CIS3110 Lecture 9 - Revised Transcript

Hello everybody. Sorry, we're a couple of minutes after time getting going here. OK. You will remember last day I gave you this diagram where we talked about the three-dimensional addressing on the disk. So the thing to keep in mind is that it is effectively a cube as far as the address goes, yeah. I'm quiet. Is that better? OK. Directional microphone too directional. OK, so you have three different dimensions, the angular position Theta, the radius R and then what I've called Z, which head we're talking about. So I want to make sure it's clear here that because there's the same number of sectors all the way in through the pie section. This is effectively a cubical space because there is a value for R and Z for every value of Theta, even though it looks like a pie right, even though it looks like a round thing, the addressing is such that we have a discrete number of positions around the disk, a discrete number of positions in and out and a discrete number of positions up and down, which you can see on these. So here I'll pass around - have your hand out long, can you look? That's one. Here we got some.

These are hard drives.

But then we have the hard drive.

But you can see you can move the arm, you can move the heads in and out. And so that's attached to your motherboard. So I got two of these. I'll pass these around. So the motherboard, which you may never have thought about this, holds the so-called daughter cards. OK, so the reason it's a motherboard is because these cards can plug in and out for all of your external devices. You can see on the motherboard your CPU. You can see your RAM. You can see your MMU in between your CPU and your RAM and if you look at the parallel lines on the board, you can see the bus going to and from the RAM and you can see it going down to all of these I/O cards. So again I'll pass that around.

People can take a look.

These are fairly old, so we've changed which plug you have to plug the disk in.

Other than that technology, but if you buy a server desktop machine, you're still going to see a motherboard like this. It still has some of the same slot types. It will definitely still have a CPU. It will definitely still have an MMU and it will have an I/O controller, which is what we're talking about here. So that more distant black chip that is closer to the slots. That's what we're talking about here. This I/O controller you'll be - you can see the bus going back and forth to RAM so that it can actually push data from the I/O controller directly into RAM, and you can see the bus between RAM and the CPU. It's harder to find the one line or the very few lines that have the interrupts and so on, because this is a much smaller bus than the wider bus that goes back and forth here, which will be your word size which on these machines is a 32 bit system. Now while we're here, I brought some other disks. So this is a very old full height disk. You can see it's pretty giant. This is much like the ones that are going around. These are the common ones still. If you go and you buy a big disk for your servers. So if you go and you buy anything over a TB, you're probably going to get something this form factor, it's going to be a spinning disk. If you get certainly multiple terabytes, it's almost certainly a spinning disk just like the ones that are going around. The only difference between those and this is that I haven't pried the top off this one, and otherwise the insides are the same. You got a laptop. They scaled them down, same deal. It's just the tiny version of this. Now we've got SSD's, but the other companion in the family.

Definitely the single platter. Their whole life.

There are two different versions of that. But it's the same idea. The only difference between this picture and the floppy is that we just have one platter. The other difference, I suppose, is that the heads which you can see coming in from the side, they're physically mounted to the computer and the disk can be pulled out. The heads stay with the computer when you go from disk to disk, which is in fact a return to the older plan when they first had hard drives on a computer. They were so expensive that you would literally pull out the disk platter stack. They called it a cartridge. So the whole thing was a giant thing that sat on the ground, kind of the size and shape of a washing machine. You would open a hatch in the side and you could lift out the whole cake. It had handles on the top, so you clamped them in. You lifted it off and then you could put a different one in. So if you needed a different disk to do different work, you actually pulled all of the platters out of the machine, stored it in a plastic cartridge, put a different one in, which made sense because you were probably paying about a million and a half for each disk drive, and the actual cartridges were only about 10,000, so you couldn't afford to have an entire million-dollar disk drive for each cartridge and so now we've come full circle. We had the cartridges, the little cartridges, but now we've basically got SSD drives which we'll talk about at the end of this unit and how they differ from this strategy. But the important thing to remember here is that the indexing plan for all of our things that look and act like a disk is the same. So because it is this 3-dimensional geometry, we've got a certain number of tracks, a certain number of sectors and a certain number of cylinders, you're positioning yourself within this indexable cube to get one sector. A sector is the thing I've coloured in dark here. We're going to find out it is a data structure holding 512 bytes. That's a hard drive sector. They just decided this when they started making hard drives. It's been the same ever since. So then we wrapped up last day saying yeah, you can do the math, you can multiply them together, figure out how big your disk is, look at the data rates in terms of how quickly you can get things off because of how fast it's spinning. And you can think about how fast you can get data from the disk into the disk drive buffer. So this is in a buffer physically in the disks that are coming around or in the I/O controller, and then how fast you can push that into memory through the controller to memory bus. Questions on that. OK.

