CIS3110 Lecture 8 - Revised Transcript

Audio File: CIS3110-Class-Audio-2025-03-06.mp3

All right. The point in time has come. So last day we were wrapping up our discussion about memory. So we talked about various allocation schemes. Is there a whistling going on to you here, or is that just happening up here? It is happening OK. It's just the wind in the door, all right.

So we talked about first fit, best fit and worst fit in our... I think I mentioned last day what people actually do with these depends on the data. So yes. Oh yes, someone actually reminded me already, so I've started that. So if you have data structures that are of a regular size. And so any graph based data structure algorithm tends to have nodes of identical size. Those work really well with best fit. Right. So if you've got a binary tree you've got, I don't know, linked list. You've got any other kind of graph structure where you've got fixed sized pieces that works really well because your fixed size piece is going to be the same size as your other fixed size piece.

Worst case are strings. We like strings. Strings are full of good data and people like to read and understand. But programmatically, strings tend to be awful because they all are of different ragged lengths. Some of them are tiny, some of them are long. You are almost guaranteed to not have a perfect size hole for a new string based on deallocating a string.

So what a lot of memory allocators will do is they will look at the sizes and see if the size is some common size, right. If you've got two or three that are coming in with the same size, it may decide yeah, that's the size of this program likes. I'm going to do best fit for things of that size, and in particular if they're not an even number of words. It'll probably just treat it like a string and then say first fit is about as good as anything else, because as someone pointed out last day, just doing the search is going to take some time. So having an algorithm that spends a lot of time finding the optimal way of saving a few bytes is not a good payoff. If you've got a program that allocates and deallocates millions of strings because you're going to have to search through that list. So therefore first fit may be the best.

One of the things that a virtual memory allocator has the option of doing that your code didn't have the option of doing is increasing the size of the data segments, and so at some point it'll decide OK, I'll just add a few more pages. I'll have another allocation block to play with. We can put things in the new block if the old block is becoming fragmented.

There's a whole discussion about how to manage and reproduce this fragmentation that we won't go into in this course, but I did want to mention the buddy system largely because it is essentially the binary search algorithm, but applied to memory. So what it does is if I have a hole in memory, then if I have something to put in the hole rather than, let's say putting it at the beginning of the hole and carving off the rest as a different sized hole, it'll say, well, I divide the hole in two, and if my remaining two holes are still too big, I divide them into two and I divide them into two until I get a hole that dividing it further would mean that we don't have enough space anymore, which means you automatically have a companion when you're trying to re-merge memory later. It works OK. It can be proven mathematically to be slightly better than a lot of the other common cases. It is algorithmically complex to manage and debug, so no one but Linux has bothered, but Linux has been using it for decades now, so that's how they do the management on Linux as they do this sub-partition. I won't go through it in detail, but if you're keen on it, Silberschatz has a whole section of the chapter devoted to going through the algorithm. (Virtual Memory, 10.4)

OK. The other thing we do with memory is we do a thing called caching. So caching comes from a term used by explorers or travelers in inhospitable climates. So let's say you need to go across, I don't know, a desert. Or the polar ice cap or some other place where you're unlikely to run into a nice little hotel part way across. And so you need to figure out how to get there yourself. You can only carry so much food and water. If the trip is long enough that you carrying your own food and water would result in you running out of food and water before you get to the destination you have a problem. You're not going to survive the trip if you just naively start marching across, let's say this desert.

But what you can do is you can take as much food and water as you can carry and go a little way out into the desert and leave most of it there and then go back home. So that you can then carry as much food and water as you can carry out into the desert, supplement a little bit when you come to your cache station you've now created, go a little bit further, leave some more further out in the desert and you can make these little cache stations proceed further and further out in the desert until eventually you can complete the trip. And going to the first cache station consuming the last of your food and water from that cache station, go to the next cache station. Pick up the stuff there. Consume that and so on and roll up your cache stations as you go across the desert, successfully arriving at the end some large number of days later.

Because you've pushed little pieces out into the desert that's called... you've set up a cache of this important resource, so that later when you need it, you can go to the cache and you can find it faster than having to go all the way home and getting more food from there. So caching in memory is the same idea. It is a place where we can put values that we think we're going to need again such that they're faster to get to than having to go all the way to our original storage.

If we think about our virtual memory discussion way back at the beginning here where I had these nice diagrams, we can think about RAM as effectively a cache for the information on the paging disk. If we had to do, let's say that desperation swapping activity where we took the whole process image memory, shoved it onto the paging disk and didn't run the process for a while.

So then when we restart the process, everything's down here. Nothing's in main memory. Every request would mean we'd go to main memory. The answer would be no. We go to the paging disk where the answer is yes, we bring the answer from the paging disk and the main memory. The next time we come back to that page, it's much faster.

So the general idea of caching is that idea we have something in a slower, more distant but probably cheaper memory. But anything that we access from there, we're going to record in the cache. So the cache physically is on the motherboard in between the actual instruction registers and data registers and RAM.

So if you're talking about, let's say an Intel based CPU system, right, I see a number of people with what are probably Intel computers here because they don't have a little apple on the back. So they actually have what they call an external or L2 cache. So if you request from memory the request goes through the cache, we go to memory, we copy it into cache, we bring it into the CPU. If we request the same thing later, we go to the cache first. If it's in the cache, we don't have to go all the way to memory.

So what we're really doing is we're saying, do you have it on your local crib sheet where we can get it things faster? We're going to bypass and I'll show you that in just a second how the actual memory lookup that we've seen so far works.

