makes more sense to do this first before that right because at least you are not losing data you have some inconsistency in your state, but at least you are not losing data if you had done if you had deleted first and then created then you would have lost some data right. So, you have to basically you have to do some ordering and depending on what ordering you are doing you have guaranteed now the kind of you have limited the types of inconsistencies that can happen on a power failure.

And, then you have a program that will that will be a long running program that will probably do a full file system traversal to figure out the inconsistencies and either fix them itself or ask the user to fix them for (Refer time: 11:21) yeah question.

Student: Sir, like in case the in case there is now in case this first and second both of you can (Refer time: 11:26) (Refer time: 11:27) and after that there was a power failure after that when the disk reboots how does it differentiate between whether or not the previous command was a mover whether it was something of the form of a short cut based or a.

Right. So, let us say I am let us say I am using this strategy were I am going to first create and then remove right and before I did a remove there was a power failure right. So, let us say there was a crash at this point and so I have a I have a file system state which has one file being pointed to by multiple directories. Now the question is how do I differentiate how do I know what was going on actually I have no idea right. So, there was no way to figure out what was going on at that time all you are going to do is you want to try to bring it in consistency state in some way.

Student: Sir no what I am saying is it might possible that this was a consistent state that there are 2 2 pointers to that like (Refer time: 12:15)

Yeah, I mean so your file system may actually; So, your file system may allow this kind of a thing and this may look like a consistent state in which case the file system may actually not throw any warning at all and it is for the user to actually now deal with things on it is own right. But, let us I mean assuming that the file system does not allow this kind of thing and this is this is actually a inconsistency from in the file systems for semantics then you know it can throw warning to the user alright ok.

So, basically the idea that I have discussed so far is that you gone to order the rights on the disk in a certain way to limit the types of inconsistency that can happen and then you are going to run a program in case of an ungraceful shut down when you come back again that that is program is going to do a global scan and it is going to figure out what inconsistency they and trying to resolve it either automatically or using users help alright.

So, this is being used for a long time in lots of different operating systems and perhaps you have seen this also in the regular operating systems, but, but this has a few problems. Firstly, as a disk size becomes really large the time it takes to run this fs check program which is a global scan of the structure becomes very large right. So, just to give you an example you know a random read.

Student: Sir

Yes question.

Student: Sir, but earlier we have said that we have 2 copies of the table file table.

Ok.

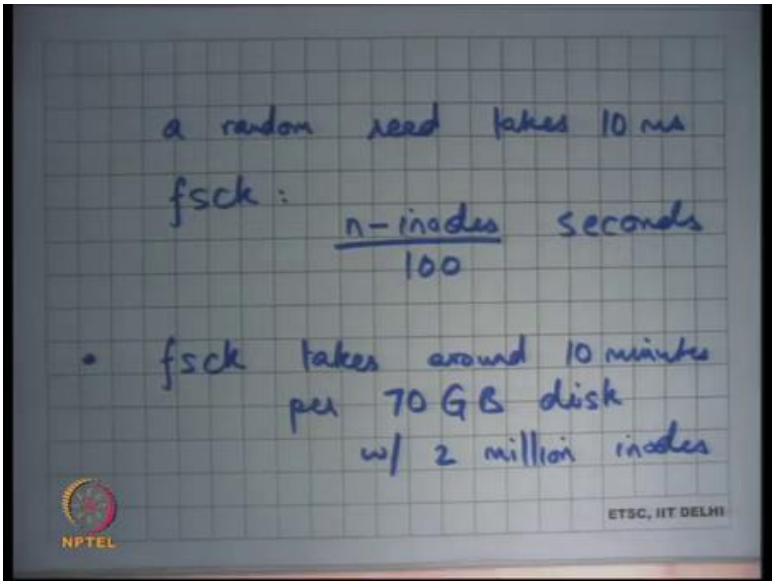
Student: File allocation table we said. So, cannot we use that to figure out what is that?

Alright. So, the question is that we have also said that for reliability we will duplicate state. Some state we will duplicate in the case of a file allocation table I will I will keep 2 tables right and so can that help in resolution of such conflicts see when we duplicate state we are basically doing it for reliability against stationary disk errors you know errors that can happen over years. And, things like that and you do not want duplicate state you know you would not deny a file system such that duplicate state basically involves updating twice for every update.

So, you know you would not design your file system typically to say that every time I make an update I will have 2 copies of every inode and every time I make an inode an update to the file I am going to write twice to 2 different inodes right. One could do that potentially and you know one of the solutions we are going to look at is similar in spirit, but, but that is not that is not a very performant design in the general case.