So fundamentally we have this giant 3-dimensional addressable space. But if we have one of these drives where it's actually a spinning drive, one of our big costs is moving those arms so you can swing the arm around yourself. Whoever has the drive is going around the room. Our biggest cost is moving the arm to the right cylinder, so one of the algorithms we can think about is how should we handle multiple requests coming in. Because if you've got a computer, especially a multiprocessing computer, maybe even with multiple cores, the chances of having only one request to your disk drive are pretty low. Requests are going to start shooting in. You're going to be asked to supply different pieces of data from within different places in that cube. So if the biggest cost is moving the arms, we can think about what is the best plan, so the obvious simple one is FIFO. You make a request, I put it on the To Do List. I walk through the To Do List in that order. The problem with this is that if let's say a request comes in for a sector on the outermost edge of the disk, and then one on the innermost edge and then another one on the outermost edge before you've gotten around to doing anything. I think everyone would agree if you had to pick up something on that side of the room and then in the back corner and over here you would not walk from here to the back corner and back. You would pick up both things because you're physically nearby. The arms are physically nearby, so FIFO is a bad plan.

Next obvious one might be shortest job first. Has the same problems we had when we were talking about process scheduling. So if you have a lot of requests and you always go to the one that is nearest to where you are, then any requests that are far away may never get served. So that seems like a problem because those requests are presumably important. They're just far away on the disk. So what we tend to do is what we call the elevator algorithm or scan. So if you think about the way an elevator works, old tiny elevators did FIFO. You could tell because you could stand in the lobby of these old buildings like hotels, and they had a big needle on the wall that told you where the elevator was and you would see it. OK. It's on the 8th floor. It's coming down. I wanna take the elevator - 7, 6, 5. It's stopping. 6, 7, 8, 9 - I want to get on this elevator. It's gone the wrong way. Comes down. Maybe even gets to the second floor and then it goes back up because somebody wanted at the top of the building and it's super annoying. So what do they do now? What does an actual modern elevator do? I'm sure you've all been in elevators multiple times. What is the algorithm? Can anyone explain to me what happens when you get in an elevator? You've got multiple floors requested, yeah. It goes to three. It goes to the 1st.

Well, that would be perfect. So it doesn't actually do that. Yeah.

OK, so what do you mean by one direction at a time? OK, so once it is established, the direction of travel, it will go to the most distant request in that direction, serving as you mentioned, anyone on the way, who has also pressed the button. And once it gets to the most extreme request in a particular direction, and has served that request, then it reverses the direction. Because the nice thing about elevator navigation is there's only two directions, so we only need to worry about up or down. So now it's in a new direction. It will go as far as it needs to go to serve the most distant request in that direction, serving the others on the way, and then again reverse. So if you're in a tall building and it's in the middle of the building and there's lots of activity in the middle of the building and no one at the top and no one at the bottom, it'll never go to the basement and it will never go to the penthouse. It's only if someone's actually there that it bothers committing the time and effort to move the elevator all the way to the end, so it will never travel the full span if there isn't a request at the extremities of that address space.

So in terms of cylinder, we schedule the head movement based on this strategy, so once we're in play, we serve everything until we see the most distant requests. So what happens if another request comes in while we're nearby? I mean, it's the same as an elevator. What's going to happen? Yeah. It may or it may not. So what are the conditions under which it serves the request? So let's say you're in an elevator. You're going up. Somebody just got on the 5th floor. Someone on the 6th and the 4th floor both press the button. What happens? It seems to extend travel. Yeah, and so. So you will pick up the person on the 6th floor because in your current direction of travel you haven't served that area yet, so you can serve requests that come in while you're doing this scan provided they're ahead of you on the scan. So what about the fourth floor person? Are they just out of luck? OK, so once you've finished serving the furthest upward most travel request and you reverse, they'll get picked up on the next pass. So they will get served before the furthest other end request gets served. You're guaranteed that everybody will get served in between the two extremes, and you go to one extreme, turn around, go to the other extreme and you can serve anybody who jumps in on the way. If they jump in ahead of your path of travel, you'll just pick them up as they go. OK.

What about block size? So if we have these discs and they have to operate in blocks now, I mentioned to you the hardware size is 512 bytes nominally, but we can group them together to make them bigger. Is that a good plan? I mean, bigger is more bang for the buck. Every request gives you more data. What's the problem? Yeah. You said it picks more sectors or what?

If the block size is big, you have to read all the blocks.

And I think hidden in there is the fact that you may not care about the data in all the blocks, right? You have to read the block - you're committed to reading a whole block to serve a request, so there's another related issue here. Blocks are multiple sectors, yeah. So the related problem along with, as you said, there's a time issue that you do actually need to read in the data. But the related problem is the space issue. Think about what happens if you want to store a 1-byte file. Yeah.

The more potential, reason to pay? You're going to have it.

You need to store a 1-byte file.

And your minimum allocation is a block. The bigger the block, the more wasted bytes you have in your file. The smaller the block, the fewer wasted bytes you have. But then you're going to have more issues moving around, so larger blocks - the end of the file is going to be likely only partially full, but you get a larger payoff for every block you find. Smaller blocks, less waste, probably more time indexing.

