CIS3110 Lecture 11 - File Systems (March 18, 2025)

All right, off we go again. Welcome everybody. OK, so last week we were talking about file systems and we specifically were wrapping up talking about the Unix file system design. So on this slide, just to be sure everybody's clear this tree structure is talking about the tree of directory entries that you're familiar with when you use a file browser. So when you open up, if you're on a Mac Finder, if you're on Windows File Explorer, it gives you a little tree-based structure where you click on directories and they expand and you're going down when you go into different directories and you're going up as you go back toward the root of the file system. Yes, it is being recorded.

Up on this diagram is the same "up" as people talk about when they say go up a directory. If you go up far enough, you get to the root directory. That is the one at the top of this diagram. It is the first directory read on your disk. So in both FAT-based file systems, NTFS file systems and Unix file systems, the boot block or super block is primarily interested in helping you find the root directory, and everything else is linked from there. It simply goes down.

In the Unix world, each such directory is simply a small data file containing the names and the inode numbers associated with the names. In the DOS world or in the FAT file system based world, you will remember that the root directory was in a specific place on the disk. And it holds an array of directory entries. So the format is different. There was a question over here. As opposed to what?

OK, so let's talk about that. What is the cost of this strategy? Because this is effectively the other option. Instead of having a root directory that is simply identified by inode number, we have a root directory that is identified by location. What are the pros and cons? Anyone. Is there any downside to having this strategy? Yeah.

So we're not talking about the block linking, we're just talking about the directory plan. OK, the block linking is a separate idea and I want to make sure you don't get the two confused. So all we're talking about here is that to find the root directory, it's in a fixed location on the disk. So you just go to that location and start reading. Those are the directory entries. What does that imply? Yeah.

"You might have to store it."

What?

"So you do have to store the one location in the boot block."

If your root directory is stored in what is effectively a file, that is true. Which is less space than storing the full array of potential directory entries in this one. When you format the disk, you have to decide what is the maximum number of entries that can be in the root directory. And you set aside all that space even if

you have no entries in the root directory. So here we've got a fixed size array allocation of directory structures. Here we've got a tiny list stored as a file.

Now below the root directory, they're actually the same because FAT and NTFS and UFS and every other file system stores a subdirectory as a small file containing the directory information. So the only difference is what's in that file. In the Unix example, all we store in the file that tells you what is in a given directory is simply name paired with inode. In the inode, we have all of the important things other than the name describing the file.

In the FAT and NTFS world, where is my entry here? There we go. In the FAT and NTFS world, in the directory entry, along with the name, we have all those other things, the attributes, the size, etcetera. So Unix has decided that the name is separate and all of the other attributes of the file are part of the file. We call this the metadata. Whereas the name is just a thing listed somewhere else in the directory.

In both cases, you start in a root directory. You can go down. You can go back up with an entry called dot dot. They actually both follow the pattern that there literally is an entry in your directory whose name is dot dot that tells you what the parent directory location is. There's also an entry called dot. What does that dot do? It's yourself. The way to find your own entry, given your own location in Unix. The parent directory of the root directory. So the parent of this one is simply also itself. Windows does that also, but they don't store it in the directory for some reason that is unclear to me.

OK.

So the the small file with additional directory entries called the subdirectory plan is common. Now, what is the advantage which I think is what you were really trying to ask. Of this more complicated plan than the DOS plan. Because there is a big advantage, there's a few. One of them was mentioned earlier, I think.

Thinking about this instead of the directory entry plan. I mean, fundamentally, what is bad about a linked list? They can only serve moving forward. You can't back up. You can't randomly seek. These are very expensive operations. This you can randomly seek in as easy as you can do anything else. Why is this asymmetrical? This is probably the only time in your career you'll see an intentionally asymmetrical tree. What is the asymmetry trying to serve?

"Ease of access, but in what way?"

"Data you need more often."

We're making quickly linked without having to recurse down into these sub structures. Fundamentally, we've decided that small files are more likely to be accessed than big files, so therefore we've optimized this because this is the same structure regardless how big the file is.

For small files, you probably only need an inode. You don't need any of this other structure. There's your data structure. You get a little bit bigger, you need one tree, so you have to fill this one indirect block and then as many data blocks as you need. Bigger than that? Well, now we're starting to spend some change. We need a doubly indirect block, which can list up to, in this example, 1024 indirect blocks.

Someone asked the other day how we figure out how many indirect blocks there could be. It will depend on how big you made the block size on your file system. Because every time you load one of these, you want as much payback for that cost as you can get. So if your block size is 8K, you're going to have 8K worth of addresses in there, because why would you not? If you have tremendously huge files, then you may need to build a big tree. But we don't have tremendously huge files very often. So Unix is specifically designed to make smaller files more efficient and to allow random or otherwise non-forward linear going access to the files work well.

This is such an improvement over the FAT plan that Windows after the NT release, the NTFS also uses a tree. Uses a slightly different tree. Uses a kind of tree called a B-tree. So a B-tree, much like a binary tree, requires balancing. One of the interesting things about the Unix file system tree is that it does not require balancing. You only fill in the parts of the tree that you need, but there's never anything like a rotation or other balancing operation. All you do is you just allocate the pieces you need on the path from the inode to your data block, and the rest of them are just not there if you don't need them. So there's never a balancing cost because there's no balancing.

