CIS3110 Lecture Revised Transcription (January 30, 2025)

Scheduling (Conclusion)

Alright everybody, let's set sail again. So we'll wrap up the process portion of the course in a couple of minutes. We'll move on to the memory portion. So the process portion of the course is the biggest chunk. Memory will be smaller, but also very interesting, so we'll answer some of the questions that have arisen so far in that section.

To wrap this up, though, I just want to wrap up talking about scheduling. So scheduling, the question is fundamentally, if you have a free processing unit, so core or CPU, you want to run something on it. How do you pick which thing to run? We talked about FIFO, we talked about shortest job first. We talked about their properties, we talked about time slicing and preemption, remembering that preemption hinges on this idea that you have a hardware alarm clock. So until they added the hardware alarm clock, it wasn't really possible to do preemption, because if you put somebody on a CPU, they would just stay there until they decided to leave because they invoked the kernel through a trap. They're not going to do a trap. There was no way to remove them.

So the alarm clock is a way to get the kernel running because you have a response to an interrupt, triggered because you've set the alarm clock. So to get this time slicing to work, just before you put something on the CPU, you set the alarm clock to say "please ring in X number of milliseconds." You put the entity on the CPU, it runs. If it doesn't ask for any I/O, so if it doesn't trap, and if no interrupt from someone else's I/O occurs, then we know we will get interrupted when the alarm clock rings. If we are interrupted early, or if that process asks for attention by triggering a trap, then we just simply reset the alarm clock when we put the next entity on, so we're always setting the alarm clock for the maximum time slice.

What we're talking about here is that based on the alarm clock timeout time, we can have a coarse time response if you set the alarm clock for a long period. That means there could be a long elapsed time before the kernel comes back and can do anything else, or response or interactive response. Better throughput because you've got fewer context switches per second.

The alternative is you set a small alarm clock time, smoother interactive response, but it just takes longer to get things done cause you've added that work of handling the context switch every time the alarm clock goes off.

Note that if there's so much I/O going on that the alarm clock never rings, then this distinction is meaningless, because if the alarm clock never rings, then the amount of time you set the alarm clock for is irrelevant. If somebody's always asking for I/O through a trap or someone's always being interrupted

because I/O requests are coming back, then the time slice can be changed without there being a need for the total elapsed time slice to occur.

OK, we have a special word for if the time slice is so small that it seems like everybody gets to run so often that they're continuously running, we call that processor sharing. And so you can get a fair number of processes in your run queue and have processor sharing work, but the reason it really works well is because even if you have, let's say, 100 processes running, the chances of them all wanting CPU time is pretty tiny.

So the command PS for process status tells you all the processes that are currently allocated a process table entry, so they're currently running in the loose parlance of the term, but they're not going to all be running in our parlance of being on CPU. The vast majority of them are probably waiting for I/O. So they don't take any processor time, they're just hanging out waiting for their I/O to get served. You can have hundreds of those and it's not going to make any difference to what's happening on the CPU.

So processor sharing is what happens on any of the modern operating systems, certainly anything you're going to likely have attached to a keyboard where you're typing on it, expecting things to happen. If the preemption time is infinite, so we just never set the alarm clock, then it's just FIFO. Whatever was running first has to complete. No one else is ever going to get a chance to run until they complete, so it is simple FIFO.

We can talk about batch, so this is the oldest style of computing. You loaded a program, a single program onto the computer. It ran to its full extent. And only after it exited did you load another program onto the computer. There's no need for context switching because there's no one to switch to.

So if you think about old time computing, as you may have seen in ancient movies where they frequently have a couple of tapes whirling back and forth, and some blinky lights in the front of your computer. Loading a program involved a human getting a tape out of a box and feeding it into the computer, and it ran as the only process on the computer. We now have replaced that human with a script management system, but we still have batch processing where one thing runs and only when it's done does the next thing run. Databases have a lot of batch processing. They essentially are miniature operating systems inside the database.

Interactive is what we typically see, so an interactive scheduler allows you to actually interact with the system in some sensible human time. Because you're interacting, the only way you can actually interact with your program is through I/O, because you don't live in memory, so therefore interactive systems typically have lots of I/O, and they're very sensitive to long gaps between being able to run on the CPU. Yeah. So you would not, for instance, schedule multiple seconds as your time slice because you're going to notice that in your editor as you start typing, and then suddenly there's a multi second gap before your letters appear on the screen.

The other important type of scheduling is what we call real time scheduling, which sounds a lot cooler than it really is, but I will say it is pretty cool. So the idea of real time scheduling is that you have usually a very well understood set of cooperative processes that are active at a particular time. And you can guarantee that if the system is running normally that you can predict how long a given process is going to need to run before it completes its current cycle.

So this happens a lot in control. Right. So robotics or other sort of navigation systems that you can imagine, if you've got a robot and it's going to drive around in a room, and you need to detect things like where there's a wall or where there's a hole in the floor or something like that. If your detector is running at too slow a rate, it will just drive into the hole or fall off the stage. We can't have that happen. Now. So you need to be able to say OK, well the perhaps navigation subsystem is going to run inside of 250 milliseconds and I know it will always be within 250 milliseconds. So therefore I can guarantee that's going to happen.

