

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


**Storage Filesystem Design**  
**Lecture - 22**  
**Filesystem Design\_Part 2: Designing Redundancy Continued, NFS\_Lookup,**  
**Abstractions**

Welcome again to the NPTEL course on Storage Systems. In the previous class, we were looking at some aspects of file system design and we will continue from where we left off last time.

(Refer Slide Time: 00:39)

### Redundancy

- Critical structures replicated
  - Superblocks
  - Large scale systems: inodes also...
- May depend on volume managers
  - RAID
- To avoid corruption (eg. from hw faults)
  - Extensive self-consistency checks in code (just like in an OS)
  - eg: is buffer alloc safe? Invariant: alloc during syscall & free at end
    - Disk drive hw problem: cannot interrupt CPU: buf lost!
  - Best Effort Service
    - Soft errors (retry), hard errors
    - If error detected in spite of checks, try to move fs to some reasonable consistent state ☹
    - Usually no non-starvation guarantees
- End-to-end checksums (across disks, HBAs, netw links) needed



So, I think we were discussing something at this point and we were talking about how it is critical that you need to replicate critical structures or find some mechanism by which you can recover in case there are hardware problems. And if you look at some of the more recent file systems for example, ZFS from SUN is a good example of this. They have discovered by talking to various customers that avoiding corruption of the file system metadata or data is quite critical.

And there is some already data saying for example, that in large-scale systems almost every 15 minutes or so there is a silent corruption, every 15 minutes. If you are you are talking about very large group hosting kind of places or cloud kind of computer places

every 15 minutes, there is an error undetectable error. Of course, you can detect it through a checksums what I mean by saying undetectable is that there is an disks which have Reed-Solomon error codes that will lot of extension error control. But in spite of all that Reed-Solomon codes, there are still errors that are undetectable unless you handle it through some mechanics and check summing at file system level.

So, ZFS for example, tries to do some of these things, all these necessary aspects. So, traditional thing is best effort service as we were discussing last time. If there are any soft errors that means, if you retry to one more time the error goes away, you retry it a few times, and if it does not go away then it declare loss. And this is typical SCSI and hard errors are things like for example, which cannot be recovered this very you get an indication that discuss dead sense of that kind.

So, as we discussed last time in case there is error in spite of all the checks we have putting, you somehow have to still salvage the situation, you cannot just walk away, you have to find some way of getting the file systems from this very state. It has to be a consistent state, you might lose some data and you have to do something that has to be over still some kind of a semi reasonable state, so that forward progress is possible.

And nowadays there are some newer standards that are coming there is something called data integrating format DIF, wherein they try to do end-to-end checksums. Now, you notice that in a piece of data when it is transiting from the CPU to memory to network to disk and back and forth. And there are various buffers all over the place. There is a buffer in the sitting in the disk itself; there is a buffer in the HBA. There is some buffer in memory of course. So, any one of these things can get corrupted in principle. And usually we have ECC memory error correcting code memory, but even that can only handle two faults, it cannot handle more than two faults. You can detect if there are two faults; if there is one error you can correct it, but more than that is going to be serious problem.

So, you have to find a way in which in spite of all these possible error locations, you still can discover whether you are doing something right or wrong. So, this disk basically is one attempt at doing what is called end-to-end checksums. And this unfortunately requires some major changes, basically because if you are talking about end-to-end checksums you need to attach information about the checksums somewhere. So, every

block for example let us say- it is 512 bytes you need to attach some information about this checksum. So, it cannot be a 512 byte sectored disk it has to be a 520 or whatever 524 or whatever; that means, that you are on disk format has to change that means, your support for sectoring as required by the needs of this checksumming has to be possible, if it is not possible you cannot do it.

And in addition there are even more interesting complications here. For example, when you are writing to disk, it may be that you are usually there is a servo motor which takes you to the particular track, it is possible that once in a while instead of going through track number 100, it might go to track number 101. So, when you write it, it actually return to 101, not 100. So, if you just do the checksumming on the data that will work out quite ok, because finally, that is a block your data you wrote it and there is a checksum corresponding to that, and then it depends only on the data. So, checksum is still ok.

So, even if you write on 101 track if you just only checksum the data you cannot figure out that something bad has happened. What you need to do you need to not only checksum the data, but you also have to checksum along with it some information about which track you are trying to write, you also have to incorporate into checksum the fact that you are trying to write tracks track number hundred or sector number so and so.

If you do that then when you try to read next time, you ask for 100 and then or sorry or you went to look at the track number 101 sector something, you pick it up, and then you again do the same checksum using information that data plus the sector number slash track number that you are thinking about then do not match. Then you know something bad has happened some somebody has wrote into the track number 101, and it actually (Refer Time: 07:06) some previous information. So, you need some additional piece of information also and these are of course, very specific to the devices.

