CIS3110 - Lecture 10 - Revised Transcription

Announcements

All right, we'll get started right on time today. So I guess, well, one announcement that the CSA asked me to give is to remind you that the Central Student Association is doing its elections right now. Apparently, they close tomorrow. Inform yourself. Choose who you want to have as your student association reps. I assume they all have websites. This is not the ancient world. So you can go on the Student Association website and find that.

One other announcement about the recordings: I am recording today. As I said I would do, I got a lot of emails on Wednesday. Some of them seemingly quite irate that I had not been able to post the recording of Tuesday's lecture before Wednesday morning. There are manual steps in this process that require me to have time to do them. So I will post the recording, but it isn't going to be instant after the end of the lecture. You have to wait for me to have time to actually do that and post it. So it will most likely be up about 24 hours after the lecture, because that's when I will have time. So just don't panic when it is not there immediately after the lecture, it will come. It will not be that long.

File Systems Structure (Continuing Discussion)

(Topics: File-System Structure (14.1, 14.2, 14.3, 14.4, 14.5, 14.6), File-System Internals (15.5, 15.6, 15.8))

So returning to our discussion about file systems. I got a couple of questions either right after class or via e-mail. So I just want to clarify a couple of things here.

Fundamentally, a disk is just a data structure. It's a list, it's an array. It's an array of sectors, so even though it's a 3-dimensional structure, we can think of them as being in this logical order where we can start from cylinder 0, head 0, sector 0 and move to cylinder 0, head 0, sector 1, sector 2, etcetera, and we generally impose this logical order that you do all of the sectors in a given track. And then you move to another head in the same cylinder and you do all the ones in that track.

And so even though it is a three-dimensional structure, we can think of it as logically one big linear list. So we can think about Block 0, block one, block two, block 3, etcetera.

So let's say there were 8 sectors per track. Then the first four would be the first half of the first track. The next 4 would be the second-half of the first track. The 3rd 4 would be the first half of the second track and so on and so forth. But even though it's 3-dimensional, we can think about it as we've mapped it logically to this one-dimensional object.

If we can think of it as a linear sequence or similar to an array (but not in memory) of sectors, that means we can always find the first one. So as far as a file system structure is going to be concerned, everything is going to depend on you being able to find the beginning of this sequence of blocks. And figure out what's happening because you know what is happening at the beginning.

So here we have our boot block. We know the very next thing in the FAT setup is the indexing information. Then we have the root directory. Then we have the data blocks, so we know the order, we know in the FAT description how big these all are. So therefore we can figure out where things are in the disk.

Every file system depends on you finding the information at the beginning, reading that block. That block gives you enough information to figure out where the other pieces must be. OK, so there's no chicken and egg issue because we literally look at location 000 for the information that describes the rest of the device.

FAT vs. Unix File Systems

(Topics: File-System Structure (14.1, 14.2), File-System Internals (15.5, 15.6))

When we talked about the FAT organization, it was a linked list. It has all the problems of a linked list. You can only go forward through the list. It's not efficient to go backward. It's not efficient to jump to different locations.

So the Unix solution is to say, well, let's not use a linked list, we're going to use this inode based tree. First on the block, first on the file system organization is the Unix super block. And when I say Unix, I should make sure that it is clear that I mean Unix and all the things derived from Unix. So Mac OS, Linux, FreeBSD, BRBOS, all of them. They all have this structure or something derived from this structure.

So the first thing is the Super block. Much like the FAT master boot record plan, it's a data structure that tells us everything else we need to know so you can see the number of cylinders, heads, sectors, etcetera. You can see it tells you where the inode is for the file system root. That's because the file system is itself a tree.

And as we'll find out, the file system is a tree and the files are indexed as a tree. So if we think about that tree, I'll come back to the inode itself in a moment. But the key thing about the inode is that along with things like the file size and the time, we have a list of the first few data blocks.

inode Structure and File Indexing

(Topics: File-System Structure (14.3, 14.4), File-System Internals (15.5))

And as we expand to the biggest file we have, we essentially have a list of some data blocks, a small tree of some more data blocks, a medium sized tree of even more data blocks and then a really big tree of the rest of the data blocks. Now when I say tree, note that I don't mean binary tree.

A binary tree, which I'm sure you've spent some time talking about in data structures, is a tree where the maximum number of children from a given node is 2. So they either have no children, one child or two children. This tree has more children. But otherwise it still follows all the properties that you would have of a tree.

So if we look at the third indirect tree, the root of the tree is the entry in the inode in this third indirect block.

And so this is simply a block address in the file system, but it's an address the same way a memory address is an address. It tells us where the information is, but rather than it being in RAM, it's just somewhere else on the disk.

If I go back to the FAT example, you can see we were talking about addresses here, so block 8 literally means data block #8. And every time we're chasing through the FAT table we went from, in this case, data Block 8 to data Block 3, which is simply an address. It's a logical address in terms of the block index in this list. But that is fundamentally not very different than the logical address of a particular byte in RAM.

