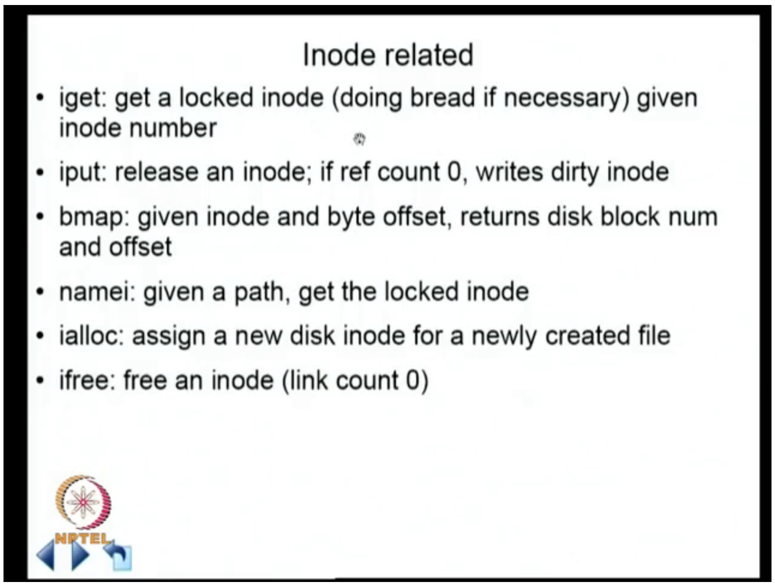
Storage Systems
Dr. K. Gopinath
Department of Computer Science and Engineering
Indian Institute of Science, Bangalore

Storage Filesystem Design
Lecture – 23
Filesystem Design_Part 3: Design of Link & Write Operations, General issues

Welcome again to the NPTL course on storage systems. In the last class, we were saying that to contain the complexity of writing the file system. Typically, some abstractions are used. Some of them are related to manipulating buffers. Some of them are related to managing inodes.

(Refer Slide Time: 00:42)

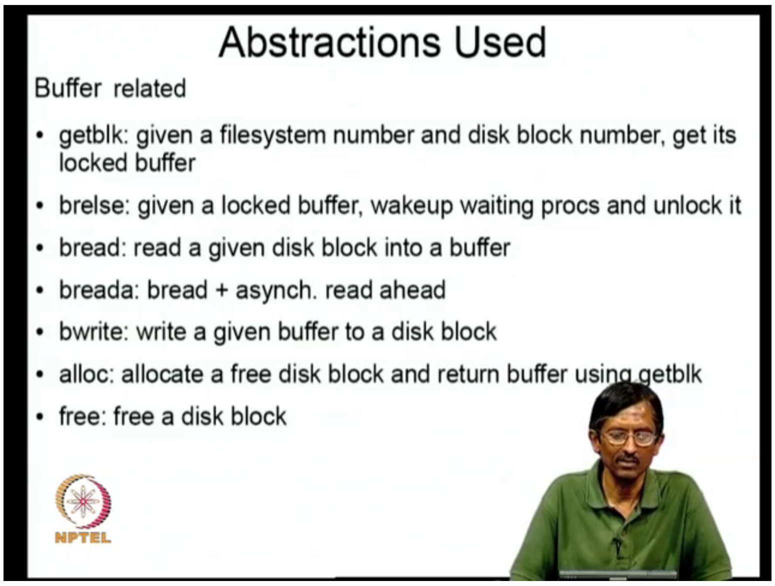


Inode related

- 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)

And there are similar ones for managing other data structures. The ones I have listed here are the ones that have come traditionally in UNIX file systems.

(Refer Slide Time: 00:48)



Abstractions Used

Buffer related

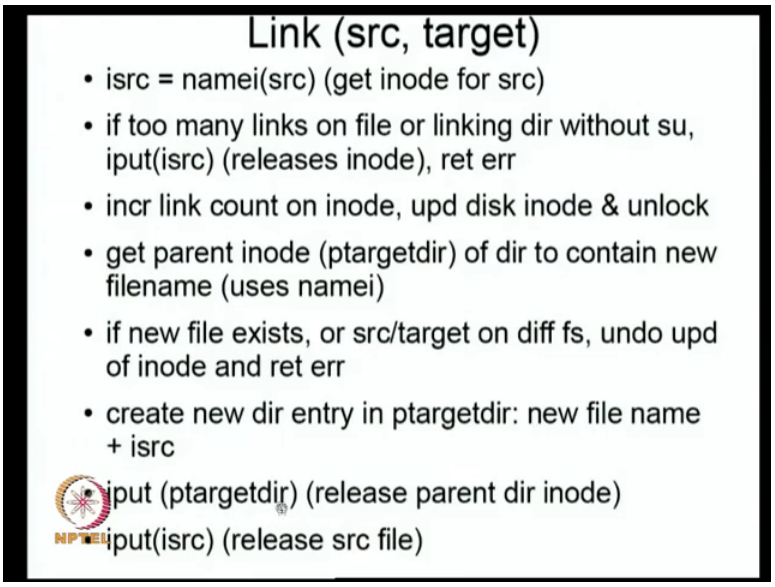
- getblk: given a filesystem number and disk block number, get its locked buffer
- brelse: given a locked buffer, wakeup waiting procs and unlock it
- bread: read a given disk block into a buffer
- breada: bread + asynch. read ahead
- bwrite: write a given buffer to a disk block
- alloc: allocate a free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

For example, the ones I am listing here now what are those found in UNIX s b 3; that is, system file release version 3. And you will find it documented quite well in the book by (Refer Time: 01:08) buck the design of the operating system. I am taking the examples from there. So, this is gives you some idea of that. There is some inherit complexity in designing a reliable file system. Therefore, the only way to get it right is to use certain interfaces that have proved to be useful as a way to, let us say structure the writing of the system call. And you notice that in each of these cases. There is a specific locking aspect also bit into it.