NTFS uses a B-tree, which is effectively what you would get if you took a binary tree, but had thousands of children in each node because it follows the same strategy that each node is a disk block and the disk block is entirely filled with references. The B-tree that NTFS uses is simply the list of all of the data blocks that you actually have in your file, so it also supports holes. If for some reason the first block is at location 10 million, then the lowest address in your tree is 10 million. Other than that, it's basically the same plan.

So Unix, they're baking into the design this optimization for small files. Windows, they've decided to use the same algorithm for all files regardless what size and expect the tree growth balancing algorithm to take care of the differential speed. So we've looked at three of them. One of them is FAT. It has the simplicity of being simple, as was pointed out, the diagram is much simpler. It's a linked list. Along with that comes all the costs of the linked list. But if your file is small enough, Microsoft still uses FAT. They assume you can put enough of it in memory in the cache that you don't need to worry about the fact you need to back up and start again because you're just skipping around inside of a cache.

Assignment Four, I've given you an assignment of literally reading FAT format file system. That is real Microsoft compliant code that you'll be writing, so you've got some Microsoft compliant test images to play with. So I hope you've all looked at Assignment 4. If you haven't looked at Assignment 4, we'll talk about it in a little more detail next day when you've all had a chance to do that, but that is literally what

you're doing is you're going to fill in how to parse the data and the root directory in this kind of format. Question. Yeah, let's wait until everyone has a chance to read it for next day.

OK, so one of the key things that I want you to keep significantly separated in your brain is this tree, which is effectively on every file system you're ever going to use from any vendor. You're going to use the tree of directory entries versus this tree, which is the Unix-based tree to keep track of where your blocks are. Also, in this structure and in the FAT structure, while we have indexing information which may come in the form of these arrows chasing through the FAT table, or may come in the form of these arrows chasing through indirect blocks, the whole point of all of the indexing is just to keep track of the data blocks that are in sequentially increasing order logically within your file that hold your actual data, because at the end of the day, why would you have a file other than to hold someone else's data that they're putting in, right? The whole point of the file is so that some user can put arbitrary data in the file and you, the operating system designer can keep track of where their data is to give it back to them when they request it. So our job is to build the indexing so that we can find the data that is in the file when they say I need the data at block offset 8092. Question.

"The number, like its location in memory."

So the difference if this was a FAT-based directory tree structure then instead of ordered pairs of only inode and name, it's a whole tuple that says name 8 character part, name 3 character part, file size, file attribute, file owner, etcetera, the whole set of attribute information is listed directly in the directory. So each directory entry is much bigger. Right. This one in fact is variable. If you have a 1 character long file name then the size of the entry is simply the size of an inode number, the bytes that are your file name, so that would be 1 byte and a null to tell you that it has come to an end. And then you're done, that whole entry. So the directory data is much smaller here, but we still have the same need to store things like who owns the file and what the permissions are. The only difference is that in this case those are in the inode. In the Microsoft case, in both NTFS and in FAT, those are in the directory entry. Same information, different places.

Well, I say same - Microsoft doesn't have as many attributes. You can't have - you've probably noticed on a Linux system, you've got user, group and other read-write-execute. Microsoft just has a read-only bit. FAT has everybody or nobody. They're just simpler. You'll see that in Assignment 4, actually, because you're going to get to play with that directly.

OK, so far so good.

"That's not leading balancing. Does that mean that there's no...?"

NTFS needs that. OK, so NTFS, if you think about a binary tree that instead of two children, let's say to mirror this example, let's say your block could hold 1024 children. OK, so it's the equivalent of a binary tree if each node could have 1024 children. So just like a binary tree, if you start adding in an asymmetric

way, it has to rebalance itself. The B-tree balancing algorithm is not quite the same as the binary tree balancing algorithm, because you have all those children. But same idea - as you grow the tree, it makes sure it's balanced. If you start removing chunks of the file, it will stay balanced.

Typically with files we don't have to worry about deleting part of the file, and you may not even know that's possible. You can truncate a file at a given location, but normally people grow files or they delete the whole thing, so there's very little deletion-based balancing, which is one of the reasons Microsoft kind of likes that. Because the B-trees are quite efficient if you don't need to delete. So because people almost never delete half of a file, they don't worry about that. Deletion is the whole deal, so they just say all of these blocks go back.

"So does the B-tree not use the small file optimization that the inodes deal with like...?"

It does not, no. So the code is there for a slightly simpler because it doesn't need to have that special case. But they do lose a little bit on the performance. OK. Any other questions?

"Native directory entries of fixed size... the fixed..."

"The root directory in a Unix file system is simply a variable link list..."

OK, so the NTFS, also FAT and NTFS, they have this idea that there's an array that tells you what your root directory is like. And below that they do that "we'll just store things in a file" strategy. But they both have this idea that by default when you format your disk, you're going to have 1024 slots for things in the root directory, so if you try to add 1025 files to root directory of a Windows machine, you will find that you are out of space. Even though your drive might be nearly empty, it will tell you I don't have space for this additional 1025th file because it has to go into a fixed size array.

So one minor difference is that you could do that on a Unix machine because you could just keep growing the file, right? If the file keeps growing, as I was in a meeting with one of my grad students today, it turns out that you can then have ridiculously large directories. So she's struggling because a piece of software that she used created a directory with 480 million entries in it. And it's taking forever for her to get the things out of there because these aren't sorted lists. This is just the list that you have to traverse from beginning to end and search if you want to find anything, which is fine if you have a couple dozen entries like you normally have in a directory. But when you have millions of entries, it's going to take a while. So I'm sure that was an issue that no one really foresaw when designing the software that she used that did this problem.