So we're going to have what we're going to call a tag, which is going to be a memory address identifier or some other similar way of talking about exactly what we like. We reach for the cache. The hardware of the cache in parallel checks all of the tags, so this isn't a loop where we're iterating over the tags in memory to see if they're there. This is a hardware device where we say do you have an entry for value 0 and the answer is no, there is no zero in any cell. Do you have an entry for value 8? Yes, there's a value here in this cell, but every cell at the hardware level is compared together. So that the value for 8 can be produced very quickly out of this dedicated hardware item.

If it isn't in the cache, we have what we call a cache miss. Then that means we have to go to memory. So you can see that memory... there's a direct link between each of these patterns I put here in what we call a cache line and the underlying portion of the real memory. So a cache line depending on your cache is going to be a different hardware configured size. But a cache line is always a smaller value than a page size. Frequently it's about 128 bytes. You've got 512 bytes on an Intel page, you've got 128 bytes and an Intel L2 cache. If you need anything within those 128 bytes, we can ZAP it into the CPU more quickly from the cache.

But notice the cache is small because this hardware parallel lookup is expensive. We don't have enough memory to back all of RAM, so we've got literally a crib sheet. I got asked this question before if I get asked this question again, I can pull the answer from the crib sheet. Otherwise I have to go to main memory and do the full job of looking it up. So you're going to have potentially gigabytes of main memory, but you're only going to have a few kilobytes of this cache, yeah. Both that we're identifying which for the page is essentially the address generated from the address, but the line is the real data for them. (Virtual Memory, 10.6)

So then we go with the cache, and if the address we're looking for is covered by the cache page or the cache line, then we can find that value here. OK. So we can think of this as a page memory. Maybe you've got, let's say 4 lines cover one page. So your actual memory address is going to fall into a region covered by a hardware page size block of memory, but a particular cache line is only covering a part of that page. You can see 0 is not in the cache. 5 is not in the cache. 8 is not in the cache, but values from memory locations 2 and three actually 1, 2 and 3 are all here in the cache. They're not in the same order because the order in the cache will have to do with the order of the request, not physical memory order on the same page.

So presumably if we ask for something that fell into this region of memory, we copied it into the cache. If we want something from this region of memory, again, we're going to be able to access it from the cache. If let's say we need something from location zero, well, we're going to need to access it from main memory. When we do that, the whole cache line width will get copied into the cache. Something will have to be ejected. It's a hardware managed least recently used for these chips. So it just kicks something out. You're just hoping that you're gonna get some value for this. So it's like a tiny paging memory system implemented entirely in hardware memory. Instead of paging something out to the paging disk, we just rely on it being also duplicated in the main RAM of the system.

So the tags are essentially page table entries and the lines are pages or pieces of pages more commonly. In an actual computer system, this being such an effective idea, we tend to have multiple caches. The closer they are to the CPU the faster and more expensive they are, yeah. A hit simply means you've looked up something and the tag by which you looked it up indicates it is in the cache. A miss simply means it wasn't in the cache. OK, so in your crib sheet example, you've got, let's say, pieces of information identified on the crib sheet. A hit means it was on the sheet. A miss means you now have to go look it up from the underlying storage.

I think I have a question here and then a question over here.

[Student question about cache lookup time]

Very much constant time because fundamentally what happens is you load the tag into the cache address request and the bits of the tag are simultaneously compared with every single tag, the bits of the request are simultaneously compared with every single tag. It is required to be unique, so as soon as it gets to a point where it doesn't match, that part of the cache knows it doesn't have to continue. If you get to the last bit, it gives you back the line.

[Student question about cache proximity]

Yeah, so when I say when I say close, I mean both in terms of the path on the data and address bus and for speed purposes physically close on the motherboard. And in fact, so close in the case of the L1 cache that modern CPUs have the L1 cache built into the chip. So that if you pull the CPU chip out of the socket, you're not just pulling out the CPU, you're pulling out the CPU and a bunch of co-processors that are right there cut from the same die on the same silicon leaf. One of them is your L1 cache.

Those of you who are running new Macs, if you have the M1 or the M2 chip, these are what they call system-on-a-chip design. So same deal except the L2 cache is also inside the same dot. It's just closer to the edge of the die. It's just a slightly slower access you're actually getting. Not only the L1 and the L2, but your RAM is built into this same chip.

I don't know, do we have a course where we talked about chip fabrication anymore or not? If people care, let me know. I'll put up an article for you to read. But essentially, once you're talking about these system on a chip designs, they make decisions based on how well the fabrication process worked for different portions of the system in order to patch together a working computer out of the parts of the die process that didn't get marred during the production. This is one of the reasons that they tend to have different levels of CPU available to you. Same deal with big screen TV's. It's kind of an interesting process.

But what we can think about here is that the CPU decides it needs some data, right? It's running instruction that says add the data from location AB92 to the data in location AB97. Those are of course memory addresses. It's going to then request through this layers of cache. If you get a miss at every layer all the way out to RAM. If at that point the RAM comes back using your virtual memory mapping as not valid page, then we have to go to the paging disk. You load your frame with the page. Then when you repeat, you're going to get it from RAM.

If you have a successful access to RAM to support a particular memory access, then you're gonna leave a copy of it in every one of these caches between there and the CPU. As they say, they get smaller and faster as you get closer to the CPU. So if let's say you're iterating through a large array, these are going to fill up first. You're just going to run out of space because if your array is bigger than your cache space, you're going to have to start ejecting things from the cache to put in more recent values. The L2 cache is bigger. It's cheaper, it's further away. Further away than that is RAM. (Virtual Memory, 10.6, I/O Systems, 12.3)

So direct mapped, we're talking about this index based lookup we're familiar with. Associative is what I am talking about with the parallel hardware search for the key. So an associative memory has this built in parallel search for any key value you come to look up. Kind of think of it as like a hardware high speed key value search. What you wish a hash table was where it could search for each of the values in parallel and tell you the value in one step.

