Revised Transcript - CIS3110-Class-Audio-2025-03-04

Audio File

CIS3110-Class-Audio-2025-03-04.mp3

OK, recording underway. Let's get this back up here. OK. Thank you for the reminder.

Yes, I don't know why I don't start.

Yeah. So.

The question is if Assignment 2 is not working as you would like, what to do? Because Assignment 3 says start with the A2 code. So A3 is about mapping the code from the A2 assignment structure, which is based on processes, to the A3 description, which is based on threads. So the grading for A3 will focus on the mapping itself; we're going to look at the changes in the code made in order to make the mapping happen.

Now, I am hoping that everyone has an A2 that at least compiles and does something, because we would like A3 to run. But if there is functionality missing from A2, you're not going to be penalized for that functionality missing in A3, unless we're talking about functionality at the level of your make file doesn't work or you have no actual functional code whatsoever. We are expecting you to have a program that you're mapping from, but if there are gaps - if you've got a deadlock, if you've got problems cleaning up, if things aren't working perfectly - don't worry about that because we're looking specifically at the mapping. Does that answer your question? Kinda. OK, yeah.

So OK, the question is, given the randomness in A2, how to help debug? One thing that I would recommend with any such issue is that you essentially remove the randomness. So one thing you could do is instead of using the random number generator, you could simply just have a list of integers that you're pulling out of in sequence, right? If you want to make it go in order.

Another thing - and I'm sorry this didn't come up until after the due date - I hope you are all aware of the fact that you can seed a random number generator. Is this something you've seen in your past courses or no? People are looking at me blankly. Have you used random number generators in past courses? A very small number of hands seem to be going up. OK, so a random number generator is a math function. It is what they call a chaotic math function, which means that each subsequent number from the function is very loosely, if at all, related to the previous number. But the way these all work is that it's just a function. You're walking through the function, so you can determine where you start in the function and walk forward in the function and get essentially the same list of random numbers.

When you call the function again, the way the random number generator works is it starts with what they call a seed, which is essentially an offset into the function. And then every time you call the function, it increments the offset. Now that we know about critical sections, I'm sure you can all see that this is a critical section, because if you've got multiple threads calling the same random number generator, they need to not step on whatever the current seed value is, so they will need to have a protection around that critical section.

But you can set the seed. If you don't set the seed, it calculates the seed for you based on something like the wall time multiplied by your process ID and some other little bits of craft that it can pick up from around the universe to make sure that the next time you run the program, it doesn't start with the same value. But if you set it - if you set the seed to 0 or 42 or whatever your favorite value is - you'll get the same set of random numbers out of a given random number routine again.

So to answer your question of how do you get the same behavior out of a program? Setting the seed will make sure that you at least get the same set of random values each time. And that is certainly one way that you can control for the randomness so you can reproduce the same problem. I'm sorry that we didn't - I didn't realize no one had yet talked to you about random numbers. I thought that had happened in an earlier course, but clearly not. Another note for next year. But in general, print statements are just going to be your friend. That's what's going to happen.

Does anyone else have a question on that? Yeah.

No, because they just finished A2, so I can set them working on A3 now, but it's unlikely that they will be done prior to the A3 deadline. They'll be back to you. I mean, as always, they'll be back to you as soon as I can get them back to you. But the TAs only have 10 hours a week to work, so they're going to work at 10 hours a week. They're doing advising, they're doing all the other things and prep. So they only have a few hours a week for grading. It's going to take a while to get through all the grading, so we'll get them back as soon as humanly possible, but no guarantee that will be before the A3 deadline.

Yes, we talked about that last time. I said I was going to set the partial mark for Apply for all questions at this time. So the update is that I've not received any naysaying, though the time period for that expired minutes ago, so therefore I haven't actually updated the midterm yet. Yes, we also noted there was an issue with a couple of the questions which I flagged, but I will update them all at the same time so that will happen probably after I post the TA regrading instructions later today, yes.

Yeah, so I asked the TAs to be in person. Someone pointed out that one of the TAs is holding their office hours during class time. That one will get updated so that they are in fact at a time you can go, but the office hours are now going to be in person for the rest of the term and for this week. And if people are making use of it for next week, we'll do office hours instead of lab times for this week period, which I think I put an announcement up about. Maybe I did not. If I didn't, my apologies.

And there are any further office hours now available.

I guess the breakdown, would you be like?

Sure, let's do that. Why can you not see anything now? Did you see something up there before? What is this doing? Was there anything on the screen earlier in the class? OK, what is this? There we go. OK. I think the previous class turned the monitor on.

So the question is: can I show you how I would approach this code? Is that basically the question? OK, so this is what should have come out of your zip files because this is what I put in the zip file that you got to start with. Obviously there's a README. The README is the same text as the PDF file that is on CourseLink because it is literally the source for the PDF file for CourseLink, so I'll skip that.