For example, you may get I get a locked inode, because this seems to be a common medium. So, the common medium have been factored into certain types of internal interfaces. Of course, these internal interfaces are not let us say cast installed. They will keep varying as designs evolve. And what I am giving you right now is only what is found in UNIX derived file systems. And if you go to completely non unixed or even very recent file systems they may not have these kind of interfaces. There are internal interfaces, make nothing is guaranteed about it is availability as time progresses. But these are good about, because you will get lots some idea about what is involved. Let just take a simple example. These kind of interfaces for designed or the time when file systems for small. For example, forty megabytes file systems, these are better. So, the chances of any errors happening were small. And so, you will notice that there is an implicit assumption in these interfaces, but the inodes are not replicated neither is data replicated. Where is by look at advanced file systems, the current generation file systems which try to handle not to just terabyte even petabyte things. If you do not replicate inode, the chances of your surviving keeping the file system intact is almost negligible.

So, these things assume that this is one is single inode, there is no copy of it. Or there is no interface by which you can talk about a single copy of an inode. None of the things there. So, if you really try to think about it. The internal interfaces will keep changing because of different needs different perspectives as time progresses. Again, as I mentioned this is an older one this is as of mid-80's or earlier 90's. And, but you will find all UNIX derived file systems typical have something similar to this. Let us just quickly look at one particular operation to make certain things a bit more concrete, to see how it looks like.

(Refer Slide Time: 03:52)



Link (src, target)

- isrc = namei(src) (get inode for src)
- if too many links on file or linking dir without su, iput(isrc) (releases inode), ret err
- incr link count on inode, upd disk inode & unlock
- get parent inode (ptargetdir) of dir to contain new filename (uses namei)
- if new file exists, or src/target on diff fs, undo upd of inode and ret err
- create new dir entry in ptargetdir: new file name + isrc
- iput (ptargetdir) (release parent dir inode)
- iput(isrc) (release src file)

In Posix kind of file systems, you have a notion of a link, and basically what is attempted is that you want to there is a source file, and you want to create a namein a particular directory which has the same inode as that of other source. This is the source inode, and this can be the target, file name, slash target directory. If it is a target directory, then you want to create the same name as this source file name, and has the same inode as the source file; that means, there is a single object single file with 2 references. It depends on what the target is target could be directory or it could a file name depending on the appropriate discrimination is made. So, let us see how this works out. So, what is to be done here? First of all, I just want to illustrate how some of the internal interfaces are used. Again, as I mentioned that this is a mid-80s design. So, concurrency levels are small; that means, the only concurrency that we are talking about here are because of other interrupt generating devices for example. This is going to think that are mainly concurrently with this kernel code. Nothing else is running concurrently. That is only one active thing in the kernel, any other concurrent thing is mostly interrupt interrupted driven interrupt routines, that is all.

So, that is why the code looks a bit simple, but if you think about more recent file systems, which attempts to really concurrent as lots of operations. It will be not this simple. It will be it will be much, much more complicated. So, let us see what is there here. First of all, as I mentioned source is a file name which has to be, let us say you need to create another file name which has a same inode as that of the source. So, what is the first step I do? I do namei source. What is namei do? It basically does parsing of the file name, and then finally, after doing all that parsing of so many levels of parsing. It figures out the corresponding inode for it. So, this is just example of an internal interface that is being used name namei. So, basically it gets inode for source. Now one thing about operating systems and of course, storage systems is that you have to distance your checking. So, if there is usual error checking going on inode. If there is single line or other line will be error checking. So, as usual if it checks whether does it have too many links on file. It may be that where do where checking for this kind of things, it may be there is a bug in the system. Somewhere somebody has written a loop it is just linking out of control. You just want to catch all those kinds of things. Usually it helps to catch those kind of errors as quickly as possible.

Normally what happens is that there are number of links is that field usually it is about let us say 8 bits or 6 bits or whatever. So, it can overflow also. So, the thing is to see if there is a way to catch it. That is how they are trying to do these all things. Or if somebody is trying to link a directories without super user privileges. It is possible for route 2 do linking of directories. Now super user can do it not users. So, in case this is happens to be that there are directories involved for that, source also is a directory, then this should be a disallowed. So, that is what you should basically returns on error, but before that you have to you already got the inode, and when you notice that when you get the inode you got it locked. That is what the semantics or name wise I. So, you have to basically put it back you release the inode.

You got it locked here you have to release it. So, because you are going to return that error you better clean up before you go. That is what you do, you cleaning up and then going back. Now the next thing is since you got it in memory you can certainly there is a disk inode there is a memory inode. So, in the memory inode increment the link count. Now this particular think is basically saying that even before I have done anything I can increment the link. And this what is this is what is process being followed here, better that I point to these are 2 possibilities. I have to avoid the following, I should not point under what any circumstance to garbage. Or to or null, I should not have a null pointer affairs that is something I have to avoid. The other possibility is that I do not mind some garbage being created. So, idea here is I do not mind garbage being created, but I want o avoid the situation where I point to illegal structures. That is basically what I am.