It's still just an index into an array structure, so when you have a memory address in RAM, we're just talking about how far away you are from the bottom of your process image memory map in bytes. When you're talking about an address on the disk, we're simply talking about how far away you are from the beginning of your data blocks in blocks.

So each piece of memory, if you want to think about it that way, is bigger, you get a whole block at a time, whereas in RAM you can talk about one byte or maybe one word. Here we're talking about at least one sector, so every block is made of some integer number of sectors. One or more sectors.

So when we're talking about an address here in this tree, we're simply talking about where a particular block is on disk. So one thing that I did get a question about, this is not talking about data Block 0 in terms of the integer block number on the disk. The zero is simply talking about the first block of your file.

Where that gets stored on the disk is literally the number that you're going to store in this little slot. So if you choose that you've got a small file, let's say you only need one data block, we're going to find an empty spot on the disk. We'll talk about that in just a second. You're going to find some empty spot on the disk. Disks are pretty big, so let's say it's block 16,254,955,012. So you put that number in here. It's still the first block of your file. It just happened to come from that location, the same way as when you call malloc and it gives you some bytes from somewhere in memory. The address that malloc gives you back tells you where the data is. You can use it because it has been set aside for you. But if that is the first, if you store that in the first position in some array, it's still in that position.

So these are simply talking about logically as you move through the file, which slot here identifies which logical position on the disk. When we start moving into the tree, notice the six here is simply one greater than the five here, because I'm simply saying for logical Block 0, the beginning of the file, I record it here. Then the next one goes here, the next one goes here, and so on until I exhaust the memory of the little table that is part of the inode.

And then I say I'm going to make a tree for the rest, so that using this single location here, I can talk about a whole pile of blocks. Because each of my tree nodes can hold in this example 1024 of entries like this. So if I can put 1024 entries in this data structure which is just stored in a block on the disk, then I can now say I represented the first 6 here and so from index 6 starting with 0-based indexing up to 1029 they all fit in this next single data block based list.

The tree, whether it's the first tree, the second tree or the third tree still follows that tree property. We've got essentially a pointer to the root. That gives us a data structure which is a node containing 1024 children. So it's just you can think of it as an array of integers. Your 1024 integers talk about 1024 somethings that are directly linked out of that block.

If you're talking about the first tree, those somethings are simply 1024 data blocks of your file. So literally the data that you have stored in the file. That you want to read after you finish reading this data block indexed here will be in the block indexed here. I have left the data blocks themselves off of this diagram, not only because I didn't want to draw a billion squares, but because it would be an enormous confusing diagram. So the red arrows are meant to say this is pointing at the actual data block over there, but I didn't draw the data block.

Multi-level Indexing and File Size Support

(Topics: File-System Structure (14.3, 14.4), File-System Internals (15.6))

Let's say that we just did this first indirect strategy. OK, so using one pointer from our inode we can get up to 1029 data blocks. Let's say for the sake of simplicity, they're 1K. So the data block is 1K, we can have 1029 of them. Then we can have a total file of one MB in size basically.

If we just copy that strategy in the second and third location, how big of a file could we talk about? 3 megabytes. That's not a very big file in a modern operating system. So if we could only talk about 3 megabyte files, no one would be very interested in this file system.

So what we need is a way of talking about bigger files. So therefore, instead of 1024 entries that we can fit in one block, by having this 2nd order tree, by having a tree where we've got a block dedicated to pointing at those index blocks, those indirect blocks, we can fit roughly 1,000,000 more blocks in our indexing structure, yet only take one pointer at the top level.

That's a pretty good return because this one pointer points at this list of 1024 pointers, each of which points at another list of 1024 pointers. Right. So we've got 1024^2 entries in this part of the list. Which gets us up into a few gig. Which is starting to sound reasonable, but there are lots of files bigger than that.

So therefore they say, OK, if that's not enough, then we can come here and using our third and last entry in the inode we can go to a tree that is 3 levels deep. That contains billions of possible references to data blocks. OK, so this is so that we can have individual files that are on the order of terabytes in size which maybe you need to do. Maybe you're working at a movie studio and you want to make another Toy Story movie that's going to be a tremendously big file.

So in order to talk about all of the locations in the file, we need to stitch all those data blocks together so that we can find them. So that's what we've done here. We've made a tree so that using only a tiny amount of indexing information at the top, we can potentially talk about all these different data block locations.

The fact is, how do you know how big the tree is? The answer to that is that it is always done in this way. So the first few blocks of your file are directly linked from the inode. And one of the key things to understand here is that this is a conscious choice made by looking at what size files people really have on their computers.

How many big files do you have? Maybe a couple. Right. You might have a file that is a GB or two in size. You probably have as individual students, you probably have very few that are on the order of hundreds of gigabytes in size, but we want you to be able to have one if you need it. But the likelihood is very low. But how many small files do you have? How many files do you have that are like one or two kilobytes? Turns out there's a whole lot.