See basically whenever you are doing system design you basically want to say I want to speed up I want to make my general case as fast as possible and yet cover for the rear cases right you it is very it is a bad design to actually slow down your general case to take care of rear and if you are doing that then that is the you know that is basically an example of that alright.

(Refer Slide Time: 15:30)



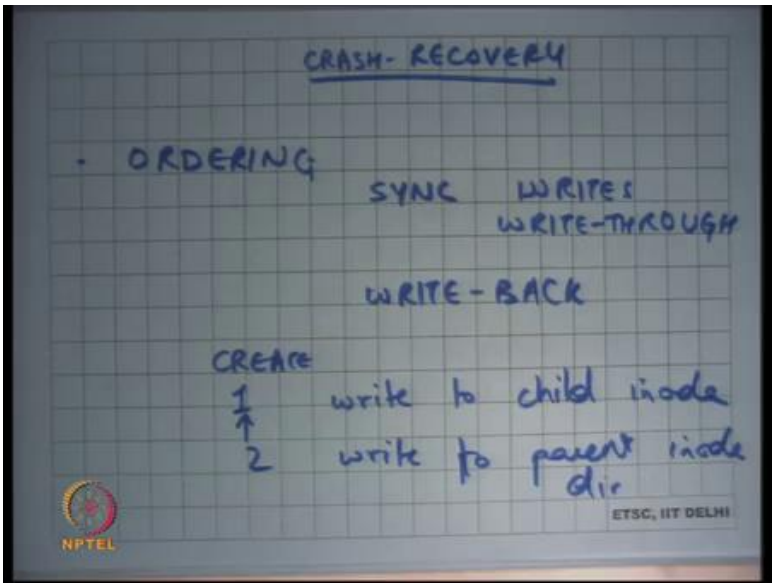
So, recall that a random read takes 10 milliseconds right and the. So, basically that means, that if I was to do fs check then the number of the time it will probably take me to do a full fs check would be somewhere like number of inodes which is representing of number of files in the system and let us say each inode is taking one random read.

So, you gone take n inodes by 100 seconds right; so, you can do a 100 inode reads per second. So, depending on the number of files you will do n inodes upon 100 seconds and that is really slow if you have thousands of files that is easily an hour write the as a data point fs check takes around 10 minutes per 70 GB disk with 2 million inodes alright.

So, clearly in doing so it is not it is instead doing it randomly read per inode the better thing to do would be if you are actually doing a full file system scan the better thing to do would be do an do an sequential read to read up all the inodes into memory and then do your traversal right, but even if you are doing that so this is the optimized statistic. So, even if you are doing that a disk which is roughly 100 gbs takes 10s of minutes right.

A disk that is terabytes will take hours and so on and so as the disk started to become larger and larger this idea of doing an fs check on a reboot started becoming less and less practical alright.

(Refer Slide Time: 17:21)



So, we so I am going discuss another method to be able to do this cache recovery, but before that let us also discuss. So, we have discussed one method which is ordering and with sync synchronous writes by synchronous writes I mean write through cache right.

So, whenever I write I write straight through disk there is no there is no write back going on. The other thing about synchronous write is it is very slow right. So, let us say I just image that you were to untar a tar file. So, and let us say the tar file has a 1,000 files inside it now just un taring a tar file requires a creation of 1,000 inodes and creation of each inode if it is a sync write it is going to take 10 milli seconds.

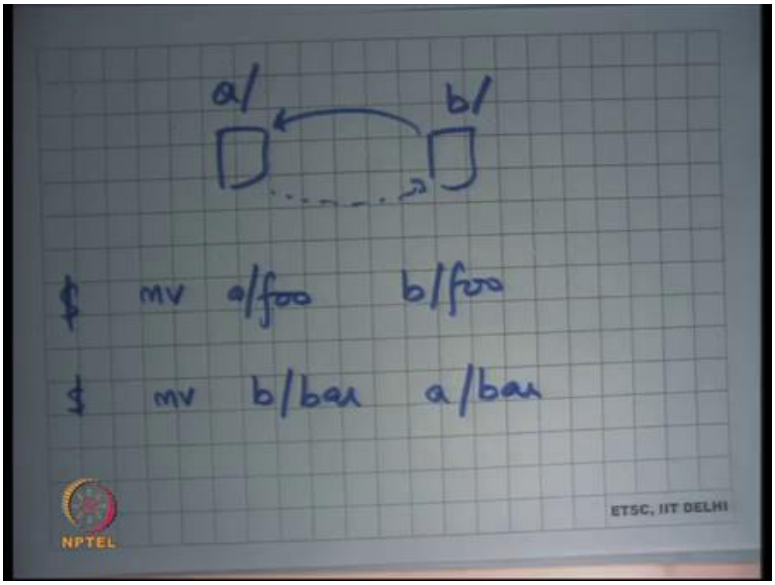
So, 1,000 nodes will take you know 10 seconds to create to untar one small a tar file which has a 1,000 files only right. So, sync write is not very practical. On the other hand, if you had a write back cache it would have done this whole untar operation in almost 0 milliseconds or less than you know less than a milli second basically.

