Revised Transcript of CIS3110 Lecture 12 (2025-03-20)

All right, everybody, off we go. So a couple of notes on assignment 4.

We're reading the blocks of data from an actual file system. So you're loading disk blocks into RAM so you can then examine the bytes making up the disk blocks. Looking at the bytes in the disk block is a lot like what you were doing in assignment one where you had a single block of memory and you were carving it up into your own strategy for allocating pieces within that block for different purposes, but you had to figure out what was happening, what the purpose was for individual bytes at different locations. Who is designing what goes into the data structures that we're using in this assignment? Where is the design coming from? Who's the author?

No. Oh.

We can't. We don't have any control. It's not me.

Let's see.

It's Microsoft.

Like.

We're reading Microsoft file system, so we can't change any of the assumptions that they've made in their file system because we are not the engineers at Microsoft. We are trying to read their file system. But we can't change the way the design works, so you're fundamentally... If we think about these diagrams, you're going to choose which physical block within the file system you want to read into memory. And based on that, you can then figure out what's going on inside that block.

We've got several different types of block. There's the boot block, of which we have one. There are the FAT blocks, of which we have some number that is described in the boot block. There is root directory blocks of which again there's a number that are described in the boot block, and then after that there are the data blocks making up the rest of the file system. So to read a file, you're going to have to fundamentally do what we're showing on this slide, where you're going to get a directory entry from somewhere, you're going to find the first block, and you're going to chase around to figure out how that works. But to do the chasing around, you have to make this be in memory. If it's sitting there on the disk you can't get to it.

OK, so if we look at the code I've given you. Let's start in the header file. So I've given you a few different data structures. This one I've given you because it is the C implementation of a structure that will align perfectly with the byte by byte layout of the bytes making up a directory entry in a Microsoft file system. So the reason that I'm asking you to only hand in the fat12.c file is in part because if you were to come in and change this structure, there's no way your assignment could work because you've now broken the format that Microsoft requires you to have. So you can't really meaningfully make changes in the header file or in the other parts of the code, because it needs to conform to Microsoft's plan. So this will tell you once you've loaded a particular entry how to interpret the bytes of that entry, provided you've got a pointer pointing at the beginning of a fat12_file_system_dir_entry type structure that is in the right place.

OK, so you've got a block you loaded from disk. You have to figure out how you're going to get these entries to line up.

OK, this structure I've set aside for you because this gives you a place to store all of the information you're going to need to keep track of the fat12 file system that you're reading. OK, so when you plug in, you've probably noticed this. If you've ever plugged in a USB key to your computer, when you plug in the USB key, some stuff happens. You got one of those keys that lights up. You'll see it blink a couple times. And then if you yank it out, your computer gets mad at you. Right? It tells you it wasn't cleanly unmounted. What happens is when you plug it in, essentially the device driver is populating a structure much like this to say, "OK, I've got a new file system attached. What do I need to know about it?" and so it's got a set of values to talk about that file system. Especially if you're writing to that file system, which I'll note we're not doing in this assignment, this assignment is all about reading.

Right.

If you write to it, then the changes in the metadata and the changes in the actual data blocks need to be saved out to that file system before you're allowed to yank it out of the computer. That's why they have like a right click and eject. Or if you're under Linux, it literally says Unmount. That's what's happening. So you're going to mount your file system by essentially populating this structure, so you can say I can talk about a given fat12 format file system.

OK. Where do we get some fat12 format file systems? I've given you some. So these are literally the block by block copy from a physical device that was formatted using the Microsoft format to make a file system. OK, so I just simply copied them using a direct block by block copy into these files. You can then access these files. This is what's happening when we talk about mdir, mcopy, etcetera.

OK, so if you have a file system, Microsoft FAT compatible file system in a file called small_files.fd0, then you can call the mdir command to say I want you to look at this image, that's what the minus i is. So it looks in this file and it will literally tell you what is in the root directory. You can copy things out of your image using this format. So what this is saying is "I want you to copy" – so mcopy is a utility provided for you on Linux. You're doing your own Linux. You can install it as part of the mtools set. You'll get mdir, mcopy, mdelete, all the things you might want to do.

So this is saying out of this image, this file system called small_files.fd0, they're using this weirdo notation, but all they're saying is that 2 colons means I'm talking about something inside that file system.

And no colons means I'm talking about something out here in my main file system that I think of as where I'm doing the work. So all this is gonna do is it's gonna go into the formatted file system, small_files.fd0 and look and see if there's something called jabber.txt. And if there is, it will copy it into the current directory and then you'll have a new file called jabber.txt which is going to be an exact byte for byte match for the file, the real file inside of the file system small_files.fd0 except it will now be stored in your home directory in the ext4 file system that we're using on Linux.

If you want to put something in, it's just the other way, so here is taking main.c, it's a file, no problem I can. And copy it into the new file system by saying I would like to put it inside the file system and I'm still calling it main.c so if you do mdir you'll then see that main.c has appeared as an entity in that file system.

You can see that I've given you a number of these including one called blank.fd0. What do you think is in blank.fd0? Anyone?