OK, so far so good. So one other entry or one other difference between FAT, where remember, we marked something free by just flagging an entry in the FAT table. So if you want to find a place to put something on the disk, you just search through the FAT table till you start finding things marked free. Unix and NTFS share a strategy where what they do is they say, well, I've got these blocks that I'm not using.

That's why I have empty blocks because I have blocks that I'm not using, so by definition I could use them for something else.

What they do is they use it to list a kind of ragged linked list of where the empty blocks are so they don't have to search for them by searching through a table, they just pull them off the list. So all they do is they say I got a linked list of blocks made of otherwise unused data blocks. And in those I just have a table of some additional data blocks so that in the Super block, it's got a list of let's say 50 immediately available unused blocks and a link to another unused block that has a list of 50 immediately available unused blocks. So that they every time they traverse the list they get another batch of 50 so they have to do a traversal for every single one. They get a whole batch at a time.

At the head of the list, the array of additional blocks may not be full because you may not have an evenly divisible by 51 number of blocks available on the disk. As you run through this consuming all the blocks, you can allocate all the blocks that came out of this as soon as you copy the table into the root block. So therefore you can use this block you copied it out of. And then at the end, when you allocate the last ones that you had pointed to and your free zero location is null, then if the user tries to write additional data, you have to tell them this disk is full.

But one of the things we're doing here is every seek, every location that gives us a new block gives us a bunch of additional data for free. So that we don't have to chase through this FAT table thing, we're getting blocks of free values at a time. It's a tunable parameter to say how big should the allocation space be in the Super block? So how many of these do you get at a time? You can play around with these parameters as much as you like. It doesn't make that huge amount of difference, but we do get this bulk shopping every time you go for one more block in this linked list of piles of free blocks.

This idea of a free list is very common. You'll see it outside of file systems also, but one of the things to think about is every time we go to a new block, that's our expensive operation, right? We have to reposition the disk head. We have to read data from somewhere else on the disk.

OK, so early in the Unix timeline, Marshall McKusick looked at the file system for Unix, so its big competitor at this point was DOS, right? The FAT file system and he said OK, given different choices, how big should we make our data blocks and how much performance improvement can we get? So we looked at different things.

So option one, no waste. We just put our data directly on the disk. Consecutive allocation, no space between files, better not have any new data later because you can't change the size of any of the files. So functionally useless, but useful to see what space allocation that would take. It takes more space if you're going to do data, but round up to a block, a disk block boundary, 4.2% waste. If you do the original Unix file system strategy, you're up at 6.9% waste and you can get up to, if you go to 4096 bytes, so only 4K in your block, get up to 45% waste.

Why is that relevant? Think about what Windows is doing with their block size. They've got cluster sizes of 64 kilobytes, and you've only got a drive size of four gig. 256 kilobytes for a drive of 16 gig. At 4 kilobytes, we're seeing half the drive is wasted with empty space. People have said several times Microsoft must be in cahoots with the drive manufacturers because you buy your drive and it's immediately full because they're doing massive blocks.

OK, McKusick said well, clearly we want big blocks for speed, but we can't tolerate that wasted space. So what they decided to do is what they call the fragment block plan. So they recognize that the issue is only at the end of the file. All of the blocks in the middle of the file by definition are completely full. It's only the end of the file where you have this problem, but because we've convinced ourselves that the small files are the most common files, we get to the end of the file really quickly in a small file. If all of our files were multiple terabytes in size, we wouldn't care.

So what they do is they say, OK, I've got a little additional piece that needs to go on the end of this file and I've put it in this block. But I've got a bunch of extra space. So I've got another file, it required 2 full data blocks and then I got a little chunk leftover. I'll add it with an offset so there's a little table at the beginning telling you for linked file 0 started the first byte, for linked file 1 start after the first one ends and so on. Very tiny files might be only inside the fragment block. So they may totally share their entire life with other files. And then there's extra space at the end.

If you need to write more data to the file, well, you're going to yank it out of here, allocate a new block and start growing so some of these may disappear. You've got kind of the same issue you had with assignment one. Then you've got chunks of memory being reallocated and freed, but you're only worrying about it inside of one logical disk block. And you can potentially tag together the loose ends of a bunch of files into one place, and you can get up to 97% disk usage here, which I think everybody would agree is a lot better than 50. You do have a little bit of extra calculation when you're reading the bytes at the very end of the file, but it is only when you're in the fragment block at the end of the file. So they've essentially sopped up all the wasted space and put all the pieces that were in those partial blocks together, yeah.

"You can. Yeah."

Doesn't matter where they put the little table, so there's a maximum number of fragments you can put in one of these blocks because the table is a fixed size, but that's specific to that block, so not really a big deal. You're going to have some bytes wasted in the block anyway, most likely. OK. So by taking all of our ragged pieces and putting them together, we get tremendous space savings.

"Give us style blocks. You said that three blocks put into a...form of list and each time you...accessing part of that linked list that have multiple blocks, yes, so."

If you took all of your empty blocks, you could easily make a singly linked list where an entry in the first block simply says there's another block over there, etcetera, etcetera, etcetera. When you needed to

block, you would just take the one from the head of the list update the head to the next. Doing that would require you to do a disk seek every single time you wanted a new block.

So all we're doing here is we're saying in my - if you want to think of it as a linked list node because it is actually already a disk blocked data structure, it has a defined size, so it's going to be plenty big to store more than just the next items, right? So you've got the index of what is the next block in your linked list, and then they basically just have an array of a bunch of other blocks that you can have right now.