So they intentionally said, let's make this more efficient for the small files that are the files that everybody has in abundance. Because you probably have a couple tens of thousands of files under about 10K on your machine. So all of those can be served with this direct mapped list that is already in the inode, so as soon as you've loaded the inode, you already have access to the locations of the first few file blocks. All those small files - we're done. You know exactly where all the file data are.

File Allocation and Growth

(Topics: File-System Structure (14.5, 14.6), File-System Internals (15.5))

So for blocks in the indirect blocks, they are not allocated until they're needed. For a zero-length file, like `touch foo.txt`, you need to record the name. You need to know it is 0 length, has permissions, doesn't have any data. So it gets an inode. And no data blocks. All of these are set to null.

You write 1 byte into that file, we're going to need to allocate a data block from somewhere so we find an empty data block. We record its ID here. We now have enough room to put one block worth of bytes in that first block. So if we just have a 1-byte file, no problem. First byte used, rest of them we don't care what's in them. We record the file size here in the inode. We're done.

I'm trying to show you here is that these 3 green arrows point at these three top level blocks and then the reason I put this green arrow super close to the first one and then a big gap is cause I'm trying to show you that I drew the first of the 1024 entries which goes here and the second of the 1024 entries which goes here but I should have probably numbered them zero and one or something here and then I'm showing you the last of the 1024 entries that goes here.

So the reason I magically skipped from 3077 up to 1,000,000 and change is because that's how many data blocks you can record in the 1021 entries in between the second one and then the last one. And then I'm doing the same thing here with these ones, which is how I'm going for 2 million in change up to suddenly I'm in the billions.

So what if we need the data block that comes after this data block, right? OK, so let's say we have a file that requires this data block and it's completely full and then we write one more byte. At that point, this entry, which previously was null, we need to fill in with a data block that we allocate to form the top of the tree. And then we fill the first entry there with the address of another data block that we access to fill the second level of the tree, and then we allocate yet a third block to be this node of the tree and then the first entry in that block becomes the entry that points at the actual file data containing block for your new byte.

So as we grow, the amount of work required to add another block to the file varies dramatically. For the first fix, every time we walk off the end of one data block and we write an additional byte, all we need to do is allocate a new data block for that file data for the file contents.

Once we hit the last entry in this block, the block marked 5 here, then for the next byte we need to allocate not only the data block that is going to contain the file data, but we need to allocate this block as well to point at it. For the block after that, we're back to only having to allocate file blocks, blocks that are actually going to contain file data until we walk off this, in which case we then have to allocate the root of this tree and one of its nodes and the data block that it points to. OK, so as the file grows, the trees get created. But only as much of the tree gets created as is required to actually reference that data block.

If you need a 1030th block, then you're going to need not only the block to put the data in, but you're going to need a place to put the entry that refers to that block. Right, that is this node of the tree. It's a leaf node. Because it is a two-level tree. We then also need a third data block to hold the reference that points at that leaf node.

So when we walk from 5 to 6, we suddenly need a double allocation. We're going to allocate this indexing block and the block of the file. When we walk from 1029 to 1030, we have a triple allocation. There's the block of the file, there's this leaf node, there's this intermediate or root node in this case, and then we stitch them together. We then are good for another 1023 entries being filled up by just allocating file blocks and putting them on there.

Then when we walk off this, we'll need the data block for the file, we'll need this indirect first indirect block here. And then we can make our existing tree root entry number one the second entry point at this indirect block.

OK, so we've got direct links to data blocks here. Single indirect block here with 1024 links to data blocks in it. Single second indirect block pointer here pointing at a doubly indirect block full of 1024 entries of the locations of potential first indirect block that can point at the underlying data. And then if we work over here, it's just one more layer of the tree.

Disk Address Size

(Topics: File-System Structure (14.4), File-System Internals (15.5, 15.6))

The question is, how do you know how big one of these disk addresses is? This has changed over time. So early Unixes, 64 bits was plenty. But now that you can buy a disk in the terabytes, they've increased the size of the maximum number for the number of bytes in a file. And also the maximum number for the number of bytes that you could therefore potentially have in some of these other indexing values.

But when you format the disks, different operating systems will do different clever things here. When you format the disk, the disk isn't going to change size, right? It's a hardware object that you bought at a store. You're not going to like install more platters or stretch out the thing to make it bigger. So fundamentally a given disk has a size. You know what the maximum number is of discrete sectors that you're going to talk about. And therefore, you know what the biggest number is that you're going to have to talk about to identify the furthest away sector.

So some operating systems have been clever, and for a given file system, they'll actually allocate a differing number of bytes to the offset size, so that if you're formatting a 30 terabyte drive, they'll make the numbers bigger because they have to talk about the whole drive, but if you're formatting a tiny drive, they'll say why would I waste space talking about part of the disk that physically isn't there and can't possibly exist in this file system because I bought a 1 gigabyte drive or maybe I have a floppy disk and you don't want to start talking about 128 bit offsets on something that only has a few thousand total sectors to begin with.

File Holes

(Topics: File-System Structure (14.5, 14.6), File-System Internals (15.6, 15.8))

Unusual, but now very popular addition to this strategy. If we only need to allocate the pieces of the tree that are actually going to refer to real data in our file, we have the opportunity to do something that at first seems very strange.