As we move out here, it is larger, slower and cheaper. So once we're at the disk, you can buy a lot of disks per dollar. The same dollar will buy much less RAM, even less L2 cache, and even less as you get closer and closer to the CPU.

[Student question about cache check order]

Only if there's a miss, right? Yes, sorry, I only got halfway through my example. Thank you for pointing that out. So in the example I said if we're looking for memory that isn't in any memory anywhere but is on the paging desk, all of the initial searches are going to fail. We'll get to RAM. It is marked invalid. We're then going to have to schedule a read from the disk. We're going to bring it from the disk, put it in RAM.

OK, if we then immediately ask for the same data again, it's not even going to leave the CPU. It's going to find it here in this cache that is close by the CPU where that cache says, yeah, you just looked for that a second ago, here it is. If that gets exhausted, so let's say we're going through that loop example. Let's say we have enough data in our loop that it's in the L2 cache but only maybe 1/5 of it can be copied into the L1 cache.

So if we start by having to go to the paging disk, we're going to copy the memory from the paging disk into the page frame that we've allocated in RAM. We're going to copy the cache line from RAM into a place in the L2 cache. We're going to copy possibly a smaller cache line into the L1 cache. And eventually the data is also going to make it to the CPU.

This is all pretty fast because as we're loading this, these are part of the hardware memory access operations, so it doesn't take additional scheduled time to update the L2 and L1 cache. That's just going to happen because they were asked for something. There's a miss, so we just write it. We schedule a write as we are bringing the data into the CPU on the data bus.

[Student question about cache line size compared to page size]

So if your cache line was the same size as your page size, then the cache tag and the virtual address would be the same. If the cache line is, let's say 1/4 of the page size, then what that means is that a single page frame has to map to four cache lines, so you're going to need an additional 2 bits stuck on to the end of your page frame to identify which cache line you're talking about, because it could be the zero through 3 cache line in there. That makes sense, OK?

[Student question about direct mapped vs associative cache]

OK, so direct map is an array style address coming with a number. You know how to skip through a number that's six slides to get... it will be the same way in the array... if you want to go to array location 7 you know how big the array... take your tile size... that experiment round about 10, so direct map... this means you're taking the number... you're directly going a stone.

There is the parallel hardware, so you don't have to figure out where to go in there by doing that. You're relying on system identifying the big pattern you supply for the thing you're looking for. Each cell in an in parallel, say either one building that you don't have to do any kind of leak through it.

[Student question about data persisting in cache when page is paged out]

OK, the question is, will it be possible for something to still be cached if you're frequently accessing it but the page has been paged out? This is a very interesting question that I have never thought about. But I think the answer could be yes.

Because if you're regularly accessing a particular piece of information and that information always falls in the same cache line. So let's say there's some integer you just keep looking at, like a counter or something. It is quite possible based on how your page replacement strategy works that a situation arises where you say I need to eject a page from memory. And you choose the page in memory that also holds that counter. And you could eject it.

Now the problem is going to be this is a topic that we don't actually formally discuss in this course. The problem is going to be a situation we call cache coherence. So you're going to need to somehow know that the data in the cache is more recent than the data that's in memory.

So I think most operating systems will probably prevent that from happening. You'll specifically avoid ejecting a page unless the cache has actually been flushed to that page, and as soon as you flush it to the page, if you're using something like least recently used, it's going to update all the counters. But you could in theory have data in the cache that was no longer backed by a subordinate level cache and memory is just a type of subordinate level cache. But you do need a strategy to make sure that you haven't got a value that exists only in the cache that should be reflected in the main memory image and never got updated there. So your rights algorithm or your page replacement algorithm is going to need to keep these in sync.

If we had an operating system 2 course, this is the bread and butter of exactly what it talks about. If you look at the distributed chapters at the end of the textbook, they talk all about this sort of stuff. I saw your hand go up. I've got one here and over here. First, yeah.

[Student question about cache replacement policy]

They mostly follow an LRU implemented in hardware, but you could do a different one. So the caches will be... the cache policy will be determined by its hardware. Right. They're frequently... they're infrequently programmable at a flexible level. So they'll just have a way of operating that the manufacturer built in.

The page replacement policy for main memory you might adapt based on various things going on with the program, so it could be that they're different, but it's quite likely that once your cache is installed on your CPU, it's just going to do whatever the hardware designer decided was a good thing for it to do, which is almost certainly LRU. That's the most common one there. Someone had a question?

[Student question about cache coherence]

So that's actually a flaw in your whole memory management. Because it shouldn't be absent from the paging disk and have any information in the page. People would say go to the page, right, so it should... if the logical situation is that it's not on a page, yes, it should be default, but it shouldn't sample before procession certification. Yes and have it there.

No. Excellent question. So this is on hardware first. So all of the processors have to share the same cache. So one of the interesting things here is what happens as soon as through the context switch. Because if you have a context switch, then the cache is suddenly full of someone else's data. You need to make sure that you're not going to accidentally think that the virtual memory address, I don't know, 200 from some other program is valid to serve a memory address request for 200 here.

So what they'll have to do is flush the cache to make sure that it is clear. Now this brings us to the situation of what happens when you've got more than one CPU. So if you've got a multicore system, or you've got a system with multiple CPUs, there tends to be an L1 cache per CPU. Frequently, the difference between a CPU and a core is what cache they actually have in common. But the outer caches are shared.