So all they do is when they make this, the first block will simply be in the list pointed out of the Super block with the 50th block will need to have its own entry. We copy the table into that block. It has then the 49 blocks that we moved out of the Super block and then we've got 49 more spaces to add there. So every time you have to iterate through the linked list, you get essentially 50 block IDs at the same time for the cost of one seek. That's what I'm trying to say.

"Has 50 blocks of free memory."

Yes, each element of the linked list is a block that has in it an array of 49 additional blocks that are available. OK. All right.

So then further file system analysis. Looking at the speed because as soon as you add some complexity like this fragment idea, it could potentially be worse for speed. But in reality, it actually improved things dramatically. Even at a 4096 block size got even better with an 8K block size, because remember, we can essentially arbitrarily increase the block size now and not worry about waste because we've got our fragment strategy. So we're going to fill that waste with fragments and use it as efficiently as possible. Modern Linux systems typically have a 16 kilobyte block size. So you get even more bang for your buck. Every time you do a disk read, you're getting 16 kilobytes of data and you're tagging the ends all together into a fragment block.

Further tricks. So if moving that arm of the disk head is our biggest cost, then we can organize the blocks on the disk in order to try to reduce that as much as possible. So one thing we can do is use what we call an extent. An extent says all of the blocks of the file are physically stored in adjacent locations on the disk.

So you'll start, let's say in a given cylinder for block zero of your file and then you'll make sure that Block 1 is the next block. Block 2 is the block after that. It fills the entire track. Then it will go to the other head, fill the next track, third head, as you work your way down to the cylinder. Once the cylinder is full, you're going to have to go to the next cylinder, but you only have to move the head the smallest possible amount to go to the very next cylinder.

If you can have an extent of physically contiguous disk blocks on the disk, you'll get your best bandwidth out of the disk reading data off of the disk surface. Databases are built on this principle. When you build a database, it spends a lot of time organizing its tabular data into extents so that any of the tables,

especially the ones that are frequently used, you're going to organize so that everything is physically optimized on the disks to avoid this kind of seek.

But the problem is what happens if you need to add more blocks to that file? Because if you've organized everything in extents, you probably have another extent that starts right after the previous one ended, so then you're going to have to skip somewhere else. DOS, Windows and to some extent still NTFS handle this problem by doing what they call defragging, defragmenting the disk. They reorder the disk blocks to try to build extents for all of your disks. So in the days when people used that as their boot file system, they would regularly do this. NTFS tries to do this in the background, but you can still greatly improve the performance of an NTFS file system by running a defragmentation operation on it, and it will try to sort your disk into some kind of order so that the physical blocks and the logical blocks are in the same sequence. But it's super expensive and to get it really effectively done has to be user initiated and it will degrade over time because as soon as you do any kind of addition or any kind of additional writing to the disk or deletion, it'll upset your whole extent plan.

So another way that most of the other operating systems use, not just Unix, but pretty much anyone you name who hasn't come out of Microsoft is they use cylinder groups. So they say locally on a neighborhood on a disk, let's create neighborhoods. So maybe we'll have clusters of 16 cylinders at a time. So we'll say it is better if you can't be the very next block, it's better to be in a nearby cylinder because that will reduce the time you need to move the head.

So what you do is you say when you're allocating new blocks for a file, and it could be either the data blocks or the metadata blocks, right? The ones making up your tree or inodes - it's better if they're from the same cylinder group. So the operating system will attempt to keep all of the blocks in the same cylinder group, which is a much, much looser constraint than trying to keep them in an extent. But it gets you much of the same benefits. You're still local. You don't have long seeks, but you don't have to worry about the fact that it's going to degrade over time, and you don't have to worry about the fact that you're going to need to reorder things to keep your extents together.

So the way you do this - if you're growing the data associated with an inode. OK, so a file is being appended to, some inode is going to have more data associated with it. Well, what you do is you allocate the data inside of the same cylinder group for a while. But you don't want to allocate all of the data blocks in your cylinder group or you're going to end up having a bunch of empty inodes in your cylinder group with no data blocks that can be attached to them because you've consumed them all.

So you allocate within the cylinder group for a while, and ideally in physical contiguous order. But then if you've got too many, then you're going to jump to a totally different cylinder group every few megabytes. So you're expecting that as you move through the file, you're going to occasionally take the hits to move to a totally different part of the disk. So you're not going to over saturate one particular neighborhood with the demands for a single file. You're going to spread it out across the disk, so that way other files,

remembering that most of them are small, can probably put all their data near their inode, because when they came here, they allocated within the cylinder group and then didn't have anymore allocation to do so, they're done. That means the inode describing the file and the data in the file are physically close to each other. That translates into speed.

When you want to make a new inode - so this means that you're creating a file, right, as soon as you're making a file, the first thing you need is a place to put the metadata for the file. So that means you allocate an inode for the new file. If the file is of type regular file, we're going to put it in the same cylinder group that the directory is in. But if we're making a new inode that is a directory, knowing that it's going to want to be in the same cylinder group as all of its children, we go to a different cylinder group that has lots of free inodes and not too many directories already.

So each of the cylinder groups will have a little bit of metadata information tracking things like how many directories are in it, how much of it is used, the free lists, etcetera. So then when you're trying to find additional inodes or additional blocks within that cylinder group, you can pull it off of the lists defining things for that group and know that you're talking about that neighborhood, yeah.

"No, no, no. No, no, no, no."