So in a way, nothing, no files. It is formatted, so it still has a FAT table. It still has a root directory. It's still perfectly set up. It's just what you get if you just format a disk. There are no files on the disk yet. It's for you to put things in. So you can put things on these discs. You can put any file you like. So if you want to test your code, you can make a file with some contents that you would like to see, and you can use the mtools to put it into this disk and then you can go find it using your code because you're going to say I stored this as, I don't know, test1.txt, and it has 132 capital A's in it, so I should be able to go into the root directory and find test1.txt and it should point me at a data block that has 132 capital A's in it and you can make them as long or short as you want, because they're just files, so I don't need to give you a bunch of fancy tools to make files.

I have given you one not very fancy tool to make files, so write_data.c does is it says make a file made only of one letter repeated a certain number of times. So if you'd like a big file that is filled with the letter Q just so you can chase through all the blocks and make sure, yeah, these should all be the letter Q. This is what write_files write_data.c is for. We'll just make a just write out a bunch of whatever letter you want.

File. Yeah.

OK, so that way you can make some files with whatever content you want. You could use a text editor. I mean it's just a file and then you can find them in your file system. So everything except blank.fd0 has some files in it. This one, which I hope is implied by the name, is a broken fat file system. The rest of them are perfectly valid. You can add more files if you look up mdel you can delete files so whatever you want to do to adapt to the file system, you can change it and then bad.fd0 intentionally. I gave you one that was broken. The FAT I believe loops around on itself so that you can see what your code does because if there's something busted. We're not going to test your code for busted things, but I wanted to show you that you can actually in fact explore what happens, because you're going to have code that you can print arbitrary things off.

OK, the only other thing I wanted to be sure was clear here. So there's a couple of other utilities that you may find useful.

Hexdump.

Will print out the contents of a file as hexadecimal numbers. If you say hexdump and then minus C for apparently canonical mode, then it will print it out. Sorry, my window is I think 2 characters too narrow. Oh no. It'll print it out in a form that will look pretty familiar if you use the memory debugger from assignment one. I intentionally made the memory debugger from assignment one to do the same format, so this is simply the offset in hexadecimal. These are simply the values of the bytes in hexadecimal, and here they're reproduced in ASCII or a dot if it's not a valid ASCII character. So if you have ASCII characters in your files, you will be able to see them if you hexdump out the contents of the disk, and if we keep going, you can see there's stuff in various files, including some capital Q's and some lower case A's, and so on.

OK, I also, possibly confusingly, mentioned a different dumping utility. And the reason I mentioned both of these is they're good at different things. So hexdump is really good at giving you simultaneously, here's some hexadecimal data so you can just see them byte at a time. Here's the ASCII. But you don't necessarily want hexadecimal bytes, you may want other sizes. So if you want to look at things by length long or by length short instead of by length byte, this utility will do that for you and also do the right thing so that even though you're on an Intel platform, you're going to not need to worry about endianness, and you're going to be able to see the value that you want to see if you want to talk about, for instance, everything as a long. The -x here is to say, do it in hexadecimal. So now I see all of the values as longs, which are of course big, so it's doing them in a large amount of data at a time. Here we are seeing the same value as short, so you can see its character and then under the characters telling you what these are as the longer format. May or may not be useful to you. Some people just want to work bytes of hexadecimal and ASCII. That's perfectly fine. I've just given you these so that you can see what is going on.

OK. Any questions? Does everyone see what we're trying to do in Assignment 4? OK. So you're making a utility that you can read FAT format file systems. I thought you'd appreciate the FAT format one because people didn't seem particularly excited about interpreting the ext4 tree, so this is a linked list. It's a lot simpler. That's why we're doing the FAT format system.

I will say if you look at our file system notes, we talked about the fact that FAT 12 bit FAT goes from tiny up to still quite small. Above that, at 16 bit you can take this code and adapt it to the 16 bit if you want. It's not hard. You'll find that the FAT table entry reading code which does a lot of work to get only 12 bits out of the table, well, you need 16 bits, which is an even number of bytes. So that actually gets a little bit simpler. 16 bit will then take you up to any FAT up to 16 gig. If you wanted to do FAT 32, you could do that again. mTools is smart enough that it reads the FAT size out of the boot block and it just uses the appropriate size. I'm not expecting your code to be able to handle arbitrary sizes of FAT table, we're just going to do 12 bit. Questions?

Yeah.

Because the files are smaller and I didn't want you to page through 100 million lines of output basically. I mean, if you can make it work with a small file, you could make it work with a bigger file. I mean, there's nothing, the math adjusts slightly, but I'm sure that no one wanted to page through 16 gigabytes of data on the monitor to see if they could find their data block. Right, cause remember you're not, you are not able to choose which data blocks it's going to use to store a new file. mTools will do that itself. So whether it decides to store it at the beginning or the end is entirely up to mTools. And so therefore having a giant file system seems not like a good thing.

And I guess part of me also if we look at the code to do the pack structure, I also wanted to make sure your had an example of what this sort of thing looks like. Because they didn't want to waste 4 bits out of every pair of bytes. So they have shoved 2 12-bit entries into three bytes, so the low or high 4 bits of an entry comes from either a byte to the left or the right of the rest of the entry so that they can get as much data in that space as they can, and I wanted to give you an example of a packed structure because this is a very useful tool for you to be able to think about in future. Like if you have things that are smaller than a byte or smaller than an int, you can just pack them together if you remember where they put them. It's the same deal as we saw in assignment one, but you're not allocating blocks of multiple bytes, you're just allocating blocks of multiple bits inside of essentially a bit string. So that is I think, a useful thing for you to have an example of there.

