Transcription - 2025-02-06

OK. There have been a number of questions about this kind of example so I thought I would just walk this example again to make sure that we all get all the little bits that we want to see here. So remember, the page offset is the number of bits making up the page size. So a 512 byte page size in example one gives us 9 bits of offset. If we're told we have an 11 bit page offset, that means we have 2 to the 11 bits representing the number for what we have as the offset which is 0x800 or 2048 bytes, 2 kilobyte page size. OK, one just leads to the other. So we've got 2 to the 11. 2 to the 11 is 2048 or HEX 800. Hex 800 could be represented as 2 to the 11 also 11 bit offset.

OK, so if we want to do a virtual memory lookup, either read or write, then in the virtual memory space of the program, such as we've all written, we're going to say things like I would like to write to a variable. It is at a particular location in my memory. The virtual memory location of my variable is something or other. So let's say the virtual memory location of another variable J making up my loop counter ends up being at virtual address 77FF. In order for the virtual memory system on the processor to be able to get the real physical memory address that we've stored J in, we need to do this conversion from virtual space to physical space. The result is that we're going to, in order to be able to make the instruction work, we're going to have to be able to turn our virtual number into some actual physical location in main memory.

So to do that, we figure out which page within the program this virtual address lands on. That simply means we divide by how big a page is for our architecture. The page size or page offset is fixed by the manufacturer of our CPU. We have no control over it. And it is not only the same for every page in our program, but it is the same for every page on every program that currently or ever will run on that CPU because it's a property of the hardware of the CPU.

When the program is loaded into memory, it's loaded into segments of page size, which are stored in these frames. So if we have an address 77FF, then to figure out what page address 77FF is we simply divide by the page size and so last time I went through this process of making this happen in binary and let's do that again because I do think that it makes this a lot clearer. SO 77FF. What is F in binary? 1111 you have another F it is obviously the same what is 7? 0111 then we have another one of those.

OK, if we have an 11 bit offset, then that means we're essentially doing a shift of 11 places. We're dividing by the binary number that is a 1 followed by 11 zeros. In exactly the same way, if you divided a decimal number by a number that was a 1 followed by a certain number of zeros, you're just moving the binary or moving the decimal place based on how many zeros you have. So if you were to divide something by 100, you're going to move it 2 positions. Because in decimal you've got a 1 followed by two zeros. We're operating in binary. But that actually doesn't matter. Because we still have a 1 followed by a number of zeros. In this case, we have 11 zeros, so we simply want to move 11 positions. Well, there's four of them. There's another 4. We end up here. This is if we were to consider this a floating point number.

A perfectly valid binary floating point number to represent what we would get if we divided the original binary number by two to the 11, we would then call this the binary point because it's a binary number instead of the decimal point, which is for a decimal number. But that's not really too exciting.

OK. But if we have this division, the only thing that makes this a little bit confused using to divide up into numbers again is that where we landed didn't line up with the division or the boundary, let's say between the hex digits. It landed in the middle of the hex number. So this 7 has been chopped apart. Part of it is on the right hand side of this line. Part of it is on the left hand side of this line.

OK, so that's why when we calculate the page number, we're simply taking the binary digits that are over here. And dividing them into groups of four. We always work from the right. Because if I give you a binary number and ask you what it was in HEX or in decimal, you're going to work from the right hand side because you can always add as many zeros in front of the number if you want, but you can't add zeros on the right hand side of the number without changing the value of the number. So this 1st digit on the left hand side of our line is a 1110. What is that in Hex?

Yeah, E.

OK, we've only got one bit over here, so to make this a hex number, technically we need to add 3 more zeros.

But I think everyone can tell what the number was going to be, whether I wrote the 1 there or the four zeros there. This is a 0. So our page is gonna end up on 0E. We take 0E and we locate it in our page table. This is direct address lookup, so or direct index lookup rather we take our E this is just an array, so we move up to row E in row E we can read the data structure at that row, which has of course the valid bit, the read only bit, the dirty bit, used bit, and our page frame number.

Our valid bit is set. If the valid bit is set, what does that mean? The page is in memory, the frame has been allocated to this page and the data for this page is already in memory. Fantastic. We can continue with our operation to get the data from memory so that our instruction can actually be executed. So this is valid. We can then continue over and get the value for the page frame or the frame number.

OK, this is a read. So the way these other bits would work if this, let's say this was a write, changing the example slightly. So instead of a read of virtual address 77FF, but this was a write to virtual address 77FF. We would also examine the read only bit. Because if the read only bit was set, then it wouldn't be. We wouldn't be allowed to write. In that case, we would have to indicate that there was a problem we would have. And we would have to call for the interrupt to occur. Our page management software would then say you're making an invalid write to read only memory, and I'm sorry program, you need to be removed from the CPU and you're no longer running.

Life, however, is good for us because the read only bit is not set. So we would check it. It's a zero. That's all good. We talked about the dirty and the used bit. So every time you do this the use bit gets set, whether it's a read or whether it's a write. The dirty bit gets set if you're doing a write.