A real time system enforces this. So what it does is it says if you exceed your allocated slice and you're not done your work yet, you haven't either ceded the CPU by calling a trap, or you haven't asked for I/O, or you haven't essentially completed your work, then you're pulled off the CPU at a time slice specific to your identity as a process. Because a different process may have a different time slice, it might be expected to run much faster or much slower.

So in a real time system you can guarantee that you can go all the way around the loop of all of the running processes in a fixed period of time so that you can guarantee that that important process for, let's say, not falling in a hole is going to get to run again within a fixed maximum period of time. It might get a chance faster, but you can guarantee that you're going to go all the way around, because if someone's slow, you're just going to yank them out of the way. They're not going to get to finish their work. But you will guarantee they will no longer be hogging the processor.

OK, so this is a fairly old strategy. Famously, you all know who Margaret Hamilton is, but a name you know or not. Looks like generally not. So Margaret Hamilton worked for NASA. She's retired now. She was part of the Apollo Space mission. She wrote essentially all the guidance system for the lunar lander and for the command module. And she developed a lot of this real time design.

So famously, her real time system for the lunar lander correctly and safely landed the lander, even though because the instructions given to them were wrong, the astronauts in the lunar lander had the wrong set of programs running. And they had this extra, very expensive program running that essentially was trying to figure out what direction Earth was in as they were landing the lander.

Now I'm sure it was very important to know what direction Earth was in once they landed because they wanted to send a message. But I think certainly Armstrong and Aldrin would agree that not smashing into

the moon to their deaths was probably more important than knowing precisely what direction Earth was in.

So Hamilton's code correctly noted that a particular item in its control loop was taking too long, and even though it was trying to figure out where Earth was and doing this thing, it just kept yanking that off and going back to running things like the distance detector, like the thrust control module. And therefore they successfully landed, and it was only after they landed, because they could tell that the system was working really hard. Right, there was a light on telling it that it had actually exceeded its CPU time. So there was an error light on, but only after they landed did they have the time to figure out that in fact there was an extra program in play that should never have been there.

So the only reason we actually successfully landed the Eagle and Aldrin and Armstrong on the moon is because of Hamilton's code. So there's a famous piece of computer history where things actually worked out well. We have so many stories where someone has screwed up. This was Margaret Hamilton, and there's some great pictures. If you look for online, you'll see pictures of her standing next to the printouts of her code, which are essentially as tall as she is. Because there was a lot of well-written code that went into the Apollo mission. She was the manager of the whole thing.

OK, so real time is that very key. Make sure no one stays too long because someone else who's equally important needs their kick at the can in a predictable period of time. And so we now use real time in any control system where it's a computational system that is interacting with the real world on a real time schedule, so all flight software, any kind of navigation software. So if you think about any of these autonomous vehicles, they're all running under real time control because you need to be able to ensure that it's going to come back and do its computation in time to not hit the pedestrian or run around the corner.

OK, the last thing in scheduling is priority. So there's a number of different ways you can do priority. This is the way Windows does it, and this is the way the old VMS system did it. So essentially all they do is they have multiple queues. So we talked before about if you are woken up after a sleep, you get added to the run queue. If there's only one queue, then effectively everybody's at the same priority. You have your queuing discipline of how you process through the queue. But somebody's in the queue. They come to the beginning of the queue. They get served.

OK, all we're doing here is we say, well, we have a queue for the processes that should get run more often. And we have a queue for processes that get to run less often. So there's some processes where you want them to run very frequently. Editors are the key example. People get really frustrated if their editor stops responding to their keystrokes. Editors are easy to put in high priority because they don't actually take much runtime. Figuring out which key you pressed is not a complex task. The biggest task the editor has to do is look up the glyph to put on the screen.

Some programs don't need a lot of priorities. A lot of these are servers. So the print server for instance, can run at a somewhat lower priority because it's talking to this slow device anyway. So there's no need for it to keep running to the front of the queue to pass off a few bytes. It may as well wait longer and just pass off more bytes at a go. So the print conversation can go slowly. So we might indicate that the print server can be down here in the slow queue, but the editor might be up in the top queue.

And then all we do is we have a discipline that says, OK, so for every maybe 7 processes you take out of the priority one queue, you're then gonna take three out of the priority two queue and one out of the priority three queue. So everything's still fair. Everybody's getting through their queue. No one's stuck in the queue forever, but the more important people get to go through the queue more quickly than the less important people.

Now importance may change. So one of the things that these operating systems do is they watch to see how long you needed in your last time slice. And if you start needing the whole time slice, they'll start saying, "whoa, wait a minute here. I was giving you lots of runtime, but you're starting to make other people seem slow. I'm just going to move you down to the next queue." So you can still run, but other people who are going to be more sprightly than you are, they can run more often. And you're just gonna get another kick at the can after more of the faster processes have gone. And if you continue to be like that, you'll get moved further down.