No, no, I'm glad you asked. I want to make sure that's clear. OK, one disk block is an integer collection of sectors, so a sector in a block may be the same. As you increase the disk block size, it's always a consistent number of sectors. So if you have a 4K disk block, you have 8 sectors. They're contiguous. OK, so it's a contiguous set of disk surface in 512 byte segments, because that's a sector - that's the smallest addressable elements we have on a disk. So a block is simply a physically contiguous, logically contiguous collection of sectors where the block size is consistent for the whole disk. Once you have formatted the disk, you say my block size is X, every block is now that size. So sectors and blocks relate.

But a cylinder is all of the sectors and therefore all of the blocks made of the sectors that are accessible without moving the disk heads. So you're going to have a set of sectors on the top of the topmost platter, a set of sectors on the bottom of the topmost platter, a set of sectors on the top of the next platter, bottom of the next platter, etc. So these are usually for a modern disk on the order of some tens to hundreds of megabytes. So therefore many blocks, and commensurately many many sectors.

OK, so if if you've got your ext3 format or ext4, sorry it is now formatted Linux machine, you've probably got a block size of 16K which means you've got 32 sectors per block. But in a cylinder you might have, let's say 3 and 1/2 gig. So all of the blocks making up that 3 1/2 gig cylindrical shape are all in that cylinder, and then when that's full, you go to the next cylinder and there's another 3 1/2 gig because the same number of sectors are in each cylinder, regardless how close to the center or the outside it is, they're just physically wider.

But the disk is spinning, so they take the same amount of time to go by under the disk. But if you were to look at a microscope, you'd see the bytes or the bits rather stretched out more as you went near the periphery, yeah.

"To a track. Tells you."

Which head you're reading from so track is vertical. Right when you're changing track, you're going up or down. When you're changing cylinder, you're going in or out. When you're going sector, you're going around. So it's a 3-dimensional address scheme. How close to the hub am I? How close to the top of the disk am I? How far around in the rotational cycle am I? Yeah.

There's one head per side of the of a platter. So by layer, if you mean one side of one physical platter, then yes, so because they're actually physically disks of metal, they have a top and a bottom. And there is normally a head above and below. Some drive manufacturers will take ones they've screwed up and stick them on the end and have an odd number of heads, which is why sometimes you buy a drive and it has like 9 heads and you're like, why didn't they use the other side of that platter? It's because their manufacturing is a bit crap and they decided they wanted to be able to sell the ones that had some defects. Yeah.

So if you walk down the platters of this disk, we have track 0 on the top, track 1 on the bottom, track 2 on the top of the next one, track 3 on the bottom of the next one, right. So each pancake in the file has a read and write head above and below it. So as you're writing your data, you're going to lay it out on a given track. When that track is full, you're going to skip to the next track. So we'd write the one on the bottom of this platter, and then we'd continue our way down. Yes, yes.

The more tracks you have on each platter, and similarly the vendors will describe it as the more cylinders you have. Right. Because you're going to have simply, as you move out from the hub, counting the cylinders on the way out. There's one track on the top platter, but you get all the data vertically in the same cylindrical slice as that without moving the head, which is why they count cylinders.

Another well - we're talking about disk vendors - another little cheat they always do to you. So as computer scientists needing to store data, 1024 is our number. Because our values are base 2 multiples. But drive vendors will sell you things in base 10,000, which is one of the reasons you never get anywhere near as much space in the file system when you format it as you think you were going to get.

So if you buy, let's say a 2TB drive, they do not mean 1024 raised to the appropriate exponent, they mean 1000 raised to that exponent. So for every terabyte you're basically losing 5 bytes out of every sector or 12 bytes out of every sector you thought you had, so therefore they can put a big number on the disk. This is a 16 terabyte disk and you format it and you say why did I only get 14 terabytes out of it. That's because they're not talking about the same terabytes. They're talking about the made-up base 10 terabytes that

only drive manufacturers use. So like a car manufacturer was allowed to use a different kilometer to tell you that it got better gas mileage, I don't know why we let them do this, but they all do that. OK.

So good question. I'm super glad to know that was a confusion. OK, where were we? So allocation then. Growing the file, we want to stay, ideally with the rest of the file, so we're going to stay in the cylinder group, but we don't want to eat all the resources of the cylinder group. So if we're a big file, which we know are unpopular out of fashion anyway, we're going to take the hit of striping it across the disk, putting chunks of it there as we go.

Small files, we know they're the bread and butter of the system, so we want them to be able to benefit by being up with their inode. Therefore, if you want to make a new file, we can have a rule that says new file should be near its directory, because that's where we're going to get the file. But to make sure that new files can stay with their directories when we make a new directory, we go somewhere else in the disk. So we're moving around the disk trying to fill in all those cylinder groups with data that is going to be used together.

So if you think about the way your files work. Well, you have a home directory. That's a directory. The files in that directory are therefore trying to stay near the entry for the directory itself, so all of your files are trying to stay near each other on the disk and your buddies' files who might be on the same disk are probably in a totally different cylinder because they're in a different directory linked out of the student course or the student Home Directory parent directory. So your files are trying to hang out together so that when you log in and do some work, we're trying to minimize that seek distance. This works really well if you're on your own machine and your buddy is not on the same machine at the same time, but it still works pretty well when you've got some subset of your user base logged in, because you don't need everybody's files available at the same time. You just need the files of the programs that are actually being run to be available. Makes sense? OK.

