CIS3110 Class Transcript Part 1 (2025-01-29)

All right, perfect timing. OK, welcome again everybody. Last day I forgot to talk about the handout for assignment two that I promised I was going to talk about. So let's get that off the decks now.

So this is a semaphore assignment. Obviously for semaphores, we're going to need to have many programs running at one time so that they can communicate with each other and there are basically two different strategies for giving you some of your assignment. One is to ask you to open a million windows and start your various programs, all in different windows, so that they can all communicate with each other. I found that not very successful in terms of getting people to where they need to go.

So the alternative is to start a bunch of child processes using fork which we will talk about most likely later today, but it allows a running program to start another running program. So in this assignment I've given you some code which will start a number of clones, so they will operate as children. And we'll talk about what that means in detail. But the important thing for you to realize is that we're modeling a network strategy.

So a token ring network is in fact a networking tool to get messages from one computer to a totally different computer, and we're going to model this with one process and another process. So this is a commonly used networking strategy for moderate to long haul. So for instance, University of Guelph is one of the nodes on a token ring network called ORANO that connects all of the higher education institutions in Southern Ontario with a number of other entities.

So when we're talking about the nodes in our graph in this assignment and I'll scroll down here to where we have our pictures. So the nodes, the circular numbered items. You can think of as separate computers. And then in between each computer is a link. Now most token rings like ORANO and like the one we're modeling really are fiber optic rings. So the only thing you can have on the data connecting one computer to another computer is the presence or absence of a blink of light over the fiber optic cable.

Now I'm not going to ask you to do things at the bit level because that adds a whole extra level of complexity, so we will pretend we're operating on one of the multi-channel fiber rings where you've got different colors. And therefore you can blink a bunch of different colors all at the same time, and the other end says, "Oh yeah, the blue is on, but the red was off," and so you can essentially transmit close to a byte at a time.

OK. So we won't go through the details of the physics of how that works, but from our point of view on this network, you've got the ability to store a byte worth of message in between two nodes. So our nodes are going to be modeled by processes. The byte is going to go into a shared memory buffer between those two processes, so there's one between 3 and 4, one between 4 and 5, one between 5 and 0, and you see, I numbered them from zero to five around in a ring.

So if three wants to send a message to 4, that's easy because they're next to each other on the ring. So it can just put the message one at a time, one byte at a time, into the shared memory and four can pick it up and that's all good. But if three wants to send a message to somebody else, then there's a problem because 3 is not directly attached to anyone except 4 in terms of its outgoing connection. Note the gray arrow I have in the middle, indicating the direction of flow.

So if three wants to communicate with two, it needs to get 4 to pass on the data in a game of telephone that goes all the way around the network. And then can come off when you get to the right node. Like riding a merry-go-round. Right. You get on at one point, you can ride around, when it's your time to get off, you can get off. But the "you" in this case is a message. There's no smarts in the data. The smarts are in the nodes that are doing the processing.

So the way this works is 3 can communicate with two by passing out information saying, "Hey, everybody, I'm trying to communicate with two. Please pass along everything I say until 2 gets a chance to take it off the network." But if the message is more than one byte long, we also have the problem that if 2 is sending out a message and somebody else decides to start talking, then it's going to get all overwritten and scrambled.

So this is why we call it a token ring network. There's a special value that gets sent around and around and around, and you can think of it like runners with a baton. They pass it from one to the next. It goes around and around and around. And if you want to speak, you wait until you get the baton. Once you have the baton, you can take your turn.

So what we have here at the top of the network with the pattern I've just arbitrarily decided the pattern is 2 slash lines. That's the baton when there's no message attached to it. You can see in the next image, I've changed the baton from 2 slashes to two pluses, but everybody recognizes that it's still the baton, but they recognize that it's in use, so the data coming after the baton is going to be part of a message.

So the way the message works is it says: Here is who is supposed to receive the message. We'll assume for the sake of simplicity that there are less than 256 nodes in our network, so we could just use one byte to talk about who's supposed to be the receiver. Next byte is who is the sender. Next byte is how many pieces of data are there. Then there's the data.

OK, so you can see here the in-use button has moved from between 3 and 4 into between 4 and 5 and the first byte of our message has been sent out on the network by Node 3, and I've got a little triangle indicating that we're that far into the message. If we go further in time, you can see that all the pieces of data are ratcheting around.

So here's our in-use button coming back to three. Here, arranged around the network are the various pieces of the message that you can see directly correspond to what node three had in its buffer of what it wanted to say. Node one recognized that the message was for node one. So rather than just playing

telephone and passing the bytes along one at a time, it says well I pass them along, but I also take a copy for me because this is a message for me.

So then as we keep going, node three knows it's the speaker. So when it gets the baton back, it just continues to speak until it is done with its message and then it puts the unused button back on the network once it is done speaking so that someone else can pick it up and the next message can come.

So that's why the triangle here got to the end. You can see the remaining pieces of data on here, but some of our data has already been thrown away. The entity that is throwing away is node 3, which knows that the data is safe to throw away because it is the speaker. No one else knows it's safe to throw away because the ordering of the nodes is not actually known to the nodes on the network.

So we know because I've just written them in ascending order that node one could have thrown the data away. But in fact Node 1 doesn't know where it is in the network and neither does anyone else. So the safe thing to do is you send the data all the way around the network and then node 3 when it is done speaking simply puts the message token, or empty token back on.

OK, so our job to model this then is we're going to have essentially a tiny producer-consumer relationship that spans each of these square points on the network. So 3 can produce to four which can consume from three. But it's actually only a one character long queue, so we've got a multiplicity of one character long queues that all go around in circle.

