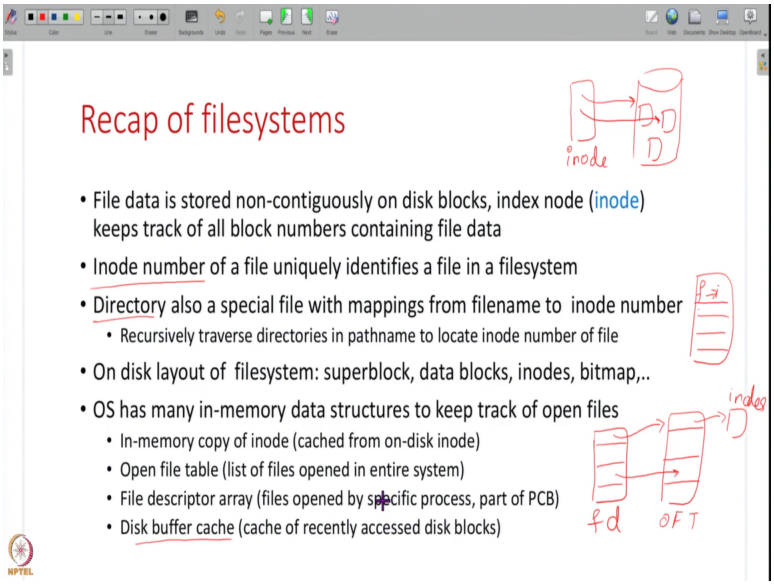
**Design and Engineering of Computer Systems**
**Professor Mythili Vutukuru**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Bombay**
**Lecture 25**
**Filesystem Implementation**

Hello, all. Welcome to the 17th lecture in the course Design and Engineering of Computer Systems. So in this lecture, we are going to continue our discussion of filesystems in operating systems. So this is a recap of the previous lecture.

(Refer Slide Time: 00:30)




We have seen that a file is stored non-contiguously on the disk. So on the disk, a file is split into fixed sized blocks and all these block numbers of a file are kept track of in the inode of a file. So this inode is the metadata, and whereas these are the data blocks of a file. And the inode number of a file uniquely identifies a file.

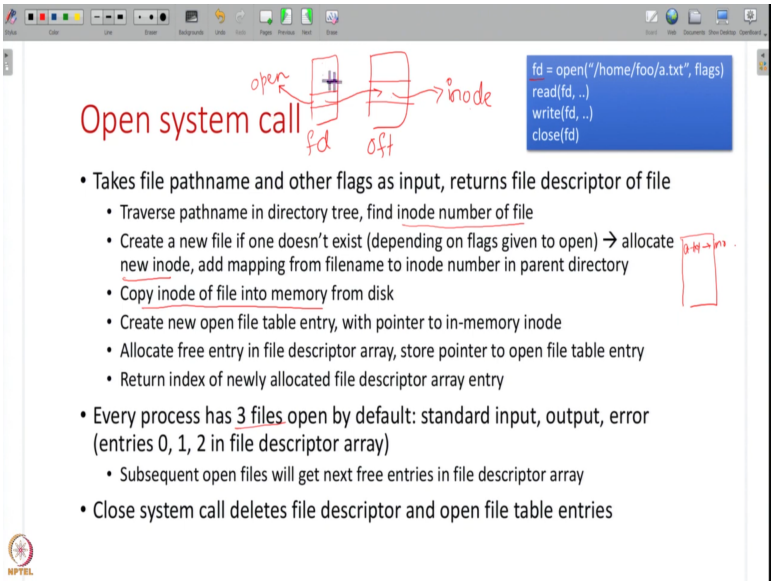
And every file and its inode number, this information is kept track of in the directory that is containing the fight. A directory is also a special file which stores information between the file name and the inode number, all of these mappings of all the files present in a directory are stored in the data blocks of a directory.

And anytime you want to find out the inode number of a file, you will recursively traverse the path name, starting from the root and finding out the inode number of a file. Then on disk, a file system consists of a superblock various data blocks, metadata blocks, like inodes, bitmaps and so on.

And in memory, a file system consists of various data structures. You have the file descriptor array, which is part of the PCB of a process. This has pointers to various open file table entries of the files that are open by this process. And this open file table has information about all open files, it has information like the offset and pointers to inodes and so on. And this inode is an in-memory cached copy of the actual line node on disk.

And you also have this buffer cache, which is the cache of that recently accessed disk blocks. So all of these are the data structures pertaining to a file system on disk and in memory and all of that, we have seen this in the previous lecture. Now in this lecture, we will see how system calls like file open, read, write all of these are implemented. So now let us begin with the open system call.

(Refer Slide Time: 02:36)




So this is simple syntax of how the open system call works, which is you take a path name as an argument and a few flags that describe how the files should be handled. And this open system call returns a file descriptor or a handle, some number it will return to you. So what does this

open system call do? It will first traverse this path name, we have seen how this is done in the previous lecture you walk this path name. And finally find the inode number of a file.

Now if this file already exists, you will find out its inode number, what if this file does not exist? then you search in this directory, you will see that there is no entry for a dot text. And if the flags indicate that create a file if it does not exist, that is one of the flags that you give. If we are asked to create a file while opening it, then we will allocate a new inode from the free inodes on disk.

And we will add a new entry like in this foo dot, in this foo directly, we will add an entry from a dot text to its new inode number, this entry will be added and you will create a file also if required during this open system call. So now, you have identified the inode number of a dot text or assigned a new inode and a new inode number if one did not exist before.