And then perhaps you finish whatever the giant task was, or ten million numbers. And now you're back to usual processing, so they'll say, "Oh, well, you're way down here in the slow land, but you're acting like a process that could have higher priority. I'll just bump you back up." And so just bubble up and down simply by saying how long did you spend on the processor last time. And they'll weight that over a few rounds. So if you're continuously slow, you'll get moved down. If you're continuously fast, you'll get moved up and there are system controls where a system operator right, like root or administrator under Windows will simply say "no, no, this is a more important process. I want it to stay higher than we would otherwise have" or "this is the less important process. I'm just going to bump it down. It'll just take longer to run. That's totally fine."

OK. Questions on that. This is the last slide of the scheduling section. Are we good there? Alright then, let us shift gears. And enter the wonderful world of memory.

Memory Management

OK. So memory we know is important. For the world of the CPU, memory is the only thing in the world other than it, so this has got to be a pretty central idea in computing.

We talked about a process image. So in a process image we have that text segment. We have the data segment, we have the stack segment and we know that if you come up with a memory address you want to access that isn't in one of those segments, you get a segment fault. Technically, a segment fault just

means you've gone from one segment to another segment. But we see sigsegv and possibly core dumps if the segment you went to didn't exist, or if you don't have the right permissions to access it in the way you're trying to access. So if you try writing into the text segment, you'll have a segment fault.

So the text is simply your machine code, your executable machine code. The stack is your local variables and then everything else goes in the data segment, and that's because the data segment as we talked about before, it's an old design. So originally there was the text segment and then everything else you needed. Which is why they're all stuffed into the data segment.

So you've got your static data, constants, etcetera, any symbols. So if your debugger knows what function you're in, that information needs to be stored somewhere. That's the data segment. Any shared memory - we're doing shared memory in assignment 2 - that will be mapped into the data segment in a way that we are going to talk about very shortly. And then on top of that, we have the dynamic data and I've got a thing here on the side, "BSS, the block started by symbol". I don't even know why they called it this, ten million years ago, when they gave it that name. That is simply a pointer that tells you where the end of the static stuff and the beginning of the dynamic stuff transition.

OK, so we have our running process. It has this map of memory. It has in that map the entire universe, as far as the CPU is concerned. So when you're running a program, the only things you can read or write have to come from this map.

There is a system call called fork. Fork is a weird thing. So fork takes a perfectly good running program. And if you call fork, right, it looks like a function. In part, it acts like a function. You call fork. It has a return value. It tells you whether it's succeeded, whether it's failed, doesn't take any arguments. But not only does it return to tell you whether it succeeded or failed, but what it actually does is it duplicates the running process.

Everybody here remember Calvin and Hobbes? OK, boy and his tiger. Really little world. He's got strange ideas about lots of fun things going on. One of the things he talks about is he got this cardboard box that he plays with. And he's got a cloning or duplication mode to his cardboard box, so one Calvin goes into the box and a lot of Calvins come out. This is what fork does - takes a running program and makes a duplicate of the running program.

So a lot of the explanations of fork will explain it in kind of correct but weird ways. One of the ways you'll frequently see it as explained is "fork returns twice." Which it technically does. But not inside the same process. So you call fork, right? You are your single unitary self. You walk into the cardboard box. And then walking out of the cardboard box are two of you. One of them is the original. We call that the parent. The other one we call the child.

In the parent, the return value from fork is the process ID of the child, so this is the index into the table for the record associated with the new running process. In the child, the return value from a fork is 0. So you

know whether you're the clone or not. Science fiction fans are now suddenly very sad, so there's no secret which one is the parent, which one is the child. But what you've done is you've duplicated the process.

So that's a lot of work. If we think about all this memory, that means we have to photocopy all of the stack memory. We make a copy of all the data in the stack because the child can look at the same local variables, return from the same functions. It's literally a clone, so we need a whole copy of the stack. We need a whole copy of the heap. Anything that was malloc'd. If the parent calls free, that doesn't mean the child's memory goes away. They're copies. They're not somehow joint.

So we need to copy all the data for all the allocations. All of our static variables, so if we have any globals? If we have any static locals, they all have to get copied. But we don't need to copy everything. Read only values can be shared. So we don't copy the machine text. The instructions are the same. It's the same program. So you can literally access the same memory and read it from 2 readers, because if you got 2 readers reading the same book, the book isn't going to change. It's only if you have a writer that the book is going to be updated, so all the parts of memory having to do with the text segment can be shared.

We do the same for any constants. So the symbols or any other constants we have like our string table, that can also be shared. So if we've got a way of sharing the same data between the two, well, that's how we're going to do shared memory. So shared memory is just writable memory that happens to be shared. We'll talk about exactly how we do that in a second. So this is what we call a process clone or a fork.

We also have a thing called a thread and I brought that up a few times, but not formally described what a thread is. So a thread is sometimes described as a lightweight fork. Because what we do is we say, OK, I'm going to clone the process, but the only parts of it that I'm gonna actually make a new copy of are the stack pieces. Anything having to do with the stack. Everything else is just going to be shared. So if it's shared memory, well it's shared. But if it was dynamically allocated, well, too bad it's shared. Many of the static - it's shared. Text - it's shared regardless whether it is writable or not.