Any entity can decide it wants to speak, and I've given you some code so that they will randomly generate their own messages and say "I'm node X, I want to talk to node Y. Here is the thing to say," some other one will wake up. "I'm node Q, I want to talk to node B, I have something to say."

So your job is to implement the data flying around the queue so that it begins to speak when it gets this empty token, and stays silent and waits otherwise, but while waiting to speak can pass the data that it may be receiving while it's participating in the ring structure around the queue.

OK, so a whole pile of tiny little producer-consumers. You will note that much like the dining philosophers, we have a ring and rings can cause problems with deadlock and that's one of the things I'm asking you to think about with this assignment, so that you can make sure that your system doesn't grind to a halt because some node is waiting on some event that isn't going to happen.

OK. Questions. I've got a little algorithm for you in here in the assignment to talk about what happens when a packet is received, there's some starter code. Anyone who was trying to get the starter code prior to I think last night - I accidentally put capital CIS in the string in the assignment here, but of course our directory has a lower case CIS in its actual name, so I fixed that and you should be good to go.

This now is in fact a directory on Linux.socs with code in it to get you started. I've given you some macros called SEM_WAIT and SEM_SIGNAL that wrap up the Linux semaphores so that they follow the rather

simple signal-wait protocol.

Linux has two - well, POSIX in general has two different sets, or two different types of semaphore. There's the newer type than the older type. The older type you got a whole set of them at one time. I've set the macros up for the older type, but I've got an example in the course directory of the newer type and we'll talk about both of them in the future.

[Student question] The last question back here. OK. Yeah. I will take that. It will be submitted the same thing. I guess the last time they did that.

[Student] Thank you.

I would like you to learn how to use tar. Car is much older. Car stands for tape archiver. So it was to read things off tape. Tar's job is to put things in archives so file comes to an end, the next bytes are the beginning of the next file and later they said it would be nice if we could compress them. Because the original tar was just interested in moving files around because they went on to tapes, you can't switch the tape up, so they weren't as interested in the compression property whereas zip was interested in compression from the beginning.

But they have an interesting relationship because zip is an archive of compressed files, whereas tar is a compressed archive of files. So each individual entry in a ZIP archive has been compressed separately. A tar file that is compressed was first made into one big archive and then compressed.

So this has some interesting issues with respect to its use, because you generally have - have you studied compression yet? Is that a thing that you've seen? Not really. OK.

So fundamentally the way all these compression algorithms work is they say if I recognize a pattern that is a long sequence of bytes in the file, right. Let's say you've got a file of... it's a phone book. You've got a bunch of addresses. You've got a bunch of people's names and so on. So let's say it's got a bunch of addresses in it, and so all the addresses might have the word "St." Well, you don't need to record the word "St." in the file every single time you say street.

If you had a symbol that said I'm going to have a special symbol and that means streets, then you just have a table at the beginning that says OK "Street" is going to be this symbol and then all the way through the file you can just put that symbol in wherever that pattern works. This is the way essentially all lossless compression works is trying to find better and bigger symbols that they can replace with a short marker. So that later when they expand it, they just put the marker in or substitute the symbol for the marker.

So if you make the archive before you do the compression, then if the same symbol appears in multiple files, you can take advantage of that because you only need to store the symbol once per archive. So tar that is compressed tends to have better compression properties than zip, which can only compress every individual file separately.

But this is the reason why zip can be split into pieces and the separate chunks of the ZIP archive can be used to some extent on their own, which was very important when people were using floppy disks and other fixed size media where they'd have to split their thing across multiple disks. Not as important anymore.

So they're both popular. I think it is important for you to learn how to use both of them, so I am glad I gave you an assignment that said, "No, no, really use tar," but as long as you've done it once, I'm assuming that you're going to remember how to do it from then on, so if one of them says use tar and the other one says tar or zip then just follow what it says on the assignment.

I'm not going to override the assignment at this point, but I would like you to be conversant with both of these strategies. We now also have RAR, which is a new version of a compressed archive, but not everyone is signed up to RAR, so it's kind of lagging behind in the industry world because it's very annoying when someone sends you a RAR file and then you're on a computer that does not have a RAR extractor.

So then we have to go and install some software before you can unpack the stupid archive and then you're saying why didn't you just send me a compressed tar file or a zip file? Because I have those natively here, both of mine. OK. Any other questions?

All right. So I'll let you read that and that can percolate and we can talk about that further next day.

OK. Last day we were talking about... We sort of wrapped up our discussion of semaphores and then I said there's a number of other IPC strategies so shared memory, which we'll also be using in this assignment. You'll see I've set up the shared memory for you. That's where all the memory that you're going to manage with your critical sections and semaphores is going to come from.

We talked about semaphores and mutexes. Now we're going to wrap up signals and talk a little bit about pipes.

OK. So last thing we talked about a signal as effectively an interrupt-like structure. But unlike the interrupts coming out of the interrupt service vector, these are not triggered by IO events. Unlike the traps stored in the trap vector, these are not triggered by someone trying to do a kernel operation for purposes of IO.

This is instead a third set of function pointers, where we essentially are saying process A wants to tell process B that some particular event has happened. And process B is able to register some code to say this is what I do when I'm told about this kind of an event.

So a signal handler, much like an interrupt handler, much like a trap handler, is a function. It's a function that gets called when something comes from the outside, right? So it's like you're all sitting here and then

someone decides you're playing dodgeball and they rip a ball into the room and bang it goes off someone's head and they react.

OK, so this is what the signal is. Signal B decides OK, I'm going to I'm going to give them a signal. Whack. Something happens to signal A. Signal A wakes, stops what it's doing. It is interrupted much the same way as an interrupt service handler would interrupt you, but instead of running an IO based routine, you run whatever the registered function is and then it goes back to whatever it was doing before unless the signal handler function calls exit or some similar thing.

