CHARLES LEISERSON: So today we're going to do some really cool stuff having to do with nondeterministic parallel programming. This is where the course starts to get hard. Because nondeterminism is really nasty. We'll talk about it a little bit. It's really nasty.

Parallel computing, as you know, is pretty easy, right? It's just work and span. Easy stuff, right? It makes sense. You can measure these things, can learn some skills around them, and so forth. But nondeterminism is nasty, really nasty.

So first let's talk about what we mean by determinism. So we say that a program is deterministic on a given input if every memory location is updated with a sequence-- the same sequence of values in every execution. So the program always behaves the same. And you may end up-- if it's a parallel program having different memory locations updated in different orders-- I may do A and then B, versus updating B and then A-- but if I look at a single memory location, A say, I'm always updating A with the same sequence of values.

There are lots of definitions of determinism. This is not the only one. There are some where people say, well, it only matters if the output is always the same. And there are others where you say not only does it have to be the same but every write to a location has to be in the same order globally. That turns out to be actually pretty hard, because if you have parallel computing you're not going to get them all updated the same unless you only have one processor executing instructions. And so we'll talk about this. We'll talk a little bit more about this kind of thing.

So why-- what's the big advantage of deterministic programs? Why should we care whether it's deterministic or nondeterministic? Sure.

AUDIENCE: It's repeatable.

CHARLES LEISERSON: It's repeatable. OK. So what?

AUDIENCE: [INAUDIBLE] a lot of programs [INAUDIBLE].

CHARLES LEISERSON: Why is that?

AUDIENCE: [INAUDIBLE] like a-- Why? Because sometimes that's what you want.

CHARLES LEISERSON: Because sometimes that's what you want. OK. That doesn't-- so if-- I mean, there's a lot of things I might sometimes want. Why is that important to want that? Yes.

AUDIENCE: Because consistency makes it easier to debug source code.

CHARLES LEISERSON: Yes. Makes it easier to debug. That's probably the number one reason, debugging. If it does the same thing every time, then if you have a bug, you can run it again. You expect to see the bug again. So every time you run through, hey, I get the same bug. But if it's nondeterministic, I get a bug, and now I go to look for it and the bug is nowhere to be found. Makes debugging a lot harder.

There are other reasons for wanting repeatability, so your answer is actually a broader correct answer. But the big advantage is in the specific application of repeatability to debugging. So here's the golden rule of parallel programming. Never write nondeterministic parallel programs. They can exhibit anomalous behaviors and it's hard to debug them. So never ever write nondeterministic programs.

Unfortunately, this is one of these things that is kind of hard in practice to do. So why might you want to write a nondeterministic program even though-- even when famous masters in the area of performance engineering, with highly credentialed-- numerous awards and so forth, tell you you shouldn't write nondeterministic programs? Why might you want to do it anyway? Yes.

AUDIENCE: You get better performance.

CHARLES LEISERSON: Yes. You might get better performance. That's one of the big ones. That's one of the big ones. And sometimes you can't. The nature of the problem is maybe that it's not deterministic. You may have asynchronous inputs coming in and so forth.

So this is the golden rule. We also have a silver rule. Silver rule says never write nondeterministic parallel programs, but if you must always devise a test strategy to manage

the nondeterminism. So this gets into you better have some way of handling how you're going to tell what's going on if you have a bug.

So what are some of the typical test strategies that you could use that would manage the nondeterminism? So imagine you've got a parallel program and it's got races in it and so forth, and it's operating nondeterministically. What-- and that's OK if everything's going right. How would you-- you find a bug in the program. How are you-- what are you going to do? What kinds of ideas do you have? Yes.

AUDIENCE: You could temporarily remove the nondeterminism.

CHARLES LEISERSON: Yes. You could turn off the nondeterminism. You put a switch in there that says, well, I know the source of this nondeterministic behavior. Let me do that. Let me give you an example of that.

For security reasons these days, when you allocate memory, it's allocated to different locations on different runs of the program. It's allocated in random places. They want to randomize the addresses when you call malloc. That means that you can end up with different behaviors from run to run, and that can compromise your performance. But it turns out that there is a compiler switch, and if you run it in debug mode it will always deliver the results of malloc in deterministic locations, where the locations of the things you're mallocing are repeatable.

So that's good because they're supported. They said, yes, we have to randomize for security reasons so that people can't deterministically exploit buffer overflow errors, for example, but I don't want to have to do that every time. So I don't want to randomize every time I run. I want to have the option of making it so that that randomization is turned off. So that's a good one.

What's another one that can be done? You're full of good ideas. Let's try somebody else for now. But I like that, I like that. What are some other ideas? What else can you do to handle nondeterminism? You got a program and it's-- yes, yes, yes.

AUDIENCE: If you use random numbers, use the same seed.

CHARLES LEISERSON: Yes. If you have random numbers, you use the same seed. In some sense that's kind of the same thing if you're turning off nondeterminism. But that's a great one. There are other places. For example, if you read-- if you do get time of day for something in your program for something, you could have an option where it will put in a particular fixed value there so you

can make sure that it doesn't-- even a serial program isn't nondeterministic. So that's good, but I also consider that to be-- it's another great example of turning off and on determinism. What other things can you do? Yes.

AUDIENCE: You could record the randomized outputs or inputs to determine correctness.

CHARLES LEISERSON: Yes. You can do record-replay for some things. Is that what you're saying? Is that what you mean? Or am I--

AUDIENCE: Maybe. [INAUDIBLE].

CHARLES LEISERSON: So record-replay says you run it through-- you can run it through with random numbers, but it's recording those things, so that when you run it again, instead of using the random numbers-- new random numbers, it uses the ones that you used to use. So that's the record-replay thing. Is that what you're saying, or are you saying something else? Yes, OK, good.

So that's using some tools. There are actually a lot of strategies. Let me just move on and answer. So another thing you can do is encapsulate the nondeterminism. So that's actually done in a Cilk runtime system already.

