**CHARLES LEISERSON:** Hey, everybody. Let's get started here. So last time we had the skull and crossbones, this time we're going to have double skull and crossbones.

This stuff is really hard and really fun. And we're going to talk about synchronization without locks. And to start out, I want to talk about memory models. And in particular, the most important memory model from a theoretical point of view, which is sequential consistency.

And to introduce it, I want to use an example to introduce the notion of a memory model. So suppose you have two variables, a and b, which are initially 0, and those variables are stored in memory. And processor 0 moves a 1 into a, then it moves the contents of ebx into b. And meanwhile processor 1 moves a 1 into b, and moves the contents of a into eax. I just chose different registers just so we can distinguish the two things.

Now let's think about this code. We have these two things going on. Is it possible that processor 0's ebx and processor 1's eax both contain the value 0 after the processors have both executed their code? They're executing in parallel. So think about a little bit.

This is a good lecture to think about because, well, you'll see in a minute. So can they both have the value of 0?

So you're shaking your head. Explain why?

**STUDENT:** So if ebx is greater than [INAUDIBLE] then it's [INAUDIBLE].

**CHARLES LEISERSON:** OK, good. And that's a correct argument, but you're making a huge assumption. Yeah, so the idea is that, well, if you're moving a 1 into it, you're not looking at it. It may be that one of them gets 0, and the other gets 1, but it actually turns out to depend on what's called the memory model. And it took a long time before people realized that there was actually an issue here.

So this depends upon the memory model. And what you were reasoning about was what's called sequential consistency. You were doing happens before types of relationships and

saying, if this happened before that, then. And so you had some global notion of time that you were using to say what order these things happened in.

So let's take a look at the model that you were assuming. It's interesting, because whenever I do this, somebody always has the right answer, and they always assume that it's sequentially consistent. It's the most standard one.

So sequential consistency was defined by Leslie Lamport who won the Turing Award, a few years ago. And this is part of the reason he won it. So what he said is, the result of any execution is the same as if the operations of all the processors were executed in some sequential order. And the operations of each individual processor appear in this sequence in the order specified by the program.

So let's just break that apart, because it's a mouthful to understand. So the sequence of instructions as defined by a processor's program are interleaved with the corresponding sequences defined by the other processors' programs to produce a global linear order of all instructions. So you take this processor, this processor, and there's some way of interleaving them for us to understand what happened. That's the first part of what he's saying.

Then after you've done this interleaving, a load instruction is going to get the value stored to the address of the load. That is, the value of the most recent stored to that same location in that linear order. So by most recent, I mean most recent in that linear order. I'm going to give an example in just a second. So it doesn't fetch one from way back, it fetches the most recent one, the last write that occurred to that location in that interleaved order that you have picked.

Now there could be many different interleaved orders, you can get many different behaviors. After all, here we're talking about programs with races, right? We're reading stuff that other things are writing.

And so basically, the hardware can do whatever it wants. But for the execution to be sequentially as consistent, it must appear as if the loads and stores obeyed some global linear order. So there could be many different possible execution paths, depending upon how things get interleaved. But if you say, here's the result of the computation, it better be that there exists one of those in which every read occurred to the most recent write according to some linear order. Does that makes sense?

So let's do it for this example. So here we have our setup again. How many interleavings of

four things are there? Turns out there's six interleavings. So those who've taken 6.042 will know that, right? 4 choose 2.

So the interleavings, you can do them in the order 1, 2, 3, 4, 1, 3, 2, 4, 1, 3, 4, 2, et cetera, et cetera. But notice that in every one of these orders, 1 always comes before 2, and 3 always comes before 4. So you have to respect the processor order. The processor order, you have to respect it.

So if I execute in the first column, if that's the order, what's the value that I end up with for eax and ebx?

**STUDENT:** 1 and 0.

**CHARLES LEISERSON:** 1 and 0. Yep. So it basically moves a 1 into a, then it moves b into ebx. b is currently 0, so it's got a 0 in ebx. Then processor 1 moves 1 into b. And then it moves a into eax. And a at that point has the value 1.

What about the second one?

**STUDENT:** 1, 1.

**CHARLES LEISERSON:** 1, 1. Good. Because they basically are both moving 1 into their registers, then they're both storing.

What about the third one? Yeah?

**STUDENT:** Same.

**CHARLES LEISERSON:** Same. OK, fourth one? We'll try to get everybody [INAUDIBLE].

**STUDENT:** Same.

**CHARLES LEISERSON:** Same? Yep. Fifth one? Same. Last one?

**STUDENT:** 0, 1.

**CHARLES LEISERSON:** Yeah, 0, 1. Good. So this is the total number of ways we could interleave things. We don't know which one of these might occur because, after all, the output is going to be non-

deterministic upon, which it is.

But one thing that we can say for certain is that if you have sequential consistency, there's no execution that ends with them both being 0, which is exactly your intuition and correct rationalization. But it turns out interestingly that of modern computers, none implement sequential consistency. Why? Because life would be too easy then. None of them do that.

So we'll get there, we'll talk about what modern machines do. So let's reason about sequential consistency. So the way that you can formally reason about this, to make an argument as you might have for example on a quiz, if we had a quiz coming up, would be to understand that an execution induces a happens before a relationship that we will denote as a right arrow.

And the right arrow relation is linear, meaning that for any two instructions either one happens before the other or the other happens before the one for any two different instructions. This is the notion of a linear order. The arrow relation has to respect. The happens before relation has to respect processor order. In other words, that within the instructions executed by a processor the global order has to have those same sequence of instructions of whatever that processor thought that it was doing.

And then a load from a location in memory reads the value written by the most recent store to that location according to happens before. And for the memory resulting from an execution to be sequentially consistent, there must be a linear order that yields that memory state.

