CIS3110 Class Transcript - January 10, 2025

All right. Welcome everybody. Here we go for class #2. So we've got our course outline now. Let me just bring that up here. So let's start there. So of course you have the calendar description. We talked about learning outcomes last day. We have a large number of GTAs. So we have a staff of eight. They're all available for consultation through office hours and they'll be running the labs. I will say please email CIS 3110 for course related discussion because my email is becoming more and more untenable. On Monday I received 147 email messages over the course of the day. So if you email me directly, the likelihood of me seeing it is pretty close to 0. Thank you. Yes, microphone. Come on. Roar. Yeah. All right, this one's working. The other one is not. I will concentrate on the podium microphone. That means I can't wander. Alright, is everyone listening? No, they were... So mic seems to be dead. OK, I will just concentrate on podium right here. Let me move my stuff over so that I can see while speaking so you can hear me better. Now, good. More or less. I know the units are up. Is that as loud as this thing goes? OK, yes. What is the coverage? What's that, sir? I think you said what is the coverage of the lab 1 quiz?

That.

I don't know what you mean by that? It is 100% the activities of the lab. Oh, OK. Yeah. So if you've done the lab there are, I guess you haven't seen the quiz. There are three very simple questions which I believe the answer would be quite obvious if you've actually done the lab. But yeah, the way to prepare for the quiz is to do the activity in the lab and then the quiz is really just to see that you actually did the activities and didn't just get to the end, yeah.

Online.

So. Let me come over there so I can hear you better.

But the.

Current.

Online.

OK. I think you're making more of this quiz than there really is. So my intention and the plan of the, so let's back up. I was going to get to that later, but let's do the lab right away. So the reason I have this lab is because last year when I taught 2520, there were a remarkable number of people who submitted code that did not work on Linux.soc and later said they had written their work on Linux.soc. But as it turned out, had not done this, had only thought they had done this. So clearly there is this point of confusion. It arises through VS Code. VS Code is a great editor. There's a lot of interesting and wonderful things, but unfortunately two of the things that it lets you do is:

Option A: Log in via SSH to the Linux.soc server, edit some code on the Linux.soc server and then it provides you a command line also on the Linux.soc server to run your code. If it's doing that, yeah. Things are pretty good. It's essentially synonymous with if you had a terminal open with SSH to Linux.soc and were running it natively there.

I would suggest if there's any confusion, to open a terminal, type SSH Linux.soc, run your code there, then you'll know what's going on.

The other situation that seemed to very commonly arise was opening the explorer, attaching to the server using SSH. It would bring the file locally. The student would write their assignments. It would end up on their local machine. And in some cases, the student would be looking at the terminal at the bottom that would say something like "C:\Program Files" and they would swear up and down to me that this was in fact the Linux cluster. If you have a "C:" backslash on your screen, you're probably not logged into the Linux cluster.

So because that was so much of an issue last time, and because I don't want there to be many long conversations with unhappy students who end up having code that isn't going to compile and run, and we're going to give it a zero. So the whole point of Lab 1 is just to make sure that everybody is absolutely sure that we're on the right server when we're running the code.

So I'm assuming that probably 90 plus percent of you are going to arrive at this lab and say "yes, I understand how the Linux.soc server runs. I'm going to open the file, look at it, compile it, run it, be done right," because you may be out of there inside of a few minutes. But I want to be sure that everybody has a good, robust tool so that they can tell whether or not they're on the right platform.

So this is 100% trying to set you up for success so that we'd avoid the problem of unhappy people getting 0 when their code doesn't even compile, because then I will say this. If you had your own Linux machine.

Question.

Either at home or in front of you now, there are many versions of Linux. There are many, many, many versions of the various tools. Make sure your code compiles and runs on the server where we're going to grade it. Because hearing "my code compiles at home" is not going to cut it as far as getting a working solution for this course. It's easy to do. You just try to get your code onto the right machine. Build it there.

I'll even give you a foreshadowing of one of the quiz questions. What tool do we normally use to get compilation to happen across a whole set of related C files in one project? There's a four letter word we use. It's a command that we type in. Anyone. Make, exactly. So if you're using some other configuration, please don't. You will use the make file that will get us to the end.

OK, so I don't know if that directly answers your question, but I'm sure you can see where we're going on this. This is not meant to be a big bait and switch. I'm not going to give you all a 0 on the 1st lab because

you couldn't prepare adequately for the first lab. I'm really hoping that your career so far has prepared you adequately for this lab and you will say "Why did he give me such a silly exercise to do?" But it's all in favor of trying to avoid unhappy unpleasantness when people accidentally submit code that doesn't compile.

We will do all of the coding work for this course on Linux under the Linux.soc environment, so you are presumably all very familiar with that environment already. We're going to essentially write code at the lowest level. You can write it in C to talk to, and in any cases build parts of the operating system. OK. Any other questions regarding the lab?

All right. So yes, please send emails here. Also, please don't attempt to email the TAs directly, that's just going to get messy and messages are going to get lost. We have a nice centralized ticket management system to make sure you get good answers. Everybody is going to read CIS 3110 email at least once a day during the business week, which means you've got nine people with eyes on the prize, you should be able to get an answer pretty quickly, even if for instance, I'm busy.

OK, I am required by the university to tell you when your class happens. You're already here. You presumably know that.