The runtime system is using a random scheduling strategy, but you don't see that it's random in the execution of your code if you don't-- if you have no race conditions in your code. It's encapsulated. So that the-- in the platform. So the platform is going to guarantee you deterministic results even though underneath the covers it's doing nondeterministic things.

You can also substitute a deterministic alternative. Sometimes there's a way of computing something that is nondeterministic, but in debug mode, ah, let me not use the nondeterministic one. And you can also use analysis tools, which can tell you things about your program and which you can control things. So there's a lot of things.

So whenever you have a nondeterministic program, you want to find some way of controlling it. Often, the nondeterminism is over in this corner but your bug is over in this corner. So if you can turn this thing off in some way, or encapsulate it, or otherwise control the nondeterminism over there, now you have a better chance of catching the stuff over here.

That's going to be particularly important in project 4 when we get to it, because that's going to be actually going to be doing nondeterministic programming for a game playing program. And one of the things is that the processors are, in this case, keeping the game positions together.

And so if one processor stores something into what's called a transposition table, which is essentially a big hash table of positions it's seen, another one can see that value and change its behavior. And so one of the things you want to be do is turn off transposition table so that you don't take advantage of that performance advantage, but now you can debug the search code, or you can debug the evaluation code, and so forth.

You can also do things like unit testing so you know whether or not a particular piece is correct that might have-- so that you can test this thing separately from the rest of your system which may have nondeterminism. Anyway, this is a major thing. So never write them. But if you have to, you always want to have some test strategy. And so for people who are not watching this video and who are not in class today, they are going to be sorely hampered by not knowing this lesson when they go into the fourth project.

So what we're going to do is now we're going to talk about how to do nondeterministic programming. So this is-- there's always some part of your code which has a skull and crossbones. Like you have this abstraction. It's beautiful, and you can design, et cetera. And then somewhere there's this really ugly thing that nobody should know, and you put the skull and crossbones on that, and only experts go in. Well, anyway, that's the barrier we're crossing here.

And we're going to start out by talking about something that you've probably seen in some of the other classes, mutual exclusion and atomicity. So I'm going to use the example of a hash table. So here's a typical hash table. It's got collisions resolved by chaining. So you have a bunch of linked lists. You hash to a particular slot in the table, and then you chase down the linked list to find the value.

And so, for example, if I'm going to insert x which has a key value of 81, what I do is figure out which slot I go to by hashing the key. And then in this case I made it be the last one so that the animations could be easier than if it were in the middle. So now what do I do is I make the pointer of x go to the first element of that list, and then I make the slot value now point to x. And that effectively, with a constant number of operations, inserts x into the hash table, and in particular into the linked list in the slot that it's supposed to be. This is all familiar, right?

So now what happens when you have multiple parallel instructions that are accessing the same locations? So here we have two threads, one inserting x and one inserting y. And x goes, it does its thing. It hashes to there, and it then sets the next pointer to be the-- to add

itself into the list.

And then there's this other thing going on in parallel which effectively says, oh, I'm going to hash. Oh, we're going to the same slot. It doesn't know that somebody is already there. And so then it decides it's going to put itself in as the first element of the list. And then it sets the value of y-- it sets the value of the slot to point to y.

And then along comes x, finishing off what it's doing, and it points the value to x. And you can see that we have a race bug here, a really nasty one because we've just destroyed the integrity of our system. We now have-- in particular, y is sort of floating, not in the list when it's supposed to be in the list.

So the standard solution to this is to make some of these instructions be atomic. And what that means is the rest of the system can never view them as being partially executed. So they either all have been executed or none of them have been executed at any point in time as far as the rest of the system is concerned. And the part of code that is within the atomic region is called the critical section. And, typically, a critical section of code is some place that should not be being executed by two things at the same time.

So the standard solution to atomicity is to use what's called a mutex lock, or a mutual exclusion lock. And it's basically an object with a lock and unlock member functions. And an attempt by a thread to lock an already locked mutex causes the thread to block-- that is, wait-- until the mutex is unlocked. So if somebody grabs the lock, somebody else grabs the lock and it's already taken, then they have to wait. And they sit there waiting until this guy says, yes, I'm going to release it.

So what we'll do now is we'll make each slot be a struct with a mutex L, and a pointer, head, to the slot context. So it's going to be the same data structure we had before but now I'm going to have not just the pointer from the slot but I'll also have a-- also have a lock in that position. And so the idea of-- in the code now is that before I access the lock-- before I access the list, I'm going to lock that list in the table by locking slot. Then I'll do the things that I need to do, and then I'll unlock it, and now anything else can go on.

Because what's happening is-- the reason we're getting into trouble is because we've got some sort of interleaving of operations. And our goal is to make sure that it's either doing this or doing this, and never this, to make sure that-- so that each thing, each piece of code, is restoring the invariant of correctness after it executes the pointer swaps. The invariance in this

case is that the elements are in a list. And so you want to restore that with each one. So mutexes-- this is one way you can use mutexes to implement atomicity.

So now let's just go back. Who has seen mutexes before? Is that pretty much everybody? Yes. OK, good. I hope that this is not brand new for too many of you. If it is brand new, that's great. But what I'm trying to do is make it-- so let's go back a little bit and recall in this class our discussion of determinacy races.

So, remember, a determinacy race occurs when you have two logically parallel instructions that access the same memory location and at least one of them performs a write. So mutex locks can guarantee that critical sections behave atomically, but the resulting code is inherently nondeterministic because you've got a-- we had a race bug there. We had two things trying to access the same slot. But that may be what I want. I want to have a shared hash table maybe for these things. So I want something where there is a race, but I just don't want to have the anomalies that arise. In this case, the race bug caused things, and I can solve that with atomicity.

If you have no determinacy races, it means that the program is deterministic on that input and that it always behaves the same. And remember also that if a deterministic race exists in an ostensibly deterministic program, then it guarantees to find a race. Now, if you put in mutexes, you still have a nondeterministic program. You still have a race. Because you have two things that are logically parallel that are both accessing the lock. That's a race. That's a determinacy race.

