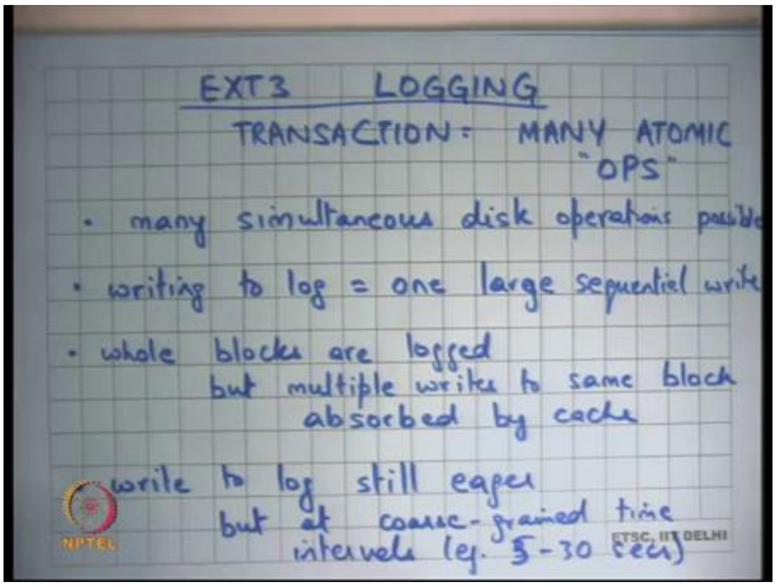
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 35
Logging in Linux ext3 filesystem

Welcome to Operating Systems lecture 35.

(Refer Slide Time: 00:31)



EXT3 LOGGING

TRANSACTION = MANY ATOMIC "OPS"

- many simultaneous disk operations possible
- writing to log = one large sequential write
- whole blocks are logged
  but multiple writes to same block absorbed by cache

write to log still eager
but at coarse-grained time intervals (eg. 5-30 secs)

So, we are discussing about logging as a way to implement fast cache recovery, we first looked that very simple way of doing logging where every disk operation that needs to be a atomic was considered a single transaction. And, there was a commit after every disk operation right and we said that there are lots of problem with that.

And then we said, but you know in practice you would the extend this idea and we were taking the example of the ext3 file system on Linux, where a transaction is lots of atomic operations bunch together into a single transaction.

So, in other words with respect to power failures either all these disk operations happen it ones or none of them happen at ones. So, it is making a one big atomic operation out of lots of smaller atomic operations. And if you do that then firstly, you; so, the way firstly, in the previous case we said that only one transaction is possible at a time.

So, even now there is only one transaction possible at a time, but because one transaction can have lots of different disk operation many simultaneous disk operation is possible at the same time right. So, transaction is no longer causing serialization across multiple disk operations where disk operations are completed two different parts of the file system.

Of course, if the disk operation that was the same part of the file system then there is in memory locking to ensure that there is no concurrency problem, that we have discussed already right. So, there is buffer level locking to ensure that there is no; there is no atomicity violation by concurrent access to the same block. But here we are talking about atomicity with respect to power failures right and for atomicity with respect to power failures we wanted to have a transaction and we wanted and we could only have one transaction at a time.

And so, that was causing serialization, but because you have multiple operations in a single transaction that serialization problem goes away. Also writing to log is just one large sequential write and we know that sequential writes are fast assuming that you are doing you commit. So, the weight works as you open a transaction and you close it after every periodic interval.

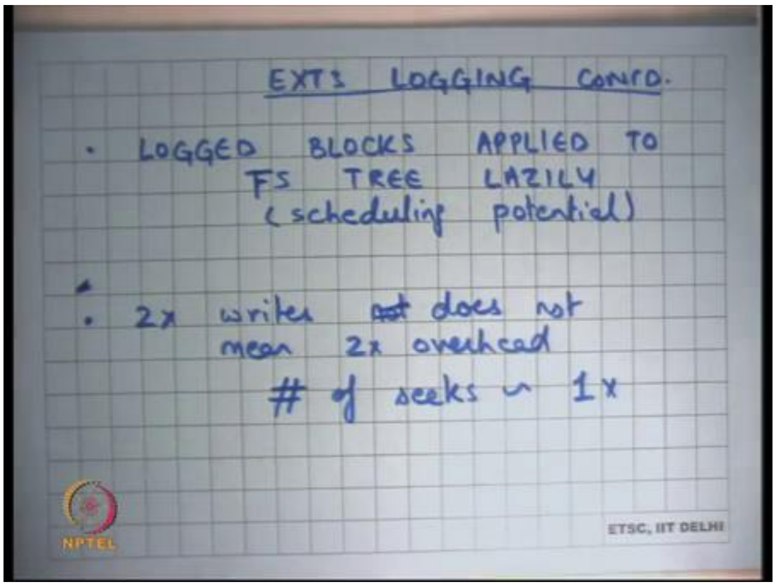
Let us say every 5 seconds or every 30 seconds depending on what kind of guarantees you want. And, after interval you going to commit the entire transaction includes which include all the disk operations that have disk writes operation that happened in during that time. And so, one last sequential write is fast, whole blocks are logged still.

So, we said one of the problems with this kind of logging is the whole block is log, if you even modify 1 byte in the block the entire block get needs to be log into the log. And that is more space overhead in the log, but it is still present in ext3. The only saving grace is that if there are multiple rights to the block which is a very common case you know this likely if you for example, creating lots of files in a single directory then you probably making lots of writes to the same directory inode and you know 100s of writes to same like an inode.

You do not need to log all those 100 versions of that block; you only need to log the last version of the log right. So, in multiple writes to the same log get absorbed by the cache, write to log is still eager. So, when you commit you have to commit you have to write the entire log at that time, you cannot lazily say I am going to commit it later because otherwise you completely lose your guarantees with respect to power failures.

So, it's still eager, but the nice thing is that these eager writes to the log are happening at very coursed in time intervals; 5 second or 30 seconds depending on what you have choose. And so, it's not that big of a problem anymore alright.

(Refer Slide Time: 04:00)



EXT3 LOGGING CONTD.

- LOGGED BLOCKS APPLIED TO FS TREE LAZILY (scheduling potential)
- 2x writes ~~not~~ does not mean 2x overhead
  # of seeks ~ 1x

Finally, log blocks are applied to the file system tree lazily. So, the commit of the log or commit of the transaction is eager, but the application of the log of the transaction blocks to the filesystem tree is lazy right. So, you can just do it whenever you know whenever you find time or whenever you find the disk to be idle or more importantly you know you can take a lot of logs and try to write them simultaneously.