OK, the land acknowledgement, I completely forgot to do last day. We are and still are this week on land that is the land of the Mississaugas of the Credit. The between the lakes purchase between the British Crown and Mississauga of the Credit awarded this land for their use forever to the Haudenosaunee confederacy. And clearly they're not using this land in the same way that they were using it at that time because we've now built university wealth on the land. So we're on occupied land. It is, however, treaty land. We do have a treaty that governs what we do here.

There's been several treaties, but the one that always gets talked about here is the Treaty of the Dish with One Spoon. And I noticed that many of the land acknowledgments when I said, well, they acknowledge the Dish with One Spoon treaty, they don't ever really tell you what it is. So the Dish with One Spoon is an ancient treaty. It's a couple thousand years old. And it is about meeting under non-aggression and community and community use of a particular state.

So the metaphor of the dish, the single dish with one spoon is that everyone coming to use the land, everyone coming together in this place is sharing the land, sharing the resources, sharing everything that we have here. And the reason we're meeting with one spoon rather than a knife is because we're agreeing to meet cooperatively. We're agreeing to meet to advance everyone's goals, so specifically this treaty demands that we meet and we protect the abundance of what is here. We share the resources in front of us. And that we must respect each other, which actually sets a really nice table for a discussion of learning.

So in this course at this university, what we intend is that the people who are here will come together, respect and honor each other and share resources. So we are going to, as much as possible, attempt to

follow the terms of this Treaty and honor these obligations, while we of course, work toward reconciliation and for a better future for everybody. I've given you a very long list of reading if you want to read more about the history or the impacts of these issues.

OK, caregiving while learning. If people need to, we understand that your lives extend past this, the university environment, and we will support people in that if and when need be. This course is scheduled as an in-person course. However, as I posted on the course link site today, a number of people wrote in saying they're concerned about the norovirus situation that is developing in residences. I understand it is now progressed past South Residence. I don't know if any of you are still in residence. But South Residence is a particularly bad place for infection.

The way it is designed is there are air columns running vertically up each of those tower blocks. And with the common air flow, it circulates air among all of the apartments in a vertical column. So we've known for decades that somebody in a tower gets sick and all of their vertical neighbors get sick. And then those people come out and hang out in the common rooms and talk to their friends in the same hallway. And you have lateral transmission as well. So South is basically built as a disease incubator. During COVID, I understand they added some ventilation filters. But they never made clear how many, where they are or how they work, so I don't. And the fact that we still have this thing happening right now means there is definitely an issue.

I will say I've got a lot of friends in public health. To keep yourself healthy. The single most effective thing you can do is keep your hands clean, so getting into a habit of making your hands staying clean. Really cleaning them when you're at all, I'm sure what you just touched. That is a huge step forward. You can wear a mask. This is mostly a physical transmission thing, but if I have another first year course, if I go down, we're going to lose weeks of the term. So therefore I'm wearing a mask. You can wear a mask that is beneficial but it is primarily touch-based contact, so also don't share cutlery. Don't share glasses. Get a fresh clean water glass if you're unsure. That will help manage this situation.

We're all hoping that this is going to pass in a week or two. Norovirus is awful. I hope none of you get it. Bad GI situation from the day you get it until you are healthy again. But it is at least relatively short term. This is the same virus that is famous for propagating on cruise ships and in care homes because it's about enclosed humans together touching common surfaces.

That said, this is still an in-person course. I will be here, but as someone, a couple of people actually requested, I'm going to record the audio. Well, the video doesn't work very well the way they've got this set up because the camera is a million miles away, so you'll see a tiny little speck. You already have all the notes. I'm going to attempt to record this audio. I found the recording through this system doesn't work as well as this microphone, so I will give you whatever audio comes out of this so that people who don't want to physically be here while this is happening have access to that material. Anyone have any questions on that? A huge room with darkness in the back. I don't see anyone have any questions. OK.

Marking scheme and deadlines, so this is the only part that has really changed since the draft version was put up. There's a new departmental rule about how we can use Moss and various other tools. So the three assignments that are strongly OS related are now 10.5% and I moved down the amounts associated with the memory management assignments.

I hope you've all taken a look at the memory management assignments. It's essentially a pointer assignment because I wanted to give you something you could get your teeth into at the beginning of the term without waiting for us to discuss some of the more complex tools, so this one, assuming you've ever used a pointer before in your life, you can do all the things that you would need to do to work on this memory management assignment.

They're essentially building the kind of tool that is behind malloc. You're building your own malloc, essentially. So that will give you a tool that is managing the distribution and reclamation of the actual memory within a process. Then we'll go on and talk about semaphores, and then we will finally wrap up with a little bit of networking because although we have a networking course, it isn't a core part of the curriculum. So a lot of people would otherwise graduate without ever hearing about the networking tools. So we'll wrap up with a nice little assignment on networking at the end.

Rest of the course is the midterm and the final. It will be primarily, if not completely, multiple choice because we have a big class and it's going to be bigger next year. So apparently I need to go in that direction.

OK, here's the textbook. Silberschatz, Galvin and Gagne. Really, any version from the 6th edition on will support you well for this. As I say, you can get 6th editions online pretty close to free. There's been a bunch in the library. It's a great book. When I used it way back when I took this course, I hung on to the addition I had at that point. It was the 4th. For years, and it was a very valuable text. Then they made me start teaching it, so I started getting new additions. I no longer have my 4th edition, but it is actually a very good text. The new ones are roughly 90 bucks, but there are lots of used ones available. And as I say, I believe there is a PDF online if you want to look for that. It is a copyright violation, but I will let you make your own decision on that front. This is the sample lecture schedule.