When you talk about disk, we can talk about track numbers. When you are talking about other kinds of devices there may be some other things so that means, that disk end-to-end checksum is going to be highly device specific and technology specific. So, and of course, in the case of networks people have known all these kind of problems. In networks typically it turns out that for example, there are some studies which have shown that they are using ATM. For example, there is something called asynchronous transfer mode ATM which is to be quite popular some time back. It is still used in the

backbone of the internet also few places. It turned out that there are some bugs where each of these 8 bit quantities can get flipped. There are some unusual situations where if you have 16 byte quantity the two 8 bit things get lift. And such being shown to resultant network packet errors as high as 1 percent or 2 percent of course, this I am talking some old information I am talking about 2001, 2002.

So, this end-to-end checksum is highly I just wanted to mention that is highly technology specific and you have to because you are essentially are trying to work around the kind of problems that the hardware is throwing up. So, by definition has to be technology specific. Now, one of the things I mentioned that these systems they usually give only better effort service there are no non starvation guarantees that is then why they cannot do it because hardware itself is not giving any guarantees hardware can be arbitrary things. So, if you are trying to work around it often you do not have a way of working around it.

For example, if they are doing soft errors retry, there is usually some timeouts involved. And the timeouts are set by your expectation of typical situations. If it happens very rarely, your timeout is going to be large; it happens quite frequently when you want to keep it slightly smaller. So, reason why you want to I think it is quite obvious if you want to it is very unlikely right you want to wait for some time thinking that it will correct itself somehow. If it is reasonably often then it is since its often then probably something actually has gone bad you might have to something about it. So, the timeout basically is highly again dependent on the context in storage system.


So, because of this if you are planning to build a real time system or any those things or you want to give some bandwidth guarantees and what not it turns out to be slightly difficult because of all these assumptions all over the place. You really cannot really provide very strong guarantees. And it also turns out that these non-starvation guarantees are not possible, because of the way some events in the system interact I think we will just take one example of that.

(Refer Slide Time: 10:29)

## NFS: dir lockup

Consider a “slow op” on a (locked) file due to NFS congestion

- VOP\_LOOKUP on that file results in a lock on its dir that cannot be released until “slow op” finishes =>
  - cascade of locks upto root that hangs the system till “slow op” finishes
- lockd: similar problem but worse as lockd a user process



This is an example of a look up we are doing a directory somebody is doing the look up on a file, and they can be some see these delays involved, let us see what that is. So, what is happening there is some suppose you have a file on which some slow operations taking place, very slow operation that means that it is somebody has done, what is called file locking. This file locking is available at user level. And the user is updating things very, very slowly let us say for example, it could be that you are picking up data from the network and the network sending the bits very slowly that means as the user has locked up portion of the file and this need will not be dropped till the operation completes.

So, in a sense you can have some you can imagine some slow operation. Now, by bad luck if you are trying to somebody else is also looking at that file let say they are coming through NFS. Now, when you are trying look up some information, you want to make sure that the information looking for is stable. The way you make it stable is by taking a lock on the directory, the parent directory. Because that basically says that nothing can change corresponding that file because the metadata of that particular file is going to be kept in the parent directory for example, its size and all those things.

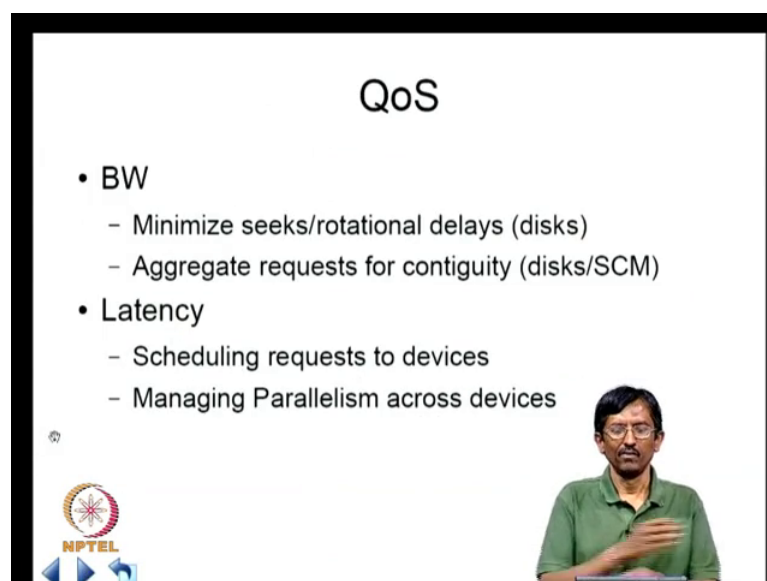
So, you basically if you will take a lock on the file basically what you do is you take a lock on its directory, then you basically guaranteeing that whatever changes that are possible on the file is not possible until you get the chance to get the lock. Now, the basic problem is that the directory is now locked, nobody else can; if somebody else

innocently does the look up on that directory, for example, somebody could be just searching for a particular file in directory, if they do not want to see if that particular file exists in the directory. So, to again make it stable, we are going to take lock on that directory; again to do that you have to take a lock on its parent.