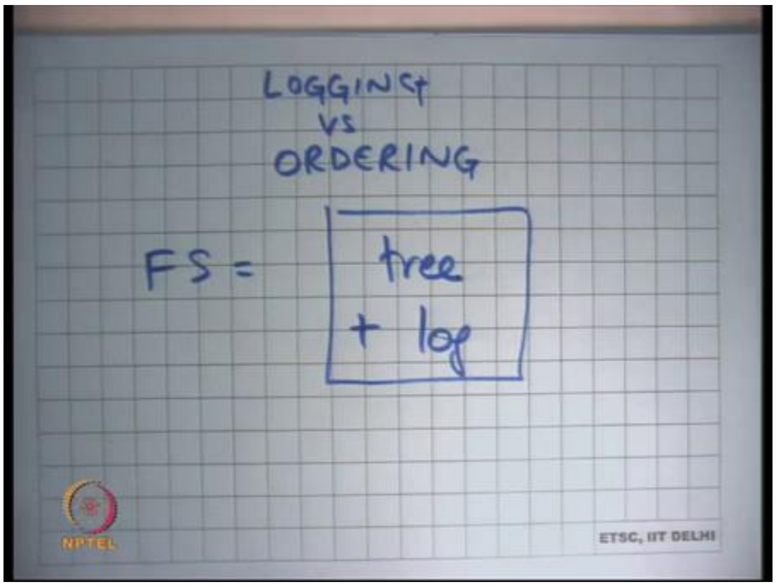
And so, that gives the disk a lot of scheduling potential. So, recall that if you do lots of in-flight IO simultaneously, the disk can schedule at much better than if you have one IO time. So, you can and the all these can be done simultaneously; so, have a lot of scheduling potential.

So, even though still you are having 2x writes you are writing each block to the disk twice, the overhead is noT2x right because the first write is part of a large sequential writes. So, it almost free and the second write is also not as expensive because you have a lot of scheduling potential because you are doing lots of in flight IOs for application of to the FS tree.

So, the number of seeks or latencies a that you pay is still roughly 1x right even though you are writing the blocks twice to the disk the amount of overhead or the number of assuming that you are counting over ahead as number of seeks you making to the disk is still 1x alright.

So, that is you getting; so, basically with logging you have used the characteristics of the magnetic disk to ensure fast cache recovery yet be able to maintain the same level of performance. The advantage of logging over the previous methods that we saw which was ordering.

(Refer Slide Time: 05:46)



So, if I just compare logging versus ordering as we saw that the performance difference is not much, logging is almost the same performance as ordering. In fact, its little simpler you do not have to worry about; so, you know recall that ordering had this problem that some invariants will get you know some inconsistencies can arise.

For example, free space they can be memory disk (Refer Time: 06:11) leaks or there can be in the case of moving a file from one directory to another it is possible that the same file is pointed to by two directly. So, ordering hard possibilities of inconsistencies, logging does not have them, logging ensures the atomicity of the entire operation across crashes.

Also, the other biggest advantages basically recovery at recovery time you do not need to do a full global files system scan right. Recall in the ordering case to check my invariants I had to actually troubles the full file system in a global way and that was becoming very expensive. And, we said it can take an hour for the sizes of disk that we have today which is 100 of gigabytes to you know 1 or 2 terabytes, its it can take up to hours to do this and that is becoming very impractical as we go along.

But logging has does not have this problem right, you do not have to do a global files system scan right. What do you need to do to recover from a crash? You need to scan the last few transactions in the log right and see which of them have not yet been applied to the disk right.

So, if we are going to look at how exactly the crash recovery works, but you can already see that you do not need a global operations; all you need to look at is the last few blocks of the log, yes question.

Student: Sir, does log exist in the file system?

Does a log exist in the file system? Yes, I mean log is part log is the structure on.

Student: (Refer Time: 07:34) file.

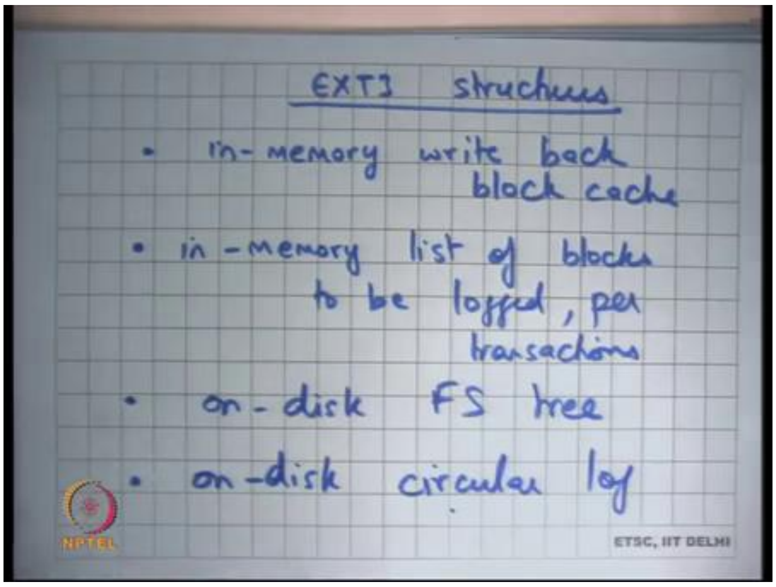
You can consider the log as a separate file in file system, but I mean its it has special semantics. It is not your you know it does not have the same semantic. So, for example, the user cannot see the log right, the user cannot read or write directly to the log, the log is basically being written by the system by the kernel.

Student: Sir, I am asking you suppose something gets corrupted while writing to the files systems structure. So, can we still access the log to recover from that?

So, depends on; so, the question is if something gets corrupted while you are writing the file system structure can we recover it by using the log? So, I mean in generally yes, but I mean depends on what kind of corruption you are talking about right. I mean the whole idea of the log is that you; so, basically your file system now is equal to a the tree which is you know tree of inodes and blocks and directories and so on plus the log right.

So, your tree can become inconsistent, but the total of tree plus log will never be inconsistent right. So, that is the guarantee that you know if you look at this whole structure as one whole thing tree and log then you know the invariance across these things are always maintained. The tree itself can get inconsistent and that is why the you know when you, if there is a crash then the log can basically correct the tree (Refer Time: 09:02). So, let us look at the ext3 structures.

(Refer Slide Time: 09:13)



EXT3 structures

- in-memory write back block cache
- in-memory list of blocks to be logged, per transactions
- on-disk FS tree
- on-disk circular log

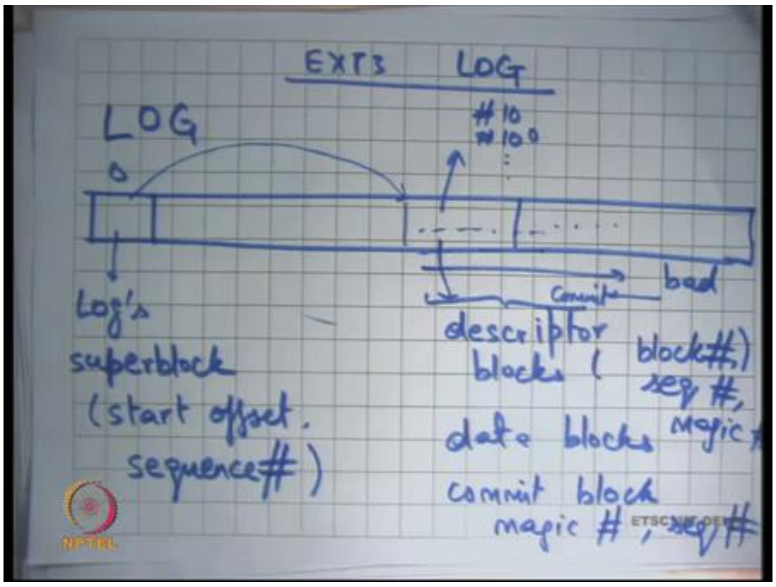
NPTEL
ETSC, IIT DELHI

So, there is an in memory write back buffer cache or block cache right, we already know this. We have seen this was present even without logging, you basically have a buffer cache which we need and needs to be right back for performance. You have now an in-memory list of blocks to be logged per transaction.