And then the math looks marginally challenging until you remember what the bitwise operations are doing, so this is just logical bit wise and and shifting. So nothing too magical.

So I did have one question at one point, so bcopy and memcopy are very similar. Historically, bcopy from comes from the Berkeley development people. And memcopy comes from the System 5 development people and they both became popular before they standardized. So they're just both in the C standard. Weird. This is copying bytes. bcopy. The other one's copying bytes of memory, thus memcopy, so it doesn't really make much difference which one you do. But bcopy isn't meant to be in any way confusing or surprising in there. OK. Any other questions? Yeah.

Sorry, the 12 that 12 represents the total number of blocks?

The FAT in 12, FAT 12 represents the number of bits in an individual FAT entry.

Would that be points to a block number though, right? So.

Yes. So it influences what the maximum addressable block is, but the actual maximum number of blocks on the disk is going to be a physical constraint of what disk you bought.

So it's just every index from the FAT that is 12 bits?

Exactly. And so this is why, if we look at, let's say this diagram here, this is why above a certain size, you just have more blocks than you can talk about in a 12 bit integer. So you have to make your integer bigger. So that's why they went up to 16. But by the same token, if you've got a small file system, you don't want to waste all your space with the indexing. So if you can talk about it using a 12 bit number and you have code that can deal with its 12 bit number, then you're going to format it using 12 bit numbers because you don't need a bigger number to talk about blocks that simply don't exist. If you were to be talking about a different disk. Right? And that's why they move up to 32 bit FAT for really big disks.

OK. All right, so I wanted to then wrap up our file system discussion today.

So last day we talked about having a physical block buffer cache for file systems that are physically attached to our machine. So our kernel is going to allocate a bunch of mbuffs to hold the metadata. This is all the indexing data. So for our purposes here, this would be the FAT table and the root directory. If it was a Unix machine, this would be all those indirect blocks, inodes, etcetera. They go in the physical block buffer cache and they're specific to whatever type of file system you have, because if you've got a NTFS file system, you're not going to bother setting aside storage space for inodes because that file system doesn't have any inodes. So why would you allocate for data that didn't have to do with the file system you're talking about? So on a file system specific basis, you're going to have some kind of data in the physical blocks. The specific data that is there is based on that file system.

On any type of file system, however, we have this concept of the file. So we have this fixed size, or perhaps a better term is known size, list of bytes in a sequence. We've got it stored on our external device, but it's still just effectively a big string. Right? If you open a file and you read it from beginning to end, one byte at a time, it's very similar to traversing a string. We now know that the length of the file, unlike a C string, the actual length is stored in the metadata so that the directory entry tells us, yeah, this file has some number of bytes.

So then as we walk through our memory, we know when to stop. The logical block buffer cache simply stores the data blocks making up our file. So if we go back, let's say to the FAT example, then, if we were to read any of these blocks down here, we would put it in the logical block cache. It's logically part of some file.

So if we look at the other things that are in here, we see that we get logical blocks associated with network file systems as well. As so physical disks can hold files, we need the data from the file available to our kernel, so we put it in the logical block buffer cache. Files coming from across the network, we don't care what their metadata is important on that farm machine. We don't see it. What we see is that it ships us the actual logical blocks.

So when you're editing in, let's say, one of the source files for Assignment 4, you don't really care what file system the actual server holding the disk was formatted with. All you care is that you can look up a file by name and get the file data blocks and modify them if you wish.

OK, so the file block data itself, the file data goes in the logical cache because it is part of the logical structure we call a file. We know that physically it's stored on some disk and it may be in some kind of scrambled eggs order. We don't care what the order is logically, because that's encoded in our metadata containing things like the indirect tree or our FAT table or what have you.

So we can think of a logical block as being associated with a given file and an offset. Right, the file is simply made of a bunch of logical blocks. So the first one is logical block 0, the second one is logical block 1, etcetera, etcetera, etcetera. We put them in our physical memory so that we can give pieces of them back and forth to our programs as they read and write. So read and write are going to either take a bit of data from a user program and put it into a chunk of our logical block in our buffer cache in our kernel, or if it's a read, we're going to take a chunk of the logical block in the buffer cache and copy it into the user program. If the block isn't in the cache, that's when we have to do all that work to figure out what to do.

So last day we ended up talking about reading. So if you want to read, you have to know where you are in the file. The file offset divided by the block size of your device tells you the logical block number in the file. We look for it in the cache. If found, then it's just a memory copy operation. If it's not found, we have to make it exist in the logical buffer cache. That requires allocating space, just like we were talking about when we were talking about paging. We then schedule a block read using the actual low level device driver, so it's going to need to go and look up the metadata to figure out what disk block we're in. If it's not valid because it's a hole, we're going to fill it with zeroes. If it is valid, we're going to have to issue a disk read to the controller, say, go get me physical block, yada yada yada put it in this buffer. We then are going to go into a block mode, so we're going to wait, which means we're no longer scheduled, we get taken out of the run queue, somebody else gets scheduled. Lots of wonderful things happen in the kernel. Eventually we get an interrupt. So the kernel is interrupted that says this device wants your attention. The attention we look at what the disk device is passing back to us and it's saying, yeah, that IO request of which there could be hundreds in play. But this specific IO request is done. I put the data in the memory where you wanted me to, so then the kernel can say "Hooray, I can copy the bytes user wanted out of the logical buffer that I told the IO system to put the data in and copy that into the user program." Now the user program can be scheduled to run again. Gets back in operation. So far so good. Yeah. Come up there.