So as a student, I would probably come in here and start by looking at the make file. See how many things are being built, see what's going on. So I see in here we've got one target that has to do with a program. So good news, we got one program. Next to go into making the program, so these are the three files that are going to actually go together to build the program. They correspond to the 3 C files here.

I would take a quick look at a header file first to see what's going on, so that I can see what is in fact going to be in play in the other pieces of code. So here I see we have a structure. It's got a bunch of character fields in it. Here I see I have a node structure that has a bunch of ints in it. I see I have an array called shared_data made of nodes. Then I've got a bunch of semaphore-based identity values. I've got all the Linux semaphore nonsense.

This is a good time to say if you look at the process_demo versus pthreads_demo in the course directory, you will see that there is a different semaphore tool used in the Pthreads demo. That's not specific to threads, that's just that they updated the way semaphores are made available to you. The older semaphores is System V IPC. You always have to get a set of semaphores, even if you just want one. You have to get a set of one, and then if you need more than one, you can just say give me a set of however many I need.

So the way the older semaphores work, you have to do all this stuff saying which semaphore am I talking about and what do I want to do with that semaphore. The newer ones, you get one semaphore, and then it's up to you to decide what to do with that one semaphore. If you need more than one semaphore, you have to request more than one semaphore. These all just sit in memory. Nothing special about the memory. So as you're working on your thread assignment, where your semaphore is sitting in memory has to do with it simply being a structure in memory.

I hope it is clear to everybody that wait_sem and signal_sem are to be strong telegraphs of the wait and signal we've seen in the notes. These are just macros. You don't really need to know how they operate, other than they're doing a wait and a signal. The way they are operating is they're calling this semop

function. You can look up the semop function, you'll see you load up a little structure called struct sembuf and then you can make the semop function do something like a signal or a wait or an initialization.

OK, also in here we have a data structure called token_ring_data and we've got a bunch of prototypes. OK, in the various C functions. In main.c, this doesn't seem to have any comments telling me to do anything, so I would probably set that aside.

Looking at setup.c, coming through here, here's my seed for the random number generator. If you want to change the seed, don't give it time(0). Just give it some hard-coded number and it will run the same way every single time.

Here's the shared memory work. Looking through here, I see these three dots, which if I was to do 'make' are being pointed out because they are causing errors in here. I did actually notice this. So the errors I intentionally left in there to try to point you toward where you're going to need to do some stuff, because obviously 3 dots by itself isn't a valid C statement.

So first thing here I would be saying, OK I need some semaphores, then I need to initialize my shared data. Shared data was just mentioned up here so you can see we're doing stuff to set up the shared data, doing stuff to create the semaphores. Now we've inherited the work of initializing them. That takes us to this point. So we need to create children using fork. Everything else in here is provided for us.

Then I would turn to simulate.c, and this is where we see essentially the hard work of the assignment. So we know we're going to need this loop of values. We've got our array of nodes in our shared memory structure. In the previous file, we could see that the shared memory structure is what is being used to allocate the shared memory. So if we see the shared memory setup here, you can see that we're actually literally looking at structured data. So here we're going to need to use our nodes from structured data in order to do these various bits.

On the worksheet we've got "start things up", we've got "do something at the top of the loop" and then based on our state, we're going to have to do different work. And then down here I've set you up with "think about the work both in terms of sending a whole packet and think about the work in terms of how you're going to get a byte from one side to the other."

I hope you're all saying or we're saying, OK, I need to get something from one node to another node. We've talked about producer-consumer - that is a relationship to get data from one node to another node using a shared queue. This is 1 byte, so it's essentially a 1 byte long queue. Pretty simple queue. We don't need to do any modulus arithmetic to do that.

So I'd probably start by saying, OK, let's see if I can get a byte to go between two of my nodes using send and receive. Once I've got the ability to pass a byte from one node, that's the node zero to node one, then I could see if I can pass it all the way around the loop. Once you can move data all the way around the loop,

I would then start thinking about these more complicated state strategies where based on the state described in the assignment, we're going to need to do something different if we're at the point in our message where we're getting the "to" address or the "from" address, or the length or the data.

Does that basically answer your question? Yeah, so I intended to do that today with a little bit of sample here. So here is a bit of code that I put in in my solution. So the first thing at the top of the loop to start the ball rolling is simply to send a byte where I'm using a 1 as the unused token.

Then in the loop, first thing each node does is gets a byte from their neighbor, then figures out what to do with the byte. What you do with the byte is different depending on if you are the originator of the message. Right, so various people came by office hours or after class, and the one thing that I tried to make clear to everybody is that if you just have everybody doing the same thing, you're quite likely to build a deadlock because you have all of these identical nodes going around in a ring, just like the dining philosophers.