So then you know that the page has been marked dirty because we've updated the memory. Which means for this example is a good one. That means if the dirty bit is set, that the version of this page,

page E that is in frame 11 now differs from the version of this page that is still resident on disk in page location 6. If we were trying to figure out who was a good page to jettison from memory, this is an important piece of information because that means if we want to jettison what's in Page frame 11, we know now that we need to save it. But if we need to find a frame and we're going to get rid of the page and this was marked clear or zero for the dirty bit, then we know that I've already got a copy of this page, so I can just wipe the copy in memory cause I can always bring it back when I need it from disk.

OK, so if we're doing the write, we would have looked at read only, we would set the dirty bit because we're doing the write. If we're doing a write or a read, we're going to set the use bit because we've touched this page. We've used this page.

OK. But to complete our operation here, we're going to get value 11. We're going to treat that as the page frame number so to treat that as the page frame number. We map our 0E into a 11. In binary, we now have. I didn't ask you what these numbers were. I'm sure you could have told me what the binary for hexadecimal 1 was. We bring these digits straight down. And then we repackage them as a hexadecimal number. Again, we work from the right because we have to convert from bases working from the right this is still an F. This is still an F.

Now we get to the point where the line separating the two halves of our register again gets crossed by us, making a collection of four bits. So now we're interpreting these 4 bits as the next hexadecimal digit in our address and it is of course hexadecimal or binary 1111. I wasn't expecting this to be a hard question.

Anyone already on the page several times? Yeah. OK, it's the same as these ones and these ones.

Yes, OK. So then we have this one. Which, somewhat annoyingly also is offset by 1 into the next quadruplet so we've got these three zeros in this one. So what is a 1 followed by three zeros as written as a hex number?

8.

OK, so our actual physical address is 8FF. So when we go over here to memory, we're going to land in memory location 8FF. And I realized I didn't explain the way I'd set these tables up enough on the page because I've confused a number of people. So these are also just arrays. But because my slide is wide and short and not tall and thin, I didn't write them all out as vertical tables. So all this is trying to show you is I've made two cells per row, so I've got zero and cell 1, 0 and 2, 0 and 3, 0 and 4, 0 and 5. And so if we go up to location 11, that's just the one beside location 10 and you can see I've said that frame has been allocated to process 1 and it contains page E, which is exactly what it says over here that page E is in frame 11, so these are just meant to be a different representation of the same information.

That's here cause the CPU is only going to be able to see the page table. But main memory contains all of the data. And of course, in different locations in main memory we have different frames. The important thing I want to make sure is clear here that I may have gone through a little quickly is that this is of course the main memory for the computer containing all of the processes, not just this process. So while this is the page table for process one, I think I mentioned on the previous page elsewhere in

memory is page information for process zero and process 5 and process A. They're happy being in memory, sharing the memory with us. Because they have their own separate frames, so their page tables will presumably have all of the locations making up their frames. But unless we have a shared memory page with them, all of our entries will point to distinct frames. They'll just be completely different frame numbers.

So our frames can live in memory somewhere, in any order because the order and the location is stored in our table. This allows us to say I don't care which frame you give me. Just tell me which one it is and I'll know to look for that page on that frame so it can be completely scrambled relative to the ordering of the data in the actual virtual memory we walk from virtual memory address 0 up to our highest virtual memory address, we're effectively walking up this array. Where every time we get to a new page we go one more step.

So because our page size here is 2 kilobytes, locations 0 through 2047 are all on the first page, so they're in entry zero of our page table. Memory location 2048 through 4095 are in the second entry of our page table and we just continue to walk up in virtual memory from null. Basically from memory location 0 upwards and in memory until we get to whatever the highest addressable memory location is for our process. Every time you get to a byte location that skips on to the next page, we simply walk to a new row in this table.

But if we were thinking about if we were to walk along through page E so we start here in frame 11. So we're here in Page E well, we have 2048 bytes in this frame, making up the data for page E. But if we walk from the last byte of this frame to the next bytes, well, now we're in frame 12. Which apparently isn't allocated to anybody. So if you walk in the virtual memory, so if you've got a loop in your program and you're walking along through a string and you walk off the end of this frame, what's going to happen? Are you going to end up here? Are you going to end up somewhere else?

Yeah. On the next page frame allocated to your process.

OK, you're going to end up on the next page frame allocated to your process. And the next page frame is identified because it's associated with the next page. So what we've effectively done is walked through all the bytes of page E because we came to the last byte on page E, so we went from the first byte on page E to the 2048 byte on page E. When we try to go to the 2049th byte from there, we don't walk across the physical memory. We walk across the virtual memory because remember every memory access goes through this process, so we'd simply be ticking along here.

So 77FF. 7800, 7801, 7802, etcetera, etcetera, etcetera. Oh, in fact, we are near the end of the 77FF we are. We are very close. That is, in fact, the last byte on page 11. So if we add 1 this might have been why I created this example many years ago. If we add 1 to this. 7800. Let's do that. 7800, excuse my eight there. OK, binary for that one binary for 8 is 1000, binary for seven 0111.

