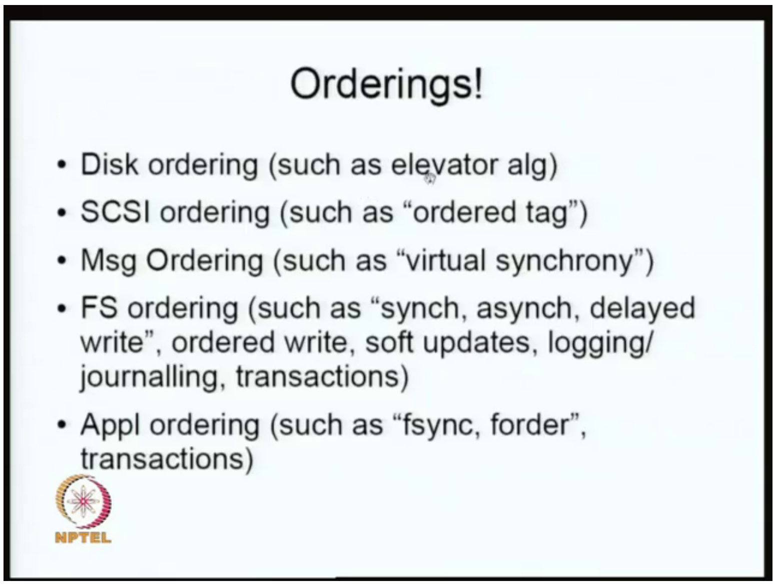
Storage Systems
Dr. K. Gopinath
Department of Computer Science and Engineering
Indian Institute of Science, Bangalore

Theoretical Foundations
Lecture – 37
Theoretical foundations of Distributed Storage systems_Part 6: Orderings in Storage systems, Orderings in File systems & Applications

(Refer Slide Time: 00:38)



Orderings!

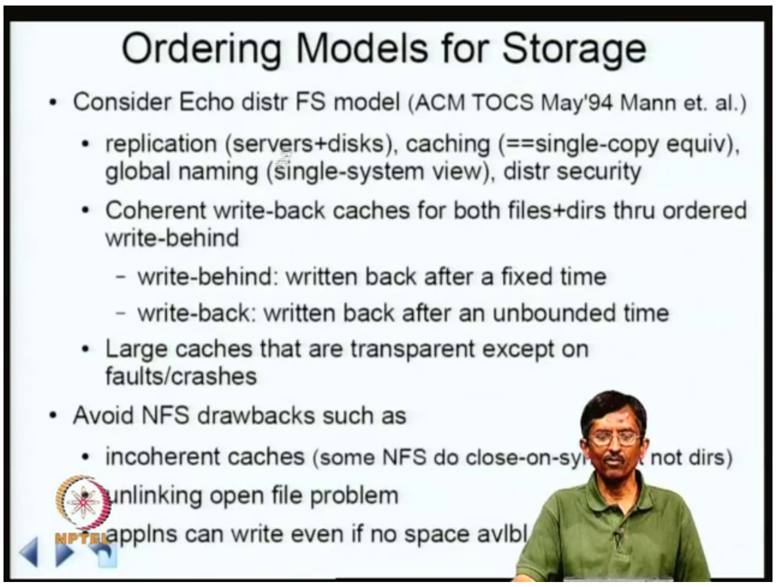
- Disk ordering (such as elevator alg)
- SCSI ordering (such as "ordered tag")
- Msg Ordering (such as "virtual synchrony")
- FS ordering (such as "synch, asynch, delayed write", ordered write, soft updates, logging/journalling, transactions)
- Appl ordering (such as "fsync, forder", transactions)

Welcome again to the NPTEL course on storage systems. So, so far we have been looking at some aspects of ordering; let me just summarize; what all we have done so far; various steps ordering set we looked at the beginning, we looked at disc ordering which is then by elevator algorithm, SCSI ordering these are the protocol level, but it closer to the do as level itself and that you just thinks like order tag etc. We next looked at message ordering such as virtual synchrony that we just listen to you, we took a bit reasonably close look at this form and now we have few more types orderings at the file system or kernel level and the application level, again as we discussed before file system also order certain things, we get we do it by synchronize or delayed writes, we can also do order writes, I will come I will briefly talk about soft updates later you also have motions of logging on transactions.

The various types of ordering that also happens here, that is also one more at the highest level might call it application level ordering and this things are things like fsync and we looked at one model called Forder which one distributor fall system called ehos, which we also discussed briefly last time. So, we can see that there is lots of ways in which the Io get reordered and therefore the exact semantics is not real to what throw all this layers is incredibly complicated and because data is extremely crucial for the function called last systems. There is a lot of conservativeness in all this aspects because, anything something gets wrong you just data may be completely corrupted and if you are unlucky your kernel data or kernel pristended with the get corrupted, then the system is usually gone said by there is lot of carefulness with this all the system.

So, we look so far up to message ordering; now I want to start looking at these aspects.

(Refer Slide Time: 02:47)

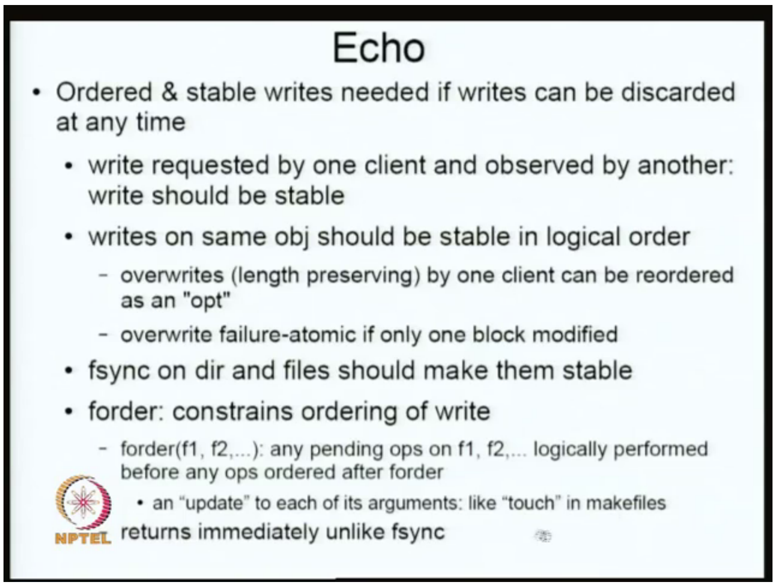


Ordering Models for Storage

- Consider Echo distr FS model (ACM TOCS May'94 Mann et. al.)
  - replication (servers+disks), caching (==single-copy equiv), global naming (single-system view), distr security
  - Coherent write-back caches for both files+dirs thru ordered write-behind
    - write-behind: written back after a fixed time
    - write-back: written back after an unbounded time
  - Large caches that are transparent except on faults/crashes
- Avoid NFS drawbacks such as
  - incoherent caches (some NFS do close-on-sy not dirs)
  - unlinking open file problem
  - applns can write even if no space avlbl

So, one model I wanted to the what I introduce last time was the one distributor fall system called echo and important thing about this particular model is that it has got a notion of write behind, again what is write behind? It is same as write back except that your write back can take any arbitrary amount of time, write back takes an arbitrary amount of time where as write behind always takes back happens with a fixed amount of time. For example, it can happen if there is something dirty in memory it will be flush to disc within a specific period for example, it could be 30 seconds, 15 seconds or 1 second, this is; what is called fall system hardening. In fall system hardening what is drew try to do is it try to limit the amount of dirty data that can be lost because of a crash in the system. So, basically this particular model is was write behind and is a well known technique, but it tries to avoid NFS problems and we also looked at some of the high level ideas that this particular system should have.