But the key to any deadlock issue - remember deadlock is because you have a ring in the allocation graph or a cycle in the allocation graph. You can't fix deadlock once you have the cycle, because everybody's locked up. So the important thing is to detect you're going to have a cycle or better yet, build your algorithm so that it never closes off that cycle.

We have in this problem the ability to say one of the nodes is always different from all of the other nodes because one of the nodes is the sender; it has all of the information. So there's a number of different patterns of how you can proceed through this assignment, but they all have to do with the sender - the one who has the data to send on the ring - behaving differently than everybody else on the ring. They know how many bytes are going to be sent. They know how long it's going to take things to come around the ring, if you want to have that in.

But basically all of these little things are going to have some kind of if statement saying "am I the sender or am I not the sender?" And if I'm not the sender then I'm going to do something differently. So you can see here, if you're the sender, sender deals with things in packet level; non-senders deal with things in byte level because the non-senders are just passing bytes along the ring. The sender is also the one who takes the data off the ring. So the same if statement that says "if I'm the sender" captures the "am I doing the cleanup" as well as the setup as well as all of the other heavy work.

So here we've got - and I'll put this little file up. I just made it up before class, but I'll put this little file up as an example. So you can see, here's the code to get things rolling. And again it has an if statement to say if I'm not the sender, then I'm just doing something different. So if I'm not sending, it's all about figuring out whether I want to become the sender.

If I am sending, then I've got an error message there because if you are sending and you receive the token, that is in fact an error. Right. There shouldn't be another token on the ring if you're sending because you

took the token away. So if the token shows up, that's a problem. And otherwise you just send your data.

Then here, "sending a packet" is very similar, we just do things at the byte level. And these are the two halves of the producer-consumer, and you can see it's just a standard producer-consumer. So we're waiting on empty or we're waiting on filled, and we're signaling on filled or signaling on empty, because it's just simply a pair of cooperating processes doing the producer-consumer relationship.

If you set the random seed to any constant, then you'll get the same sequence of random numbers each time. So what I've done here is I've used the wall clock time to get a different number if you run it later because the wall clock time will differ. Although one thing as we're thinking about multiple processes - this isn't a particularly good pattern if you've, let's say, got threads or processes, and they all are going to start with a random seed, because if you create a bunch of children or a bunch of threads, and then they all set their random seed based on the wall clock time, chances are they all run within a second, in which case they'll all choose the same wall clock time.

So the default random seed takes, as I say, some other information like what's your process ID, what's the drive hardware address - some other piece of information from the system to make it less likely that you get processes that happen to start at the same time that are going to have the same sequence of random numbers.

[If you want to use a hard-coded seed] any number will do, but once you pick a number, you can instead of time(0) here, you can put whatever your favorite number is - 12345. If you always start with 12345, then the sequence of random numbers you get by calling random will be exactly the same sequence as any other time you start with 12345.

So the question is, how do the nodes relate to the actual assignment structure? So in the description and in the intended solution, there's one process per node. And so if we see the setup here, you can see we make a shared memory region that has in it that array of node structures. And then what I'm assuming we do here - sorry, this is initializing the shared data. It's down here in run_simulation - so you can see here's a loop up to N_NODES and I'm expecting you to call fork once, one time per loop.

So that if you look at the assignment handout, that means that in this example we have 7 processes. We have the parent that started out the whole thing, and then the parent calls fork 6 times, one per round node. And then the square blocks are the shared data queue of one byte in length in between, let's say, node one and node two.

So then in Assignment 3, instead of having 7 processes, you're going to have one process with seven threads. And again, when you start a process, you get one thread. Can't run anything without having a thread. The thread is the structure that has a program counter, has all those things required to actually execute instructions on the CPU.

So when you run a process, unbeknownst to you in the past, when you ran a process, it created one thread to run your process, and that thread was the thing that actually ran all the instructions. Now you can say, using that one thread, I can request another thread and it can run other instructions at different times as we're running through this.

So we're going to move from one parent process with six child processes, each communicating through some data in one shared memory block, to one process with six created child threads, communicating with each other because all of the memory associated with your data segment and your text segment is now shared. You don't need to specifically allocate shared memory because the memory that came out of malloc or that is made of all of your global variables or shared or static variables - it's already shared.

You've got shared memory whether you want it or not. And this is one of the both advantages and disadvantages of threads: is that you're automatically needing to make sure you're not accidentally creating a critical section between two threads looking at the same variable because they happen to be running at the same time. So you have to decide which parts of the memory are actually intended to be shared, and the parts that aren't intended to be shared, either don't look at at all or have some kind of management around them to make sure you're not accidentally stepping on another thread's data.

Yes, we're going to cover some more content on this.

Any other assignment-related questions?

Then, as this fellow over here just mentioned, let us get to wrapping up memory and ideally into I/O.