So, what are we doing out here? What is the policy being adopted by this mid 1980's thing it says increment link count on inode. So, the question for us is why is it safe to do it. Is it safe or it is I could have created a form that is something we have to think about. Now if in case you have increment inode, and when you update the disk inode; that means, even for I have done anything you already updated. It suppose is the crash here. Now the question is it possible for me to recover from. It turns out that f s c k. Can recover from this. Because it follows all the pointers and defines that it is actually pointing one thing, what is on that. And therefore, it can say that it can reset the link count to exactly 1. So, as long as you assume that there is a f s c k which can track every single pointer and disk. And then counter the number of (Refer Time: 10:06) then this is the disk into that.

So, increment is a link count on inode update disk inode and unlock. Now what is that you unlock here? Because I am going to do now another namei which can take a long time. Because I can use again path traversing because a target is here now to both path traversing. So, it is like it take can a long time. I want to let other people who could be using who want to just look at it quickly. For example, somebody could be doing a stat call. So, luckily the thing is already cached in memory, that guy can really look at the quickly and mock walk out. I want to allow those kind of things to happen. That is why I drop the lock. And then when I will need it. I will take it again I can use an example by which they are very conscious about the fact that we are dealing with a slow devices. Extremely slow devices, we are talking about milliseconds, tens of milliseconds. So, these lot of things can happen therefore, they unlock it.

Now you go on try to get the inode of the parent inode, where this particular new link has to be said has to be presented. You could have specified it directly here; that means, I have to get the inode corresponding to that directory. Or I could have specified a target file in which case I have to look at the target file, and then remove the end part take the directory part, and then look it up. That is what the namei is going to do. Ts going to come now it is going to figure out what is the directory in which this new target file is going to come in. Again, it is this namei again as I mentioned it can be a long operation. It is a path traversal it can have multiple components slashes namei slash whatever it is.

So, in principle it cannot take one 50 milliseconds. If you are struck and it has to be put up from the directory, it is a wrong operation. That is why this analogue is written. Now standard things if new file exists or source target on different file systems, something is not we can drop it. So, if the new file already exists on the directory on the directory if it has to put, you cannot do it. Source target on different file systems. Now you are doing what is called a hard link. Hard link demands that, because you are sharing the inode now. You are not you are not creating any new that is data you are just creating a new namein the file system new space, and then pointing the same object. So, it has got only one representation.

So, it cannot be on a different file system, because different file system could be having a different altogether different type of file system. For example, it could be e x t 2 on 1 and e x t 3 or let us say z f s on something else. Now these 2 guys cannot share the same inode, because they are 2 different inodes. That is why it checks whether source and target are in different file systems. And then if that is a case, then it is not possible, but I have already updated the disk inode I need to undo it.

Now, of course, there are some problems here also. It may be that at this point I crash. Again, we are hoping that f s c k can fix it right. So, at every point it can crash. At every point you need somebody who can tell you whether they can recover cleanly and come out. So, luckily here also there is no problem. Just like we could recover from here we could also recover from that nothing else is happened. All you have done is we just picked up a we just did a path name traverse, and then feed over the inode, nothing else has happened. Now after this you create new directory entry in page directory in the parent directory corresponding to the target file. Or yeah, and then basically in the new directory, you have to take the new name, and the corresponding inode. So, the basically you direct directory is basically nothing more than a knot between namei file, and it is inode that is all it takes. So, for you have to introduce these 2 components. The new file, name and the inode that you put up here for the source.

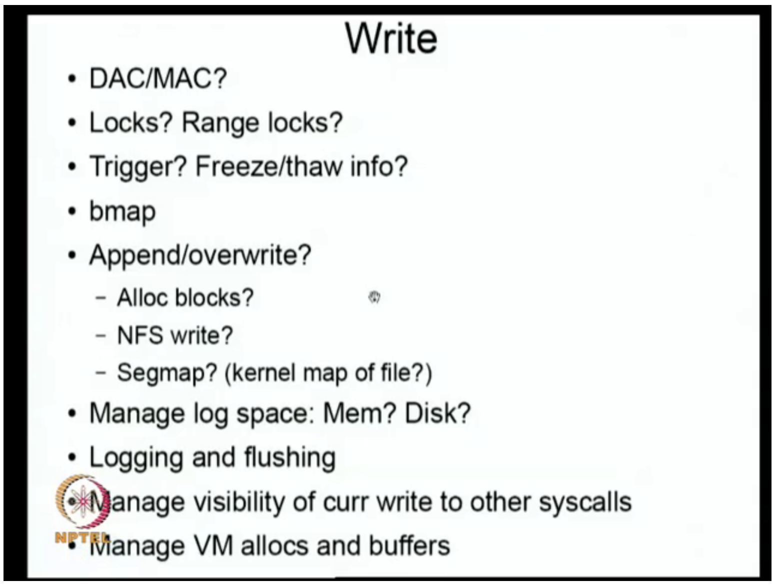
Once this is done, I have to input the parent directory, because I want to release it. What does release mean? Because notice that I may got all these things namei it is always locked. So, I am again releasing the lock. And that way it is staying in memory. So, somebody else wants it they can again get it from memory they do not have to go to that disk same thing with the source also, the source file I also then input I release. Now this is roughly outline of a link comment, and it is quite simple the way it looks. But in practice with concurrency with multi-threaded kernels, this is not going to be easy it is going to be that simple. What will happen often is that you might find that as your doing things can change. For example, as I mentioned here you unlock the inode here, right? In a concurrent multi-threaded kernel, it may be that by the time you get to doing this, somebody has deleted this particular source inode itself, it is possible.