(Refer Slide Time: 03:53)



Echo

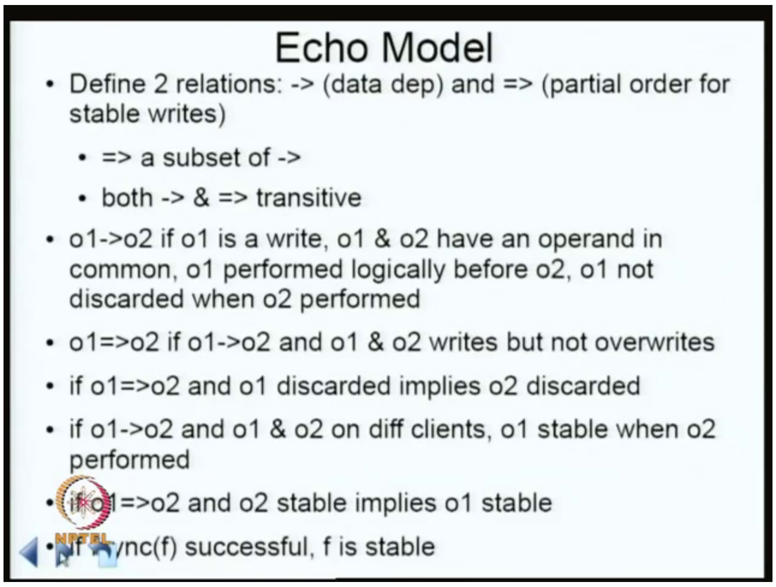
- Ordered & stable writes needed if writes can be discarded at any time
  - write requested by one client and observed by another: write should be stable
  - writes on same obj should be stable in logical order
    - overwrites (length preserving) by one client can be reordered as an "opt"
    - overwrite failure-atomic if only one block modified
  - fsync on dir and files should make them stable
  - forder: constrains ordering of write
    - forder(f1, f2,...): any pending ops on f1, f2,... logically performed before any ops ordered after forder
      - an "update" to each of its arguments: like "touch" in makefiles

returns immediately unlike fsync

For example, write requested by one client and absorbed the another write should be stable, this across 2 nodes on the same node write some same a object should be stable in logical order and we said that you can have fsync or the application level and echo introduces a new model Forder which constraints ordering of write. For example, if I say Forder f 1 f 2 etcetera what it means is that any pending operations on f 1 f 2 or completely performed before you think of proceeding further. So, again this can similar to this is actually an interesting operation, it does not do anything it just does ordering that is all it does similar to it. We are use to make files you have most of touch basically, touch is basically what it basically changes the time stand that is all it does, those make file you know that there is an important thing called touch it similar to this, all it does is it does a null updates on this changes a time, that is all it just basically says that this happened at this part this is logically happening at this time that is all it says.

So, it is a purely some kind of a null update. So, that you know that some updates has taken a actually nothing has changed though basically what it says, another thing about this is that it returns immediately unlike fsync, basically bits for the disc operation to be completed before it returns back. So, Forder is basically a way to make sure that certain things are sent to disc and particular order and this quite important because we are talking about the distributed file system and different parties could be communicating through files and whether that particular information returns a file makes it to the other side or not is very critical; for example, if a do lot of writes on my crash nobody knows nobody notice it is equivalent a crash because, I wrote something but in order to make it to other side nobody saw it. But if I saw it somebody have saw it now it better be stable on the side that is sent it because otherwise system equivalent a crash ok.

(Refer Slide Time: 06:14)



Echo Model

- Define 2 relations: -> (data dep) and => (partial order for stable writes)
    - => a subset of ->
    - both -> & => transitive
- o1->o2 if o1 is a write, o1 & o2 have an operand in common, o1 performed logically before o2, o1 not discarded when o2 performed
- o1=>o2 if o1->o2 and o1 & o2 writes but not overwrites
- if o1=>o2 and o1 discarded implies o2 discarded
- if o1->o2 and o1 & o2 on diff clients, o1 stable when o2 performed
- if o1=>o2 and o2 stable implies o1 stable
- if fsync(f) successful, f is stable

So, that is why there is that is why echo has got this particular model, now again we briefly we went through it last time. I just we just go through at 1 more time it has 2 relations 1 is the standard data dependency it could be within a node and across nodes, but this 1 is a partial order for stable writes, the fact that there is a data dependency does not mean, I succeed in completely writing it all through to a disc, I am it only signed up portion of it.

And this is only for stable writes it is not for other op other data and for this reason because it is only for stable writes this is subset of this and this like your listed inputs transitive relationed across events, this is also we can introduce a relation which is transitive which is closed on either this or this operation. So, basically namports model was with respect to messages this is going to be with respect to dependencies and this one is specifically tell at this particular one specifically tell it for stable writes. What is this one saying o 1 arrow o 2? this means o 2 is dependent at o 1 and that means o 1 and o 2 have a operant in common and o 1 is a right and o 1 has to be done logically before o 2 and in case o 2 gets done.

That means o 1 has been o 1 cannot be discarded because a right that means, if it has dirtied some the buffer cache that contains cannot be discarded, it has to make it little bit discs that is what this is this basically some kind of guarantee. Again this particular course system has a small optimization for that reason there it turns out that, there is this particular rule o 1 double arrow o 2 and this basically saying that in case, o 1 arrow o 2 and it has to be that o 1 and o 2 have to be writes, but not overwrites basically because this particular system they have decided that this is an interesting optimization possible, which is mostly applicable to most systems that you do not care about if you are having overwrites you do not care which order they go in.