So if you swap... if you do a context switch on a given CPU and you switch from one running process to another, you're going to have to clear all the caches associated with that process, and so in that case it's going to have to be the... if you have a shared L2 cache between multiple cores or multiple CPUs, the actual process ID needs to be somewhere in the tag so that you know whose memory you're actually looking for. Otherwise it's just going to be meaningless scrambled eggs. You're going to get slices of someone else's program.

So the actual context switches here are going to take into effect on the CPU, but you've only got one paging disk. You've only got one RAM. We already know that you've got a different virtual memory table to help you figure out whose RAM is whose. But at the cache level, if we've got multiple CPU's, then we're going to want to have inside the same cache data for different programs at the same time. OK, there's a question still over here, yeah.

[Student question about updating caches on reads and writes]

Yes, and you do that implicitly by pressing it by just doing the reader. So if you do a read in the data bus and you're granting back the balance from wherever you have to go to get them, you're updating the cache lines as you bring it toward the CPU. If you do a write, then the write is going to go out to some level of cache and be stored there.

If you need to flush the cache, all pending writes need to be flushed out to at least RAM, or you're going to lose the effect of that, right? So a flush is not particularly cheap. And this is another reason why you don't really want to do lots of extra context switches that you don't need to do. OK. So you're going to need to make sure that the important values the CPU are generating get stored somewhere. OK. Any more questions? And this is a thorny little bit of the course here so I want to make sure it's...

[Student question about cores vs CPUs]

So I probably did mention before we started talking across the front section. A process is simply the data structure in memory of a running program. So we load the text segment up, create the data segment, create the stack, allocated process context that we have a program counter set that away. We go in order to do...

The way to go, we need some hardware... in fact we're going to do the thing right. So we have, we need a physical instruction register or physical program counter register. All these are the things the process context into that and then return.

If you have an ancient machine with only one CPU you want... that's the whole start. If you've got any CPU any team that has either multiple CPUs or multiple cores as you hear the lingo around now, or you buy a machine and it'll tell you something like 8 CPUs, 20 cores and everybody goes, "That's great," but until now you probably haven't thought about what that really means.

So a core is a place where you can run a program. It's got all those physical things, the instruction register, the program counter, all that stuff. The CPU you can think of the simplest CPU is essentially containing one core. The main difference between a core and a CPU is simply how the caches are organized.

So typically in a given CPU, let's say you have one CPU with two cores. They may have their own L1 cache, but they're sharing the L2 cache. So it's just simply at this level, each core gets a certain amount of resources private to it. And then as you work out toward main memory, things become more shared.

If you have a system that only has CPUs, let's say you've got a dual CPU system that you bought in, I don't know 1996, somehow it's going to have its own set of caching provide it to that CPU and one RAM, but if you got a modern system like this one, I forget what it is 24 cores out of eight CPUs or something like this. So that means each CPU has three cores. They'll each have like an L1, but then that CPU probably has one shared L2 and then the whole system has one shared bank of RAM.

So by core we're just simply saying an element that contains all the pieces to run a program. And by a CPU, we're talking about a bag of one or more cores where they will share some resources within the CPU and there will be global system resources outside of that. Does that answer your question? OK. Another question, yeah.

[Student question about cache flushing]

Right. What they say, what you say happens when you flush your... they we put them to release the next level cache quite possibly. All the way around. Pushing into RAM is certainly the easiest way to solve modern cache coherence issues, whatever quite right.

Yes, for that, for most operating systems are going to do because, frankly, on a multi-user system. If you take the parallel course, you'll find some other interesting edges in here, but if you have a specific parallel algorithm that knows it's made of several pieces, then they can take advantage of the fact that they're sharing data in a way that if you just have a Linux kernel running, I know an editor and I know a print server and a web server and something else, and they're all just sort of randomly chosen things, it is unlikely that they'll be able to do anything cooperatively that is important.

[Professor answering potential exam questions]

So good sample questions here. The difference between an L1 and an L2 cache, maybe difference between direct map and associative memory. The staging of events between the CPU's request, let's say to read or to write and where the data gets updated in various scenarios those might be interesting questions.

Ohh yeah. Can I repeat questions, Chris? Oh, so I said. So the ordering or connection of the events. The difference between the various levels. The difference between the different lookup types so direct mapped and associative. I think that's what I... I was making them up as I went, so it's quite possible that list is slightly different than than what I said a second ago. OK. Are we ready for the big finale of memory? (Virtual Memory, 10.5)

OK. So we were talking about that page table. This is how we do the whole mapping from the physical memory to the virtual memory. But what you might not have thought about is that if we were to think about paging back when paging was invented, you had a few 1000 pages covering your whole process because you're probably on a 16 bit processor. At that time, having a 512 bytes long page was a pretty big section of your available RAM.

But now we're on 64 bit machines. If you've got a 512 byte slice of the full 64 bit address space, you've got billions of the damn things. And you don't need billions. So you don't want an array with a billion entries where you're using like 16 of them. 12 at the bottom and four at the top. That's a huge empty array. That is itself a vast data structure to store in memory. And we would use almost all of our huge amount of RAM, just keeping track of the fact that we didn't have anything in it, which sounds like a really bad way to use RAM.

So we need a way of saying I don't want to store the parts of the page table that I don't need. I need somehow to be able to skip those parts. So one of the very common strategies is the hierarchical page table.

[Student question about associative memory]

So associated memory is that further memory parallel search. You've given a key and you're looking out and in parallel each one of the cells determines if they hold that.