If you're going to write code without locks, it's really important to be able to reason about what happened before what. And with sequential consistency, you just have to understand what are all the possible interleavings. So if you have n instructions here and m instructions there, you only have to worry about n times m possible interleavings.

Actually, is it n times m? No, you've got more than that. Sorry. I used to have good math.

So one of the celebrated results early in concurrency theory was that fact that you could do mutual exclusion without locks, or test and set, or compare and swap, or any of these special instructions. Really remarkable result. And so I'd like to show you that because it involves thinking about sequential consistency.

So let's recall, we talked about mutual exclusion last time and how locks could solve that problem. But of course locks introduced a lot of other things like deadlock, convoying, and a variety of things, some of which I didn't even get a chance to talk about, but they're in the

lecture notes.

So let's recall that a critical section is a piece of code that accesses a shared data structure that you don't want two separate threads to be executing at the same time. You want it to be mutually exclusive. Most implementations use one of these special instructions, such as the xchg, the exchange instructions we talked about to implement locks last time. Or they may use test and set, compare and swap, load linked store conditional.

Are any of these familiar to people? Or is this new stuff? Who's this new for? Just want to make sure. OK, great.

So there are these special instructions in the machine that do things like an atomic exchange, or a test and set. I can set a bit and test what the prior value was of that bit as an atomic operation. It's not two sections where I set it, and then the value changed in between. Or compare and swap, we'll talk more about compare and swap. And load linked store conditional, which is even a more sophisticated one.

So in the early days of computing back in the 1960s, this problem of mutual exclusion came up. And the question was, can mutual exclusion be implemented with only the loads and stores as the only memory operations. Or do you need one of these heavy duty instructions that does two things and calls it atomic?

Oops, yep, so I forgot to animate the appearance of Edsgar. So two fellows, Dekker and Dijkstra, showed that it can, as long as the computer system is sequentially consistent.

And so I'm not going to give their algorithm, which is a little bit complicated. I'm going to give I what I think is boiled down to the most simple and elegant version of that uses their idea, and it's due to Peterson. And for the life of me, I have not been able to find a picture of Peterson. Otherwise, I'd show you what Peterson looks like.

So here's Peterson's algorithm. And I'm going to model it with Alice and Bob. They have a shared widget. And what Alice wants to do to the widget is to frob it. And Bob wants to borf it. So they're going to frob and borf it.

But we don't want them to be frobbing and borfing at the same time, naturally. You don't frob and borf widgets at the same time. So they're mutually exclusive.

So here's Peterson's algorithm. So we have widget x. So I'm just going to read through the

code here. And I have a Boolean variable called wants. I have an A_wants and a B_wants.

A means Alice wants to frob the widget. B_wants means that Bob wants to borf the widget. And we're also going to have a variable that has two values, A or B, for whose turn it is. And so we start out with that code, and then we fork the two Alice and Bob branches of our program to execute concurrently.

And what Alice does is she says, I want it. She sets A_wants to true. And I set the turn to be Bob's turn. And then the next loop has an empty body, notice. It's just a while with a semicolon. That's an empty body. It's just going to sit there spinning.

It's going to say, while B wants it, Bob wants it, and it's Bob's turn, I'm going to just wait. And if it turns out that either Bob does not want it or it's not Bob's turn, then that's going to free Alice to go into the critical section and frob x. And then when she's done she says, I don't want it anymore.

And if you look at Bob's code, it's exactly the same thing. And when we're done with this code, we're going to then loop to do it again, because they just want to keep frobbing and borfing until their eyes turn blue or red, whatever color eyes they have there.

Yeah, question?

I didn't explain why this works yet. I'm going to explain why it works.

**STUDENT:** OK.

**CHARLES LEISERSON:** You're going to ask why it works?

**STUDENT:** I was going to ask why those aren't locks.

**CHARLES LEISERSON:** Why are they not locks?

**STUDENT:** [INAUDIBLE]

**CHARLES LEISERSON:** Well, a lock says that if you can acquire it, then you stop the other person from acquiring it. There's no locking here, there's no waiting. We're implementing a mutual exclusion region. But a lock has a particular span-- it's got an acquire and a release.

So when you say A wants to be true, I haven't acquired the lock at that point, have I? Or if I set the turn to be the other character, I haven't acquired a lock. Indeed, I then do some testing and so forth and hopefully end up with mutual exclusion, which is effectively what locking does. But this is a different way of getting you there. It's only using loads and stores.

With a lock, there's an atomic-- I got the lock. And if it wasn't available, I didn't get the lock. Then I wait.

So let's discuss, let's figure out what's going on. And I'm going to do it two ways. First, I'm going to do the intuition, and then I'm going to show you how you reason through it with a happens before relation.

Question?

**STUDENT:** No.

**CHARLES LEISERSON:** No, OK. Good. Not good that there's no questions. It's good if there are questions. But good we'll move on.

So here's the idea. Suppose Alice and Bob dropped both tried to enter the critical section. And we have sequential consistency. So we can talk about who did things in what order. So whoever is the last one to write to the variable turn, that one's not going to enter. And the other one will enter.

And then if Alice tries to enter the section, then she progresses because at that point she knows that B_wants is false. And if only Bob tries to enter it, then he's going to go because he's going to see that A_wants is false. Does that makes sense? So only one of them is going to be in there at a time.

It's also the case that you want to verify that if you want to enter, you can enter. Because otherwise, a very simple protocol would be not to bother looking at things but just take turns. It's Alice's turn, it's Bob's turn, it's Alice's turn, it's Bob turn.