So let's think about that. So I'm going to take you through a couple of file systems here. Here's the good old fashioned FAT file system. So they still use this if you have a small enough device. So if you've ever bought like an SD card for a camera or small USB keys, etcetera, they're all done with FAT file system. So the way this works. We've got our device. The first block on your device is what we call the master boot record. It has information to boot the computer, so when you boot a computer that is set up for master boot record booting, it loads that sector. And in that sector is a pointer to some other physical block on the disk that has a tiny program that tells the computer how to boot. So in those 512 bytes, there needs to be enough information to tell the computer what to do. Anyone of you who put multi-boot stuff on your computer, it is that program. So LILO or any of those other multi-boot loaders, they're the things that the master boot record points to. Other things in the master boot record tell you how big the disk is and so on. We'll talk about that in a moment.

Right after the master boot record, we have two copies of all of the indexing information. It's called the file allocation table or FAT table. Why would we have two identical copies of the indexing information? Seems like a waste of space. Yeah. It's a backup. So this is a physical disk. If you think about the disks that are moving around the room, there's a little read-write head on the end of an arm right above the medium. It's possible it can hit the medium.

We call that a head crash. It has been likened - if you think about the scale - to a 747 jet traveling at jet plane speed three inches off the ground. That is the distance between the read write head and the disk surface relative to the size of the read write head. So it's pretty close. So this is why you don't pick up your desktop computer and shake it, because you're likely to hit the head into the disk, in which case you now have a bad sector on your disk. And we can't read or write anything there anymore, because you've destroyed the media. You've literally scraped it off the disk surface. You can't record anything anymore. You can't read anything anymore. So if you do that in the middle of your indexing information for your file system, what does that mean for your file system? You only have one copy. It's gone. You've lost your entire disc. If you think about this like it was a linked list, which you'll find out it is, it would be like losing the head pointer to your list. The list exists, but who knows where in memory it is. Are you going to be able to find it by just looking through memory byte by byte and hope that you can recognize the data? Unlikely. So they give you 2 copies of that key information in case you have a problem with one of them, at which time you hope you can read the information off the disk and put it somewhere.

Then we have a couple of blocks that have the root directory information. And so the way this is set up is it's simply an array of directory entry records. You're only allowed on a FAT file system, or in fact an NTFS file system, which is the new file system that Microsoft uses. You're only allowed 1024 entries in the root directory because it's a fixed size array that they've set up on the disk. So it starts out. They're all set up. They're all blank. But you've got 1024 entries for the root directory. As you add things to the root directory, you simply fill in values in this list. It contains things like what is the name of the file, how big is the file, where does the file start in terms of the block number, what was the modification time, etcetera, yeah.

Yes.

Is actually. It is actually the first physical sector on the disk. Yeah, which is the hub side, not the outer side, but yeah, it's the first addressable sector of the disk, the one at location 000. And then the FAT starts at same cylinder, same track, next sector. And then once you fill up that sector, you're going to go down one level to be the track on the underside of that platter, and you're going to work your way down until you've gone through the cylinder, yeah. I gloss that a little bit. It is going to be multiple sectors which you'll see in in a moment. So you got 2 full sets of the file allocation table, which depending on how big your disc is may have to be many many sectors. OK, so if you buy a big disk, obviously you need more room to index the big disks than a small one. So 2 copies of the FAT, then root directory, which again is going to be multiple sectors and then after the root directory we have the data blocks for the files and we're going to find out that the files - there's a special type of file called directory which simply has another array data structure of entries. So if you have a subdirectory that's simply stored in a file. OK, so let's unpack this.

Here we have an example disk. Now for making it simple, I've only put one copy of the FAT shown here, so we've got a boot block. So your master boot record goes in the boot block. The FAT - this is a tiny disc, an unrealistically tiny disk, so we have only two sectors or two blocks. Rather set aside for the file allocation table. I've got only one sector, sorry. I need to make sure I don't confuse with the sector and block. Our block size is 1024 so there's 2 disk sectors per block. Got a boot block, 2 blocks, so 2 kilobytes of information for the FAT, one block - one kilobyte of information for the root directory and then the remaining blocks are all set up as data.

So I've got two example files here in the root directory: ABC.DAT and A2.C. In DOS, therefore, in the FAT file system, the names have to follow an 8 character and then a dot and then three character format. Unless you're running an extension where they do a little trick, we'll talk about in a few moments to get a second longer name, but anyone who's had a Windows machine, if you type DIR with either the minus X or I think the minus I option, it will tell you the short name for all of your entries, and then there's a longer name. So if you look at program files, you'll find out that program files is actually called PROGRA~1. Because that's how they fit it into that 8.3 standard.

So we've got a name which is just in the structure, that is the entry for the directory entry for FAT in the 8.3 format. We've got the size in bytes of the file. We've got the modification time, which for rather arcane reasons are in even numbered seconds since 1970. It only counts 2 seconds at a time and then we've got a location for the first block. So notice they have two different first blocks, so we could go and find the beginning of the data by going to that block. But there's a big thing missing here. This is a 7200 byte long file. A block only contains 1K. So where are the rest of the blocks? We know that it works. We know that you can read data through the file. So somehow there need to be additional information to tell you how to find the rest of the file once you know where to find the beginning of the file. That's what's in the FAT or the file allocation table of this file system.