If you have two things, they're in parallel, they're both accessing the lock, that's a determinacy race. It may be a safe, correct one, but it is a determinacy race. And so any codes that use locks are nondeterministic by intention, and they're going to invalidate the Cilksan guarantee of finding those race bugs. So you will end up with races in your code if you're not careful.

And so this is one reason it's important to have some way of turning off nondeterminism to detect stuff. Because what you don't want is a whole rash of false positives saying, oh, you raced on gathering this lock. Nor do you want to ignore that and then discover that a race has popped up somewhere else.

Now, some people feel that-- so this is basically talking about having a data race. And a data race is similar to the definition of determinacy race, but it says that you have two logically parallel instructions and they don't hold locks in common. And then it's the same definition. If

they access the same memory location and one of them performs a write, then you have a-- then you have a data race bug.

But if they have the locks in common, if they both have acquired at least one lock that's the same, then you don't have a data race, because that means that you've now successfully protected the atomicity. But it is still nondeterministic and there is a determinacy race, just no data race. And that's the big distinction between data races and determinacy races. And on quiz 2, you better know the difference between data races and determinacy races, because they are different. So a program may have no determine-- may have no data races. That doesn't mean that it doesn't have a determinacy race. In fact, if it's got any locks, it probably has a determinacy race.

So one of the things is, if I have no data races, does that mean I have no bugs? Suppose I have no data races in my code. Does that mean I have no bugs? This is like an obvious answer just by quizmanship, right?

So what might happen? Think about it a little bit. What might happen? How could I have no data races and yet there still be a bug, even though-- I'm assuming it's a correct piece of code otherwise. In other words, when it runs serially or whatever, it's correct. How could I end up having a code-- no data races but still have a bug?

AUDIENCE: It's still nondeterministic [INAUDIBLE].

CHARLES LEISERSON: Yes, but that doesn't mean it's bad, right?

AUDIENCE: Well, you said that it runs correctly serially.

CHARLES LEISERSON: Yes.

AUDIENCE: So the order that things are put in or generated might still be--

CHARLES LEISERSON: Might still be different, yes.

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: OK. Yes. Let me give you an example which is more to the point. Here is a way of making sure that I have no data race, which is I lock before I follow the table slot value. Then I unlock, and I lock again and then I set the value. So I haven't prevented the atomicity. Right now I've got an atomicity violation, but I have no data races, because I never have two things-- any two things that are going to access things at the same time is protected by the lock. But it didn't solve my atomicity, so there's a-- you can definitely have no data races, but that doesn't mean you have no bugs.

But, usually, what happens is, if you have no data races, then usually the programmer actually got this code right. It's one of these things where demonstrating no data races is in fact a very positive thing in your code. It doesn't mean the programmer did right. But most of the time, the reason they're putting in the locks is to provide atomicity for something, and they usually get it right.

They don't always get it right. In fact, Java, for example, had a very famous bug early on in the way that it specified locking such that the-- you could look at the length of a string and then modify it, and then you would end up with a race bug because somebody else could swoop in in between. So they thought they were providing atomicity and they didn't.

So there's another set of issues here having to do with benign races. Now, there's some people who argue that no races are-- no determinacy races are benign. And they make academic statements that I find quite compelling, actually, what they say, about races and whether races are benign. But, nevertheless, the literature also continues to use the term benign race for this kind of example.

So suppose we want to identify what is the set of digits that occurred in some array. So here's an array with a bunch of values in it, each one being a digit from 0 to 9. So I could write a little piece of code that runs through a digits array of length 10 and sets the number of digits I've seen so far of each value to be 0. And now I go through-- and I'm going to do this in parallel-- and I'm going to set, every time I see a value A of i-- suppose A of i is 3-- I set the location of A3 to be 1. And, otherwise, and now-- otherwise, it's 0 because that's what I had it before.

So here's the kind of thing I have. So, for example, I can have both of those 6's-- or in parallel, we're going to access the location 6 to set it to 1. But they're both setting it to 1. It doesn't really matter what order they do it in. You're going to get the same value there, 1. And so there's a race. Maybe we don't too much care about that race, because they're both setting

the same value. We're not going to get an incorrect value.

Well, not exactly. We might get it on some architecture. On the Intel architectures, you won't get an incorrect value, on x86. But there are codes where the elements-- the array values are not set atomically. So, for example, on the MIPS architecture, in order to set a bite to be a particular value, you have to fetch the word, mask out, set the word, and then store it back in. Set the byte and then store it back into the word.

And so if there are two guys who are basically operating on that same word location, they will have a race, even though in the code it looks like they're just setting bytes. Does that make sense? So nasty. Nasty bugs. That's why you should never do nondeterministic programming unless you have to.

So Cilksan allows you to turn off race detection for intentional races. So if you really meant there to be a race, as in this case, you can turn it off. This is dangerous but practical, it turns out. Usually you're not turning it off for-- because here's what can happen. You can turn it off and yet-- then there's something else which is using that same stuff, and now you're running Cilksan without having turned it off for exactly what your race might be.

There are better solutions. So in Intel's Cilk Screen, there's the notion of fake locks. We just have not yet implemented it in the open Cilk compiler and in Cilksan. We'll eventually get to doing that. And then people who take this class in the future will have an easier time with that, because we'll be able to check for that as well. So any questions about these notions?

So you can see the notions of races can get quite hairy and make it quite difficult to do your debugging, and in fact even can confound your tools that are supposed to be helping you to get correct code. All in the name of performance. But we like performance. Any questions? Yes.

AUDIENCE: So I don't really understand how some architectures can cause some error in race conditions.

CHARLES LEISERSON: Yes. So how can some architectures cause some error? So here's the thing, is that if I have a, let's say, a byte array, it may be that this is stored as a set of let's say four-byte words. And so although you may write that A of 0 gets 1, what it does is it says, let me fetch these four values, because there is no byte set instruction on some architectures. It can only set, in this case, 32-bit words.