And we don't want a solution like that because if Bob doesn't want a turn, Alice can't go. She can go once, and then she's stuck. Whereas we want to be able to have somebody, if they're the only one who wants to go to execute the critical section, Alice can frob, frob, frob, frob, frob. Or Bob can borf, borf, borf, borf, borf. We don't want to force them to go if they don't need to go.

So the intuition is that only one of them is going to get in there because you need the other one either to say you want to go in, or else their value for wants is going to be 0. And it's going to be false and you're going to go through anyway. But this is not a good argument, because this is handwaving.

We're at MIT, right, so we can do proofs. And this proof isn't so hard. But I want to show it to you because it may be different from other proofs that you've seen.

So here's the theorem. Peterson's algorithm achieves mutual exclusion on the critical section. The setup for the proof is, assume for the purposes of contradiction that both Alice and Bob find themselves in the critical section together. And now we're going to look at the series of instructions that got us there, and then argue there must be a contradiction. That's the idea.

And so let's consider the most recent time that each of them executed the code before entering the critical section. So we're not interested in what happened long ago. What's the very, very last pieces of code as they entered the critical section? And we'll derive a contradiction.

So here we go. So without loss of generality, let's assume that Bob-- we have some linear order. And to execute, noticed a B in the critical section, Alice and Bob both had to set the variable turn. So one of them had to do it first. I'm going assume without loss of generality that it was Bob because I can otherwise make exactly the same argument for Alice. So let's assume that Bob is the last one to write to turn.

So therefore, if Bob was the last one, that means that Alice writing to turn, so she got in there so she wrote to turn. So her writing B to turn preceded Bob writing A to turn. So we have that happens before relationship.

Everybody with me? Do you understand the notation I'm using and the happens before relationship?

Now Alice's program order says that true to A_wants comes before her writing turn equals B. That's just program order. So we have that.

And similarly, we have Bob's program order. And Bob's program order says, well, I wrote turn to A. So Bob wrote, turn equals A. And then Bob, in this case I'm going to do Bob read A_wants. And then he reads turn.

So the second instruction here, up here, so this is a conditional and. So we basically are doing this. And then if that's true, then we do this. So this turn equals equals A. That's reading turn and checking if it's A happens after A_wants. So that's why I get these three things in order.

Does that makes sense? Any question about that? Is that good?

So I've established these two chains. So I actually have three chains here that I'm now going to combine. Let's see. So what's happening is let me look to see what's the order of everything that happens.

So the earliest thing that happens is that Alice wants to be true because-- where's that? So Alice wants is true is, yes, is coming before. That's the earliest thing that's happening here.

So that instruction is basically this-- Alice wants is true, it comes before the A turn equals B. That comes before the A turn equals B. So it comes before the write turn equals A, write B turn equals A. And then B turn equals A.

So do you see the chain we've established? You see the chain? Yeah, yeah. OK, good. |

So it says A_wants is first. A_wants equals true is first. Then we have the turn equals B. That's all from the second line here. That's from this line here.

What's next? Which instruction is next? So turn equals A. That comes from the top line there. What's next?

**STUDENT:** B [INAUDIBLE].

**CHARLES LEISERSON:** So I read B. Bob reads A_wants. And then finally, Bob reads turn at A.

So this is all based on just the interleaving and the fact that if you saw that we have the program order and that Bob was the last to write. That's all we're using.

And so why is that a contradiction? Well, we know what the linear order is. We know that when Bob read, what did Bob read? What did Bob read when he read A_wants in step 4?

He read the last value in that chain, the most recent value In that chain where it was stored to. And what was stored there? True. Good.

And then Bob read turn. And what was the most recent value stored to turn in that chain?

**STUDENT:** [INAUDIBLE] A.

**CHARLES LEISERSON:** So then what?

**STUDENT:** Bob gets stuck.

**CHARLES LEISERSON:** Bob, if that were in fact what he read in the while loop line, what should be happening now? He should be spinning there. He shouldn't be in the loop.

Bob didn't obey. His code did not obey the logic of the code. Bob should be spinning. That's the contradiction. Because we said Bob was in the loop.

Does that makes sense? Is that good?

So when you're confronted with synchronizing through memory, as this is called, you really got to write down the happens before things in order to be careful about reviewing things. I have seen in many, many cases engineers think they got it right by an informal argument. And in fact, for those people who have studied model checking-- anybody have any interaction with model checking?

What was the context?

**STUDENT:** 6.822.

**CHARLES LEISERSON:** Well, and were you studying protocols and so forth?

**STUDENT:** Yeah.

**CHARLES LEISERSON:** So in 6.822, what class is that?

**STUDENT:** Formal programming.

**CHARLES LEISERSON:** Formal programing. Good. So for things like network protocols and security protocols and for cache protocols in order to implement things like MSI and MESI protocols and so forth, these days they can't do it in their heads. They have programs that look at all the possible ways of

executing what's called model checking. And it's a great technology because it helps you figure out where the bugs are and essentially reason through this.

For simple things, you can reason it through. For larger things, you use the same kind of happens before analysis in those contexts in order to try to prove that your program is correct, that those protocols are correct. So for example, in all the computers you have in this room, every one of them, there was a model checker checking to make sure the cache analysis was done. And many of the security protocols that you're using as you access the web have all been through model checking. Good.

The other thing is it turns out that Peterson's algorithm guarantees starvation freedom. So while Bob wants to execute her critical session, Bob cannot execute his critical section twice in a row, and vise versa. So it's got the property that one of the things that you might worry about is Alice wants to go and then Bob goes a gazillion times, and Alice never gets to go.

Now that doesn't happen, as you can see, from the code because every time you go you set the turn to the other person. So if they do want to go, they get to go through. But proving that is a nice exercise. And it will warm you up to this kind of analysis, how you go about it. Yeah?