So, as your as the disk operations are happening you make you are keeping an in memory record of these are the list of blocks that belong to this that have been dirtied, that have been modified and they belong to this particular transaction. So, when this transaction commits these are the blocks that need to be flushed to the disk right.

So, you need to maintain this data structure in memory alright and then of course, you have the on-disk FS tree and then you have the on-disk log. And, this log is maintained as a circular buffer because you know the log cannot keep growing in you know infinitely. So, you basically just wrap around the log and you also keep freeing the log as you apply the log changes to the file system tree, we are going to see how.

(Refer Slide Time: 10:45)



EXT3 LOG
LOG
0
#10
#100
:
commit
bad
Log's superblock (start offset. sequence#)
descriptor blocks (block#, seq #, magic #
data blocks
commit block magic #, seq #

So, let us look at the ext3 log. So, ext3 log let us say I if I draw the ext 3 log this is my this is not my disk, this is my log on the disk. So, this is the part of the disc which stores the log let us say alright and the log has let us say those 0th block in the log is the logs superblock. This superblock is different from the file system super block, this is the log superblock.

By superblock I basically mean basically mean that it contains meta level information about the log right for example, where does it start right. So, this will contain you know start offset and a sequence number. The sequence number indicates what the current, what the sequence number of the current of the transaction that the first transaction at this start off set right.

So, the idea is that you have you maintain global sequence number and each time you close a transaction you increment the sequence number for the next transaction. So, one after another the sequence each transaction will have a new sequence number right. So, this is basically pointing somewhere here the start of set and you can find the log starting here ok. At any time, you reach the end of this area you wrap around like this alright.

What does the log have? The log has two types of blocks descriptor blocks and data blocks. So, descriptor block basically says that the next block; so, it just describes the next block for example, or in the next few blocks.

Basically, says the next few data blocks are for the next few data blocks are for block numbers x y z and so, these are the data blocks. So, for example, you can say here is a descriptor block which says number 10, number 100 and so on. And, then after that you have the blocks the respective blocks, the contents of the respective blocks right.

So, there are two types of blocks, there is descriptor block and there is a data block. The data block contains a full data, in the descriptor blocks describes what data it contains. Are the descriptor block similar to inode in structure? No, not at all, descriptive blocks are just something specific to the log which just says that no. So, basically the log contains all the dirty blocks right that need to be applied to the file system.

So, the descriptor block just says that these are the dirt you know the next few blocks are data blocks and these are the these data blocks are the contents of block numbers xyz. So, you know exactly where to apply these blocks that is all. So, the descriptor blocks contain information like block number alright.

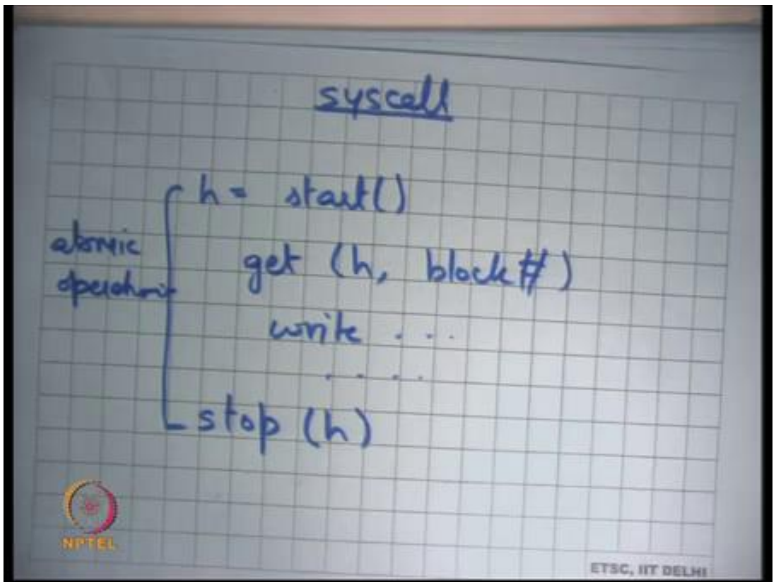
So, which block it is we have already discussed this, it also contains information about sequence number we going to see why need it. So, it every descriptor block contains a number which says which transaction do I belong to right ok. So, this sequence number is the same. So, as the sequence number in the block and we going to see why it is needed, why do we need a copy of the sequence number in the descriptive block. And finally, we have something called a magic number.

By magic number it basically means that there is some number there which says that yes, it is a descriptor block it is some identifier. So, it is not some random block, it is it if the magic number is present you can be sure that it is your log block, it does not have any garbage in it. So, it is just an identifier which says it is a block of ext3 log.

And so, in general the magic number is used in many different places and the purpose of a magic number is to identify or to disambiguate a valid block or valid piece of data from completely garbage data ok.

And then so, that is there is descriptor block, there is data block and then there is a commit block alright. It basically says that you know at this point this is where the whether the transaction committed or not right. And, the commit block contains once again it contains a magic number and it contains the sequence number of the transaction, that it commits .

(Refer Slide Time: 15:21)



syscall

```
atomic     h = start()
operation      get (h, block#)
                write . . .
                  . . .
           stop (h)
```

So, just to review how does the syscall work, it basically says let us say I wanted to make a disk write. So, I will say h is equal to start, this is like start transaction as we have discussed last time or start operation as we have discussed last time. Then you are going to say you know get h block number to indicate that you are dirtying this block you are going to you know write to these blocks and then you going to say stop h right.

So, this is how you basically say that this operation should be made part of the current transaction. And, this operation should be atomic with respect to crashes right. So, it is basically its so, all you have to do is basically look at areas where are you doing multiple disk writes and identify points where you want to say that these multiple disk writes should be atomic with respect to crash failures and then you put a start and a stop around them right. You know one simple thing to do could be that you put a start and a stop around at the big at the start of the system called and at the end of the system call always, you know that is one safe option to do also right.

So, this has a lot of similarity with transactions as we have discussed earlier right, you say begin transaction and end transaction and it just runtime system that takes care of what are the things that you modified and whether they going conflicts or not etcetera similar thing here right. Because, their transactions are not logs you do not have to worry about fine grained versus coarse grained, you do not have to worry about deadlock something like right. So, you are get all the advantages of transactions alright; alright so let us see.