Now, this can keep on going all the way up that means, that your operation can be arbitrarily slow depending on when the user has released that file locking. There are no guarantees now, so that is why it is extremely difficult to because the way these things inter link the events in a system they are extremely difficult to predict. And if it happens in a way which is unexpected they can be arbitrary delays, so that is why usually these kinds of systems never give you non starvation guarantees. If you really want to worry about these things, if you really are interesting a real time system etcetera, then you someone make sure that these things are in memory if possible even in memory the same problem can be there, there is no escape here also.


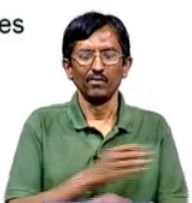
So, because if you build a let us say a memory based file system, the identical situation also can happen, there is nothing which says that it cannot happen, except that the time involves the amount of time involved for you is smaller, same will happen with a disk speeds. So, you can actually be hit by it seriously. So, this is I just want to mention that usually most file systems never make you do these no starvation guarantees, because it is difficult to provide.

(Refer Slide Time: 14:39)



**QoS**

- **BW**
  - Minimize seeks/rotational delays (disks)
  - Aggregate requests for contiguity (disks/SCM)
- **Latency**
  - Scheduling requests to devices
  - Managing Parallelism across devices

And in the case of again coming into the same aspect lateral time there is disks etcetera there is a issue of quality of service and usually these things are usually also not seriously provided. Basically because you need to really take control of bandwidth and latency issues which are somewhat difficult to manage guarantee. For example, one way to minimise they provide high bandwidth in the case of mechanical components electro mechanical components of disks is to minimise the number seeks and rotational delays or you aggregate requests the contiguity we are doing some of the things, extensive amount of aggregation, clustering all these kind of things have to be done. This is a only ways to ensure that your bandwidth does not decrease, because the minute we start having seeks and rotational delays we are talking about some multiples milliseconds that itself will kill your bandwidth.

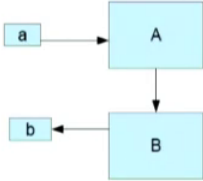
And in the case of latency, we have to do some scheduling, we have already look at some aspects of this, and we also have to manage parallelism across devices. So, now the issue for us is when you trying to manage parallelism across devices, we also have to somehow understand that there are issues of failures also. Then you have to inter link ability to flush certain data to devices, do not do it parallel, but in the context of failures.

So, if failures occur again you have in a slightly tricky situation that is a reason why some of these things are usually not really done in a file system level, it is expected to that of at higher levels, but the user what to do when some bad thing happens. The file system not being given information about what to do when multiple parallel activities are sent out and then sudden failures occur, it has no idea about what is the important, what is unimportant? So, this is best handled at user level in case user knows about how to handle it; again these usually not something that is know in the file system already.


(Refer Slide Time: 17:00)

### Types of Designs

- Journaling FS
  - Transactional
- Soft updates
- Update in place?
  - Log structured FS
- Disk optimized?
- B-Tree based?
- Support triggers?



```
graph TD; a[a] --> A[A]; A --> B[B]; B --> b[b];
```

 NPTEL

We will just look at some high level designs, what are the types of device design parameters. One thing about a file system is that it attains to many update of its structures usually it touches multiple pieces of data. For example, there is metadata and data. So, when you update a file, you not only update the contents, you also update its access time the file size etcetera that means, that you have to update both metadata as well as data. So, since it is not possible to update multiple disk blocks atomically, you know hardware does not provide you that information provide does not give you that capability; that means, that you have to figure out the way of forcing these changes in a particular order in spite of whatever the devices are doing.

So, one of the first earliest model is what is called the ordered writes. You figure it out that this has to be something has to happened in a particular order. and you just synchronise writes one after another. Synchronise writes the thing you do is you send the write to the disk and wait for it to be completed before you proceed; now that is very slow extremely slow. So, you need to find some other ways of doing it, and the one there are many, many methods of doing it, and this basically these methods journaling etcetera.

So, if you are not bothered about correctness that much or you willing to take the risk ordered write is still a good model, so that is reason why the previous file systems in a past up till our 1990s for example, most of them is based on ordered writes. They were nowhere really they just said that in case something bad happens, power failure or



whatever we have to figure out what to do with it, whatever happens. And typically it is to be the case that in those days personal computers are not still not that very common. So, usually any machine UNIX like machine was managed by a system administrator. So, if something some crash took place somebody was like reasonably competent person was there to take care of it.