OK, So what we're doing here is we're saying if we already have this strategy to take my potentially huge program and cut it up into blocks of a fixed size and call them pages so that I don't have to talk about all the memory in the middle where there's that gap between the data segment and the stack segment, what? Why don't I just use that idea to chop up my enormous page table into segments so that I don't have to talk about the chunk in the middle where I don't have any data? And so that's all they do is...

So therefore we take our virtual address. We still have a hardware defined fixed offset. But what used to be our single entry for our page IDs, we now divide into two pieces so that the same way we took off the bits for the offset into the page frame that we're going to finally look up, we simply take the higher end bits and say I'm going to take M of them and that's going to be used to figure out how far I go into my slice T here from the page table which I found by taking the first... that is an L. Sorry, I meant to update that someone asked me last time. Why does it say one bit? It's actually meant to be the letter L.

So I've got L, M, and N. In the L slice I've got a value that I generate simply by taking the most significant bits out of the virtual address that gives me a value I'm going to call P. In our previous example, that was our page value. We used the page value to find a slot in the page table that took us directly to our frame. Now we're going to do that in two steps, so we simply take P, go to our global now page directory table. So this tells us which chunk of the page table we actually want to look at. We skip P entries down into that, using that direct addressing or direct mapping.

That gives us a value you can see here the value I've stored there is a T so that says I take the value I got out of the virtual address from the middle piece. I take page table chunk T and I now go Q positions into the value T and I find my actual frame pointer or frame pointer frame number, and then I combine that with a physical address just like I did at the beginning of this chapter.

So all I'm doing is I'm doing this recursively. So if I skip back all the way to this kind of example we had, I was taking a virtual address, dividing it into the two pieces where I had the page number and the offset. I took the page number, did direct mapping into this table. This happened to be page 0, so therefore I go to slot 0. It tells me that I need frame 16. I'm just doing that again.

To figure out first which part of the page table I want to look at and then which line within that part of the page table. So I've done exactly the same thing, except I've now done it where I've got one value from the most significant bits to say in essentially a page table for a page table table right - a second order page table, which piece of the real page table do I want to talk about? I find that piece using data also from my virtual address. I figure out where in that piece I want to look using the value I got from my first level lookup that gives me my frame value. I combine my frame value with my offset. That gives me my physical address. Looks super complicated. Is really just what we did before, twice.

Questions on that. Yeah. (Virtual Memory, 10.5)

[Student question about how the page table directory works]

So I think that your painting as being made of the combination of the leftmost umbilical there. Right. So to go from address 0 up to the highest compressible address in memory, you're going to have a set of those. Middle one the green there in a second. So if you were to write down all of the slices for all of the possible values of Q from here, that would give you a long sequence of these page tables. If you wrote them all down together, that would be the same page table as the one we had on the earlier slot.

All this is doing is telling you which of those in the sequence you're talking about, where if you're talking about the first one in the sequence, it would be indicated with the first slot here. If you're talking about the second one, it's the second slot here, etcetera. So let's say this was... you know what's a good number? Let's say it was an 8 bit value for the chunk here of L bits. So if L is 8, that means we have 256 possible values. The first byte of our virtual address would tell us which of 256 entries in the page directory table we're talking about. The remainder of the address here we then take the next piece here. So this would be starting from bit 9 on to whatever the next piece is. That would then tell us how far into the second level page table block we actually want to talk about.

OK, so this would then have those 256 entries. What am I saying, 256? There's one byte is... yeah, one byte is 256 and tell how tired I am. I can't even do base 2 math anymore, right? So if you've got one byte worth of data for the first chunk there are 256 here. If we had 9 bits of data in the first chunk, then we've got 512 that we could have here.

OK, this one you need to cover the whole address space, but if you've got a tiny program where maybe you've got one page of stack and 2 pages of text and data, then all of these would be empty right? Nulls other than the first couple and the last one to say yeah, you need a stack, you need the other two, and then you'd only have your 3 pages of data, so then you'd need... in fact, if you got 3 pages, you'd have an entry here for the low ones, an entry up here for the high ones for the stack. Your entry here would say I have a second order table which will contain a certain number greater than 1. So if we need only three pages, we would have an entry to the first chunk, it would have two entries for two different physical frames. And then our stack would be in the last entry which would need one second order table and then that would talk about the last physical frame, everything in the middle where we say we don't have anything to talk about in here because I don't have any data in the program. There doesn't need a second order table. It doesn't need to be allocated because there's no memory to put in it.

So therefore we can save a lot of memory because what we've done is we've said however many slices I have sliced my page table into, I only need to keep the slices that refer to frames that actually exist. The ones that I haven't used, we just skip over. And we get as was mentioned earlier, our friend, the segmentation violation when you try to go to that data because it doesn't exist. So there's no way if you say read me those values that the memory system can tell you what they are because it it, I mean maybe they're in another dimension, but they're not a reality in this world. So therefore you get a segmentation violation. That makes sense. Yeah.

[Student question about staff]

Your staff were awesome.

[Student question about the offset]

The O here is what we were calling the offset on the first set of examples. Is that what you're asking? So in this example we had two pieces of our virtual memory address, so we have a page offset of 11 bits, so we're going to carve 11 bits off of the least significant end of our value. So we take here our 77 FF and we curve off those to produce A7, FF remainder and then the ones that are most significant in those 11 bits, we turned into a single frame number E.

All I'm doing here is I'm saying, OK, the bits that are more significant than the 11 bits, I'm dividing that into two pieces. Where the least significant of those are going to tell us an offset into our chunk of the page table that we found by taking the most significant bits going into the page directory table and saying where is the entry here to tell us where in RAM I have placed this piece of the page table so that I can use my value Q to say if I've located the piece of the page table I called T which I know because I put the value here in the page directory table. There's no need for them to be in any particular ordering and memory, right? I just write down where I put it the same way I wrote down which frame I wanted to use in our earlier exam.