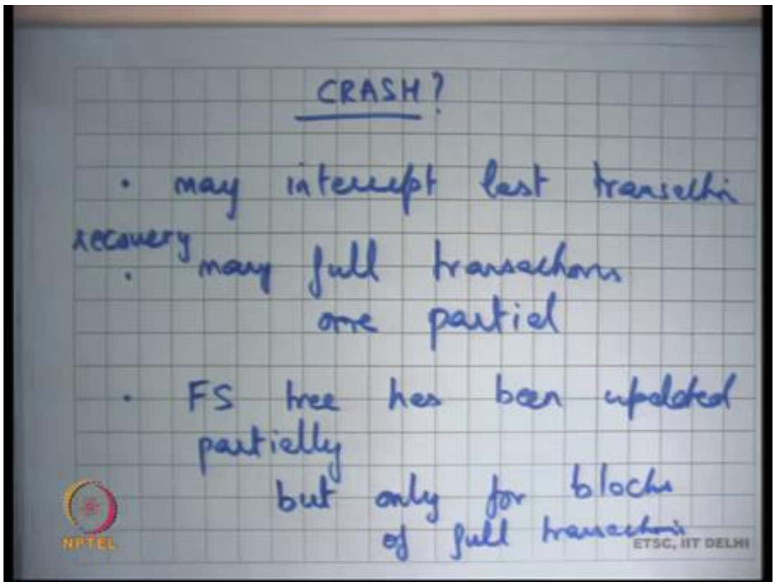
Student: (Refer Time: 17:07) reference. So, here also like multiple transactions can execute at the same time and then it later checks whether there is some discrepancy and reverts back.

So, the question is you know in these transactions it’s possible that multiple transaction is happening at the same time and if there is a discrepancy then you rollback. Notice that these things here are not what I am calling transactions, these are I am these are the terminology I am using for these are atomic operations.

And, their multiple atomic operation that in go on concurrently and they are all part of a single transaction right, atomic operations do not have any conflicts with each other. So, we are not talking about you know. So, one atomic operation and the another atomic operation as long as they touching different buffers have absolutely nothing to do with each other. But we want to make sure that you know their atomic with respect to crash failure. So, here atomic is not with respect to each other, here atomic is with respect to power failures right.

So, and so, you can you know so, this the way you deal with that is different. So, now, you put lots of you allow these atomic operations to execute concurrently, but you put them all in a single transaction. And, that transaction is made atomic with respect to power failures, that is what you are doing alright.

(Refer Slide Time: 18:31)



CRASH?

- may interrupt last transaction
- recovery: many full transactions, one partial
- FS tree has been updated partially but only for blocks of full transactions

So, let us see how what happens on a crash ok. So, it may interrupt last transaction while it was writing let say the transaction has decided to commit, its writing the transaction to the log and before it could write the commit log, the power went off that is possible alright. So, it is possible that you have many full transactions, when you recover; at recovery time you see many full transactions and one partial transaction in the log right ok. Also, you may see that the FS tree has been updated partially alright.

So, when you recover from a crash these are the possible things that can happen, you can you know you can see lots of full transactions and you can see one partial transaction. And, you can see that the FS tree has been partially updated which makes it inconsistent, but you know together with the log you can you know that tier structure is consistent.

So, but the what is the other thing? Can it be updated partially? Can it be updated with the transactions that have not to yet completed? So, the FS will up only, but we update a partially, but only for blocks belonging to full transactions. So, the then we again following is that after only after we commit a transaction to the disk, do we start applying that transactions blocks to the tree?

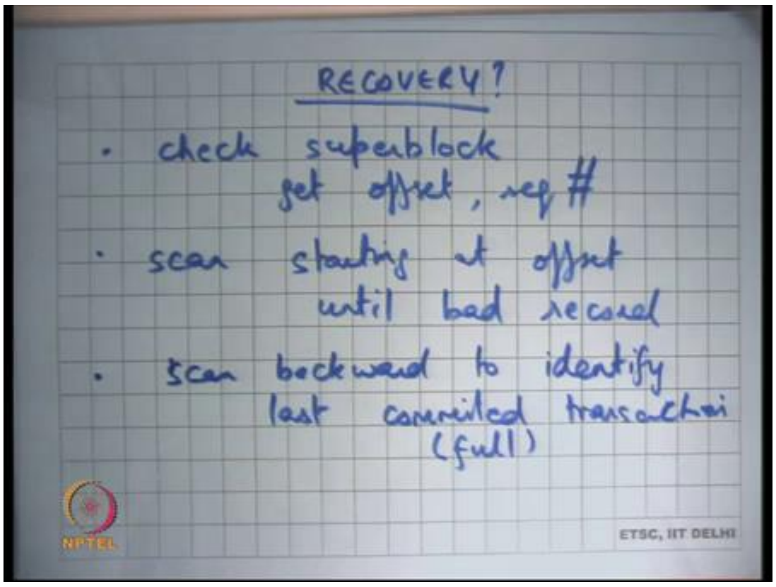
So, the one that is partial none of it is applied to the filesystem, you can be sure of that; its only the ones that I have completed whose blocks have been applied to the file system. And so, if partial blocks have been applied to the filesystem no problem you can you know because application of is completely idempotent.

So, you can reapply them, this is the same contents that you are writing again. So, all you need to do is look at all the full transactions and reapply all of them to the filesystem tree. And, completely ignore the partial transaction, it's as of the partial transaction did not happen at all right.

So, what kind of semantics is giving the programmer? It basically saying that if you for example, create a file and the there is a crash after you know even though the create system call return. And, after that you have displayed something to the user saying you know I have created the file etcetera and there is a crash and then you come back again there is no guarantee that the file is actually on the disk.

Because, the transaction which contains this create operation may not yet have committed. And, let us say its committing every 30 seconds then that is not committed. But, given that you have some reasonable interval for the commit transaction, then be show that something that you wrote you know few minutes back or 1 hour back is on disk alright.

(Refer Slide Time: 21:51)



RECOVERY?

- check superblock
  get offset, seq #
- scan starting at offset
  until bad record
- scan backward to identify
  last committed transaction
  (full)

ETSC, IIT DELHI

So, let us see how does recovery work. So, first check superblock ok, to get offset and sequence number right. Then scan starting a offset until back record right. So, basically at recovery time I look at the super block to get the start of the transaction and then I just keep scanning forward till I find something that is bad. What do I mean by bad? No. So, by bad I mean I see a block that does not have the same sequence number that I am expecting ok.

So, how do I know that something has not committed? Basically, I am going through all the blocks and I am looking at the sequence number of these blocks right and then I suddenly see a sequence number that is older right. So, that basically means that this transaction could not write all the blocks right. So, the sequence number is disambiguating between blocks of the current transaction and blocks that may have been written earlier recall that it is a circular log right.

So, it you know there could be some rights that I have living from the previous log. So, you just scan the log till you till you say something wrong and something wrong could mean a bad sequence number or it could mean a bad magic number. So, it is possible that you know the disk had completely garbage, the log had completely garbage contents in the beginning and then you basically you know started writing.