And once you have this inode number, what you will do is you will copy the inode into memory. Whenever you open a file, you will bring its inode into memory for faster access in the future. And then you will create all of these connections in the file descriptor array, in the open file table, so you will create a new entry in the open file table, which has a pointer to the in-memory copy of the inode. Then this pointer to the open file table entry will be stored in the file descriptor array of a process. And this index in the file descriptor array, this will be returned to you as part of the, as the return value from the open system call.

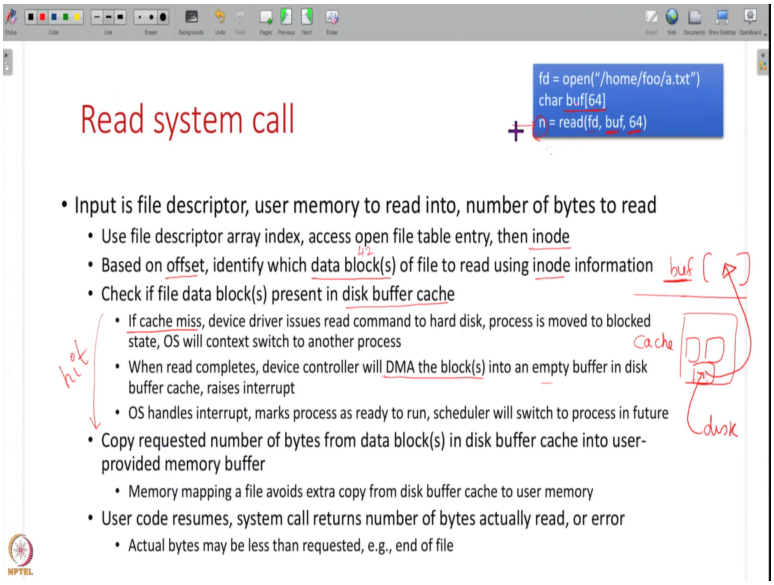
So every process by default, when the process is created, it has three files that are open, that is, in the file descriptor array, entry 0, 1, 2 are already allocated for standard input, standard output standard error, that is the way you take input, the way you standard output, which usually points to screen, standard error, all of these are already opened by default. So subsequently, if you open a file, as soon as you start a program, you will get the fourth entry, will be returned to you as a file handle.

And of course, the close system call is simply the opposite of it. It will free up this entry, this entry and so on, all of these connections that were created as part of the open system call will be freed up in the close system call. So this is a summary of how the open system call works.

And once you get this file descriptor, you will use this in all subsequent system calls to read or write a file and so on. How? From this file descriptor the pointer to the open file table entry, from

there a pointer to the inode, and in the inode, you have all the data block numbers of a file. From that, you can figure out which block on disk to access, to read or write from the file. Therefore, this written value, this file descriptor is important.

(Refer Slide Time: 05:59)




Now, how do you read a file? So let us understand a little bit more detail about the read system call. So the first step I have just described, use the file descriptor, and finally, you will access the inode of the file. That information is known to you. Now from the inode of a file, you will identify which data blocks of a file to read.

So this read system call takes these arguments, the file descriptor, then there is a buffer or user space memory and how many bytes to read. This information is given to you as part of the read system call. And you also have some offset, if you know, have you read a 100 bytes in the file so far, have you read a 1,000 bytes in the file so far, this information, the offset is also maintained in the open file table.

From all of this, you will know next, which set of bytes in the file should I read and where are they located on disk? This information you will get from the inode. So now you have some block numbers and the read system call can request large amounts of data also that can span multiple blocks also. So for simplicity, let us assume, there is a single block that you have to get from disk in order to satisfy this read request.

So how do you do that? So first suppose, we need this, the next 64 bytes are located in say block number 42 on disk. First, what you will do is you will check the disk buffer cache. Has somebody else or this process in the recent past accessed the same disk block from the disk, in which case, this block number 42 will be present in the disk buffer cache. So we will check that.

If it is present, of course, we will directly go to the next step. If it is a hit in the disk buffer cache, we will go to the next step. Otherwise, if it is a miss in the disk buffer cache, that is whichever data you want to read in this file is not present in your disk buffer cache, then you must go to the actual hard disk in order to fetch it. So this is where all of this thing we have seen about issuing a command to the disk, all of this comes up. If it is, all of this is only if it is a cache miss in your disk buffer cache.

If it is a cache miss, then the device driver will ask the hard disk to read the data and this process that has made the read system call at this point, it will block because there is no point going to the next line until this buffer has the data. The program will assume that this buffer has the data present from the file. If that is not present, what is the point of executing the next line in the code? Therefore, this process will block and OS will context switch to some other process. And at a later point of time, when the disk slowly finishes reading this block number 42 from its hardware, it will then DMA the block directly.

Where will it DMA the block? Not into this buffer, but into the disk buffer cache. So you have the disk buffer cache has many slots. For recently red disk blocks and the hard disk will DMA the block, say this block number 42, it will DMA it into one of these free entries in the disk buffer cache, and then it will raise an interrupt.

So by the end, when a interrupt is raised, then the data is already present via DMA in the disk buffer cache. So then the OS will handle the interrupt, it will say, okay, this process that was blocked, it is ready to run. The scheduler will run this process and so on. So if there is a cache miss and you have to go to disk, all of this will happen.

On the other hand, if there is a cache hit, this block of 42 is already there in cache due to a previous read, then you do not have to do all of this. So then what do you have to do? The only

thing you have to do is copy whatever bytes. So the, in the user space, there is this buffer, the 64 byte buffer that is given to you.