And then I use what is effectively the offset of this PQ combo in the L to M bits in this virtual address to find the data inside this piece of the page table, which gives me the actual frame number that I'm then going to combine the same way I combined the actual frame number in our previous example with the offset from the virtual address to find the real physical address in memory.

Now these you're most likely going to make one page in size because you already have hardware that deals with things that are one page in size and so then they would fit nicely in a page frame and you don't need to worry about things of totally different sizes. So whatever size you decided to carve up your program into in pages, you're going to carve your page table into the same size pieces so that they can fit in the same size storage. And then your question becomes, how big does your page directory table need to be to talk about a page table carved into those sizes? That covers whatever the address space is for your CPU. Does that answer your question?

OK, so I saw various hands go up and down. I think I have 3. So yeah, in the back of the Gray hoodie first.

[Student question about second page table]

The page table is the final location of the indexing the second page table. So the current page table is defined the piece of this aggregate table. To find the index into the second page table that's directly out of the virtual address. So that's the value of the following Q at the top. From the previous, we didn't use 2 tables because we didn't need... we didn't have to use...

Well, so in the earlier example, just essentially assume that you have all the green table lined up in one big massive place. And therefore we didn't have to consider chopping that in the piece. What we're saying now is that that would be a really massive list of 64 bit architecture. Gigabytes in size and you don't want to have a multi GB data structure to keep track of our program, and that seems like a ridiculous...

Right. So we've simply said I have this trick where I can make a program work, cut it out and get these fixed sized pieces of my hardware to heal. So why don't they do that for my big data structure and I'll just part of the data, the page table up into pieces, the same socks and so essentially it is a 2 step application with the same thing. Think of if we think about the two leftmost peoples there the the red...

The P and the Q in isolation, you're effectively doing a lookup of exactly the types we did in the previous examples to figure out where inside of the key or not. So then we're simply saying based on the value I get there, I just do it again. Pull the same trick to figure out now where in memory and define the actual frame pointer.

That. Yeah. OK, so. He...

So it is the page table twice. We only have one page and we only have one page table for this front, but we have several processes.

Right. OK. That question questions over here. Right.

[Student question about three-level page tables]

Engineer. Right. I don't say I do not know the architecture that actually has three bits for that. Well, you might want to use one because you don't want to be able to reject eight states. I'm going to look back. Right, right. The initial answer is no, but I now can't see why. OK. Any other friend? Yeah, the last question.

[Student question about dirty bits in page tables]

Well basically... but when the dirty bit for the page directory table and round for the page table itself and... probably ends. But I've never been asked this support, so I would have to look at it. And it's the... page the directory table that's its own hardware and memory. It's not random.

No, this is all going back to that 100% based solution for how to stitch together your program. So it will be managed by the MMU. So the MMU is going to do the lookup in dedicated hardware, but the data all has to live in RAM.

And sorry, what last question at any. Page table T represents the three numbers. That's the frame number. Yeah. So I chose that the frame number because that is exactly the same information as the frame number we saw from the earlier examples. We just had to do a 2 step look up to get the frame number, but you could think of fee as essentially the analog of the frame number. If you were looking for a piece of the page table, not a piece of a program.

OK. Lots of good questions on that one. Anymore questions on that? OK. Just before we leave this module then I wanted to ask you a question. As we think about threads versus processes. (Processes, 3.1, 3.4; Threads, 4.1-4.3)

So if you have a process, the cost of generating a new process is incurred by fork. If you generate a new thread, it's less work. You only need a new stack. If we think about the fact that a page table needs to be created to make that happen. The page tables going to need to keep track of potentially a lot of new pages. So threads are frequently referred to as much more lightweight. They don't need the heavy memory backing that a process does because it's sharing most of the process information, including most of the page table information. So you can generate a lot of threads much more efficiently than you can generate a lot of processes.

So why don't we just do everything with threats? What are the downsides in threats? Yeah.

[Student response about thread management challenges]

Manage the... you have to manage... because if it's collision like back in the... we're just like what they're all accessing the same data, it's more to management program, OK, you're you're talking about critical... so yes, we do have more thought...

Yeah, to make sure that each thread is only accessing the data it should be accessing. If you have a shared data segment, everything in the data segment now requires that cost, so you need to make sure that they're not stepping on each other's toes. And if they are going to have to manage the same memory, you have to have some kind of strategy involving semaphores, Mutex is something to make sure that they don't have a race condition, they don't try to do it at the same time. That's certainly a cost. Conceptually, it's more work for your brain. Processes make things simple by saying you're going to go with that memory. I'm over here with this memory. We're not going to talk to each other, so we don't need any weapon? Yeah.

[Student comment about processes]

Pretty much where you process it all, so they, or at least they certainly did not treat it properly. They have the need to have pets or they have accidental comments. That is certainly true.

So if we look at actual software, web servers for example. The Microsoft Web server is thread based. It gets a connection from a particular client program. A thread is spawned, the thread responds to the request. Apache, which I think you'd all agree is a pretty popular web server, it's process based. It creates a process to handle the individual request. Both of these are very popular models.

One of the downsides of threads is it because you only have one process, everybody's riding in the same boat. So one of the things that can happen in an Apache web server is one of the child processes can encounter a problem and crash. And the parent just says that child had a problem and makes a new child and continues to serve pages. Whereas in the in the Microsoft version, everybody's part of the same process. So if that process gets a seg fault, well, that process got a segfault. That process is now being ejected from the CPU, so your web server just goes away because one of the page requests had a problem.