So, you want write back and so the way this is done if you wanted to implement ordering with write back cache you would make the updates in the cache, but also store in the cache in memory ordering dependencies between disk blocks right. So, let us say I

wanted to do a create I will first write to child inode initialize it and then write to parent inode right parent directory inode let us say. So, let us say these are the 2 things had to do there few other things that that is to be done, but let us say just this 2 things to be done you have to write to the child inode and then you have to write to the parent inode.

So, what you will do is you will do these writes in the cache, but you will attach some ordering number to these. For example, you will say that this should be done there is an ordering dependency between the second block and the first block. So, the second block should be flushed only after the first block is flushed to disk. So, at the time of actually doing the replacement or flushing or writing it back in a bunch you are going to make sure that the things are ordered.

(Refer Slide Time: 19:55)



So, for example, you have multiple disk blocks let us say this is a's disk block and this is b's disk block and let us say a and b are directories then if I say move foo a/foo to b/foo. And, I want to make sure that creation happens before unlink then I will say that I will draw an edge from b to a saying that b should be flushed before a is flushed right. Creation should happen before unlink right.

On the other hand, let us say after that somebody executed a command called mv b slash bar to a slash bar. So, somebody create executed yet another command and so this time you wanted to draw an edge like this right this time you are moving in the opposite direction. So, you basically want to create you going to first create a link in bar and then

remove the link from in b, in a and then remove the link from b right. This time you basically want to say that this should happen before this right and here is an example where this dependency graph could have a cycle.

And, if it has a cycle then at a time of flushing it back to disk it is unclear which one you should flush first right. Let us say you flushed a first then you have lost it is possible that crash that happens after you flushed a then you may have lost the link to foo write. So, foos contents may have been lost. On the other hand, if you flush b first then bars contents could have been lost ok. So, that is cycles now how do you resolve something like this.

Student: In data driven what is to be updated in a and what is to be updated in as in not just the whole of data of a and then whole of data.

Okay so here is the suggestion I am I am maintaining ordering at block (Refer time: 12:54) I am saying this block should be of committed to disk or written to disk before that block I am without going into semantics of what is inside the contents of the block. And, so suggestion is instead of saying that this block should be committed before that is that block say these contents in this block should be committed between before that content those contents in that block.

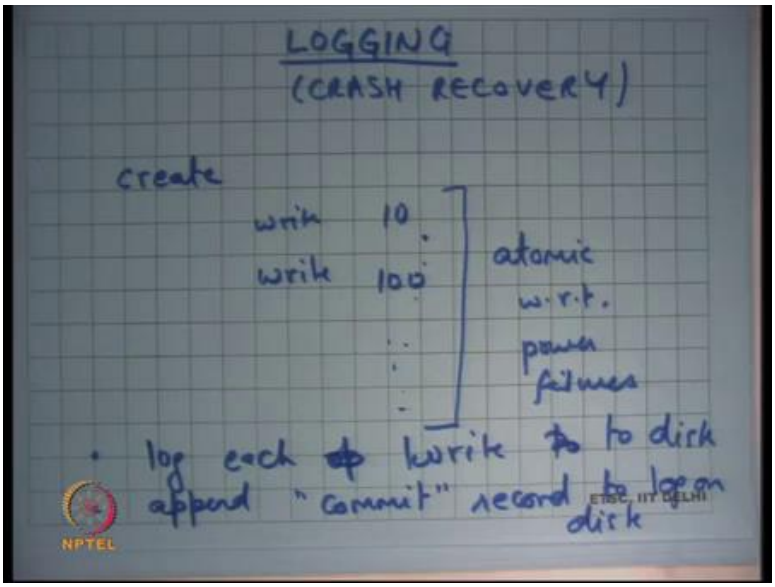
That is not a practical solution right because that is there is too much semantic information that needs to be stored and that it is basically almost like writing you know at the at flushing time you have to basically look at this semantics what is written at what bytes that has been written it is a directory or a etcetera I mean these are the kind of things you do not want to worry about at flushing time. All you want to care about is here some disk blocks that are in the in the cache they are dirty they need to be written back to disk.

What is the other thing that you can do? Well here is here is one suggestion. When this operation gets executed in memory the OS figures out that hey there is a this like this is going to be a cycle in your dependency graph. So, when you have when you see that there is a possibility of a cycle you stop that operation and you flush this disk blocks to disk such that all the such that this disk this edge gets removed.