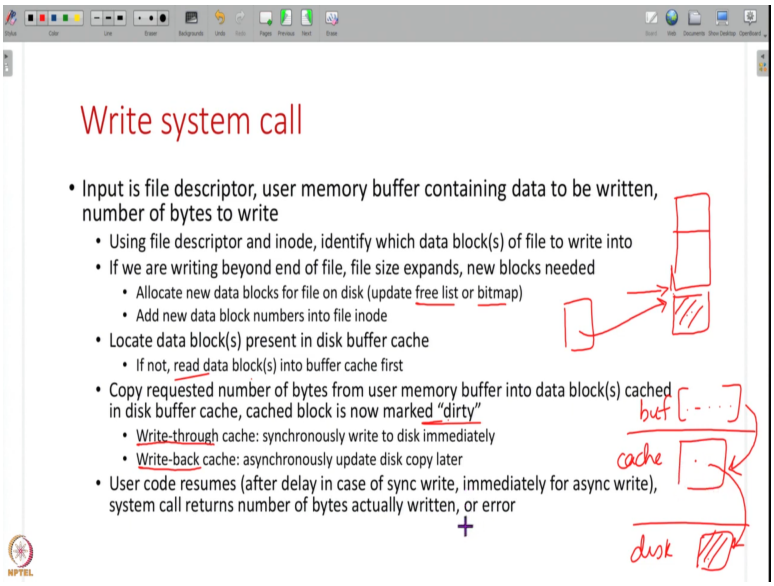
If the user has requested 64 bytes, copy them from this disk buffer cache into this user buffer, and then you can return the read system call will return. So if your memory map a file, then these extra copies are avoided. When you memory map a file, you will directly read from this cached copy using different virtual addresses, we have seen that concept before.

But if you are using the regular read system call, because the read system call may request only smaller it may request 5 bytes, 10 bytes. So therefore, you will not give access to the entire block that you have read from disk which is there in the disk buffer cache. Whatever small amount is needed, you will copy it into this user space buffer.

And then the return value from the read system call is the actual number of bytes that have been read. So sometimes if you say 64 bytes and if your file only has 32 bytes left, you might only return 32 bytes, or you might return some error value say if this read could not happen, whatever. So the written value will depend on how all of this processing actually happened. So this is a summary of how the read system call works.

And note the importance of the disk buffer cache here. If it is a hit in the disk buffer cache, the process need not be blocked and it can immediately return from read. But if it is a miss in the disk buffer cache, then it might result in the process getting blocked.

(Refer Slide Time: 11:16)




So next we will see the write system call, it is similar in structure to the read system call, the arguments are the file descriptor, some memory buffer, which has the data that has to be written and the actual number of bytes to write.

So using this file descriptor, you can identify the inode, and from the inode, you can identify which data blocks of a file should I write into. So if you are writing beyond the end of the file, that is suppose your file has 100 bytes so far and you want to your offset is at this point and you want to write the next a 100 bytes, then you will have to allocate new data blocks for this write system call. You are not writing into the existing blocks, but you are writing beyond the end of the file. If you are writing beyond the end of the file, then you need to allocate new blocks, that is you will have to go to your free list or some bitmap on your file system.

Find out some new blocks, get this new block number, then in your inode, this new block number has to be added, because this is a new member of the family. You have to tell the inode, hey, keep a pointer to this new block also. So all of these changes to the bitmap indicating that this block is occupied to the inode, adding a pointer, adding this block number, all of these changes also have to be made if you are expanding beyond the end of your file. Otherwise, if you are writing to an existing block in the file, you simply using the offset and the block numbers inode, you identify the existing block number.

And then the next step is you look for this block, is it there in the disk buffer cache? If it is not there, you will read it into the disk buffer cache first. All writes happen in memory first, they do not happen on disk. Therefore, if this block is not there in your disk buffer cache, you will first read the block into your disk buffer cache. And now, so this block number, wherever you wanted to write this file data, that block is there in cache. And then your write system call has given you a buffer of some number of bytes to write.

Then at this point, once you have obtained a copy of this block in the disk buffer cache, which might this reading might cause the process to block and all of that is done, after all of that is done, then you will copy the requested number of bytes from the user space buffer into this copy of the block in the buffer cache.

And now, this cached copy of the disk block is different from whatever was there or is there in the hard disk. You got it into the disk buffer cache, and you modified this cached copy. Therefore, now this cached block is marked as dirty. And then when do you update this disk copy, this final copy and eventually everything on disk has to be updated, because this is this disk buffer cache is just in main memory that is RAM, that is volatile when the power goes off, this is gone.

So finally you have to update the on disk copy, but when do you update that on disk copy? There are two ways. If your disk buffer cache follows the principle of a write through design, then as soon as in the write system call, you modify this a disk buffer cache entry, immediately, you will also modify the disk entry you will do a write to the disk.