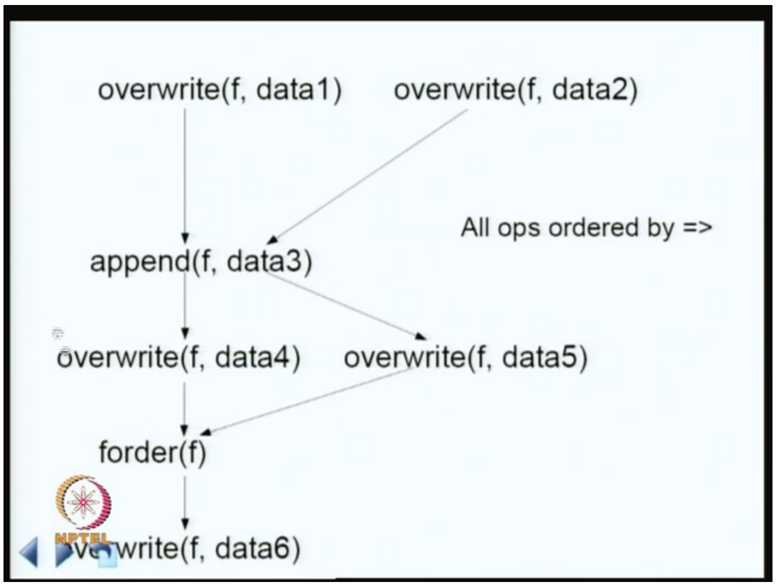
So, that is why they decided that it is that we do not have to constrain the way overwrites go through, they can happen in any order in case you need that constraint also it is always possible using Forder you can always order it through Forder is there any. So, I am only saying that o 1 double arrow o 2 only in case o 1 and o 2 both are writes, I do not care if it is writes I do not care ii will not have this particular thing, whereas in the data dependence it may be the that is important that is why o 1 o 2 is a this double arrow is subset of this single arrow and this 1 is there princely because we talk on stable writes; if o 1 has been discarded it better be that o 2 has o 2 will also be discarded it is very critical, if o 2 gets does not get discarded that means, again there is some inconsistency in the system essentially some bodies unrolled over if that happens which is not but wanted, again if o 1 or o 2 and o 1 o 2 are different planes because here distributed file system.

This can this operation can be on different lines o 1 stable and o 2 performed and basically what is it mean? it means that if o 2 has seen something then it better be that o 1 is stable on this mode otherwise somebody has seen something and even that was there on some other node actually disappeared, again the same consistency if this also not right and these are major important because if you are using normally in regular machine often times you use files a way to transmit information.

Now I want to have the same semantics in a distributed file system. So, it better be that in the similar kind of model is present also here so that I can still rectifies and they can communicate through files, what a write in 1 file other parties able to see it; that is why and this is guaranteed that still is the case and o 1 double or o 2 and o 2 stable implies o 1 stable ok.

This is the most let us say with basically says that if o 1 has been center disc; sorry o 2 has been available in disc o 1 also even disc and fsync of f is successful f is stable.

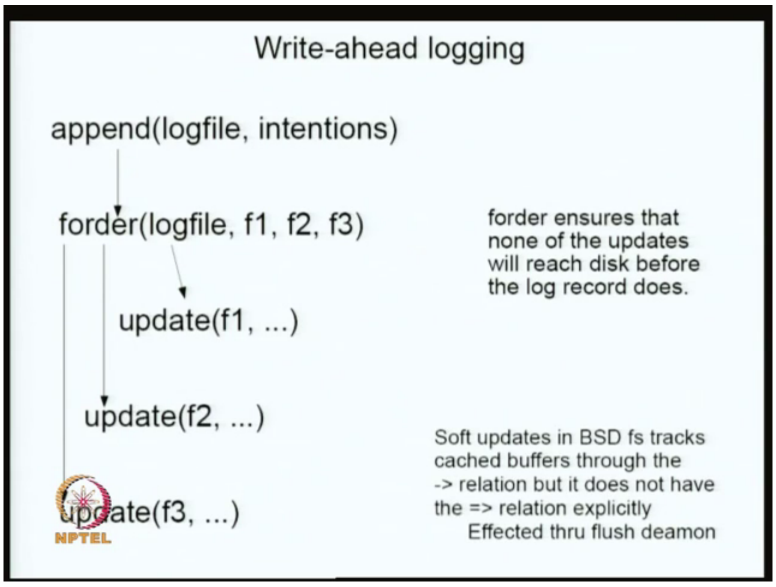
(Refer Slide Time: 11:45)



So, terms out that each of these things takes sort of some particular interesting case and therefore all these things are required, take an example you have overwrite f with data 1 again overwrite f with data 2 and as I mentioned these are overwrites; that means, that you are not changing the size of the file and as far this echo ordering semantics this can go anyway order, we do not care whether this guy goes to this first or followed by this, but if it is append I want to ensure that suppose I am interested in ensuring that, this 1 happens only after all these things have been center disc.

So, I have this kind of model and so because there is dependency between this operation; this operation there is a dependence because you notice that o 1 depends sorry o 1 arrow o 2 if they have an some operand in common. So, this operand f is common therefore if this is done logically before then this has to be after, similarly f these 2 things also can be done in parallel because, I do not care about overwrites being in a way they go in parallel way in much order they go. Whereas, if I am interested in making sure this overwrite happens only after these things, these 2 things, then I can ensure this is something called Forder and what is guarantees is that we have to make sure that this are completed before we can get to this 1, again this is dependent on your model the applications models about how things should be fixed to disc.

(Refer Slide Time: 13:38)



Write-ahead logging

$$\begin{tikzcd}
\text{append(logfile, intentions)} \arrow[d] \\
\text{forder(logfile, f1, f2, f3)} \arrow[d] \arrow[dd] \arrow[ddd] \\
\text{update(f1, ...)} \\
\text{update(f2, ...)} \\
\text{update(f3, ...)}
\end{tikzcd}$$

forder ensures that none of the updates will reach disk before the log record does.

Soft updates in BSD fs tracks cached buffers through the -> relation but it does not have the => relation explicitly
Effected thru flush deamon

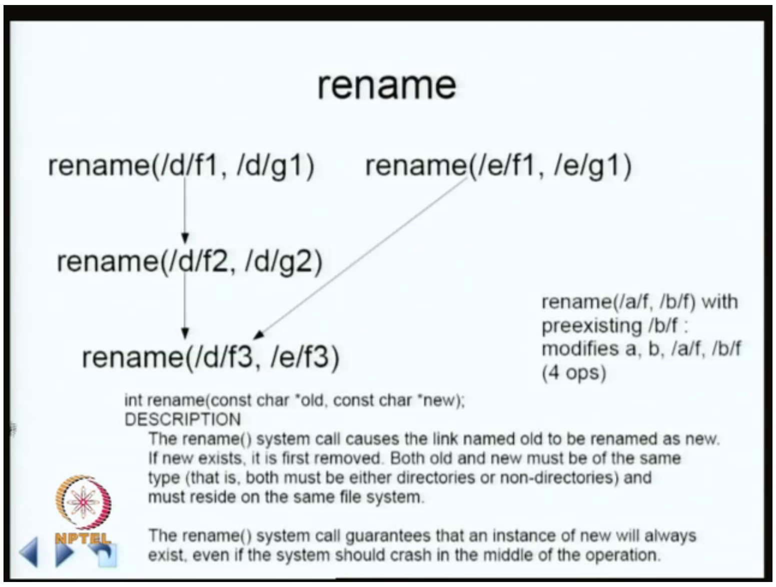
Now, with this thing we can essentially implement things like write ahead logging, what is a write a head logging? You append your intentions means what are the things that you going to update to log file and then you say Forder log file.