So, you have to e careful, you have to keep checking things again. Because notice that nobody has taken the lock on the directory where source is present. I think nothing is here as far as the another no place is here, which takes a lock on that; that means, that in principal some other party could have come and deleted this thing. So, you have to as you o along you have to keep telling yourself, f there is concurrency in a system could somebody have come and take it away from you. I thought that it going to be there, but as I am working on doing something, I am doing that path transversal for the target somebody has could have comer and ran away with my inode.

And I thought t was present in here. So, again I have to check it. So, if there is a extensive amount of checking that goes on. And so, things are not of course, the easier thing is to just take a lock on this itself. If it take a lock on it, it is possible that you are extremely slow. Because basically you are saying I lock everything and then I want to something. So, if you are yeah basically pessimistic totally pessimistic. Let us assume some all things are some bad things happen. So, it is something like going around with an battle tank. You are going to be slow. Whereas, here what the typical because finally, a file system has to be quick and fast, otherwise it is going to be extremely slow.

So, a strategy is to say that I will be optimistic, I will release the locks as and when necessary. There are some situations where the kind of stuff I am looking for somebody unfortunately is continuing for at the same time as long as I assume that there is infrequent, then n those cases only is suffer these the checking cost. I check the cost, it is available immediately. I do not have to worry about it. I do small amounts of the as incur the cost checking, but typically it is always successful. Typically, nobody runs away with it. So, it is not a cost serious cost. So, concurrency really changes the whole game it is not trivial, and that is the reason why even today you will established file systems you find no bugs. It is not because the it is just inherently the fact that there is a lot of concurrency in a system and there are lots of intervening they intervening things that can happen, and there are no models of what is correctness in a file system or for that matter in a operating system. Essentially it is a the long practice certain things are considered reasonable. With I will come to that I will show what is the semantics of a file system. And because of that it turns out that it is not easy to specify what exactly it is called a correct file system operation. Because it will it will become clearer as we talk about it.

(Refer Slide Time: 19:02)



Write

- DAC/MAC?
- Locks? Range locks?
- Trigger? Freeze/thaw info?
- bmap
- Append/overwrite?
  - Alloc blocks?
  - NFS write?
  - Segmap? (kernel map of file?)
- Manage log space: Mem? Disk?
- Logging and flushing
- Manage visibility of curr write to other syscalls
- Manage VM allocs and buffers

So, again I will give a then a slightly high-level idea a write. This is a again extremely simplified version. And we will just go through what are the steps. Typically write tends to be some of the most complex operations in a file system. First one thing is should already mention is that, which I have already mentioned is that when you do a write, you can be coming from an application which is using something like f write. F write is a life c call, and life c somebody has to look at f write and convert it to a system call. One system call comes in you now have entered the kernel. Once you enter into the kernel, you first go through what is called the it is called z f s what is it tend for file system independent operations.

So, you have to go through the file system independent operations. And there are certain things which are done in the file system file system independent operations. What are those typically it is security. What is involved in a security if there are multiple models in file systems. In older systems or simpler systems, now what is called discretionary access control mechanisms. In the more secured type of file systems, they try t give what is called mandatory access control systems. What is the different because these 2 things? In discretionary access control mechanisms, what happens is that; if somebody is given a permission, he in turn or she in turn can give that object to anybody else, you have got an object in your hands. You can give it to anybody as long as you got it. So, whereas, in mandatory access control, even if you the access to the object, you cannot give access to it to anybody else. It has to be checked it will again checked by some secure part of the system to see if it can be given to anybody else. So, when you look at the write call, it will check whether you have which of those credentials do you have. You have dac credentials or mac credential. And only if you have the appropriate ones you are the owner.

Other thing is writes can have what are called range locks. For example, there are some locks available at user level, here are some locks available at a kernel level. What are the kinds of locks available at user level? There is something called f n c t l. F n c t l basically says it says file control. So, you can take a lock and a file striating from particular off set to another off set. So, this is actually visible this to one or visible at the at the programming level. So, this has to be therefore, it is will be supported k. So, this one lock that is visible from the outside itself. But the other locks also which have to be kept for internal consistency of the file by the file system. We will come to that a bit more in detail. So, this is a another lock that has to be handled. It actually turns out that the number of locks is can be dramatically large. If you look at a production quality file system, you will find that in addition to the user visible lock. You might have as many as 30 locks for a single file inside the kernel, basically, because you might have various locks predicting various parts of the files as a file or the metadata.

So, you basically then being that you want to allow as much concurrency as possible. So, invariably turns out that if you have a course lock, it turns out it hinders some other operation which could have preceded in case our file grind locks and it dint actually contain further part of it. So, as I mentioned earlier file system operations can be slow compared to memory operations. Therefore, anybody hindering it because of this can be delayed by as much as disputes. So, it can result in arbitrarily slow performance if an loop or whatever. And therefore, sophisticated file systems try to given much ore grinder locks, but that itself creates uneasy complications.