Cutting at the same point. We are now on page. This was difficult last time. It's still F. 0F if we'd like. Page 0F is the next page. So we didn't walk physically off here. We walked virtually to this. So, so now we're saying, well, I want to read a byte from this page, OK. Is it valid? No. OK, now I need to do all that work that we talked about last time. I said OK, it's not valid. Is there in fact data somewhere? Yes,

it's on disk. So I'm going to put the data from location C and I should point out this is simply a 4 column version. So here's 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 or A, B, C.

Page C is flagged as process one, page F we're going to put it somewhere. Where do we want to put it? Someone give me a number. What frame do we want to use for this? 16. We use that in the other example. Give me a different one.

Wait, why do you pick 16?

First number that popped into your hat.

OK. It's also a good choice because it's empty. We don't need to do any work to make it empty. So other good choices are the other frames that are empty. Let's skip 12 as well because that would be the one that we would have walked into.

So let's say we want instead of 0E. We want 0F. That's an odd symmetry. We looked up page 0F we're going to write an F in here because we're going to put it in frame F. That's fine. We've just decided that the page frame happens to be with the same name as our page number. You've never had two friends with the same name? I hope somewhat familiar with this scenario. This certainly makes the math easy. We should have done 16. Because we're just going to reverse, OK, take it back. We'll do 16 for you because otherwise all I'm doing is writing this number down again.

OK 16, 0016 is 0001 0110. We should let you do these things, but I'm just rushing ahead here. So that we can write down our virtual memory address or sorry, our physical memory address, this is still a zero. This is still a 0 packaging. These four together we get of course another 0. Packaging these four together, what do we get? B.

So physical address B000. Which ends up in this frame because if we were to divide that by 2048, we would get 16 because that's how we got it in the first place. So we would be at the first location in this frame and we would walk all the way along that frame and then if we walk out of that frame, we have a big problem because we don't even have a page table above that point. So then we would presumably get an illegal address interrupt because you've accessed something that not only isn't part of your program, but isn't physically installed on your computer at all. Possible to do, but how many of you have gotten a sig ill when running a program? Right up there with bus error. One of those really esoteric my program did something violently bad and was torn from a CPU in disgrace.

OK. Does everybody have a clearer idea how this all works now? So what we're doing and I should make say that all I'm doing on this page is absolutely symmetric with the mathematical operation of simply saying I divide by this number because by dividing by the number I simply wrote down the division by saying we move the binary point. Because we're dividing by 1 to the 11 or two to the 11 rather. OK, so same way dividing by 100 as you move two positions in decimal, dividing by two to the 11 is you move 11 positions in binary. A lot of moving, but there's only one binary digit at a time, so we tend to move a long way in binary.

Everybody good with us? OK, so then I hope this is not news to anyone in the room. So we are having the midterm on Monday. Please wait for us to set up. We'll have it all nicely set up for you in the room. We will have exams on the table. Please sit where there's an exam. Yeah.

Sir, Tuesday, Tuesday during class time. That is when we're doing it, yes.

Sorry, what's the difference between like the dirty bit and used bit again? OK, so the dirty bit gets updated if you're doing a write to the page. It tells you whether the page has been changed.

Dirty, finally decided to call this dirty I don't know, but the content of the page no longer matches the clean version that you have stored elsewhere. So it's telling you that there's a pending update you need to have on disk. It is now the only copy of the right data that you have. It has been made dirty, I guess relative to the clean copy, whereas the use bit simply gets set every time you look at the page. And so we'll come back to how we use these bits in a couple of minutes. But it was built into the architecture that every time you talk to a page you use that.

It is therefore possible, as we talked about the last day, that you could have a page replacement policy that looks at the use bit. If there's some way for them to ever to get unset again. But that would have to be a software hardware.

If you tell me about, hey I'm remarking you.

Yeah. Would you? How did I decide where the vertical line is? OK. So the vertical line when 11 positions from the right of the number, when the number is written in binary. And I knew it was 11 positions because our page size is 2 to the 11. So if we go back to example one, the page size is 2 to the 9, so I would have drawn the line here 9 bits in. Note that if it was let's say 2 to the 8 or two to the 12 or two to some other nicely divisible by 4 number, then it gets much easier cause the hex digits all line up perfectly. Sadly, they almost never choose page sizes that work out like that. They all tend to be between 9 and 11.

The way higher. Can you just like use placeholders because like you're? Sorry if it was 2 to the 14 you said.

Ohh OK, so yeah, let's do that example. OK so here if I go back to the one we just had. So if we have what you're talking about arises if we have, let's say, a very low memory address. So let's say we want to access memory address I don't know hex 0035. OK, 5 is.

Anyone.