OK. The last day we talked about the clock algorithm. So the clock algorithm was a page replacement algorithm, just like FIFO, just like LRU. The clock algorithm is simply a way of trying to use the use bits and the potentially the dirty bits by having a virtual hand going around the clock, picking a page found based on whether or not it has been marked as used since the last time the hand came around. The enhanced clock simply adds the dirty bit to that.

OK, so that was the last of the page replacement algorithms we talked about. What we want to talk about now is a page replacement policy. The page replacement algorithm is the algorithm to choose which page we kick out of memory when we need space for a new page.

So backing up a moment, we have the working set and the resident set. Remember, the working set is the set of pages we need to have in order to allow whatever our current portion of the code is to run efficiently. The resident set is the pages we have. The working set is what we need, resident set is what we have.

If they're the same, life is good. If the working set is a superset of the resident set - we need more than what we can have in memory - we have a problem. We need more things than we have space for. So

we're going to need to continuously be kicking things out of memory and replacing them in order to get that work done.

On an instruction by instruction basis, we can still proceed because an instruction is only going to look at a few bytes of memory. We can satisfy having a page in memory to satisfy those few bytes. But if we have a loop that keeps touching the same, let's say 10 pages, and we only have room for 8 pages, then every time through the loop we have to kick two of them out and bring 2 back in, and that's going to be expensive.

So if the working set is bigger than the resident set, we have a problem. If it's the other way around - if the working set is smaller than the resident set - then for that algorithm things are OK. But that means we've got chunks of memory hanging around that aren't doing much, which, if we have lots of RAM, might be fine. But if someone else has the first situation, where they don't have enough space, then us hogging extra space is causing a global system problem. Because every additional I/O event that is initiated because we have a page replacement happening that could be avoided is making the whole system slower. We're going to have work on the I/O bus, we're going to have additional work happening. Ideally we want to avoid that.

So on top of our page replacement algorithm, we have what we call the page replacement policy. So the page replacement policy tells us how we're going to apply the algorithm across all of our pages.

So a global page replacement policy simply says we apply our algorithm across all pages in memory, regardless of who they belong to. So that means that if process A needs a page, process B may lose a frame. Its data gets kicked out to make room for process A. So if we have a global policy, we simply say all of the frames are in one list. And as soon as we need an empty frame, we're going to pick across the set of all the frames for all the processes and apply our page replacement algorithm, let's say LRU, on that set.

So if the working set is smaller than the resident set, then that means that if assuming it is per process - I'm going to come back to this in a minute. So if you've got one process whose working set is smaller than its resident set - so let's say it needs 8 pages, but it has 10 - those 2 pages are taking up space. If nobody needs the space, that's not an issue. But if other processes need that space, then that is going to cause a cost because other processes are going to have a higher page replacement rate than would otherwise be the case. OK, so this is what we're trying to manage with these paging policies.

So if you've got a global policy, you're hoping to avoid that problem by saying I'm just going to eject pages indiscriminately depending on who they were owned by. I'm doing least recently used, I'm saying for whatever frame is in memory anywhere, whoever used it the least, that's the one that's going. So a global policy says I don't care which program the frame belongs to, I just put them all in one list and run my algorithm across the list.

A local policy, which is much more common, says a given program is given a certain subset of frames, and the policy is related only to those frames. This is particularly popular because different types of work may benefit from different policies. In particular, things like database joins where you've got two big tables and you're walking one slowly and you're walking one quickly. One of the algorithms that works particularly well in that case is most recently used because if you know the table's too big to fit in memory, then the page that you just touched is the page you're not going to need until you come all the way around the table again.

So you can therefore change your policy - your algorithm rather - for that process and that process only because that one's running the database, but someone else is running an editor and that would be a terrible policy for somebody running an editor. So if you've got a local page replacement policy, it opens the door that you can have a local page replacement algorithm.

So if you do that, then we have the problem that we just mentioned: someone might be hogging a bunch of memory. Some programs are big. Some programs are small. It makes no sense for us to say as a system-wide constant, everybody gets the same number of pages. Because if you've got a giant program that needs a lot of memory, it needs more pages than a tiny program that doesn't need much memory. So why would you say everybody gets the same slice in terms of the apportionment of memory they get?

So if you do that, a smart strategy is to be able to not just have a fixed size policy where you say "I load the process and I allocated a certain number of frames and that's the number of frames it's going to have forever until it exits," but rather saying "Maybe as this program runs I want to tune up the policy."

This makes particular sense if you think about the way programs run. Programs that hang around for a long time tend to do different work at different periods of time. Think about programs that you have running for hours. Microsoft Word is a good example. Usually you're just typing in it. But every so often you want to do something like print your document or run spell check, which is a whole different algorithm inside that giant program. And it may require a different amount of memory in order to efficiently do the printing process than you needed when you were just typing in your essay in the first place.