Yeah. So for that that is totally different computer. We want the planning to go down that road and this part of the fundamentally recognized a lot. I think you're going to contact your computer and say hey, file XYZ.T block 12. And it would then respond by giving some work and finding that that whole block that you did the network respond with the message in it where the message contains the actual response data.

It's a good question, but yeah, we wanted time to unpack that this term unfortunately. OK. Any other read related questions?

OK, so writing obviously is an important thing too. Writing and reading are quite similar. So again, we were at a given offset in the file. We can calculate the logical block number of the offset, which could potentially be after the end of the file. Right, we seeked off somewhere distant into the file, there just may be no block there, or we could be appending bytes directly onto the end of the file, and we have a new block. So you calculate a block number. Again, we look in the same logical buffer cache for that block. If it's not found, well, then we have to allocate a buffer for the block, just like we did for the read.

If that does not correspond to something on the disk because we're either in a hole or past the end of the file, then we fill that whole block with zeros. If... Yeah. Then if the write will not fill the whole buffer. OK, so we've, the user has given us a few bytes, but our block size of the disk is bigger than that. If the write won't fill the whole buffer but the disk block was valid, so the logical block is on the disk, but we only have a few bytes to go into it. Then, before we can put those bytes into it, we need to know what bytes are there already. Right, this is the situation that you're now in the middle of an existing file overwriting some bytes. You open up your C source code and decide that something on line 12 needs to be renamed.

So you're going to have to read in that block, because we already know we're only down in this if statement because it wasn't found. So we read in the block. So essentially then we just go to the read routine. We do all the work of the read routine. Now we know we have either a fresh block that we filled with zeros, or the correct data from the disk, or the partial block we're doing. So then we can copy the user data into our logical block buffer at the right location. Mark it modified. Now we schedule a write. So we say to the I/O, OK, I want you to take this and put it on the disk. So you're going to have to do the same metadata look up that we did for reading or we're going to need to adjust our metadata to incorporate the new block. Save our metadata, save our block, right? And then we have our block safely stored on the disk.

OK, so writing is a fair bit of work. I think everyone would agree. OK, we have a lot of work like this. We may want to save some of the work because if you think about the way people write to files frequently, you write a few bytes at a time at the end of the file. Right? So you've all written programs where printf "Hello", printf "72", printf "i = 5". If these are all going at the end of the file, you don't want to be doing all of this work for every one of those tiny updates, because they're like a handful of bytes. And if your block size is 16 kilobytes, there's going to be an awful lot of tiny updates. And if every one of them has to do all this IO work, that's going to be a disaster.

So we can do different things. One of the things we can do is we can have different scheduling strategies. This is actually controllable, so there's a utility in in the Unix land called fcontrol. I forget what it is on Windows, but there's a similar one where you can actually tell a given open file descriptor communication whether it should or should not be synchronous. So synchronous says you'll issue a write and your process will not get scheduled again until those bytes rest safely on your external device. Right, on your disk. This is if you're writing some safety critical system, databases, all these sorts of things they want to make sure the data is safely stored on a disk. Banking is particularly interested in this. If you take money out of an account, they don't want a crash to mean that the money didn't actually get decremented properly. So synchronous says I'm gonna let my software run really slowly if need be, but the data needs to be pushed out to the disk before I get to do anything else, I will wait for that to happen.

Asynchronous is exactly the opposite. I throw it out there and I don't even pay attention. Hopefully it lands. But I'm going to move ahead and do more computation as soon as I, the program, have written it to the logical buffer cache. And I don't care how long it takes to go out to the disk. It'll be faster, but it's much less safe.

Finally, there's a thing called write back. Let's talk about that in a second. Or write... delayed write... right behind. Sorry, I will update that slide.

OK, so the last thing I want to talk about is some IO tricks to try to make this go faster and decrease number of disk access requests we have to write or wait for. First trick is to notice that although we convinced ourselves that Unix file system tree strategy was a good one, in part because it supports random access, you can jump around in your file and it is relatively quick to do so. Most programs don't do that. Most programs, when they're processing data, process the data in order.

So if most programs are going to process the data in order, we could guess that the program we're trying to run is going to do that. And so it says, well, I can probably guess what block is going to be needed next if we're sequentially moving forward in the file. And when I have some time. Right? So when you're down in the IO subsystem in your kernel, if there aren't a huge number of read requests, you might just issue some more read requests for that program. It hasn't asked for data yet, but you're assuming it will need it.

So if you can pre-populate the logical buffer cache with file blocks that that program is going to need, it will appear to run faster. Because then when you enter the read code, the answer to the question is the block in the buffer cache is yes. So then you just copy it from that block into the program and the program appears to run much faster than it would if it had to sit and wait for the IO to finish to get the block to come into the buffer cache. So this is a very effective trick if you have enough memory rather to play that game.