If you open a file descriptor for a new file on a Unix type platform, or indeed on an NTFS formatted disk which is more or less the same structure (I'll talk about what's different with that when we summarize), but if you open that and let's say you seek to offset 1,000,000, you haven't written anything yet, you just say I want to be logically a million bytes away from the logical beginning of the file. And then you write a byte. Say you write capital Z. How much of the disk do you actually need in order to represent what has been stored?

I mean, at the end of the day, we only wrote one byte. It seems silly that we would have to allocate all of the blocks that we didn't write. And they don't. So they will allocate a data block to store your capital Z at whatever logical offset it is. And they will install it in this structure at the position for the logical offset.

So let's say that's at position three. That just means you got a couple of null pointers here, and then you've got your data block and you're good. But maybe you didn't seek to a million. Maybe you seeked to a couple billion. So maybe the only data block is, let's say this one. Then everything else is null except this entry which points at this indirect block, and one entry in it which points at this indirect block, and one entry in it which points to this indirect block, which points to the block with your data in it.

So the question becomes, what logically is in the file? If you were to save it, reopen it and start reading bytes from the beginning, it just gives you nulls. So anything - we call this a file hole. If you have a file with a hole in it, then it just logically has nulls for any bytes that would be sitting in a data block that you didn't bother to allocate because no one bothered to write to it.

Now, if you do actually write a null, let's say if we think about these two options: Option A - Open the file, seek to location 1,000,000, write 1 byte whose value is null. Option B - For a million and one locations, starting at the beginning of the file, write a byte whose value is null.

In option A, we only store the one data block plus whatever indexing information was required to track that data block. In Option B, we've actually allocated every single one of those bytes to a place on disk. You literally have a million and one bytes stored on the disk that are null. When you read the file, you can't tell the difference.

The only way you'd be able to know the difference is if you can actually look at the indexing information direct so utilities like tar and rsync and other higher level utilities for this kind for file work, they know how to do that. So if you tar a file that is in fact mostly made of holes, it's not going to generate blocks of data to store in your tar file for the holes, it'll just say there's a hole, it's this big. And then the data that comes after it is what's in the next block of the tar file.

Directory Structure and Inodes

(Topics: File-System Interface (13.1, 13.2, 13.3, 13.4), File-System Structure (14.1, 14.2, 14.3))

The incoming arrow is identifying the block that is the indirect block. So you could think of it as it is effectively pointing at the 1st byte of the indirect block. If it was in memory, right? So because this is an entity coming off the disk as this is, let's say, disk block 45. Right. Then the entry here would be a 45. We'd go and load block 45. It would have our 1024 possible entries in it, and we would iterate through those entries and if they were non-null then we can skip along to the next one which again is just going to be some block in the disk. Maybe it's like this is block 1,000,002.

That's based on the offset in the file. So if you open a file, let's say for reading, and you ask read for some information, you're at offset 0 in the file to start with so that means you want the first byte of the entry in the first block, so it'll go here, it'll say load up the data block that is in that it whose address is stored in location 0, and what is the first byte in that that data block?

If you repeat 1024 times, you will then see every byte in that block, and then you will have skipped to the first byte in the next block. So when you open a file, part of the file handle information that it has to keep track of is the file offset.

So when you perform a read, from the top as a user program, you perform a read. Right. Maybe you call scanf. It fundamentally calls read, traps into the kernel, the file descriptor points into a table of information on how to process the file. One of the pieces of information is what is the current logical offset we're at in that file.

So if you say I want to read 35 bytes, read traps into the kernel. The implementation in the kernel says look up the file descriptor that has been passed to read, it's file descriptor, I don't know, five. That says we're at offset, I don't know, 78. So then it's going to say from 78 plus what I said, 35 bytes. That's a range that is going to have to be loaded from one or more data blocks.

So let's say it just does it in a loop, so it will figure out byte 78, what is the max in this inode base tree to figure out which block it is, so that will be because we got at least 512 bytes per sector, so we know byte 78 has to be in the first one.

Now let's say it's 78 billion. If you just seek to location 78 billion, or if you have called read for 78 billion bytes because you're moving through, I don't know a movie that you're playing or something. And you do a read again if it needs to give you the byte at offset 78 billion in the file, it's going to go to the inode. It's going to figure out based on the formatted block size on the disk which block logically 78 billion falls into.

Let's say it's a 1024 byte block so that basically turns our 78 billion into 77 million and change. So then it's going to say, OK, so 77 million and change, where is that block, so it's going to come down here into somewhere in this gap. So it's going to say, OK, 77 million and change is bigger than 1,000,000. So I have to go into my indirect third indirect list. Chase down to the right leaf node here, which block services the request at offset 77 million.

So these numbers here, although they're in data block size format to turn into byte format, all we need to know is how big did we make the block size on our file system. That's just an argument we passed to format.