So notice I just turned on numbering here so that the big numbers number the locations on the disk in block numbers. So you format it with a block number. In this case, we've said two sectors per block or block of 1024 bytes. So therefore block 0 is the first block on the disk. But logically, the first block we can put data in is the first block that occurs after all of this overhead that we need to keep track of everything. So physical block 4 is logical block 0 because it's the first one that wasn't incorporated in all of this root directory, FAT, boot block stuff. And then after that they just count together. So physical block 5 is logical block 1, 6 is 2, etcetera, etcetera. And you can see that although we have 16 physical blocks on this disc, we only get 11 as the highest index for our logical block, which is of course the 12th block because we started from zero. So we can have logical block 0 through B in physical block 4 through F. Everybody with me so far? So the logical block is the one we're talking about here. So first block 2 is actually talking about logical block 2, full of G's. First block 8 is going to be logical block 8.

But the FAT is the additional index information. So I've got my file and I've pointed it at this slot in the FAT, which is simply an array of integers. The 8th integer in the array tells us information about the 8th logical data block on the disk. So for FAT we come here and we see logical entry 8. Next one is 3. Logical entry 3, next one is 6. Logical entry 6, next one is A. Logical entry A is marked end of file. What is this data structure? It's a linked list. You can only go forward through it. OK. That's a big cost for FAT because if you ever want to back up in your file, how do you back up in a linked list? You start again at the beginning and you chase forward until you come to the right place. It's very expensive in the FAT allocation scheme to do random access in a file. If you do seek, it's going to cost you a lot of time if you're trying to back up. If you're going forward, it's good.

So here we have a linked list where we know from the start position 8, that tells us the next one, so on and so forth. This is a linked list with no data because the data is a parallel linked list in the data blocks. But the data blocks don't know how to find each other, so you go to block 8 and you read 1024 bytes of data from the data block. There's no other information. So if you come to the last H in data block 8 (physical block C), in order to figure out where to read the next byte, you have to refer to the FAT table, so you're going to say, OK, I was in 8. I need to look up location 8 in the FAT table. That says to go to location 3. So then I can go to logical block 3 and then I know that right after these H's the next byte is going to be A, and I'm going to have these 1024 As, and I get to the end of that. And I say, oh, I need to know what happens next. I'm in logical block 3, so I go to slot 3 of the FAT table. It says move on to 6. So I move on to 6 and I'm reading E's. After you're done the E's you move on to logical block A which is full of O's. How many O's are valid in the file? All the information is up there on the screen, yeah. So there's 1024 physically in the block, but how many are logically part of the file? Because we have exactly what was talked about earlier, that we have a partially filled block, which we know because 7200 is not evenly divisible by 1024. If it was evenly divisible - right. So if the size was a multiple of 1024, the last block would be perfectly full.

Yeah. Well, it's simply given - it just literally holds, or points to, the next block from any particular block.

OK, so how do we know how many bytes are in the last block?

And so how did you get - because you have 7200 - or 1024 x 7. Yeah, 7 x 1024. You know, you've had.

OK, that is a valid way to calculate that. There is another, probably more commonly expressed way. Because if it's evenly divisible by 1024, how many are going to be left over that we can't fit in the last block? Let's phrase it that way. You just modulus by the size of the block. That tells you the remaining piece, right? Literally the remainder that is going to be in that last block. So if the remainder is zero, you don't need to allocate the block. If you have a file that is of 0 size, you in fact don't even bother with the first block. You just say the first block is not valid. First block is end of file, there are no bytes in this file. We do not need to use up any disk space representing the 0 bytes that are in this file. We simply tell you there are no bytes. If the math works out that the last byte of the file is the last byte of the block, you don't need to go to the next block. But if you do need to go to the next block, that means that block probably isn't full. You might, in the worst case, have a 1 remainder, in which case you need to allocate a whole block for that one byte, and you've got 1023 wasted bytes in that block. In the best case it's completely full or nearly completely full. So let's say you've got 1023 bytes in the last block. Well, now you've only got 1 byte of extra space.

OK, when you're reading it. So as the operating system, when you're scheduling the reads, you're getting back whole blocks. When you're passing the data back through some M-buffer to user space because they made a system call to read, you're going to know how many bytes are valid in that last block, and instead of giving them the extra bytes, you're going to tell them read has come to a stop because you're at the end of your file. So even though you read it off the disk, you're only going to pass the valid part back to the user. So far so good. OK, but the disk doesn't just have one file on it. That would be a pretty irresponsible use of your disk, yeah.

OK.

How we traverse the FAT or this or both? OK. So we start at 8. Logical slot 0 is the first slot in the FAT table. It's just a list of integers, right? So if we want slot 8, we go up to 8. OK, Eight has the entry 3 and so this is a linked list. This is basically just a table of your next pointers. So slot 8 says go to 3. Now we come here. Slot 3 says go to 6. Slot 6 says go to A. Slot A says end of file which if it was a linked list in memory would be a null. We use a different value for end of file, but it's the same concept. OK, so far so good. So we have one FAT table and it has to hold all of the indexing for all of the files. So that means ABC.DAT. Oh sorry, there's my end of file discussion slides. So we had 904 bytes of waste inside of physical block E (logical block A).