Nowadays with large-scale computer systems available to end users, it is not the good idea to assume that user knows what to do in case there is a crash. So, now, this kind of models have become more appropriate and reasonable, it is always good to make it as surprise free as possible these file systems; if something bad has happened, it let the file system to figure it out how to fix it. So, the kinds of things that people attempt today are things like generic files systems and other models will come will go through each one of these in a bit.

So, in generic file systems, what happens is that you do logging and then you first make sure that all the changes are in a intent log and from there it goes to the what is called home locations. Whatever changes you make first go to the log; and then once the transaction is completed you trigger back from the logged values, it goes back to its home location. For example, if you are writing a file A, the data corresponding could be of the write of the file could be returned to the log it would not go directly go to the file.

And then once the transaction is completed when A is successfully complete there is no power failure or any other these things then you can think of writing that data from the log to the file, and you read the log only in case there is a failure. Sorry I made a mistake. If you are doing logging, once you are going to trigger back the information from the log to the file and that is the normal thing. And then if there is a failure then you go back and read from beginning and see what is happening that is what I meant to say. So, the problem with generic file system is that you know how to write two disk places, one is the log and then from there to the home location. So, it is going to be slow.

Now, good thing about this is that you write into the log whatever changes are going to take place; so that you would not roll back it is easier to roll back. Now, typically in file systems you use what is called read do only logs because the transaction is typically small, they are small duration. Whereas, in a database kind of situations the transactions can be long, so they usually have redo undo locks. So, basically in some sense in file

systems we assume that we cannot always move it forward, you can always take a trial take some changes in a forward direction. And whereas, in database sometimes there could be incompatible transactions you have to roll back ok. So, this is one type and it is quite commonly used.

Now in this generic file system kind of models, we might log only metadata or you can log data there are various possibilities here also. So, there are some file systems which only log metadata because logging data is going to be expensive. So, or a file systems allow you all the possibilities, but you have mount options in the mount options it tell exactly how you want to log it whether you want to log the data as well as metadata or only metadata. If you log only metadata then there could be a problem there also because you might have dependencies between data and metadata of that had to be captured. And if you only log the metadata then some of the dependencies with respect to data might they could be sitting only in memory is a power crash, when there is loss of power then you are only have information about the metadata, but no information about the data. So, that also can get you in some problem.

But these are the kind of things that people have done for in the context of widely used file system. For example, EXT 3 has some of these models it can do metadata only logging it can do metadata plus logging it is up to you, you decide for your application what kind of guarantees you want and you decide it.

This was another model called soft updates which is there in the BSD file systems, I mentioned previously that when you do lot of clustering you can get into some trouble. And soft updates is one way in which what you do is any things that have to be flushed to disk first you have all those modifications in memory and then you basically try to flush the changes from memory to disk in a particular order similar to ordered writes, but keeping in mind that you are doing clustering.

Let us see how clustering I preferably mentioned last time that clustering can introduce certain dependencies should not be there. This is an example of the kind of problems that occur with if you are trying to keep track of dependencies in memory, and you want to flush them in a particular order to disk. Suppose, you have the following order in which you have to flush things that is let us say a is an inode and small a is inode and capital A is the data let us say and similarly capital B is the data and small b is the inode. I am just

taking a random graph here please do not write read meaning into this thing, but this is possible all these things are possible.

So, what it means is that this inode A should be flushed first followed by the data corresponding to a, and then let us say for some reason A has to be flushed first followed by B and then for some reason data of B should be flushed first followed by its inode. Let us say there is something serious there. As I told users told the fictitious, but this kind of real dependencies can be easily possible. Now, so in a soft update you basically all this is sitting in memory and you are trying to flush them to disk in a particular order; that means, you are buffers are there, there some headers in the buffers you keep track of these dependencies. Any time the flush demon comes in that guy looks at each of these other things it goes through all the dependencies and does it in a particular order. It looks fairly straightforward.

Well, let us see if I am trying to do in addition to this flushing suppose I am trying to also do for efficiency of the disk I need to do what is called clustering. What is clustering and notice that a and b are only let say 128 bytes and this is 128 bytes, whereas this is 4 kilo bytes and let say this is 4 kilo bytes. And I tell myself why do I should I do this flush of because what is the problem in just flushing a small a basically it becomes a read modify write cycle. Because I am only updating 128 bytes so that means I have to read and let say the I am only accessing the block device at 4 kilo byte chunks let us say. Then in that case I would not update 128 bytes, I have to read in the 4 kilo bytes corresponding to the block where a is residing get the full 128 bytes then update in memory exactly at 4 k what I am reading update that small portion corresponding to that small a then write by the whole thing. Set become what is called the read modify write cycle, so that itself is costly.