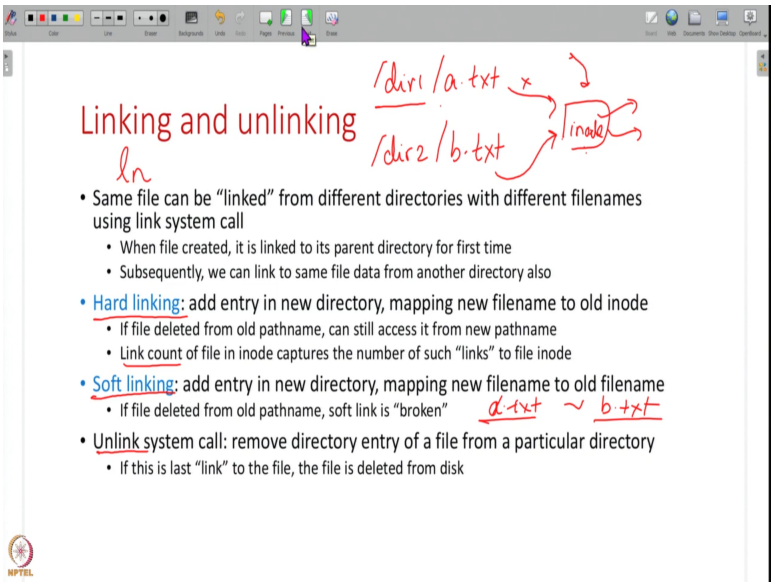
But if your cache is a write back cache, then you will say, okay, there is no hurry to do this. I will do this later. That is you will simply keep a changed copy in cache, the on disk copy has not been updated, and later on, when some such changes accumulate or something, you will write.

So now your user code, when does it resume? In the case of, these are also called synchronous writes, write through cache is also called a synchronous write. In the case of synchronous write, in the, when you make the write system call, then itself, the disk access also will happen, therefore you will return from the write system call after a delay. But if it is an asynchronous

write, if it is a write back cache, then your write system call will just dump the data into this cached buffer and it will return immediately.

Then the system call, when it will return will depend on the kind of your cache. And when it returns, it will actually tell you the number of bytes actually written. If there are some error, if all bytes could not be written, all of that information is given back to you in the form of the written value from the write system call.

(Refer Slide Time: 15:45)




So the next concept is what is called linking and unlinking a file. So the same file. Say you have a file directory one slash a dot text. So this a dot text is actually pointing to some inode number and it has some data blocks, the pointers of which are stored in this inode and so on. So the same inode the same file, you can actually link it from a different location in the directory tree, say, directory 2 slash B dot text.

So this final name can also refer to the same inode. So this is called linking a file from different locations. So when a file is created, when you create a dot text, then this inode number is linked to this directory for the first time. This is your first link. But once this inode is created, you can add many different links from many different path names to the same inode and the same file.

And in the inode, there will be a counter, a link count that basically tells you how many such links are there to this file data from different directory locations. So this type of linking, where you add a mapping between an existing inode and a new file name in a new directory that is called hard linking. And these are just different aliases or different path names to access the same underlying inode.

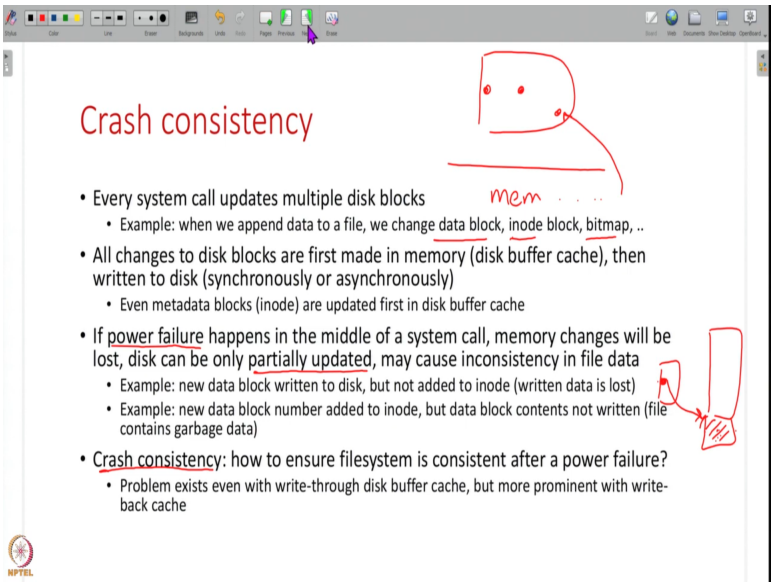
Now, if you delete this file directory one A dot text, then this link is gone, but you can still continue to access the file from this link. And once all such links an inode are gone, once its link count goes to 0, that is when the file is actually fully deleted from disk. But as long as one of these links are alive from different directories, the inode continues to exist, the data in the file will continue to exist. So this concept is when you, we say we link a file to a directory, it is a, this is called hard linking.

So the flip side of this is what is called soft linking, which is you simply add a mapping from a new file name to an old file name, that is a dot text, you will say this is equivalent to some other b dot text in some other directory. You are just storing a mapping between file names and not a link to the underlying inode number.

So now if you just soft link this B dot text to A dot text, then if you delete A dot text, you can no longer access B dot text. The soft link will be broken. Why? Because you are not storing a mapping to the underlying inode number, you are only storing a mapping to another file name. That is called soft linking and there is a difference between hard linking and soft linking. So the LN command in Linux, if you have used it, you can do both hard linking and soft linking with it.

Then there is another system call called the unlink system call, which is basically removing these links. So normally when you delete a file, when you say do RM in Linux, you are not actually deleting the file fully, you are simply unlinking it from one of the directories. And if all such links are gone, if the file is unlinked from here, unlinked from here, then it is truly deleted on disk. So this is the concept of linking and unlinking. And there are system calls for this also.

