CIS3110 Lecture 5 - Complete Revised Transcript

All right. Hello, everybody. So one of the questions that I've gotten that I actually meant to discuss last day is how the midterm is going to work. So next week at this time, we will have a midterm. Midterm will be in this room. So I will ask you, please do not come into the room until I and the TAs have had a chance to lay out all of the exams on the desk so that we're not all climbing over each other because I do want to give you the full time for the midterm, even though it is fully my intent that you do not run out of time. I do want it to be fair, however, so we want to make sure that we get a chance to get everything on the desk. And then you'll just come in and sit where there's already a midterm. OK, the content of the midterm will essentially go up to and including the end of this example that is on the screen that I was talking about last day. OK. So we'll finish this example. I'll come back to that in a moment. And I will say because I've gotten some, let's say, slightly peculiar questions. I'm not intending to pull a bait and switch on you. So the exam will be based on things we talked about in this room. Yes, there is a textbook. Yes, I have given you links to chapters in the textbook, but I'm not asking you to read and memorize the entire text. That is there so that if I explain something and you're still unsure, you can turn to the textbook for a second voice explaining the same material. So that you can better your understanding. So I'm not going to just choose textbook readings and make questions based on those textbook readings. Everything will be directly based on things we talked about in this room.

Right.

OK, that's clear.

So the exam. I'm not very happy with this. But because of the size of this class and everything else that's going on, I've had to make the exam fully multiple choice. Now. I don't like multiple choice exams because I see lots of people grinning because people tend to do well on multiple choice exams that I've written. I don't write questions where the provided answers are the obvious thing, something stupid, something stupid, something stupid, none of the above. OK, so I'm asking you to think as you go. I'm therefore expecting you to not rush through the questions because I'm expecting you to unpack what is going on in the question. But it will be fully multiple choice which does have the advantage that I can then get the grades back to you very quickly. This also means that I can get my TA time for marking your assignments so we can get those back to you as soon as we can. OK. Any other questions on the midterm?

But then the other thing I wanted to draw to your attention. Just before class and several times earlier today I got asked are there any review questions? Yes, they're in the part marked suggested exercises. So these are literally midterm questions that I have used, midterm and final exam questions that I've used in past versions of the course. Now they're a little bit mixed up because sometimes I went in slightly different orders. So if you see a question and you don't know what it's talking about, it is most likely because it is for the second-half of the course. If you're unsure, send an e-mail in to CIS 3110. We'll get you sorted out, but these are intended to be suggested review problems for you. There is a set of the review exercises and my strong suggestion to you is that you, after doing the review exercises, then turn to the review exercise solutions. Because if you just read the solutions, it's easy to fool yourself into thinking that your knowledge is deeper than it is. If you do the review exercise, it will then help you understand whether you in fact really knew the answer. OK. Any other questions on that?

OK. So then let's wrap up this example. So I also had a couple of questions about this kind of example and I want to do. I'm going to show you where the information flow is at. Because although it looks like we're doing a division. And it looks like we're doing a modulus. We're actually doing something far, far simpler. So if we have a number at the top, OK.

So if we have a number in hex. So I got a 0, I got A7, I have a 2 and I have an E. OK, what is the hexadecimal representation for 0 in binary? Sorry. What is the binary representation of the hexadecimal digit 0? Anyone it's easy. All zeros. OK, so 0000. OK 7. OK. But it's 4 digits, remember? So it's 0111. OK. What about the two? 0010 and what about the E? So we're not as used to looking at those higher numbers, yeah. 1110 it's easier to remember E if you remember that F is as high as you go, so you can kind of count down. If you are unsure, I will also point out there are only 16 of these. So you could just write them all out if you want a nice quick reference for them.

OK, so here we have a 4 digit hexadecimal number. Or a 16 digit binary number that is the same number. We have an 11 bit offset. 489-1011. So all we're saying is that the binary digits to the right of the 11th position are our page offset. And the binary digits to the left of that position are our frame number. So yes, that generates this number. So what is the hexadecimal number that I get if I combine all these bits together?

00 right. So I've got 4 bits, this bit and three of these, and then I've got another bit, so I need to put it somewhere so we can put it in a separate digit assume there's leading zeros. OK, so I know that my page number is simply these values. So yes, I technically have done a division, but I'm dividing by an even multiple of base 2. So the other way to look at this is I've just done a bit shift. I've just taken the bits and shifted them 11 positions, but I can say well if I throw away these 11, what do I have left? OK, so it looks like we're doing this fancy modulus and division math, but in fact all we're really doing if you think about the fact this is literally part of your computer is we're just keeping the data that are on these 11 wires separate from the data that are on these wires. We take these over into the part of our memory management unit where we're going to look up our page. OK, so 0 remainder is 72E well, that's pretty clear because if I ignore the leading bit here, it was already a zero, so this is still a 7. This is still a 2. This is still an E because we're shifting by 11. These digits are never going to change. The only one you're going to get any kind of chopping up is this digit. So if this was, let's say, an 8 instead of A7. You're going to lose this bit and only keep these three in your offset. OK. So we've just literally said give me those 11 bits. I'm going to keep them aside. Give me the other bits. I'm going to keep them as the page number. So yes, our remainder is

72E mathematically, but that's simply because we literally kept those bits that were in this part of our register. OK, so far so good.