B gets no notification that this has been in any way successful, or even that A responded. So we can simply say I send event type X to signal A. Boom. Hope it got there, hope they got the message.

OK, so then I finished up last day. We had this table of signals. I've extended the table of signals with the rest of the numbers that I left off. For some reason, I realized what the reason was, so some of these signals... First I'll say the signal strategy is really old. Predates Unix. When they were writing Unix in the first place, and we'll come back to the history of how that happened in a few weeks.

But when they were writing Unix in the first place they said OK we should do... We should include signals. So in the table of signals they included the ones that other people were using. So HUP or hang up is the signal that is used to indicate that the communication to that terminal has gone away.

So if you close the terminal, the shell running in the terminal gets a hang up signal from the terminal. Terminal says, "Yeah, we're going away." Same deal if you use secure shell and you kill or lose the secure shell connection. The way the shell at the other end finds out that that's gone as it gets told that by hang up.

So a hang up signal is sent to a program to tell it that you've probably lost your connection to your input terminal. Not all processes have input terminals. Specifically, server processes or demons as we sometimes call them, running to help the system itself work. They did not attach to a terminal.

So they've overridden the meaning of hang up to say if you send a server hangup, what the servers tend to do is reread their config file. Which is why, let's say, if on your home machine you're running a secure shell server and you change the configuration, and you'd like your secure shell server to reread the configuration you can send to the server a hang up signal and instead of hanging up and going away, it'll say "Ohh this is my cue that I need to reread the configuration and essentially start all over again."

So all the servers tend to do that. So if you see someone talking about sending a hang up signal or signal 1 to a server, that's what they're doing. If you send a signal one to hang up signal to something that is attached to a typable terminal, it's probably going to decide "My connection's gone away and I'll hang up gracefully rather than hang around forever."

Because you can imagine what Linux.socs would look like if every time a student lost a connection to it, if that meant there were programs just hanging in the background waiting for something to happen. So this is to allow them to clean up nicely. They go "Ah, the Internet went down. I guess I'll just go away." So that's hang up.

Signal 2, interrupt. Whenever you press control C at a terminal, it generates an interrupt signal for the process running in the terminal. The default handler for interrupt simply calls exit. We'll see in a second how you can register a new handler, because you might want to do something like save some data to a file and then exit.

Quit is what it sounds like. Please go away gracefully.

Illegal instruction #4 you may have seen this. This typically only happens if you've got a binary, right, so a compiled program that is compiled for some other processor in the same family as the one you have, and you've duped the system into running, let's say, a newer binary than the one it knows how to run, so everything will be fine, and then all of a sudden it gets to an instruction it's never heard of. And it exits with SIGILL.

So it will have... the system will send the signal to the process to say the instruction that was literally loaded into the instruction register is an instruction that I, the CPU, do not know how to perform. So goodbye. We're done here.

Signal 5, trap. Kind of related to the trapping to the kernel. This is a trap into the kernel for debugging purposes. You may not have thought about how a debugger works, but the debugger needs to take control of a different running program and control what that program is allowed to do. How many instructions in a row it is allowed to execute? This is handled through the trap.

So the program that is the debugger rewrites the executable of the program it's debugging, installs a trap signal handler and then it uses that trap signal handler to control what's happening so it can run it one step at a time.

Signal 9. Kill. OK, this differs from quit in that quit you can write a signal handler to say "you sent me a quit. I'm going to do something about that." Signal 9 cannot be interrupted or overwritten. Rather signal 9 is there so that if things are going badly and you really, really need to get rid of a program and that program is refusing to go away, you can send it a #9 signal and it will then be removed.

So those are the ones that are at standard numbers. Now in the history of the development of Unix, Unix was developed by a group at Bell Labs in the US. So lots of smart people worked at Bell Labs, Kernighan worked there, Ritchie worked there. All the names that you hear associated with C and Unix. So John Bourne, Rob Pike, Kirk McKusick. They all worked in this glorious Research Center that Bell Labs used to have.

So Bell Labs was a phone company, a monopoly. U.S. law says if you are a monopoly, you are not allowed to expand into any new markets. This was because of the robber Baron era in the late 19th century. So Bell had this problem. They wanted to do new, exciting things. They wanted to develop stuff. They had a lot of money. But they weren't allowed to capture new markets, so they took their research arm and said, "OK, we're just going to call it Bell Labs. All you folks sit here in this three building complex, think big thoughts, make us famous, come up with good ideas."

Other companies have done this. Xerox, for one, is the one that came up with Ethernet. Is an Ethernet cable lying around here? You're probably very familiar with this cable. This came from the photocopier people. It was designed so photocopiers could phone up head office and say, "Hey, I'm jammed, send a technician."

So Bell Labs invented a pile of stuff. A lot of programming languages, C among them. A lot of operating systems, Unix among them.

One of the things that happened when they developed C and developed Unix is they were not allowed to sell it because they were a monopoly. But they were allowed to give it away. So you could write to Bell Labs and send them a bit of money for a tape and they would send you a tape filled with the Unix operating system and provided you had a computer, you could load the tape into your computer and then you could use Unix.

And across the world, really, the states at first, then North America, then really around the world, you've had all these entities called universities, that owned computers but didn't want to pay license fees for an operating system on every single one of those computers.

And there was a company called DEC that hadn't quite figured this out, but kind of liked the plan. Anyway, they would sell you a pile of computers and then notice that you only bought one operating system license, even though you bought 10 computers. And then you just ran Unix on the rest of them.