Now, if you want to avoid those kind of situations, the right thing is to do is clustering. See if I can put as many inodes as possible which can fit into four kilo bytes and flush all that inodes in one short I will not mix with the data because data usually is in four kilo bytes of course, I can mix inodes and data also a in case you want to do it. But let us for simplicity assume only inodes are clustered together and data are clustered together let see simplicity. Now, our problem is that if I let say in some clustering scenario if i get a and b to be clustered together. Now, what will happen it is basically saying that this particular graph now is circular it is basically saying this has to be flushed before A, this

A has to be flushed before B, B has to be flushed before this combined data this is circular this gives a graph is now a cycle.

Now, that means, that once you start doing clustering, they have been you have started introducing additional dependencies spurious dependencies, they should be under exists before. So, this soft update basically takes care of how to handle these things and the way you do it is again something similar to what people do in the case of transactional models, where you have to enrol some things. You cluster some things and then as you are flushing them you discover at runtime you figure out that having flushed so much if I were to flush this particular thing I am at actually create a cycle. I have to enrol some things, so that I still have (Refer Time: 28:24) graphs and then I do in that way. So, basically you cluster them tentatively, but enrol some of them in case if find that that creates some kind of cycles.

So, this soft updates also it is a good model and the only thing is that there are some complexity involved in figuring out these cycles. So, basically you have to keep on running strongly connected components always kind of always all the time. So, that can be expensive, but probably it is less expensive than kinds of delays that are there with respect to disks if you do it synchronously. So, again there is a some kind of trigger off whether if I do synchronous rights, is it more costly than doing all these computation at run time as I am flushing it. The BSD file systems use soft updates.

There are also other issues about whether you update in place or not there are multiple designs here. I can have what is called a log structured file system for example, where I write everything to a log, and the log is actually doing the file system, there is nothing other than the log. So, if you want to see the most recent version of some particular block I have to scan it backwards, if I want to read for example. I am just keeping on making modifications to blocks. And in some sense there is some kind of versioning going on if i want to get the most recent version, I have to scan form the most recent end of the log and then go backwards and find the first copy of it. So, writes are simple because I am always writing at the end reads are complex because I have to scan it backwards.

And the issue what this kind of file systems is that once you are over writing a block, the previous blocks become garbage because they have to be garbage connected. Question is how do I eventually garbage connects, because that becomes an issue here. So, we have

to again once you garbage collect then what will happens is that there will be in that log for example, there will be holes now. So, do you compact it, if you want to compact it you have to copy things. So, all these issues becomes slightly big issues. So, the standard thing is to do all the writes onto buffers. So, you have so many segments what are called segments and then keep it around for some time and then you keep cleaning them with respect to dead blocks for example, and then once you find that they are reasonably complete then you flush it to disks, so that is something typically is done.

So, log structured file system have been quite are is a very nice idea and it has found its way in many, many different ways I think for example, when we discussed the Google file system they also have some kind of log structured file system here. If you are talking things like flash based devices, they also find that having a some kind of log structured file system which is useful to get all the writes in one place, and then later do updates as necessary. So, this log structured file system is a good idea and has been used in different ways in different places.

So, basically in some sense here what you are doing is you are not updating in place you always writing it to some new location. And this also has been used in some other context, we do not update in place. What you can do is any time you are writing, what you do is you do not stick to the place where the data is you write to some nearby place wherever the disk head disk let us say. And then keep the pointer to point to the fact that logically it all should be pointing this new place where I rotate rather than to the place originally generally existed.

So, what you can essentially do is you have a disk on which is moving, if you are able to find a way in which wherever you are there are few blocks always available let us say, let say it is always available. Suppose, you make sure that the system is structured in such a way that wherever you are there are some free blocks available that means, when the write comes in you look for the closest free block which is available not to the original home location, not where you actually the data resides. Because you are not going to update in place, you will just basically look at the nearby place where there is a free block available you write there and switch the pointer originally it was pointing to the home location wherever it had the actual data, which you would have gone to if you want to do a read.

Now, instead of going to that place you just took the pointer saying that the newer contents are pointing to the place where I am just right now. Of course, this creates some problems like fragmentation that is if I want to read something some part of it is here some part of it is there some part of it is here. It is not exactly in sequence; that means, I can read through the performance nearly goes bad, but you write performance is good because now you are really seeking to you know trying to write. So, various things have been attempted like this. So, the good thing about update in place is that it can be married to a transactional model also because in transactional model what is the issue the issue is that you want to either be the old or the new.

Now, the thing is if I keep on modifications write to so that it does not go to the home locations because nothing is uploaded in place. Then basically what it means is that you can place the pointers, so that you have an old copy in the new copy and then you can just switch you can go back go from the old to new in one short. Basically I have a clay of pointers and then if I want to go to the new thing I just switch this clay of pointers to the new one that can be done atomically because it is a memory operation. Or it can be if you want to make it to disk you can flush all those things to disk one by one the updates, but the very last one which actually does not switch that is seeing the blocking off you send it that can be atomically at a sector level for example. And typically most disks give you some guarantees about writing a single sector that is some capacity in a system whatever it can write that that particular sector atomically.