And now the sequence number is see somehow correct, but the magic number you know there assumption is the magic number cannot be correct, if you know if it is some garbage. The probability that the magic number is also correct is very small basically ok. So, you basically start here, and you keep scanning till you find the first record that is bad and then you scan backward from there to find the first commit record right.

So, you just go like this to find something which is bad and then you find the first commit record after the before this and these are the transactions that you want to apply to the file system tree. And, when you apply this transactions to the filesystem tree you can free these blocks. What does freeing the blocks means?

Student: Update.

Update the start offset right. So, you basically when as you apply the transaction you also update the start offset to point to the next transaction and so on ok. So, that is basically how recovery works. So, notice that once again if there is a crash you identify a bad block by either a bad sequence number or a bad magic number and that indicates an incomplete transaction. And, from there you go back to figure out what are the last full transaction that you saw and that is the one you apply to the filesystem.

And, once you applied to the filesystem tree you although free up the space by advancing the start offset and this is a circular log so, you just wrap around and so you basically keep on using the log. It is possible that you know while you while the transaction was executing the log is completely full right.

So, let us say you are executing the transaction on the log, you are writing disk blocks on the log for this particular transaction; let us say your system call that involved deleting many files. So, lots of different writes are happening and so, you basically have a large transaction and this transaction is not being is not fitting in the log. So, what do you do?

Well. Firstly, if the size of the transaction itself is larger than the size of the entire log space that you reserved for this purpose then there is nothing you can do. So, you should and the programmer should ensure that the size of a transaction can never become larger than the size of the log; either the log should be allocated large enough about you know the size of the atomic operations that you are taking it should be should be limited, it should be bounded to some size number 1. Number 2, but still it is possible that while you are doing something; so, the you know you hit the end of the end of the log.

And, clearly you know you hit the end of the log if you ever you know reads the start offset again, you will wrap around, and you read the start offset again. So, it is very easy in memory you can figure out that you at the end of the log. So, what you can do is you can just commit the previous free the previous transactions do not commit, but they already committed.

But you free the previously committed transactions, by freeing the previous commit transaction means you apply their contents to the file system tree, and you have advance start offset. So, that the this is more space that is freed up in the log.

Why do we have the start offset? To indicate a start of the log. Why is not the log start log starting at log number 1? Because, it is a circular log right. So, I want that you know; so, there is a producer to the log and there is a consumer from the log and so, it is a circular buffer. So, you need to (Refer Time: 26:39) both head and tail right. So, let us say you just had a log with always started from 1 right. So, now if I wanted to a free a transaction, it would mean copying the just for the log all the way. So, it would have been a global copy to block number 1 to free a log.

So, let us say I wanted to the free the first transaction in the log right, then I have to copy all the transaction starting from transaction number 2 that is a very expensive operations. So, the better way to do it is maintained as a circular buffer and have a head pointer which is the start offset and a tail pointer which is the which is not tail pointer here recall right.

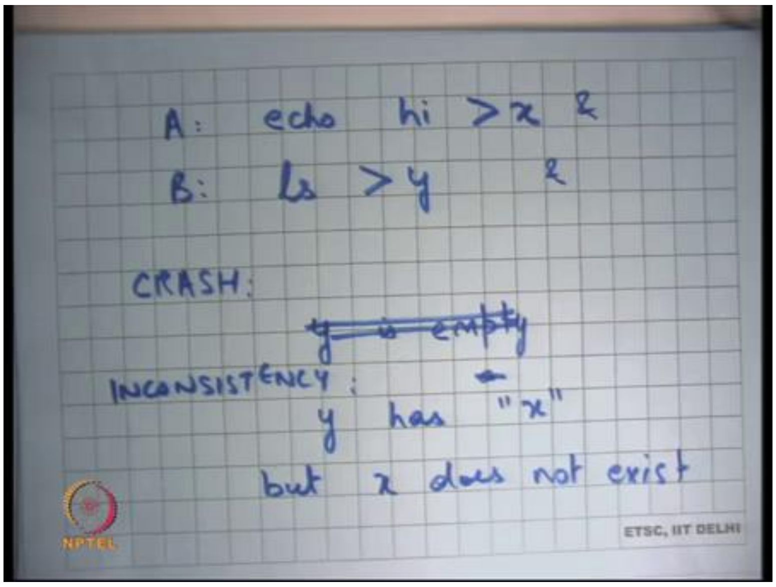
You are using you are finding you are inferring the tail pointer using the sequence number and the magic number ok. So, a scans starting at offset until bad record, scan backward to identify last committed transaction or you know the this is the full transaction, this must be a full transaction.

And, then you can apply those transactions and completely discard last transaction ok. Also, even without recovery the there should be an asynchronous thread that is bring up space from the transaction log right. The asynchronous thread works similarly, it starts at the start offset and looks at the first transaction.

You know; so, how does the asynchronous thread work? It just looks at the start offset and looks for the first transaction and freeze it up, by freeze it up it basically apply that is transaction to the file system. If it finds that transaction is really small, it can club multiple transactions and free all of them together right to have better to this scheduling; so, this is the kind of freedom it has to get better performance.

So, as I have discussed so far, we are logging the entire file system including the inodes, the free block lists, the bitmaps the and the data logs themselves alright. Let us look at you know what kind of semantics does it give us.

(Refer Slide Time: 28:51)


A: echo hi > x &

B: ls > y &

CRASH:

~~y is empty~~

INCONSISTENCY: y has "x"

but x does not exist

ETSC, IIT DELHI

So, let us say there were two operations: echo hi > x and then there is ls > y right. If you execute two commands one after another and let us say they execute concurrently. So, you know let us say I put an ampersand operator here. So, they can execute concurrently, there is not sequentially here. And so, basically if there is a crash, the consistent so, I should either see that y is let us say initially the directory the current working directory is not there. So, either I should see y is empty and right.

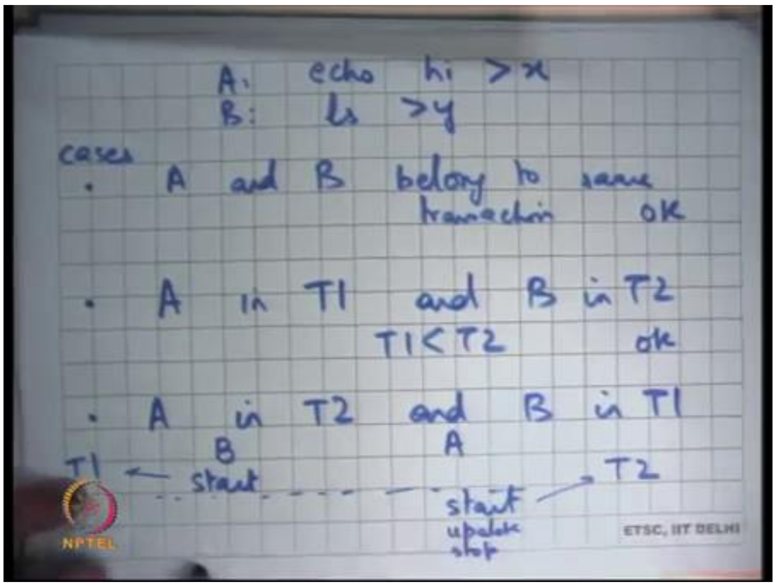
So, what I should not see let us see is that y has x written into it, but so, they could an inconsistency could be that y has x, but x does not exist. So, this would be an inconsistency, I created a file in the directory and then I did a ls to read the contents of the directory and wrote it to another file y right.