As with all the courses, a number of you have taken courses with me before, we have a code of conduct. It is really important that everything we do here welcomes everybody and that everybody has an equal chance to participate. So please do participate. If things are going pear-shaped in any way, let us know. We will do something about this. In particular, harassment, discriminatory or derogatory behavior can't be tolerated, so if that is happening, we need to have it stop immediately. If asked to change behavior, we expect that to take effect upon being asked. If that doesn't happen, we will have to escalate through the channels of the university. Anyone have any questions there?

All right, there's always an academic misconduct portion. The key idea in academic misconduct is that there is no misconduct provided you are authentically demonstrating your proficiency with the material.

Here.

There are two ways this goes wrong. They can both easily be avoided. The first is the plagiarism problem. If you see a cool idea from somewhere else, so on the Internet, I don't know, from a textbook, there's some code you want to use. If you have an idea and you put it in your work and simply hand it in with then no acknowledgement into someone else's idea, you have committed plagiarism. You have included their idea and you're pretending it's your own.

Just like in an essay. If you have someone else's idea and you put it in an essay and you put quotes around it and you say this is somebody else's idea, you're not committing plagiarism, you're in fact doing what we want you to do. You're acknowledging that the idea came from somewhere else, and you're then presumably elaborating on or otherwise using the idea.

If you paraphrase the idea, so if you take somebody's idea and you write it up in your own words, it's still their idea. So you still need to cite. Now paraphrasing is common because frequently people will have a great idea, but they're not particularly good at explaining. So sometimes you have to paraphrase. Because you're going to say "so and so had this great idea. It works like this" and you explain it, but you're still saying it's so and so's idea.

There's this mistaken belief that if you paraphrase it into your own words, that you somehow become the person who made up the idea in the first place. And I think this comes from a particular type of assignment we give out in high school where someone gives you an essay or work and says "put this in your own words". Because they know who wrote the original one. They gave it to you, so you don't actually need to cite it in that context, because it's clear to everybody where the idea came from. But if you're bringing in an idea, and the reader won't know where it came from, just be clear where the idea has originated.

So if you find something in this or another course, so you want to use an algorithm or you want to use some tools, including them and saying "these tools came from this place" totally avoids plagiarism. If the assignment is "write your own version of this tool", then you hand in one you found on the Internet and say "I found it on the Internet" - no plagiarism, but you're unlikely to get a high mark if they told you to write this tool yourself, but you've avoided the plagiarism issue. So it's quite simple. If an idea comes from somewhere else, say where it came from. On that note, I'll take a... Oh yes.

Excited. Would it be like as in comments?

In the comments in the code is good. Yeah, you know, little readme file as long as it is accompanying the thing you're submitting. That's all fine, yeah. Going to come over so I can hear you.

Something. Sorry.

If we're talking about well-known algorithms. People will frequently cite bubble sort. Right at some level like, OK, let's talk about quicksort that Hoare invented in 1943, right. So we're talking about super common things like that. Probably no one would expect you to cite that. If there's any sort of questions, however, right. If you're talking about a tool that you're not regularly using, it never hurts to say "I got this from this link."

And probably the best litmus test to that is that if you can produce the algorithm out of your head without thinking about where it came from, you're not really going to have someone who expects you to cite where it originally came from, but if you're saying "I'm going to use the somebody or other algorithm here" and I'm typing it in from a textbook beside me, that's when you're going to say "I got this from this textbook, page 375."

That noted, I'll say if you're ever citing anything anywhere for anyone, please don't just give them a bare URL. The URLs are incredibly fragile. In my career, even something you think is common and well used as the Microsoft knowledge base has been completely reorganized about 6 times now. So if you just have a URL, it's useless if they decide to reorganize.

Right.

So URLs are always handy, but only if they still work. So yes, including the URL, but we also will want to see things like a title, and maybe an author, and preferably a year of either when the article was written, or at least when you accessed it, so that when you come back later and the URL is busted and you get a 404 error, think about what information you would want to have to try to find that working again.

Because, especially with things like the knowledge base, seeing like a 17 character long hexadecimal alphanumeric string where it's clearly just some kind of hash code, you're not going to be able to guess what that article is based on the hash code they put into the URL. So having actual text really helps. OK.

So then the only other thing I wanted to talk about in the academic misconduct portion. This is unfortunately started to come up more commonly, so we need you to demonstrate your proficiency with the course material. If something has happened in your life that is interfering with your ability to show the proficiency in the course material, talk to us about that before it is graded.

OK, so if you need a couple more days for a deadline, if something has happened and you can't do something in the expected way, right, this is effectively how accessibility services operates. Talk to us then. What we do not want to hear is after the fact "Oh, I only got an X on this assignment, but I really would like to have a higher number because I'm sad or because something's happened in my life" or anything else that is not related to the actual demonstration of the proficiency.

So we're happy to talk to you about different ways in which you can demonstrate the proficiency. But we can't just change grades. If the proficiency hasn't been demonstrated, it hasn't been demonstrated. OK, so if there's an issue coming up, talk to us in advance. We can talk about alternative means that you can demonstrate proficiencies that might otherwise be a problem. Does that make sense? Everybody see where we're going there?