Let us say it is updating f 1, f 2, f 3 this particular operation whatever it is doing, whatever file system operation that you are planning do it actually updates f 1, f 2, f 3. Then a basically have Forder log file f 1, f 2, f 3; that means, that the log file should be flushed to it is before a doing any of these things logically, that is if this has become stable that means, this also would definitely has to be stable there is no question about that is what I am asking you to do and by putting f 1, f 2, f 3 I mentioned Forder is essentially doing some kind of touch operation on these things, therefore this operand is earl with op modification though Forder is earlier than this one, therefore this has to come before that is why this is ordered later than this one, similarly this operation order later than this 1 this oper. But notice that there is no connection between these 3 guys, they can go in any order they like not bothered about which order this 3 things go only thing, I am interested is this 1 this has to go before any of these 3 things. So, the sense you can express certain trains of concurrence that you want in a system and this Forder gives you that handle. So, basically this Forder ensures that none of the updates I will reach disc before the log record does.

So, it is been locked completely then only it moves, now some this similar this also possible and actually turns out in the BSD file system they have, but they have file system the 4.4 first file system, the later version there was an earlier version also which did not did not have this, but the later version post 2000 has something called soft updates and in this particular file system all the cached buffers are tracked through the dependence relation in the kernel; they actually the kernel actually tracks all these things. So, the buffer cache mechanism and anytime you flush any of the buffers you make sure that it is always a consistent cut of the relationship, that is you never ever do it in such a way that look has the if in there was a crash, it looks as we have gone back in time that is something which they will make sure does not happen

So, you group the graph structure in such a way that there is always in something's forward motion, now this turns out to be complicated also. So, I am not going to do in a details, but it turns out because a sharing there can be some circularities and this circularities have to resolve and essentially this soft update mechanism does some localized un dos. So, that you still get a some kind of a let us say a direct graph of a some kind I do not have time to go through it, but it turns out soft updates also does some ordering, but is then all the all this is done using the color mechanisms of flushing.

(Refer Slide Time: 17:11)



rename

$$\begin{tikzcd}
\text{rename(/d/f1, /d/g1)} \arrow[d] & \text{rename(/e/f1, /e/g1)} \arrow[dl] \\
\text{rename(/d/f2, /d/g2)} \arrow[d] & \\
\text{rename(/d/f3, /e/f3)} &
\end{tikzcd}$$

rename(/a/f, /b/f) with preexisting /b/f : modifies a, b, /a/f, /b/f (4 ops)

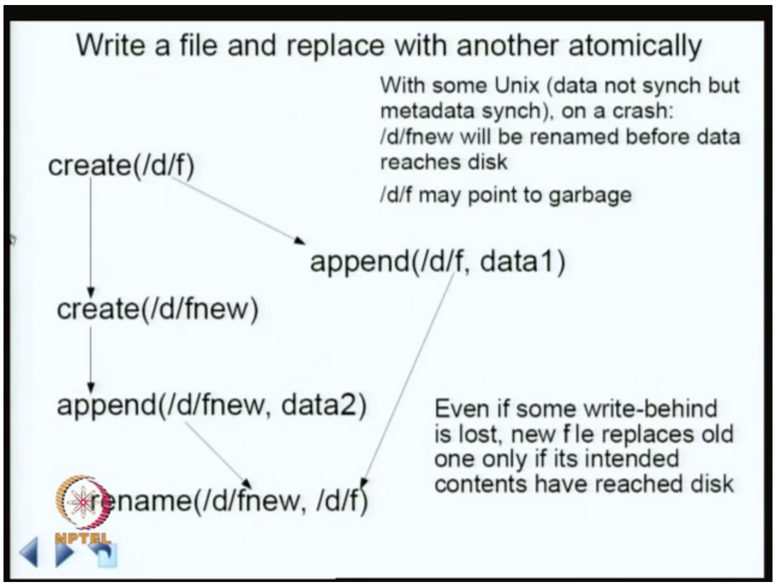
```
int rename(const char *old, const char *new);
DESCRIPTION
    The rename() system call causes the link named old to be renamed as new.
    If new exists, it is first removed. Both old and new must be of the same
    type (that is, both must be either directories or non-directories) and
    must reside on the same file system.

    The rename() system call guarantees that an instance of new will always
    exist, even if the system should crash in the middle of the operation.
```

Now, let us quickly look at other possibilities, suppose you have the notion of rename actually rename is some is one of the most complex file system operations and for example the definition is here we have a source and destination. You can say it as if you look at the description carefully, it is got lots of details both must be directories or both can both have files they should result in the same file system. So, that is also some guarantees about crashes, you know system crashes in the middle operation they were either say that it had the old or new ok

So, they have lots of conditions here and as I as we can see here right; you will find that anytime you do rename it can touch multiple things for example, you can in this case remain af and bf. For example, with this thing existing previously you notice that you are what are we going to do, I am going to take this particular file and going to delete the old file f here and slash b it is a different file and I am going to take this file and put it into these directory, now h director has changed because this file has disappeared from this directory, these directory has disappeared as changed because a new file has come in which is got the same name as the previous one, but it is a different file that also is gone and this file is gone sorry, yeah this file has become this new file and the old file f is gone. So, the 4 different changes a changed b has changed this file f has disappeared on this fs come here so modifications, so that the clash can occur any place any time. So, if you look at so what this let say the file system guarantees some atomic silicon condition.

Let us say the file system already guarantees so they are now talking about how to order rename sensors. So, there is multiple renames going on example rename slash d f 1 slash d f g 1 rename slash e f 1 slash g 1, what is that none of these operands have anything in common so they can go in parallel. But this one shares the d directory between this therefore, they have to be ordered, this 1 has both d directory and e directory therefore this one has to be ordered this way. So, what you are guaranteeing is that if because all these are ordered through that double Ro relationship, if this are become stable you guarantee at this stable, similarly if this is already stable then all these guys are also stable .

(Refer Slide Time: 20:19)


Write a file and replace with another atomically

With some Unix (data not synch but metadata synch), on a crash:
/d/fnew will be renamed before data reaches disk
/d/f may point to garbage

Even if some write-behind is lost, new f le replaces old one only if its intended contents have reached disk

$$\begin{tikzcd}
\text{create(/d/f)} \arrow[d] \arrow[r] & \text{append(/d/f, data1)} \arrow[dddl] \\
\text{create(/d/fnew)} \arrow[d] & \\
\text{append(/d/fnew, data2)} \arrow[d] & \\
\text{rename(/d/fnew, /d/f)} &
\end{tikzcd}$$

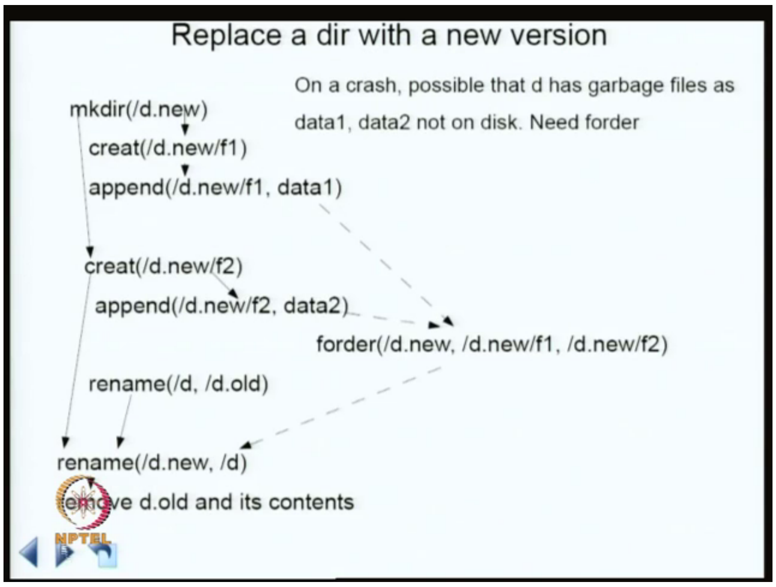
Now, we will let us consider one important EDM that is widely used write a file and replace it with another atomically, why is this so critical because all of us is editors, editing it is very critical that you are working on a particular file f you update it was update is completed, you want to replace the old file with the new file atomically.