Well.

Page 0. OX00. Not in memory. OK, so if we get page zero, we're going to go to our page table. So remember, we just take our page table index and we walk up until we find the right page. We don't have to walk very hard or far because we're looking for 0. Right, this is just an array, so it's an easy offset based indexing scheme. We just simply walk to position 0. And we look at the data in the table. OK. Valid bit not set. That tells us what. If the valid bit is not set. Are we looking at like a seg fault? What are we looking at? It means it's not in memory. It doesn't mean it's not part of your program. It just means that the memory management unit has no idea what to do with this page. So the memory management unit sees or uses all of the data inside the dark border box of the page table. So what we have here is what we call a page fault. We've tried to access our page, it wasn't there. The memory management unit gives us a fault, so to implement the page fault, the memory management unit, which is simply an IO device, raises an interrupt. Says yeah, you asked me for some stuff. I can't do it. I raise the interrupt. Oops. Sorry. Let's. Uh. So we have a page fault, but the page is on the paging disk, so we raise an interrupt. We then can go over here and look in the extended information. We say, oh, I do actually have data for this page. It's just not in RAM. OK. So when you've heard about swapping or paging or your page disk or your page area. This is what they're talking about. OK, so you have this extra information for each of the entries in your individual programs page table. OK, so this is independent per program. This particular process did a virtual memory read at this location called the page fault. The interrupt handler for the MMU is going to say what can I do about that? Oh, I have data. But it's not in memory. OK, So what does it need to do in order to make that data be in memory? Because we can't continue to run the program.

Alright.

Until we can load the important page information into memory, into a page frame. So what do we have to do to make that happen? But we got to do some IO. Right. Literally, we're going to have to read some disk information. So we have to say, OK. Location 5 on my paging disk, so 012345. This is saying process ones page zero with the read only bit set. I've marked it here, so we need to take this data. And copy it into RAM. If we copy it into RAM, we can restart the instruction and it will then succeed. But this is an expensive operation, right? We're talking about playing around with the whole disk, so we're going to schedule some IO with this, right? The CPU scheduler is going to run some other processes, a bunch of stuff is going to happen. Eventually we can copy this page up into somewhere in RAM. Let's say we decide to put it in location 16. I chose that one because this location happens to be empty. That seems like a good strategy if we've got extra RAM hanging around that no one's using, let's use that. So we say, OK. Let's use 16. We load it up there. We're going to update the page table so we have to write to the page table to update this cell in the page table so that it now has the 16 in it and then we just start again from the top.

So from the CPU's point of view, it's trying to run an instruction. You know load from memory, add whatever the instruction is, some instruction that needs data from memory so it starts again from the top. It tried to do its instruction and it got interrupted. We are backing up, we're going to do the instruction again from the beginning. So we simply say, OK, I got a page offset of 11 bits. Remember that is a hardware constant. You buy a CPU, it is going to have a page slot. So if you buy an Intel CPU, it's probably 512 bits. You buy a Sun, like 11 bits or sorry 512 bytes, or 9 bits. Sun likes 11. They're Oracle now. It doesn't really matter, but each CPU is going to have built into it. Right baked right in. Here's how big our pages are. All the pages are going to be the same. That's just the size we deal with as our block of memory. So we start again. Notice it's exactly the same address. So unsurprisingly, the math is exactly the same. But now page 0 is in memory. We updated the valid bit too, so notice that when we updated the page frame with the page, we also updated the page table with the indexing information to find it and we updated any of these bits that are important. OK, so the valid bit is going to get set because we've said it's now in memory. It was a read only page, so we're going to say this is read only. We're doing a read, so that's good. What would happen if we were doing a write? You've all seen it happen. What happens if you try to write to a null pointer? Segmentation faults. So if the MMU sees that it is a write operation and the read only bit is set, you get a page fault. The interrupt handler says oh, that's not allowed and we take that program out of the run queue and it is no longer running. It crashes. So we then deallocate all of its resources because it's done something illegal, yeah. Sorry. Because hexadecimal 800 is decimal 11. It's just the same number. Yeah. So the reason I wrote it out in hexadecimal is so that it is clearer to you where the zeros are in this number right? Because in hex there's only one bit on in an 8. So if you divide by hex 800, you're effectively doing a rotate, right, because you're taking away all of the bits in the binary number that would be in these zeros, and the three that were part of this 8 until you get to the significant bit. OK, so page 0 is now in memory. So we say well page 0 is in frame 16. So again, it looks like we're doing math, but all we're doing here is we're saying, OK, hex 16. Well, what is hex 1 in binary? 0001 OK, what is hexadecimal 6 in binary? 0110. The four bit, the two bit. OK. And then all we're doing, even though it looked like this was all this fancy math, all we're doing is we're just carrying these values down. So the actual memory address is going to be quadruplets of this information. So the last one is obviously still the E because we're not really doing anything over there. The second last one is still the two. This column is these three bits from the offset plus this bit. Well, the Hex 16 still had a zero in that condition, so this is still A7. And then this value 00001011. So what is 1011? We've got an 8 in it. So as soon as you're in hex, if the first bit is on, you know it's the high half of your number range. OK, so we've got an 8 in it. No fours. A 2 and a 1. So what is 8 + 2 + 1? Eleven. OK. What is 11 in Hex? B. So we just go to memory location B72E. That's where our byte is. Now, if we're doing word based addressing, the only thing we do is instead of collecting 1 byte, we collect a whole word starting at that location. But one of the nice things about having a page size that is an even base 2 multiple is that your words that are even base two multiples will fit on a page in a nice integer relationship. So if you've got a four byte word on a 512 byte page, it all lines up nicely. If it's an 8 byte word, it's still good. If you somehow have a word that is bigger than your page, well then someone built your computer wrong because that would be quite a peculiar situation. So we're just taking the page and we can think of it as one buffer of bytes or a buffer of words. But if we're doing an operation on a given data type char, whatever, all we do is we just collect the data starting from that location and walking upwards in memory until we get the number of bytes that our operator demands. So we're doing an add of a, let's say, a long then it's going to collect 8.