**STUDENT:** Does it work with another [INAUDIBLE]?

**CHARLES LEISERSON:** This one does not. And there has been wonderful studies of what does it take to get n things to work together. And this is one place where the locks have a big advantage because you can use a single lock to get the mutual exclusion among n things, so constant storage.

Whereas if you just use atomic read and atomic write, it turns out the storage grows. And there's been wonderful studies. Also, wonderful studies of these other operations, like compare and swap and so forth. And we'll do a little bit of that. We'll do a little bit of that.

So often, in order to get performance, you want to synchronize through memory. Not often, but occasionally you want to synchronize through memory to get performance. But then you have to be able to reason about it. And so the happens before sequential consistency, great tools for doing it.

The only problem with sequential consistency is what? Who is listening? Yeah?

**STUDENT:** It's not real.

**CHARLES LEISERSON:** It's not real. No, we have had machines historically that implemented sequential consistency. Today, no machines support sequential consistency, at least that I'm aware of. Instead they report what's called relaxed memory consistency. And let's take a look at what the motivation is for why you would want to make it a nightmare for programmers to synchronize through memory.

This has also led software people to say, never synchronize through memory. Why? Because it is so hard to get it correct. Because you don't even have sequential consistency at your back.

So today, no modern day processor implements sequential consistency. They all implement some form of relaxed consistency. And in this context, hardware actively reorders instructions, and compilers may reorder instructions too.

And that leads you not to have the property that the order of instructions that you specify in a processor is the same as the order that they get executed in. So you say do A and then B. The computer does B and then A.

So let's see instruction reordering. So I have on the left the order that the programmer specified, and the order on the right what the hardware did. Or it may have been that the compiler reordered them.

Now if you look, why might the hardware or compiler decide to reorder these instructions? What's going on in these instructions? You have to understand what these instructions are doing.

So in the first case, I'm doing a store and then a load. And in the second case, I have reversed the order to do the load first.

Now if you think about it, if you only had one thing going on, what's the impact here of this reordering? Is there any reason the compiler or somebody couldn't reorder these?

**STUDENT:** I think we reorder them is the reason that it affects the pipeline. If you have to store first, the write [INAUDIBLE] you have to [INAUDIBLE].

**CHARLES LEISERSON:** Yeah, in what way does it affect the pipeline?

**STUDENT:** That basically the load doesn't do anything in the [INAUDIBLE], whereas the store does.

**CHARLES LEISERSON:** I think you're on the right track. There's a higher level reason why you might want to put loads before stores. Why might you want to put loads? These are two instructions that normally if I only had one thread, reordering them would be perfectly fine.

Well, it's not necessarily perfectly fine. When might there be an issue? It's almost perfectly fine.

**STUDENT:** [INAUDIBLE]

**CHARLES LEISERSON:** If A was equal to B. But if A and B are different, than reordering them is just fine. If A and B are the same, if that's the same location, uh-oh, I can't reorder them because one is using the other.

So why might the hardware prefer to put the load earlier? Yeah?

**STUDENT:** There might be a later instruction which depends on B.

**CHARLES LEISERSON:** There might be a later instruction that depends on B. And so why would it put the load earlier?

**STUDENT:** So by doing the load earlier, the pipeline [INAUDIBLE] happens. Earlier on, [INAUDIBLE].

**CHARLES LEISERSON:** Yeah, you're basically covering over latency in a load. When I do a load, I have to wait for the result before I can use it. When I do a store, I don't have to wait for the result because it's not being used, I'm storing it.

And so therefore if I do loads earlier, If I have other work to do such as doing the store, then the instruction that needs the value of B doesn't have to necessarily wait as long. I've covered over some of the latency. And so the hardware will execute faster. So we've got higher performance by covering load latency.

Does that makes sense? It's helpful to know what's going on in the hardware here to reason about the software. This is a really great example of that lesson is what the compiler is doing there that it chooses to reorder.

And frankly, in the era before 2004 when we were in the era of what's called Dennard scaling, and we didn't worry. All our computers just had one processor, it didn't matter. Didn't have to

worry about these issues. These issues only come up for when you have more than one thing going on. Because if you're sharing these values, oops, I changed the order.

So let's see, so when is it safe in this context for the hardware compiler to perform this particular reordering? When can I do that? So there's actually two answers here. Or there's a combined answer.

So we've already talked about one of them. Yeah?

**STUDENT:** When A is not B.

**CHARLES LEISERSON:** Yeah, when A is not B. If A and B are equal, it's not safe to do. And what's the second constraint where it's safe to this reordering? Yeah, go ahead.

**STUDENT:** If A equals B, but if you have already one [INAUDIBLE].

**CHARLES LEISERSON:** Ooh, that's a nasty one. Yeah, I guess that's true. I guess that's true. But more generally when is it safe? That's a benign race in some sense, right? Yeah? Good. Good, that's a good one.

What's the other case that this is safe to do? Or what's the case where it's not safe? Same question. I just told you. When might this be safe? When is it safe to this reordering? I can't do it if A is equal to B. And also shouldn't do it when? Yeah?

**STUDENT:** [INAUDIBLE] value of A.

**CHARLES LEISERSON:** Yeah, but. Yeah?

**STUDENT:** [INAUDIBLE] if you have like when a processor is operating.

**CHARLES LEISERSON:** Yeah, if there's no concurrency. If there's no concurrency, it's fine. The problem is when there's concurrency.

So let's take a look at how the hardware does reordering so that we can understand what's going on. Because in a modern processor, there's concurrency all the time. And yet the compiler still wants to be able to cover overload latency, because usually it doesn't matter.