OK, so these are all posted now on the course link site. Let us move back to discussing the tools around the processes in a computer, so just quick review. We talked about there being a CPU. From point of view of a running program, the CPU is where everything happens. Everything else is external to the CPU.

RAM is a huge ordered list of buckets of data we can bring onto or push to from the CPU. But the instruction running on the CPU can only deal immediately with the registers it has available, or it's going to perform some kind of load or store instructions to get data to and from RAM. This is going to be key in a minute when we're talking about how interruption of a running process works, when the data in RAM is shared between multiple readers, multiple writers, or both.

So physical RAM, this is what is all physically in your computer. Has this one relatively complex area at the bottom. That's where all of the IO ports are mapped to. So if you've got some IO device that is going to change based on changes within that physical device, that is visible to the CPU because of a memory-apparent memory location in that zero page just starts changing value when the IO device wants something to happen. The notification that a change has occurred is, of course, an interrupt.

But physical organization of RAM keeps all of that nifty stuff down in the first block, which we call zero page for reasons that will become apparent probably next Thursday, and above that we have plain old RAM where the way that works is you store a bit in RAM and unless you turn the power off, the bit just stays there. So later if you ask for it, it can be given back.

So by far the largest fraction of RAM is just this part I have done with the brickwork texture. It is just storage. All the values in it are going to have come from the CPU. And the only consumer of those values is the CPU. Our complexity is going to arise from the fact that although there's one CPU in this model, we'll talk about scaling that out to multiple CPUs in about a week. But if we imagine even if there's only one CPU in the model, different programs will be running on the CPU at different points in time. But the physical RAM remains the same.

So data in the RAM can be accessed by the CPU even though it might now be a different program running. OK, so the sharing of the data between multiple programs is going to be possible because the data is hanging out there in RAM, so if we swap the contents of the CPU for a different program, the CPU can then operate under the context of the new program accessing this RAM.

OK, so on the CPU we have these tools to keep track of a single running program, which we'll call a process. The program counter of the CPU tells us where in the current running process we're getting our

next instruction from. Stack pointer, frame pointer - they keep track of the stack data structure. Very important for that process.

All of the other things, the registers, the memory management unit. These are all key parts of the hardware available to a running program. Many of them we're going to have to keep track of in the context switch. So if we switch from one process to another, we're going to need to copy all of this data to somewhere in memory so that when we want to switch back, we can recover it to exactly the state it was in before.

OK, so the interrupt vector, a vector of function pointers indexed by the hardware ID associated with that device type. So you buy a physical computer. The physical computer has the possibility of supporting some set of interesting devices. For that set of devices, there's going to be an entry in the interrupt vector.

It may be, this is especially common on Windows desktop machines, or I should say really PC desktop machines, because you have cards you can take in and out. It is quite possible that the machine you bought doesn't have all the capabilities of that class of machine around the world, so you could have a machine that you bought that maybe doesn't have the RFID read/writer that is potentially possible on the machine. You'll just have an empty slot in your interrupt vector for where that hardware would show up if it was to be physically installed, so the interrupt vector is defined by the set of hardware that could be plugged into your machine as designed by the physical vendor of the hardware. Right, so so far we're really talking not about operating systems, we're just talking about the physical thing that is your computer.

OK, the instruction cycle we all know it rattles on, gets data from memory, increments the program counter, executes the instructions, rinse, repeats, everything just goes nice and linearly through memory. Unless you're doing a function call or you're doing a loop or a branch. Also in here processor status. If you're in supervisor mode, what this is going to mean is you're running the kernel. The kernel is going to be allowed to do anything.

Non-supervisor mode, you're only allowed to do user instructions. If we think about the design of the full running complement of the hardware, the kernel and all the processes, it becomes clear why we need supervisor mode. So the kernel needs to be able to do things like create and destroy processes. It needs to have full permission over all the parts of memory. But we don't want one user program to also have all that permission or it would be impossible to keep the other programs secure.

If you go back and you look at the early windows, windows started out as effectively a program under DOS. So early windows all the way up to Windows 95, it operated like that. It was just hoping that meant that program A over here didn't have a bad pointer, right, and stab program B in the heart. And that happened a lot. So I'd have people would say, "yeah, you can use Word, but don't try to run this game at the same time or your computer will crash." Because what has happened is somebody hit a bad pointer

and rather than having a segment fault, what they did is they just poked their buddy in the eye and their buddy fell over.

We don't like that as programmers. That sounds like a bad world to live in, and this is why we don't do that anymore. So we want to have supervisor mode that says the only program that is allowed to poke another program in the eye is the kernel. And then we spend a lot of time debugging the kernel to make sure that it's not going to accidentally poke a program in the eye, but is rather only going to destroy a running program if it has been asked to destroy a running program. Right.

So if, let's say, process three tries to write outside of the memory it's been allocated, this is when you're going to see a segment violation. And process 3 is going to be summarily removed from the CPU because it's not behaving very well. It's trying to attack other programs, probably not on purpose. But it still isn't playing by the rules and it would destroy the system if allowed to do what it is trying to do.

So supervisor mode is about entering the privileged kernel state. We go into the kernel state. We're then allowed to do anything. We can rewrite any program's identity. We can move data from anywhere to anywhere else. We can adjust any resource lists in the whole system. We can do literally anything that the hardware can do, but when we go back into user mode, we're protected. We can't accidentally blow anything up.