OK. What does that mean for critical section thinking? Anything that is shared between two or more processes essentially needs to be managed with a critical section. We don't need to do that with fork because although it's shared, if no one can change it, it is irrelevant if we protect it with a critical section. There's no writer, there's no problem.

With a thread, what we find is that everything having to do with allocated memory or any globals or static values, we are going to need to protect with a critical section or our two different threads can stomp on each other and potentially screw each other up. But they do get their own private stack. So they do have a place private to themselves to store values, to figure out what's going on.

So in the thread world, a typical design is to have a local variable space where you can keep track of what other parts of the data you're allowed to write to. And manage everything else through semaphores or mutexes.

So in assignment 2, what I've given you is code to fork your program. And one block of shared memory. So it is obvious where the shared pieces are. They're only in the shared block. Assignment 3 will play with threads, so then we get to do fancier things on our own.

One might ask, why do we bother having both of these things? Speed. So it's expensive to duplicate a process. Fork is a lot slower than creating a new thread. We also talked about the fact that sometimes you might want to have a particular thread do part of the work. Very common to have one doing I/O work and then another thread doing background work, so they're both writing to a shared dynamic data segment, shared static data segments. They're keeping track of what they're doing with their own stack.

And I'll say the consensus now is that a thread has its own stack. But I will say if you go back a few years before really the dust had settled on the idea what a thread was, different vendors had different definitions of threads. So this is a POSIX thread. If you go back and you look at other threads, Sun, for instance, they gave you a single word that was the only thing that you had of your own, and everything else was shared. And it was up to you to figure out how to make do with just that. So you used your ID value to keep track of who you were. And then you somehow said, OK, I'm going to have maybe a set of tiles. I'm going to talk to tile three. My buddy is going to talk to tile 5. We'll figure out how not to step on top of each other. But it's only the stack that is available to you without management of potential race conditions. Otherwise it's up to you to make sure that your different threads are not writing to the same memory at the same time.

We now call that super lightweight version a fiber. Where you just have a single word for your processing element and every other part of the work has to be managed manually. Oh no, sorry. That is Tannenbaum's definition. The more common POSIX definition is that a fiber is a thread that doesn't have preemption.

So backing up a moment. A fiber in the POSIX standard is a thread as far as the shared memory goes. A thread in the POSIX standard is scheduled just like any other process. Run queue element. OK, so it actually gets a separate entry in a process table, even though it's got shared memory. So you can therefore assume that preemption is happening, that the scheduling algorithm is figuring out who gets to run when.

A fiber in POSIX is a thread that within the process requires the thread to manually put itself to sleep to allow another thread to be run. Extra work, but possible to make sure that you then, because you don't need a scheduling algorithm to take some time to figure out who's going to run, you're essentially manually passing the baton around among the various fibers making up the single process.

So a process always has at least one thread. If you call fork, then that thread going into fork will cause the entire memory to be duplicated, creating a new process with one thread regardless how many threads

were in the parent process. So you've got 10 threads running around in a parent process. One of them calls fork. The result is a photocopy of the memory of the parent process, but with one thread in it.

In any process you can ask for more threads to be created. As soon as you ask for a thread to be created, it does this lightweight duplication. So you can have within a given process as many threads as you like. You then decide what work the various threads are going to do inside the shared memory map of your single process.

If you were to, let's say, run the same program again. So now you've got another copy of that program. It will have an identical set of pages potentially. So I haven't gotten to the page part yet. It'll have an identical memory map. But because it's been started independently, it's not part of the sharing. So if you got two people, let's say running vim, they're not sharing memory. They're just both running a program called vim, got loaded from the same file, got set up in memory the same way, but they're two totally distinct processes. If that program decides to call fork, it's going to photocopy itself. It's going to make its own duplicate. So far so good.

OK. So we will come back to fork in a few days. But first I want to talk about how the memory itself works here.

So in the physical world of your machine, you have a certain amount of RAM installed. So the RAM or random access memory is a storage bank of a fixed size based on how many chips you bought at the store when you bought your machine. And it has to run everybody. That one fixed size storage has to accommodate everybody's memory requirements.

But you might be running hundreds of programs. And several of those programs may not even be runnable at the moment. They may be blocked for I/O, so taking the memory and simply partitioning it up in, let's say, equal size stripes among your 100 processes seems like a foolish waste of time.

What we do is we do a virtual memory strategy. So when your program is running, your program is running in a faked up world where it sees what looks like RAM going from null up to some big number. But that number line is unique to that program. The data at a particular location can be different for every program, even if it's the same location.

OK, so we're going to call this a virtual address space. The kernel works in the real world, however. So it has to see and manage all of these virtual address spaces, but it is the only thing that can look at the real memory world underlying all this.

Because we've got a private number line for each of the processes, they literally cannot see each other's memory. There is no way on a virtual address machine that one running process can reach out and grab values out of a different running process. They're all on a distinct number line.

We're going to use this strategy that we use to make the distinct number line to also break the limit so that we can actually satisfy large space requirements to store potentially hundreds of programs in a limited amount of RAM. And it's going to hinge on the fact that even though we have in our process image this potentially really long distance from null up to the top of memory, it's actually mostly empty. That gap in the middle is enormous.