So it fetches the values. It then-- into a register. It then sets the value in the register by

masking. So it doesn't set the other things here. And then it stores it back so that it has a 1 here. But what if somebody, at the same time, is storing into this location? They will fetch it into their own register, set their byte, mask it, et cetera. And now my writeback is going to-- we're going to have a lost update in the writebacks. Does that make sense?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: OK. Good. Very good question. Yes, I know. I went through that orally a little bit quicker than maybe I should have. Great.

So let's talk a little bit about implementation. I always like to take things down one level below what you necessarily need to know in order to do things. But it's helpful to sort of see how these things are implemented, because then that gives you a better sense at a higher level what your capabilities are and how things are actually working underneath.

So let's talk about mutexes. So here, first of all, understand there are lots of different mutexes. If you look at an operating system, they may have a half a dozen or more different mutexes, different locks that can provide mutual exclusion, or parameters that can be set for what kind of mutexes.

So the first basic difference in most things is whether the mutex is yielding or spinning. So a yielding mutex returns control to the operating system when it blocks. When a program tries to get-- when it tries to get access, when a thread tries to get access to a given lock, if it is blocked, it doesn't just sit there and keep-- and spinning, where you're basically-- spinning means I just sit there checking it and checking it and checking it and checking it.

Instead what it does is it says, oh, I'm doing useless work here. Let me go and return control to the operating system. Maybe there's another thread that can run at the same time, and therefore I'll give-- by switching myself out, by yielding my scheduling quantum, I will get better efficiency overall, because somebody-- some other thread that is capable of running can run at that point. So is that a clear distinction between spinning and yielding?

Another one is whether the mutex is reentrant or nonreentrant. A reentrant mutex allows a thread that is already holding a lock to acquire it again. A nonreentrant one deadlocks if the thread attempts to require a mutex it already holds. So I grab a lock, and now I go to a piece of code that says grab that lock.

So very simple. I can check to see whether I have-- if I want to be reentrant, I can check, do I have that lock already? And if I do, then I don't actually have to acquire it. I just keep going. But that's extra overhead. It's faster for me to have a nonreentrant lock, where I just simply grab the lock, and if somebody has got it, including me, then it's a deadlock. But now if there's the possibility that I could reacquire a lock, then that might not be safe. You have to worry about-- the program has to worry about that now. Is that clear, that one?

And then a final basic property of mutexes is whether they're fair or unfair. So here's the thing. It's the easiest to think about it in the context of spinning. I have several threads that basically came to the same lock, and we decided they're going to spin. They're just going to sit there continually checking, waiting for that lock to be free.

So when finally the guy who has it unlocks it, maybe I've got a half a dozen threads sitting there. One of them wins. And which one wins? Well, they're spinning. It could be any one of them. Then it has one.

And so the issue that can go on is you could have what's called a starvation problem, where some guy is sitting there for a really long time waiting while everybody else is continually grabbing locks out from under his or her nose. So with a fair mutex, basically what you do is you go for the one that's been waiting the longest, essentially. And so, therefore, you never have to wait more than for however many things were there when you got there before you're able to go. Question.

AUDIENCE: Why is that better?

CHARLES LEISERSON: It can be better because you may freeze out our service if there's something that's-- you may never get to do the thing that you want to do because there's something else always interfering with the ability for that part of the program to make progress. This tends to be more of an issue in concurrent programming, where you have different programs that are trying to accomplish different tasks and you want to accomplish both tasks.

It does not come across-- in parallel programming, mostly we deal with unfair-- often unfair spinning locks because they're the cheapest. And we just trust that, a, we're not going to have any critical regions-- we write our code so we don't have critical regions that are really long, so nobody ever has to wait a very long time. But, indeed, dealing with a contention issue, as we talked about last week, can make a difference. good.

So here's an implementation of a simple spinning mutex an assembly language. So the first thing it does is it checks to see if the-- the mutex is free if its value is 0. So it compares the value of the mutex to 0. And if it is 0, it says, oh, it's free. Let me go get it. It then-- to get the mutex, what it does is it moves a 1 into the-- it basically moves 1 into a register, and then it exchanges the mutex with that register eax.

And then it compares to see whether or not it actually got the mutex. And if it didn't, then it goes back up to the top and starts again. And then the other branch is at the top there. It does this pause, and this apparently is due to a bug in x86 that they end up having to put this pause instruction in there. And then, otherwise, you jump to where the Spin_Mutex is and go again. And then, once you've done the Critical_Section, when you're done you free it by just setting it to 0.

So the question here is-- so the exchange instruction is an atomic exchange. So it takes the register and the memory value and it swaps them, and you can't have anything come in. So one of the things that might have you confused a little bit here is, wait a second. I checked to see if the mutex is free, and then I tried to get it to test if I was successful. Why? Why can't I just start out by essentially going to get mutex?

I mean, why do I need any of the code between Spin_Mutex and Get_Mutex? So if I just started with Get_Mutex, I would move a 1 in. I would exchange, check to see if I could get it. If I had it, fine. Then I execute the end. If not, I would go back and try again.

So why-- because if somebody has it, by the way, the value that I'm going to get is going to be 1. And that's what I swapped in, so I haven't changed anything. I go back and I check again. So why do I need that first part? Yes.

AUDIENCE: Maybe it's faster to just get [INAUDIBLE].

CHARLES LEISERSON: Yes. Maybe it's faster. So, indeed, it's because it's faster. Even though you're executing extra code, it's faster. Tell me why it's faster. And this will take you-- you have to think a little bit about the cache protocols and the invalidation issue. So why is it going to be faster? Yes.

AUDIENCE: Because I do the atomic exchange.

CHARLES LEISERSON: OK, good. Say more.

AUDIENCE: Basically, just to exchange atomically, you have to have [INAUDIBLE]. And you bring it in only just to do a swap.

CHARLES LEISERSON: Yes. So it turns out the exchange operation is like a write. And so in order to do a write, what do I need to do for the cache line that it's on?

AUDIENCE: To bring it in.