So, this non updating place can be easily married to a transactional kind of system quite well and you will see that for example, many commercial systems do this. For example, there is something called a NetApp that is something called WAFL, WAFL write anywhere file system this particular system gives us these two ideas you know its transactional write anywhere. And another thing another example is a ZFS which also does amazingly.

So, one other thing I just want to mention is that there are lot of dimensions to the design of these file systems, but I just wanted to mention few other dimensions. So, far we looked at all these topics connected with how to atomic updates. But there are other aspects the designs also whether you are disk optimised for example, do you think that disk or what is holding a file system or something else suppose an disk optimise for example, suppose I assume that I can only do with disks.

Then my problem is a following I have data and disk metadata right usually metadata and data can be in different places. That means that when I am writing file I have to go to the metadata as well as data. That means, by definition I am forced to go, I have to make a seek. I am forced to make a seek I am forced to make a seek first suppose I want to make the file write committed because there is something critical for me. That means, I have to do to only go on finish the flushed out that metadata, but also do a seek and get to the data also.

Now, if this is slightly intolerable because you are essentially talking about a seek to metadata flushed to disk again seek to data flush that means, you are talking about minimum two or so of seeks or rotational delays plus two disk writes. And this can come close to about anywhere in the current disks it can be about 50 milliseconds. So, at the most you can do 20 writes per second. Suppose, this kind of speed is extremely low or you cannot handle this, you want to do something else. Again if it is only disk for example, we have with you with you what you can do is there a way in which I can separate metadata and data. And this often times called multi device report that is what you do is I have two set of disks one of them is holds only metadata another holds only data that means that I can now keep on writing.

For example, if my data if it let us say a large quantity let us say some mega byte then I can keep on continuing writing onto the disk without being disturbed by other concurrent activities in system which are also doing metadata writes. The metadata writes are going on other side. So, I can essentially have some ways in which I can partition the system, so that the metadata is in one place and they are getting accessed somehow sequentially somehow, and data is also getting accessed sequentially.

And therefore, I can get slightly to do better than the previous situation where data and metadata were together in one place. And this kind of things are also can be attempted in the case of those systems file systems which are based on things like flash and other kinds of devices. Because even in flash for example, it turns out that some people say that flash lights are faster let us say small flash lights because you do not have the component of seek and rotation there.

So, even if disk bandwidth is about the same as flash bandwidth right now, the fact that you do not have few milliseconds or seek or rotational delay can make your write faster.

So, you might want to if your metadata intensifies for example, you do not want to put the metadata in a flashed based device. So, various kinds of possibility permutations are possible. It is up to you to decide depending on their context and their requirements you can partition, your data structures in a file systems differently depending on characteristics of the device. So, all those design optimization are possible, that is also a question of whether you want to be based on your file system design, is it based on set of pointers to various disk blocks like a typical file system.

For example, the inode essentially keeps track of various blocks and is a pointers to various blocks right. Do you want to do it like that or do you want to have an explicit b tree kind of an model, so that you can somehow bound the amount of the height of the tree. If you are able to do that it will be helpful this is especially true for multimedia file systems. In multimedia file systems typically what happens is that you are talking about match files and regular traditional file systems, what is called dead blocks which access about the first 40 kilobytes or so directly from the inode, because there are pointers to blocks directly.

But once you go after 40 kilobytes or so you have to go through indirect blocks and also what we call triple indirect blocks also that means, you have to go through another levels of disk based indirect. Which if you are doing a multimedia kind of file system where ten minutes is important, because we are talking of video playback and other things this indirect also can create some glitter in the systems it can essentially sudden the sudden frames cannot be displayed fast enough.

So, in these kind of systems, your design becomes simpler if your file for example, irrespective to the position it has got the same kind of leadership behaviour. So, in the case of a regular file systems, your first ten blocks got easy service you were able to do the service very quickly, but latter ones always suffer through indirect there is a bit everybody gets semi treatment. So, again it turns out in your systems are typically also (Refer Time: 41:56) most of the new ones big ones they are usually (Refer Time: 42:00).

The other aspects of design also has there are many dimensions I just wanted to list a few. You can also support what are called triggers often times what happens is that you want to know when a particular file has changed you want to be notified. And for example, you might have the compressed file systems and then you try to read it, so there



is compressed top is on the disk and the minute you try to read I want to provide that information to a process. And the process wants to look at look at only the uncompressed part, it does not see forget it is not going to so let say the process not interested in doing account, it does not want to decompression it wants to have a transparent way of looking at data.