(Refer Slide Time: 19:09)




So the next step topic I would like to describe as what is called crash consistency, that is we have seen in all of these system calls, every system call updates multiple blocks on disk. You know, you, when you are appending data to a file, you are changing the data block of the file, then you might, if you have allocated a new data block, maybe you have, you are changing the inode block of the file in order to add this new block number, you are changing the bitmap block in order to mark this newly allocated data block as occupied.

So you are changing multiple blocks. If this is your disk, you are touching this disk location, this disk location, multiple things you are touching in one system call. And all of these changes are being made fast in memory, and then they are being written to disk. So your memory will first, you will change them here and then you will write it to disk. And whether it is write through cache, write back cache, whatever it is, the changes are first made in memory only then made to the disk.

And this holds true for everything. Even your inode block, if you are changing inode, that also, that inode block is also there in your disk buffer cache. You are changing that copy first, then changing your disk copy. So now, when you are doing this in two steps, first memory, then disk, what happens if a power failure occurs? The user has made a system call, the user has written some data, but before you can write that data into the hard disk, your power has failed, your memory contents are wiped out clean.

So then in such cases, what can happen? Your disk can only be partially updated and you might see some weird behavior, some inconsistent behavior in your file data because of this partial updates. For example, suppose you are writing to a file, adding data to a file.

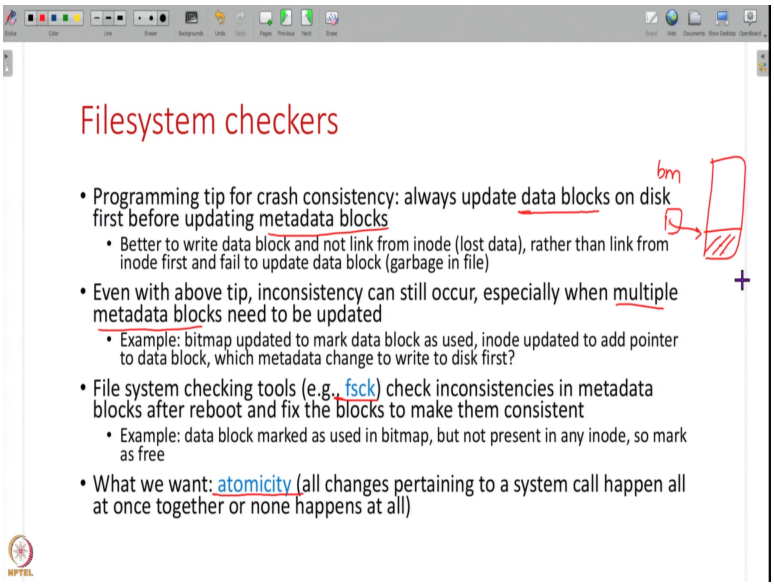
You have to, in the file, at the end of the file, you have added a new data block and you have managed to write the data to this data block, but the pointer to this data block that is there in the inode, this pointer, you have not managed to write, that is the inode on disk is not updated, but this data block is updated.

Then what will happen? You can no longer access this data. Later on when you start, if nothing, if something is not stored inode, you cannot access it. Therefore, it is like your written data is lost. Another problem, suppose you made a change to the inode, saying hey, keep a pointer to this new data block, but this data blocks content you have not written.

This data block still has some empty values or some garbage values from a previous life. Whatever it is, it has some garbage here and this pointer to this data block number you have stored in your inode, but this itself has not been updated, then what will happen when you read the file, you want to get some garbage values.

So all sorts of these weird things can happen. Therefore, when you boot up after a crash, your file system has to do some things in order to ensure that this crash, inconsistency due to a crash does not happen. And how do you do that? That is the problem of ensuring crash consistency in filesystems and the many techniques that we are going to see now. Note that this problem will exist even with write through cache, even with write back cache, anything. So then how do we guarantee crash consistency?

(Refer Slide Time: 22:38)




Modern filesystems have a many ways to deal with it. Clearly, power keeps going on and off and all of that. So we have to handle it. So one tip that a programmer who is building a file system can keep in mind is that always update the data blocks before updating the metadata blocks. So if you are writing new data at the end of a file, it is better to write this data first and then store a pointer to this data block in the inode. So that, even if this first step complete, second step does not complete, it is okay, you have just lost data, but if you do this fast and this later, then what? You have written some block number into your inode, but that block number, that block has garbage.

That is even more of a bigger problem. Therefore in general, it is good to first write data and then add all the pointers to that data. When you know, your data is safe on disk, then at a pointer to it in your inode and things like that. But this alone is not enough to guarantee consistency, because you could have cases where there are multiple metadata blocks, then what?

In this case of appending to a file, I have to change the inode and I have to change the bitmap, marking a certain data block as occupied. So there are two metadata changes, which one do I do first? So therefore, just this principle of writing data before metadata is not enough. We need something more.

And therefore, we have tools like file system checking tools. In Linux, it is called fsck. So whenever you boot up after a failure, if there is some inconsistency in these metadata blocks, if some metadata changes have been written and some have not been written, if there are inconsistencies, then this file system checking tools will run through all the data structures in a file system at boot up and try to identify these inconsistencies.

For example, if you find that the bitmap marks a certain data block as free, but that data block is present in the inode of a file, that should not happen. If you have allocated a block for a file and you have stored its block number in inode, then that block should be marked as occupied in the bitmap. Otherwise somebody else may use that block and overwrite your contents, we do not want that.