So being able to say "during printing, we're going to change the amount of memory this program has" and then later we'll bring it back when we go back to what we were doing before - that makes a lot of sense. A lot of programs work this way from time to time. They'll need a bunch more memory, then they settle down, do different work. They may need less memory, so we would like to, as an operating system, be able to adjust the amount of memory they get.

We don't know what the program is. As an operating system, all we know is that somebody loaded this into memory by calling exec(). It came out of a program file. We can look at some of the attributes of the

file. We can say things like how many kilobytes of program text are there? How big is the stack? How big is the data segment? But that's about all we know.

We don't know whether it's Word. We don't know whether it's a database. We don't know whether it is Assignment 2. We just know it is an executable file and we loaded it in memory. So we're gonna have to make decisions based on what the program is requesting.

So what we tend to do in a variable partition policy is we look at the number of page replacements - page faults - that the program is generating. If the page fault rate is high, this means that our resident set does not well reflect our working set. The page fault rate being high means that per unit time we're generating lots of requests for new pages. It could be pages that we put in the paging disk. It could be pages for new parts of the program text. It could be brand new pages. It could be that it's just allocating a lot of memory. But there's lots of page faults happening. That probably means it needs more memory.

So we generally increase the frame allocation a lot. And when I say a lot, I don't mean 10% more each time. I mean some usually multiplicative or maybe even exponential curve to say, if your frame rate is above some threshold, double the amount of memory this program has. If that's not enough, double it again.

If the fault rate is low, however, that means we have the situation where somebody's working set is smaller than its resident set. It's fine, it's doing all the work. Our suspicion, then, is that it might have a few pages, or a few frames that it doesn't need, so we want to reduce that, because if somebody else needs more, we want to take it from the one that doesn't need it in the first place. But we don't want to multiplicatively change again, so we're going to shoot the value up, but we're going to let it settle slowly. So that ideally we can solve the problem of the hungry process by giving it lots of pages, giving it lots of frames rather. But if it's been overfed, we just slowly sink until the frame rate starts going up. At that point, we know that the working set and the resident set are close.

Now, if you've got extra memory, [a student question] so you're doubling your allocation when increasing it, but when reducing it tends to be linear and in the small number of pages each time. So you're going to say, like, I'm going to reduce it by maybe 5%. But you may increase it by 50%.

But these pages have to come from someone. So if you've increased somebody's allocation by 50%, if all of the frames are full, you have to pay for that somehow. So you're going to have to gather that amount of frames across all of the other running processes. So everybody's going to shrink except the one you're pushing up.

OK, so now the question becomes what happens if you have extra memory? If you've got literally empty frames. This happens on your computers when you turn them on, right? You load the operating system. Nobody's running. You got lots of empty frames. OK. At that point, if somebody needs new frames, you

don't have to steal from anyone to feed them, so you can just feed them until you're running out of empty frames.

One of the things that has happened to a couple of different operating systems in the past is they'll say, OK, if nobody's using their frames, I want to just take away from them in case someone's going to need them later. So for a long time, Windows, if you left it on but you didn't touch it, would take quite a while to respond as soon as you type a key or move the mouse, because it would literally page out everybody.

So you'd have programs that were running, but they were blocked on I/O, right? So maybe you've got Word leftover from yesterday because you went to bed. It's waiting for a keystroke. It's not in the run queue. Windows would move the entire thing onto disk in favor of nobody at all. And now that this is on solid-state drives, it's a lot snappier when it responds, but it used to be quite embarrassing. You would come in to like any of the computer labs on campus, move a mouse and you would hear the hard drive just sit there while it tried to bring in all the programs because it had literally put everything on disk in order to satisfy nobody's request.

But if there was a request, it had a lot of memory to satisfy it with. So they got rid of the most embarrassing part of that because it would actually page out the operating system. It no longer does that, but it will actually page out application programs that are just sitting around in order to make memory empty. It's not a particularly great strategy. No one else follows that anymore, but Microsoft has still doubled down on it.

And they're just doing that because they're noticing the fault rate is low, because everybody's waiting on I/O. So they just decrease the frame allocation. Eventually they reach the point where to decrease the frame allocation, they have to start paging individual pages out on the disk.

But you can still do that. You can have your whole process on disk if it's not in the run queue. You need 0 pages in order to keep that a running process in the sense that it is loaded into the process table, because it's not receiving any runtime, so it's just an entry in the process table and a bunch of pages stored on disk. Everybody with me so far?

OK. Now we come to a famous diagram that is in every operating systems book. It is in the Silberschatz book. It is in the Tanenbaum book. It is in every operating systems book I've ever seen. This likes to tell you that there's this graceful trade-off between the number of page frames and the number of page faults, so that as you go down in the page frames, the faults go up, and as you go up in the page frames, the faults come down. And many authors including ours will say, "so you can try to tune for the sweet spot here where you're trading off memory versus time," which would be great if it wasn't a big lie.