Which meant that people were really excited about this free operating system that did great things and it became a big research platform at a number of different places across the world, one of them being Berkeley. So Berkeley Systems Development Group is a research group out of Berkeley University. They started extending Unix. They invented networking. So all the networking and all the Internet built on the networking came out of Berkeley's extension to Bell's Unix.

We now had two centers growing new ideas, of course. What happens as soon as you have two centers growing new ideas is they come up with similar ideas but do slightly different things with them. For that reason, there is a string.h in the C standard library and a strings.h in the C standard library. For that reason, there is memset and bzero and memcopy and bcopy because these are all functions that were written by two different groups of people who then said, "Ohh yeah, we did that too last week. OK, well, we'll just leave it there for now."

CIS3110 Class Transcript Part 2 (2025-01-29)

OK, relevant to our discussion now though, they both started adding new signals. But of course they didn't use the same numbers, So what you'll find is that the column on the left, the signals are always at those numbers. The column on the right, the signals always exist, but depending on whose version of Unix you have, they may be given a different ID.

So over here we have bus error which basically means you supplied an address that for which there is actually no hardware memory available. We have segmentation violation. Everyone's favorite signal to receive, which says you have supplied an address which, while valid in the logic of your virtual memory system, does not actually belong to part of your program that has been allocated, and so you can't have memory at that address. It also happens if you try to write to memory that is marked as read only. Which everyone who's trying to write at the null pointer is familiar with that problem.

Sig power. This is an interesting system maintenance signal. So if you're running a big computer with important data and many people using it, you frequently have a little bit of battery backup associated with the computer. So if we have a big power outage, the computer doesn't immediately go down. However, a battery backup is only going to last so long.

So UPS's, uninterruptible power supplies, will send a signal to the hardware. And the kernel will then send SIG power to everybody who's running, so that they can say "The system is going down, if you have some important data you wish to save, do it now because the computer is about to be turned off."

If you run a shutdown command, it will do the same thing, so if you run a shut down command as root at a Linux prompt, it tells all the other programs "We're going to shut down now" so they can actually do something sensible. Whether they actually do something sensible or not has to do with whether they have a signal handler registered for SIG power. If they don't, then by default they just exit.

OK, Sig term is another process termination. The subtleties between quit and term aren't worth going into here, but it is always around. And then they have defined 2 user defined signals. So you're guaranteed that SIG user one and SIG user two are signals on your system. They both exist and that there is no system-wide meaning for them.

So a lot of servers, if you send them SIG user two, they might increase the verbosity of their logging. And then SIG user one tells them, "Yeah, yeah, that's enough. You can tone it back down now." So this kind of go up, go down control is frequently controlled with these SIG user signals.

OK, so the left typically found at these numbers, right? Your mileage may vary. I've given you the numbers for Linux. You want to see the numbers, the command kill - which is a really badly named command. You

can use it to send a kill signal. By default, the signal that it sends is a kill signal. But you can tell it to send any signal you like.

So kill -1 send signal, signal one. You give it a process ID, so you can say I'd like you to tell that program signal type 3. You can do things like say, "Yeah, I'd like to tell that program over there they're having a segmentation violation." They're not really having one, but I'm just gonna... I'm going to stir the pot. I'm going to send them a signal. Let them see what they do. So that command is kill. You can indicate the signal, then you give a process number. OK. Questions on that. Oh, Ethan. Yeah.

[Student question] Does Windows have something related or similar?

They do. They have a C implementation on DOS slash Windows. They do have these signal handlers.

[Student comment] You've talked about some of the kernel. So... which we can't actually access the kernel, but this seems to be something our program is actually able to like, work with. How do we learn to do that?

On the next slide. OK. So on the next slide, I have a signal handler. You can see if you include signal.h, then it makes available to you the list of all the signals on your platform.

To make a signal handler work, you need to write a function that corresponds to the signal, sorry, that corresponds to the signature of a signal. So that means it returns void because remember it's going to just randomly get called. You have no control over when it gets called. It gets called when someone else sends you a signal.

So whatever you were doing will suddenly be interrupted, and then you're running the signal handler and then unless you call exit in the signal handler, you will return to what you're doing. So there's no sense having a return value because no one called you from within the program, you're just suddenly running this function, and then you just suddenly go back.

So it's a return type void, name of a function, and then it takes an integer code. So the code is the signal that you are being called - that is invoking, as we say, the signal handler, so this allows you to write one handler for multiple signals and still tell which signal you got.

The body of the function can be anything you want, it's just C code. So print statements, I don't know, do a pile of work, call exit - if you call exit, obviously your program stops. You're not then going to return to what you were doing. If you do not call exit, then you just run to the end of the function, as soon as you run out of the end of the function, you're going to go back to whatever you were doing when the signal knocked you off. Yeah.

[Student question] But is it doing this in the language of C? Is there a model where all these virtual tools and programming languages are built on top of C?

So C is kind of the lowest common denominator. But the operating system itself is almost exclusively written in C for pretty much any operating system you want to name. There are a couple of exceptions in the distributed world where they went for C++, but those are very kind of academic niche sort of operating systems.

OK, so we have our function. It is a signal handler. I called this one literally "signal_handler", but you can call it whatever you want. So the only thing that remains is: How do you tell the system that you want the signal handler signal_handler called when you get a particular signal?

That is done, counterintuitively, by the function called signal. So signal the function does not send the signal, the function signal instead takes your function, your signal handler and a signal ID, and registers it in the table of signal handlers, so that later, if you get that signal then that handler will get invoked.

And it actually does - I didn't include it in this slide, but it does have a return type. What signal returns to you is the old signal handler that was registered at that point in the table. So if you want to do the hokey pokey, if you want to sub one in for a while and then put the old one back, it passes you back the old one, so you could hang on to it until such time as you wanted to put that one back in the table.