So we can think about some of the things we could do to try to increase performance. A lot of the performance issues have a payoff with poor reliability. So we can make things reliable by having multiple copies, right? We have multiple FAT tables. We have multiple super blocks. This is such important information that if we lose it, the disk is trash. It's worth investing in that, but it takes space - space that you would like to use for actually important purposes like the files themselves. So the more overhead, the more your disk, the more of your disk is just consumed without actually being able to use it directly.

Another way we can improve reliability is with a synchronous write. So that means every program that wants its data on disk, we schedule the writes and we wait for the write to finish before we tell the program it can proceed. That means that everything that is coming out of the program is guaranteed to be on the disk. But what if we've only written part of a block? This could be a big cost if we have to keep rewriting the same block because somebody is using printf to put little short messages into a 16 kilobyte file block. So we could delay the writing. But then the problem is what happens to writes that have been

written only to a cache, but not to the disk? If you have a disk crash, it's just gone. That can be particularly bad if these are writes to, let's say, the tree of indirect blocks. A big chunk of your file might go away because the stitching together information is all gone.

What BSD, which is one of the Unix versions, does? It tries to balance this by saying OK, if I'm going to make a new file, I ensure that the inode makes it out to disk before the directory entry that points to it. So you're not going to have a directory pointing to nowhere because you're going to have allocated and committed the inode before you write to the directory. So if you make a new file and there's a crash in the middle of doing that, you may have committed an inode that is now not attached to anything, but you don't have a directory that is broken. Similarly, we remove the directory before deallocating the inode. And finally we write the deallocated inode to the disk before we walk through deallocating disk blocks.

You may have noticed that if you have a bad crash, right? Kernel panicked or power out or something like this. And you turn your machine on, it does a whole bunch of work on the disk. There's a whole bunch of scrubbing going on. What it's doing is looking for lost pieces. So if you had removed the directory entry but you hadn't deallocated the inode, you essentially have an orphaned file. Right, it's a memory leak on disk. But you can know that because nothing points to that inode. So if it traverses the entire directory structure and nobody knows what this file is, then you can delete the file. But to do that you have to traverse the entire directory structure. So this is what happens if after a power outage type crash, you turn your machine on and it starts chugging away. It's doing that kind of work.

Until fairly recently, and still with some versions of Unix and Linux, it likes to do that before it lets anyone log in. But other ones are a little more loosey-goosey, and they'll say, yeah, I'll let you log in and I'll just sort this out in the background and you'll see it doing a lot of work. But it still has to find any of these lost pieces.

Final BSD speed up is to try to collate the writes into one and keep the writes together, and it also looks to see how many requests there are in the IO queue. So that reads can be served and the writes can be shoved out when the disk is otherwise not busy. OK.

All right, this diagram I don't know why the microphone is extra sensitive today. This diagram is trying to show you logically how the system calls that you're making, so open, close, read, write, relate to what actually is happening under the covers with the data on the disk or in the various caches.

So when you make a system call, right, you may not have thought about this, but those of you who've done programming on your own machine and then you copy it to Linux.socs and you run it there - fundamentally, you are expecting that when you have a local disk installed directly in your computer, that that disk works the same as some virtual magic disks that we have at the university, where it's not even in the same room as the computer. We do this through what we call a virtual file system layer. So that holds a set of data structures that has enough information about how a file system must work, right, so it holds

information about index nodes, information about file offsets. But it doesn't know anything about second indirect blocks or FAT structures or anything like that. It just knows things like file system attributes and offsets.

Below that, we're going to have a record for the actual type of operating system we have. So you can have in the virtual operating system the generic information in what we call the vnode, the virtual node, but then the file system specific stuff is appended on to the data structure where the first few fields have all of that good generic information. So the generic information is the same regardless what file system you're on.

The file system specific information has to do with some individual file system, so maybe you're on UFS, right? So ext4 or whatever you might be using on your Linux or other Unix platform and it'll have all sorts of great stuff like user, group, other bits and sticky bits and user ID bits and all that wonderful thing to keep track of the Unix information. Or maybe you're on an old DOS FAT system, in which case it doesn't have any concept of a user, it's just a file. It's there on the files and it can have its read-only bit set or not, and it also has a thing called hidden bit. Those are pretty much the only two attributes you have. But things like file size, file access time - those are in the generic node because they're common across all of them.

So here we see three different families of file system. The Unix and Linux file systems or things like DOS are both associated with physical disks. You access the physical disk by going through a physical block buffer cache, so this is literally blocks of data from your file system that are being stored in memory in a cache. So if you access it, it gets copied into memory and it's going to hang around in memory for a while until we have so many runs in memory that we have to push something out, probably using least frequently or least recently used.

But we have actually the data from the files, metadata or file data hanging out there in a memory buffer in memory. We call this the physical block buffer cache because it's mirroring data that is on a physical disk that is plugged into our actual CPU. Right. So if you enter the kernel, you do your trap to read, it's going to schedule a read, put the data in a buffer, and then the kernel on passing things back and forth is going to move things into a user process from that buffer, and the buffer is used by the device driver as a place to put the data from the disk. OK, you had a question?

OK. OK, so this is mirroring things physically attached to this computer. But as you all know, you can have a file system from somebody else's computer. This is part of the magic we now call the cloud. We used to just call it a network file system, which is why it's called NFS here, which stands literally for Network File System, which was a project from Sun Microsystems, now part of Oracle.

All they do is they say, well, OK, if we can have a buffer containing data about a file here in the kernel, as long as I have some code to go across the network to get the data, I could have data in a buffer here in the

kernel representing a file that is potentially far away. And accessed over the network or accessed from some local or remote disk, so the physical block buffer cache and the logical block buffer cache are both just collections of disk data.