ABC.DAT is also in the FAT table. So I took away the big arrows to the blocks to make it a little bit less confusing, but you can see I've got 2 sets of interspersed indexing arrows for two different linked lists, yeah. OK, let's back up 1. So what I'm saying here is that when it was pointed out the number of bytes that were required in this block to represent the last file. So we could do file size. So that was 7200 / 1024. What we find is that 7200 / 1024 gives us 7 with a remainder of 1024 - 904 = 120. That's 96 + 24 = 120, which is I think what he said. So he said how big is the part before my little brick work? There's 120 bytes in the file. All I'm saying is that if the whole bin is 1024 bytes and there's 120 in the file, then there's 904 that aren't in the file that still have to be in the block. So the first 1024 is stored in logical block 8. The 2nd 1024 is in logical block 3. The 3rd 1024 is in logical block 6. And then the last is here. That's still less than 7200, I think I have a hexadecimal number here screwing me up. Or after many many iterations of this slide, you've noticed a new problem. No, it's gotta be a new problem. I've gone through literally dozens of times, OK.

Yes.

Noted. Did I just leave one whole block out? 1, 2, 3, 4 - No, OK.

Right.

OK. Noted. I will go back and look at this and make sure that the example gets updated to be consistent.

And.

But what I want to point out here is that in our linked list from A2.C - purple arrow set of links. Because presumably none of those bytes or none of those blocks were involved in the blocks we need for ABC.DAT. That means none of the FAT entries are involved in the representation for ABC.DAT either. So we can fit all of our linked lists in the same set of next pointers, because they have to be discrete blocks. You can't have the same block in two different files, or you have a broken structure, yeah.

Yes?

Is that? Parallel linked lists.

I understand how the FAT is one each like the next one, but then how does it reverse through the actual?

Oh.

OK, that's OK. I yes, I probably did say parallel linked list. So what I'm trying to say is that in this diagram, when we see two sets of purple arrows both starting from here. This is what I mean by the parallel linked list, so you know from this number two things. You know that to find the data for the first block of the file, you go to logical block 8 of the disk. And you know that to find the information for the next pointer associated with that block you go to slot 8 of the FAT table. All of the next pointers are here, so if you think about CS 2520 programming assignment - write me a linked list. You probably made a structure. It has a next pointer in it and it has some data and it's together in memory. What we did is we said, OK, let's make a linked list using an array model to store all of our pieces. But what I've got is 2 arrays, 1 for all the next pointers, one for all of the data, so that at a given position in the two arrays, they talk about the same logical thing, so slot 8 and block 8 are the two halves of that linked list node. Moving on then to slot 3 and block 3. Those are the two halves of the next linked list node. So you're working in parallel through the FAT table while you're working simultaneously at the same indexed offsets through the blocks in the data block portion of the disk. OK. Does that make it more clear?

All right. And then to have our second file. So first block here is 2. 2 was, if I back up, not involved in the other file. So we have here another linked list. 2 takes us to this slot, which says the next one is at position 4, which says the next one's at position 9, which says the next one's at position B, which says that the next one's at position 0, which says the next one's at position 7, which says it's at 5, which says it's the end of the file. OK, so we're skipping forward and back through our set of - you can think of them as next pointers or index locations, but that's pretty cheap because these are just two separate disk blocks. So we're going to be able to like, if we load this disk block into memory, we get all of these numbers. This skipping around doesn't cost us very much, but we have to simultaneously skip around following the same pattern in the data blocks, and that is going to cost us a lot because that's going to take us when we went from slot B to slot 0 from 1 end of the disk all the way to the other, yeah.

It does the boot block, sorry.

Yeah.

So.

OK, so this is a good question. You going to talk in detail about the?

Boot block in a moment.

But for simplicity of formatting for file.

Process.

OK, so the question is, does the boot - does the presence of the boot block indicate whether there's a operating system on the disk? I think this is basically what you asked. Right. So if you don't have an operating system on the desk, there's no need to boot from that disk. So you could say, why would we bother having a boot block? And that would be a very good question, especially back in the 60s when you're paying millions of dollars for the disk. But what they do is they actually waste a bunch of your disks, so they take the entire first track for boot information, regardless whether you have anything to boot on that disk or not. It's just an overhead that they take up. It is annoying, but it is true. So there is in fact always a boot block in the sense of a block set aside to hold boot information, but it may not actually have any information to tell you to boot.