101. A 3 is. I can see people's lips moving. The zero is I hope I don't need to ask. OK. If we shift over 11. Then 8, 9, 10, 11. We're over here. But and I think this is the question you were asking, if instead of telling you I wanted to go to memory location 0035, which I just said I wanted to go to memory location 35, you can put in as many zeros as you like in order to make the number wide enough that you can shift over there.

And so I think this one's pretty clear. This is a zero this is still 35, if we went to page 0, that one's again on disk. So we would have to do a remapping. So let's say we put this one in 12 or 12 rather. I try to

avoid mixing people up by using decimal names for hexadecimal numbers, even though it looks like a. It looks like a 12, but it's reality a hex 12.

So if we do a hex 12. So this is obviously still the five and the three packaging this up, we still have the zero hardest question on the page. What do we get with the 1001?

9.

OK, physical memory address 9035 is on page 0. And it is in frame 12 because that's where we just decided it should be. Everybody's feeling more comfortable? Somewhat, yeah.

OK. Yeah, he didn't use building. Yeah.

Yeah, or at least before and again have the problem maybe, but yeah.

So the use bit.

So it's probably better when you called the access bit, it tells you whether you have accessed that page. We buried it could be, they give you update, OK, but if they avoid updating when they have decided on you for you because you created another, you probably just gonna be terrible for you. 0212. Because they decided we would put it in the page frame 12 and.

The reason I get page 0 back is because I looked up page 0 and found that it was not valid and then just short-handed. The "oh we're going to have to load page 0 somewhere, let's choose to put it in the remaining empty page," we didn't talk about which was frame 12. So yeah, I kind of highlighted that rather quickly. I'm glad you asked that question.

I can't hear.

Thank you.

If the page. So when I draw a diagram on the page to show you the frame, when I have drawn it on this side I have left them blank or empty if there's nothing logically. But I want to make sure you avoid the name the word blank. The representation got muddled. But if you have a page frame that is unused, then that is your prime choice or choosing for a place to put it in.

Because there's no. Don't have anything of value there. So you could say that is a fantastic place to put my stuff because there's nothing about me there. I can just walk in and plop down whatever else you wanna have in that frame. Like an empty shelf. Only because it's an empty memory shelf. It of course always contains values.

OK, so if you if you add more RAM physically to your computer. Does that immediately and irrevocably translate into more page frames? I will say that the short answer is yes.

The longer answer is that there are more uses for memory than just page frames, so I've mentioned this, we'll come back to this I mentioned this happening earlier, but essentially from the kernel view we have important memory making up kernel data. And then the big structure called memory box. And you and memory gets allocated for all of its other memory needs. A whole pile of them get allocated as page frames. And the rest of them are for I/O buffers.

And so come back when we talk about I/O and talk to go back to me because it just says.

We're talking now. We'll talk more in-depth about tuning how much memory you give to individual processes. It's also possible for the operating system to say, well, I have this much memory trial and this much memory do I need to have page frames. But all of a sudden everything running I/O and some programs see if you need it so I have to be balanced by making some of my kernel page frame memory. And then one more. Any frames neglectfully fly back and forth into that one.

But fundamentally, you're deciding to be exactly correct. You have more addressable physical RAM in your program in your in your computer that provides more memory, there will be more page frames made of that point of view once it's used up, whatever it needs. The only other main things are buffers, kernel data and page tables.

Thank you.

OK, everybody good on that. Alright. Well then let's move back to talking about all that good stuff. OK. So we talked about the page fault handling. So as was just mentioned, if you have a page frame in the allocation for this process that is, as you said, blank I say here empty absolutely the first choice. The only thing on this page before that is if something went bad and you have to terminate the process, then you don't need to look for an allocation.

Everything else is about the work you need to do in order to get an allocated frame to become an empty frame and then what you do with that once you have that. OK, so the timeline. While you're waiting for all this expensive reshuffling of the frames to happen, that process is blocked. That is expensive. Therefore, we care a lot about what strategy we're going to use to manage our page fault strategy because if we can allocate pages more sensibly and choose better which one to throw away we can reduce the number of overall faults.

So we talked about working set what you need, resident set what you have. Talked about how that changes over time. Ideally, we want the resident set to be the working set. That is only possible if we have enough memory. So as was mentioned, you can go to the store and buy more memory and then operating system writers have an easy job because they say just yeah, roll up with your cool bucket of cash and you don't have to do any of this stuff because you just got lots of memory. We tend not to live in that world, so to evaluate our page replacement policy we have to look at real strings and say, how do they turn out?

So we are going to talk about these 4 strategies. So Belady's is optimal. We need to look into the future. FIFO, we did one example last time. It's the simplest. We just replaced whatever page has been hanging around the longest period of time. Then we have least frequently used, replaced the page that we almost never need. So that's an interesting strategy. Least recently used, replaced the one that we needed the longest time ago.

OK, here's the use bit coming back. So that use bit or access bit that's set every time we touch the page, regardless whether we're reading it or writing it. If we have a software process to clear it, then at some point in time, some of our pages may be marked used and other ones may not be. If there's no software

process to clear the use bits, they'll just be set all the time, because by getting the pages into memory in the 1st place for your process, you need to touch the pages to put the memory there. So they will only become clear if a software process runs to clear them.