One of the things we can't do is read or write anything on zero page. It's all hidden from us. So we can't screw up the IO because we don't have permission to even look there. OK, so this is the way that we keep the processes safe and happy and able to do their own stuff. But of course processes need IO.

So as we talked about last time, this is handled through a trap. If a process needs to do something that only the kernel can do, the process causes a supervisor mode transition to occur by calling a numbered trap. This would be a trap like "I want to write some data." Slip. You're in the kernel. You're in the write routine. The write routine now has permission to copy to zero page. It's going to do all the hard work of making sure that the write goes to the right place. And eventually it can return to the user process once the write is finished.

Obviously, we don't want the whole system hanging around waiting for the write, so we'll find out that we'll do other things in the interim, but that's how we crossed this line here at the top of the kernel, going from user space into kernel space. We cross the bottom line from the hardware up into the kernel in the way we discussed relatively extensively in 2030. This is a hardware interrupt. Both of these, however, caused one running process to have to be swapped for a different running process.

In both cases, we went from something to the kernel. If we're doing kernel work and we get another interrupt, we're going from kernel to a different part of the kernel, but we're still going from something to something else. That's what we're going to call the context switch. Every time that happens, we have to save the contents of all of the information on the CPU somewhere so that we can come back.

That can happen in between any two instructions in a running program. There's no way the program has any control over, for instance, when your keyboard is going to send you a character. So therefore the program has to be able to react to being interrupted. It cannot prevent the interruption from happening. So we have to write our programs in a way that at any point after one instruction we might get switched out, a bunch of other stuff might happen, and then we get switched back in before we pick up at the very next instruction.

We will notice that here's talking about what goes into the context switch. We'll notice that only if we could say my timeline is progressing at a certain speed, I get to some events and then it looks like I was put in... I was frozen, I was put in, right. If you're a Star Wars fan, you're Han Solo in Carbonite for a while, you don't know what's happening, the world doesn't go by, you're just not there. And then you later come back. Right. The clock was ticking, and then all of a sudden it went... half an hour went bob. What happened? You're running again. Even longer went bob. You don't even know that multiple events happened.

All you know is that you called for one instruction to run, and now you're ready for the next instruction to run. But boy, it took a long time for that previous instruction to happen. We have to be able to come back to where we left. So that's an important thing about this graph that we leave here at time B. Switch up to something, but eventually we come back to exactly where we were, even if something else happened in the meantime.

Notice that we come back from the second interrupt here on the right. So interrupt at time D switching to the mouse. Running the mouse until time E. Switching to the disk. We don't go from the disk back to the process. We go from the disk back to whoever, whatever we were doing when we were interrupted, so an interrupt always returns to the same context that it was in when the interrupt began. This keeps things much more simple and therefore controllable, but the only way we get back to the process is if all of our potentially multiple layered interrupts have now all resolved hierarchically and unwound themselves. It's effectively a stack.

Right.

Right. We push something on, we push another thing on. We come back to where we were. Everybody good so far? Yeah. No, not yet. We're going to talk about that. It happens to be in priority order, but I'm not trying to say anything about priority here. I'm just literally saying we have three different things that we could be doing and we're switching between them. Good question though.

OK, so I think this is where we were getting to right at the end. So we can switch using context switch between processes. This is a hardware supported activity. We can, however, as Rocco was just saying, create it to be a priority order. If we can mask off less important interrupts. So what we're going to do is we're going to construct a system where we can say, OK, I'm going to go into the disk interrupt. The disk

is hungry. It needs lots of data quickly. We do not want the pokey slow mouse interrupting the super fastness, so we're going to tell slower devices, you're just going to have to wait. And then when the disk interrupt is complete, we're going to remove the mask and go back to the old model.

So this will mean that a mouse which is only able to submit its interrupts at a very slow rate because it's a very slow device, it's barely even going to notice that it had to wait. Because it's operating at such a low clock rate. The disk is going to be able to jump in, do its work and exit, and the mouse is only going to be marginally inconvenienced by this happening. OK, so because some devices are slower than others, we can tell them to wait a little bit for faster devices.

OK, going into the interrupts, we save all this stuff. Return from interrupt. We have to go back exactly where we were. So save the PC obviously, because we're going to need to know where we're coming from, save the mode. Load the program counter from the interrupt vector or trap vector using the appropriate ID. Have to save all the registers, continue execution in the new state, but we restore everything we saved, and reset the CPU back to the mode it was in before.

So not just set it back to user mode

So not just set it back to user mode because we might not have been in user mode. If we're already running as the kernel and we get interrupted, we switched to supervisor state, but we were already in the supervisor state. So when we finished that interrupt, if we went into user mode we would suddenly have a kernel that could no longer talk to zero page. And that would be a big problem, because you suddenly would not have access to all the tools you need to manage the system. OK.

So that's all well and good. Oh yes. How does it know to interrupt? Like that over like do they all have? I can hear about every third word. Have some ending that sends it back? Yeah. So, OK, you're about four slides ahead, but the short answer is that when you write an interrupt handler, you're writing what looks very much like a function. The only real difference is that instead of a return or return from call to give it its full name, we have a return from interrupt. It's just a slightly different return instruction. It is literally an instruction at the assembly level, but an RTI instruction, returned from interrupt instruction rather than recovering it using the stack. It recovers using a context switch. But literally at the end of what looks like a function that you've just written, you're going to have an RTI instead of a return.