So you can view hardware as follows. So you have a processor on the left edge here, and you have a network that connects it to the memory system, a memory bus of some kind. Now it turns out that the processor can issue stores faster than the network can handle them. So the

processor can go store, store, store, store, store. But getting things into the memory system, that can take a while. Memory system is big and it's slow.

But the hardware is usually not doing store on every cycle. It's doing some other things, so there are bubbles in that instruction stream. And so what it does is it says, well, I'm going to let you issue it because I don't want to hold you up. So rather than being held up, let's just put them into a buffer. And as long as there's room in the buffer, I can issue them as fast as I need to. And then the memory system can suck them out of the buffer as it's going along.

And so in critical places where there's a bunch of stores, it stores them in the buffer if the memory system can't handle. On average, of course, it's going to go at whatever the bottleneck is on the left or the right. You can't go faster than whichever is the bottleneck-- usually the memory system. But we'd like to achieve that, and we don't want to have to stall every time we try to do two stores in a row, for example. By putting a little bit of a buffer, we can make it go faster.

Now if I try to do a load, that can stall the processor until it's satisfied. So whenever you do a load, if there's no more instructions to execute it, if the next instruction to execute requires the value that's being loaded, the processor has to stall until it gets that value. So they don't want the loads to go through the store buffer. I mean, one solution would be just put everything into the store buffer. In some sense you'd be OK, but now I haven't covered over my load latency.

So instead what they do is they do what's called a load bypass. They go directly to the memory system for the load, bypassing all the writes that you've done up to that point and fetch it so that you get it to the memory system and the load bypass takes priority over the store buffer.

But there's one problem with that hack, if you will. What's the problem with that hack? If I bypass the load, where could I run into trouble in terms of correctness? Yeah?

**STUDENT:** If one your stores is the thing you're trying to load.

**CHARLES LEISERSON:** If one of your stores is the thing you're trying to load. Exactly. And so what happens is, as the load bypass is going by, it does an associative check in the hardware. Is the value that I'm fetching one of the values that is in the store buffer? And if so, it responds out of the store buffer directly rather than going into the memory system. Makes sense?

So that's how the reordering happens within the machine. But by this token, a load can bypass a store to a different address. So this is how the hardware ends up reordering it, because the appearance is that the load occurred before the store occurred if you are looking at the memory from the point of view of the memory, and in particular the point of view of another processor that's accessing that memory.

So over here I said, store load. Over here it looks like he did, load store. And so that's why it doesn't satisfy sequential consistency. Yeah, question?

**STUDENT:** So that store bumper would be one for each processor?

**CHARLES LEISERSON:** Yeah, there's one for each processor. It's the way it gets things into the memory, right? So I'll tell you, computing would be so easy if we didn't worry about performance. Because if those guys didn't worry about performance, they'd do the correct thing. They'd just put them in in the right order.

It's because we care about performance that we make our lives hard for ourselves. And then we have these kludges to fix them up. So that's what's going on in the hardware, that's why things get reordered. Makes sense?

But it's not as if all bets are off. And in fact, x86 has a memory consistency model they call total store order. And here's the rules. So it's a weaker model. And some of it is kind of sequentially consistent type of thing. You're talking about what can be ordered.

So first of all, loads are never reordered with loads. Let me see here. Yeah, so you never reorder loads with loads. That's not OK.

Always, you can count on loads being seen by any external processor in the same order that you issued the loads within a given processor. So there is some rationale here.

Likewise, stores are not reordered with stores. That never happens. And then stores are not reordered with prior loads. So you never move a store earlier past a load. You wouldn't want to do that because generally it's the other direction you're covering over latency.

But in fact, they guarantee it doesn't happen. So you never move a store before a load. It's always move a load before a store.

And then in general, a load may be reordered with a prior store to a different location, but not

with a prior load to the same location. So this is what were just talking about, that A has to be not equal to B in order for it to be reordered.

And at the point that you're executing this, the hardware knows what the addresses are that are being loaded and stored and can tell, are they the same location or not. And so it knows whether or not it's able to do that. So the loads basically, you can move loads upwards. But you don't reorder them. And you only move it past a store if it's a store to a different address.

And so here we have a bunch of things. So this is basically weaker than sequential consistency. There are a bunch of other things.

So for example, if I just go back here for a second. The lock instructions respect a total order. The stores respect a total order. The lock instructions and memory ordering preserves what they call transitive visibility. In other words, causality, which is basically the happens-- says that the happens before a relation, you can treat as if it's a linear order. It's transitive as a binary relation.

So the main important ones are the ones at the beginning. But it's helpful to know that locks are not going to get reordered. If you have a lock instruction, they're never going to move it before things.

So here's the impact of reordering on Peterson's algorithm. Sorry, no, this is not Peterson's algorithm yet. This impact of reordering on this is that I may have written things in this order, but in fact they execute in something like this order. And therefore, the ordering, in this case, 2, 4, 1, 3 is going to produce the value 0, 0, which was exactly the value that you said couldn't possibly appear.

Well, on these machines it can appear. And also let me say, so instruction reordering violates this sequential consistency. And by the way, this can happen. Not just in the hardware, this can happen in the compiler as well. The compiler can decide to reorder instructions.

It's like, oh my god, how can we be writing correct code at all right. But you've written some correct parallel code, and you didn't have to worry about this. So we'll talk about how we get there. Yeah?

**STUDENT:** Is the hardware geared to even reorder [INAUDIBLE]? Or [INAUDIBLE] it might happen?

**CHARLES** It might happen. No, there's no requirement that it move things earlier.

**LEISERSON:**

**STUDENT:** Why is it not always [INAUDIBLE]?

**CHARLES LEISERSON:** It may be that there isn't enough register space. Because as you move things earlier, you're going to have to hold the values longer before you're using them. Yeah?