Typically, people don't do that. Typically people say "I have written some code to handle a particular type of event. I'm going to register it and then it's in play until my program stops running." Yeah, so I guess right?

[Student question or comment, unclear]

Well, I guess I'm disappointed in this case. Well, but you're not... So remember, while you're using a function pointer, at no point in this process are you actually declaring something of type function pointer. The argument of type function pointer is the argument to signal. So to use it all you need to do is provide the name of the function that you've written and it needs to have the right type for the pointer that is the argument.

Yes, it is one of the inconsistencies in C that the name of a function is deemed to be synonymous with the address of the first byte of the first instruction of the function. But if you put an ampersand, it means the same thing. I hate it. But it is true.

It should, but there is... Yes, it is a... It is a silly thing, but that is... that is basically where we are. You would... Well, you would have to make a pointer to function and then pass the address of that. Right. Because the problem is if ampersand name of function is meaningless because you've taken the address of an instruction, but you don't have anything that points at the address of the instruction, you just have the address.

So then to say what is the address of the address is really meaningless. So to my mind, ampersand name of function should be syntax error, but in reality they've decided they'll just pretend that you didn't type an

ampersand. You would... yeah.

OK. So that's it for signals, they're pretty straightforward. Integer list of events you want to indicate to some other program that an event has happened. So you can send a signal - the the function to do that is also called kill. You give it a process ID and a signal ID. You can also do that from the command line with a process ID and a signal ID.

OK. Next Inter-process communication strategy is a pipe. If you've ever typed at a command prompt and used a single vertical bar in between 2 commands, you have used this inter-process communication tool. This is why the vertical bar is frequently called a pipe.

So if you say something like cat my_source_file.c vertical bar pipe into say more. Cat is printing to standard output. It is somehow being collected into a pipe and more is then printing it on the terminal.

So the way this works is it works with a shared buffer of exactly the type we've been talking about with producer-consumer, managed by a semaphore. But this is such a common pattern that we just wrapped it up for you to say I'm going to give you a way of allocating the semaphore and the memory all in one go, so you then can just write to it using one file descriptor and read to it from another.

Sorry, I've got the wrong version of the file here. Open—so there's a little bit of crust on the screen. I'm just going to open the correct version of the file. OK. There we go. So actually the same.

So we've got a process I've shown it with a circle. P1 I've got another process, another circle P2. In between them is our fixed size buffer. P1 writes into the fixed size buffer, P2 reads from the fixed size buffer.

We already know how producer-consumer with the semaphore works, so you can see that P1 being the producer is going to be able to write into the buffer until the buffer's full. And at that point it has to go to sleep because the semaphore will force it to sleep because you've consumed the amount of available space in the buffer.

Similarly, P2 can be reading from the buffer and if it runs out of things to read, it's going to go to sleep. Makes sense?

So it's basically like a file. It's got actually 2 file descriptors. That you're familiar with a file where you open a file and you have a file descriptor to communicate with the file, but the data for the file is on disk. This is essentially a file that hovers in memory. It's never allocated any space on the disk.

So it's just a buffer floating in memory. And we manage the two with a semaphore. So that a write to the file, will go—or the file descriptor will go into this buffer. But then writes will be blocked if the buffer is filled. And can resume as soon as the companion process reading makes some space by consuming some

data. OK, so we've got that nice producer-consumer thing all wrapped up. Hanging in there. We're going to find out who's ready for any slow days.

OK. So we're going to talk—to go back to the next event but for now, we'll say that all of the physical hard drive, mouse, keyboard, videos... physical devices. Outside of the relationship of peripherals and RAM, so all the devices you probably remember from 2030 when you're writing your IO assignment, you communicate with an IO device with your mapped location in RAM. Interrupt comes back with an interrupt handler.

That whole delay takes things out of the world of RAM. RAM is simply one array of physically addressable memory locations where the only thing you can do with a location in RAM is read or write. It doesn't persist if you turn the power off (unless it's EPROM or something like that).

[Student question] Here and there, a little fast and loose there in your definition, so by buffer, what do you mean?

What I mean is a block of contiguous storage in RAM. OK, so you can think of it as like an array, but we're just talking about somewhere in RAM. We have a block of contiguous storage, much like the pool you were managing in assignment one. So a buffer is just a linear structure of some number of bytes.

Now a circular queue is a higher level structure where you could take that buffer and say, "OK, I'm going to introduce an algorithm, a protocol, we call it that says when you get to the end of the buffer, you loop back around to the beginning." So you've implemented a queue that goes around and around and around by using a buffer.

OK, so in the kernel, what we'll learn is that there's a data type called an mbuf, a memory buffer, which is the generic—you can think of it like the pickup truck of the kernel. It's the generic thing to hold stuff. Holds a fair bit of stuff. And then different mbufs are used for different purposes at different times.

So we'll find out that IO happens by reading from or putting it into an mbuf. You'll find that running program memory is implemented by assigning a bunch of mbufs to be virtual memory, so that in that case the buffer is this block of a size the kernel has decided is a good size to use as its basic data type, and then it uses that for all sorts of different purposes. So great question. OK anyone else? Yeah.

[Student question] File descriptor just identifies a section of memory in the file?

Oh, OK, so file descriptor is an integer number into your small file type structure. So when you call fopen, what happens is you allocate a bunch of buffers for your program.

So you are probably going to include stdio.h and you're probably familiar with sprintf, maybe familiar with that. Then you declared that, right Ethan? Yeah? Yes, we have. Yeah.