OK, so the key here is I want you to see that you can have multiple different linked lists in the same FAT table provided the same entry never gets added to both lists or multiple lists. What would happen if we had the same entry in multiple lists? What if two entries, let's say pointed at location 0? In what sense? Overwrite? Would it? It would. Yeah. So if we had, let's say we had ABC.DAT as it is and let's say we swapped this eight out and just made it a 0. Well then the data that was in the second-half of ABC.DAT would be the data that was at the beginning of A2.C. It makes no sense. It's because you've essentially got 2 linked lists, and you've pointed 1 into the middle of the other. So the latter half of the linked list has to be attached to both of them. You're going to be iterating along, and then all of a sudden it's like your two trains joined. And the rest of it is the same data because it's in this. It's literally in the same blocks, so that would only happen if the file system was broken, because that is not a logical thing to ever have happen. If you have some kind of error where it corrupts FAT table, you can have exactly this kind of nonsense. I've seen files that go around in a circle because they just link back to themselves, and I mean, that's no good. Right, you've got some file that is just endlessly repeating. So it is important that the FAT table does contain correctly structured linked list data, where each file has a separate linked list that does not involve any of the entries and therefore does not involve any of the data blocks of any of the other linked lists. Was that a question over here or no? OK. In the back, yeah.

How does sector - so a data block is a choice where you decide that the sector is too small and you'd like multiple sectors to be considered as one block. So here we've simply said two sectors per block. Therefore a block is 1024 bytes. OK. The other thing in the FAT table is an entry to say whether or not something is used. So this free value. It's a specific value we put into the table to say this is not in fact part of any file. Says that logical block 1, so physical block 5 is the one full of C's. It isn't attached to anything. So if we want to grow a file or if we want to make a new file that is a block that we can add to the file.

So if we had, backing up, if we had this, we could write 904 additional bytes of data into the end of block A. And then at that point, we could add in block 1. Append it to the file, so we're going to take the end of file here, replace it with a 1. The free then becomes end of file and then the bytes in here can be added in. What happens when we get to 1024 bytes into block 1? How many entries did we have marked free? There was only one entry right. So what happens? Disaster. Your disk explodes, no? So you're trying to write to the disk. You try to write the first byte after the end of logical block 1, which is the last one we've now attached to this file. What occurs? Oh yeah. So we try to do it - write. The traffic to the kernel, the kernel says OK, I have some data I'm supposed to write to the file. It goes to the disk handling code. It says, yeah, I'm appending in here into the end of this block, and then when it reaches the end of the block, it looks for another free block. There is no additional free block. So at that point write returns an error. It tells you it only was able to write a certain number of bytes and if you try to write again, you're going to get a -1. You do not have anymore disk space. If you look at the error code in errno, it will tell you that you are out of space. That's literally the error message you get under Linux, so your disk is full. You're not going to be able to do anything until you free up some space on the disk by, say, deleting another file or something.

No.

So the free entries are just flagged in this way where we have a specific value that we put in the FAT table and so to find one it just scans through the FAT table until it sees a free entry and then it can decide to use that. You've got multiple free entries it could potentially do something smart to figure out whether you had one that was closer physically to the one you wanted. We'll talk about that again probably next day.

Is everybody solid on this so far? OK so here is I think the slide that you wanted to see earlier. What is in the master boot record? So there is the jump location. So that just gives you a physical block address where there's some code that can be run to make the system boot. Most modern boot loaders are what we call a two-stage loader, so there's enough code in that 512 byte sector to do a little bit of file system code to find a slightly larger boot program that knows how to boot Linux or Windows or whatever it is. Also, the size, the sector size, some devices are not set at 512 bytes. Notably Sun for a while was making a lot of their optical media. You had potentially sector sizes up to 4096 bytes, but it's almost always going to be 512 bytes, and certainly that is the size on all the hard drives. So we take up an integer worth of space in the master boot record to tell you that your sector size is 512 bytes even though your sector size is pretty much guaranteed to be 512 bytes. Tells you how many copies of the FAT table we've put on the disk, default being 2. Then it tells us some physical stuff about the disk. What is the total number of sectors available on this disk? How many sectors are available per track? How many heads are there? How many sectors per block so that it implies the block size? How many entries does the root directory have? And I as I mentioned the Windows standard is 1024 and then we have a couple of other things like media type which is a code to tell you what the actual media format is. This lets you do fancy things based on if it's a CD-ROM or a DVD versus a hard drive etc. You can treat them in different ways. And then what is called the OEM name. And so this is how boot loaders know that they're on a drive that was made, let's say, for Linux because you can put that information here in the OEM name. I think there's 24 bytes there.

The master boot record, although I've got it here in the FAT data structure slides. This is the same for every file system you're going to have that is a master boot record file system. It really has nothing directly to do with FAT or DOS. This is just the structure that is in the 1st 512 bytes.

A directory entry in FAT has an 8 character long, so 8 ASCII byte file name, a three ASCII byte extension. It's padded with spaces. So there's actually no difference on DOS between A2 space, dot C and A2 dot C it literally can't tell the difference between those two. We've got attributes, so these are things like read only, hidden, etc. We've got a modification date or time. The file length in bytes, and that first block. OK, so to make a linked list based file, this is how we do it.