Because once I have so many locks. Then we have dead lock possibilities coming left and right. The number of possibilities just explode. So, you have to really have a extremely tight coding standards to get anything working. And it is not just the locks that is as a file system. It will also be the locks of the beyond sub system the actual memory sub system. As we discussed before there is a strong interaction between the file system and the virtual memory sub system. And the virtual memory sub system also take certain locks. For example, a drift based locks. If you are doing a nmap t will take address-based locks. Another kinds of locks, corresponding to the dealing part of it

So, when you are doing some operation, there has to be some ordering of locks. Occurs all these also. So, designers of these kind of file systems carefully think about ordering of these locks, then only proceed with this it turns out to be non-trivial. These things are better done by experts, and let us say people with lot of experience. Of course, one way to avoid all these problems is to say that a file system does not belong in the kernel. It should be outside of it. They can be macro kernel operating systems, a file system like minix 3; it puts the file system outside.

So now, the interaction between the file system locking and the d m locking can be eliminated a bit, but it turns out that it will do other problems. Because concurrency is concurrency you if you can you can exile the file system outside kernel, but if you are planning to do concurrency, there will be some locks at the file system level at user level. Even those things also if you want to do it efficiently you need to have multiple locks there. And they will also interact to their locks and he same kind of problems will occur, just like Posix p thread programmings is complex. You can get into dead locks of the p threads level. So, for the same reason you can also do the same problems at in the file system level, even if you take it outside the kernel.

So, you cannot really wish away the problem of concurrency, wherever you have to face it. T just that it might make it easier if you want to debug it. Because if you are outside the kernel debugging is that much simpler compared to debugging inside the kernel. You have more lot more infrastructure outside the kernel if compared to inside. So, again as I mentioned before if you are wanting to report access of a file at any time that s file is accessed, you need to have some triggers. Again, there is some rapper routines out here. Now one thing that you have to remember is that, you need to figure out how to continue after reporting this event.

So, in general you need what is called continuations. You have walked into the kernel, and then you detected the situation that you needed to report it somebody. And, but after reporting you have to proceed; that means, that there is some continuation that has to be kept around. So, that you report typically you do an up call, you go outside the kernel. It does whatever it has does it and you have to come back and start exactly that where you left; that means, you need to save the state of the kernel stack. And then the switch stack to the stack of the called procedure which is outside the system k. So, you need to manage this business of stacks happening etcetera carefully. Again, if you are talking about sophisticated file systems. These are complicated systems.

There are many reasons why if you have these file systems have to be frozen; that means, that you have to freeze all that little bit within it. So, that we can do some surgery on it somewhere. Often times just like you anasthasia the patient, you have to have to anasthasia the file system. Because there are so many components in it you cannot do some major surgery without ensuring that there is no f not you should make sure there is no activity while doing a surgery. Otherwise some things can go really bad. So, you might have to freeze a file system; that means, that you have to before freezing you have to figure out what all the activities there are there. And you have to make sure each of these activities is going to be dominated in clean manner, made to somehow come out of the file system if possible or if it is cannot be made to come out of the file system you have to somehow get into a consistent state. Some easily check point of the state.

Now, this might look this simple, but actually it turns out to be fairly complicated. Reason being as I mentioned before a file system interacts with a v m sub system another sub systems, and they can do mature recursive calls; that means, that you could have come in through the file system and walked into the v m sub system. And again, sub system could be doing strapping operations which s running on one of the file system, or on the same file system; that means, you have made a recursive entry into the file system. Now we have to keep track of how any times have I where have I come in from, and where we have to go out in some sense.

Again, as I mentioned to you need to keep track of the continuations. This turns out be non-trivial, this is something also we have to be careful and you need to provide some infrastructure by which you know how you came in and how you can safely come out. So, basically, you can freeze a file system you can thaw it also. You can thaw it means you have already you passivated the file system, and then once that surgery is done you want to let it go here again. Let me let it go again you have to see what state we left t in and slowly bring back it in the right way.

And unfortunately, the details are complex so, but given in a file system there is some with some careful engineering you can get it back, but it is nontrivial. So, this is some of the things that you also have to provide basically I am saying is that when you enter the file system for a write web have to record the information somewhere. And what level you came in did you come in the first time or you come in through some other or you came in a recursively entry etcetera. You have to record all these information somewhere. And you have to do it efficiently you cannot remember all kinds of things you realise that structures become too complicated.

So, you note some information about how you came here. And the next thing is that you have to do a bmap as I mentioned to you what is a bma it basically given a file and a offset. It tells you what is the block corresponding to that particular starting, starting address of the starting block address. You have to deal that. Again, this one can be on non-nontrivial because to be a bmap you may have to go on pick up the direct blocks already available, but indirect blocks have to pick up; that means, that as I am doing bmap, I have to go to disk and pick up direct and indirect blocks. Now I have; that means, that your system is sort of your call is waiting while a disk operation is going on. So, what it means also is that, while that is going on it is possible that concurrent activities might invalid this some information. Again, you have to some rechecking there. So, there are issues of even in something like bmap in a concurrent multi-threaded kernel there are could be lots of complications here. So, these things the actual code is really amazing the complex.