CHARLES LEISERSON: To bring it in. But how does it have to be brought in? Remember, the cache lines have-- let's ima--

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: You have to invalidate on the other ones, and you have to hold it in what state? Remember, the cache lines have-- if we take a look at just a simplified protocol where-- the MSI's protocol.

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: Yes. You have to have it-- in MSI or MESI, you have to bring it in in modified or at least exclusive state. So exclusive is for the MESI protocol. We mentioned that but we didn't really do it. Mostly we just went-- but I have to bring it in and modify it, where I guarantee there are no other copies.

So if I've got two guys that are polling on this location, they're both continually invalidating each other, and you create a whole bunch of traffic on the memory network. That's going to slow everything down. Whereas if I do the first one, what state do I get it in?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: Then you get it in shared state. What does the other guy get it in?

AUDIENCE: Shared.

CHARLES LEISERSON: Shared state. And now I keep going, just having it spinning in my own local cache, not generating any local traffic until the-- until somebody releases the lock, in which case it invalidates all those. And now you can actually get a little bit of a storm after the fact. There are in fact locks where you don't even get a storm after the fact called MCS locks. But this kind of lock is, for most practical purposes, just fine.

So everybody follow that description of what's going on there? So that first code, for correctness purpose, is not important. For performance, it is important. Isn't it great that you guys can read assembly language?

Now suppose that-- this is a spinning mutex. Suppose that I want to do a yielding mutex. How does this code have to change? So this is a spinning one. It just keeps checking. Instead, I want to return control to the operating system. So how does this code change if I do that? Yes.

AUDIENCE: Instead of the pause, [INAUDIBLE].

CHARLES LEISERSON: Like that. Yes, exactly. So instead of doing that pause instruction, which-- the documentation on this is not very clear. I'd love to have the inside scoop on why they really had to do the pause there.

But in any case, you take that no op that they want to have in there and you replace it with just a call to the yield, which allows the operating system to schedule something else. And then when it's your turn again, it resumes from that point. So that's the yield. So that's the difference in implementation, essentially, between a spinning mutex and a yielding mutex.

Now, there's another kind of mutex that is kind of cool which is called a competitive mutex. So think about it this way. I have competing goals. One is I want to get the mutex as quickly as possible after it's released. I don't want-- if it's unlocked, I don't want to sit there for a really long time before I actually acquire it. And, two, yes, but I don't want to sit there spinning for a really long time. And then-- because as long as I'm doing that, I'm taking up cycles and not accomplishing anything. Let me turn it over to some other thread that can use the cycles more effectively.

So there are those two goals. How can I get the best of both worlds here? Something that's close to the best of both worlds. It's not absolutely the best of both worlds, but it's close to the best of both worlds. What strategy could I do? So I want to claim it very soon.

So the point is that the spinning mutex achieves goal 1, and the yielding mutex achieved goal 2. So how can I-- what can I do to get both goals? Yes.

AUDIENCE: [INAUDIBLE] you could use some sort of message passing to [INAUDIBLE].

CHARLES LEISERSON: So you're saying use message passing to inform--

AUDIENCE: The waiting threads.

CHARLES LEISERSON: --the waiting threads. I'm think of something a lot simpler in this context. Because the message passing, you're going to have to go through-- to do message passing properly, you actually need to use mutexes that are to implement it. So you want to be a little bit careful about that. But interesting idea. Yes.

AUDIENCE: Could you try using an interrupt?

CHARLES LEISERSON: Using an interrupt. How would you do that?

AUDIENCE: Like once the [INAUDIBLE].

CHARLES LEISERSON: Yes. So, typically, if you implement interrupt you also need to have some mutual exclusions to do it properly, but-- I mean, hardware will support that. That's pretty heavy-handed as well. There's actually a very simple solution.

I'm seeing familiar hands. I want to see some unfamiliar hands. Who's got an unfamiliar hand? I see. You raised your left hand that time instead of your right hand. Yes.

AUDIENCE: You try to have whichever one is closest to being back to the beginning of the cycle take the lock.

CHARLES LEISERSON: Hard to measure that, right? How would you write code to measure that? Yes. Hmm. Hmm. Yes. Go ahead.

AUDIENCE: I have a question, actually.

CHARLES LEISERSON: OK, good.

AUDIENCE: Why does it [INAUDIBLE]?

CHARLES LEISERSON: Why doesn't it have a?

AUDIENCE: [INAUDIBLE]. Why does yielding mutex [INAUDIBLE]?

CHARLES LEISERSON: Because if I yield-- so what's the-- how often does-- if I context switch, how often is it going to be that I-- how long am I going to have to wait, typically, before I am scheduled again? When a code yields to the operating system, how often does the operating system normally do context switching? What's the rate at which it context switches for the different multiplexing of threads that it does onto the available processors? What's the rate at which it shifts? Oh, this is-- OK, that's going to be on the quiz. This is a numeracy thing. Yes. Do you know how frequently?

AUDIENCE: [INAUDIBLE] sub-millisecond [INAUDIBLE].

CHARLES LEISERSON: Not quite, but you're not off by more than an order of magnitude. So what are the typical rates that the system does context switching? So in human time, it's the blink of an eye. So it's actually around 10 milliseconds. So it does a hundred times a second. Some of them do. Some do 60 times a second. That's how often it switches.

Now, let's say it's a hundred times a second, 10 milliseconds. So you're pretty close. 10 milliseconds. How many orders of magnitude is that from the execution of a simple instruction? So we're going at more than a gigahertz. And so a gigahertz is 10 to the ninth, and we're talking 10 to the minus 9, and we're talking 10 to the minus 2. So that's 10 million instruction opportunities that we miss if we switch out.

And, of course, we'd probably only switch out for half our-- where are you along the thing. So you're only switching out maybe for half, assuming nothing else is going on there. But that means you're not grabbing the lock quickly after it's released, because you've got 10 million instructions that are going to execute before you're going to have a chance to come back in and grab it. So that's why a yielding one does not grab it quickly.

Whereas spinning is like we're executing this stuff at the rate of gigahertz, checking again, checking again, checking again. So why-- so what's the strategy here? What can I do? Yes.