So if your memory constrained, you don't have the ability to say I'm going to have a lot of spare blocks hanging around in memory just in case someone wants them. If you're IO constrained, if you've got lots of IO events going on, this could actually make things worse because you're scheduling more IO and you're not guaranteed anyone is going to ever need it. So this is a game to play if you've got, if your number of IO requests is relatively low and your available memory is relatively high, you can pre-populate the buffer and then it will make the program run faster. Which can be to everyone's benefit because maybe the program then finishes its work, gets the heck out of the way, deallocates all the memory it needed, stops making IO requests and everyone else can live a happy life without that program trying to use the resources.

OK, so right behind is either fully asynchronous. Oh, that's why I had the two here, so delayed right and asynchronous are slightly different. Right behind says schedule the writes and just assume it's going to work. So you schedule the write and then you go back you run the program. So you schedule it back in the write queue, total asynchronous. If there's a disaster, all the data that didn't go to the disk will be lost.

Delayed write is an attempt to combine several writes into one, so we can do both of these. Delayed write is simply we notice that multiple writes are happening inside the same logical block. So what we do is we wait until we've seen all the updates across the logical block. Again, this is easy to predict if they're all moving forward through the file. And rather than scheduling every update as a separate disk write, we only schedule an update once the block has been filled. So we generally implement that by noticing whether you've crossed a block boundary and also with a timer. So that you don't wait forever. If a certain maximum time passes, then we schedule the write anyway, so all we're doing is we're taking lots of little writes in the same block and we're updating them in memory only, until we think it's a good time to schedule the write of the full logical block.

If this happens, one of the things that is a benefit if you think about the things in your directory information, so your inode or your directory entry, one of the pieces of information there is the file modification time. Even every single update that you make to the file changes the modification time. So if you're writing to the file line by line or byte by byte, in theory it should write a new modification time to the metadata every single time you do that.

What we typically do is we say, well, I'm going to take the updates in my in-memory version of my inode, but I'm not going to write that inode out to the disk every single time. That would cost a lot of additional writes, and that is going to be very slow. You can also turn off some of the utility, some of these piece of information there.

Sure. And #2, if you're running on a on an SSD, you're actually going to save your disk life because you can only write to an individual block on an SSD a certain number of times before it says, yeah, I can't take anymore updates. If we have time at the end of the course, I'll take you through that whole algorithm, but fundamentally every block on an SSD can only be written a very limited number of times, where for most the blocks it's around 1024 times and then they've got a little bank of high write capacity, expensive part of your SSD and the SSD actually shuffles some data around to try to make work well for you, but you'll save a lot of your SSD lifespan by not saving your access time on that device. OK. Questions on that?

So although we have more slides in the IO deck, I want to make sure we talk enough about security to hit the big points in security, so I'm going to leave the IO discussion here on page 43. If we do not have time to come back to it, then that is all that is going to be on the exam is up to page 43.

OK. So then security is the last of the big modules we're going to talk about in this course. So operating system security is of course a large and important topic. If we have any kind of modern operating system, you generally have multiple users potentially using the operating system either concurrently or consecutively. And one of the things we want to do is we want to make sure that none of those users can damage the work of other users, so we have to protect the system from accidentally being damaged by the users. We have to protect the users from each other. And then we also have to protect the system and the users if we're on any kind of network based set up. We want to protect them from nefarious outsiders who may be trying to attack us or just incorrectly use them.

So if you've got anything you care about as a personal, is something you personally own, that obviously you don't want someone to come by and break it, and the same is true for the important data you have on your computer managed by your operating system. So in order to limit this damage, we need to think about who and what gets to access different pieces of data and different operating system resources. Because all these things we've been talking about, like scheduling and inode IO management, the buffer cache, if people could just play randomly in there, things would get really screwed up really fast.

So we need to make sure that the operating system not only is in the job of trying to give everybody the resources they're trying to ask for as quickly as possible, but is also making decisions on whether that resource requests should be served at all, because maybe that would cause damage.

So if there's damage, if there's system damage. So it's misconfigured, or there's some kind of physical abuse, someone throws coke into your keyboard. Obviously you're going to have system malfunction. Resource unavailability, this can cause impeded function for everyone's deadlock. If resources are being hogged, then it is going to be very difficult for other processes to get their work done.

One of the reasons I gave you the shared memory assignment where you're doing shared memory inside of a program communicating with its sub programs that are created by fork is because then the system can know that that shared memory needs to get cleaned up if that program dies. There's another model of shared memory where you create it with a name and then some totally unrelated program can say yeah, I want to talk to the memory that has that name. And you can both talk to it. That works really well, except if it never gets cleaned up because the problem now is there's no cue to the operating system when anyone's done. So it's quite possible for a given machine to have all of its shared memory handles used up because someone just accidentally created thousands of shared memory blocks and forgot to release them. Now you have no handles left to talk about any new ones. Maybe there's unwanted data. Or sorry data access or someone's manipulating data that they shouldn't be so then you get loss, corruption. Someone stolen your data. These are all bad.

OK, so how do we protect against different types of attack? First consideration. We do not let anyone at the console of a computer if it is going to be dangerous for them to do so. There is a reason that you do not have keys to the basement to the vehicle services building where the linux.socs servers physically are, because then someone could come in and start doing ridiculous things right. So only this half of actually CCS are allowed in that building. I can't even go in there and accidentally delete your files.

OK, so physical considerations. We make sure that important computers are behind locked doors. No one's going to dump Cola on them because no one's allowed in the room with the cola. But that leaves us with a whole bunch of software attack possibilities. These happen all the time. I forgot to actually update this slide, so these are a little out of date, but all of these things still happen, so there's three different main strategies.