There are even worst part of it is that there are no specifications. You have to do what is required to get most of the application code running correctly. As I mentioned to you earlier, there are lot of scripts out there, if you get some of the script if you get your file systems semantics slightly different then what to we are expecting. They will say that file system is bugging, but here is no specification there is no way to you can not prove that somebody is bugging because no specification, but they will tell you that you are bugging you have to go. And fix it and things typically are connected with all these flushing. All these kind of things it means, you can leave certain things in memory you do not flush it and somebody is coming through some other interface and since they are not coming.

So, file system interface; they can see something else. For example, in many file systems you can come though you can access the same structures thought the file system interfaces or you can come through device interface. Because in UNIX a device is also a file. And you can export it a file is also a device. And f you do all these calls look back, there is something called loop devices. You can take a file and think of it as a as a device, and then you can put a file system on top of it. That is where a single file, one gigabyte file and then you can tell the file system that I am going to think about it has a device and I am on top of it I am going put another file system on it. That full ne gigabyte will be a single it is going to multi internal structure is going to be there. So, because of these kinds of looping and recursive structures. It is non-trivial to get a semantics fact will go through a bit.

So, other thing it has to do it to think of appends and overwrite the difference between append and overwrite. Appends are slightly easier, because only you have to do is to go to end bend and to extend it. But then you have to worry about concurrent appends, how do you ensure that only of them appends you have to make sure that append is complete before he second append comes in; that means, that you need to take specific locks. And the way sometime you are going to do is to have a lock on certain length sizes for example. And you can also have overwrites, and overwrites can have holes in it; that means, that you can do allocations here also. So, bmap can do reads of disblocks. Overwrite what can happen is that you have a file, and somebody ahs without writing anything has to seek to it a person has seeked further lock. So, UNIX has the semantics that it is seek, you can have holes in it; that means, the blocks are not allocated. Now you can write to that hole; that means, that there is no block there; that means, you have to allocate a block. So, in overwrite t can turn out that you might have to allocate blocks.

So, that means, that as a part of write you could be talking to various block allocation routines. And you will also updating the number of free available blocks and other kinds of things. So, sometimes you also have to worry about some specific things. It turns out NFS write is a major issue, because they have synchronous. So, for a long time NFS was synchronous byte and synchronous byte as everybody knows is slow. So, the idea is to see if there is something can be done to make it fast. So, there are lot of heuristics to discover a write is a NFS write; that is, you have a this local file system, and through NFS client is asking something to be written, it comes to the NFS server and finally, it is going to the local file system. The local file system sees it as a write. It does not know it is coming from NFS that is coming from NFS client, because it is being redirected through NFS server.

So, it has slime heuristics to figure out that it is probably coming from NFS thing, because usually 8 k is usually considered a signal that that is a called a NFS write NFS writes are typically 8 k 8 kilobytes. So, it will actually guess that something is NFS write. And probably it can do some logging, it basically instead of writing it, because the problem with writing it synchronously is that you are going take the discrete. So, what you want to do is to see if you can log. It anyway you going to do lot of writes to disk as a part of other operations and you are creating a log. And you will lock the NFS write itself onto the log, and it will go as per part of every other so many other writes, k so many other file system activities that are going on concurrently.

So, you might actually and what is the cost f flushing to disk. So, there is some support to these kind of things also. And other issue is in advanced file systems, typically writes are written by kernel maps. What you do is; it take the file that has to be written, and you map it to that a kernel disk base. And you copy it from the user buffer into the kernel memory. This way what they allow you is that that is a single copy of the written stuff it is not there in 2 places. For example, if you do it in a older type of file systems, you will find that you will be having it one copy in buffer cache, and one copy in the in the v m sub system.

So, by doing these kernel maps, you have only single copy, and that turns out to be quite efficient. So, again all these sigma (Refer Time: 39:00) etcetera all there; that means, that you re now actually using virtual memory sub system calls to implement this. So, there is an a another example how a file system is very tightly connected with the virtual memory file system. And as part of writing it, you may actually suffer a page fault and now again you will enter the v m file system. Because that has to be handled by the kernel when the when the it when you try to write from the user buffer to the kernel map region, you might suffer a fault.

And then the kernel will notice that this has been because of the segmap write, and when it knows that it has to bring the corresponding blocks, or corresponding information that is present in the disk. It has to bring it back. And so, some of those things also have to be done. And usually what happens is that this particular disk actually could be is present as part of the file system because you are writing a file system code; that means, that only file system can participate. So, the kernel might not understand how to pick up that particular block. It only knows that it is a file followed by a particular offset, that might again it means that it has to talk to the file system. And the way it is done by calling something called a get page routine. You are basically saying, the kernel basically says there is a fault here, and you are the only person who knows how to read it because all I know is a file in a offset. I do not know anything beyond it. And this is where this file and offset information is not file system specific it is file system independent. So, that is all it knows

So, again it has tell the file system that please go and bring your file with this offset. So, that means that again it has to a file system s again re-entered for the get page. So, you enter the file system. And so, basically you doing a write, as part of the write you have gone into the v m sub system, and v m sub system there is a fault while you are writing it, from the user address user buffer to the corresponding place in a file system that we are supposed to write that is because it is not already present in memory you fault because you fault you have all you know is that you have to pick up that buffer from the disk. And only the file system knows how to pass that information. So, again; that means, that the v m sub system itself now calls the vob get page. And so, get page is done, the corresponding block is picked up and brought to memory by the file system code independent is a second operation now. Once that is done, then again sub system comes into picture then it now finally, gives that page it is available in memory, now the write actually can start. The actual write this can happen every block. Every page it can happen these things.