What you are interested is making sure that whatever happens I should not lose my updates if possible and most important it is not the case that I lose both the old file and the new updates, that is a catastrophe and I want to avoid it. So, that is reason why I need to replace with another atomic. So, when I do vi when I do write and exit somebody has to do this when you are using e max, somebody has to do this if you notice in earlier file systems that is before 1980. For example, you could lose files this one of the things that most people were scared of, but now most file systems take a because will be closed this is the most unpleasant in that one can imagine. So, this can be done in our system by the following. So, I create a file I add some data to it I create because I am going to replace this is another thing, so it is basically in the same directory I am going to create a new file and then. So, this was some you might call it some version of the file.

Here what I am doing is I am not really doing editing job here I am just talking about 1 files is just been created and something has been important to it here, I create another file add put some other data into it and I want to rename it; basically what I want to make this whatever is data is here should actually present in slash df. So, because of this double arrow relationship even if some write behind is lost because, if it is lost what is it mean means that and guaranteeing the registering one of this things died, then I will not get to this point that is a guaranteeing with the double arrow. So, even if some write behind is lost new file replaces old own replaces old 1, only if it is intended contents or each disc the problem with many. Let us say some unique file systems in the past has been that usually the data was not first to disc immediately because of write behind or delayed writes or re synchronous writes, but metadata always a synchronous; that means, that which are meta data here this a meta data operation.

In this case what will happen is that you do something here this goes to disc right away, this also can go to disc right away where as these 2 stuffs could in memory and that means, that if there is a crash this slash is f could be pointing to something which was under related to what is actually put in that is possible. With this things it is possible for us to avoid this with the double arrow you can always order it right here we do not need n Forder, but this ordering the because of their relationship between these 2 we can still ensure that this happens earlier is become stable then this becomes stable later.

(Refer Slide Time: 24:25)



Replace a dir with a new version

On a crash, possible that d has garbage files as data1, data2 not on disk. Need forder

```
mkdir(/d.new)
  creat(/d.new/f1)
  append(/d.new/f1, data1)

  creat(/d.new/f2)
   append(/d.new/f2, data2)
                         forder(/d.new, /d.new/f1, /d.new/f2)
  rename(/d, /d.old)

rename(/d.new, /d)
remove d.old and its contents
```

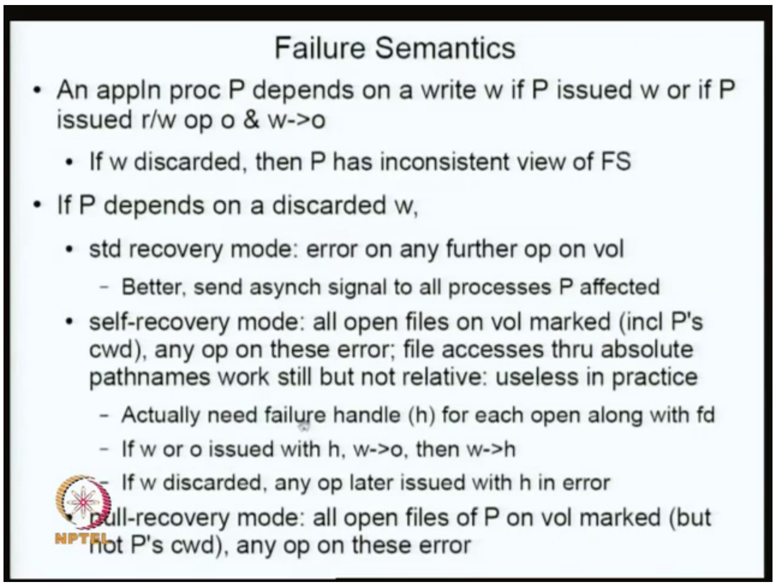
Now, if suppose sometimes you have this notion also replaces a directory with a new version, it is also a common operation suppose I have a installation on a particular application it is called a Firefox, you know that firefox keeps changing. So, you have a directory with all kinds of stuff and you want to replace it atomically with new firefox information, we want to make sure that this happens correctly in spite of any crashes. So, let us look at exactly the situation here, so make dir slash d slash mu this is the new directory. So, I want to make sure that I cannot be keeping on writing arbitrarily into the firefox directory because, I might write something and it die or anything can happen. So, that way means I do not know what has happened it becomes so what will do it a new directory completely?

So, I do it a new directory and I create some file their append data to it and then here I am taking this is the first directory this is the second directory, again I create a new directory sorry I made a mistake. I create a new file and append this, now once this is done I want to rename if I think I made a mistake here just a minute; if you look at this one. So, there is this operation mkdir created a new file there. So, you created 1 file f 1 here you created another file f 2 here you put data 1 into this new directory and you created some data d 2 into this file f 2 and you have an old file and you basically the directory d and replace the contents of that d old with d, now comes this new directory. Now I am interested in renaming this new directory as this particular directory which also actually has replaced the previous directory. Now our issue is that if there is a crash it possible that data 1 and data 2 they are not on disc and the only way to do it is by using this Forder which basically make sure that this and this they get stable.

Because this is before this you are guaranteed that these 2 things become resident and disc then only this operation will precede and let this particular operation proceed. So, normally with if you do not have Forder, it may be that you will the d has from junk files in it because, data 1 data 2 has still not made it to disc because disc this path could be quite fast, make they are create with Forder you guaranteeing that this happens in same order.

So, let us invite this is happened is because you notice there is no connection between d and this operations there is no operands are common, that is why there is no way for the system to order it; even with the even if you think that there is a if you take this standard greater dependence. So that say then why you will find that these 2 data may not be resident and you might actually directly go here and if there is cash then d must be pointing to some might have some files, which are not the correct data.

(Refer Slide Time: 29:27)



Failure Semantics

- An appln proc P depends on a write w if P issued w or if P issued r/w op o & w->o
  - If w discarded, then P has inconsistent view of FS
- If P depends on a discarded w,
  - std recovery mode: error on any further op on vol
    - Better, send asynch signal to all processes P affected
  - self-recovery mode: all open files on vol marked (incl P's cwd), any op on these error; file accesses thru absolute pathnames work still but not relative: useless in practice
    - Actually need failure handle (h) for each open along with fd
    - If w or o issued with h, w->o, then w->h
    - If w discarded, any op later issued with h in error
  - pull-recovery mode: all open files of P on vol marked (but not P's cwd), any op on these error

So, basically what echo failure semantics is this follows, an application process here P depends on a write w if P issued a rewrite operation oh and the w arrow o ok.