So, the minute you touch it, you want to interpose a process which will actually decompress it on the fly and provide that information to the process. So, this requires some kind of triggers that is the minute it is accessed, you need to be given a call back, the kernel notices that something special has to be done. It will in turn will give up call to a process is the decompression routine or decryption routine for example, which will in turn make sure that the data that is going to be given by the file system actually it is read from disk, but it is again message appropriately. Finally, it is decompressed and then made it look like a regular block of memory. And then this is what the file system actually returns back to the process the neither the file system nor the process knows that somebody has come in done some decompression or decryption and done it.

So, say a delicate design, but there are ways in which this can be done the there is something called data management API by which this kind of stuff is done. What is critical about this is that the both the file system and the process do not know that decompression or deencryption has taken place and one of them they know it ok. So, well actually lot of things happen in between otherwise there it become non transparent. So, you will find that for example, some of them are advanced file systems provide it and Linux for example, XFS provides this capability not temporary file system provides it because slightly difficult to provide. Because they interacts with multiple things it interacts with the virtual memory file system, it interacts with of course, the file system it interacts with a kernel abilities to do up calls to this random procedures or anything there is lot of things that have to be provided, so that you can support it.

But this ability to support regards is extremely useful and it is used extensively in many I will just give you two examples by the time you see lots of situations. It can be used for example, somebody reads something want this some error that has taken place I want to call a error routine to fix the thing let say all kind of things are possible. Now, the typically it turns out these file systems are clearly complex based because as I told it


interacts with lots of things. So, the way that reduce the amount of complexity is by using abstractions systematically, so that becomes slightly more easy to manage.

(Refer Slide Time: 45:32)

### Abstractions Used

Buffer related

- `getblk`: given a filesystem number and disk block number, get its locked buffer
- `brelease`: given a locked buffer, wakeup waiting procs and unlock it
- `bread`: read a given disk block into a buffer
- `breada`: `bread` + `asynch.` read ahead
- `bwrite`: write a given buffer to a disk block
- `alloc`: allocate a free disk block and return buffer using `getblk`
- `free`: free a disk block



So, the abstractions are typically like for example, buffer related, inode related various other things, they are kernel related etcetera various kinds of abstractions are used. So, that these abstractions make the writing of the file systems slightly simpler. Of course, these are all kernel-based abstractions that mean that these are all not let us say these are not available at user level, but these are used inside the kernel there. For example, in the case of buffer related, there are various types of calls for example, get block, so it given a file system and a disk block you want to get a unique buffer, but it is also locked.

So, if anybody else ask for the same file system number block number is still should get what was given a rear out it should not be too compiles of the same thing at because otherwise then each of the parties can write to it and there will be inconsistent. So, there is a written call get block. And of course, this depends on various portions of file systems. If it is highly concurrent is you know multi threaded kernels that will all consider differences here, but in general there is some function call dead block. By which what you are guaranteeing is that if there is block in the disk on the disk there will be only one copy of it in a file system in the memory only one copy actually. Across all users, across all threats there is only one copy of it.

So, if any modifications take place it goes to only one place that is a basic guarantee that they want to give, so that is what get block does. It can also have b release, we can for example, once the write has been done, you want to release a buffer. And of course, if it is dirty, it has to flush to disk, but the idea is that you release it if somebody wants to reuse it then there you will be notice that it is dirty, their business to flush before they can be used later. And if it is left dirty for some time, if somebody wants to look at it they can still look at it, because it is cached on in some sense that buffering basically is giving you the caching capability.

Now, at the same time, if you get a locked buffer, it might be possible that whole you are writing it other people were looking waiting for that, there is you got something from somewhere updated it, but while there was going on other people also wanted the same thing. So, they all will be waiting for it. So, as a part of its job b release actually wakes up any process that is waiting for it also and then locks it. So, this is a typical kind of usage that is there I would say that if you are looking at any file system that is derived somehow from the old UNIX bell large file systems, we are talking about file systems that were there from 1980s they were using these things.

And you will find similar things even current generation file systems are UNIX based if you go to windows based file systems you may not find similar exact abstractions, but it is there for example, Linux actually use some of all these things there are similar functions that differ in details that mostly it is we have similar functions. Again if you want to read of disk block the ways breada b read sorry it can also be breada basically you do a b read of one block and then you do the single b read happen in the next block next logical block.

Of course this actually you should notice may be wrong because what it is going to read next is the logical block, and it is usually should be actually again mapped again to see where exactly the physically its located, but that is too costly to do it. What they do is they just feed the next physical block. With a assumption that typically things are there is some corresponding between logical and physical let us say layout. And therefore, the async read ahead actually goes on and reads the next physical block not the next logical block in principal it should be next logical block that is not original.