OK, so after that right? You take a fixed size buffer and operate by either reading or writing that many bytes into or from that document. All of these f-based functions—fopen, fclose, fread, fwrite, fprintf, fscanf, etc.—they have this roared relationship between the lower level I/O and your program by having these buffers so it can carefully arrange data in the nice way that let's say printf would like to do it.

So printf says "Oh, I have a percent 7s. That means I need a 7-byte wide representation of some floating point number. I'm going to stick it in here." fread and fwrite don't have that smarts, but they use the same file or pointer to file structure interface.

The lower level interface is read and write with no F. What they do is much like fread and fwrite, they take a pointer to a buffer and a number of bytes, and one of these open integer-based file descriptors.

File Descriptor 0 is also called standard in. File Descriptor 1 is also called standard out. Two is standard error. The next thing you open gets file descriptor 3. So if you say I want to do a read operation on file Descriptor 3, then it looks it up in that table. The open has attached the entry in the table to that entity you're talking to, which we'll find out, could be a file—right? open talks about files.

What we'll find out later in the course is that there are lots of things that are structures of byte streams that look kind of like files and you can talk to all of them with a file descriptor. So a pipe is one of those. When if you want to talk to a pipe, the command for—the function pipe gives you back 2 file descriptors, one for reading, one for writing. You can call the routine read and write to put data into or out of the pipe.

We'll find that sockets, which are used for communicating across networks work the same way—you get a file descriptor to communicate with another computer on the Internet. We'll find out that there is a whole host of different things where the fundamental way you interact with them is you write a series of bytes, or you read a series of bytes and the interface for all of them is file descriptor.

They added the file—the pointer to file structure strategy later to give you access to formatted IO. So the F is actually for format. We probably never thought about why it was F, printf or scanf or fopen. It's because you're doing formatted IO. And it can handle therefore the vulgarities of "someone pass me an integer. How many bytes do I need to represent this integer? Ohh this one is 7 billion. Ohh that one is 5." And so therefore the number is going to grow and shrink.

The F level interface also lets you do things like ungetc—I want to back up one position inside my buffer. The lower level interface says if you need that fancy stuff, you're going to call a function to create that higher level interface.

There's actually a function called fdopen which allows you to take an open file descriptor and generate a file pointer to file structure interface on top of it if need be. But normally, if you're, let's say transferring data across the Internet, it's usually pre-formatted, so you're more interested in "I want to send 16K

bytes," not "I want to send the letter Q." So a good question. I'm glad we I got a chance to fill that in some. OK anyone else?

OK, so that's all I want to talk about with inter-process communication. To wrap up processes, we're going to talk about scheduling.

OK so I'm going to back all the way up to my favorite diagram up here on page 7, where we have the kernel in the middle and the hardware at the bottom, and the processes at the top.

So so far, we've talked about the fact you've got this vector of processes, or vector of devices rather at the bottom. Keyboard controller, we got a disc in there, we got the network, we've got a serial port, we have this thing called an alarm clock. Remember the alarm clock? We're going to talk about it in just a minute.

Then we have the kernel in the middle. Then I have put the user processes at the top. We have got 4 user processes—Process 1 through Process 4. The line between user process and kernel, we say crossed because of an interrupt. Hardware wants attention or a trap. We're going to ask for attention ourselves because we're going to say things like "I want to read some data" or "I want to write some data."

In the kernel you can see I've got four blocks for the process table entries corresponding to our 4 processes. So process 1 is a block. Process 2 has a block, so on and so forth.

OK. The other thing that is on this diagram that we haven't talked about yet is marked "run queue." So we have what looks like a singly linked list here. So the words "run queue" point at a process table entry which has a next pointer that points at a different process table entry, not in fact, the next process table entry, but just some other one. And then that one points to NULL.

So in this case, half of our processes are in a linked list called run queue, and the other ones are not.

OK, so if we return to our scheduling slide way down here, I'm nowhere near far enough. Yeah, so scheduling. Scheduling is about figuring out which process to run next on an available processing unit.

So I say processing units because nowadays we have a mixture of things called CPUs and cores and so on. Right. So if you've got a computing device, it will have at least one CPU on it. In that CPU there will be an instruction register. There will be a program counter register. There will be all those things you talked about in 2030 that you need in order to follow along and run a thread of control through one program. There may be more. You could have a hundred of these things. We know you have at least one or you don't have a functional computer.

OK, the scheduler is the software that says if I have an available processing unit, then someone should get to run on it, if there's someone who wants to be run. So when a process is running on the CPU, it is executing its instructions. There's a couple of things that may cause it to stop executing. The most common is that it wants IO.

So as we just mentioned a moment ago, if you want data from a device, you trap into the kernel, you ask the kernel to get you the data. The kernel then does its IO device driver stuff to signal that device that some work is required. That's going to then take some time.

OK, so it could be hundreds to thousands of instructions of time is going to have to pass between when the kernel organizes your IO to happen and the actual data from your IO comes back. So during that time, we do not want you hogging the CPU. There are no instructions you can run. So you should just not be on the CPU. You should be in a wait state.

So what happens is if you ask for IO, you're taken off the CPU and you're taken out of the run queue. You're in what we call a blocked state.

Another reason that you might stop being running is that you called exit. In which case you're about to no longer exist. And clearly you should not any longer be on the CPU because keeping you around is just a waste of resources because the program is done. OK. We'll talk about what happens when you call exit in a few minutes.

But the third interesting opportunity here is it says your time slice has gotten used up. OK, if you write a program intentionally or by accident that has an infinite loop in it—right? while(1). If you think about the instructions that get executed, there is never a time when a process that just says while(1) is ever going to ask for IO, it's just buzzing and buzzing and buzzing around. And so if it just was put on the CPU and left to run, the machine would never do anything else. Which sounds like a terrible idea if there are multiple programs running on the computer.