If you have such a software process, then at the time that you need to decide who to eject, you could look to see which one hasn't been touched since the last time you cleared the use bit and pick one of those. Dirty bit, set whenever a page is modified. Ohh when someone asked me a question once. What if you modify it such that it has the same value? So you update a variable that has a zero in it to still have a zero. It doesn't care. You did a write. The fact that you put the same bits back on the page as was there before, kernel doesn't have time to care about that kind of stuff. You did a write operation, therefore it is dirty. You've committed the dirty deed by doing that update operation.

OK. Here is. Ohh. OK, I have the FIFO example here. So I accidentally just before class, this is how these things go just before class I deleted from my computer the version of these slides that have the reveals let me actually just see if I can bring that back. Oh, I do.

Yes. Excellent. OK.

But there's no fun if you're just presented directly with the answer. So we filled in FIFO which works in this very predictable forward-looking way, because of course it's first in first out, so you just operate across the table whatever you put in first comes out next.

OK, the reason I did the FIFO example first is because it is the simplest. The Belady's remember looks ahead in time to say which page are we not going to need for the longest period of time. So to get the residence set to be the same as the working set may require there to be more memory than is actually available. So what the Belady's gives us is the ability to say what would the optimal strategy be given the actual amount of memory really had, even though our resident set may be too small to accommodate the entire working set.

So this is the same page reference string. We're comparing the results with the previous result because we're saying I'm just changing the algorithm, but I have the same access pattern so I can say for the same pattern of accesses to the same pages, if I swap out the algorithm, am I going to be able to do better than 15 page faults? Still have the same number of frames because obviously changing that would change a pretty dramatic part of our constraints.

So starting with page A. We're going to load it in exactly the same way as before because our page replacement algorithm doesn't have anything to do with what happens when there's no replacement happening when we're loading the first frames, we're not doing page replacement, we're just doing page load. So we're gonna put them into memory in the same way. So the first 3 pages, if you have 3 frames, you're going to end up in the three frames. OK, so that takes us to the C.

Page D. We now need page D we do not have page D. Where do we look to eject somebody for page D to replace? OK, so the Belady's says look forward in time to see what is the best. We can only eject page A, B or C. So there's no point considering injecting, say, page E, because we don't have page E, so the only things to consider are the things that are actually in memory. Well, when do we, when do we need

page A? Turns out we're gonna need page A pretty soon. Page A is in fact the next thing we're going to need. So page A would be the worst choice.

Looking at any other page such as B, it turns out that we need B after we need page A, so page B is a preferable page to eject than page A. And we have another one to consider, Page C, well, we don't need C till we're over here, therefore C is the one that we should eject. But knowing that we can only figure that out if we can in fact look into the future to say which page is the one that is going to be accessed the latest, and we know that for a particular test reference string, because we simply wrote down what happened when running a real program to figure out what pages it needed.

So we just run a program, we record all of the virtual memory addresses that are being accessed. We record the page numbers of those addresses and we just write out a string like this for some real problem. OK, so page C is the one to go. Page D replaces page C. OK, then there's our access to A we've been satisfied moving on to B, still satisfied, moving on to E. Can you repeat? Which page do we eject now?

Yeah.

Yeah. D we eject D because of A, B and D, A and B and D, D is the furthest away. OK, so that page frame 2 getting a lot of activity. The rest of them are still hanging on to their first page. But it turns out that's working well because we need A again. And then we need B and now we need C. OK, what is the best thing to get rid of now at this location when we need C? Well, the next part of the string doesn't actually have the same kind of stuff. Something different happens, yeah. A. So A we never need again. Bye bye. We don't need you at all. So we therefore put our C in there. Sorry we're about C.

Now we need a D. Who do we get rid of to make room for D? OK, we're going to eject the page we just got because we're going to need both E and B before we need C again. So we get our D in, complete this access, use E use B. Now we need C who do we jettison now?

Yeah.

B. No, because we're going to need B right here. So our D, B and E we're going to need B. We're going to need D after that. And then E is the last one to come, so therefore E is the one that gets taken away.

Now we need F. Never seen it before, the newcomer to the game. Who needs to go to make room for F? What? So D goes, F comes in. Our old friend B we're visiting again. Visit B three times. Come to the C. Still have that. Now we need D. OK, at this point who goes?

Yeah.

We could go B. But technically it could also be C because we don't need either one of them again. Is that going to matter in terms of the number of page faults remaining? We can see that if we were to throw away F that would be a bad idea, because we're going to need F again. But if we throw away B then we don't need B again. We throw away C we don't need C again. Is there a difference? No. Because either one is simply useless memory, so whichever one we pick. And B is perfectly fine then. We can satisfy the need for D.

Now we need E. Well, the only one we're going to need again is F, so we can jettison anybody but F? In this case I took C and then we're at the end of the sequence. How many page faults were incurred here?