OK, now if we have in RAM data that is shared between more than one process. And when we were wrapping up last time, I pointed out that, let's say if you're doing any kind of IO read or write, the kernel and the process that either created or consumes the data they're sharing them. Right, you've got two processes now, one the kernel, one the user process. They have a shared buffer. We're going to find out that there's another one between the hardware itself and the kernel to get data from the hardware side up into the kernel. And we need to make sure that in this interrupt land where you're doing your work and all of a sudden you can be yanked off the stage and then thrown back on sometime later that no one else can come in and play with what you're trying to do in this room.

Because you have no way of knowing that is gonna happen because you have no way of detecting that you've been interrupted. What we do instead is we have to protect the action you're taking on the shared data so that the interruption can't happen at all. OK so. I still have my timer set half an hour early. I'll fix that by next time.

OK, so in order to... If we consider this example in order to see why this is a problem, I think I walked through this last day and I maybe didn't make it totally clear like later got a bunch of questions about. So imagine you have two processes that are both going to manipulate the same variable I. They need to know... I don't know how many eggs are in a basket. Both processes need to know. Both of them are going to do something where they might increment or decrement this value.

If both processes are running willy-nilly with no coordination between them, one of the scenarios that could easily happen is that one process, let's call it process A. Loads the value of I out of RAM. It goes into a register on the CPU. Let's say the value is 45. We've got 45 eggs in our basket. OK, we're trying to

increment this. So we've got the value stored in the register is 45. Context switch occurs. 45 is copied from the register to where we're putting all of our context switch information in RAM.

Maybe even a bunch of other things happen on the register or on the CPU router. Could be 45 different processes come in and run in between these stored value and something else is going to occur. But eventually, let's say your friend process B, with whom you're sharing I, is going to start to run. If process A got halfway through and then was picked off the CPU, but it's going to come back. Process A believes it knows what the value of I is because it has already performed the load.

So if process B comes in because it's been given a turn and it says, OK, I'm going to run this code. It's going to load from the same memory location the value which is still 45, right? No one has updated it. We pulled the 45 from memory into a register. It's still in register one, but now we have a totally different process running. So we pull the same value from memory, put it in a register. But now let's say we don't get interrupted, so we pull it into the register, we increment it now to 46. Then we store it back to the same location in memory. Then we, now process B, go off and do whatever else we're going to do.

Process A gets brought back. It's now running again. It says OK, I'm ready to do the next thing, but the next thing is not the first instruction here. We already did that. So it believes that the value it has in the register is the value from memory. But that's not true anymore. Someone else went and updated memory and we didn't get notified. Right. We got frozen in time after getting the value. So we're going to proceed as though our old stale value is still in play. We're going to increment it. We're going to write it to memory. We're thinking now that we've incremented it correctly, but process B fully incremented it. Process A got frozen and then completed incrementing it. Two increments occurred. Only one got saved.

How would you feel if that happened to your payroll bank account? Not very happy, right? One paycheck got temporarily suspended and then completed while we're writing the second paycheck you got. That would be that would be a pretty lively pay scheme. So the idea of the critical section is we're going to identify in our code.

Oh.

Consecutive portions of the code, so consecutive instructions where we're going to say from this instruction until the end of this operation, where we're playing with a particular piece of data. Where that data is shared with a different process. While I'm in here, my friend doesn't get to also come in here. You need to finish the update before you're going to let your friend do their update. Then everything's fine.

Using the paycheck example, if you've got two paychecks coming into your bank account, you don't really care whether the bank processes the first paycheck and updates your account and then immediately processes the second paycheck and updates your account, or if they decide to do it the other way around and process the second one, and then the first one because they're additive. The result is that you got paid the two times you were supposed to get paid, so therefore it is correct provided the first update

completed before the second update began. This we're going to do in software. So this is the tool that we're going to be able to use to coordinate our various software processes together.

OK so here is a simple example. If we set the value of capital A into C, we iterate along incrementing I. Bring into array at location I the value from C with post increment. So as a reminder, C post increment means take the value of C and then update the value of C by adding one to it. So C is going to go A, B, C, D etcetera and then here if C has become greater than Z, we set it back to A. So if this was all by itself, it would just put in A, B, C, D, E, B, C, D, E, etcetera.

Let's say we have an interrupt service routine that sets A to the value C. Well here. If we have brought C... where was I going with it? If we brought C... Yes, we can bring out the value of C. It's not a very good example. You know what? I'm just going to confuse you with this example. I will rewrite this example and we'll come back next day. I think I accidentally broke it coming from the old copy of the notes. But I will have a good example in one moment.

Because I'm going to talk here about how device drivers work, so you probably all heard of a device driver, right? So a device driver is the piece of code that is written to allow a given hardware device to do its job and interface with the device from the kernel. We have three different categories of device based on how big of a buffer we use to communicate with the device.

The first and most common type of device is the block device. So it has a fixed size buffer that is used for every conversation with the device. So your disks for instance work this way. CD-ROMs work this way. Your video, as it turns out, works this way. Every time you do an update to the device, you give it the same number of bytes. Even if you have a small update you have to use a buffer of the same size and you're only going to fill in part of the buffer, but the whole buffer is part of the communication. This is done so that we can have a faster conversation with the device because each conversation, each utterance has a whole bunch of bytes in it. OK. But.

Yeah.