So, basically if P has issued this 1 is clear if because P you should write it assumes that it is going to be sitting on disc, that is a elementary observation because P issued at therefore it has to be there or if it. So, happens that P issued some operation o and this o dash actually depends on w; that means, that since it has issued it has been completed somehow now because this operation could actually be read also it could be a read; that means, that P actually depends on write behind stable.

If it is not there then there is something essentially again P will have a inconsistent view of a file system. So, what echo it is following if P depends on a discarded w because P depend on w, but it turned out that w has actually discarded, if that is a case the various recovery modes they have I just go give you some kind of idea in a practice, things can much more complicated by I am just giving those some simpler once what they have come up with.

The standard recovery mode error or any further operation now. So, basically what is happening is that they was the process issued a write or it dependent on some of it depended on the write? So, P depended write in some way because of that crash or whatever it could not guaranteed. So, when it comes up you want to make sure that some recovery does is happened, but the same time there other notes on system could also be doing operations, since on the particular thing which is on other parts of system because notice that this particular file could be cached from other places.

So, what you planning to do is you go basically saying that anybody does any operations on those related files for which the write behind was dropped, you want to make sure it is an error ideal thing would be to send a signal to all the process is affected. So, but in the echo they actually just say that any further operation is an error, there is an why this is not a good solution is because this immediate as soon as this is notice happens you are telling the process this happen, here it is only noticed when we actually do the next file system operation can happen much layer, so it is not such a good thing.

In self recovery mode the idea basically is that why constrains, everybody the only thing that we are worried about is a open files, now the open files only they might have lost something because somebody crashed somewhere. So, instead of saying that any operation is a problem on that set of directories or parties of the file, we are only bothered about only those files which are open and also they had some other model in mind, the model was that is there is a problem if the application restarts it will start using absolute pathnames again, therefore those should the restarted application should be able to proceed.

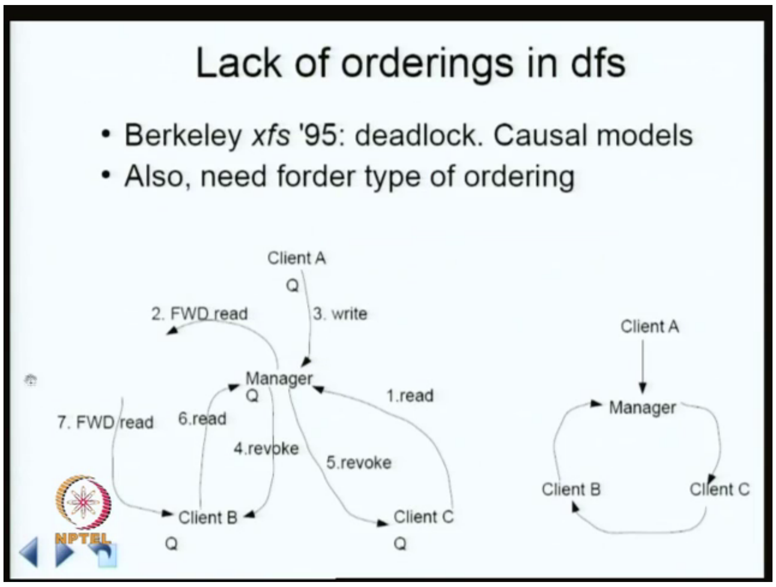
The old process had some difficulty fine something was lost, but I in the availability I will just assume that the new process a new process started and the new process typically at the beginning at least before it gets a file handles etc; we will do using absolute path names and therefore, it could be but actually turned out.

The fine distinction was not really used in practice therefore, it did not talk about actually what you need is something like most settle; what you need is something called a figure handle. That is for each open operation this like you get a file descriptor; we need another handle what you call the failure handle. So, if a write or some operation o is issued with this handle; that means that this and w arrow O, then the basic idea is that this h also depends on w. So, basically what are we saying if there was a write that was issued with h? So, first already is again one thing you opened it sometime in the past, now after sometime you are being a write and this write was issued with h as this another operation that are issued with h; that means, that there is a dependence between if there is a dependence between w and O, we say that h is actually dependent on w. So, basically what we are saying is that anything they trying to mark all operations.

That are in somehow connected with what we should do in case there is a when that write is actually discarded, if w is discarded then any operation issue with h is going to be in error because h is the thing that keeps track of what really happened and the some certain different it is 1 more thing they talk about null recovery mode, which basically the problem with this was that all open files it includes unfortunately the process P also because, when if a you are you have a current working directory and the process has a current working directory. So, that files that directories also open; now if you mark that also I say accessible then you have no precious stand because causing a problem actually, that is why you need to make sure that P is current working that it is not marked ok.

So, there is some certain differences between some of this modes and so it turns out that, if I really want to take your application needs because lot more things you have to do tomorrow to handle the situation correctly.

(Refer Slide Time: 37:05)



Lack of orderings in dfs

- Berkeley *xfs* '95: deadlock. Causal models
- Also, need forder type of ordering

So, we just looked at aspects relating to storage ordering again, let me just show a distributed file system where both the orderings are important and this is I can this research file system that has developed at Berkeley about mid nineties and you can see that if you have a lack of ordering, it can create some problems either you can dead lock or as you saw before we can get inconsistent inconsistency. So, this an example of how you can get a dead lock; what is happening out here there is a client here client c, client B and the some manager you can assume as the party who gives access to a exclusive access to files, that various parties trying to update certain things and you go through a manager who keeps track of consistency requirements.

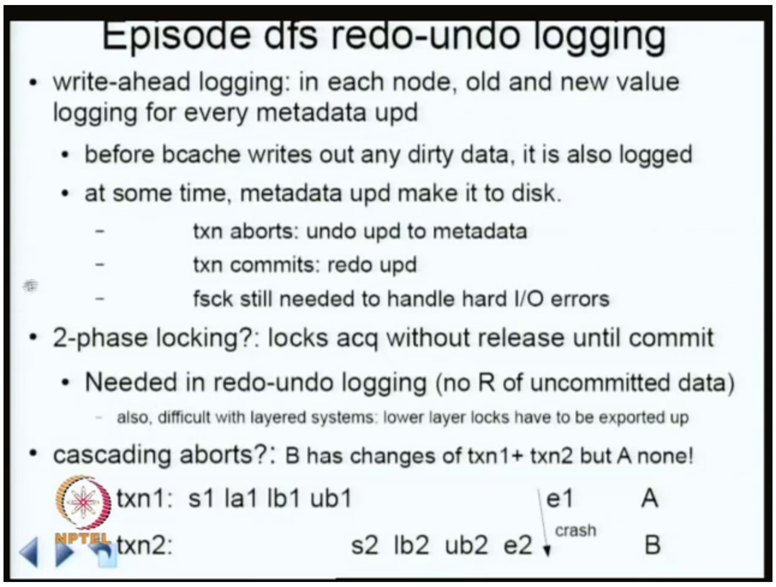
So, what is this first operation if they read let say that this file actually is crashed by b, now in this system that the notion of something called forwarding read; that means, that client C request a read this read is forwarded to b, but there is a problem in the network whatever he does not go immediately that is why there is gap here, it takes some time it is not instantaneous it is taking time. In the mean time before this forwarding read made it to client b, you get a situation where client has made a request to the write now this manager seeing that 3 has made a request for write and it knows that somebody has got a thing, he does not know who it has got it could be either b or c it generally sends a revoke to everybody other than the party who requested. So, sending report everybody it goes to this it goes to this. Of course, 5 this revoke coming to c it drops it because it knows it does not have it, just it is an offer it for 4 this 4 part when comes to b it actually has to it is buried for this.