And, you know if there was no crash then definitely you will always see consistent behavior because you know all your answers, all your answers that you getting whether what exists in the file system and not being surf in the buffer crash.

But, if there is a crash then you know it you should not see this kind of a behavior that y seems to have seen the file x, but the file x does not actually exist right. So, there should be some civilization with basically says either the file x does not exist and y does not contain x or the file x exits and y contains x, but it should not be that the y contains x for the file x does not exist right. So, that would have been an inconsistence.

Now, let us see whether this can happen in the ext3 transactional system that we have discussed so far ok. So, let us say these two operations are A and B and they are happening concurrently. So, and we basically want to make sure that this kind of an inconsistency does not happen. So, we do not like this.

(Refer Slide Time: 31:20)



A: echo hi > x
B: ls > y

cases
- A and B belong to same transaction ok
- A in T1 and B in T2
  T1 < T2 ok
- A in T2 and B in T1

T1 ← start (B) - - - - - - start (A) → T2
update stop

ETSC, IIT DELHI

So, let us say so, let me just write it here A echo hi greater than x, B ls greater than y. So, let us take cases if A and B belong to same transaction. So, my question is if A and B belongs to the same transaction can this inconsistency happen on a crash ok.

So, I see some people nodding the heads; so, it cannot happen. Why cannot it happen? Because, if they belongs to the same transaction either the transaction would have committed in which case you would see hi in x and x in y or the transaction will not have been committed in which case you will not see either.

So, you will never see this inconsistency right either the transaction would commit in which case you will see both the updates and the transaction will not commit in which case you will not see any of the updates. It will not happen that one of the update is seen and the other is not seen. So, if they belong to the same transaction there is no problem, let us say case 2: A in T1 and B in T2, where T1 commits before T2 alright.

So, A is in T1 and B is in T2. So, in this case you write so, if the crash happens it is possible that T1 has committed, but T2 has not committed. In which case what you will see is that the file x has been created, but the file y has not been created at alright that is also fine right. So, there is no inconsistency of this type that y has been created and it seems to have x, but x does not exist; so, that is also not a problem at alright.

So, basically if A is an a transaction T1 and B is an transaction T2 there are few things that been happened either at the crash time none of them have committed in which case none of these updates had happened or it's possible that both of them have committed in which case both are the updates had happened.

Or, it is possible that T1 has committed and T2 has not committed in which case As updated has happened, but Bs update has not happened in which case you know the y has not been even written to the file system right. So, there is no y, but x has been created, but y has not been created. So, that is not an inconsistency, the inconsistency y has been created with the contents which are inconsistent.

So, even this is right. So, let me just say and now let us say is it possible that A is in T2 and B is in T1 right. So, A is in T2 and B is in T1, can this inconsistency happen in this case?

Student: Yes.

So, there is an answer yes. Why?

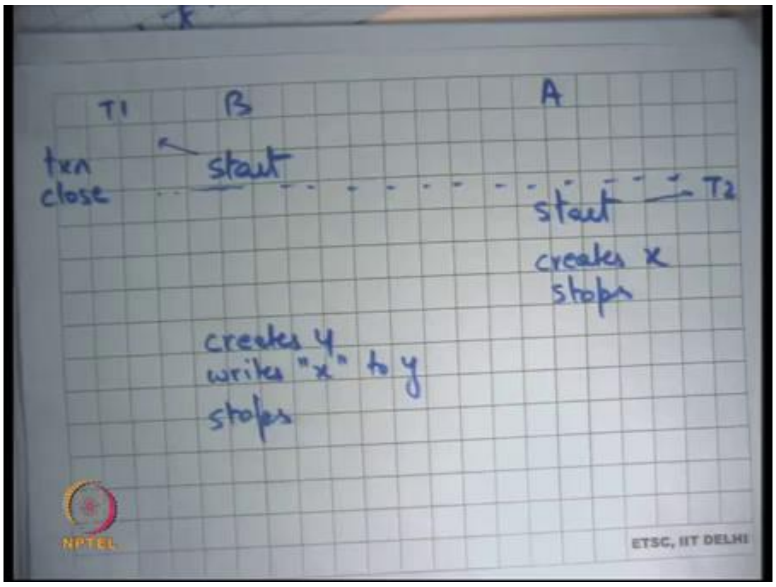
Student: Because, since we are first writing to y; so, basically not of so where the x was not there.

Right. So, it is possible that y is in; so, because it is for now if there is a crash and so, once again if there is a crash then either none of them have committed in which case we are or both of them have committed in which case we are ok. But, if T1 has committed, but T2 has not committed then y's contents have been written to disk, but x's contents have not been written to disk alright.

So, but you may say you know if y's content has been written to disk, but y must, but y must have seen x's update right. So, how can this happen? It can happen if let us say; let us say this was B and this was A. So, B started first alright and then A started later. So, this and here was the closed point; so, the transaction got closed after this.

So, this B's transaction gets into T1 and A's transactions gets into T2 right. Because, there was a transaction closer between these two starts, recall that when you close the transaction then all the existing transactions all the already opened operations are completed and all the future operations are made part of the next transaction right. So, if B's start happened before A start then B could be in T1 and A could be in T2. Now, A could have you know made its update and then stopped and then B will let me write it on a different page.

(Refer Slide Time: 35:59)



So, let us say this is B A and this is start and this is start, this a dotted line which says close transaction close right. And so, this is part of T1, and this is part of T2 and then this updates creates x and then stops alright and then this says creates y writes x to y and stops right. So, even though B could see A's update, B happens to be in a transaction earlier than A right. So, in this case B could see A's update, but because B started before A, B is in a previous transaction.

So now, it is possible that you know at this point I commit T1 and you will see B you will see the file y containing the contents x, but the x, but because T2 has not committed you do not see x in the filesystem. So, you see an in consistency and so it's possible that this happens. So, the question is in this case we are running two transaction simultaneously.

So, recall so, let us look at what are the transaction mean in the ext3 sense. A transaction means that you know only one transaction is open at anytime. All operations that have started during at this point get assigned to the current transaction right. Then you close the transaction at some point you would side have close the transaction. So, all operations that start after this will start in the new transaction.

Student: But we do not close until all the operations and then transaction are closed also.

No. So, by close I do not mean commit. So, there is a difference between transaction closure and transaction commit. So, I must decide some point where I commit the transaction right, close the transaction.

Student: But sir I am saying (Refer Time: 38:02).