Does everybody see what's going on here? It looks very complicated and mysterious and I'm always really glad that this lecture always takes two days because it gives you some time for things to sink in from the first round because it's really not that complicated. All we're doing is we're chopping up binary numbers. Keeping one side for one purpose, one side for another purpose. Look it up in a table. OK, we've all done that stuff in the table we find. And this is literally the most complicated thing, which is when you have a page fault and the page fault is when the data is not valid or the operation is inconsistent with the protection. So if you're trying to write through a read only page, you will be summarily removed if you go try to get page A. This is also a seg fault. You've fallen into the gap in the middle of the program, so you've got some bad pointer. Whether you're reading or writing, there's no information to give you. So the MMU is going to fault. The handler for the MMU is going to say I have no data buddy and then you're just going to get descheduled and your program crashes. As long as you stay within the bounds of the memory that you've got attached to your program, you're fine. And every program we've ever written does exactly that. It's just jumping around from location to location. Any order doesn't matter. But for every single virtual memory address, we do this little calculation daily to figure out where in physical memory that is, because remember the CPU can only see physical memory. So if the CPU is doing an operation and the data it needs isn't in memory, the CPU cannot complete that operation until we the kernel do the work to put that data in memory, and then the instruction can restart. At that point, they presumably program can go rattling on for a while, unless the very next access is also to invalid or causes a page fault. Right, but each instruction is going to get things satisfied or not because the data is in memory. There is a separate page table for every process. So each process can have its own map from its memory landscape into this combined storage of RAM and the paging disk. So notice I've got other processes numbered here P0, P5, PA. So they may have pages in frames in RAM. Right. I mean, you're familiar with this, you run multiple things on your computer at one time, they all have to have different parts of the memory allocated to their purpose. But as long as the entry in the page table is for a unique frame, then they all have totally separate memory spacing. They can't hit each other. No matter how badly any of them are programmed, there's no way that anyone of them can cause a problem for any other one. It can cause a problem for itself. But it can't do something to cause its neighbor a problem. This is what we call protected virtual memory. Now if you go back a few years, there's a lot of people in here with Macs, Mac OSX. It's an X because they found they couldn't trademark numbers. This is why all the Intel machines after Pentium started having funny names. So Mac OS X the X is meant to be a 10, but it's also meant to be not a number, so they can trademark it. And at that point, Apple decided that their operating system was having too many problems, mostly because its memory management was crap. And so they went and got free BSD Unix and repackaged it up, and that's what OSX is is it's an installation of free BSD Unix that they've extended. Mac OS9 had non protected virtual memory. One global page table with everybody in it. And you could accidentally stab another program in the heart by having a bad pointer. Which meant it was very difficult to have robust computing because anybody with a badly written program on their computer, as soon as they ran it, something else would go badly. Right. They would literally overwrite some other program so old Windows was like that. This is why Windows NT family came around. They got rid of their unprotected memory at that time. The old Windows 95 and the even older like Windows 3.1, Windows 32S that was all unprotected virtual memory, but protected virtual memory gives you a private page table for each one of your processes, so they cannot stab each other in the heart, on purpose or accidentally, unless you specifically ask for shared memory, we're doing that in our assignment. How do you think you do shared memory? I mean shared memory. You literally want part of the process address space for process A to have the same data to be shared with the process address space for process B.

Yeah.

So you don't actually share a page table, but what you have is a common frame entry in your table. You just have a frame where it's the same physical frame mapped into your two programs. And so they don't even know its shared. The one process does its math, lands on that page, updates value, does some other stuff. Next process runs, does some math, lands on the same page, reads some data. This is why it's our responsibility to manage the sharing with semaphores, because they have no clue it's even happening. The only way a process would be able to even figure this out is if it very carefully looked at all the memory locations and noticed that some of them were suspiciously changing without it actually doing the change. But you'd have to like peer through memory. That sounds like a huge waste of time. So we can manage the sharing by allocating a shared memory block that just is simply mapped into our two processes or more processes, you can have as many sharing the same pages as you want. But now it's a critical section, so it's up to us to make sure that it's managed properly and that all the writers and readers of the critical section are playing nicely with each other.