So, it turns out again I simplified it very much. It turns out there are so many other concurrent operations going on at a same time. Because the pages can be relocated, it can be for example, there could be another file system call which is doing, what is called truncate. What does the truncate do? It basically says, I want to chop off this portion of the file form here to here; that is, have a file which is let us say 8 megabytes. I can set truncate to 0 bytes. So, somebody is running this particular system call in parallel with my get page. What is the semantics of this? We have to decide. Normally the understanding is that if somebody is a get page, and there is a truncate going on, then you have to think about inter links. So, either we write whether you have a get page or you can either do a get page first for arbitrary truncate, or a truncate for a get page. Normally the semantics is that anyway the truncate is supposed to if it is done, right irrespective of which order you do it. If you are resulting any pages to be given; that means, you have to stop all get pages right.

So, you need to prevent get age to go through when truncate is going on at the same time; that means, there is a specific lock for this purpose. Again, we will talk about it. So, there are lot of issues of these kind, that a file system reason has to worry about. So, it is highly specialised kind of work, with more specifications that are available across consumers and file system designers slash implementers unfortunately. So, it creates lots of complications. In addition to all these things you have to manage the lock space. You might be running out of memory, because first of all when you do logging, you first do t in memory.

Then only it pushes out to disk. It is possible that you run out of memory. In run out of memory what can happen? You want to see if there are nay buffers that can be flushed to disk, right? Or you want to see if any delayed writes are there which is storing which is holding on certain files which are in memory which could be profitably sent out to disk; that means, that as a part of this whole write itself, I might be doing some disk freshers. Which also you can take some space time, while that is going on I cannot really proceed, I just have to stop here right. So, and what is interesting is that while I am being at flushing, it may turn out that they might suffer some errors. Because nobody guarantees that any time I have to do a write it is going to go through correctly. The disk might fail anything can happen; that means, that again there has to be some infrastructure in the kernel by which while I am trying to do some operations of these kind if there is a problem I come out of it clean idea; that means, that there has to be some kind of minimal information that I have to keep here in some sense it as to be somewhere at the checkpoint of information.

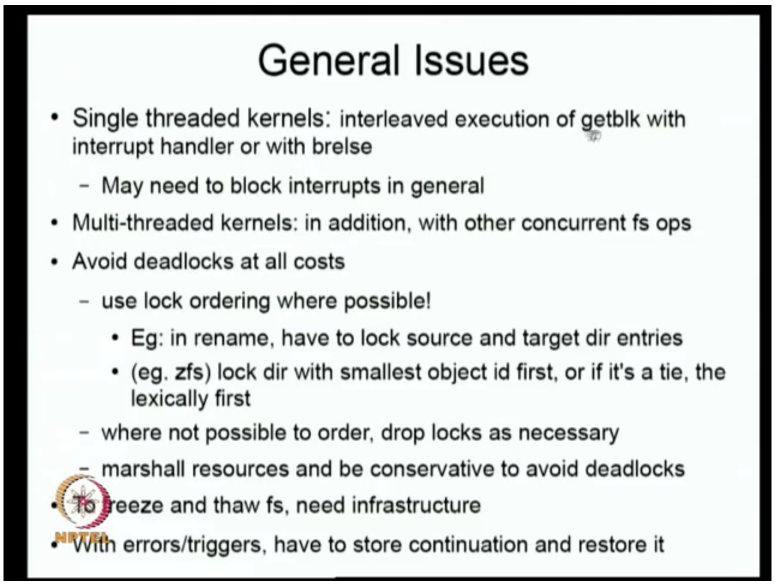
So, at any point you have to keep doing this. And so, you have manage lock space at time you have to do the logging you have to do the flushing. All these things has to be done first of all manage the space ensure that everything is, then you start doing this logging and flushing. And other thing that we have to worry about is manage visibility of current write to other system calls. As I mentioned to you earlier if you are writing to your appending to a file, the question is what can ever people see as you are doing it. Can they see it at all or can they you have to complete the job for anybody can see it. Now different file systems again this is one example where the specifications are not available.

Nobody tells you what these specifications are so, different file systems aggressive file systems do one way the simple file system do a simple way. For example, if your file system you basically say until I finish my append I would not let anybody see; that means, that the other parties who are concurrently looking at it, they will see this file as a static object. They will just see it as not being extended. Whereas, if you are an aggressive file system, you want to ensure that as you are writing as you are appending it f it has crossed a certain threshold of a set of a sector or of a block or whatever, I want to keep on excluding to other people that the file has been extended. Even a math file can see it is extensions, even though it looks a bit heavy, but it can be done.