So, once you flash them to disk the old edges will get removed which means you know the old state will get consistent and now you can perform this new thing right. So, every time you are making an operation you check the dependency graph if you see a cycle you hold on you flush the blocks that are involved to the disk. So, that the old edges get removed and so the new edges can get created without having a cycle in your dependency graph ok.

So, ordering with a write back and coupled with an fs check program has had been a has been a popular solution from a long time.

(Refer Slide Time: 23:51)



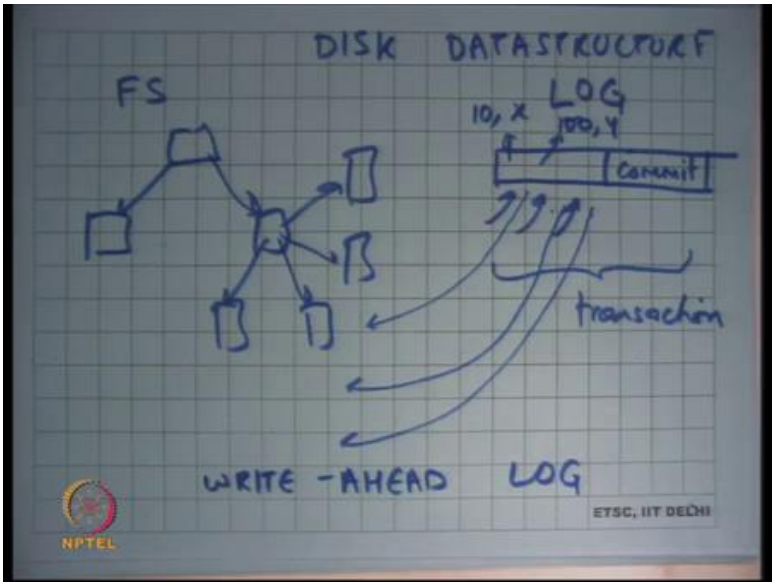
But with growing disk sizes it has become relatively unpopular and the other way to do cache recovery is login. So, let us see how login works. So, the idea is that let us say there is some system call let us say there is create hence going to do write to block 10 then 100 and so on right. And, then and that is it and want to make sure that these operations are atomic with respect to power failures ok. So, there are some operation that is gone to have multiple disk writes and then you want to have them atomic with respect to power failures.

Here is one way you could do it. Each time you see something like this you basically do not write the disk blocks to the file system at the time at this point you keep recording these operations in a log. So, basically start logging you maintain a separate data structure called the log on disk and each time you want to log write something you

basically say you basically log these operations. So, you say that I want to write to block 10 with these contents and you put that in the log. When you put it in the log you basically saying you're you basically indicating your intention that you want to write to this disk block you are not actually written it to the disk block right.

So, you log each operation each log is write I should say to log alright to disk. After you have done after you have done logging each of these writes you append a commit record to log on disk right. So, you basically first write that I want to write to all these blocks and then append a commit record to the log on disk alright and your operation is complete only after the commit record has been written. If there is a power failure before the commit record has been written it is as though none of these writes have happened. If there is a power failure after the commit record has been written it is as though all these writes have happened.

(Refer Slide Time: 26:34)



So, let us see I have a disk let us say I was to draw the data disk data structure. So, I have some tree like structures you know different types of tree; one is a directory tree and other is an inode tree and so on. So, they are tree like structures on the disks. So, this is let me call this is the file system and then there is a sequential structure which I call the log.

Each time I want to make an operation I basically start writing to the log and then I write a commit recall that we made the assumption that the write of a sector is atomic to the

disk. So, either the entire commit will be written or none of the commit will be written it is not like half of the record can get.

So, this operation is atomic and you using this operation to basically make the entire operation the entire sequence of writes atomic and after you have written the commit log commit block you are going to asynchronously make these writes to the file system alright. It is after you have written the commit record.

So, let us see once again. I wanted to make an atomic operation which involved multiple disk writes I will create a new transaction. So, this entire thing can be called a transaction. So, I will create a new transaction I will log the blocks that I need to write and log them to the log and then I will write a commit record to the log.

After that I am done as far as I am concerned the disk block the disk is not in consistent state and asynchronously I am going to flush I am going to copy the blocks in the log to their respective positions on the file system right. Recall, that all these logs have annotation saying block 10 contains x block 100 contains y and so on. And, so asynchronously I am going to write these blocks from log to the file system.

Now, let us see what happens if there is if there is a crash if there is a crash before the commit record was written no problem it is as though nothing happened right. If there is a crash after the commit record has been written, but before you have started applying the changes to the disk no problem the operation has finished atomically and now you can now do the same thing.