AUDIENCE: Maybe we could spin for a little bit and then yield.

CHARLES LEISERSON: Hey, what a good idea. Spin for a while and then yield. So the idea being, hey, if the lock is released soon, then I will be able to grab it immediately because I'm spinning. If it takes a long time for the lock to yield, well, I will yield eventually. So yes, but how long to spin? How long shall I spin? Sure.

AUDIENCE: Somewhere close to the amount of time it takes to yield and come back.

CHARLES LEISERSON: Yes. Basically as long as a context switch takes, as long as it takes to go out and come back. And if you do that, then you never wait more than twice the optimal time. This is competitive analysis, which the theoreticians have gone off-- there's brilliant work in competitive analysis.

So the idea here is that if the mutex is released while you're spinning, then this strategy is optimal. Because you just sat there spinning, and as soon as it was there you got it on the next cycle. If the mutex is released after the yield, you've already spun for the equal to that. So you'll come back and get it within at most a factor of 2.

This is-- by the way, this shows up in the theory literature, if you're interested, is it's called the ski rental problem. And here's the idea. You're going to go-- your friends have persuaded you to go try skiing. Snow skiing, right? Pu-chu, pu-chu, pu-chu. Right?

And so you say, gee, should I buy the equipment or should I rent? After all, you may discover that you rent and then-- you buy it, and then you break your leg and never want to go back. Well, then, if you've bought it's been very expensive. And if you've rented, well, then you're probably better off. On the other hand, if it turns out you like it, you're now accumulating the costs going forward.

And so the question is, well, what's your strategy? And the idea is, well, let's look at what renting costs and what buying costs. Let me rent until it's equal to the cost of buying and then buy. And then I'm within a factor of 2 of having spent the optimal amount of money for-- because then if I break my leg after that, well, at least I-- I got-- I didn't spend more than a factor of 2. And if I get it before, then I've spent optimally. Yes.

AUDIENCE: So when you say how long a context switch takes, is that in milliseconds or--

CHARLES LEISERSON: Yes. 10 milliseconds. Yes. So spin for 10 milliseconds, and then switch. So now the point is that when you come back in, the other job's going to run for 10 milliseconds or whatever. So if you get switched out, then if the lock is released, you're going to be done in 20 milliseconds. And so you'll be within a factor of 2. And if it happened if the lockout released before then, you're right there to grab it.

Now, it turns out that there's a really clever randomized algorithm-- I love this algorithm-- from 1994 that achieves a competitive ratio of e over e minus 1 using a randomized strategy. And I'll encourage you, those of you have a theoretical bent, to go take a look at that. It's very clever. So, basically, you have some probability of, at every step, of whether you, at that point,

decide to yield or continue spinning. And by using a randomized strategy, you can actually get this to e over e minus 1.

Questions about this? So this is sort of some of the basics. I'm glad we went over some of that, because everybody should know these basic numbers about what things cost. Because, otherwise, you don't know where to spend it. So context switching time is on the order of 10 milliseconds. How long is a disk access compared to-- yes. What's a disk access?

AUDIENCE: 150 cycles?

CHARLES LEISERSON: 150 cycles? Hmm, that's a--

AUDIENCE: Or is that the cache?

CHARLES LEISERSON: That would be accessing DRAM. Accessing DRAM, if it wasn't in cache, might be 150 cycles. So two orders of magnitude or so. So what about a disk access? How long does that take? Yes.

AUDIENCE: Milliseconds?

CHARLES LEISERSON: Yes. Several milliseconds. So 10 milliseconds or 5 milliseconds depending upon how fast your disk is. But, once again, it's on the order of milliseconds. So it's helpful to know some of these numbers, because, otherwise, where are you spending your time?

Especially, we're sort of doing performance engineering in the small, basically looking within the pro-- within a multicore processor. Most performance engineering is on all the stuff on the outside, dealing with networking, and file systems, and stuff where things are really costly, and where, if you actually have a lot of time, you can write a fast piece of code that can figure out how you should best deal with these slow parts of your system. So those are all sort of good numbers to know. You'll probably see some of them on quiz 2.

Deadlock. I mentioned deadlock earlier. Let's talk about what deadlock is and understand this. Once again, I expect some of you have seen this, but I still want to go through it because it's hugely important material. And this is the issue, that holding more than one lock at a time can be dangerous.

So imagine that thread 1 says, I'm going to lock A, lock B, execute the critical section, unlock

B, unlock A, were A and B are mutexes. And thread 2 does something very similar. It locks B and locks A. Then it does the critical section, then it unlocks A and then unlocks B. So what can happen here?

So thread 1 locks A, thread 2 locks B. Thread 1 can't go and lock B because thread 2 has it. Thread 2 can't go and lock A because thread 1 has it. So they sit there, blocked. I don't care if they're spinning or yielding. They're not going anywhere. So this is the ultimate loss of performance. It's like-- it's incorrect. It's like you're stuck, you've deadlocked.

Now, there's three basic conditions for deadlock. Everybody understands this, right? Is there anybody who has a question, because just-- OK. There's three conditions you need for deadlock. The first one is mutual exclusion, that you're going to have exclusive control over the resources.

The second is nonpreemption. You don't release your resources. You hold until you finish using them. And three is circular waiting. You have a cycle of threads, in which each thread is blocked waiting for resources held by the next one. In this case, the resource is the lock.

And so if you remove any one of these constraints, you can come up with solutions that won't deadlock. So, for example, it could be that when I try to acquire a lock, if somebody else has them, I take it away. That could be one thing. Now, they may get into other issues, which is like, well, but what if he's actually doing real work or whatever? So all of these things have things. Or I don't insist that it be mutual exclusion, except that's the kind of problem that we're trying to solve. So these are generally the three things that are necessary in order to have a deadlock situation.