Yep. Yeah. Oh OK so. So the question is how I ended up with the specific bits in the last line of the exam. OK. I think everybody's with me that this one becomes E because they're the four bits from the right. And then all I'm doing is repeating the next 4 bits are two, the next 4 bits span our division line. So I have to take a bit from the other side of our eight or what was it, 11 bit offset line in order to get a 4 bit combo so I can write 1 hexadecimal digit because one hex digit is always 4 binary digits. So I got the seven by saying I'm going to take the bits that were at the high end of my offset plus the one bit from the low end of the frame number because I need 7 bits in order to write it down as one hex digit. And then the next hex digit just has to start where the first one left off. That's how it became AB. So the 16 became a B and part of the seven because we need 4 bit multiples working from the lowest significant bit.

OK.

Over here to follow up on significant. We have a number. Let's say let's say we had 0110.

This is my binary. If I just write down a binary number 101110. If I want to write that in Hex. I have to say, starting from the least significant position, I pull 4 bits off to say this is an E and then I pull 4 bits off here. So effectively I add 2 zeros and that's a 2. I don't get to say 101110 and say I'm going to pull 4 from this side. Right. And then but you can't have. If you need more bits or if you need fewer bits, even if you need more digits in a number. You can only add zeros on the most significant end of the number. That's why we have to start from the right to move over and that is what forced us into this peculiar seeming situation where I needed to take the bit out of the six to incorporate with the three bits I had here. Right. Are you are you seeing it now? Yeah. OK, there was a question.

OK. Yeah. The whole thing. Yeah, yeah, probably. After you go from one step to another step.

But certainly I could in theory say 11 bit offset read 72E. Here's your page table what happens but in a multiple choice exam, it's very difficult for you to have an articulated response, right? So you're just going to be able to say A or C or whatever. So it is more likely that I'm going to say things like given an 11 bit offset and this page virtual address, what is the page number? In which case the answer would be 0, which presumably could be one of the things written down for that question. Yeah.

Everything determined. But I will say that I have noticed that the likelihood of making an error doing that is much higher than doing it in the hexadecimal. Right. I would just literally do this, write them out, draw some lines. It's much easier to see, but at the end of the day, as I say, we are technically doing a little bit of math. So you're just dividing 2 numbers. If you correctly map from this hex number to a decimal number and from this hex number to a decimal number. And then correctly map back and forth all the other ones and have the right thing at the end in order to recognize the hex number I give you the answer. There's no reason it wouldn't work. I'm just saying you've added like six or seven additional steps, any of which can go wrong. But yeah, I like. I won't even know because you're going to color in a bubble and I won't know how you came to be. So I can't possibly say ohh you shouldn't have done it in on whatever base you wanted to do it. You could do it in octal. You could do it in a base pie, whatever. What? What? What? Whatever you want to do. But I think it will be. I I'm hoping that you see that in binary we're actually doing something really simple here. And so, because we're doing it simply in binary, it therefore is in your advantage to do it in binary, because we're just literally copying numbers down.

The page at that point. Yeah. So this is having been registered, it's just a wired connection.

Right.