From what you've said, my understanding of this is that essentially, there are no pointers from one block to the next block. The pointers from one block to the next indirect block, they only come from down through the tree. We figure out how to go to the next block by doing the math.

So don't imagine you were reading the file in a completely random order. When you open the file and you say read, it's going to give you the next byte based on your current offset. That is the conventional way of accessing the file.

But let us say that instead of that, we have a program and you're just going to read all the bytes in a totally random way. They're going to figure out how big the file is, generate a random number, say that's the byte they want. You figure out where the byte is by simply doing the math to figure out which tree and where in the tree to go.

So you could simply name an offset in the file, any offset from the beginning of the file up to the maximum possible file length. And you simply take that number, divide it by the block size, and now you know which of these little red numbers you're looking for.

These numbers are just in sequential order because the leaf nodes of all of the trees are essentially just a continuation of this list. So you can, if you had them all lined up in one long strip, it would just be an array. You could simply say OK I need block 3077, I just go 3077 entries away from the beginning of my list.

The only difference is that we've said I directly store the beginning of the list in the inode, because I'm going to need it a lot because people like having small files. And then if the list is longer than that, I just have basically like a crib sheet where I've said, OK, if it didn't sit here, I just have an extra part of the list. And I've said here is where my extra part of the list is. So just skip over here and look through this as though it was a continuation of this list.

And then if that isn't enough, we've said, well, I'm going to need a whole booklet of crib sheets. So I have an entry to this double indirect block, whose entries simply say where is there an indirect block that just continues my list.

So to go from one byte location to another byte location in the file, we start at the top every time and just say take my offset in the file, divide it by my block size, which data block number do I want? If it is bigger than what is here, I need to go into one of these trees.

I know how big the trees are because I know the first tree can only hold 1024 entries. I know the second tree can only hold 1024^2 entries. And then I know the third tree can only hold 1024 to the power of 3 entries. So if my offset is bigger than that then I know it's physically impossible to have it in my file system and I return an error.

If it's smaller than that number, then you're just saying which tree is it in. So then as you move through the file reading byte by byte, the code is going to say, well, is the next byte in the same block as my previous byte, in which case I have that block in memory, presumably because I've already loaded it in. Why would I do a whole bunch of ridiculous math just to find something that I already have?

But then as soon as you walk off the end of a data block that you have in memory because you're reading it, then you're going to find the next one by just coming to this tree, exactly as if I started naming random offsets and we worked them out each time.

OK, so you don't have a link from Block 6 into Block 7 or Block 1029 into Block 30 etcetera. There is no linked list to tell you how to get to the next piece. The FAT system, you only have a linked list. This system you only have a tree.

File System Tree Structure

(Topics: File-System Interface (13.2, 13.3, 13.4), File-System Structure (14.1, 14.2))

The data blocks are at the ends of the red arrows, simply telling you there is a data block over there, and what the logical order of that block is.

All of the rectangles that have Red Arrows in logically form one massive list of all the possible data blocks that could be in the file from zero up to one million 74,791,421. So this is saying if you wanted to have a file that had, with 1024 data block size, you're going to have one trillion possible bytes in the file.

So then to keep track of those one trillion bytes, you're going to need a billion data blocks. This is simply the structure to record where those billion data blocks are. OK, so all I've done is I've said I need a list of a billion entries of where the data blocks are going to be.

I've taken the first part of my list and said I'm putting it directly in the inode. I've taken the next part of the list and said I'm just going to link this little table in with one pointer from the inode, and that's because small files are super common. Slightly bigger files are still quite common.

And then I'm going to say if I need a really tremendously big file, which we know is not very likely, then I'm going to have a lot of work to do. Because to keep the list going, I'm going to need to allocate a whole tree with these internal nodes so that I'm going to say to keep track of this list, all 1 billion entries I've got here at the tail of the list, I'm going to need a million of these indirect blocks to keep track of the million indirect blocks, I'm going to need 1000 double indirect blocks to keep track of the bottom layer and then to keep track of the 1000 doubly indirect blocks, I can fit them in a single triply indirect block.

And because there's a single, triply indirect block, I can keep track of where that is with a single number that I put in this entry in the inode. OK, so all this is just to keep track of this ongoing list where I have the Red Arrows pointing at the actual data blocks.

We don't need this in FAT. We don't have a tree in FAT because in FAT we just have a singly linked list. And we set aside essentially a table for all of this stuff at the beginning of the disk. And that table only contains integers.

OK, so in fact, we've essentially taken all the indexing information for all the files and put it in one global table, whereas here we've said individual files are going to contain their own indexing information. We still need the indexing information somehow. We need to know how to find a particular block of a particular file, so that if you say to me I want Block 4 of file, I don't know ABC.c, that I know where to find that block. In the FAT world, you start at the beginning and you chase through a linked list until you get to location 4 and this one you simply have this long list stored in a tree indexed out of the inode.

Directories as Files

(Topics: File-System Interface (13.2, 13.3, 13.4), File-System Structure (14.2, 14.3))

We had a tree on the last slide that tells us how to find all of the blocks of a file. This slide is a tree describing the named entities in your file system. So if you think about any file system, we have this idea of a root directory.