I'll do character next. So a character device operates one byte at a time. This is very convenient for devices where you don't know how many bytes you're going to need to send, like the mouse which is going to give you small little updates as you move it around. Serial ports, touch screen, joystick, anything that is not able to talk in blocks.

Except in the middle we have networking. So networking is a hybrid between these two because your network device block size could be reconfigured to change over time so the network is effectively trying to do batches of information. But the actual size of the batch is going to change over the time of the running of the computer. So we have a whole separate design for network. But other than that we have these two very simple strategies.

In both character and in blocks, reading and writing are super simple because we simply say I would like to... let's say we're writing. I would like to provide you new data. Here is a unit of new data. I would like to provide you new data again, here is another new unit of new data. The only difference is the size of the unit. The algorithm to do the transfer is effectively the same because it's a unit, a block of some data. Where for character your block is only one byte long, 100% of the time. For block devices it is a larger block, but for that device type it is always exactly the same size.

So as C programmers, I'm sure you can already see this is going to work super well in a fixed filing array format. You could have an array of blocks for your disk. They're all going to be the same size, so they'll fit nicely into the tiles of an array. Similarly, if you want to talk about your touch screen, you're getting fixed size things, they're just bytes. So you've got an array of bytes talking about your touch screen.

OK, so here's how we write to device driver. The items down the left hand side are essentially names of functions that have to exist in the driver. In the actual code, they'll be prefixed with some letters indicating what kind of driver it is. You might have mouse_probe, mouse_open, mouse_close, etcetera. Disk_probe disk_open disk_close.

Actually.

For the disk subsystem, the top half is dealing with code called by the kernel's main body. The bottom half is dealing with code responding to interrupts. So that piece of code ending with RTI is the thing right down here at the bottom called the interrupt service routine. That is the code whose starting address goes into the interrupt vector. Everything else is from the kernel side setting up or using profiles.

So probe, this is what happens at boot time. You may have seen this if you have some sort of open source operating system. They frequently like to print these out in the console, but the kernel as it is booting will start wondering what kind of devices are around. So one of the first things it wants to know is, is there a disk with some additional configuration on it that I could use? Is there a mouse? Is there a touch screen? New radar array. Is there any type of device that the writers of the kernel thought you might actually have in your computer?

So probe gets called to say hey, does this computer in fact have this type of device attached to it? If the answer is yes, then obviously we need to set up the device. So during the boot process, probe tells you whether or not things are present and does any configuration in order to make sure that, let's say your graphics subsystem is going to work.

PCs you can buy literally thousands of different types of graphic cards. They need different massaging to get them up and running properly, so that's where the massaging happens. Similarly, maybe you have a USB keyboard, maybe you have a Bluetooth keyboard. Before you can type any commands, the kernel better figure out what kind of keyboard you have so that it can get the right keyboard device installed and set up properly. So this is the work of probe.

Once you're finished booting and you are ready to go, cool stuff happens when we do IO, so if you're going to do IO, the rest of the functions are there to allow that to happen. Now remember this is the IO subsystem within the kernel. So if you're in a user process and you say you want to open README.txt, you call open. Right in your C program you're going to type OK, I want to call open or fopen or whatever it is. That calls a trap for the IO subsystem for that action. The trap then says, oh, well, we're talking about opening README.txt, so we come in here and call open at this level to open a communication with that process, with that hardware, sorry with that device.

All Unix design systems, there is a directory where there is a file system handle for every one of the devices on your system. Frequently there's extra handles for devices that aren't even on your system, but could be on your system. This is the directory /dev. So if you're looking down, you'll see lots of what looks like files. They're not really files. They are the access points to this interface in the kernel.

So you can for instance open your mouse. If you open /dev/mouse and start reading bytes, you will see bytes that get produced as you scroll your mouse around the desktop. Now, obviously you don't want to break your Windows system. So I would probably not do that while you have a Windows system up. But if you were to be booted, let's say to a kernel, you can just drive the mouse around. You can find out what the mouse is doing. You can see data coming out of these devices. So /dev/disk0, that's the interface to talk to a particular disk.

So.

These devices obviously need permissions to protect them from nefarious users just doing things like "yes, I'd like to overwrite the contents of the disk by just shooting a lot of data on /dev/disk0." I mean that would destroy the format of the file system of the disk so you don't have permission to do that unless you're the root. As root you could do that. Back when you formatted disk, that's how it gets formatted. You probably don't want to shoot random data into your devices if the content of the devices are important to you, but this is how it's done. They're actually attached to open calls by name.

There's a couple of software devices there. Does anyone know what /dev/null is?

It's in the devices directory. It's a character device that deals with its data one character at a time. OK, this is sometimes called the bit bucket. It is a place where you can put streams of data that you want to have go away. So let's say you have some you're writing, let's say a script. And you want to run a command, but you don't actually care about what is printed by the command. You maybe care about the exit status. So you care about something else the command is going to do.

But this command, whoever wrote it, is shooting lots of debugging information which shoots lots of things out on the screen and you just want all that crap to go away somewhere. You really wish there was a garbage can in which you could put all of the stuff coming out. That's what /dev/null is for. If you open /dev/null and write to it, it will happily accept all the bytes that it is given and simply discard them.

Another handy one is /dev/random. /dev/random is a device for producing data. So if you open /dev/random and start reading from it, it will give you random numbers. Just produces integers. So you can pull random numbers anytime you need random numbers. So there's a number of these funky devices. And this gives us a user space handle where through a file descriptor we can simply communicate with the device.