So we need some allocation to real memory to store the stack in and we need some allocation for real memory to store the text and the data in. But this gap in the middle doesn't actually need to be stored anywhere, we just need to know how big it is so that if you make a pointer fall into it, you can say no, that's not allowed. But beyond saying there's nothing really there, we don't actually care what could be there or should be there.

So because we have this big empty space, we're going to use the big empty space to balance off part of the need for large environments for many programs. We're going to do this using a strategy we call paging.

So a page of memory is a fixed size piece of a program. Only the pages we currently need are gonna be put in real memory. The rest of them are gonna be stored in essentially an external, you can think of it as a backing, full memory we might need later.

So paging works like this: we take our memory image. We're going to use another loaf of bread metaphor, so if we've got our long loaf of bread, imagine it's a wonderful French baguette. OK, you've got the one end. You've got your text and your data. You've got the other end and it's the stack. We cut our bread up into many fixed sized slices. Each slice is a page. The pages are all the same size. This is going to be critical.

So if we have a page of a given size, that means if we have an array of places to put pages, we can put any page in any spot in that array. OK, so this is usually defined by the hardware of your motherboard, so a typical Intel page size is 512 bytes. You've taken your whole running process image and divided it up into 512 byte slices.

So if we think about the first page, which I note is not zero page, page #0 - page is a RAM location having to do with the hardware of the underlying machine. This is just the beginning of your program. This virtual space completely omits the hardware cause, remember, the kernel is the only thing that needs to be able to talk to hardware. All we need to do here is arrange the bytes in RAM so that our program can run.

So the first page starting at 0 is just the bottom of your program. The first few instructions of the program. So from byte location 0 on page 0 up to, let's say byte location 511 on page zero, that's the first 512 bytes of your memory. From byte location 0 on page 1 through byte location 1023 of your program, which is byte location 511 of page one, that's the next bit of your program, the next chunk. So the first

page is simply the first slice, second page is the second slice, and you can talk about the bytes on a page by simply knowing what page you're talking about and then talking about how far into the page you are.

So here you can see I've said Page 3, byte 2. If we happen to have an unrealistically small 40 bytes per page, is simply the absolute address in our program of three because we're on page three times the page size of 40, plus 2. Well, 3 * 40 + 2 is 120 + 2 or byte 122 within our program. Right. Simple addition. Little bit of multiplication. We're great.

OK, so this allows us to make the data arrangement on a page independent of where the pages actually are. And notice that the ones in the gap we don't even need to think about what's going to be on them. All we need to think about is what is on the partial page that is the top most data segment page, what is on the partial page that is the lowermost stack page, and then all the pages that are completely filled.

This is going to allow us to map between a physical storage and a logical storage, because as long as we've numbered our slices of our bread, we can take the bread and jumble it up and store them in any order. And if we can pick up one and say, "yeah, this is page 7," then you know that you have 7 pages from the beginning, up to the page size. That much memory out of your program. It could be stored at the beginning or at the end, or anywhere in memory as long as you know that it is the 7th page of your program when you're accessing. Everybody with me so far?

OK, so if we're doing this jumbling up and then unscrambling in order to find things later for all of the memory making up all of the running programs, it sounds like we're going to need to do this work a lot. We need to do it so much, we do it in hardware. So this is what the memory management unit does. It takes a virtual address. So this is exactly the address that you've seen in any pointer you've ever made in your life. It takes the virtual address and says, OK, in that virtual address a certain number of the bits are talking about where we are on this page. And the rest of the bits are talking about which page we're on. It's simply a most significant, least significant relationship.

So let's say if we do this in decimal, let's say we had 1000 bytes per page. Then the math in decimal is quite easy. You take your pointer value. You say if I divide by 1000 and throw away the remainder, that tells me which page in my process I'm on. If you take the value and modulus by 1000 to compute the remainder, that tells you which bit on the page you're on, if you knew which page it was.

So we simply take part of it is the offset, part of it is the page number. But we don't do it in decimal. Why do we not do it in decimal? We're going to do it in binary. Because we would like an integer number of bits to be in the offset and an integer number of bits to be in the page number, because then it's much easier.

This coming down from the offset then doesn't have to be fancy math where you're calculating modulus. This is just simply some wires. Because it simply says whatever bits you have here in the offset, well, let's just bring them on down. All I do is I simply don't keep the bits to the left. You don't have to do the math. You're just saying, "Yeah, just that chunk of the register."

So if you're using that chunk of the register as the offset, well, obviously you want all of the other bits for the page number, so we simply take our register and literally draw lines from the bit storage locations to take the offset portion down into an addition unit, and then we take the page number portion over to what we call the translation table.

So the translation table is a very simple scavenger hunt. It simply says if we look for a given page, we take that number and say "how far in the table do I have to go indexing this based on this number?" Let's say this number is a four, then you go to slot 0, 1, 2, 3. You read that number out of the table. This is what we call our page frame identity that tells us in our storage which slot we've put the bread for, in this case, page 4.