**STUDENT:** In the previous slide, [INAUDIBLE] load 3 [INAUDIBLE] also.

**CHARLES LEISERSON:** That load 3 in the previous-- I'm sorry, I'm not following.

**STUDENT:** In the previous slide.

**CHARLES LEISERSON:** Oh, the previous slide, not this slide. This one?

**STUDENT:** Yeah.

**CHARLES LEISERSON:** OK.

**STUDENT:** So [INAUDIBLE] load 3 [INAUDIBLE].

**CHARLES LEISERSON:** Well, I had said there's some things that I said we're no good, right? So here it was, what did I do? I moved the loads earlier in that example. But there were some earlier ones. Are you talking about even earlier than that?

**STUDENT:** Yeah, this one.

**CHARLES LEISERSON:** Oh, this one, OK.

**STUDENT:** So, load 3 can come before store [INAUDIBLE].

**CHARLES LEISERSON:** So let's see. So this is the original thing. Store 3 is before store 4, and load 3 and load 4 are afterwards, right? So the stores have to be in the same order and the loads have to be in the same order. But the loads can go before the stores if they're to a different address. So in this case, we moved load 3 up two, and we moved load 4 up one. We could have maybe move load 4 up before store 3, but maybe they were to the same address.

**STUDENT:** OK, so load 3's store doesn't mean that they're from the same address?

**CHARLES LEISERSON:** No, no, this is abstract. You got it? OK.

So this is why things can get reordering. And in that case, we can end up with a reordering that gives us something that we don't expect when we're synchronizing through memory. Never write non-deterministic code, because you deal with this stuff-- unless you have to. Unfortunately, sometimes, it's not fast enough otherwise.

Now let's go back and look at Peterson's algorithm and what can go wrong with Peterson's algorithm. So what reordering might happen here that would completely screw up Peterson's algorithm? A hint, we're looking for a load that might happen before a store. What load would be really bad to happen before a store? Yeah?

**STUDENT:** If you load turn to [INAUDIBLE] before the store turn [INAUDIBLE].

**CHARLES LEISERSON:** You load turn earlier. Maybe. Let me think, that's not the one I chose, but maybe that could be right. Well, you can't move it before the store to turn.

**STUDENT:** All right.

**CHARLES LEISERSON:** OK, yeah?

**STUDENT:** Maybe Alice loads B_wants to early?

**CHARLES LEISERSON:** Yeah, if Alice loads B_wants to early, and if they both do, then they could be reordered before the store of A_wants and B_wants, because that's a load and B_wants-- well, Alice isn't touching B_wants so why can't it just move it earlier. Those are not the same locations. So suppose it reorders those, now what happens?

**STUDENT:** So [INAUDIBLE] B_wants [INAUDIBLE] too early?

**CHARLES LEISERSON:** Yeah, it would be false too early, right?

**STUDENT:** And the same with A_wants.

**CHARLES LEISERSON:** And the same with A. And now they discover they're in this critical section together. And if there's one thing, we don't want Alice and Bob in the same critical section. Does that makes sense?

So you've got this problem. There's reordering going on. And, yikes, how could you possibly write any parallel code and any concurrent code? Well, they say, well, we'll put in a kludge. They introduce some new instructions. And this instruction is called a memory fence.

So don't get me wrong. They need to do stuff like this. There is an argument to say they should still build machines with sequential consistency because it's been done in the past. It is hard work for the hardware designers to do that. And so as long as the software people say, well, we can handle weak consistency models, [INAUDIBLE] says, OK, your problem.

So Mark Hill, who's a professor at University of Wisconsin, has some wonderful essays saying why he thinks that parallel machines should support sequential consistency, and that the complaints of people not having it supported, that those people they could support it if they really wanted to. And I tend to be persuaded by him. He's a very good thinker, in my opinion.

But in any case, so what we have-- yeah, question?

**STUDENT:** How much of a difference does it make to sacrifice?

**CHARLES LEISERSON:** So he talks about this and what he thinks the differences is, but it's apples and oranges. Because sometimes part of it is what's the price of having bugs in your code. Because that's what happens is programmers can't deal with this. And so we end up with bugs in our code. But they can reason about sequential consistency. It's hard, but they can reason about it. When you start having relaxed memory consistency, very tricky.

So let's talk about what the solutions are. And his argument is that the performance doesn't have to be that bad. There was a series of machines made by a company called Silicon Graphics, which were all sequentially consistent. Parallel machines, all sequentially consistent. And they were fine.

But they got killed in the market because they couldn't implement processors as well as Intel does. And so they ended up getting killed in the market and getting bought out, and so forth. And now their people are all over, and the people who were at Silicon Graphics, many of them really understand parallel computing well, the hardware aspects of it.

So a memory fence is a hardware action that forces an ordering constraint between the instructions before and after the fence. So the idea is, you can put a memory fence in there and now that memory fence can't be reordered with things around it. It maintains its relative ordering site to other things. And that way you can prevent.

So one way you could make any code be sequentially consistent is to put a memory fence between every instruction. Not very practical, but there's a subset of those that actually would matter. So the idea is to put in just the run one.

You can issue them explicitly as an instruction. In the x86, it's called the mfence instruction. Or it can be performed implicitly, so there are other things like locking, exchanging, and other synchronizing instructions. They implicitly have a memory fence.

Now the compiler that we're using implements a memory fence via the function atomic_thread_fence, which is defined in the C header file stdatomic.h. And you can take a look at the reference material on that to understand a little bit more about that.