Thread programming tends to be a lot more fragile because you haven't made a whole separate environment for each child. But if you have no memory errors in your program, you may never have that seg fault to worry about. You therefore may be able to get more power out of your machine because you're not copying all these pages.

So as you're designing software, you can think about these trade-offs. Thread based programming is famously a lot more work than process based programming. This is why I gave you a process based assignment, I meant to do the actual thinking about how all the pieces work together before I asked you to think about the threading where you're simply mapping from what you had in Assignment 2 to a thread based model. So you get to focus directly on thread concerns and only thread concerns for Assignment 3.

But if you're starting a whole new thing, you're probably going to say, am I going to build this as a process or a thread based design from the outside? Threads are cheaper to create, cheaper to destroy. But you do have to worry about accidental or on purpose sharing of data between the various children. I will leave it there because there isn't really a right answer for different groups of programmers, they'll think about different things. (Processes, 3.1, 3.4; Threads, 4.1-4.3)

OK, we have enough time to just get started on the next module, which is I/O. So this is operating systems, we basically have to talk about all the pieces before you see how they all link together. So we've talked about I/O in various situations, but we haven't really formally talked about I/O as a task by itself. (I/O Systems, 12.2-12.5)

OK, here we have another map of the computer. As I say, next day I'll bring in a couple of motherboards. We can pass them around. You can see how all the pieces physically sit together. So the CPU memory relationship we've been talking about for weeks. But we know that the CPU memory relationship is not the be all and end all of a modern computer.

So we talked about the device driver with a interrupt vector served mechanism for calling the lower half of the device driver. What we haven't talked about is how that plays out in the larger computer. So you've got your CPU including all caching. We've got a transfer back and forth on the data bus to memory so we can bring in individual words of memory to do things with them. We can operate instructions on memory. We also know that we have this control bus to go to the I/O controllers where we can ask the IO subsystems to do stuff.

There's a back door into memory that I believe you talked about in 2030. Where you can do direct memory access. So this is a way for the I/O controller, which is effectively a second computer, right? It's a simple computer. It can say things like I just got a disk block. A whole whack of data at one time. Let's say it's 4K. So if a disk block is served to the controller and the CPU needs to get the data, we essentially have two strategies. One is that the controller is going to have to pass it back in what we call serial mode, one word at a time, and the CPU is going to have to be interrupted for every word and receive the word and figure out what to do with that word.

Or using DMA, the controller can say I'm just going to ZAP all 4K into memory. CPUs do another stuff. I just push all 4K into memory and then I have one interrupt to the CPU to say yeah, I'm done that big shopping that you asked me to do. It's all over there. All 4K starts at this address. See, I got more work to do.

So DMA is this back door in demand memory for bulk shopping for devices that support block access. If you've got some other device that doesn't support block access, like the absence of a keyboard or a mouse or a printer because they're all slow character devices, they have to do this route where they're doing one byte at a time or one word at a time, but your mouse isn't going to have 4K of data to show up with at one go, so it's not going to have this DMA bulk shopping plan.

So if we've got a block device, we can have a block device driver that says I need X amount of data, you know, 8K of data from such and such a file. Put it here. Tell me when you're done. I'm going to be doing something else until that interrupt happens. So you can very quickly get data to and from the CPU because it doesn't have to be involved in the laborious getting it off the device, it simply requests it and then gets informed later when it's done. OK, the most common block device is a disk. (I/O Systems, 12.2-12.5)

[Student question]

I'll bring some disks next day too, and we can pass them around. So a disc is called a disc because historically it had a round thing in it called a platter that spun around at speed and we stuck our data on the platter. (Mass-Storage Structure, 11.1-11.3)

So the platters in a disk drives have a way of storing digital information on the platter. How many of you have seen a cassette tape? Has anyone seen it? You've all seen a cassette tape. OK. Cassette tapes are usually kind of brown. They're brown because they're rusty. I actually had back in high school a wonderful little physics experiment where we took some plastic tape like, you know, Scotch tape. And we took a bunch of rusty stuff and we filed off all the rust and made a big pile of rust and dragged the Scotch tape through it and found that we could actually use it as tape and a tape recorder. You could just record stuff on there because you've put iron filings on your tape and it's magnetic.

So you put a magnetic image on the tape because you've got your little bits of metal has to be like steel or other ferrous metal and as you put sound on it, it has different magnetic information. You drag it across the tape recorder, read head, it plays back the sound. Disc is the same, but rather than one long strand of plastic, we're talking about one flat hub of metal, but we still have the ability that if we could position a magnet at a particular place, we can put a strong signal or a weak signal, and if we're going to have strong versus weak, we could do one and 0 and today we have the ability to store digital information. (Mass-Storage Structure, 11.1)

So our big problem becomes how do we find the same location again for where we put that bit so that we can figure out whether it was a 1 or a 0. So the disc we simply have mounted on an axle so that we can spin the disc around. Right. I'm sure you've all seen a record player similar kind of idea. So by spinning it around, if we had a read/write head just sort of hovering over the disk, all of the bits on a given stripe would eventually come under that read/write head. You don't have to do anything to read all the bits in that particular circular path.