So, at recovery you can just apply the log to the file system. What happens if the crash happens in the middle of application of this log to the file system? So, let us say you know I have I have written the commit record after that I was asynchronously writing block 10 and before I could write block 100 there is a power failure no problem.

Student: (Refer time: 29:30) 10 we will do.

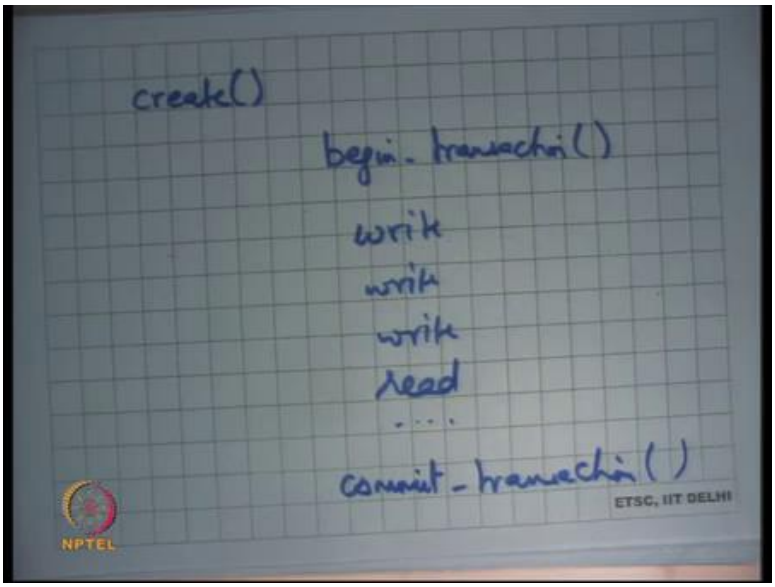
At recovery time you will again write ten, but with the same contents. So, there is no problem right. It is a you will just overwrite the 10 again, but with the same contents that is no problem either right. So, this is nothing, but a write ahead log where whatever you want to write you write it to the log first and you keep doing this and then you write a

commit record after that you actually push all those writes asynchronously to the real file system. So, we are all convinced that this would ensure consistent state even across random crashes on the in terms of power failures.

Notice that the write of the commit record on the log is acting as a serialization point. If a power failure happens before the commit record it is as though the transaction did not happen at all. If the cache happens after the commit writing the commit record as though the transaction happened in completion.

Even if there is the crash that is happening in the middle of your application of the log to the file system it is still consistent write this data structure of the file system plus log will always remain in a consistent state. There is no inconsistency that may happen alright. So, how does one write code?

(Refer Slide Time: 31:11)



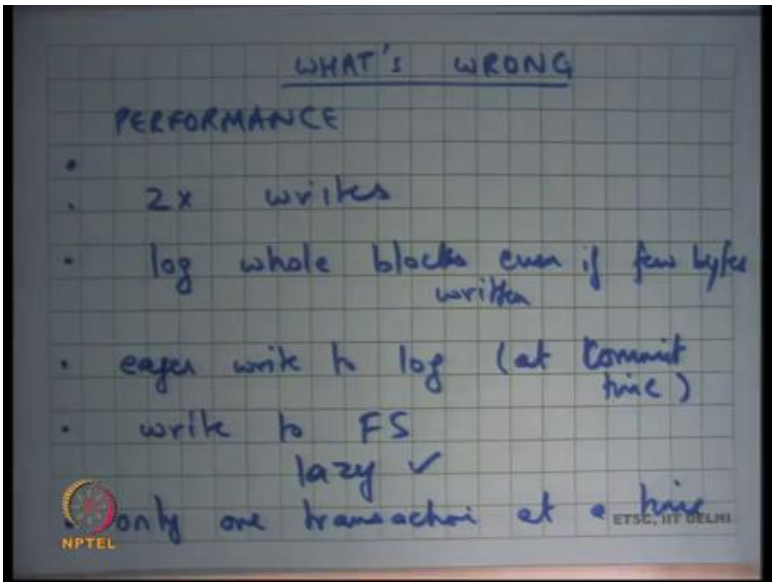
To do this let us say I have a cis call like create. I will just say begin transaction then you know write, write read etcetera and then I will say commit transaction. So, all I have done is that all the all the operations that I wanted to make atomic with respect to power failures I have bracketed them with begin and commit transaction calls right just like very similar to your locking acquire and release right.

And so, begin transaction is going to write to the log that I am starting a transaction. All these operations are going to append records to the log and then commit transaction is

going to write the commit record on the log and there is going to be another thread that is asynchronously going to apply the log to your file system alright.