The typical cost on most machines is comparable to that of an L2 cache access. Now one of the things that is nice to see is happening is they are bringing that down. They're making that cheaper. But it's interesting that Intel had one processor where the memory fence was actually slower than the lock instruction.

And you say, wait a minute, the lock instruction has an implicit memory fence in it. I mean, you've got a memory fence in the lock instruction. How could the memory fence be slower?

So I don't know exactly how this happens, but here's my theory. So you've got these engineering teams that are designing the next processor. And they of course want it to go fast.

So how do they know whether it's going to go fast? They have a bunch of benchmark codes and that they discover, well, now that we're getting the age of parallelism, all these parallel codes, they're using locking.

So they look and they say, OK, we're going to put our best engineer on making locks go fast. And then they see that, well, there's some other codes that maybe go slow because they've got fences. But there aren't too many codes that just need fences, explicit fences. In fact, most of them use [INAUDIBLE].

So they put their junior engineer on the fence code, not recognizing that, hey, the left hand

and the right hand should know what each other is doing. And so anyway, you get an anomaly like that where it turned out that it was actually fastest-- we discovered as we're implementing the silk runtime-- to do a fence by just doing a lock on a location that we didn't care about the lock. We just did a lock instruction. And that actually went faster than the fence instruction. Weird. But these systems are all built by humans.

So if we have this code and we want to restore consistency, where might we put a memory fence? Yeah?

**STUDENT:** After setting the turn?

**CHARLES LEISERSON:** After setting turn. You mean like that?

**STUDENT:** Yeah.

**CHARLES LEISERSON:** Yeah. OK, so that you can't end up loading it before it's stored too. And that kind of works, sort of. You also have to make sure that the compiler doesn't screw you over.

And the reason the compiler might screw you over is that it looks at B_wants and turn B, it says, oh, I'm in a loop here. So let me load the value and keep using the value over. And I don't see anybody using this value.

Right, so it loads the value. And now it just keeps checking the value. The value has changed on the outside, but it's stored that in a register so that that loop will go really fast. And so it goes really fast, and you're spinning and you're dead in the water.

So in addition to the memory fence, you must declare variables as volatile to prevent the compiler from optimizing them away. When you declare something as volatile, you say, even if you read it, if the compiler reads it. When it reads it a second time, it's still got to read it a second time from memory. It cannot assume that the value is going to be stable. You're saying it may change outside.

And then you also, it turns out, may need compiler fences around frob and borf to prevent them reordering some of frob and borf because that stuff can also sometimes get moved outside the loop, the actual code in frob and borf, because it wants to, it says, oh. It doesn't realize always that there's no what's going on.

So the C11 language standard defines its own weak memory model. And you can declare things as atomic, and there are a bunch of things there. And here's a reference where you can take a look at the atomic stuff that's available if you want to do this dangerous programming.

In general for implementing general mutexes, if you're going to use only load and store, there's a very nice theorem by Burns and Lynch-- this is Nancy Lynch who's on the faculty here-- that says any n-thread deadlock-free mutual exclusion algorithm using only load and store requires order n space-- the space is linear. So this answers the question that I had answered orally before.

And then it turns out that if you want an n-thread deadlock-free mutual exclusion algorithm, you actually have to use some kind of expensive operation, such as a memory fence or an atomic compare-and-swap. So in some sense, hardware designers are justified when they implement special operations to support animosity, as opposed to just doing using these clever algorithms. Those algorithms are really at some level of theoretical interest.

So let's take a look at one of these special instructions. And the one I picked is compare-and-swap because it's the one that's probably most available. There are others like test-and-set, and so forth. And so when you do lock-free algorithms, when you want to build algorithms that are lock free, and we'll talk about why you might want to do lock-free algorithms, there's loads and store, and then there's this CAS instruction, Compare-and-Swap.

In stdatomic.h, it is called atomic_compare_exchange_strong. And it can operate on various integer types. It cannot compare and swap floating point numbers. It can only compare and swap integers, and sometimes that's a pain.

And so here's the definition of the CAS instruction. Basically, what it does is it has an address. And then it has two values, the old value and the new value. And what it does is it checks to see, is the value that is in that memory location the same as the old value. And if it is, it sets it to the new value and says, I succeeded. And otherwise, it says I failed.

So it swaps it if the value that I'm holding, the old value, is the same as what's in there. So I can read the value, if I want, then do whatever I want to do. And then before I update it, I can say, update it only if the value hasn't changed. And that's what the compare and swap does. Does that makes sense?

And it does that all atomically,. And there's an implicit fence in there so things don't get

reordered around it. It's all done as one. The hardware ensures that nothing can interfere in the middle of this. It's actually comparing the old value to what's in there, and swapping in the new, all as one operation. Or it says, nope, the value changed, therefore it just returned false, and the value didn't get updated.

So it turns out that you can do an n-thread deadlock-free mutual exclusion algorithm with compare-and-swap using only constant space. And here's the way you do it. And this is basically just the space for the new text itself. So you take a look at the lock instruction, and what you do is you spin, which is to say you block, until you finally get the value true. So you're trying to swap in true.

So true says that somebody holds the lock. I say the old value was false. If it's true, then the swap doesn't succeed and you just keep spinning. And then otherwise, you swap in the value and now you're ready to go. And to unlock it, you just have to set it to false. Question?

**STUDENT:** Why does it de-reference the pointer in the lock?

**CHARLES LEISERSON:** Why does it de-reference the pointer? Because you're saying, what memory location are you pointing to. You're interested in comparing with the value in that location. So it is a memory operation. So I'm naming the memory location. I'm saying, if the value is false, swap in the value true and return true.