Strategy 1 is what we call a virus. So a virus works by being delivered as part of a completely legitimate program where the program got modified. And the modification now has this malicious code. So your buddy gave you a virus because they thought you were... they were giving you some cool program, but it got modified. We typically track viruses by literally knowing important byte strings in them. And then virus scanners look for programs that have that quote fingerprint and start reporting. That looks suspicious cause I have in my database a virus that says it has that pattern. This program has that pattern. Do you want me to remove the virus portion of the program? So perfectly valid software gets changed. That's the earmark of a virus.

Trojan is similar, except it was custom created to be a problem. So they're what they've done is they've made a program that will do something terrible to your computer. But they've done it not by taking a valid program and hooking in a little bit of additional code. They've constructed the program straight up and then try to convince you to put it on your machine. OK, so the all the various app stores are very interested in tracking attempts to distribute Trojans through the app stores. It happens. You'll occasionally get a security warning from whoever your phone or or a tablet manufacturer is saying such and such, and you downloaded and installed such and such a software from here we've determined this is a Trojan, though sometimes they use the word virus incorrectly. You should remove it because it's stealing your secrets or doing something.

Yes.

OK, a lot of Trojans, a lot of viruses. The reason they want to do this in the 1st place and there's some people who just do it for kicks. I got like my code to run on someone else's computer. But the big reason people want to do this is because they want your processing time. They want your processing time either for mining cryptocurrency or, sadly, for distributing other terrible types of data. Child porn in particular is very interested in viruses and Trojans because that's where they store their data on your computer. This is one of the reasons we want to avoid that.

OK, the main way we keep viruses and Trojans off our machine is through education. Don't go and download something from "Fred's Super Great Warez" and put it on your computer unless you have a pretty good idea of who the hell Fred is because they're trying to sell you something. Be sure that you know what they're trying to sell. This is fundamentally what the app stores are trying to provide to you is a third party screening to say we only put things in our App Store because we think they're legit. So if you're going somewhere, source forge, GitHub, whatever you're grabbing some code on your own, you're on your own. You need to be making some careful decisions about whether you trust that vendor. That's where the Trojans come from. And then the viruses they're hooking along with something else.

So we've seen a big decrease in viruses, but still a lot of things like there's still word macro viruses and so on where you get a word file from someone and it has some crap in it. And now it's stuck on your computer and being distributed with other files you send. We haven't seem to have closed the door and most of the e-mail viruses, but think about what you're getting and what you're running because that's how you can compromise your system. Which leaves us with worms.

OK, so a worm is a piece of software that replicates itself with no user intervention. So this is particularly interesting because no matter how well we educate the users, that can't stop a worm because the users aren't even involved. So there was a book called Shockwave Rider where they talked about this worm, which was a program that would replicate itself around network. There was a guy called Robert Morris in 1988 who thought it was kind of cool that software might be able to do that. So he wrote one. He got an if statement backward. So his if statement was supposed to say only infect the 8 machines in the lab. What he did is he said only infect the machines that are not in the lab. Within a few hours, he took the entirety of what was then the Internet down.

So the idea is that you have one program that is going to try to infect some other running program. So it's a program that is on the Internet and will respond to messages coming in across the Internet. It will copy itself into the other program. Right? So this sort of... you've probably seen all this discussion about zombie ants with viruses taking over their body. There was a whole set of movies of this protocol. So something infects it, the brain gets replaced. You've got a new thing in the body of the old thing. This is basically what's happening. So all of the running programs that have been taken over and turned into the pieces of the worm, you can think of as in a big chain across the Internet and that's why they say worm.

Now there it must not be super clear on the worm, because it actually stands out. So instead of each segment being attached to the next segment, each segment could be attached to thousands. So it looks more like a big pile than one linear thing, but they still call it a worm. The way this works is because of a combination of poorly written programs written by people like you and I, and the way the stack works. So I'm going to come back to the stack because there's a cool thing we do now to try to cut this down.

OK. You think about the stack. You've got your stack pointer pointing at some location in memory. Upwards are things having to do with how you got here. So this is your old program counter, the parameters to the current function, etcetera. So below that you've got the data for the current function. So up here there's stuff having to do with the parent context. Down here there's the stuff having to do with the current context, including all the storage for your local variables.

So the way a worm works is you've got a program that is listening for data from the Internet. Doesn't actually have to be the Internet, but people are more interested in Internet worms than text console worms, where you would have to type in the thing manually. So it's listening across the Internet and it's going to accept data and put it in a local variable so that it can figure out what the data is. This is fundamentally how our programs work. You get some input, you look at the input, you see what's going on.

So if we think about how the stack works when you start a new function, you allocate your space for your local variables. You got your frame pointer, the program counter that you're going to use to return to is accessible relative to the frame pointer. This is going to be super important. Also below that is where you put the local variables. When you write the program, you are deciding how much memory goes for the local variables.

So if we think about how this runs. If we were to read a byte from our input, as let's say is coming across the Internet, let's say you're going to read a string. So you take a byte, you put it in your character buffer. So you've decided you've got a place to put the string. Your C program, you don't know yet how big the string is going to be. Right, we've all written a lot of programs like this, so you're going to need a buffer to put the string in until you can figure out how big the string is going to be, and then you can do something like, allocate some memory for the string, but until you see the end of the string you don't know how big the string is.