So what we do is we use that alarm clock device. When you get scheduled, there is an amount of time configured on the system that is called the time slice limit. That is the maximum amount of time that a process is allowed to run.

So we put your while(1) process, it's running on the CPU immediately after setting the alarm clock. So we say to the alarm clock, I would like you to ring in, let's say, 200 milliseconds. And we load while(1) on the CPU and it just starts buzzing. And then 200 milliseconds later, the alarm clock rings. What happens? It's a device that—what is going to occur? What happens when a device wants attention? Yeah. It's going to generate an interrupt.

So the alarm clock interrupt handler suddenly gets called. The alarm clock interrupt handler is a function within the kernel. So what the alarm clock handler does is it says "If I have been called, that means that the process on the CPU has hit the end of its time slice. It's time for it to be shifted out and someone else can run."

We're not going to destroy it, but we would like another process to get a kick at the can of being on the CPU, so we context switch out from that one. And then we enter this scheduler loop.

So you start at the top of the loop, because whoever's running is no longer running, either because the alarm clock ran, because they requested IO, or because they're exiting. So you mark them either runnable, if they simply exceeded their time slice, or blocked.

So if tsleep is involved here, this is how they get marked blocked. We also indicate why the blocked status. That's that argument to tsleep to say why we're sleeping. Maybe we're sleeping on the disk. Maybe we're sleeping on the network. Maybe we're sleeping on the monitor. We're sleeping for some reason. If you're blocked, it's because you're waiting for some event to happen. That'll be a hardware related event.

So then you save the process context that you've just taken off the CPU into the entry in the process table, so each one of the entries in the process table has a record set aside for all of the CPU register contents when you're doing a context switch away from that task.

So any process that's not running is in stasis. It's frozen there, ready to go as soon as we disable the Mr. Freeze Ray and put them back in the CPU, they start running again.

OK, so we've made this one either runnable or blocked. If it's runnable, it's still in the list of processes we can run, but no longer running. The kernel then searches through the process table for another process that is marked runnable. If there's only one process that is marked runnable, it may be the one that just stopped.

If we find one, then we mark that process as running, and we restore its process context onto the CPU. And then we run that process, it will pick up exactly where it left off, just like a hardware interrupt. Because it is a hardware interrupt, it was either the alarm clock interrupt or we called the trap and then tsleep.

OK, so we've got this list of things that are runnable. Something from that list we're going to put onto the CPU, then it will be running. If we've got multiple CPUs, then we're simply doing this across the full set of CPUs. Everyone with me so far.

OK, so the question becomes: How does this search work and who do we return from the runnable process? This is called the queuing discipline or the scheduling algorithm.

One possibility is FIFO—first in, first out. This is essentially what I had a picture of on that earlier diagram. So it's just a simple queue, frequently implemented by a singly linked list. When you get yanked off the CPU, you simply get added to the far end of the linked list. Whoever is at the front of the linked list you put on the CPU and you keep cycling around. Eventually you come back up on the list.

If you do it often enough such that a process is going to repeatedly join the queue, we call it round robin. Typically it's preemptive. So a preemptive queuing discipline is one with this alarm clock strategy.

If you're running an interactive multi-process operating system like probably anyone you've ever used, then they're all preemptive. There are other operating systems that are non-preemptive. They're called batch systems. The idea there is we're going to just run a process until it is totally done and then we'll load the next one.

They are not useful as anything that you might have attached to a monitor or keyboard because we want to make sure that if you're typing at the keyboard, that the process that is processing your keystrokes is going to get run often enough that your keystrokes are getting recorded and likely often enough that you see the characters that you're typing popping up on the monitor and you get that feedback to say, "Yeah, the things I'm typing are in fact being processed."

And if a human doesn't see that within a couple 100 milliseconds, they're going to start saying things are busted. So that alarm clock has to be going off often enough that even if you've got a bunch of people who wrote while(1) that you're going to get all the way around through round Robin to the end and come back to whoever's typing in their editor and get those keystrokes processed and the editor updated.

OK, so this is what we call preemptive task scheduling or a preemptive queuing discipline and FIFO is a very common way to do this.

OK, you can see at the bottom I say a preemptive discipline permits the job to be preempted before it finishes running.

The other common one that gets talked about in all the textbooks is this shortest job first strategy, which says if you have a bunch of processes available to be run, you will get more processes completed per second if you always pick the one that is going to run the fastest—the one that is going to complete in the shortest period of time.

Does anyone see the problem with that strategy? Yeah. If other processes allowed huge time passes, that would mean never ever run.

The shortest job first, while mathematically, provably the most efficient for getting more units of work done per unit time is in practice never used on any operating systems because of this problem. It's deemed to be unfair.

Now we may make some tuning up of FIFO to not simply mindlessly choose the same order all the time. We can do that, but we typically don't go all the way to shortest job first, so its definition is simply "pick whatever has the shortest total runtime."

Note you need somehow to magically know what the run time is of the process in order to do this as well. Something that is glossed over to some extent in the text. But because it's non-preemptive, that means that the long processes may never ever finish.

OK, so if we want to fix this up... Oh yes, I've said here FIFO and round Robin are considered to be fair, fair here simply means no one can starve to death and never run at all. A job is guaranteed to run at some point between now and the end of the universe, but we're making no claims on how long that's going to be. That's going to be dependent on how many processes are in the queue.

Obviously, if you've got a computer running 10 billion programs at one time, it's going to take longer to get around in the queue than if it's running two programs. So "fair" is a pretty loose definition of fair.