Yeah.

11.

How many were on the last page, do you remember? I think everyone would agree that 11 is a much better number than 15. If you're gonna pay, you had your choice of paying 11 or $15 for something, I'm sure that you would, you would choose the lower number. OK, let's see what happens if we bump up the page frame. Now I've got 4 frames. Same strength. The Belady's algorithm. OK, we said the first three were no brainers. What about the next one? It's still a load. We have an extra frame now, so the first four now we just put in memory. OK. ABC D. Now we need A again, great. Now we need B. Now we need E. So who do we jettison?

Yeah.

C goes, E comes in. Need A again, B again. Need C again need D. OK. Who are we going to jettison in order to make room for this D?

Yeah.

A goes, D comes in. We have B. We have C we have E. Now we need F. OK, who disappears for F? We're going to need B again. We're going to need C again. We're going to need D again, and we're going to need E again. But one of them is the furthest away, so it is.

E.

There are access to B, B, C, D. Now we're up at the point where we needed E. We just threw it away. So who do we throw away now? Anybody. Total count. This is bargain pricing. We got a much lower number of page faults by just adding one more frame. So the number of page faults is very sensitive to how much memory we've allocated to the process. If we're going to have a large difference between the working sets and the residence set, it may introduce a huge number of extra page faults, so we want to manage the allocation as much as we can. Let's fill in the other two algorithms though here.

Least frequently used.

I've given it 4 frames.

OK. Load in the first four and I've added a little dot above to do a little tally of how many times we've touched the frame because now we're back in the real world. So we have to manage the data to make the decision about who to get rid of because we can't just cheat and look forward into the future and say with my perfect crystal ball I can decide which page should go away.

OK, so we bring in ABC and D we need A. We need B. Oh, and I put the dot in and the choice from the same ones with LFU. We got rid of C, we got rid of C because at the time that we were at this page, we've touched A twice. We've touched B twice. And we touched the other two once, so it's a tie. Notice they said fall back to FIFO to break any ties, just to give an unambiguous answer here, but in reality we

don't have any information so we're hoping FIFO would be good, but we don't really know. Obviously, FIFO doesn't really have any relationship to the future whatsoever, right?

Because what it's about. OK, so for each page we're gonna have to keep a counter. This is in fact extra overhead in memory if we would need an integer of some size allocated for every page to say I'm going to have to count how many times I've touched this page, it can't just be a bit. Because if it's a bit. You can only say yes or no. And having a counter that can only count up to 1 is a pretty small count. So we would need at least a few bits to say, right, maybe a byte. I want to count up to 200 and 55 accesses to this page before I'm going to say that I can't count anymore, so here I've got 2 touches of A, 2 touches of B, and then I needed to get rid of something in order to make room for the E and so I left them as they were. Because they've been touched a lot relative to a little over here for frames 2 and 3. And then using the fall back to FIFO rule, I broke the tie and put it in frame 2 and replace C.

OK. So then we touch A, touch B. Touch C. We're going to have to get rid of one of either E or D. The falling back to FIFO tells us that since we loaded E after we loaded D, we eject E. A and B have been touched a fair bit. B now, which notice we just threw away. The Belady's. If it was an animate thing instead of an algorithm, would be crying in its pillow because we just threw away the thing we're gonna need right now.

I'm sure you've all done this in some capacity in your life. Right. So we needed E, we didn't know we needed E, we just threw it out. It's gone away in the collection. OK, you need to put somebody else in there. Well, we're not going to replace A, A's been touched a lot. We're not gonna replace B, B's been touched a lot. It's either going to be E or C. E we loaded C first, we get rid of it. And what do you know? We're in the same boat. Now we need E back. We're going to replace C. These counts aren't going up because we keep replacing it. Whereas these are hanging around.

OK. So where am I? I'm here, so I need B. Sure, B's fine. I need C. OK, C. I'm going to have to jettison D again. OK, I'm going to need an F. So I'm going to jettison E again. B I Touch a bunch more. I need C, I have C. That's great. OK, now I need a D. OK, D is going back in. My only remaining frame that has a count of one. So I think you can see what is happening with least frequently used. If something gets touched a lot at the beginning, cause notice A, A last got touched way over here. It was the first page that for the Belady's, we said we never need A again. We're going to keep A forever because it's pretty much impossible at this point for anything else's count to grow high enough to exceed that of certainly B and probably even A. Because we're going to keep stuffing everything into this pool of page frame 3.

So least frequently used is seems a lot more naive than it initially seemed. When they do at least frequently used implementation, they usually do some kind of clock based reset the same way we do if we're just doing simple use bit, we're going to wipe out all the information periodically just to make sure that old information from potentially minutes or hours ago isn't influencing you too badly now. But this got 13 page faults. You remember what it was for the Belady's?

8.

That's almost double. OK, how about LRU? The least recently used.