So, again b writes that a given buffer to a disk block, alloc or basically allocates the free disk block and return buffer using get block. Basically you have to allocate a free disk block here then only you can do a get block on it. So, this is higher level routine which uses get block. And of course, we look at allocate a free disk block; what you are doing is you are updating your information about which blocks are free right. And the question for us is do you keep this information also persistent or do you keep it in memory.

Now, if it is a generic file system, what you do is when you allocate a free disk block you have to flush the intent that you are actually going to make this non rather than you are going to essentially look at a disk block that particular thing has to go to disk. So, what you do is again if you going to do it that way then for every allocation you have to flush to disk which is again costly. So, basically you keep on writing to memory, we keep on having a log in memory which is keep on added to it, you do not flush it, but flush it at some point in future.

Now, in case some bad things happens you lose all these things, but everything is still in the everything is consistent. You lost lot of things, but nothing has ever made to disk. And if you assume that whatever you are writing to disk is always consistent suppose you always guarantee that then whatever is in memory is lost, but it has not made this system inconsistent. So, the idea would be to keep on buffering each of these kinds of allocation decisions keep on buffering them at the end once it has reached the certain size of the updates, then that particular log is actually flushed to disk. And now in case there is a failure, we can go back to the log and see what has to be done to fix it.

So, there is some multiple levels at which these delay in terms of making the impact of particular decision happen immediately they try to cluster it, so it is getting clustered in memory first if before you flush that particular log block end to disk, you lose it you lost essentially lost whatever recent changes have taken place. Then once it has achieved a certain amount then the whole log actually will go to disk. And then in case there is a problem then I can you can use the log itself because it is persistent to be fix it.

And even here there are some interesting tricky things it turns out when you are trying to flush the disk block, your block could be in multiple sectors, it is possible that its multiple sectors. Now, it may be that because of the way the sectors are laid out on the disk it may be that the disk is closer to the end of the log than to the beginning of the log.

Now, if you are elevator algorithm let us say we are using that one it goes ahead and flushes the end of the log first because it is closer to the head than the beginning of the log.

Then what will happen if there is a crash you will see some minute thing in the disk and you will not know what to make of this thing because basically the head of the log gives you all the information about how to read it. It tells you there are so many pieces I am putting together there is something like when you have a document you say page 1 of 12, page 2 of 12. So, you know that if page 1 is missing you know that it says page 2 of 12 you can figure out that some page 1 is missing. Or if you say page 11 of 12 and if 12 is not there you can figure out that page 12 is missing.

So, essentially what happens is that the beginning of the log essentially tells you how to read rest of the log, and because it is circular log this information is important typically these logs are circular. So, the only way you can read the log and a crash is by being able to figure out where head of the log is. Now, if it so turns out that the because of the disk elevator algorithm it flush the one side of the log first the end rather than the front side then you can get into the problem.


So, basically you have to when you are flushing it, you have to ensure that you have anchor the write of the log itself. What is anchoring you first send the first sector of the log read that I have to do the very first sector. So, one frightful byte actually has to go first they are careful about if you are really paranoid about data security data being not lost. You even though you are writing something like 256 kilobytes of log you first have to write the first sector to force that thing to the right place, then rest of it can go any order, because the first sector has all the metadata about how to read the log in case of failure.

So, all these things are done, so that is why even simple things look at in the free disk block etcetera can have lot reunifications in terms of then actually it makes it to disk. This all can serve things are coming in a picture between for it happens. Similar way you can also have a free disk block.

(Refer Slide Time: 56:22)

**Inode related**

- `iget`: get a locked inode (doing bread if necessary) given inode number
- `iput`: release an inode; if ref count 0, writes dirty inode
- `bmap`: given inode and byte offset, returns disk block num and offset
- `namei`: given a path, get the locked inode
- `ialloc`: assign a new disk inode for a newly created file
- `ifree`: free an inode (link count 0)



There are things relating to inodes also. So, again lot abstractions are used as I mentioned. You can get a locked inode which is using buffer read because again inodes are kept in the disk and we are going to read it through buffer read. And then once you get the basically somebody is doing the mapping between inode number and the block and then you go to the get the block and then read that portion of the inode. Similarly, if there is an `iput` that means this is `iget` we get it from disk `iput`, you want to push it to disk and then you keep some reference counts. And if you turns out that reference count is 0; that means, there are no users in memory then you can write the dirty inode. And this can be delayed also or it can be let us say done immediately depending on the seriousness of issue.

Now, other important one is `bmap` which basically given inode and byte offset, returns the disk block number and the offset corresponding to it. And there are few things like `iallocate`, `inode allocate`, and `ifree`. Again, all these things just like `block allocate` they also have to go through some logging in case it is a logging based file system. So, `namei` is basically a slightly longer one given a path get the locked inode it is we have to parts each of the components with the name and then you have to keep traversing the file system hierarchies, so it is a very so fairly long operation.

So, I think at this point I will stop, I will continue form here next class.