So this is telling us that for our running program, the first page, the one at location 0, was stored in frame 5. And the second one was in frame 9 and the third one was in frame 7. And as long as the translation table can tell you where to find where you put the data, right, it's like going to the library. If it tells you where to find the book on which shelf, they can have an arbitrarily complicated filing system as long as you can come in and say, "I'd like this book" and they give you "it's on that shelf." They're just looking it up.

This is simply an array, so all we're doing is we're taking the number here based on the leftmost bits, using that as the index into this table, looking up the value in the table to access it from our page frame where we've stored the real page data. Add on the offset and that tells us in our physical memory where our data is. So we've simply sliced our physical memory up into page frames of the same size as our page size. And then we can access any byte out of any frame, even though they're no longer in order.

You ready for an example? Oh, so I have a summary page first, so here's the amazing math. So if we divide our memory image into pages, think of it like a book analogy. So if you got a book, you've got pages in a book. You've got a certain page number on that page, there are letters you can talk about which letter is on that page of that book. I'll also find I did not invent the slice of bread analogy. There's lots of people who will talk about carving up your loaf of bread into fixed size slices.

So then mathematically, to get the virtual page number, we're simply taking the virtual address, integer division by the physical page size. And the virtual address modulus the physical page size to get the page offset. We take our virtual page number, look over to our lookup table, the page table, look up the frame number entered in that table based on the page number we just calculated. We take the frame number we got from the table, multiply by the page size to undo this division, add on the offset so we can combine the offset from this operation and that gives us our physical address.

So this happens with a whole separate subsystem on your motherboard in between the CPU and memory. So the CPU, it turns out, speaks only in virtual memory addresses. The MMU does this magic and then the actual physical address is what comes out of the back end of the MMU. So it's doing this so the

CPU can be blissfully in its own world of virtual addresses and the MMU fixes it up to find the real data on the motherboard in the RAM chips.

OK. We are getting to an example. A little more information before we get to the example. In the hardware page table, we have more information than just the list of frame numbers indexed by page numbers. For each entry, we can talk about whether the page is valid or not. That means it's in memory, not to be confused with not part of the process. This just means whether or not the page is in memory. We'll talk about what happens if they're not in memory in a minute. Whether it is writable. Depending on your vendor, this is either called RO for read only or overbar W for not writable. The overbar you probably remember from an earlier course I hope is essentially logical not in the Boolean world. So overbar W simply means not W so not writable means read only.

There's also two bits for dirty and used. We'll come back to those shortly, and the page frame number. So this is the record at each location in the hardware page table. We're going to find out that we have an additional page table entry talking about where this is on disc cause you've probably heard of your swap space or your paging disk. We're about to talk about how that works.

OK, so the valid bit - this is a common misconception. A valid bit tells us whether the page is currently in memory. We use our knowledge of the top and bottom of the data segment and stack respectively to figure out whether the page is legally part of the process. So figuring out whether we have an invalid segmentation fault is somewhat confusingly, not directly related to the valid bit. As you can have an invalid page, it just means it's not currently in memory. Of course, if it's not part of your program, it is also not going to be in memory. Everybody good. Can we move on to the next one?

OK, so here's our first example. OK, we've got basically 3 tables of information here. The one marked page table is the one we've just been talking about. For everything inside the dark border are going to be the pieces of information about a given page lookup in memory. So we've got V for valid, RO for read only, D for dirty, U for used, and then a wider column for the page frames.

You can see I've indexed it from zero up to S so we've only got 16 pages in this extremely tiny processor, but that's enough for us to see what's going on. You can see that not all of the page frames have been filled in. Some of them are empty. Not coincidentally, the empty ones have a zero in their valid bit, so the one in the valid bit means there's data in the page frame column. A 0 valid bit means there's no data in the page frame column.

But we can see that for some of those we have this additional gray colour where there's an additional entry over here. Sometimes there's an entry for both. Sometimes there's an entry for one, sometimes there's an entry for the other. We'll talk about all these situations in just a moment. You can see that there's a big empty space in the middle where things are both invalid and there's no page frame inside the

black border, and there's no entry in this gray disk address column either. That's the gap. We try to map there. We're looking at segment fault.

OK. Also on this page, I've got another tabular structure that I've labeled main memory. These are our page frames, so this is a place in RAM. We can put the data for a particular page. The location in main memory is stored in the page frame column of the page table. So each of these numbers in the page table refers to one of these frames. If we come to, let's say, page E, we're going to go to frame 11. I've got 2 columns here just for saving space as zero and one are in the same row, then 2-3 and so on. SO 11 is the one that is the 2nd frame in row 10 in the main memory. So that's a 512 byte allocation which you can see is the allocation I've written down here in the example. So that's 512 bytes of RAM set aside to hold the data for page E but in frame 11. Everybody with me so far.

The other thing I have here is labeled paging disk. It is a very similar indexed set of page values. The only difference between the values on the paging disk and the values in main memory is that the disk is a different device. It is literally a disk. So as a disk, getting the data onto or off of the disk is going to be a very expensive endeavor. We have to schedule some I/O, we have to make all that happen.