It's a big lie because it is a very dated view of the world. Notice the number of page frames along the bottom of this graph. It goes up to six. Your frames on an Intel platform tend to be 512 byte frames. You

can configure them up to about 4K if you want to, but that's a pretty small number in today's programming.

I could generate this graph. So this is using real hardware. I needed a task that was a heavyweight task, so I just put recompiling the kernel in a loop. It would just compile the kernel, delete the files, recompile the kernel, and so yes, if I make the memory small enough, you can see this kind of trade-off curve. As soon as it leaves the X axis here, the hard drive light starts being on continuously.

So as you're climbing this, what we see is that below some critical point, we need more memory than we have available, and so it has to keep pushing things out of memory and looping around so that as it comes around in the loop, it's going to read back in that page. But it's cycling things to the drive in order to do the work. So the hard drive is just burning away.

But again, look at how much memory I have. Megabytes. 8 megabytes. How many bytes of memory do you have on your phone? On your smart watch? You have way more than 8 megabytes. So yes, this is true, but if we look at real computers - so here this is still in megabytes, but I take it up to 256 - you can see that yes, this is in fact an effect, but it's an effect that nobody really cares about anymore because it's cheap to buy more memory. We're now way over in the linear side of this curve. So what that means is if you have a situation where you do not have enough memory, it is going to go up in this exponential fashion, but we're not really talking about trading off on this round curve. We're in the situation where we can quite clearly see being down at that end of the line is a dumb place to be. You're going to have more memory available unless you've done something really foolish.

So this is really all about emergency planning. We're down at the part where your system is desperately trying to cycle. We need to do something different, so typically what we do is we feed processes that are hungry by increasing their allocation a lot. That's taking them from this tiny world into - we're just going to shoot you to the right on this graph to where things look nice and flat.

If we can't do that, if there's so much going on in this machine that stealing from one process to pay another just causes the problem to move from process to process, if we have this situation where things are so bad that we're churning away burning the disc there, then we do the activity that is formerly known as swapping.

Now you may think you knew what swapping was, but really what we were talking about previously was paging, which is confusing because many operating systems like Linux will have what they call the swap partition, in which they put pages that are ejected from memory because of page faults. So paging is taking an individual page frame and saving its data on disk. Swapping is this desperation strategy where we say "I got a bunch of running programs, nobody's getting any work done because all we're doing is we're burning through the disk, paging in and out, everybody all the time."

So instead of paging out a single piece of a program, I'm going to swap out a whole program. I'm just going to take somebody and say "you you have to wait on the disk until somebody else is ready to go." So you're going to take a whole program, throw it onto the disk. It gets no runtime for a little while. And you've freed up all of the frames that it was using, in the hopes that whoever is doing this ridiculous work finishes their work, exits, and you can go back to running the rest of the processes normal.

So swapping is this desperation attempt to try to keep what would be this exponential increase - I lost my mouse here - you can see this dotted line. If we don't do swapping, you're seeing exponential increases in time. And it's not only exponential, it's asymptotically exponential. You'll reach a point where you're just going up vertically. Nobody gets any work done because the amount of time it takes to do the page replacement is such that nobody's moving appreciably forward in any of their instructions.

You may actually reach the point - if you remember our discussion about satisfying a page fault when you had an invalid page - if you need to bring in a page from disk because your entry for that page in your table is invalid, and memory requirements are so high that by the time you've satisfied that and the process gets to run again, that frame has been re-ejected from memory, nobody will ever get any work done.

This is what we're trying to avoid, so the desperation effort is taking a whole program, throwing it onto the disk. It's still in the process table, so it is still eventually going to be able to run, but we've introduced a huge delay before it runs again, hoping that someone else is going to get to complete their work, or at least stop doing whatever they're doing that takes so much memory so that we can then make this degradation more linear. We're taking this line and pulling it down.

So swapping is always this desperation push - whole programs out. Paging is the normal operation of saying one individual frame needs to be ejected to satisfy the needs of one other page going into that frame. OK, so paging to the disk 1 frame at a time, swapping one program at a time. Swapping only happens in desperation.

Yeah. When I say program, sorry, [Student comment] I said programs. Technically we should be talking about a running process. You're exactly right. Program is the thing on the disk that we created the process out of. So I'm talking about an entire running process being put on disk.

And when this happens, they tend to do it 1 segment at a time, so the whole data segment, the whole stack segment, etcetera. Now that you know that threads have their own stack, you can probably see why that's a sensible plan, because you may take one stack of one of the threads and throw it onto the disk, but the other threads may be able to get some work done. Any other questions on that?