Right. So, let us listen understand this. So, let us I had I close a transaction and so, there are some there is possible at some ongoing operations. So, what you are suggesting is that when you close a transaction you do not take any new operations and you wait for all the existing transaction operations to finish.

Student: So, sir I am saying like suppose in operation has bigger in current transaction like in this case we have taken start from D.

Ok.

Student: So, we will not close that transaction until we get stopped for that.

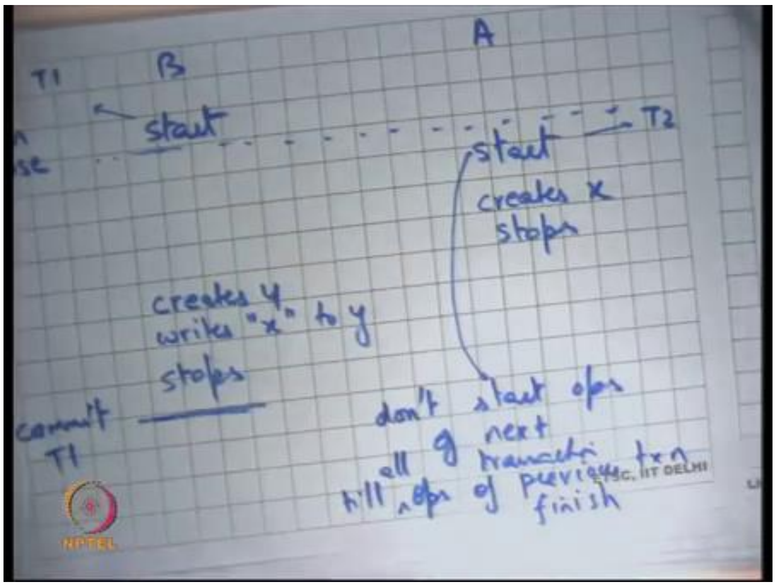
But let us say then A starts you know let us say a wait for the transaction to close till here and then you know this transaction start. So, this transaction becomes part of this existing transaction.

Student: Yes, Sir.

But this can continue forever right. So, let us say there are lots of disk operation happening. So, I will never get a chance to close the transaction right. So, the idea was that I just decide that I have close the transaction and then all the future operations become part of the next transaction. If I wait for all ongoing transactions to finish before I close the transaction then it is possible that I never get a chance to close the transaction, because there are lots of ongoing transaction that are going on right.

So, the idea is that I just decide to close the transaction which means that all the transactions that are already started they will get through in this transaction. But all everybody who has not been able to catch the bus will get the next bus basically right. So, you know that way you basically are sure that you can close the transaction at periodic interval, otherwise you will never be able to do this.

(Refer Slide Time: 39:40)



So, to avoid this consistency you basically add one more rule to this that do not start ops of next transaction till ops of previous transaction, till all ops of previous transaction finish ok. So, the idea is that you do not till this one finishes; so, it similar to what you are saying accept that you still have guarantees on you know you can choose when you want to close, you do not have to wait for things to be able to close a transaction.

So, you say that if you have decided to close this transaction before you start exuding any operation at the next transaction you are going to wait for all the operations that are ongoing in the previous transaction to finish, to avoid to prevent these inconsistencies.

So, in this case you will make sure that this start happens after this stop right, in which case you have complete you do not have any inconsistency, you have serializability in the behavior with respect to power crashes right. So, the idea is you do not start the operations of the next transaction till the operation that the previous transaction has finished.

Notice, that I am just waiting for the operation operations of the previous transaction to finish, I am not waiting for any disk writes right. These operations may happen in memory right, I am just waiting for them to finish. So, that what I read from the file system including the buffer cache is serial with respect to the previous transaction ok. So, it’s not a very long wait, you are not waiting for the transaction to commit, you are not waiting for any disk write, you are only waiting for the operation that the previous transaction to stop, yes question.

Student: Sir, we have done this actually prevents the kind of in consistency that we are talking about and there, but sir in this case what happened is that x did not, could not got written into (Refer Time: 41:41) because, there was not any x.

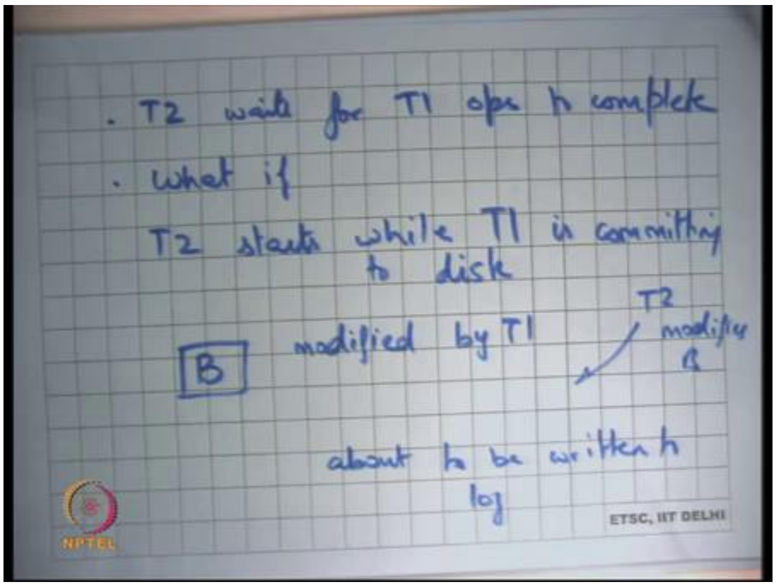
So, in this case what happens is that x did not get written into y and if there is a failure. So, it’s so, basically this in consistency you know avoided, but you are saying that in this case x has not been written to y.

So, is not it that in consistent, well that is not inconsistent; what is consistency mean there is some serial? So, there is some serial order in which things appear to have happened right. If you delay this, the serial order seems to happen the seems to that seems happen is that B happened before A right.

It is completely acceptable if B happened before A, it’s also completely acceptable if A happened before B, but what is not acceptable is that after A crash it seems like A has happened partially, but B and B has happened partially right. So, that was a inconsistency. So, A seem to have happened partially you know the contents of the directory seem to have changed when as I had read them, but actually that I directory does not seem to have that contents; so, that is an inconsistency.

But B happening before A and A happening before B both are legal. So, those that is consistent and so, what is this doing is basically making it look like B happened before A and that is totally fine.