So, it is very easy to write to basically there is the code structure does not change much you just have to enclose things that you want to make atomic with begin transaction and (Refer time: 32:25) append transaction. After you are done applying the log applying the transaction to the file system you can delete the log you can delete the transaction from the log so that it can get reused right. So, till it has not been applied to the file system you have to keep it around, but after you applied it to the file system you can now free it for use for the next operation alright.

(Refer Slide Time: 32:53)



So, what is wrong or what do not we like about this performance right what are something that are that are happening. Firstly, everything that I need to write I need to write twice I need to write first to the log and then to the file system. So, every operation basically has a 2x overhead in some sense right. So, 2x writes ok.

The second thing is I log whole blocks even if few bytes written. So, even if I just update one byte in the block, I have a log the entire block in my log. So, my log space overhead is larger than then what you would have wanted it to be. Eager write to log. I am very eager to write to the log every time I do a write I basically immediately want to write to the log right. And, as soon as I do a commit, I want to basically make sure that the

commit record has been written to the log and then later I am going to write it to the file system.

So, is this avoidable? Can we not have eager write to log to the log. Well we will see this in the next lecture, but any I mean as it stands now this thing has is a has a very bad performance each time I want to make a write I actually need to go to the log and makes this write. One simple optimization could be you do not do the eager the writes to the log as they are being done. You wait for the commit transaction to happen and then all these 4 or 5 records that you written can be written in one batch to the transaction to the log right.

This is a small optimization, but this is still not good enough why is this not good enough I still need to you know do this eager write on every commit. So, there is no not write back in the true sense every operation needs to be synchronously committed every atomic operation. I would have wanted that even atomic operations can be written back using a write back cache right. I want my write back cache to have more freedom in or more batching than just 4 or 5 disk operations alright then.

Student: Sir.

Write to file system. So, write to file system as we discussed can be done can be done lazily. So, this is this is write. Here we are saying that once you written it to the a log you basically lazily apply the log to the file system, and you can batch it in a and that gets a lot of performance right. Ok, there is a question

Student: Sir, suppose if we update a suppose if we update the log after fs 4 5 operations and whenever there is a commit sir, but in that case and before that we are just writing to a cache now, but suppose if there is a power failure while I am writing to the log from the cache and the commit keyword is written, but other operations are not written.

Ok so, good question I am saying you know there is an optimization that a transaction need not be written to disk or their blocks in a transaction need to not be a written to disk eagerly you can you know keep them in cache and then at commit time write the entire log transaction to the log in one go and the question is in when you are writing it back if the commit happens before if the commit is written before the other blocks then there is a problem right.

So, at the time of flushing it you have to basically make sure that you know there is some ordering in which you are done doing it typically the way this will be done is that you will issue all the transaction records in one go. So, all the transaction records happen in one go and then the commit record happens in a second iteration. So, in that way you have 2 sequential writes to the disk right.

So, you waste one full rotation assuming that the log is written sequentially you first make one sequential write to write the entire transactions blocks and then you make one sequential write to write the commit and just to make the ordering you have to do 2 instead of one. So, that the disk does not reorder them internally. Modern disk interfaces allow you to specify that you know here are 5 reads 5 writes, but make sure that this 6th write is after this all these 5. So, you know that way you can even avoid this extra overhead of multiple writes good.

So, we have seen we have seen logging in it is very raw form where we are basically, we are using the log we are eagerly writing to log we are logging the whole blocks and eagerly in the sense at commit time. Also, we are making another problem with this thing is that only one transaction at a time right. Why is it is true. Let us see let us say I have this code begin transaction and then I start writing something then I commit transaction to ensure atomicity across between multiple access to the file system the way I have discussed it so far you basically want to make sure that only one of them one transaction is active at any time.

If the multiple transactions active at any time, then they are more problems to be dealt with right. So, the way we are discussed it so far there is only one transaction that can be active at any time and that in itself is a very big performance problem because if there are you know lots of users running lots of programs to completely different parts of the file system they get serialized because of this common log across the entire file system right; so, not a good idea at all alright.