The big difference is that the physical block buffer cache will contain metadata, because if we're chasing through, let's say a UFS file where we have indirect blocks, or if we're chasing through a B-tree or we're chasing through a FAT entry, we need that data somewhere. It has to be in memory. So that goes into the physical block buffer cache, but the logical data of the files itself. The data making up the bytes you think of as "this is my file" - we just put those in the logical block buffer cache and then we can share it with fancy things like network file systems so that you can access data from someone else's computer if they have a program willing to give you the logical data from that file, based on that request and the reason it's transparent to you.

That when you log into Linux.socs that your files are actually on a totally different computer is because you're coming through this indirection layer. So you just see the virtual file system part where it's interested in things like what is the file name. It goes down to the network file system and starts requesting the logical blocks of that file on your behalf, which then you can compile or read in your text editor or whatever it is that you're trying to do. Question over here. No. OK.

So the key thing from this diagram is to recognize that we can with a given data structure have further been allocated to the common pieces of information we need for anything in a file system. We're going to want to talk about. And then we extend it with the things that are specific to our file system that we're actually specifically using for whichever location this file is on. So you've got in your computer, you've got probably 1 file system if you've got multiple disks, you've got at least one file system per disk. If you've partitioned it, you've effectively divided sets of cylinders into non overlapping groups and said I'm going to build a different file system in this set of cylinders than in this one, which is why you can have an NTFS boot partition, ext4 boot partition. When you're talking to them from your kernel, you're simply going to the specific implementation for that file system, so that when you open a file, if it's on a FAT formatted partition, you're going to go to the FAT implementation code in your kernel. If it's from the UFS Partition, you're gonna go to UFS code, ext4 partition ext4 code, and if it's a network type setup, you're going to use some kind of network code. So far so good.

OK so here I'm trying to show you that if you have multiple physical disks or partitions, they each have a separate device driver. OK. So you're going to have a separate device driver for, let's say, an NTFS partition. But from your user process, there's nothing to stop you from reading data from an NTFS partition and then I don't know, writing data to the temp directory on your ext4 partition. From your programs point of view, very little was different. You called open in both cases, you called read or write. Some data went in or out of your program as a byte stream. Your program doesn't really care. All of that information is hidden down inside of the kernel.

The kernel is backing its ability to show you different visions of this file strategy with a set of buffers that have to exist in physical memory, along with all of the other things we need in physical memory, like our frames holding the memory for the programs. So we have a thing called an mbuf, which is a fancy name for memory buffer, which is essentially a bunch of bytes of a known size, where the memory buffer can be used to hold interesting things like disk blocks worth of file data, disk blocks worth of indirect data, or if it turns out that our memory needs for running programs are big and our memory needs for file information is small, we could turn some of the mbufs into frames so they just become empty frames, and then we allocate them to somebody's frame requirement.

If things change and we have fewer programs and some of them are really data hungry, we're going to say, well, maybe I need fewer frames and more data buffers. So I'm just going to reallocate some of the frames from the mbuf pool back into the data buffer allocation, and then I have more space to put file data. OK.

So we're just essentially we have fixed sized chunks. We're using our fixed size mbuf to hold blocks of general purpose storage. From the kernel point of view, it is a big list of mbufs. Some of the mbufs are going to be used as frames, because presumably you want to run some programs, so you're going to need some frames to do that. Some of the mbufs are going to be used as IO disk buffers because it's very popular to have IO in a program, so you're going to need some of those. And all I'm saying is that the kernel has the ability because it has this idea of a generic buffer to say "Oh I seem to have allocated more frames than I need, but I need more IO buffers."

So it can simply reallocate from one to the other. The idea of the mbuf is just a general purpose storage, so the kernel thinks of the world as "I have a bunch of RAM, I've divided it into mbufs. What am I going to use all these mbufs for?" Some of them are going to be needed to run programs. Some of them are going to be needed for IO and you don't need to push anyone out of memory until you've filled them all up. So if you look at any of the utilities to look at your current memory usage, if you're not at 100% memory usage, that means you have spare mbufs just hanging around, and when you need something, the kernel can just say, yeah, I'm going to use those mbufs for this purpose. There's no point pushing someone into the paging disk if you've got extra memory available. OK.

So then back to our tricks to make things go faster. So for reading, well, we know where we are in the file. Right. So when you call open? I guess maybe I should back up slightly. I hope you all know that when you call fopen internally it just calls open.

OK, so fopen gives you back a pointer to a structure of type capital FILE, one of which, one of the fields in that structure is an integer file descriptor, so that when you call open, you just open the file descriptor directly. That's your table of things that are open, gives you back the next number in the table. By default you have three of them open - standard in, standard out, standard error at location 0, 1, and 2. If you open

something else, it'll go in location 3. The 3 tells you where to look in the table for important information like what is the offset in the file? Yeah.

Yes, because fopen has to call open. So fopen, I think I mentioned this earlier, but if not, fopen is just the file descriptor plus some buffers, so you can do fancy things like formatted input or output. So if you want to do printf or scanf, you're going to need fopen because you're going to need those buffers to arrange the bytes so that they look right when they go to the file. open is interested in things like read and write, where I say I have these 17 bytes, please put them on the disk, but you will have formatted them beforehand. You just give them the bytes. Right.