And if it's true, then don't do anything and tell me that you didn't succeed, in which case in this loop it'll just keep trying again and again and again. It's a spinning lock. Question?

**STUDENT:** [INAUDIBLE] when you saying that you're [INAUDIBLE] the value at that address before passing it into CAS.

Yeah, there shouldn't be a pointer de-reference after [INAUDIBLE].

**CHARLES LEISERSON:** Oh, you're right. A bug. Gotcha, yep, gotcha, I'll fix it.

So let's take a look at a way that you might want to use CAS. So here's a summing problem. So suppose I want to compute on some variable of type x. And I've got an array that's-- what is that-- that's a million elements long. And what I'm going to do is basically run through my array in parallel and accumulate things into the result.

And so this is actually incorrect code. Why is this incorrect code? Yeah?

**STUDENT:** Extra like a floating point taken [INAUDIBLE] and so forth?

**CHARLES LEISERSON:** Maybe, let's assume we have fast math. Yeah?

**STUDENT:** You have multiple transfer and updated results at the same time?

**CHARLES LEISERSON:** Which means what?

**STUDENT:** Which means you have a race.

**CHARLES LEISERSON:** You have a race. You have a race. Everybody is trying to update result. You've got a gazillion strands in parallel all trying to pound on updating result.

So one way you could solve this is with mutual exclusion. So I introduce a mutex L. And I lock before I update the result, and then I unlock. Why did I put the computation on my array of i? Why did I put that outside the lock?

**STUDENT:** It's [INAUDIBLE] function is very expensive. That way, you're only locking the--

**CHARLES LEISERSON:** Yeah, whenever you lock, you want to lock for the minimum time possible. Because otherwise you're locking everybody else out from doing anything. So that was a smart thing in that particular code.

So that's the typical locking solution. But look at what might happen. What if the operating system decides to swap out a loop iteration just after it acquires mutext? As you go down, it says lock. You get the lock, and now the operating says, oops, your time quantum is up. Somebody else comes in and starts to compute. What's going to happen now? What's the problem that you might observe? Yeah?

**STUDENT:** [INAUDIBLE] if they're [INAUDIBLE] computation [INAUDIBLE] have to [INAUDIBLE].

**CHARLES LEISERSON:** Yeah, everybody's going to basically just sit there waiting to acquire the lock because the strand that has the lock is not making progress, because it's sitting on the side. It's been these scheduled. That's bad, generally. You'd like to think that everybody who's running could continue to run. Yeah?

**STUDENT:** Well, I guess under what circumstances might be useful for a processor to have this running on multi-threads instead of multiple processors simultaneously?

**CHARLES LEISERSON:** No, so this the multiple threads are running on multiple processors, right?

**STUDENT:** What do you mean by the time quantum?

**CHARLES LEISERSON:** So one of these guys says, so I'm running a thread, and that thread's time quantum expires.

**STUDENT:** Oh, that processor's multiple threads.

**CHARLES LEISERSON:** Right.

**STUDENT:** OK.

**CHARLES LEISERSON:** So I've got a whole bunch of processors with a thread on each, let's say. And I've got a bunch of threads. The operating system has several threads that are standing by waiting for their turn. And one of them grabs the lock and then the scheduler comes in and says, oops, I'm going to take you off, put somebody else in.

But meanwhile, everybody else is there trying to make progress. And this guy is holding the key to going forward. You thought you were only grabbing the lock for a short period of time. But instead, the operating system came in and made you take a long time.

So this is the kind of system issue that you get into when you start using things like locks. So all the other loop iterations have to wait. So it doesn't matter if-- yeah, question?

**STUDENT:** How does the [INAUDIBLE] reducer have [INAUDIBLE]?

**CHARLES LEISERSON:** So that's one solution to this, yep.

**STUDENT:** How does it do it?

**CHARLES LEISERSON:** How does it do it? We have the paper online. I had the things for explaining how reducers work. And there's too much stuff. I always have way more stuff to talk about than I ever get a chance to talk about. So that was one where I said, OK, yeah.

**STUDENT:** OK.

**CHARLES LEISERSON:** OK. So all we want to do is atomically execute a load of x followed by a store of x. So instead of doing it with locks, I can use CAS to do the same thing, and I'll get much better properties. So here's the CAS solution.

So what I do is I also compute a temp, and then I have these variables old and new. I store the old result. And then I add the temporary result that I've computed to the old to get the new value.

And if it turns out that the old value is exactly the same as it used to be, then I can swap in the new value, which includes that increment. And if not, then I go back and I do it again. I once again load, add, and try to swap in again.

And so now what happens if the operating system swaps out a loop iteration? Yeah?

**STUDENT:** It's OK because whenever this is put back on, then you know it'll be different.

**CHARLES LEISERSON:** It'll be new values, it'll ignore it, and all the other guys can just keep going. So that's one of the great advantages of lock-free algorithms. And I have in here several other lock-free algorithms.

The thing you should pay attention in here is to what's called the ABA problem, which is an anomaly with compare-and-swap that you can get into. This is a situation where you think you're using compare-and-swap, you say is it the old value. It turns out that the value is the same, but other people have come in and done stuff but happened to restore the same value. But you assume it's the same situation, even though the situation has changed but the value is the same. That's called the ABA problem. So you can take a look at it in here.

So the main thing for all this stuff is, this is really interesting stuff. Professor Nir Shavit teaches a class where this is the content of the class for the semester is all these really dangerous algorithms. And so I encourage you, if you're interested in that.

The world needs more people who understand these kinds of algorithms. And it needs to find ways to help people program fast where people don't have to know this stuff, because this is really tricky stuff. So we need both-- both to make it so that we have people who are talented in this way, and also that we don't need their talents.

OK, thanks, everybody.