So, it has to actually flush it through it has to essentially take it is dirty data and make sure it is flush to disc that is what b has to do and let us say just after sometime b actually has a read. Now it has to go through the manager again because this is a distributed file, it has to have get the token so that it can have access to it. Because you going to have what is called multiple disc and have a read lock, but only a single writer can have a write lock. So, that is why is trying to get the read lock from here and so, after 6 let us say that magically the same what you did here finally, comes here. Now this point system is dead locked why is it dead locked? Client is waiting for the manager to do something about the write it is requested client C write what is it doing? It trying to do a read and it is. So, try to do a read and it is waiting for B write because client C is read became a forward read, but that came it is sitting in the Q much later than the other revokes and it basically waiting for client B is waiting for the manager Q to give it the file does not even have it therefore, B is actually waiting for the manager. C is waiting for B and the manager is waiting for C, because why is the manager waiting for the C? Is waiting for C because let me see what is what is the reason why manager is waiting for C basically because

C has initiated a read and we have to ensure that the revoke that is that has to be completed by client C, but there is some confusion because read might actually bring the file back here and has to be revoked again. So, for this reason the operation read has to be completed by C then only the state is becoming consistent across all the all the machines. So, because of this you can see that you need some kind of a causal models for example, if you notice that client C is asking for a read and a is also requesting on the same file then if you already shoot the forward read we can see the connection between the forward read and this write and you can essentially ensure that 3 is delayed and wait for this to be completed. You are doing this revokes which is again broadcast again you have to ensure that these 2 things are handle delivered to client B and client C in particular order. So and there is all these things happen casually that is why casual orderings is important here.

Imagine to all these things you also have to ensure that this Forder type of ordering otherwise you may get inconsistency operations right now we are talking about read and write that is you introduce things like directory operations, it could have similar thing like what we discussed before. So, the first say why distributed file systems are slightly difficult to engineer, there lot of a ways in which things can go wrong and process file systems especially clustered process file systems are very difficult to engineer.

(Refer Slide Time: 43:45)



Episode dfs redo-undo logging

- write-ahead logging: in each node, old and new value logging for every metadata upd
  - before bcache writes out any dirty data, it is also logged
  - at some time, metadata upd make it to disk.
    - txn aborts: undo upd to metadata
    - txn commits: redo upd
    - fsck still needed to handle hard I/O errors
- 2-phase locking?: locks acq without release until commit
  - Needed in redo-undo logging (no R of uncommitted data)
    - also, difficult with layered systems: lower layer locks have to be exported up
- cascading aborts?: B has changes of txn1+ txn2 but A none!

```
txn1:  s1 la1 lb1 ub1                    e1        A
                                           crash
txn2:                 s2  lb2  ub2  e2             B
```

So, let me take one more example of another attempt of a distributor file system, which also tries to do something interesting. This episode distributed file systems, it uses what is called redo undo logging I will explain what this is so. And for efficiency it does some other interesting things. So, will just try to understand what is going on this episode distributed file system, it uses write ahead logging, but because it is redo undo in each node old and new value logging for every metadate update, it has a old and new value. In a read only logging you do not keep the old values you only keep the new value in the log. So, because of this because it is got redo undo, it is important that if buffer cache writes out any dirty data it is also logged ok.

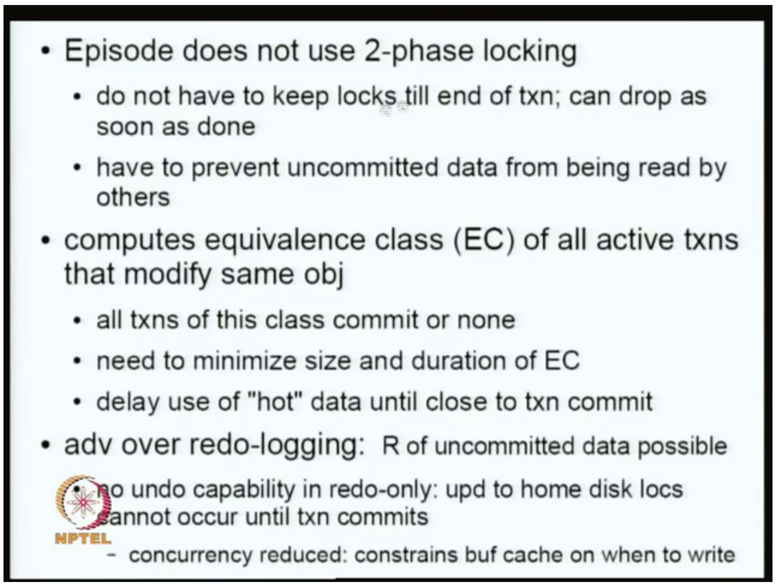
So, it is going to be more expensive, but it gives us some additional concurrency. So, at some time in the feature the metadata updates make it to disc and in case if there is a crash, we check whether the transaction about it if the transaction about it then whatever metadata update that make to disc. You have to unroll and you have you can unroll because you have the old information whereas, if the transaction committed, then you say look at the log and see which of those metadata update saturated made to disc. If some of those things you did not made it to disc you redo them. So, in this model even if there is it handles everything you do not need fsck except for hard IO errors, buts only replace (Refer Time: 45:42) because rest of it is been handled by this logic transaction logic. So, episode distributed file system each local file system does redo undo logging and this is in contrast to other file systems you do not redo only logging, again what is the difference in redo only logging you need to keep only the new value that shu that is going to the change value that has to go to the log.

In the case of redo undo logging you save both the old and new value, so that you cannot roll back. In redo only system you can only go forward on a crash, you cannot think or coming back. So, we have to much more careful I will come to that is or. So, typically turns out you do; what is called 2 phase locking in redo only redo only logging. Reason is that in redo only logging we have to ensure that there is no read of uncommitted data; that means, if I take a log you have to keep the log till the transaction commits. So, basically what you do is you keep on accumulating locks until commit then only release it. So, I made a mistake here, it should need only redo only login I this is not correct it should be read only login. So, that is another problem with 2 phase locking, which is basically it is also difficult in layered systems. So, 2 phase locking is not advisable in redo undo login but is required in redo only login.