There are registers. In the end I don't know how to answer this right. There are registers that are the interface that we have to do, but yes, it's in the register. But yes, it is at some level, doesn't matter. Reaction like from that read curve, we simply take the 11 bits of offset and put it in another register combined with the value we got on the base register. Right. That's the important part is you look up in the page table. The MMU has to do work in order to index in here, but because it is offset based indexing, it's not like it's searching. It's just going to take your number and it's an array, so you just step to the right part of the array so it is nice and fast. OK. Are we good on that? OK, so then now entering the part that is not in the midterm. OK, so page fault handling. When we get a page fault, what are we going to do? Well, First off, if the address is garbage, we just terminate the process. There's no legal way to continue because it has asked for something that doesn't exist. There's no way we can satisfy its request, so we take it away. Otherwise. Now remember this is the page fault, so we've calculated our virtual address and we've gotten to the situation where the valid bit wasn't set and we're trying to satisfy what to do when we went to the page table and the valid bit wasn't set. So if it's invalid, terminate the process. Otherwise, if there's a page frame in the allocation amount for your process, we're going to talk about that again shortly. Right. So big processes don't just get to steal all the memory. Right. We're going to have what we're going to call the allocation. So we'll say you can have, you know 35,000 pages and you can have 63,000 pages. So if you've been allocated a frame and that frame is empty. Well, we just use that frame. Great. We don't need to do anymore work. Otherwise, we have even more work to do. Because that means that all of our frames are full. But we need a frame to put this page in. So we can store a page onto the paging disk. Once we've stored it onto the paging disk, then it is no longer required in memory because we can use that memory for something else. So we choose which page is going to get replaced. If the page has been modified, so this is what that dirty bit is for. If there's been a modification to the page while it is in its page frame, then we need to save the current copy of that page or we will lose the information that got updated on that page. So if it is a modified page, we need to save it on the disk. OK, if the page we need is in the paging area, then we just schedule a read, we read it in into our now empty frame from the disk paging area. Otherwise, we can do some tricks. If the page is a text page, so it's read only. No one can play with that content. We don't need to fill up the page area with copies of perfectly good information that already sits in program files. So we can directly page the information for our page frame out of the executable file that the program is stored in. OK, let's think about that for a moment again. So in our process image, we've got our text segment, our data segment, our Stack segment, the data in the STACK segment can be manipulated. We do that all the time. You make a variable, you change it has a new value. OK, so data pages that make up the Stack segment can be modified by our program and therefore if we're going to eject them from memory, we have to save that information so that we can bring it back later. The only place we have to save the information is the paging disk, but if it was a text page, so if it's a instruction, a page full of instructions from the bottom of our process image. Then we don't need to store that in the paging area because it is already available to us in the executable file that we ran to get this program into memory in the first place. So if we need a page from the text segment, we can just reload it from the program file. So that if we need to eject one of them. We can just delete it from memory and reload it again from the program file. So if it's a text page or it's a like initialized data page, so that's shorthand for constants like if it's your string table or something, you can also just load it from the executable file. So far so good. Otherwise. We have what is effectively a new page. So if you're running a program that called malloc or whose stacks grew. You may well need a page to put the new stack frame on or need a page to satisfy the storage for the new malloc allocation, but that's never been part of your program before. So we can map the page frame into our program, right? That's fine, we say, OK, it is such and such a page frame, I'm going to map it in my program, update the page table. But if this page used to be in someone else's program, that can be a huge security risk. What if someone who was editing the password file and now you've suddenly mapped it into some users program? Right, here's a bunch of passwords, right? You don't want that. So for security reasons, if it is a new page that we're adding to the allocation in a given process, we zero it. Leading to the mission of all first year C programming students, that variables are magically filled with zeros when you run your program. This is why if you run a program that allocates and deallocates memory and reallocates memory, it may have garbage in it, or if the stack grows and then shrinks and grows again, while the second time it grows, all those zeros are filled with whatever your program had, because that's legal. That's just information from elsewhere in your program. So the only time we do this painting with zeroes is when we're putting a fresh page into a running program. So we just ZAP it all with zeros and then we're good to go. So then we set up the page table, stitch everything together, Mark our process runnable, and now it can be scheduled like any other process. When it comes around in the run queue and is selected again, it will restart whatever instruction it was running. Presumably this time it is likely to actually get to do its work, though I will say of course some instructions have multiple memory operands, so you could easily fault twice, so I'm not saying let's say multiply instruction if you're multiplying two different values from two different pages that weren't there. Then we just have to go around this Mulberry Bush twice. So you fault with the first one, you run it again. It would fault again. You have to satisfy the second one and then we're good to go. So far so good. OK. So if we think about the timeline of the page faults. So if we've got some process process X, it's running along. And it's trying to do some virtual address mapping and that caused a page fault. OK X, can't run. We need to satisfy the data demands for the fault before we can run process X again, so we have to set X's status to be blocked. It's no longer in the queue of runnable processes it's waiting on IO. So it's going to be blocked until that IO comes back. It's waiting potentially a long time. It has to wait for the needed page to be loaded maybe other steps cleaning up page tables, initializing things, whatever. But other things might be runnable. So we run them because why would you just have your CPU perched waiting on someone's fault? So you get to run anybody who's not part of this, this blockade that we have here, if nobody's runnable, then the systems in a state we call idle. The run queue is empty. So some operating systems have a default task that it does other most of the ones you're going to see now will just literally halt the CPU. There's a halt instruction until one of the wake ups gets called on the IO, and then that CPU will wake back up. So you can actually save power by putting your CPU, not just the thread but the whole CPU will go to sleep until there's some, it can be done very nice, especially on a portable device, so if you got a phone or something like that, you don't want your CPU burning up cycles generating heat and wasting time you want it shut down in a sleep. OK, no matter what happens though.

Yeah.