Shortest job first is optimal with respect to throughput if and only if you always run the one that's shortest time. You always somehow need to know which one is the shortest time. So that will always finish more jobs in a time, but it's unfair, impractical and not used directly. Though we can approximate it. And that's what we'll talk about next.

OK, so in general, if we have an interactive system, we're going to need preemption.

[Student comment - Ethan] It was actually not even theoretically possible, because in order to prove which job is going to run shortest, you have to solve the computability problem. Yes.

So they lost that entire... You're given a list of times for each of your process and they're implicitly assuming runtime is going to be infinite. But not just that in the sense that I believe that determining through the algorithm which program would end, or predicting its output, or running time... and that's what I mean by your loss. So in order to avoid that issue where it is not possible to definitively provide the program run time based on that.

Sorry, yeah, they're simply saying for purposes of the mathematical proof that shortest job first will get most jobs done in the shortest period of time, let's assume that someone has done that magic work for us and is simply provided the times at the beginning and they are correct. Then you can compute shortest job first.

You're probably talking about this in the algorithms course with the halting problem. It's a non-starter. So they're running the program. And it also will give us an unfair system.

OK, so preemptive schedulers will perform better. They'll have a smoother response, but when we say perform better, I don't mean be more efficient. I mean give a smoother user experience, than non-preemptive schedulers.

But we do like to have biases in here, so one of the things we would like to do from the shortest job first strategy is bias towards short jobs. So if we've got tasks like an editor. Editors are really short jobs, they don't complete in the sense that the program has done all of its work and exits. But they do complete in the sense that the program has done all of its work that it could possibly do with that last keystroke and is now waiting for you again on the keyboard.

So we can feed in a lot of editors and switch between them all. So if you were all logged in to the same Linux.socs server, you could all type on it at the same time, and if that's all that was going on, we could make you all believe that you were all alone on the computer and you had a nice, snappy response time.

But then if we throw an elephant into the mix where someone has written while(1), well, that's now suddenly going to take up the whole time. So to make it smooth for the typists, what we want to say is if we've got what looks like an editor, we can serve that one more often and every so often I'm going to have to just run that while(1), but I don't want to run it after every single editor or everybody will be slow.

So we want to be able to bias towards short jobs. It is possible that users can or the OR the root operator can set or adjust the priority for a job. There's a utility called "nice." Nice. You can adjust the priority setting of a job. So this is frequently used for jobs where the result is important, but you don't need it right away and you would like to not burden the other users on the system. You can set your nice value high to say "only run me when there's no one else in the run queue. I'm willing to wait. Everyone else is more important." This is why it's called Nice.

If you're root, you can also set negative values of nice. You're being not nice. You're saying, "OK, I am more important than anyone else. I should get more of a crack at the CPU than the regular user because I've given myself a negative value of nice. This program is important."

There's in all the current operating systems that priority is dynamically updated based on how long it took to run the last time. So if you've got an editor that inside the editor occasionally has a while(1) loop for a while, then its nice priority is going to go up and down depending on whether it was a hog or not the last time it was on the CPU.

There's always a trade off. So you can decrease the total time slice time. But if you do that, you have what we call fine granularity sharing. How many context switches per second are there going to be in fine granularity sharing versus if we make the total time slice big? And they can run for a longer time. Are you going to have more or fewer context switches per second if the granularity is fine and everyone only gets a tiny amount of time versus if it's coarse, then everyone gets a really good long run?

[Student] There will be more. Yes.

OK. Well, context switching is an overhead. Every time you do a context switch, you're running the kernel. The kernel has to do the juggler. Remember the kernel is not what you want to be running. Running the kernel is in a certain sense, a waste of time because you bought your computer to do important and exciting work on it, you didn't buy your computer just in order to run the Linux kernel or the Windows kernel or the Mac kernel or whatever kernel it is.

The more time the kernel takes running, the less time your computer has to do the things that you actually purchased your computer to do. So we have this trade off—the finer the granularity, the

smoother the feel, the slower it is overall because the more time it is spending doing the context switching.

The lower the granularity, the coarser, the more work each program gets to get done each time it gets a kick at the can. So if you've got a program that's chugging through some big task, "I'm sorting all these integers." It can actually get the sort done in bigger chunks because it gets more time to do work before it's yanked off the CPU and someone else goes in there.

So you're going to get more computing done per second the longer the time slice, or the coarsest the granularity, but you're going to see it in your interactive response, cause your editors are going to lag whatever else you're doing and the machine is going to lag. It's going to start getting choppy.

The gamers in the room probably know exactly what's happening here. When you get a choppy response in your game, so it does some stuff and then it has to stop because there's some other things going on. And then it comes back and you run ahead, right. So the same thing can happen in an editor, but editors have so little work to do that the machine needs to be pretty bogged down before you start seeing the problems on the editor side.

OK. So I'll just wrap up by saying, if the preemption occurs very frequently, then we call it processor sharing, which is effectively what is happening on all of your CPUs. Whereas if the preemption time is infinite, well then we have FIFO. Whoever gets on the CPU is going to keep the CPU until they're done, at which time they leave and the next one comes on and the process repeats.

Notice there we're assuming that everyone will get through because it's first in first out, so any late arrivals go to the back of the line. They will only get to be on the CPU if all of the earlier arrivals have already finished. It's fair. So there might be slow, but it's fair, and if there's no preemption, then all the process context switches that would have occurred because of the time slice just don't happen. We still have them for IO. So if you ask for IO, you can't do any work. You have to go back to the queue. If you don't ask for IO, you get to compute and then you're done.

OK. We'll leave it there for today. We'll wrap up the next two slides tomorrow, the next deck is the memory deck, so we will answer the question that someone asked a moment before about memory and RAM.