OK. So then some last tricks. If you have a page that is read-only, we mentioned before, it is cheap and easy to simply point at the same frame for that page from all the processes that share the same data. So if you call fork, all of the pages in your text segment are read-only, there's no need to allocate new frames for them because it's going to be the same data throughout the entire life of the program. So all the

copies of the program can be sharing the same read-only data, because no one can change it. There's no need for a critical section.

If you have a read-write page, however - so this is essentially a permission on the page, you have a read-write page which is defined in our page table by not having the read-only bit set - then that means that two different processes probably need their own copy of that page, because if they update the data on the page, they need their data. If it's a shared page, this is different, but for read-write pages, we're going to need our own private copy.

But this is where we can be clever. If no one has made any updates yet, then it's still the same. So we do this operation, we call copy-on-write. So let's say we've got our process; it's running. We call fork. We're going to need, let's say, new data on the stack. We're going to do something like probably save the process ID of our new child into a variable. So wherever that variable is stored within the stack segment, we're going to need a new copy of that page because both the parent and the child are going to update it and it needs to be different.

But all of the other pages making up the stack haven't been touched. We're talking about one int. Our pages are minimum 512 bytes. So there's no reason to copy the whole stack. We can do this trick called copy-on-write. So when you see the write operation occurring, a write operation to the virtual memory module can say "OK, this is a writable page. Have I made the copy yet?" If the answer is yes, then you just write to your private copy. If the answer is no, then what the MMU is going to do is it's going to say "Ohh, now I need my own private workspace." Now it's going to copy the data from your old page into a new frame so that the old page can stay with the parent process and the new copy can be with the child process and therefore they can both have their own copy, but you don't have to copy all the pages that you haven't yet touched. Which can be a lot of pages.

Right, if you allocate a giant structure at the beginning of your program and then call fork, if nobody touches most of the giant structure, you can share all that information. If you're about to call exec anyway, it would be a huge waste of effort to first copy every single page making up the process, and then literally microseconds later say "Ohh yeah, that was all wasted work. We don't need any of that information. I'm deleting it and writing over it with new data from a new program file." So this saves massive amounts of effort. So we only update the data in a writable page when a write operation happens. If no write operation has happened on that page yet, it is still a copy of the one the parent had, but if the parent or the child issues a write, they need to copy the data to their own private place.

OK, so that's copy-on-write. And we don't need that for read-only pages because we're guaranteed that it is going to be the same, because no one's ever allowed to edit it.

OK, so we have this read-only bit on each of the pages in memory. So far so good. Ohh yeah. So if you've got shared memory like you did in Assignment 2, if we specifically say "I would like some memory to be

shared between this process [excuse me] this process and some other process," you can either do that by creating it and then calling fork as we did in Assignment 2, or there's an example in the course directory you can see you can create it with a name, and someone else can say "I'd like that shared memory over there mapped into my process. It's called this," and then you can share it even though you don't have a parent-child relationship with that other process.

So if it's shared, you want the same frame to be mapped into the two page tables, because that means - I mean it's shared, so one process goes to the page in that frame, updates something; another process goes to the shared frame, sees the update. You don't need to save the data twice, you're literally using the same frame to represent the data in your two processes.

As a side effect of this, that means that if you allocate shared memory, the minimum allocation has to be rounded up to the page size. So if your page size is 512 bytes and you're going to represent that by mapping a page into two different processes, then even if you say "I would like 16 bytes of shared memory," what you get is 512 bytes of shared memory, where you're going to use the first 16, and then there's basically some padding that you're not supposed to touch. It'll just allocate some more memory there, same deal as if you get memory allocated through malloc - you're going to have to allocate a page at a time.

We talked about in the next slide, yep, from a process perspective, when running with a stack/data/text segment and one thread, they use that stack to run. If you have another thread, it allocates an additional stack segment. So each thread has its own private stack. But the data and the text segments are shared. That's not a big deal for the text segment because it's read-only anyway, but the data segment is shared and writable. Then it's up to you as the programmer to make sure that the multiple threads and the common data segments are not stepping on each other's toes, but you do have a private stack per thread.

But fork makes the whole process with the private data segment. The thread analog for fork is pthread_create. So when I put the notes up about thread versus process, and there's an example in the course directory called "threads versus processes" or "processes versus threads," you can see I've listed there - the analog for fork is pthread_create. The analog for wait is pthread_join. The analog for exit is pthread_exit. Yes.

So the purpose of copy-on-write is to save resources - to avoid doing work. So all successful computer optimizations involve postponing work that you're not sure you'll need to do. OK, so copy-on-write says "do not allocate a new frame and fill it with a copy of this writeable page until I know I need it" and the time I know I need it is when someone issues a write to that page.

So you need to keep a count of how many processes are attached to that page. If the count is greater than one and a write occurs, then you need to make a copy so that the write ends up in a private one, but

you've avoided doing work for pages that just never got updated, because you could have 100 processes all looking at the same page, and if the data never gets updated, it's never an issue.