This is enormously expensive. Because we have to run the interrupt handler for the MMU, figure out what we're going to do, find things from memory, probably load things from disk so it is usually thousands of instructions worth of time. You can do a lot with thousands of instructions. So then it becomes important for us to think about if we do some smarter processing, can we decrease the likelihood of a given process needing to do page faults? Because if we can come up with a scheme where it is more likely to still have the data it needs in memory, then the whole system runs fast. Because you're not even invoking the kernel for the page fault, every page fault saved is a vast savings in time because you don't have to invoke the kernel at all. Approach. The process just gets to run, so therefore it's worth spending some effort trying to figure out how to decrease the number of page faults. OK. Paged virtual memory has some interesting properties. So when you write a program, when think about the way you allocate data in a program, if you've got, let's say a big array, where are all the memory locations of your big array relative to the other memory locations in your big array. Are they all higgly Piggly? No. They're consecutive. They're nearby each other. OK, this is starting to sound good, because if they're nearby each other, then as you walk across an array, it's very likely that the next data cell you need from the array is going to be on the same page as the last data cell you needed from the array. So that's a good thing, because you're really unlikely to need the page faulted in if you just had it a second ago. And we also have the issue as I say here. That not only are there adjacency issues. But the parts of memory that you actually are using at any one time tends to be intimately associated with that task and not other tasks in your program. OK, so if your program does, if it has several features, if it does various things. You're only going to require a small portion of your virtual address space to satisfy the needs of one of the algorithms for one of your program components, and we can make use of that. So the set of pages that makes up the data requirements for some programs current algorithm, it's called the working set. Depending on the program, the size of the working set might vary enormously. Right. So if you just got, I don't know, a couple of strings and your formatting and reformatting the same 2 strings, maybe they're on one page. If you've got some massive array data structure, half a GB of of information that you're sorting. The working set is much larger. But the set of pages you need to get the job done is what we call the working set. OK. In a real physical computer. We have some RAM. It is of a fixed size. That RAM is going to have a certain amount of it allocated for page frames. Of the amount allocated for page frames on that computer a certain amount, the allocation for this process is set aside for the use of a particular running process. That's what we call the resident set. So the residents set is what you got. The working set is what you need. Obviously there can be mismatches. So if the resident set is bigger than the working set, what does that mean? Let's say it is a big superset of the working set. Anyone. Remember the resident set is what you got. The working set is what you need. So if the working set contains the resident set and more, is that a problem? I mean, clearly your program can run its resident set, satisfies all the things from the working set. The problem here is that you're using RAM that someone else may need. So if you've got lots of extra pages that you don't actually need, that's not necessarily a good strategy because somebody else's resident set maybe smaller than their working set. If the working set is big and the resident set is small, we're going to have lots of page faults. So what we want to do is we want to as much as possible make these the same. OK, so here's a figure from the textbook. This is in the 8th edition. I don't know why I picked the 8th edition. This is figure 919. It's in all the editions. All this is is it's a memory map. So the vertical axis here is page, or sorry, is page not page frame page. So if you go up on the vertical axis, you're going across your virtual memory space. If you go across the X axis, it's just time. So what this is showing us is that as this single program ran different parts of its address space came into or out of play. And you can see this great big empty space here in the middle. But then as soon as we're 3/4 of the way across, boom, it's just doing all the work in there. You can also see these diagonal lines. What do you think's happening to generate these diagonal lines? Sorry.

No. That was one process one. But if you think about the memory map, what happened in the very last. If you were drawing a straight up the leg with you. Probably half.

What we're seeing here is that we're doing probably some kind of comparison between the data in this array and the data in this array and it's iterating in two for loops as it goes across these. There's some additional funny business going on here in the middle. Here's a smaller array iteration. And then these great big blocks, it's just accessing all of that memory. It's doing some huge memory transform. But you can see that not all of these regions are even used at all. Like there's some here that seem to be totally omitted. And other ones only start being used when the other algorithm's done, so this is probably some kind of join. Right. Doing some sort of data join and then once you've done the join, it's doing whatever it needs to do with the other data structure. So it's done the the hard work of the join and now it's saying OK, based on that I have some other work to do down here. OK, So what that says to us thinking about the resident set and the working set is what. Is the working set the same over here on the right as it is kind of in the middle where we're doing the joint. Who says it's the same? Who says it's different? It's different. OK, so here we need these pages to do the work, these pages need to be in memory, but these ones where it's all just blank, they can be wherever they want to be. And then as soon as we finish that work well, we can throw them all away. We don't need them anymore. But we do need these ones. We better get them into memory quickly so that we can get that work done. So our working set changes over time based on what? So here we have a single process with differing memory needs overtime as that process runs. This is just the run of one program, right? You've all done it a million times. OK, So what we want to do is we want to make an algorithm to choose which page gets replaced such that the resident set magically stays the same as our working set. The ideal page replacement policy achieves this optimal behavior. OK, so if the resident set in the working set are identical, we have no waste. We have everything we need. Life is good. OK, so if you come up with a page replacement policy, you can figure out whether it's a good page replacement policy only by calculating how many page faults you get when you run real programs. So the page reference string is just the list of all of the page numbers that are being processed in the MMU. So each memory access is going to generate a page number. We write down all those page numbers. We can say for this program these are the page numbers we got. And I think you can probably see the different programs may have wildly different patterns of their page numbers. So a database that does joins all the time probably very different than, let's say, the word processor. Right, a word processor you're typing at one place. You tend not to teleport around in the in the program. So you're typing at that location. You're storing all your data relatively nearby, so we take a real program. We record what the page reference values were, and then we can say how does that work with my strategy? OK. So we'll look at 4 strategies here. The first one is an optimal strategy. That sounds fantastic. The only problem with the optimal strategy is it requires perfect prediction of the future. Which is not likely to happen on our computer, but if we have stored the page replacement string or the page, I'm sorry, the reference string of a given running program, we can use the optimal strategy to say how good could we get. Because it's not possible to have no page faults. I mean, after all, we have to bring some of these things back in. So Belady's optimal replacement says using future looking skills. I can look forward in time and say with perfect knowledge what is this program going to need next? The sad reality is that we don't get to do that. We only get to see what has happened so far as an MMU. So our other 3 here, one of them is plain old FIFO. Replace whatever page has been getting old and moldy and sitting around the longest. OK, LFU, or least frequently used, says, well, if we're talking about a given page a lot it probably makes sense to keep that one. So if you've got a page that got used very infrequently, that's probably a good one to throw away. Right. Maybe it just has some, I don't know tiny reference at the end of some giant loop, so we could count how often the page has been referenced. Least frequently used, says jettison the one that has been referenced the fewest times. Now least recently used simply says replace the page whose last reference was the longest time ago. We haven't needed it for the longest period of time. Therefore we're going to get rid of it. So far so good. OK. There's hardware support for some of these because this sounds like lots of big, expensive data structures, and especially on small processors. Maybe we don't have that much RAM, so there are other things that we can do. One of them is just to have a use bit. So most MMUs have a bit that gets set every time you do something on a page. As soon as you touch the page, it's marked used. At a given period of time, that will tell us that some of them have been used and other ones haven't been used. And if we have some kind of alarm clock type strategy where we wipe them. Then, if we're lucky, the use bit will be set on some of them and not on others, and we can say, yeah, that's a really crude approximation of LRU, but I can say anything that hasn't been touched since the last time I wiped them all out clearly is an older reference than anything that has been touched. So there's a bit in the MMU. If it's clear that means that the software who's cleared it. The software cleared it and then no one has referenced that page. As soon as the hardware accesses the page, it gets set to 1. So software does the clear hardware does the set. So we have some regular clock. Every time the clock rings, we wipe out all the reference bits, and then when we need one, we simply look for somebody who's clear. Pick that one. If we're unlucky, we just reset them and they're all clear and we have no information. If we're also unlucky, they've all been touched. We still have no information. OK. So here's an example of FIFO. So yeah, this is actually a midterm question I pulled from some years ago, so 64 bit machine, 64 kilobytes of RAM, page size of 512 bytes. This page reference string. How many pieces of that information is relevant to this problem? If these are the page numbers. Does it matter? It's a 64 bit architecture. If I'm telling you that it referenced page, ABCDAB, etc. Is anything about our mask going to be perturbed at all by the fact that it's a 64 bit architecture or an 8 bit architecture. The relevant information. We're just talking about a page reference string. We don't care how much RAM we have, what we do care is how many frames this process has been allocated. So this is a red herring because this is telling us how much memory is on the computer, not how much memory has been allocated to this process. We need to know how many process, how many frames have been allocated to this process. If we have a three page frame allocation, this is all irrelevant, page size, etcetera. OK, so we're running along the little triangle is going to indicate the use bit gets wiped out. If we're doing FIFO, the use bit doesn't actually matter. So we're doing first in first out for page. Or a frame allocation of 3 pages. We do a reference for page A. We're going to need to put it in a frame. So if you think about a program when it is first run, there's going to be a lot of page faults because nothing's in memory.