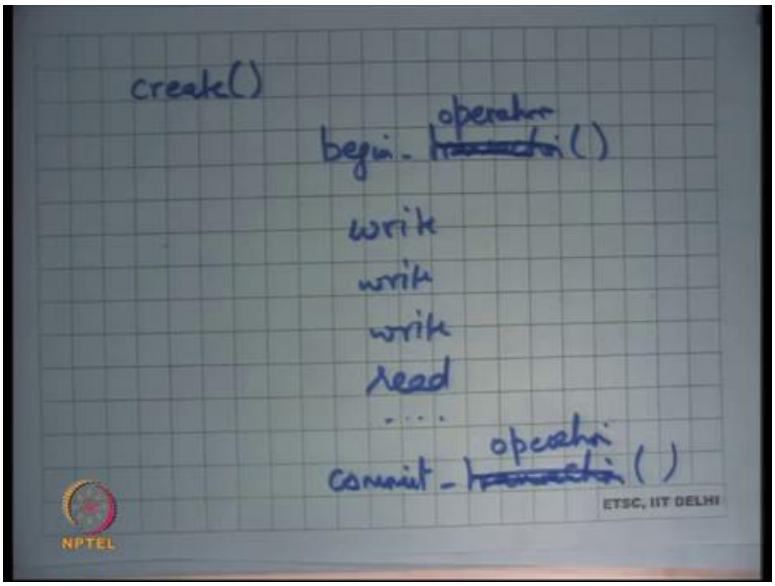
(Refer Slide Time: 39:28)

So, let us see how we can fix it and I am going to look at the ext 3 file system on Linux right. So, the ext 2 file system was basically based on the ordering and fx check program that we that we discussed earlier, but the ext 3 file system introduced logging and, but in an efficient way and we gone see exactly how this is done. So, we saw some problems with the way that we have discussed so far and let us look at what can be done. Firstly, a transaction is not one operation, but multiple operations right. So, if there are a 100 creates happening at the same time, they are all made part of one transaction.

So, one; so, a transaction basically represents locality in time. So, basically, we say I start a transaction and the all the file writes that are happening now will belong to this transaction. And, then at some point I am going to say stop transaction when I say stop transaction all the operation that are completed belong to this transaction and all the transaction that are ongoing. I will wait for them to complete. And, when they complete all these operations together form one atomic unit right and this atomic unit gets flushed to disk in a log.

And, then there is one commit record for this entire big chunk of writes possibly by multiple users multiple processes different parts of the file system completely different operations right. So, what you are doing is you are making you are clubbing lots of different atomic operations into one large transaction.

(Refer Slide Time: 41:25)



So, basically what this means is that these begin transactions and the commit transactions can be replaced by begin operation and commit operation. And, it is not necessary that at the time of committing the operation you actually commit the transaction on disk you just basically say that the operation is finished. And, so the invariant is that a transaction will have either the entire operation in it or none of the operation in it and a transaction will never half of an operation and so you basically club lots of different operations into one transaction and so that gets rid of some problems.

So, for example, only one transaction at a time you have solved it right. So, you can have lots of different operations at the same time and then you can choose to commit them at your at your own will. How do you choose when to commit a transaction? You can say every 5 seconds every 30 seconds completely reasonable choice right.

So, for example, you know modern operating systems do not give you any guarantee about if you are writing something whether it is actually persistent on disk or not, but every 30 every 30 seconds let us say the transaction will get closed. And, so because the transaction got closed it will get written on to the disk and at that point you can be sure that the data your whatever you wrote 30 seconds ago is very likely on the disk now right.

So, this interval; so, you can just choose to close the transaction at will whenever you like and when you close the transaction at that point you make sure that all the operations

that started before the close of the transaction will get committed on disk right. So, you fix this you also fixed eager write to log.

So, now you can because you are using a transaction worth of thirty seconds of writes you can actually batch all of them and write them all together and. So, it is not an eager write of log to log you actually instead of writing 6 operations you are writing 6,000 operations together to the disk. So, that is much more efficient right.

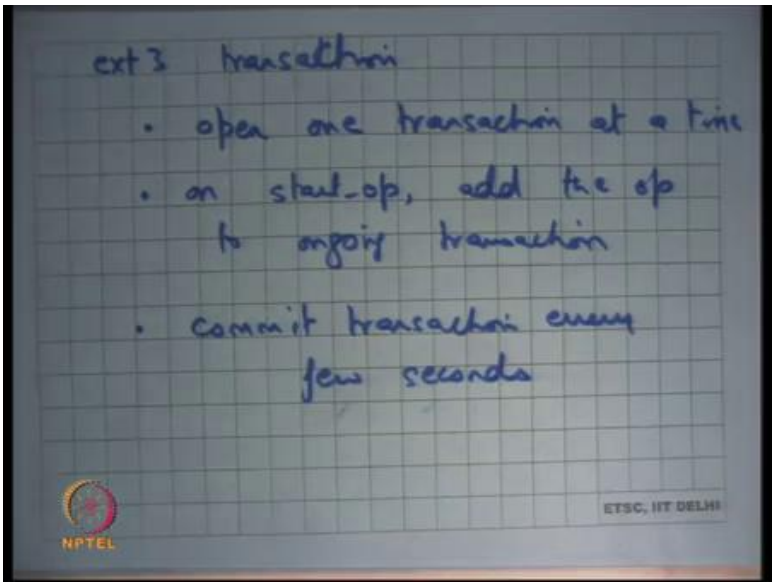
So, this one is already good write to fs is lazy log whole blocks even if few bytes are written. So, there is still log whole blocks even if few bytes are written because the you know it is extremely hard to keep track of which bytes have been written and which not. So, you log the whole blocks, but the nice thing is because there you are you are making out transaction. So, big if there are multiple operations that wrote to the same block. So, you log the whole blocks even in ext 3.