So it's lower level. So if you look at printf, printf calls write. scanf calls read. Read and write are simply the system call level interface to the kernel where it says I have some bytes at this location. It's this many bytes that's associated with this file descriptor. Make the thing happen. Either the read or the write. So if we're reading, we know what the logical offset is of the byte in the file. That's the logical offset in the sequence of bytes we think of as making up the file body. Not an actual physical location on the disk.

So what we have to do is we take the offset and we turn it into a block ID of the logical block in the file by simply dividing by the block size and rounding down. Right, so if we've got a 16 kilobyte block size and we're 17 kilobytes away from the beginning of the file, that means we're in logical block 1. We've gone all the way through logical Block 0. We're in the next block.

Knowing our block number, we then look in the logical buffer cache. That's this thing because maybe we already have the data in memory. If we have the data in memory, there's no point bothering the rest of the device driver to do something with the disk. That's just going to cost us extra. So if it's in the logical buffer cache, if the logical buffer cache contains this block number, well then we just copy the data from that part of memory to the user. We're done.

If the logical buffer cache does not have the data we need, this is very similar to what happens when we have a page fault. We have a need for data, it is not already in memory. Just like in a page fault, we go and we get it. So first we need a place to put it. So we're going to allocate a block in the logical buffer cache to store the block that is on disk. Then we read the whole block.

So even if the user asked for just a couple bytes, remember the smallest IO operation we can have with the disk is a sector. And we've divided the disks into blocks made of sets of sectors. So we read the entire file block from the disk into our logical block. So if we've got a 16 kilobyte block allocation, that means we read all 16 kilobytes from the disk surface and stick them in that logical block buffer in the logical block buffer cache.

To make that happen, we need to know where physically the logical block is, because all we know so far is where it is relative to the beginning of our imaginary data structure of the file, not where it is physically on the actual disk surface. So we need to go look at our metadata. We chase through our FAT table or we

look at our indirect tree to look up where for this logical block, we need to go on the physical disk. OK. So that's what that whole big tree was about. That's what the whole FAT table linking thing is about - is figuring out how to find the physical block for a given logical block in the file.

If it's not there, that means we're in a hole. Every file automatically has a hole that starts at the end of the file. So if you seek past the end of the file and read - you're going to - well, actually, I shouldn't say that. Forget I said that. I'm going to come back to that later. If we have a hole in the file before the end of the file, then we're going to fill it with zeros. So we just fill our logical buffer cache with zeros.

If it is on the disk (sorry I should have an else here), if it is on the disk, then we issue a disk block read to the disk controller. So now we actually get the device driver to schedule the read request with the disk controller. The disk controller then eventually is going to do the work. Presumably we're going to get blocked at this point and we're going to run some other program. Once that lengthy IO process is complete, which we know because the interrupt comes back, then we know that the logical buffer that we chose has been filled with the IO from that disk block. And then we can copy from there the right user data into the user process.

If it's a block in the middle of the file, we have a whole logical block filled with data for this file. If it's a fragmentation block at the end, then we're going to have to copy out of the appropriate part of the block the data for the user, knowing that other parts of the block may belong to other people. But we're the operating system, we can do that because we're the ones who set up the fragmentation block in the first place. The user doesn't get to see that. All they're going to see is what we copy back to them after we're done this operation.

"Why does what happens?"

In the Unix or NTFS file system, if in our big tree structure, let's say we open our file, open a new file, seek to location 1,000,000, and then we start writing data. We're writing at our offset X. The first million bytes of the file were never written. But the offset starting at location 1,000,000 is valid. So we don't bother storing any of the data blocks that would correspond to the first million bytes. So in that case we probably skipped somewhere, maybe even here down into this tree. Well, that's a million blocks in, so we come in somewhere in this tree and say, OK, I'm going to maybe start needing a data block here at location, I don't know 5000 or something and then everything prior to that we just leave with nulls so that we don't need to allocate disk blocks for parts of the file that are by definition full of zeros. Not very common, but there are certain applications that do like to play with that sort of strategy. OK.

Oh, that's right. So is everybody solid here on read? So on read, we're basically saying if I already did the work to put the logical block in the buffer cache, then I can just take the data that the user wants to read out of the buffer cache. If I didn't do that work, then I have to actually populate the buffer cache with data from somewhere which is either going to come as a fill with zeros if it was a hole, or it's gonna come

laboriously from the disk surface which is going to involve scheduling the I/O controller, moving the head to the right place, waiting for the disk to spin around, load the disk data from that location into the logical buffer cache, and only then be getting the interrupt from the IO controller which, remember, is a totally separate chip on the motherboard.

OK, so we asked the IO controller to do stuff. The IO controller is effectively its own little computer. It's going to do the work to get the disk block that we requested and put it in memory and then phone us up and say, yeah, it's there now. That's what the interrupt is back to the CPU. So we make a request in the CPU to get that work scheduled. And we will get notified when it's done. But in the meantime, we're going to run some other program because the program that needed that IO is now blocked, but other programs can presumably run. And we can do that because it's all a completely separate set of silicon managing the IO subsystem. So we've simply made the request through the controller for the appropriate physical blocks after we've calculated the math from the logical block number, where do I go here? Right.

So we calculate the logical block number, look in the cache and if not found then we have to do the metadata to find physically where we are in the disk, bring that in. Away we go. Are we good? OK, so we'll do writing next day and then I'll wrap up IO because we do need to do some of the security. I would like to leave both of the classes in the last week for review. So we'll leave some of the IO stuff unexamined, but we're meeting all the core requirements for the course. So we should be good. OK. Questions anyone? And next day we'll do a...