OK.

So we could just keep writing it in the buffer. Now notice the buffer's full. If we don't test to see whether the buffer is full, what's going to happen if there's more data coming? Where is the additional data going to go? It will overwrite the old frame pointer. Is that an immediate problem? Does this cause a crash? Not yet. We're just running the local function. We don't need the old frame pointer until we call return. What's happening now? You're sorry.

Got it. So that is that improper.

We've lost our ability to go where we used to go where we recorded, where we came from. Where? Yeah.

It might work, you might have. It is different in there. It won't work. Who is controlling what's in there?

The nefarious person is trying to overwrite memory. So if they put a key value in there and they've included some additional bytes. If they can figure out where in memory this location is, then they can make our function when it returns load this value which is the address of this data into the program counter. It's then going to start running this program. We've overwritten part of the stack with data that came from who knows who on the Internet. And exactly as you say, if they didn't know what they were doing, then there'd be gibberish here, and we try to return and we would probably just see a segfault. But if they knew exactly what they were doing because they know exactly what program they're attacking and can predict what offset in memory the stack is going to have to have when it's in that function, then they can work out where in the where in memory this is so they can hard code in the right value. This is exactly how the original worm worked. This is how all the current worms work, and so the problem is that we have this while loop that's just taking data from the Internet and storing it in a buffer and never checking to see whether the buffer has been fully consumed.

OK, this is called an array bounds write or a buffer overflow because the buffer has flowed over the other parts of the stack. So the worm writer needs to know what architecture you're on. I mean, you're not going to be able to attack, let's say, an Intel chip with a sun instruction sets. So if the instruction set miss is different, they're out of luck. But there is an awful lot of PC's out there. There's an awful lot of Macs out there. They have the same instruction set, so you're going to write a worm specific to that instruction set. Which solves the architecture and the instruction encodings.

Program design comes because they're going to attack a particular type of program. So maybe they'll attack, I know the Windows Web server, or maybe they'll attack a print server. So one particular program, particularly one for which they have source code is an open door to people trying this out. But anyone who has a particular architecture, let's say you go and you buy a Windows machine. Well, you can just look at what's in memory yourself. If you do the byte trace, you can find out what is actually happening when you run, I don't know program monitor or anything else. It's laborious and annoying, which is why people don't just write viruses all the time. But you can do it so they need to know the location of the frame pointer.

So one thing you can do. Make sure you're not allowing array bounds overwrites. This is still the number one way that we see viruses propagating in the wild, some programmer loaded data into a buffer and then didn't stop. So having a limited size read is absolutely critical. This is why if you try to use gets, it'll tell you, don't use gets, use fgets so gets simply reads a string into memory until it comes to the end of the string. It was put in the C standard library because of that, they don't want to take it out to the C standard library because some Paleolithic code written in 1972 will break. But they don't want to use it, so if you use it, you'll see an error message or a warning message come out. fgets takes the size of the of the buffer, right? The memory location of the beginning of the buffer, and the size. So if you try to give it too many bytes, it'll just stop accepting. So fgets returns. fgets is therefore not guaranteed to give you back a terminated string.

Yeah.

Because if you gave it 128 character buffer and it got to 128 characters, then they're just in the buffer and it came back. So you need to check that to make sure that you get the right thing and it won't obviously have the character return at the end either. You could use malloc. Does that make things better? Where is the memory from malloc coming from?

Load stack so you can still write into the stack.

OK.

Maybe it is below the stack? Where? Where is it? What? What segment is it? It's in the heap? And which segment is it?

Dynamic data. OK. And which segment is the dynamic? They've only got 3 segments.

Not that, so the text segment is the program instructions, the data segment is the one that contains the heap. Your string, table constants, globals, etcetera. But remember that there... this is... Oh no, this is security there. There's this huge gap in between the two, so it is pretty safe because if you start growing into the gap, what's going to happen? If you are writing and it's going up and up and up and we go into the gap, what is going to occur? If we come into this area here. Well, if you were to fill the whole area, but before we get to collision, what is? What is going to happen? What is true about the pages in the middle here? You're going to get a page fault because they certainly aren't in memory yet. Because they're invalid, but they're not in memory. But beyond a page fault, they're not going to be a valid part of our program. So not only are they not valid in the page frame, they're not part of our program, so we're going to get a segmentation fault. If you start walking up here and hit the gap, you're going to see segfault. This is when we want to see because somebody's trying to climb across the pond here to hit the data up at the top so it is safer if your buffers are all allocated from the heap because it is very difficult to climb one byte at a time through the program.

OK, so that's safer. There are, however, system protections because, oh, I shouldn't have actually closed that, because did I have on here? The memory addresses? So here this, it turns out, was a little bit, little bit of a FIB, I guess. Little bit of a slightly dated view of how this works. So until about 7 or 8 years ago, this is how we laid things out in memory. But viruses are such a problem. One of the things that we now do is instead of loading the text segment starting at null and the stack starting at the highest addressable address on your architecture and leaving literally 10s of billions of bytes empty in the middle, what they do is they do a what they call a dynamic segment location. So they just pick a random number and add it on to the start location of the three segments, ensuring they don't overlap. This is 100% to make it really difficult to guess where a given location on the stack is going to be because you have to guess the random number that was added to where you thought it was going to be.