Now, in any discussion of deadlock, you have to talk about dining philosophers. When I was an undergraduate-- and I graduated in 1975 from Yale, a humanities school-- I was taught the dining philosophers, because, after all, philosophy is what you find at humanities schools. I mean, we have a philosophy department too. Don't get me wrong. But at Yale the humanities is huge. And so philosophy, I guess they thought this would appeal to the people who were not real techies in the background. I sort of like-- I was a techie in the midst of all these non-technical people.

Dining philosophers is a story of deadlock told by Tony Hoare based on an examination question by Edsger Dijkstra. And it's been embellished over the years by many, many, many retellers. And I like the Chinese version of this. There's versions where they use forks, but I'm

going to-- this is going to be-- they're dining-- I'm going to say that they are eating noodles with chopsticks.

And there are n philosophers seated around the table, and between every plate of noodles there's a chopstick. And so in order to eat the noodles they need two chopsticks, which to me sounds very natural. And so here's the code for philosopher i. So he's a philosopher, so he starts by thinking for a while. And then he gets hungry, he or she gets hungry.

So the philosopher grabs the chopstick on the right-- on the left, sorry. And then he grabs the one on the right, which is i plus 1. But he has to do that mod n, because if it's the last one, you've got to go around and grab the first one. Then eats, and then it unlocks the two chopsticks. And now they can be used by the other dining philosophers because they don't think much about sanitation and so forth. Because they're too busy thinking, right?

But what happens? What's wrong with this solution? What happens? What can happen for this? It's very simple. I need two chopsticks. I grab one, I grab the other, I eat. One day, what happens? Yes.

AUDIENCE: Everyone grabs the chopstick to the left and they're all stuck with one chopstick.

CHARLES LEISERSON: Yes. They grab one to the left, and now they go to the right. It's not there, and they starve. One day they grab all the things, so we have the starving philosophers problem. So motivated by this problem-- yes, question.

AUDIENCE: Is there any way to temporarily unlock it? Like the philosopher could just hand the chopstick [INAUDIBLE].

CHARLES LEISERSON: Yes. So if you're willing to preempt, then that would be preemption. As I say, it's got to be nonpreemptive in order for deadlock to occur. In this case, yes. But you also have to worry in those cases. Could be, oh, well if I couldn't get both, let me put them both down.

But then you can have a thing that's called livelock. So they all pick up their left. They see the right one's busy, so they put it down so somebody else can have it. They look around. Oh, OK. Let me pick up one. Oh, no. OK. And so they still starve even though they've done that.

So in that kind of situation, you could put in a time delay. You could say-- let everybody pick a random number to have a randomized scheme so that we're not-- so there are other solutions if you don't insist on nonpreemption. I'm going to give you one where we have nonpreemption

but we still avoid deadlock, and it's to go for that cyclic problem. So here's the idea.

Suppose that we can linearly order the mutexes. So I pick some order of the mutexes, so that whenever a thread holds a mutex L sub i and attempts to lock another mutex L sub j, we have that in this linear order-- L sub i comes before L sub j. Then you can't have a deadlock.

So in this case, for the dining philosophers, it would, for example, number the chopsticks from 1 to n, or 0 to n minus 1, whatever. And then grab the smaller one and then grab the larger one. And then it says then you would never have a deadlock. And so here's the proof. You know I like proofs. Proofs are really important. So I'm going to show you that if you do that, you couldn't have a cycle of waiting.

So suppose you had a cycle of waiting. We're in a situation where everybody is holding chopsticks, and one of them is waiting for another one, which is waiting for-- all the way around to the first one. That's what we need for deadlock to occur. So let me just look at what's the largest mutex on the cycle. Let's call that L max.

And suppose that it's waiting on mutex L held by the next thread in the cycle. Well, then, we have something that's bigger than the maximum one. And so that contradicts the fact that I grab them-- whenever I grab them, I do it in order. So very simple-- very simple proof that you can't have deadlock if you grab them according to a linear order.

And so for this particular problem, what I do is, instead of grabbing the one on the left and one the right, as I say, you grab the smaller of the two and then grab the larger of the two. And then you're guaranteed to have no deadlock. Does that make sense?

Now, if you're going to use locks in Cilk, you have to realize that in the operating-- in the runtime system of Cilk, they're doing-- they're using locks. You can't see them. They're encapsulated, as we talked about. The nondeterminism in Cilk is encapsulated. It's still going on underneath the covers. And if you start introducing your own nondeterminism through the use of locks you can run into trouble if you're not careful.

And let me give you an example. This is a situation-- you can deadlock your program in Cilk with just one lock. So here's an example of a code that does that. So main spawns off foo. And foo basically locks the lock L and then unlocks it. And, meanwhile, after it spawns off foo, the continuation goes and it locks L itself, and then does a sync, and then it unlocks it.

So what happens here? We sort of have a situation like this, where the locking I've done with an open bracket, and an unlock, a release, I'm doing with a closed bracket. So I'm spawning off foo, which is the lower part there, and locking and unlocking. And up above unlocking then unlocking.

So what can happen here? I can go and I basically spawn off the child, but then I lock. And now the child goes and it says, whoops, can't-- foo is going to wait here because it can't grab the lock because it's owned by main. And now we get to the point where main has to wait for the sync, and the child is never going to complete because I hold the resource that the child needs to complete.

So don't hold mutexes across Cilk syncs. That's the lesson there. There are actually places you can, but if you don't hold them across that, then you won't run into this particular problem. A good strategy is only holding mutexes within strands. So there's no parallelism. So you have it bounded.

And also, that's a good idea generally because you want to hold mutexes as short amount of time as you possibly can. So, for example, if you have a big calculation and then you want to assign something atomically, don't put the big calculation inside the critical region. Move the calculation outside the critical region, do the calculation you need to do, and then acquire the locks just to do the interaction you need to set a value. And then you'll have a lot faster code because you're not holding up other threads for a long time.

And always try to avoid nondeterministic programming. But that's not always possible. So any questions about that? Then I want to go on a really interesting topic because it's a really recent research level topic, and that's to talk about transactional memory. Who's heard this term before? Anybody?