OK. So to adjust the address space then, we can do two things. We can make the stack bigger. Stack grows down. Stack frames get allocated because you make a function call. So by calling a function, you may need to allocate more pages onto the stack to hold the local variables that are backing that function call.

So if you call a function and it has one int variable because there's a loop, it doesn't need a lot of additional memory. If you have a function that has a local buffer in it, because you're going to read lines and text from a file, you got a 16 kilobyte buffer, it's going to need some data, some pages somewhere to back that. So you're going to need to allocate pages into the process, growing the process page allocation downwards so that you can have somewhere to put those values.

Similarly, the data segment can grow. You start calling malloc. Your program gets bigger, it's just growing in the opposite direction. We're now growing upwards in memory because the heap simply grows up. So the function call is how we adjust the stack. When malloc or if you're a C++ programmer, new, or if you're a Java programmer, also new - if you're doing any of these things and you want more memory, there are kernel routines to do that.

Break sets the end of the data segment based on an absolute virtual address, you can just say "Yes, stretch my data segment up to this point in memory." Again, we save effort by not actually allocating the pages until someone writes to it, because until that time, we don't actually need the data. And then sbrk is just the relative version of brk, so you can basically say "I want to move my end of memory marker up or down" as I allocate or deallocate from the stack.

We know that in both cases, as you're mapping new pages into your process, for security reasons, we're just going to paint them with zeros. If you read from them before you write to them, therefore you're going to get zeros back.

That also means because you're moving the end of data segment pointer in bytes, but allocating in pages - even if you malloc a tiny string, you may need to allocate a whole page for that to be on. You can't actually say "I'd like part of a page in this process." You either have a page allocated to the process or it's not allocated to the process. So it's a very block-wise kind of action as we grow either the data segment or the stack, which is why, if you've had memory issues where you only walk slightly off the end of your array frequently, it's fine. Because the thing that is going to notice that you have a segmentation violation - that you've fallen off the end of your segment - is because you've gone past the allocated page.

So if you have a 512 byte page and you only have one valid byte on the page, you can walk 511 bytes after the end of your allocation until you go to the next page, at which point you'll get "segmentation fault: core dumped" because when the page fault happens for a page that isn't there, that's when you get removed.

This is one of the things that makes pointer-based programming so hard, is that unless you're using tools like Valgrind to peer in your code and make sure you're not doing anything wrong, the actual, let's call it the infraction, the punishment for going out of memory frequently doesn't happen the first couple bytes you do that are wrong. There's this rounding up that happens to the next nearest page before you see the problem.

OK. Yeah, we got 3 left. We can talk about this one anyway, because this had to do with Assignment 1.

OK so we have several algorithms. You explored them all in Assignment 1. If you've got, let's say your data segment, your heap, and you have to satisfy an allocation, we could do various things. We could walk until we found the first one that fit, even though there might be a little bit of leftover data. We could find the one that best fit. We could find the one with the biggest overhead, hoping we could fit more allocations in the big overhead.

What were your findings? Was it conclusive? Which one of these was better? Pretty tough to say. They're actually all better under different circumstances. If you have a program that is allocating and deallocating many structures of a fixed size - so let's say you've got a binary tree, all the nodes are the same structure - deleting a node and then reallocating a node, deleting the node makes a perfect size hole for that reallocation. If that's what you're doing, then best fit works really nicely because your holes are going to be the right size.

You seldom have a program that only does one kind of work, so we have this mixture of "I've got a bunch of records of the same size and maybe I've got a bunch of string work happening." Strings are awful, they're all different sizes. They're frequently tiny. Sometimes they're huge, they make ragged size holes. It's difficult to say what is the best. If you've got lots of strings, worst fit isn't bad. Worst fit actually works pretty well in that case. But first fit is generally good enough, because you don't really have that likelihood of getting a really good fit all the time.

So a lot of allocation systems will start looking at the size of the allocations. If they're a consistent size, or if they're an even base 2 multiple, it'll use a best fit algorithm. If they look like strings where they're small and ragged, it will just say "Yeah, not worth my time" and they give you the first fit.

Generally, [Student comment] so yes, so if you have a large number of allocations, just searching is a big cost. That's a very good point. And so again, they try to figure out what your data kind of smells like, and if it is a string, they're just going to say "Not worth my effort."

There's an additional one - the buddy system. It's basically binary search of memory. It used to be written up in every textbook as theoretically better, but nobody was ever going to bother implementing it because it was such a large, complicated algorithm - and then Linux went ahead and did it. So that's the Linux allocator. Is the buddy system. They're the only ones that bother. However, everybody else is using some variants of these, usually based on grouping different types of memory together.

OK, we got 2 pages of this. We'll finish up next day, then we'll talk about I/O.