Basically because you cannot have read of uncommitted data; the further issue that this kind of file system have to handle whether how to handle cascading aborts. Example suppose I have a an machine A your transaction one like this, and machine B have transaction 2 is start other transaction end of the transaction and you are updating a 1 b 1 a you doing updates or some kind and suppose there is some connection between this data and this data. So, trans basically in some sense this particular transaction has looked at these data is a modified versions and then it completes a transaction, but before the transaction will commits let us say this a crash. So, all this was sitting in temporary memory. In a sense what is happened is that this is sub transaction of 2 is a sub transaction 1. So, in case this happens you might have to go and undo this transaction. Luckily for you if you are using redo undo login it is possible because you have the old information make you may be have to undo some part of it ok.

So, that. So, basically you have to keep the possibility that there can be cascading aborts and only when the ancestor transaction is completed then only you can release the stuff release the data for the sub transactions. Otherwise you will get a situation where for example, B has a changes for both transaction and transaction, but A does not have a anytime, because this as essentially this is volatile at disappeared. So, you have to handle this also.

(Refer Slide Time: 49:32)



- Episode does not use 2-phase locking
  - do not have to keep locks till end of txn; can drop as soon as done
  - have to prevent uncommitted data from being read by others
- computes equivalence class (EC) of all active txns that modify same obj
  - all txns of this class commit or none
  - need to minimize size and duration of EC
  - delay use of "hot" data until close to txn commit
- adv over redo-logging: R of uncommitted data possible
  - no undo capability in redo-only: upd to home disk locs cannot occur until txn commits
    - concurrency reduced: constrains buf cache on when to write

Now it turns out episode does not have use to phase locking and because you do not have 2 two phase locking you do not have to keep locks still have a transaction. Can drop as long as you done with it you can drop it and there is and why you do it is because you have the undo operation. Even if a completing transactions coming it is read something and somehow you have not able to proceed for that you can unroll it. So, in the sense each of the transactions can be unrolled therefore, even if by because of some way the concurrent transactions have been ordered, if somebody read some data which is uncommitted you can always unroll it.

So, episode actually what it does is to handle both these problems it does something called takes all the active transactions it computes, an equivalence class of those transaction that modify the same object and all transaction class either commit together or not. And so, this some kind of optimization on top of what normal transaction models, and the basically in why they can they have to do these kind of things is because you want to let each of those local file systems, flush their things to disc instead of a coordinated 2 phase model.

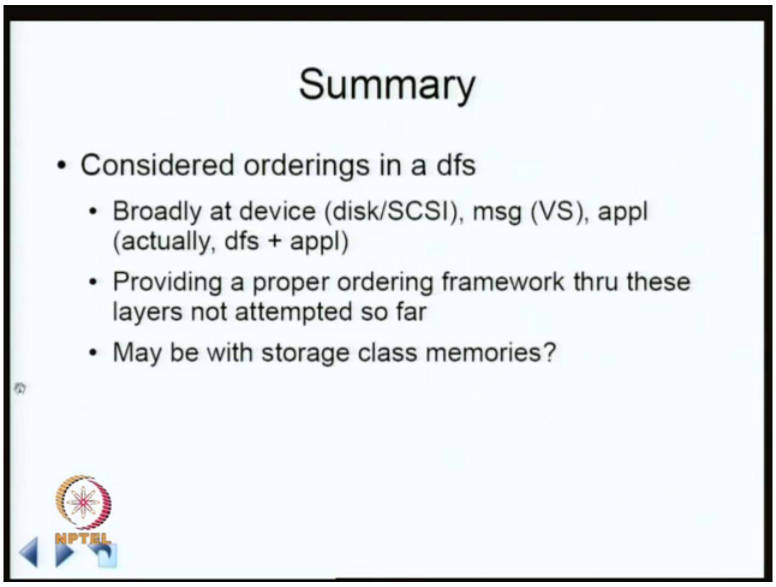
So, each sky each particular local file system can flush things to it is disc and only on failure give you have to figure out what to do with it. And if the failures are much less than normal operation then you are certainly going to their. Where as in the case of read only logging, you have to wait till whole commit has taken place. So, it can be problematic again in the sense what is happening is that just like a difference between a file system data base a file system typically has much shorter transactions whereas, databases can have very long running transactions.

Because that determined by the; is determined by the application not determined by the (Refer Time: 51:57). So, file system typically dos does small things, because basically it handles only storage (Refer Time: 52:06) operations therefore, typically it is operations are small. Whereas, database transactions they involve application level data and they can be arbitrarily long arbitrarily stressed out. So, for that reason or having concurrency you need to ensure that multiple transactions can be in place at the same time and you decide how they are going to be flushed, there is a conflict you roll back. Now distributor file systems also are coming closer to the database model, because they can they are doing across a network and therefore, they can take longer time than a typical local file system operation.

Therefore you also need to possibly do something better then redo only login. If you should do read only login then your parallelism is seriously constrained whereas, if you do redo undo login then all the updates when metadata can going parallel across all the nodes, without having to wait for the commit to complete and only on failure situations do you have to do something serious.

So, that is what basically we saying there is some advantage of redo; if you do redo undo login it is better than redo only login, basically because uncommitted data read also possible and because there is no undo capability in redo only update to home disk locations cannot occur till transaction commits; the concurrence are reduced it even constraints a buffer cache on when to write. So, this makes it first slightly slower less concurrency, but the problem with this is also that you have to say both the old data new data; that means, what you have to log also becomes higher. So, you are sending more stuff and you have study more complicated algorithm also ok.

(Refer Slide Time: 53:59)



Summary

- Considered orderings in a dfs
  - Broadly at device (disk/SCSI), msg (VS), appl (actually, dfs + appl)
  - Providing a proper ordering framework thru these layers not attempted so far
  - May be with storage class memories?

So, I think I would like to conclude today by summarizing what we located at so far. So, basically a lot of orderings in a distributed file system. If you look at NFS it was very (Refer Time: 54:16) in terms of what it decided because you basically a very simple model, it only did was in this side I cache something every.

So, often I what is a discard my cache and if it. So, happens that in that intervening periods somebody else has done some other operation somewhere else, I am not going to be consistent that is what this. So, where as if you look at the more elaborate distributed file systems, they attempted to do handle things better and I talked about a echo file system, which had a failing well thought out model and we also looked at echo file system with is are episode file system which also tries to give some guarantees, but it is done through a transaction models ok.

So, the interesting things about for us is that there are varieties of orderings are given in a system like distributed file system. It can happen the device level and a message level application level; when I mean application I also mean command level because you have file system is sitting on top of a messaging system. So, the distribution for system is actually sitting on top of this messaging system and top of the file system there could be another application sitting down of it. So, they have various orderings is going on at each of this levels.

So, we can do like what has been done other areas what is called crossed area optimization, in this in principle possible come with a ordering frame work across all this layers, but they have just proved to be very difficult and nobody has seriously attempted it because it is quite tricky. For reasons of availability and reliability and consistency, it is not something trivial. So, basically people have tried to avoid getting into cross layer optimizations here; by think someday in the future probably some of this if will happen especially with storage class memories, I think it might happen. So, with this I am going to conclude today's talk today's lecture.

Thank you.