The CPU can't see the disk. We just said the CPU can only see anything in RAM. So if the CPU wants to run an instruction or get data for the instruction or store data from the instruction, all of those activities can only happen with respect to some part of main memory. So what we're going to do is we're going to talk about what happens during the instruction address cycle so that it can get the data from memory in order to actually execute an instruction. You're with me so far? Yeah. Oh, we had a question.

OK, I'm really glad you spoke up earlier. OK. So the key to keep track of your frame, let's take your frame. You could think where the place of football. But if you just have a picture frame, it's not very different. You're going to put a picture in the picture frame. And exactly the same way we're going to put a page in the page frame.

So the frame table, also called on this slide main memory, is an array of tiles of the page size, but until you put something in the array location it's not interesting. So it's an allocated size where we're going to store the data. The page is the data. So the page or the values we're going to put in a particular page frame. So the page table is telling us which frame, so which tile within main memory, the data is for a specific page in our program. So the page is figured out from the virtual address. So this is the example we're about to do. We're going to take a virtual address, carve it up to figure out which part is identifying the page, which part is the offset, using the part identifying the page. We're then going to use this table to figure out which frame that page is in. We're good so far. OK. Did anyone else have a question that I didn't see? OK.

Oh there is one here, yeah. "512 bits. Pieces of your bread. So when you say pointer I want to make sure it's clear in your program this pointer again. They're just stored on page. In the memory they seen by the book. In a sense." We're talking about a pointer here, but we're not talking about a pointer in terms of a

word by memory address pointer until we get to the final state. OK, so what we're going to do to get an address, which is what you would put in a pointer in main memory, we're taking a different address in a different number space, the virtual address. So that's an address inside your program. And figuring out where in main memory we've stored the data associated with that virtual address. So they're both addresses. But we're not going to do a typical kind of pointer arithmetic approach here.

OK, let's do an example and then we'll loop back on this and make sure this is clear.

OK, so let's say on our processor, so we went to the store, we bought a processor, it has a page size that the manufacturer decides, we can't change it. This manufacturer might be Intel, has said 512 bytes is your page size. That's Intel's favorite size so that's what we got.

OK, so that in every one of our virtual memory mathematical operations is based around this 512 byte page size, so 512 base 10 is what in base 16? And then when you gonna jump in or you want me to just go ahead? Oh yeah. I usually say 200 just to make sure we keep in mind that those zeros are not base 10 zeros. But yes, it's going to be 200.

OK, so that means we have two full bytes plus one bit worth of offset. 9 bits of offset, the remaining bits are our page number.

OK, so on this machine, if we're trying to read virtual address hexadecimal 72E, we're gonna take our 512 base 10 or hexadecimal 200 which is 2 to the 9, 9 bits of offset. So what is the offset number we're going to get from taking 072E mod hex 200?

It's actually pretty easy if we do it all in hex. Because what happens? Let's say you were doing a decimal modulus. If I told you you were going to take something mod 1000, well, you're just saving the last digits. Right. What if it's mod 2000? You're still saving the last digits, but you got a little bit of work to do now in that two column. Same deal here. So that 00 here simply means we're preserving the last two digits. And then we're going to take from this column.

What is hexadecimal 2 in binary? Well, hex digits are how many bits? Anyone? Four. OK, so we only have to worry about four positions. Hex 2. How many bits are turned on? For the two, of our four bits, how many of them are active? One. OK, which bit is it? We've now got 4 possibilities. So what is the binary pattern for hexadecimal 2? It's...well, there's only four bits, right? So we're going to have 4 zeros at the end. Ignore these zeros for a moment. If we're just talking about hexadecimal 2.

"No." Almost, but it swapped the middle 2, right? 0010 cause remember binary one is 0001, binary 2 is 0010. Hex at this point is the same as decimal until we hit 9. After nine, we're going to go to A. But you've only got 16 possibilities. Right. This is the whole point of hexadecimal is it allows us to say inside of that binary number, I want to talk about these particular ranges because I don't want to write out ones and

zeros all day long, but I do want to be able to say "oh yeah the 9th bit on the left, so I'm going to do something with that" and the rest of them, I'm going to do something else.

So all we're gonna do for mod hex 200 is save the 9 bits that are the right-most 9 bits. Well, we've already convinced ourselves that the rightmost 8 bits are quite easy. Because whatever digits we have here, we're just going to propagate into the offset. So the only question we now have is what goes in bit 9. What is gonna control the value in bit 9? Which column is it of the 3rd digit in from the right? There's only four columns because it's one hex number. So is it the ones column, the twos column, the 4th column, or the 8th column? It's on the edge, so it's going to be either the ones or the eights, right? It's the ones column. It's the least significant bit in that column.

So our question of what goes into that 9th column effectively is the question "is this an odd number?" Because if it's an odd number, then we're going to have a one in the ones column. And if it's an even number, then we're going to have a zero in the ones column.

OK, so to do this math, which I'm sure looked terrifying when I first put it up here, all we're doing is we're writing down the two digits from the right hand side and looking at this and saying "is it odd?" Oh, and then we do the...sorry I did them in the other order here, so I've given you the answer for the next part which is we take the remaining parts of our 07 and we're now dividing by hex 200, little bit more tricky.