So, the question is how much visibility you are providing into this operation which is let us say do you want to do it as a atomic operation, or you want to do it as a somewhat let us say long operation with multiple small things being atomic. So, again depending on what you do some applications will work certain applications will fail. For example, if there is a mail reader and then somebody is extending the file, and then while the f the write is going on, let us say right the write can be long because, for example, it may be that somebody has taken a huge file has coming from somewhere. And the region appended t could be megabyte or so. During the file that is being written. You will probably get some information like no files have been referred to which can actually possibly cause wrong information to the given while the you are updating the file may be. So, there are differences in file system how they handle this. In nature all these things you have to allocate virtual memory areas and buffers those also have to handled. Again, once you are done with the write you have to drop the locks, you have to the freeze down the information, we have to update it a trigger information, we have to again let us say complete whatever has to be done at the end of this triggering operation, etcetera, etcetera. All these things also have to be concluded at the end.

So, generally what are the kinds of issues, in single thread kernels as mentioned before, intuitive execution of things like get block with interrupt handler or with you can block with b release.

(Refer Slide Time: 49:30)



General Issues

- Single threaded kernels: interleaved execution of getblk with interrupt handler or with brelse
  - May need to block interrupts in general
- Multi-threaded kernels: in addition, with other concurrent fs ops
- Avoid deadlocks at all costs
  - use lock ordering where possible!
    - Eg: in rename, have to lock source and target dir entries
    - (eg. zfs) lock dir with smallest object id first, or if it's a tie, the lexically first
  - where not possible to order, drop locks as necessary
  - marshall resources and be conservative to avoid deadlocks
- To freeze and thaw fs, need infrastructure
- With errors/triggers, have to store continuation and restore it

If there are some interrupt executions, which can create problems you have to work out their details; that means, that every single interface how many interacts everywhere interface. Every interface internal interface how it interacts with interrupts. It has to be worked out. It is a (Refer Time: 49:51) taking work it as to be done. So, generally if you want to avoid this thinking about his interrupt handler you have to block it interrupts. So, n multi-threaded kernels, you have to also worry about other concurrent system calls.

In addition to these things, you have to worry about others concurrent system calls. You have to important thing about this is that you have to avoid deadlocks. At all costs your file system gets deadlocked you are dead. Critical piece of the infrastructure kernel infrastructure is dead it is not possible; that means, that we have to do what is called dead lock avoidance. And in case there is actually a dead lock avoidance at as much as possible, if there is once in a while if you really get dead locked, you may want to really provide some ways to break the deadlocks. Usually if dead lock avoidance is what everybody attempts as much as possible.

So, for example, you have to use lock ordering where possible. Example in a rename what is a rename? You basically saying that a certain part of the file system tree is going to be given a different name at a different place. You basically do grafting one part of the file system is placed to some other place. So, you basically have to lock the source and target directory entries. What can happen is that somebody can be doing renamein the opposite direction. Nothing prevents it from happening that is I am doing renaming from a to b somebody could be doing from b to a, you have to assume the worst case. So, then basically what you have to do is, you have to find the way of getting the some object let us say example v node or inode or z node or whatever, right. Then it tells z f s is called z node, you basically find lock directory with the smallest object I d.

First these are numbers does not matter what it is just take the smallest I d. You take the lock on it first followed by the one with higher number. It so happens that you are dealing actually has the same directories, because I can always rename 2 files which are in same directory. I can rename a file in one directory to the some other name of the same directory; that means, that the source and target memory will be the same there is a tie therefore, then you have to figure the name of the thing figure out which one is going to be out there. So, we have to have some kind of mechanisms. If it not possible to order then you always take locks opportunistically, and then the minute the it looks has it is certain lock kind of situations you drop locks appropriately come out of it you are doing all the time. And as I have told you one thing about concurrent operations is that while you are taking a lock if you are waiting the whole might have changed; that means, that the state of your model of the system while you are waiting for a lock will not be same after you got a lock. Again, you have to revalidate all the locks again to see if you are not, if you can not get again you loop it again you have to do all these things.

And generally, you have to get all the standard way to avoid deadlocks is to get everything you need first in your hands then you will do the operations. If you find that you do not get all the resource one you just drop you give back all the resource and retry it again. That is a standard method basically the thing is that something called you want to avoid the get and hold condition, hold and get condition. If you hold and wait then other person can do hold and wait and you can get locked. So, the ideal thing is never to this hold and wait; that means, that you basically have to get everything you want if you cannot get it you drop all of it again retry again. So, suppose these kind of issues and generally locking is the biggest headache in concurrent file systems and highly multi-threaded kernels. And this is the most difficult problem that you have to handle, given the high number in locks, and the lack of complete specifications, and what supposedly a file system supposed to do what a system call supposed to do.

It is based on based on experience, how you do these things. As I mentioned earlier, you freeze and thaw file system you need some infrastructure to count how you how you came in how you are going out that you need to keep that information actually. With errors and triggers any time there is a error in the kernel or in the file system code you may have to find the way to coming back you have to get back in some reasonable condition. So, you basically have to store continuation. Every time you go to disk there can be a failure, anytime you go. So, this is something you have to worry about.

So, I think in the next class, I will talk a bit about file system semantics, and see how that is handled. And then I will go into one file system the I have decided to look into what is called the z f s which some majorly a designed and now it is with oracle. So, I will go through z f s, a source code on z f s is available. So, it is an example of a advanced modern file system, which tries to lot of interesting things. So, t is good to look at it if you really want to take a look at it. Because the source code is available, and it is in the context of fairly sophisticated operating system like solarise, it will be if somebody wants to really look into details they can take a look at it.

Thank you.