If we want a different circular path, we can achieve coverage of the whole platter by just moving the head toward the hub or toward the exterior. And so we have what we call read write heads attached to an arm where we can swing the arm toward the hub or or toward the periphery, and therefore we get a circular path on the platter that the head can pick up. As long as the platter is circling around and around, we'll eventually be able to cover all those pieces. (Mass-Storage Structure, 11.2)

OK, so if you have a simple single platter disc which you probably have thought of as a floppy disk, it is again a little piece of plastic, basically exactly the same plastic as you would use for a tape, cassette tape, except now it's a round cut with ferrous material glued to the top and sometimes the bottom. You can therefore get twice as much space and storage out of your single platter if you have a read/write head above and a read/write head below, and if they're attached together to two arms that move at the same time, then moving it in or out is going to allow you to have two circular paths, one on the top, one on the bottom.

Yeah, that sounds so good. Why don't we do it more than once? So we stack up the platters with a little spacer in between so we can have a head above and below each one of the sets of discs, which, once we do that, we usually make them out of metal, not plastic. So you have a certain number of disks or platters technically. You have twice as many heads, one above and one below each one of the platters. And then we can move them in and out.

So that gives us a three-dimensional coordinate system. We can talk about which disc or which surface the data is on. That will tell us which of our set of heads we're going to activate to do read or write. We can talk about how close to the axle the heads need to be. And we can talk about Theta how far in rotation the disk needs to be as it zooming around and around. So they have a marker on there so it can tell when it has come back to the beginning so that it can count rotations. (Mass-Storage Structure, 11.2)

So if we park the set of heads at a given radial position, we're going to cover that circular path on all of the platters. We call that a cylinder. The nice thing about a cylinder is that within the amount of time that the disk has whirled around, we're going to be able to get every single piece of data within the cylinder read or written without moving the heads. To get to any other cylinder, we're going to move the heads either in or out.

So if you think about the disk as a concentric set of all of these cylinders. And simultaneously, a vertical layering of a set of platters. And simultaneously a set of pie sections around the disk. Then you have the three-dimensional coordinate system we use to talk about spinning disks. So we talk about that as a sector, which is a chunk of pie, a cylinder which is a radial distance and then a track or sometimes called a surface or a head that is simply which vertical coordinate you have within the disk. (Mass-Storage Structure, 11.2)

Now you might be thinking this sounds old fashioned because I have an SSD in my computer and do we still do this? Well, if you've ever bought a big disc, yes, because SSD's are expensive enough that you don't see them yet much above the TB level and they cost a lot even there. So if you're going to go down and buy like a multi TB disk. So this is exactly what it's still doing inside. And so probably for the rest of our careers, there's going to be a division between the small SSD based devices where you can wave them around and there's no problem. And the big cheaper spinning devices which are way slower.

But because these came first and because they're both disks, even though the SSD doesn't actually have any moving parts and none of them are round because it's just RAM chips, they still call it a disc and they still index it exactly the same way, because the device driver we want to be able to talk to both types of device without it having to spend a lot of additional code saying oh this is an SSD so I have to do a totally different thing.

So when you go to any kind of disk, whether it is a memory stick based or a grid based SSD or a spinning disk it is going to have cylinders, heads and sectors to describe where within that data structure it is. Even though the SSD's are fundamentally just a great big array, it still has this three-part cylinder, head sector indexing strategy. (Mass-Storage Structure, 11.3)

OK, so if it's a spinning disk, we can talk about the data rate using the physics because it literally is constrained by how fast the stupid thing is spinning. So one of the old Seagates runs at 10,000 RPM. That's about as fast as the faster disks go. They go at either 7200 or 10,000 depending how much you want to spend on the hardware to do it. 10,000 RPM means you have a six millisecond revolution. That means on average you're going to need 3 milliseconds for the disk to spin around to the right place to get your data.

It's going to spin at the same rate. So if it just happened to go by before you get there, just like taking the bus in Guelph, you see it disappear and you're going to wait till the next one comes. You can't make it come faster just because you would like it to come faster. It's going to come on its own speed because it's just attached to this whirling platform, so that gives it a three millisecond average latency you show up at a random time on average, you'll have to wait 3 milliseconds because sometimes you'll wait zero, and sometimes you'll wait 6. (Mass-Storage Structure, 11.2)

OK, so if we have 4 physical platters and eight heads. This particular disk has 49,854 cylinders, so these are discrete positions in which you can put the head between the hub and the exterior, and then on a particular track we can have 772 sectors. Now this means it's because it's a pie section. The data is actually more compressed together at the hub than it is at the outside. Because it's just spinning at the same speed. So if you move toward the hub, the actual bits are flying by faster. If you move outside, they're flying by more slowly, but they're actually bigger. The amount of the media allocated to them is more spread out.

This can have an effect on where you're likely to have a disk failure. But we have a fixed number of sectors per track. Most of these devices have 512 bytes per sector, and this particular device supports a 5 millisecond average seek time. That's how long it takes to move the head from one [cylinder] to another.

So if you know that you can just by multiplying these things together figure out how much time it's gonna take to get data off the disk. This is me just calculating how much is on the disk so that was 146 gig disk, which I got by just multiplying the heads by the cylinders by the sectors per track at half a K each.

Data rate is simply sectors per track times number of bytes you get per sector. And then we can calculate how fast this thing can get its data from the disk into the IO controller buffer. Now remember the IO controller buffer is that dedicated secondary computer where like a catcher in baseball, its whole job is to deal with the data coming in. CPU doesn't need to be bothered, so it's going to sit there, be ready to receive data at the speed the disk can feed it, and then it's going to pass it into RAM at whatever speed that can map.

So the disk to buffer speed we can calculate here and then the buffer into the computer speed is that of the speed of the controller to memory bus, which may be significantly slower. This is what it was for SATA and SCSI. (Mass-Storage Structure, 11.3)

OK, let's leave it there for today. We will start talking about how we can manage these things next day and then we will go into file system.