(Refer Slide Time: 42:55)



- T2 waits for T1 ops to complete
- what if

T2 starts while T1 is committing to disk

$\boxed{B}$ modified by T1

T2 modifies B

about to be written to log

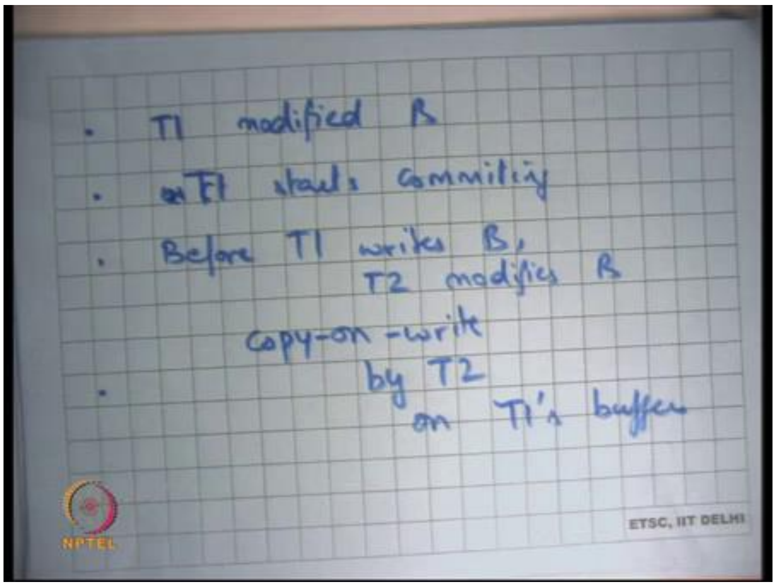
ETSC, IIT DELHI

So, basically, we said that T2 waits for T1 ops to complete right, but it does not wait for T1 to commit, T1 can you know commit later that is not a problem. So, what happens if T2 starts, what if T2 starts while T1 is committing alright. So, T1 is committing to the disk and T2 has started its operations ok. So, what has what is some bad thing that can happen?

Well, in this case what can happen is you know you have decided that you are going to write this buffer to the filesystem tree and that that buffer basically at to the log actually. So, let us say there is some buffer B that is modified by T1 and about to be written to log.

But before it is return to the log T2 modifies B right. So, if you are allowing T2 to start running before T1 has committed then it is possible that a buffer that belong to that was dirtied by T1 is again modified by T2. And so, what can happen is that if you if T2 modifies B here then what gets written to log is are the contents of T2 which include T2s modifications right. Let me just rephrase it.

(Refer Slide Time: 44:47)



- T1 modified B
- ~~on~~ T1 starts committing
- Before T1 writes B,
  T2 modifies B
- copy-on-write
  by T2
  on T1's buffer

ETSC, IIT DELHI

So, let us say T1 modified B writing T1 starts commit, starts commit means starts writing its modified buffers to the log right. So, before T1 writes B T2 modifies B. So, what can happen? The value the contents are get recorded on the log contain T2's modification and that is not something that you want right. So, what is a simple solution? So, you just make a copy of this buffer B right.

So, you basically say that this buffer B belongs to the commit buffers of a previous transaction and these buffers has not yet been written to the log. And, if there is a if so, you do copy on write sum basically by T2 on T1's buffers ok. So, if T2 tries to write to a buffer that belongs to T1 and has not yet been written to disk you make a copy of it right.

And, it is a most recent copy that is the that is the valid copy, the previous copy you know it's only for T1 to commit to disk yes question.

Student: Sir, why cannot we use the logs, you are already having the logs or buffer?

Why cannot we use the logs? That is an interesting question; you want to keep a log on all these buffers while you are committing this transaction T1.

Student: While, the transaction is while I am writing by buffers at this program, I will release the logs only at the transaction gets committed.

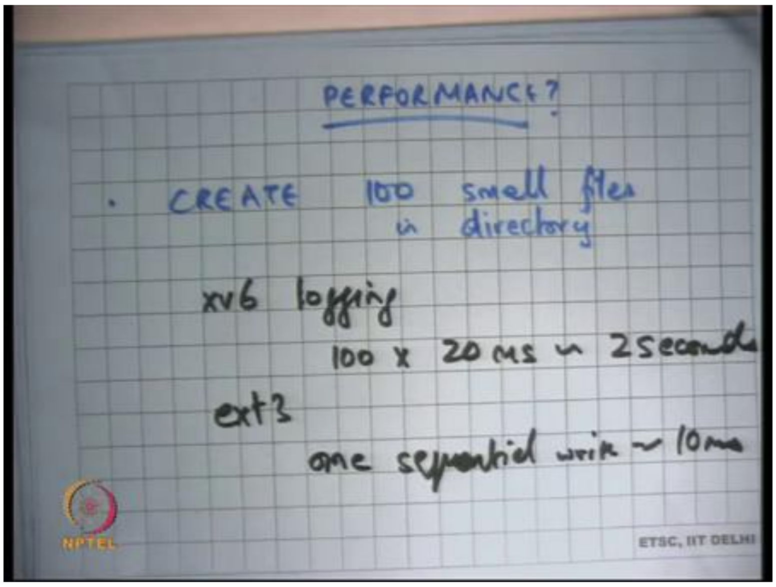
So, here is the suggestion I will release the logs only when the transaction gets committed, is that an option? Well, that is an that is seems like an option to me, you know you basically making sure that till the transaction commits you basically log all the buffers that belong to the transaction. But, is not it that too long to hold onto the log of a buffer, right?

You basically want that thing should happen concurrently as much as possible, imagine that you know you are constantly writing a you are constantly modifying the same file or you are constantly adding or removing files from the same directory..

Then you know if that directly happened belongs to the previous transaction you are holding logs to it and next transaction just gets completely serialized with respect to the previous transactions. So, that is not a good idea, the better idea is to basically you know do this versioning of the blocks. So, you are basically doing versioning you are saying this block has you know has the T1's version of it is this and this is the current version of this block.

And so, that is some much more efficient way of doing things. So, its once again I think it is a classic trade of example of a trade of between transactions and logs right. Here transactions are provide more concurrency and yet come at a small cost right because you are because here transactions involved disk accesses, so doing this figuring out of you know what is needs to be copied etcetera is possible alright.

(Refer Slide Time: 48:00)



PERFORMANCE?

- CREATE 100 small files in directory

xv6 logging

100 x 20 ms ~ 2 seconds

ext3

one sequential write ~ 10 ms

So, let us quickly look at the performance of ext3. So, let us say I create 100 small files in a directory. In the simple logging or I am going to call at the xv6 way of logging, xv6 logging it would have involved each create would have involved 5 or 6 or 8 disk writes and a commit. So, each write would have been 1 commit and that would have been let say 100 into 5 or 6 writes.

So, let us say you know 20 milliseconds is roughly 100 into 20 milli seconds is 2 seconds right. Is that right? And, but with ext3 all these 100 small files creation happens on in inside memory and then there is one sequential write that is 100 10 milliseconds.

So, 2 seconds versus 10 milliseconds and then there is an asynchronous copy from the log, or you know asynchronous application to the filesystem tree, which is not in the critical path and so, that is can be done later. So, it much faster in that sense. Also, you know I say one sequential write, but you may want to do two sequential writes.

One sequential write for writing all the blocks in one go and then after they have been written then you write the commit record after that. You do not want the disk to also schedule the commit record right, you do not want the disk to reorder the commit record with respect to the other data blocks.

If it does that then you have a very bad situation, the commit block has been written, but the data blocks have not been written. So, you first give the disk all the data blocks to be written and the descriptive blocks; after that have been written the disk say I have written them then you give him the commit block. And so, there may be two revolutions there ok.

Let us continue this discussion or finish this discussion next time and I am also going to discuss security and access control next time.