But if you think about the fact that we're just using a binary number and chopping off 9 digits from the end, all we're doing is we're taking the remaining 3 bits from this 7 + 1 bit from the next number over. Binary 7 is what? How do we represent a 7? Yeah, 111. So in a hexadecimal 7 it's a 0111. We need that fourth bit. So we took 0111, we chopped the least significant one off. And then now we're going to shift it so that the twos column of the seven is now the least significant bit. So that one one, chopped off, becomes 011 which is 3.

Hex 72E over 200 is the rightmost 3 bits of the seven shifted down to the bottom location. Therefore we get a three. We're going to be looking at page 3.

OK, so we're looking at page three. We now simply go to that location of this table, 3. It has a one in the V column, what does that mean? The valid bit is true. What does it mean if it's valid? It means it's valid, but what is being valid mean? Yeah, it's in memory.

OK, so if the valid bit for our row is 1, then we can walk over to memory. We got a D. That means we're in frame D. So we can take our D, oh yeah, and there's the remainder, the 2E. So page 3 maps to frame D by simply looking it up in the table and noticing it's valid. We now undo our math, so we multiply D by hex 200. Shifting that over 9 positions, we get 1A00. We now add in our offset.

The nice thing about this addition, if we were to build it out of out of chips, we wouldn't even need a carry bit because we've guaranteed to ourselves that we're never going to have a 1 and a 1 added together

because we've shifted these values left nine positions and we know that we generated this one by keeping only 9 positions. So all we're doing is, we say add, but we're really just saying bit-wise OR. We're just combining the bits together. That gives us physical address 1A2E.

So the MMU has said "I took this address. I did my divide. Found a frame. Frame was valid. Held the indexed frame ID out of the page table. That was frame D. Multiplied by my offset to get the frame address, added in the offset within the frame address. I have a physical address."

So what the MMU is doing is taking this number in. It is hardware configured to do the math for 512 bytes, cause we said that's a property of the processor. So that's just built into the silicon. So it divides by 512, multiplies by 512, and just carries the offset bits down so it can combine. This is a read, so we don't need to look if the data is read only because we're doing a read. You're allowed to read read-only data. You're just not allowed to write read-only data, so we could have skipped that step here.

You ready for another example? Let's say we have an 11 bit offset. OK, so now we're going to have 11 wires brought down. The 11 is simply hex 800. Again, there's only one bit on in this offset number, it's just where the bit is. You can think of it like our knife. We're cutting our virtual address into the offset portion and the non-offset portion.

If we're doing a read again here for address 77FF, we take 77FF, divide it by hex 800. Well, again, we see some nice happy zeros here at the end, so we're going to shift over at least that far. How far through the hex digit is the one for the 8? Is it in position 0, 1, 2 or 3? Is it the ones column, twos column, 4th column or 8th column? I mean, it's an 8, yeah. In the fourth column. Yeah, the 1, 2, 4, 8. You can write it. It is therefore the leftmost column.

So we're going to save 3 bits from this particular 7. We're going to have one bit leftover. So we're going to have a final number, which is the bits in this seven with the single one we're curving off or the single bit rather we're carving off of the left hand side of this digit. What is the bit on the left hand side of a hexadecimal 7? Is it a zero or a 1? A 0. It's less than 8, so it has to be a 0. So our result of our division is going to be the three bits from the seven followed by a single 0 bit. What is 1110? It's an E. Yes, 14.

We then have 7FF remainder. We go to page E, page E up here is valid again. Now read only is 0, but we don't care because we're doing a read. We come over here and it's in frame 11. So we say page E turned into frame 11 by looking it up in this table. We then do our multiplication. SO 11 shifted over 11 positions. Add in the 7FF, we get 8FFF remainder.

OK. How much time we got? Time for one more. Here you go, offset of 11 bits is example #3. Same offset. This is the same example that I accidentally... Yeah. Ah, because I didn't show you the math of multiplying it on the previous page. OK, so example 2 and example three are actually the same example. Sorry about that.

So that's example 4. Different virtual address coming in. Same division remainders now 72E cause again 7 being less than 8, the most significant bit is off. If the most significant bit is off and we're only carrying the things to the left of the 11th bit, we don't have any data over there. So that's page 0. We come here. Not valid. OK, not in memory. Page fault. Page fault means we did not find the answer in the page table. Page fault results in an interrupt from the MMU. So at this point, what happened is that while rattling along executing our program, the MMU just sorting out the memory requests as we go. The MMU has said "help, I can't satisfy this one." It asks the kernel to solve this problem. So we've suddenly jumped to an interrupt handler from the MMU back into the kernel which is going to be free to do arbitrarily complex tasks to sort out this problem for us.

OK, so we'll come back next day. And we'll talk about how that all works. OK, so page fault simply means the MMU encountered a bit that was not valid, so it knows that if there was any data here it's garbage. The MMU cannot solve the problem of "where should I go in physical memory to give you back the data you request." OK.

Alright.