But the nice thing is because you are doing lots of different operations batching into one transaction you do not have to these blocks may have been modified many times within the same transaction. And, you do not need to record all those blocks all those different versions of the blocks you only need to log the final version of the block.

So, let us say if this block was modified a thousand times in this thirty second interval all you need to log to disk is the last value. So, you still log the whole block, but you just absorb lots of writes and doing so this also becomes better alright and 2x writes remains, but the nice thing is because you are batching. So, many writes together in log right it is all it is complete sequential write to the log. So, that is fast as we know there is only one seek and one rotation and you would do write at 50 to 100 megabytes per second to the logs. So, that is fast.

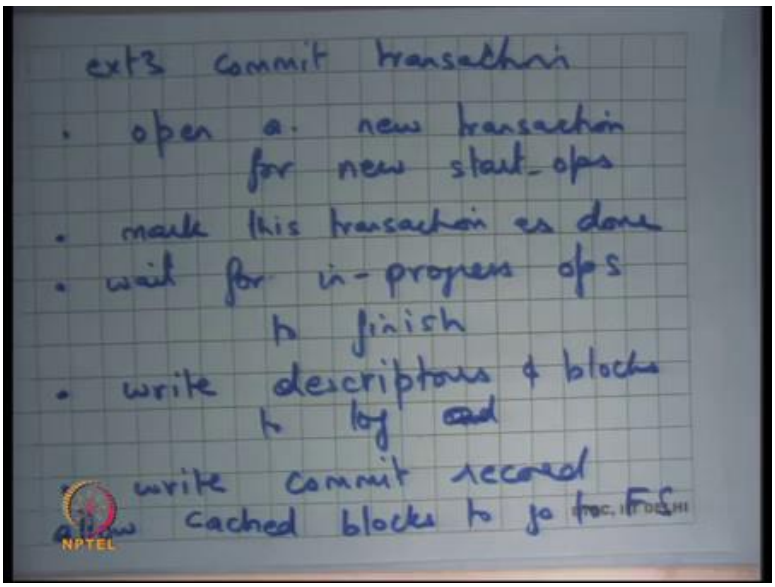
Also, now all these writes now need to be applied to file system, but that is a huge transaction. So, because it is a large transaction you can have lots of IOs and flights simultaneously. So, writing to the disk is also faster you have much better disk bandwidth utilization. So, essentially the larger the transaction the less the 2x write problem is right and because I have made my transaction as large as I want there is no problem right.

(Refer Slide Time: 45:29)



So, let us look at how ext3 works. So, let us look at an ext3 transaction. Open one transaction at a time on start op add the op to ongoing transaction alright. This is all in memory operations. Each time somebody starts an op you basically add the op to the ongoing transaction and commit transaction every few seconds or few tens of seconds depending on what you want how much reliability and performance trade off you want.

(Refer Slide Time: 46:30)



Let us see how would you commit transaction. Firstly, open a new transaction. So, when you are committing a transaction you basically say that all new ops will now belong to

the next transaction. You basically first close your current transaction which basically means any operations that are happening after this point will now belong to the next transaction. Mark this transaction as done. Wait for in progress ops to finish.

So, there could be some ops that are started, but are not yet committed or haven't stopped finished right. When you close the transaction there are some partially completed ops you just want to wait for those ops to finish before you actually write the committed record right. So, that is how you basically maintaining atomicity right. You have basically said that I am I can close the transaction at any point you want, but when you close it you also wait for all ongoing or all operations that have started before this those to finish. So, that is how you basically ensure atomicity of each operation.

Then write descriptors. So, descriptors are basically these blocks which say which contain meta information about which block and etcetera which block which version number etcetera. And, then what is sort of data right descriptors and blocks to log and wait right and then write the commit record and that is it right that is your finish, that is your commit transaction and then asynchronously you are going to write the contents or whatever the log is saying to the file system right allow. So, after you written the commit block you can now allow the blocks that have whose contents have been logged in the transaction to now get written to the actual file system alright.

Let us stop here and let us discuss the ext3 file system in more detail next time.