Close does what I imagine you think close does. It's the opposite of open. It shuts down the communication that we started here. OK. That leaves us with read and write and so these are kernel level routines to handle the action of what should happen when a user says I want some data out of my device, I'm doing a read, or I'm trying to supply some data to the device, I'm doing a write. So far so good.

OK, so here's a sketch device driver. So we could have a get_key from the keyboard. User space is going to get this result. But somehow we need to attach it to an input, so that when the keyboard actually has a key of data, the keyboard interrupt is going to fire. The hardware supporting the keyboard subsystem is going to put a value in a register, that is on zero page. The location, sorry, I said register. I shouldn't have said register. It's going to put a value in a location that is accessible through zero page. That is, we call it a magic number or magic location in your physical address space.

Your computer manufacturer would have told you "keyboard values show up at this address." Right. It's written in the manual for the keyboard. Every time the keyboard has some data, it makes it show up at that address. It highlights that this has happened by raising an interrupt with the kernel.

The kernel was totally free to ignore that. Right. The interrupt handler could just be a null, you just go "forget it, I don't care about the keyboard, keyboard's not important." Keyboard keeps tapping away. Nothing's happened. You want something to happen, you need to register a function at the interrupt vector location associated with the keyboard.

So that when you start typing, the interrupt happens. Response is that this function gets called. The function is going to say OK, I want out of the memory map location for the keyboard data to take the value that is at this hard coded address that I read out of an instruction manual for my hardware motherboard architecture. And I put the value into some data structure because someone's going to want to know we typed capital T.

There's another companion value. Usually you can see right beside it on the zero page where they have the status of the device is also showing up. There's a lot of statuses. It can say "I am very sad. Life is not working very well." I don't know. The printer is empty. The disk hasn't been spun up. Whatever kind of problem it's trying to tell you about, it puts it in the status location, so these are memory addresses that have been initialized to these hard coded locations, landing somewhere in our memory space so that we know to read values from those locations to find out what's going on from the device.

Similarly, we might write to those locations to tell the device something. "I want you to write the buffer now. I want you to do something for 275 bytes." You might write to these locations, so this is the interface between the kernel and the hardware. I imagine in a 2030 assignment having to do with IO you did exactly this. You had some kind of magic location. You put data in the location, read data from the location and then probably stored it in your memory.

OK, the difference between this and that experience is that you had one process, one program counter ticking along doing the stuff. But in the interrupt, we're doing the work so we can hand the data off to someone else. So we can say things like OK, while the keyboard status tells us there's still data to be read, we can loop along doing something with the data. But somehow we need to take the value from the memory location for the keyboard and put it somewhere so that it magically shows up here. Because this is where get_keyboard is going to have the data to return it. So we need somehow the values to just appear into the other routine. OK, we're going to do that with a shared data structure.

We do this typically with what we call a ring buffer or a circular queue. And this is because of the difference in speed between events happening in user space and events happening in hardware space. OK. So we would like there to be some sloppiness, some slack there, so that a device can perhaps give us two or three pieces of information before we have to deal with the first one. Because especially if we have a whole array of different programs running at the same time, we can't guarantee that there's someone willing to leap in and handle that piece of information at a moments notice.

So we're going to have a buffer of buffers. We're going to have a set of locations where we can put data from multiple IO interactions. So we call this a circular queue. We initialize them to some size. And then as data is exchanged, we can add it to the queue or consume it from the queue. But we have the ability that the queue can be partially full. Right. So we can add data to the queue until the queue fills up with no problem.

Adding to the queue, very simple. Here we initialize a location on the queue with a value. The type of the location of the queue if it is a character device, is going to be a byte. If it's a block device is going to be a whole block. But as I said, it's still one unit update. So the type of queue in this case is character, but it could easily be a full block size.

Once we've initialized it with the value, the important thing to know is how much data is in the queue and where it is. So we increment the end position of the queue. We're placing it at the current end of the queue. Walk the end position forward, modulus by the queue size to wrap back around once we've come to the end of the queue and increment how many values are stored in the queue.

Removing is very similar. Except now we don't care about where the end is in the queue. We care about where the start is of the queue. And notice that this lets us take something out of the queue without having to shuffle up all of the other pieces of data.

So we've put something initially at location 0. Add after that. So if we were to put 2, three things in the queue, they would end up at 0, 1 and 2. But now if we remove something, we remove it from location 0. So the queue now starts at location one. It has two elements in the queue. We didn't need to move those elements. They're just sitting where we put them in the queue. There's empty space at the beginning. But that's not wasted space, because if we keep adding to the queue, eventually we wrap around the end and we'll reuse the beginning of the queue.

So the size of the queue tells us how many things we can have in our IO buffer. But the buffer could be empty or full, or somewhere in between. And the point of the circular queue is that once we flip something in the buffer, we don't have to do that costly moving of the data to make sure that there's no gap at the beginning of the buffer. We're not expecting our new sequence of data to start at location 0. We're expecting it to start at the position recorded in start_pos, which is going to walk forward through the queue every time we do a consumption of a value from the queue. OK. Does everyone see how that works?

OK. We have a minute and a half left, so let's leave it there. I will just foreshadow, but I'm finally going to talk about masking interrupts off so we can do priority and answer Rocco's question of earlier in the period. We'll do that next time we meet. Alright, have a great weekend everyone and we will see you.