So if you're on Windows, you have C:\ (C colon, backslash). Every disk has its own private root directory. You can add more disks. You can have D:, E:, Z:\ (backslash), whatever it is.

In the Unix world, we have one root directory. The root directory is simply information stored in a little tiny file. And the data structure of that information is simply tuples (ordered pairs really) of an inode number and a name.

So you can say to find file A, go to inode 3. To find file BC, go to inode 5. To find file "file", go to inode 8. So this is a tiny file simply containing information about entry names and the inode number in the file system. It is a file as we say, of type "directory".

When you did an `ls -l`, you've probably noticed that in the first column, if it's a file, you have a minus sign. If it's a directory, you have a little 'd'. That's telling you that that entry has Type directory, which means that if you read it, it is expected to contain these ordered pairs.

So A is a file, but it turns out it is also a file of type directory. So when you go to inode 3 and you walk through the data associated with inode 3, reading the entries from that file, we get more ordered pairs. We get an entry called J, it's stored in inode 7, it turns out to also be of type file. We have an entry called KM, it's at inode 11, it's not a directory, it's an actual file.

So the only difference is that the file type is an ordered list of bytes with whatever structure you want, you just open it and whatever you put through write gets stored in the file. An entry of Type directory is expected to be these ordered pairs.

In a Unix file system, they do have a limit for the maximum number of bytes you're allowed in an entry. But otherwise you can have variable length entries. They do prevent you from having an entry with no name whatsoever, because it would be very difficult to talk about such an entry in your file system to move it or delete it, so they require you to give it at least one byte. And most of them stop at 255. But you can have any number of bytes in between.

So whatever is in the file will be in the data blocks. If you read those entries so here file, let's say test.f, we'd have to go to inode 26. We load inode 26 and then we're chasing through this kind of structure.

So these inode numbers are all you need in order to read the data from the file. But notice that unlike in the FAT example, the name is stored separately. So the name and the file data live in different places.

In FAT, we had a directory entry that had the name and all of the other information that we now see in the inode, so we had the directory entry that had the 8.3 name and it had the file size and it had the first block ID and the permissions and all that sort of stuff. But the inode has all of the things except the name. It also has this thing called link count that I said I was going to come back to.

In the directory J in directory A, test.f has inode 26 and it's going to point at all the data blocks making up test.f. Inode 7, the data blocks associated with inode 7 contain this information. Inode 7 refers to a tiny file that only has this information. It will have inode numbers and then the entries "a.c" and "test.f".

So back to the names. So we can have a name, it can be a short name, it can be a long name. They're just in this format, so you read the inode ID, which is a fixed size integer field. And then you read some bytes for the name. And if you have lots of short names, then the actual file defining your directory can be smaller than if you have a bunch of long names.

Whereas in DOS they decided you have a standard name format and they're all the same. Which means they can have a table in DOS where you just index in a particular place. So you could just jump to entry 7 and read the entry at that location. Here we would have to start at the beginning of the file and parse the file as we go.

Hard Links and Link Count

(Topics: File-System Interface (13.2, 13.3, 13.4), File-System Internals (15.5, 15.8))

Now, on the inode notice I just talked about the link count a second ago. So if you want to move a file, if you want to rename the file, what a Unix system does is it opens the directory in which you want to put the new name, adds the new name to that directory with the inode number of the old file. It then increments the link count because you've got 2 entries. Then it goes to the directory of the old name and it removes it and then decrements the link count.

So in the middle of a move which takes almost no time because none of the data blocks actually need to move, all we're doing is writing enough bytes for the new name plus an inode, and then updating the old name to remove it from that list. So it's a very cheap operation. But in the middle we've got a link count of two.

There are some advantages to this - if someone yanks the power cord out of the wall halfway through the move, we haven't lost our file. We've in fact got two entries to the file, which is a lot better than losing your file.

OK, because you can do that, they have this idea that you can just create more links to a file, so if you have a file on a given file system, you can put more entries for that file in directories in various places if you like, that all refer to the same inode. Every time you add a new entry, it's incrementing the link count. We call this a hard link.

This can be handy if you've got a bunch of people who need one giant data file and the data file is the same, so you can put them in all of their home directories, let's say, and then I know people who aren't your friends can't come and see them because it's not in some public place. You've got your own private link to the one shared thing.

So if I, let's say, wanted to add test.f to this directory, then I would say find this directory, open the directory, write test.f, add 26 because it's the same inode here. The link count, which is in the inode structure, this integer would be updated.

So if you have a hard link, multiple directory entries pointing at literally the same inode, the inode defines the whole tree, so therefore it's pointing at the same file.

The numbers in each case are inodes. I have not drawn any hard links here, so therefore all the numbers are unique. When we talk about an entry of Type directory, that is simply an entry attached to an inode as everything is, whose contents are these ordered pairs where the entries are an inode number and a string that is a name.