Is a lot like the Belady's except it operates in the real world, so it doesn't look forward because of course we can't look forward. It's going to look back. So it's hoping that because you touch something the least long time ago, that's a good approximation for whether you're interested again in the short term. So if you haven't talked about it for a long time. Hopefully you're done with it or will need it again for a long time.

Yes.

Yeah. So we need to talk about Belady's for example because it is. OK, so it tells us that we had perfect information. How good is it possible to get? Because otherwise you're going to look at a given access on a given string and come up with a number of page faults and say, yeah, is that bad or what?

Right. The Belady's is the mathematical optimal number we can get for that string, it is mathematically impossible to get a number of page faults smaller than the one the Belady's were giving, but because the Belady's requires looking into the future.

Yeah.

We can't actually implement it because we would need to somehow well know what the future of our future accesses are gonna be. We can try to guess.

Yes.

But of course you can't predict what memory a program is gonna access until it actually makes the request. If your programs have no loops, or if they could send data, it would be much easier. They had no calculations it would be totally determined forever where they all were. No problem. Depending on the higher program before it's been written and as soon as you introduce any kind of, you need to make you in the program where someone's going to say, I don't know, figure out which elements in the array based on the result of some calculation. Then you can't know, not at the level of the operating system, what is going on in the program with enough information to make that prediction?

So LFU (least frequently used), LRU (least recently used) and FIFO are all alternative algorithms we can use with the real world information. FIFO is a, I don't know. If optimal is the Belady's, you can think of FIFO as maybe a friend of mine calls it the pessimal, right? That's about as bad as you're going to get because you're not using any information. I suppose you could make one that was even worse, just like randomly change, but FIFO is going to be pretty close to just random choice because it's making its decisions based not on anything having to do with the actual access, but just having to do with load order.

But I suppose there's a slight implementation, there's a slight implication you could say of if you loaded a page longer ago maybe you're less likely to need it again than one you loaded more recently, but it's much more likely that you can make use of when did I last use it? Not when did I first put it into memory. OK so that's what we're doing here in the least recently used. We're going to do the same searching through the string we did for Belady's. We're just going to go in the opposite direction.

OK first four easy. ABCD A, B, they're both there. We get to the E. Who do we jettison? Yeah. C gone. OK. A is there, B that worked out well. C. OK, we jettisoned C we actually needed C so who are we going to jettison in order to get room for C? Remember looking backward only so we've seen just saw B, just saw A, just saw E. We're therefore going to jettison, you can see it coming, D which we need again in the very next step.

OK, so we're gonna need D, so we need to jettison somebody. So C OK, it needs to be kept. B, we just talked about that, A we talked about that pretty recently. OK, so here's where we really see the difference between the Belady's and LRU. The Belady's not only would have told us, hey, don't get rid of the page you're going to need in the next step, but it would also be able to say we're never gonna need A again. This is the time we can get rid of A, but it's close enough to the time we last used A that LRU says, well, yeah, maybe we'll need A I don't know. We just touched it. So I'll get rid of something. I guess I'm getting rid of E.

Ohh again D. OK we just touched the D we just touched the C we just touched the B. We're at least getting rid of A now. It's doing better than FIFO. A which we never need again, we have now jettisoned, so we will no longer consider it in our computations. OK, B we have C we have F we don't have, OK, so to get room for F. Who is it? Not C, not B. Ohh not E it's D. You said D right? OK, we got our F. We still have our B notice that B, much like LFU, is just hanging out there in its originally allocated frame. But in this case, that's what we want, because we're using B a lot.

So here we need F, we need B, it's there, we need B, it's still there. Even more B we need C, we need a D. OK, here who are we getting rid of? We're getting rid of E because we just touched F prior to our eternally needed B and our B was right before the C, but even though it doesn't actually matter because we're never going to need them again, we don't know that. So we have our D. We're going to then have to make room for our E, which we got rid of F because it was the longest to go. Now we need our F again. So that was a bad choice. But for F we're going to replace E.

So there were some bumps along the way. There were some bad decisions. How many total faults were there though?

12.OK.

The Belady's gave us how many? 8 significantly more expensive than Belady's for this string. Is it better than LFU? It was a lot better than LFU. It wasn't a lot better actually. It was only one better. OK. And I didn't actually give you the FIFO example for this one, but I'm sure you could work it all out. I know I put it in there in the suggested exercises.

But in general, we tend to use LRU because LRU is both fairly good and pretty low cost. How would we implement LFU? What do you need is overhead? The good thing about FIFO is there's no overhead. You just simply what did I basically like a clock hand? Where? Where did I load it? As soon as I do that, I just rotate it around. You don't need to know anything about each page, you just need to know one thing about the entire set of frames.

Here we need to know things about pages. We need to keep them in an order. What kind of data structure can we have if we want to keep a clear list of things in order and we want to be able to easily edit it in a particular way, cause what happens when we pull something up in the least recently used, what happens when it goes from used some time ago to the least recently?

Sir. Somebody.