Little brief - I'm going to go through the next slides pretty quickly here. All I really want you to see here is that as time went on and we got bigger and bigger devices, the response from Microsoft was to try to tune up the number of - they call their blocks clusters, so the number of sectors per cluster as you get different size files. So when they were tiny so a 360K file or drive rather that's a small floppy disk. These are the standard size floppy disks. The ones I handed around were either 1.44 or 2.88. One or two sectors per cluster, so each one of the data blocks is either 512 bytes or 1K. Then as you go up, you can see their response was to start putting more sectors in a cluster. They didn't want to run out of entries in their 16 bit table because at the end of the day, if you've got a 16 bit integer, you can only count so high. You can only have 65,536 entries and you're going to need free, end-of-file, etcetera. So to make it be able to span the size of the disk with only a 16 bit table, notice they went up from 1-2 when they went to a 16 megabyte device. You can see it starts scaling up pretty fast. By the time you're in a 4 gig disc, it's a 64 kilobyte block size. Every one byte file is using 64 kilobytes of space on the disk. It doesn't get better as they get bigger. So when you're at 16 gig each and every block is allocated at 256 kilobytes, which is one of the reasons that you see that you go through Microsoft disks pretty fast. You run out of space really quickly because there's lots of little files on those file systems. Most of the files are nowhere near 256K so it is only one block per file.

Cool.

But it's mostly empty. So you're chewing through your disc and you're not even storing anything in most of the space because it's the other half of that half filled block, yeah. Sorry. So the only difference in terms of this construction to a solid-state drive is that all of our scan discussion is out the window because you don't have a head to physically move, but everything else is exactly the same. You still have a sector 000 and then 001. It's just not physically arranged in cylinders. Sorry.

Like or is this like modern? Yeah.

So if you have like 1 file per cluster and then I mean I feel like you have like every like one minute right then. If you have as many files as you do where you like, load the free space, but know that.

Welcome back and they want to look at the plan. But yeah, they still do this. So this is why.

Like you may have noticed, you buy a big disk and then two days later it seems like it's full. This is what's happening. OK, so this is FAT. NTFS is proprietary and somewhat complicated, so I'm not going to go through it in the same level of detail, but I will come back to how NTFS works. Oh yeah, these are just cluster sizes for even bigger disks. They get a little bit better, but not a huge amount better. And then here's the same cluster sizes for NTFS. You can see it's the same kind of exponential scaling. This will, however, so this is all the way up to a 256 terabyte disk. Maybe this is what you're asking me about. Modern versus older. Yeah, so. You cannot format a disk above a certain size using FAT. It simply tells you no. You have to use our newer file system, which is called - there's ex-FAT, extended FAT and then NTFS is the one that goes bigger and the smallest indexable file system in NTFS is 7MB but the biggest one is 256 terabytes, so we'll be alright for a while. We can't buy a disk at that size yet and they'll just have an update when you can. So this is how this works is they just update. You got a Windows Update and then you can plug in a bigger disk. If you never run Windows update, you could potentially plug in a disk and it will only use the first few cylinders because it can't talk about the rest of it.

OK. We'll give you the overview of the Unix file system and we'll go through some details next day. So FAT is a linked list where we store all of the indexing information in a table that we can find at the beginning of the disk. And remember, in FAT, we had another table with the file names in it. The Unix plan - we're going to take that and we're going to basically put it on its head. So instead of a linked list, a Unix file is a tree where the root of the tree is a data structure called an index node or inode. The directories are just stored in files and we'll come back to how the Unix one does that when I've done the Unix file example. Under NTFS and FAT subdirectories are also stored in files. It's just a file with a list of those directory structure records in it. In Unix, the root directory is the root of the tree - the file system tree. And instead of a boot block we have a structure called the Super Block, which tells us how a Unix file system is set up.

OK, so in the Super block. It is the first block of the file system. And it is then on a fixed repeating pattern through the rest of the disk again, so that if you have a disk crash and you lose the first block, you can then scan ahead until where the second super block should be and read that one, and if that one's bad, you can move ahead to the third one and one of the complaints about Unix and Linux by extension is that when they planned this out, file systems were at the size where you had like 16 to 32 Super Blocks, whereas now you have millions and it is really unlikely that you're going to have millions of disk crashes across your disk and still have a functional disk. So a few people have said why do we save so many copies of the Super block? But they're still doing it.

So the fields: Offset of the first data block. Again size of the whole file system. Which inode has the roots. We have free lists, so we'll talk about this. This is a linked list to tell us where the information is about what is free. All the device information for cylinder head sector. A dirty flag to tell us whether we've made any modifications to the file system that have not yet been stored to the disk, but basically a caching issue. How many blocks there are per cylinder. A volume label which is again just a character tag and then the number of free inodes and blocks.

You've heard, sorry.

That's right. And.

There so. A file. So a directory is technically a file that contains directory information on both of these file systems we talked about. The way the directory information works is going to be very different here than just a table of directory entries with names and sizes and so on. So I don't think I've totally answered your question, but I think your question will get answered as soon as I show you the file structure and the directory structure and then I think we can move back and make sure that that's all. Was that a question over here? And I'll warn you, we may not get to the end of that discussion today. But I will loop back. OK.