Yeah.

You just said run this program on the disk. This reminds me I need to come back and talk about fork, so I'll do this one and I'll come back and talk about fork. So first thing we do is we have to populate the frames. We've got three empty frames. So we say I need page A I have empty frames, I'm going to put it in an empty frame. Presumably I put it in frame 0 because why would I put it somewhere? OK, I need page B. Well, I still have an empty frame, so I just put it in B in frame 1. I need page C I put it in frame 2. I need page D. OK, so now FIFO's coming into play. Which one do we jettison in order to make room for page D to be put in some frame.

Yeah.

Right.

Frame zero. That was the first one we loaded. That's the first one we overwrite. OK, so we lost page A, but now we have page D. Bad news for us. We need page A. OK. What are we going to overwrite now? Frame one, it's FIFO. It's pretty straightforward every time. We need a new page. We just overwrite whatever the oldest one was that we have now. We get to this point. We need A. Is this a page fault? No, we already have it. Hooray, so we say Hallelujah. I need A, I got A, fantastic. I need B. I got B. Fantastic. Oh, I need C. OK, so now which one gets jettisoned in order to make room for page C? Frame one, because we're still just walking across our table. And again D. B. We have B. We do not have E, so we jettison E. We have C. We need F. We need B. We have B. We need B. We have B. We need B. We have B. We need C. We're out of luck. We need D again, we need E. We need A. Total number of page faults. Is what? 3 columns with five faults. Well. There's five in each column. Don't forget I think this is a common mistake. Just because the page is still there doesn't mean you didn't have to page fault it in. OK, so we're not counting the overwrites. We're counting the number of faults or the number of pages that have been loaded into memory. So we're counting the total number of letters we put on the page, regardless whether they're stroked out or not.

Yes.