So in order to defeat the viruses, instead of putting this at the highest possible address, they'll put it at some address in the middle. So this is why if you print out the address of your local variables in your programs, they're usually bigger or higher than these. They still put them usually in this order, but they won't start with several F's because it just doesn't put it up at the top to make it harder to find. They're basically hoping that somewhere in the forest they can hide the stack, and then the worm riders can't find it.

OK. But not every system has stack execution controls, and even the ones that have them, they're not always turned on for some reason that I still don't understand. Some versions of Windows have them turned off. You can turn them on, but they were shipping them with it turned off for a while during Windows 10. I'm like why? Why would you do that? OK. And a lot of programs are badly written, so I'm hoping to solve problem 2 by making sure you are aware of this and you then will yourselves not write programs with array bounds overwrites because you're going to worry about that and make sure that you ensure that your arrays are cut off.

OK. No prevention. We've talked about these basically. So physical considerations, we lock the door. Education for virus and Trojans. Worms. We can configure things so that... I guess I didn't talk about this part, so if you have a program that responds to data coming across the Internet, one of the things we can do is, as we just said, try to limit the ability of it getting compromised. But another thing we do is we set up separate users on our on our computer. So all of your accounts, you can have multiple different users. Most of them will support 10s of thousands of different users with no trouble. One of the reasons to do this is so that you can let software that you want to run that is likely to potentially get compromised run in a different user account. One that maybe doesn't have permission to do terrible things to your computer.

So we run all the web servers this way. They're under attack all the time. So typically when you run a web server, let's say Apache, Apache runs as user "www" who only has write access to a tiny directory where it can keep important files that it needs to update. Only has read access to the parts of your file system that are actually supposed to be on the web and has no access whatsoever to any other part of your file system. So even if you have a file in your home directory with all of your credit card numbers written in it, your Apache web server can't possibly give them out over the Internet because it doesn't have permission to read your directory. Therefore, if it is affected by a virus and turned into some nefarious version of itself, it still can't do that because it's still the same process with the same permissions. So let's talk about how we can use user access restrictions to do this kind of thing.

OK. If we have a user access based policy, the first thing we need is to be able to identify different users as different. So typically, even though you're logging in with a character based name, the way the system thinks of you is as a number. And it just has a table that says user ID, I don't know 5008 is in fact so and so because there's a record at location 5008 in a table that says, oh yeah, that user's name is such and such. Here's their login. Here's all the other information I need to know about them.

So with the user ID scheme, we can uniquely identify each user. This means that we can have different privileges for different accounts. So typically we have at least a two level privileged scheme. We've got the administrator or root who's allowed to do basically anything. And then we've got the other users who are much more limited. They're only allowed to do the things that it says they can do in their particular set up in their particular record. So we can then start attaching different processes and different events with the user ID. We can even sort out different types of, let's say as low as kernel traps. This user is allowed to call these system functions. This user is not allowed to call these system functions and you can do this by simply having a list of what things they're allowed to do or not do in their user ID entry.

We still want to have a super user, so administrator or root because the super user needs to be able to solve problems and the super user can grant and revoke permissions from anybody else. So we typically make it that whatever the super user account is is seriously protecting you don't want someone accidentally being able to become administrator. So as long as that's protected, everything else is pretty safe. This is again, I always feel whenever I teach this course that I really bang on Microsoft, but this is one of the things that they've done really badly for a long time that finally in Windows 11 seemed to be recognizing that not all user accounts should have administrator privileges because it's really easy to blow your machine up by accident if you have administrator privileges.

So you set up a user. You give them as little permission to do stuff as you can and still allow them to do their work, and then it is much less likely for them to accidentally cause a problem because they simply don't have permission to do the things such as change file system objects, I'll talk to different programs, use various software resources. And most of the operating systems will track when these important attributes have changed so that if you give somebody more permission that will go into a log file. So that later if you asked the question, how on Earth did the print server get the permission to reformat my disk, you can look at a log file. If it wasn't on the reformatted disk to say who gave the print server that permission to do that, we're going to log user activity.

So on all the Linux machines they log every time you log in, every time you log out. We have fixed resource allocation strategies so we can give people quotas. We don't use quotas here at the moment, but we have in the past, we made in the future if somebody decides they want to put 10 billion terabytes of space on one of the student drives, they'll probably pretty quickly get slapped with the quota to keep them from taking up everybody's space. So you can match who owns at every resource. And as I said, you can have quotas.

So to do that, we need authentication because we need to make sure that every user doesn't have the possibility of saying, "I would just like to be so and so now" because they have more permissions than I do. So just treat me like them. So next day we'll talk about authentication and go down the road of password encryption and that sort of stuff.

So next day is the beginning of the second last week of the term. My intention is to get through the core security enough next day and the day after. And then the last week we'll do review. So I will ask you to think about what questions you have for review, because it is much better for you to tell me what you don't know because I can tell you what I think you don't know, but that might not map to what you actually don't know. So use the time between now and a week Tuesday to come up with said question. I will put up a, maybe I'll get... I don't know whether to get you to e-mail them in or I'll put up maybe a course link for them so you can dump some questions there so that I can prepare some answers if there's anything particularly thorny, but otherwise we'll just do questions from the audience in the last week. Questions for today. Before we go anyone.

OK. Then we'll leave it there.