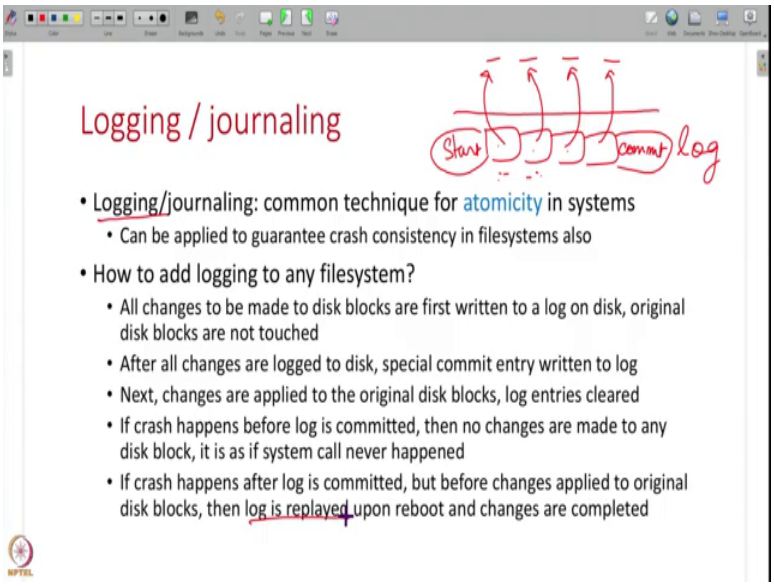
Therefore, all such inconsistencies, you will check. These file system checking tools will look at all the bitmaps. If a block is occupied, is it there in some inode. If it is not there in some inode, mark it as free. All of these checks will be done by these file system checking tools. Whatever invariants you want to guarantee on your file system will be checked.

But this is, of course, it takes a long time to do these changes. And it is, it may miss some of the changes, some changes might still be lost. So this is not the only way, especially if you are very particular about the consistency of your data, like a banking system that is very important for you to keep data safe, then you might do some other techniques beyond just file system checking tools.

So the fundamental problem here is what we want is this property called atomicity that we have discussed in an earlier lecture. What is atomicity? If I am making multiple changes in one transaction or in one system call, I want either all the changes to happen or a few if there is a failure, it is okay if all changes are also lost.

But I do not want this case where some changes happen and some changes do not happen. Remember the example, you do not want an e-commerce website to charge your credit card and then forget to ship you the product, that is not right. So what we want is atomicity, either do everything or do not do anything at all. So how do we guarantee atomicity?

(Refer Slide Time: 26:29)




A common technique across computer systems that we will see in many places to guarantee atomicity is the concept of logging. And this can also be applied to filesystems. So what do you, what does this concept of logging mean? Suppose you are changing multiple blocks on disk, the data block, the inode block, the bitmap block, you are changing multiple blocks on disk.

So instead of directly changing these one by one and have the problem of power failing in between, what you will do is you will write all of these changed blocks into a log, into a special area of disk called the log. Block one, block two, block three, all of these changes that are have to be made atomically are first written to a log.

The original disk locations are not touched. And after you store all these changes, then you will write a special commit entry saying, okay, this is all. This is the start and this is the end of all my changes. You will make note of this on disk. And then, once this is done, then you will start making the changes one by one. You will update the inode, you will update the actual bitmap, everything you will update. And once all these changes are done, then you will clear out this entry.

So what has this logging achieved? You are writing it two times. You might say, okay, what is the big deal? You are just writing the same thing two times. Here is what it achieves. Suppose a crash has happened before the log has been committed, you only wrote these two, three blocks

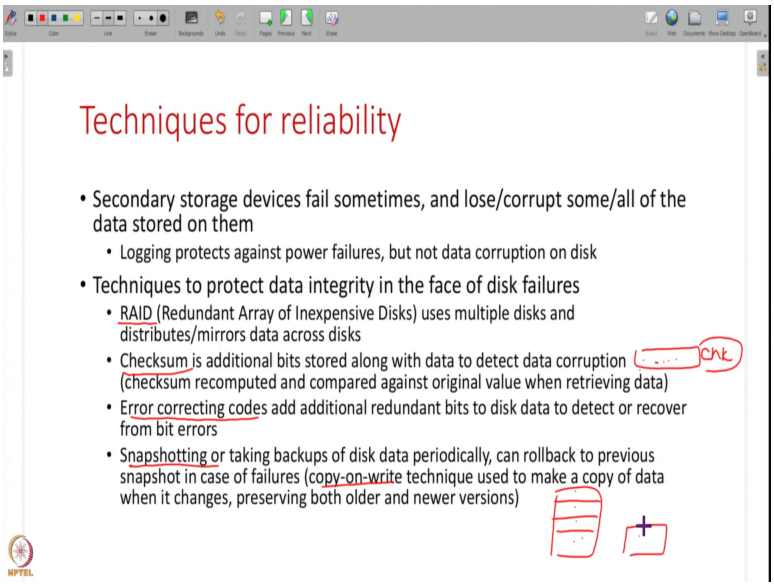
and the crash has happened, then when you restart, there is no commit entry. So you do not know, are these the full set of changes or not? You do not know. Therefore you will not go back and touch these actual disk locations. Then no changes will be made.

On the other hand of the crash happens after the log is committed, then you have a record of all the changes, you remember all the changes you have to do. And then after a crash, when you restart it, you will go back and apply all of these changes. You will modify the actual inode, bitmap everything you will correct modify. You will, this is called replaying the log, so that all the changes are done atomically. So either you do all changes or you do not do anything at all. So this logging is a good way to guarantee atomicity.