You're basically moving it up to the head of the data. Right, you've got, yeah. It's a linked list. Linked lists work really well for this. You never need to search it. So if you keep moving things up in the linked list, then the far end of the linked list is the thing you want to get rid of. You never need to iterate a list. The list just needs to hang together, so you need one index structure pointer from each frame to the next frame. And then you just need to update whichever one you're currently talking about, back to the head of the list.

OK. This gets used by a number of operating systems over time, so it comes and goes in popularity. The big advantage of clock is that although it's algorithmically a little bit interesting, it just is a use bit algorithm. So you don't need that overhead structure, you don't need the pointers to your linked list. You definitely don't need your integers for your counters for your least frequently used.

OK, so we have a conceptual clock. It already looks like there's points, right? Everybody I think would agree with me that there are arrows on this page, so it looks like they're all pointing at each other. But if you look at the ordering, all it's saying is you go around them in sequential order from zero up to the highest frame and then back to the beginning. So you don't need a data structure to tell you how to count from your lowest number to your highest number. You just count.

OK, so the ordering is actually a product of the layout of memory. So then all we need is to keep track of where we are as we go around the clock. So we're going to go around and as we go, we read and we clear the use bits. When you need a given page you're gonna access the page and set the use bit. Right, that just happens automatically in the hardware, so whatever has happened in the clock, if you don't need to run your page fault handling and you therefore are not running your page replacement algorithm, the clock hand looking at the clock isn't even running.

So in between times the clock runs some use bits will get updated from the last time the clock runs. The strategy of the clock is to simply say if you're invoking the pager replacement policy algorithm, wherever the clock hand is, it simply walks around the clock until it finds clearing, use bits as it goes. Until it finds a page whose use bit is clear. Worst case it goes all the way around the clock. So if you're clearing them as you go, it's going to get set or it's never going to get set while you're going around the clock until you basically come back to where you started, you're going to find the first page you were at. That one is now your choice.

However, as you go around the clock, if you come to a page that you cleared the last time you were clearing, you were sweeping that part of the clock and no one has talked to that page since. It's still clear. So then you simply say, OK, I walk until I find a clear page, either the one I started at or a page that I haven't touched since the last time the page replacement algorithm was put in place.

So this is better than having your use bit cleared on a fixed time schedule because you don't know whether the fixed time schedule has any relationship to how often you're actually having page faults. Whereas you only clear it as part of the page replacement algorithm, then you know that the frequency of the clearing is related to how often you have to make a decision. And therefore you're not going to be clearing it without a decision having been made since last time. And if you reach the situation where you made a decision, you replaced the page and you come in here and everything has been set again, you're actually really lucky because that means you perfectly reused all the pages, maybe dozens of times before you got to the situation where now you need to replace some.

So the clock lets you simply go around until you find one that is clear, possibly one that you cleared as you went around. There is an enhanced clock algorithm which simply uses the use bit and the dirty bit. So as you're going around the clock, it simply looks for these situations. If the use bit and the dirty bit are clear, number one choice. We don't need to save it. It hasn't been used for a while. If the use bit is clear but the dirty bit is set. We haven't used it for a while and we do need to spend energy to write it to the storage, but in theory, if we haven't used it for a while, we might not need it again. So that way if we save it now, we may be able to clear it from memory, potentially for a long time or forever. And therefore it's a good investment of that time.

If the use bit is set but the dirty bit is clear. Well, then, we keep reading this page. Jettisoning it is probably not that great an idea, because yes, it isn't going to need any cost to get stored, but it might be full of valuable values that we're using. Right. Maybe it's a bunch of constants that we're using in a calculation. If you throw that away, you're immediately you're gonna have to bring it back in. And the worst case? You're using it all the time and it's dirty, so you're updating it and so you not only do you have to pay the cost to save it, but you're probably gonna need to bring it back in so.

You essentially search for these classes and it may require multiple sweeps around the frames in order to figure out which is the one in the earliest occurring non empty class of this form. You basically have a hand for each of the classes and they move until they find their scenario. We're out of time for today, but I'll do a clock example maybe first thing not next day, not Monday, but after the midterm.

So we'll see you on Tuesday midterm in this room. I think you've all been here long enough. You know how midterms work. But I will just say, cause there was some confusion about this when I taught data structures last time. There's no talking. But also bring your own pens. Don't expect to share a calculator or an eraser with a friend. If you want to bring a calculator, a simple calculator, that's fine. If you want to bring a calculator that has graphing and memory, that's not fine, but anything that is like a simple arithmetical operation calculator is fine.

I will also say because there seems to be a fair bit of confusion about this right now, we changed what we're using for multiple choice scanning. So it is no longer important for you to colour in the bubble with a pencil. It doesn't care whether it's a pencil, the old ones. It needed to electrically activate on the pencil lead to detect the coloured in bubble. But please do not use a red pen that shows up very badly on the scan, so Red Pen may mean no answer as far as the scan goes, but pencil versus black or a blue pen equivalent. OK, I will let you go. And we will see you on Tuesday.