OK, I'm going to come back to this next day because I just want to do a quick overview here. Which I will also do more of next day, but I wanted to give you a heads up on the world of fork. So we talked a little bit about fork. We talked about how fork is like Calvin's cardboard box. Where he has the duplicator so you have a process. It goes into the cardboard box. Multiple processes come out fork one child based on the one call. OK. It's an exact copy of the running parent. Maybe useful, but it would be more interesting if we could run other programs rather than just lots of copies of 1 program. We have a tool to do that. We're going to find out. It's called exec. But you all know that when you exit a program, there's a number you put in the exit commands argument list some integer number, so actually technically a byte. This is the exit status. If you give it a zero, what does that mean? Has anyone talked to you about exit status meanings? So 0 in the world of the Unix command line means everything's great. It's exactly the opposite of the logical true in a C program. So at the command line exit zero. You can think of it as how many things went wrong. Oh. If nothing went wrong, you give a zero. If something went wrong, you can give another number and whatever number it is might mean something we need to collect that number somehow because that's in a totally different process. Recall exit from a program somebody else needs to get that number, or they can't possibly make a decision on it. So we're going to use a command called wait to make that happen. The third part of our toolbox is called exec. Executive Calvin's other cardboard box. It is the transmogrifier is what Waterson calls it. Somebody goes into the box, somebody else comes out-of-the-box. In Calvin's case, it was usually Calvin as a dinosaur. OK, so exec is a magic wand where we're going to turn ourselves into a totally different program so we can make new programs by cloning and then we can turn the clones into other programs that we want to do other interesting things. OK. So fork, as we said last time, returned twice is how people normally talk about this. I hate this terminology, but everybody uses it. You enter fork. As part of being in Fork, a second copy is created and so the return from fork happens in both the parent and in the child, so this is typically how the code works. So we have a process ID. We populate the process ID with the return value of fork. If the process ID is a negative number, then like all system calls in C, that means the wheels fell off and something terrible happened, so it wasn't able to make a child process. So if you get a negative number back, it failed. OK. So there's two other possibilities. We either get 0 back or we get a positive number back. 0 means I am the child. So if you look at the return value from fork and the value is 0, it's telling you you're the clone. There's no ambiguity. This isn't a cheesy science fiction show, you know, whether you're the clone. So you're the clone. You can decide you're going to do the clone work. The parent knows they're not the clone because they got a process ID that was positive. The process ID is in fact the index into the process table to tell us all of the information about that running process image. So we can use that to look up all sorts of interesting things about this running program that we just created. OK, so we know which what our child's identity is because it is a positive number. OK, wait, is to collect that important byte, or did the child succeed? So if you're running a utility, let's say you call the copy command you want to copy some stuff and you want to know whether copy in fact did the work, or maybe it ran out of space or had a permission problem. So you're going to look at the return status from copy and say, did you or did you not work? This is what weight does. So weight you give it a pointer to an integer and it returns to you a process ID. The thing you get back is not only the exit status, but a bunch of other information as well. The exit status is actually only one byte. So if it's an integer, you've got lots of extra space for other interesting piece of information. One of the things you can ask is you can take the status that this is going to populate and say did the program in fact exit or did it crash? Because there's no exit status if we're looking at segfault. So we can say if it exited well then I want to get the exit status which pulls out the appropriate 8 bits from the integer number. If exited is false, there is no exit status because the program crashed. So far so good. Yeah.

The top quarter. You have to do other work. Yeah. Right. So Monday running Webster. Right. That. Support.

Yeah, long. I need a pile of friends so. I'm just making clones myself. OK. So then you make it child. If the child exits, the parents that made the child can call wait. It just works like this, so you give it an integer so address of an integer it will populate that if we exit. If it is true you can ask what the exit status is. If you do not call waits on your child, it enters this peculiar mode called Zombie mode. So it's trying to exit, but it has this super important eight bits that it wants to hand off to somebody. It doesn't need its memory anymore, but it can't leave the process table until it hands that off. So you'll have a process table entry with status Z because your process is in zombie mode trying to exit once the parent calls exit, it's cleaned up and it's gone and you get that that one byte back. If you want to know more exciting things, you can ask questions. I'll come back to this next day, but I wanted to show you the last part of this, which is the transmogrifier. So if you have yourself or your child and you want to run a new program. Right, you want to run code that is not the code of the parent. You can turn yourself into that program, so you say exec something. I'll show you in a second and then you become that program if that fails. It will return a negative number, but this is the one and only time you never need to look at the number back from Exec because exec will either have turned you into somebody else successfully, in which case you're no longer running this program. Or if things went badly, well, you're going to get to the next line because you weren't able to turn yourself into somebody else. OK, the weirdest thing about exec. They've got two different families of exec functions. One of them you give it all of the program name and arguments in a list. The second one, you make a vector or an array and you put them all in there. So if it's exec L, we simply say here's the full path to the program. Here's Arg V1, Arg V2, etcetera. And then you put a null character at the end so that you know that you've come to the end of the list. Exec LP very similar, but it's going to search for this name on the path. This you have to give it the full qualified name. For security reasons, sometimes you want to do that. I'll skip this one for now, but come back next day. So here is an example of running LS as LS minus L my prog dot C. So then the other one is that you have a vector where you're giving it literally the argv argument. R Star RV Open close is exactly what you're passing in here. Again, we have with and without the path. And so you make a list, you put the various things in the list, and then you say please run me my program, but I wanted to do this today so that when you're looking at assignment two, you're not confused what the heck these weird things are, so I'll leave it there. We'll come back next day and we can answer all the questions, but now at least I pulled back the veil a little bit so that you have some clue what you're looking at but I have set all that up for it, so an assignment two you don't actually need to write the fork weight exec code yourself, but I do want you to not be confused when you see it. OK, we'll see you.