So this 9 is telling us that this small set of data, this structure here is stored in a file that we've given directory type and because we've given it directory type, it is referenced from some other directory here because this entry which says E (it seems to have slid down a little bit) has inode 9.

So anything that you think of as a directory, the data in it that tells you what is in a directory is simply the list of the names of the directory entries paired with their inode numbers, stored as a very tiny file. You could open it just using like file open and start reading it a byte at a time and you'll see the names appear, and then you'll see the integer information for the inode.

Hard Links vs. Soft Links

(Topics: File-System Interface (13.4), File-System Internals (15.8))

The difference between hard links and Windows shortcuts: Let's first talk about what Unix calls a soft link. A soft link is a different way of referencing something else on the file system. The way a soft link is implemented is it is again a tiny little file with type "soft link" where the content is one string of some other directory entry. It simply says "go over there." When you open me, I actually want you to go over to that place. It's kind of like a pointer stored as a name on the file system.

A soft link is very much like a Windows shortcut, which is essentially a name of something else plus some permission information. So a soft link and a Windows shortcut is a tiny file saying "I'd rather be over there." It's fragile because there's nothing to prevent you from deleting the thing that the little file talks about in either case, so you can have a shortcut that points at something that no longer exists. You can have a soft link that points at something that no longer exists.

A hard link, you won't have that problem because if you try to remove the thing that the hard link is attached to, it simply decrements the link count. The advantage of a soft link or a Windows shortcut is that you can refer to things on other disks. Because if you've got 2 disks in your computer, they're both going to have an inode 0 and an inode 1 and an inode 2.

So if you want to talk about something on the other disk, you can't just say that's the thing that got inode 35. You have to say that's the thing over there on that other file system and the method for doing that is that you simply refer to it by the directory entry base name. Because it may not even be the same type of file system.

I'm sure you're all aware you can have a dual boot Linux-Windows system. Your Linux machine can mount the Windows disk. It doesn't have the same format, so you can't start talking about inodes on an NTFS disk. It's a different strategy, but you can talk about the name because all the file systems have this name-based strategy where from the root directory we work down into subdirectories and eventually get to our file.

Link Count and File Operations

(Topics: File-System Internals (15.5, 15.8))

Back to link count. If you open a file, we increment the link count. So if you open a file and your program is slowly reading the file, let's say you've got a loop and you put it to sleep for 10 seconds after every byte, you're walking slowly through the file. With the link count strategy, we could delete the file from the disk in another window, but the link count will still be positive because we opened it and incremented the link count, so we can still be reading through the file. We don't actually delete the file until the link count drops to 0. So if you decrement the link count and the result is a zero, we're going to release all the data from the file. We take all those blocks and put them back as free blocks in the file system.

This is a pretty cool strategy because it means you can't screw up a program that has a file open by removing the file. Which is very important if we think about our paging strategy where we're using the program file, the executable program file as the source for the pages for the text segment of a running program.

Because if we could delete the file while it was running and then some page fault said give me back one of the pages of the text segment and the answer was "oh sorry it's not there anymore," then your program can't run, it's just dead. We'd have to kill it.

But with the link count strategy, even if we remove the program while it's running, we don't deallocate the blocks until the program closes and the link count goes to 0. That's pretty cool.

Another thing we can do - you may have noticed that when you update your OS, they asked you to reboot after you've done the updates. And one of the reasons for this is that you can take really important files like the kernel and remove them and put a new kernel in there with the same name. And the running kernel is still attached to the version that now has no name. But it can get all of its pages from there. It can keep running until you reboot.

So you actually don't release the data blocks making up any of your system files that are running until you reboot the computer. Which means you can replace system files in place while the programs are running without crashing your computer. But if they're going to go and, like, read another configuration or some library or something and you've changed it, they will get the new version when you open it by name.

A hard link is very much like, let's say you have in RAM, you've got some big structure, right? You allocated your structure with malloc, you filled it with some important information. You've got a reference, a pointer to the structure. A hard link is equivalent to having a second pointer to the same structure. The structure is the same. The only thing that is duplicated is the address which you've stored in now two different pointer locations.

This has nothing to do with user versus system. This is simply saying if you make a file system hard link, all we're saying is that we literally have two named entries on the same file system where they have the same inode. Right. So if we have KM at 11 and let's say ABC also said 11 then ABC and KM would be the same file as you go to the same inode and as soon as you're starting at the same inode, obviously you're going to get the same data because you're walking through the same list.

What I'm saying is that the hard link is simply a file system entry where we've got two things that talk about the same inode. What I'm additionally saying is that anything else that opens the file, opens an inode, a file with the same inode, increments the link count. Right.

So if you open it for reading, you open it for writing, you open it to run a program, whatever reason, you have to open the file. If you've called open, it's going to increment the link count. When you close it or when you remove an entry from the file system directory information, it decrements the link count. The actual blocks, so this whole big structure, this only gets deallocated when the link count goes to 0.

So that's why if you have one entry for something in the file system and a program opens that file, incrementing the link count to 2, if someone comes along and removes the file system entry, you still have this structure. It just has no name anymore. There's no name in the inode. So you've got this nameless file. But it's still a perfectly valid file. It's still got all the data blocks allocated, so if you've got a way of finding it, you can move around in the file.