So in general, there are many reasons why your hard disks, your secondary storage devices can fail and they can lose some of the data. So logging is of course one way to protect against power failures, but there are also other reasons, like the data on the disk itself can get corrupted due to various reasons. You have written a bit 1 after some time, due to age, the hard disk might make it into bit 0. We cannot guarantee that things will stay forever.

So therefore, we need techniques to protect the integrity of the data in the case of failures. So logging is one way to protect against power failures, but there are also other techniques that modern filesystems employ in order to guarantee data integrity.

(Refer Slide Time: 29:24)

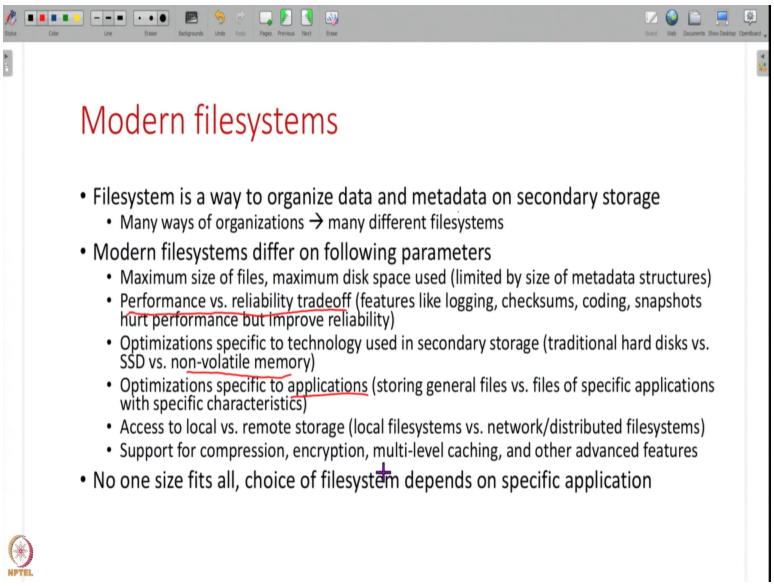


For example, you have the concept of RAID disks, which is you use multiple disks, not just one hard disk, but use, mirror your data across multiple disks. So that even if one copy is corrupted, the other copy can stay. You also add things like checksum a set of bits, and then you add some checksum at the end of it some extra bits that you compute as a function of these bits. Why? Because if you are data is corrupted, later point of time, you recompute the checksum, it will not match. Suppose you are adding all these bits and storing the value here. If some bit changes, then the sum will not match in the future. Therefore, this will help you detect data corruption.

The other thing you can do is, for every set of bits, you can add a few what are called error correcting bits, some extra bits for redundancy to your data on disk. So that even if some bits get corrupted due to hardware failures, you can still recover them using these extra error correcting codes. All of these are ideas that are widely used in computer systems. Then the other thing that you can do is you can also do snapshotting, that is periodically you take a backup of your entire disk data. So that if something bad happens, you can roll back to the previous copy.

So this is done using by a technique that is called copy-on-write, that is you make a copy of all your disk blocks, and in the future, if you are changing any of them, then you make a, when you are writing a block, you make another copy, so that you have all your previous versions and you have the new version. You never overwrite, you always make a copy when you have to modify a block. This copy on write will ensure that you are preserving the older versions and the newer versions, and therefore, you have snapshots available two days back, what was my file system, I can recover it for you.

(Refer Slide Time: 31:09)




So therefore different filesystems, modern filesystems they differ on all of these parameters, they differ on how they organize your data and metadata on storage, what is your maximum file size, what is the disk space used, what features are you using for reliability? And not that there is this trade-off, there is no one correct answer for how you build a file system. There is always a trade off.

If you do all of these things for reliability, like logging or checksums or snapshots, all of these are good for reliability, but they hurt your performance. In logging, you are writing everything to the disk twice, which is wasteful. But if you need reliability, that is a tradeoff you have to make.

Similarly, there are different filesystems that are optimized for different storage technologies. You have traditional hard disks, you have solid state devices, you have newer technologies like non-volatile memory. For each of these different technology, your design decisions in your file system will be different. You know, you might say, I might want to use a different way of organizing my data on disk for different technologies. Similarly, specific to different applications.

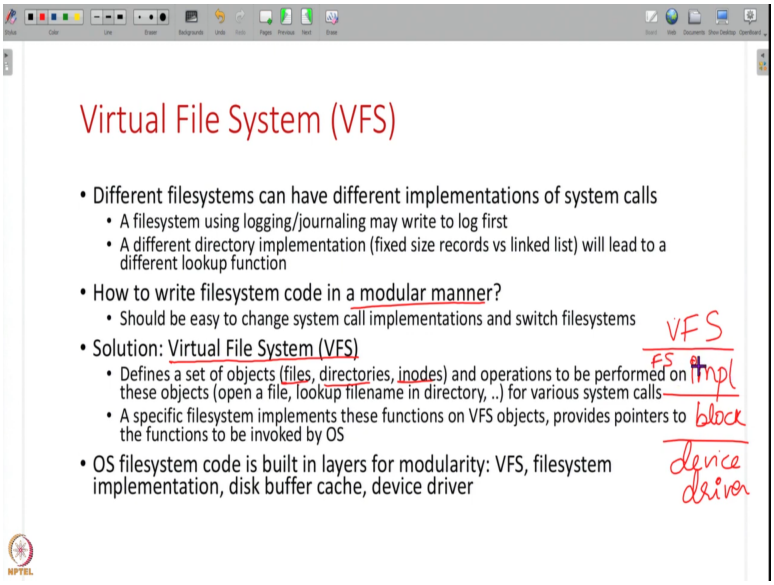
I know the ideas we have seen are for general purpose filesystems, but suppose your files have a specific structures, you are storing only a certain kind of files, files of only a certain size, then you can optimize your file system data structures for this particular workload. So there are

different files systems that are optimized like that. Then there are different filesystems based on are you accessing a local disk, a remote disk over the network, then features like compression, encryption more than one levels of caching.