So in an inode - so this is one of the index nodes. They are in tables that we'll find out are sort of interspersed in the disk between data blocks, but they're in tables, much like the FAT table. But unlike the FAT table, it's not just an integer. It's a whole structure. And it is a structure complex enough to describe an entire file. So here is where we store the mode. So if you have ever used chmod or change mode, you were storing the information that you typed as an argument to chmod in the inode associated with the thing you were changing. So this is where your read, write, execute bits are for user, group and other. This is where the sticky bit is and so on. That goes in the file mode. The link count we'll come back to. It is a very interesting idea. This tells us how many things are referring to this file system entry. So when you move a file, we're going to make an entry in the new directory, increment the link count, and then remove the entry from the old directory. You can have multiple links which we call hard links and we can do additional tricks having to do with programs that I'll come back to next day. Owner ID. This is a number corresponding to the user ID number of whoever owns this file. There's a similar group ID, and then things that we expect like file size in bytes, time for last access, time for last modified, time for the last time this inode was accessed. And then an array of file block addresses. But it's not all the file blocks.

So the way the inode works, it's a relatively small data structure. In that table of inodes. Has all that information here at the top. And then in this example I've got 6 identities of data blocks where we just reference them directly out of the inode. So if you've got an inode for a 0 length file, you need the inode. It has a mode. It has the size. It has all that. But these are all just null. And the size is set to 0 so it knows that they're all null. If you've got a 1 byte file, then the first entry can point at a data block to hold that byte, and you know from the file size, just like we did in FAT, that you only want one byte from that file. If you've got a file big enough to require more than one data block, we can just move to the next one. This allows us to have all the data blocks for relatively small files instantly accessible from the inode. What happens if we back up? What happens if we're in the fourth data block and we want to seek back to the beginning? Remember, in the linked list model, if you want to back up, you're going to have to go to the beginning of the linked list and chase pointers until you find the right place.

It's the thing.

It's a table in this data structure, so all we do is we say you don't want to be in block, let's say 4 anymore and you want to be in block two. Well, I just look for the entry for Block 2 and there's the block and I load it.

Right.

One disk seek, 1 disk access, we can scatter ourselves around in here. No problem whatsoever, yeah. Don't worry about that. We'll have a whole discussion on link count because there's some really fun things we can do with it. OK, so I haven't mentioned block size here, but whatever your block size is, the first block is referenced here, second one here etcetera, etcetera, etcetera. OK. But this only lets us do small files. And even if we had a really large block size, we all know that we've stored files on Linux that are bigger than 5 or 6 blocks in length. So what do we do? Well, I told you it's a tree. What we have is we have these three additional pointers which you can see right here. I've got set to null.

The first one of these is called the 1st indirect block pointer. It refers to a full data block size block that holds references to the next few data blocks in the file, so we have to from the inode, go to this block and then that's going to give us the address of data Block 6 through 1029. So tiny files we can do directly. Moderate size files we can do with this one skip to get this additional piece of our index table. It's essentially the entry of the table after this one ran out. You can see this is entry 0 through 5. This is simply entry 6 up to entry 1029. When we get to the end of the data block that is in entry 1029 and we want to go to the next block of the file, we go to what we call the second indirect block entry, which points at a block full of pointers to 1st indirect blocks that work exactly like this. Right. So we've got a tree. Our tree points at this node, which then has child nodes. There are 1024 child nodes in this example. The first of these child nodes has an additional 1024 entries for the tree. And then as you write to the file, you just keep allocating data blocks and adding new blocks to this tree until you've gotten to the point where you've stored 1,049,605 data blocks in the tree.

At that point, if you write again, we pull the same trick. Only now it's called a third indirect pointer. So if the first indirect pointer pointed directly at an index block, and the second indirect pointer pointed at an index block that pointed at the index blocks. I don't think you'll be too surprised to see that the third indirect pointer is simply a tree with two layers, so you come from here to a block with 1024 pointers to second level index blocks, each of which have 1024 pointers to first level index blocks. And this list just continues. So block 1,049,605 is here, 1,049,606 is here. And then we walk all the way up here, and by the time you get to the other end of the tree, the very last block that we could index is block 1,074,791,429.

Sorry.

Yeah.

But the key I want you to see here. So the take away today. That was a linked list. This has all the advantages that you have of a tree. So wherever you are in this giant world of nonsense lengths, if you want to go anywhere else, sure, you might have to go back to the index node and go down. But it's a really short, really broad tree. So the most disk accesses you'll need is inode, third indirect route, second indirect middle layer, indirect node index and then you know where 1024 data blocks are right there. If you're reading sequentially every time you read one of these blocks, you have the whole list for the next 1023 blocks until you skip to the next indirect block segment. It's only when you go from one tree to the next tree that you have a little bit of extra cost where you have to go back up to the top. But again, a limited number of additional seeks required. So this is one of the reasons that the Unix style system is way more efficient at seeking around than FAT. Now NTFS they also went for a tree. We'll come back and compare that at the end. OK. I'll let you go for tonight.