So if you've already opened it, obviously you can find it because you've got a file handle in your code. So then until you close it and the link count goes to 0, it will be there for you. You can do whatever you want. You can read it, write it, etcetera. But then if you close it, if someone else had somehow got a hold of it and incremented the link count as well, it would only be deallocated when they close it too.

If you remove the file system entry (you type rm and the name of something in your file), it decrements the link count. If you have opened the file for any reason, so for reading or let's say it's a program, and you're running it - if you run a program, obviously you need to open the file to get the data from the file into memory.

So if the file is open, for every time you've got an active file handle (file descriptor-based handle) to the file, we have incremented the link count. So let's say we're running a program. Maybe it's the kernel, maybe it's a three.o or a three.exe - whatever you want to call it. Whatever it is we're running the program. Because it's running, we've opened it. It therefore has a link count 1 greater than it had when it was just sitting there as a file on disk.

If it had only one entry, then the link count it used to have was one. Right. So you can compile your code, you get an executable, it has link count 1 because you've got one entry in the file system talking about this file.

OK, so we run the program. We load it into memory, we increment the link count. The link count is now 2, so it's running. Let's say it is a lot of work to do, runs for hours doing whatever it's doing, right. Maybe it's just an infinite loop. We don't care. But as it's running, if it has paging operations that require us to bring back in pages that were in the text segment, we need it to still be there.

That's why we have the file descriptor telling us where the file is so that we can access that data. So let's say it's running and you come along and you decide to delete the executable from your file system. `rm myfile.exe`. Running the delete decrements the link count. So what's going to happen is our link count of our running file will have gone from 2 to 1. And that's all that happens.

Until the running program exits, at which time we close the file descriptor, the handle we have talking to this file. When we close a file descriptor that is referring to a file, we also decrement the link count to undo the increment we did when we opened. Regardless of why the link count is being decremented, if the link count becomes 0 at that time, we deallocate everything in this entire tree. So all the data blocks, any indirect blocks, and the inode all get returned as unused resources that we can then use again for new files.

Directories and Their Structure

(Topics: File-System Interface (13.2, 13.3), File-System Structure (14.2, 14.3))

A directory is a file the same way any other data file is. As part of its file mode, it has a bit that says this file is for directory information. It is a directory. That's the only difference in terms of the storage in the actual file system.

So if we have a program running, remember that as we're going through our pages, right, we've got a limited number of page frames available. We may push out a page frame or a page from memory to make the frame available because we need to put a different page. Our paging strategy says if the page we've ejected is a text page that came from the text segment, we know we didn't need to save that page because we have a read-only copy of it available in the executable file.

So if we later need that code again, we're going to simply recover it from the same executable file that we originally loaded when we ran the program in the first place. So the problem is if we didn't have link counts or something like it, then we would try to go get that from the file, but the file doesn't exist anymore and so then our program would have to crash.

If you got an old enough version of Windows, we'll talk about this when we're summarizing the course, but Windows was originally a program that you ran on DOS. And the whole family up to Windows ME were just further glorifications of the one on DOS. And Microsoft eventually realized that there was a lot of bad ideas in there, including this one, because things crashed all the time. And this is one of the reasons why.

And so they said, OK, we got to redo everything from scratch. We need a plan that is much more like the Linux, the Unix plan. They moved to a tree-based index scheme, they adopted a whole lot of things from the Unix world. They called it "New Technology" or Windows NT. And all of the subsequent versions of Windows have been glorifications of the Windows NT strategy. But the new Windows NT file system (NTFS), which is what you now format your Windows disk with, is also a tree-based strategy not quite like this. We're out of time for today. So we'll talk about what the difference is for tomorrow.

But the last thing I wanted to talk about on this slide is that this explains why there's no name for the root of a file system. So if you think about it, on your Linux machines, the root we say is called slash. But it's not really, that is the delimiter between names. The root actually has no name. That's because the way we identify it is just with an inode.

Once we have an inode, we can read the value in the file. We know which inode because it was listed in the Super block, so that first block on the disk that we read in, we say where is the inode containing the information for the root directory. That lets us find this directory and read the entries that are in it, so the root really has no name. But it does have contents.

Everything else has to have a name because the only way we're going to find anything else on the file system is to say start from the root directory and then work our way down, going subdirectory to subdirectory until we find the entries for the name we want and then we can say OK what inode is that and then we load the file.

Final thought here to avoid resulting nonsense: they prevent you from making a hard link to a directory. Because otherwise you would have the ability to make a tree that links back to its own parents, and you could go down forever and they recognize that this is probably a bad idea. So you're allowed to make a hard link to an entry that is not of Type directory, but you can't make a hard link to a directory type entry to avoid loops in what is meant to be an acyclic directed graph, otherwise known as a tree, of all the names on a given file system.

OK, so let's leave it there for tonight. And we will come back and wrap up the file system discussion next day.