So there are all sorts of features available. This is an active area of research building different filesystems for different kinds of applications and workloads, for analytics workloads, for machine learning workloads. This is a very active area of research. And therefore there are many different filesystems and there is no one particular most optimal file system out there.

If you are designing a computer system, you have to make this choice, understand what are all the options available for you and pick one of them based on your requirements, your applications, your technology, your performance and reliability tradeoffs and so on.

(Refer Slide Time: 33:28)




So now, if there are all of these different filesystems, which have their different implementations, for example, if your write system call implementation will depend. If you are using logging, you will first write to the log and then make changes later, otherwise you will directly make changes. So your implementation of your file system will be very different for these different options.

So then how do operating systems support multiple filesystems? Is it that after delete like a million lines of code in the OS in order to move to a different file system? No. So today

filesystems are built in a very modular manner in modern operating system, so that you can quickly change from one file system to the other.

And how is that done? That is done with an idea that is called virtual file system or VFS. That is, the operating system, the virtual file system in the operating system defines some high level concepts, some objects like files directories, inodes, every file system will have this. So the operating system code is written in terms of operations on these files directories and inodes. For example, to open a file, open the root inode, then read the data blocks of the root inode, then open this directory, find out this inode number, that is how you will write your code for a system call.

Now, the underlying file system can implement this look up a directory or open an inode or open a file, all of these operations can be implemented differently by different filesystems. A file system that implements a directory as a records or as a link list or as tree. Depending on that, the traversing or looking up a file will be different. But the concept will be the same.

So what the OS code, the way it is written is, first, these system calls are implemented in a virtual file system on some abstract objects, like files, inodes and everything. And then the actual implementation of the file system will provide pointers to all of these data structures, the functions, function that will look up a file name in a directory, all of those function pointers, they are implemented by the actual file system.

So that a lot of your file system code does not have to change, only these function pointers have to change. If you want to move to a different file system, you provide a different set of function pointers to your VFS, and you are good, but you are not having to change a whole lot of code in your OS.

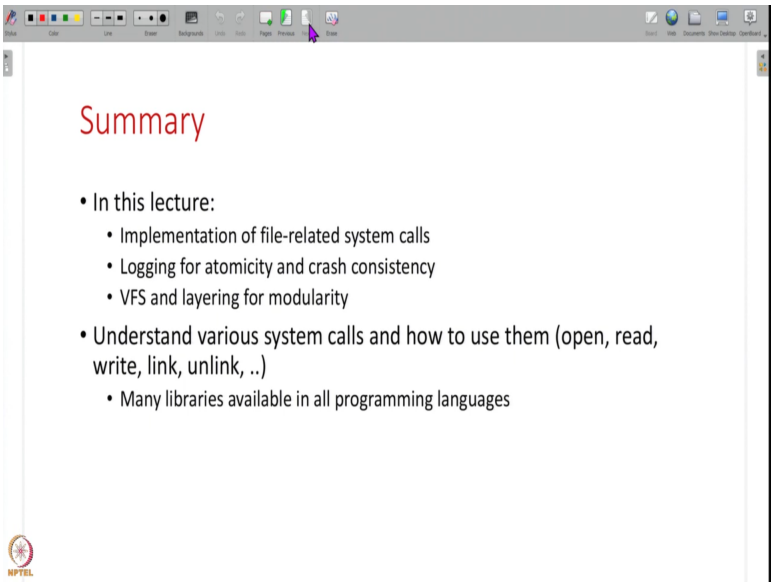
So in this way, this is the concept of layering. We have seen this in the second lecture of the course. Instead of building everything in as one big block, you build it in layers. There is the VFS layer, then there is the file system implementation layer. Then underneath this, there is the disk buffer cache, where blocks are cached. And then there is the device driver.

Each one has their job. The device driver only deals with reading and writing data from the hard disk. The disk buffer cache, the block layer only deals with storing the recently accessed disk

blocks. Then the file system implementation layer will only implement functions on these different objects, and the VFS layer will look at actual system call implementation in terms of these objects.

So everybody has their job cut out, so that you can easily move to a different disk buffer cache implementation, a different device driver, a different file system, all of these switches are easy to make.

(Refer Slide Time: 36:47)




So that is the end of our discussion on filesystems and operating systems. We have seen how some of the file system, system calls are implemented. We have seen how you can guarantee things like automicity and crash consistency, what are the various different ideas that are being used in filesystems today, and how the choice of your file system depends on your underlying application. And you have a choice of filesystems, you can switch easily using mechanisms like VFS that modern operating systems provide.

So with this, I would like to wrap this lecture up. And just to help you understand the concepts of this lecture better, I would like you to try out some of these system calls like open, read, write, link, unlink play around with them, write some code using these system calls, and not just, directly these system calls, but there are also many libraries available in different programming

languages for file access. So understand them in order to assimilate the concepts of this lecture better. So thank you all that is all I have for today. See you in the next class. Thanks.