So the idea is to have database transactions, that you have things like database transactions where the atomicity is happening implicitly. You don't specify locks. You just say this is a critical region. Don't interrupt me while I do this critical region. The system works everything out.

Here's a good example of where it might be useful. Suppose we want to do a concurrent graph computation. And so you take people involved in parallel and distributed computing at MIT and you say, OK, I want to do Gaussian elimination on this graph. Now, you guys, I'm sure most of you know Gaussian elimination from the matrix context. Do you know what it means in a graph context?

So if you have a sparse matrix, you actually have a graph. And Gaussian elimination is a way of manipulating the graph, and you get exactly the same behavior as you get in the dense one. So I'll show you what it is. You basically pick somebody to eliminate.

[STUDENTS LAUGH]

And now what you do is look at all this vertex's neighbors. Those guys. And what you do is you eliminate that vertex-- bye bye-- and you interconnect all the neighbors with all the edges that don't already exist. And that's Gaussian elimination. And if you think of it in terms of matrix fashion, the question is, if you have a sparse matrix, where are you going to get fill in? What are the places that you need to update when you do a pivot in Gaussian elimination in a matrix?

So that's the basic notion of graph-- of doing Gaussian elimination. But now we want to deal with the concurrency. And the problem occurs if I want to eliminate two nodes at the same time. Because now they're adjacent to each other, and if I just do what I expressed, there's going to be all kinds of atomicity violations, et cetera. By the way, the reason I'm picking these two folks is because they're going to a better place.

So how do you deal with this? And so in transactional memory, what I want to be able to do is just simply say, OK, here's the thing that I need to be atomic. And so if I look at this code, it's basically saying who are my neighbors, and then let me identify all of the edges that need to be removed, the ones that I just showed you that we removed. Now let me get rid of the element v. And now, for all of the neighbors of u, let us add in the edge between the neighbor and-- between the pairs of neighbors. So that's basically what it's doing.

And I'd like to just say that's atomic. And so the idea is that if I express that as a transaction, then the idea is that, on the transaction commit, all the memory updates in the critical region appear to take it happen at once. However, in transaction, remember the idea is, rather than forcing it to go forward, I can have the transactions abort. So if I get a conflict, I'll abort one and restart it. And then the restarted transaction may take a different code path, because, after all, I may have restructured the graph underneath. And so it may do something different the second time through than the first. It may also abort again and so forth.

So when you study transaction, transactional memory-- let me just do a couple of definitions.

One is a conflict. That's when you have two transactions that are-- they can't both complete. One of them has to be aborted. And aborting, by the way, is once again violating the nonpreemptive nature. Here we're going to preempt one of them by keeping all the states so I can roll a state back and restart it from scratch.

So contention resolution is deciding which of the two conflicting transactions to wait or to abort and restart, and under what conditions you do that. So the resolution manager has to figure out what happens in the case of contention. And then forward progress is avoiding deadlock of course, but also livelock and starvation. You want to make sure that you're going to make-- because what you don't want to have happen, for example, is that two transactions keep aborting each other and you never make forward progress. And throughput, well, you'd like to run as many transactions as concurrently as possible.

So I'm going to show you an algorithm for doing this. It's a really simple algorithm. It happens to be one that I discovered just a couple of years ago. And I was surprised that it did not appear in the literature, and so I wrote a very short paper on it. Because what happens for a lot of people is they-- if they discover there's a lot of aborting, they say, oh, well let's grab a global lock. And then if everybody grabs a global lock, you can do this sort of thing. You can't deadlock with a single lock if you're not also doing things like Cilk sync or whatever.

But, in any case, if you have just a single lock, everybody falls back to the single lock, and then you have no concurrency in your program, no performance, until everybody gets through the difficult time. So this is an algorithm that doesn't require a global lock. So it assumes the transactional memory system will log the reads and writes. That's typically true of any transaction, where you log what reads and writes you're doing so that you can either abort and roll back, or you can-- when you abort-- or else you sandbox things and then atomically commit them.

And so we have all the mechanisms for aborting and rolling back. These are all very interesting in their own right, and restarting. And this is going to basically use a lock-based approach that uses two ideas. One is the notion of what's called a finite ownership array, and another is a thing called release-sort-reacquire. And let me explain those two things, and I'll show you really quickly how this beautiful algorithm works.

So you have an array of anti-starvation mutual exclusion locks. So these are ones that are going to be fair, so that you're always going to the oldest one. And you can do an acquire, but

we're also going to add in a try acquire. Tell me whether, if I tried to acquire, I would get it. That is, if I get it, give it to me. If I don't get it, don't wait. Just tell me that I didn't get it, and then release.

And there's an owner function that maps all of the-- function h that maps my universe of memory locations to the indexes in this finite ownership array, this lock array. So the lock has length-- array has length n, has n slots in it. To lock a location x in the set of all possible memory locations, you actually acquire lock of h of x. So you can think of h as a hash function, but it doesn't have to be a fair hash function or whatever. Any function will do. And then, yes, there will be some advantages to picking some functions or another one.

So rather than actually locking the location or locking the object, I lock a location that essentially I hash to from that object. So if two guys are trying to grab the same location, they will both grab the same lock because they've got the same hash function. But I may have inadvertent locks where if I were locking the objects themselves, I wouldn't have them both trying to acquire the same lock. That might happen in this algorithm.

So here's the idea. The first idea is called release, sort, and reacquire. So that's the ownership array part that I just explained. Now here's the release, sort, reacquire. Before you access a memory location x, simply try to grab lock of x greedily. And if you have a conflict-- so if you don't have a conflict, you get it. You just simply try to get it. And if you can, that's great.

If not, then what I'm going to do is roll back the transaction but don't release the locks I hold, and then release all the locks with indexes greater than h of x. And then I'm going to acquire the lock that I want. And now, at that point, I've released all the bigger locks, so I'm acquiring the next lock. And then I reacquire the released locks in sorted order. So I go through all the locks I released and I reacquire them in sorted order.

And then I start my transaction over again. I try again. So what happens each time through this process, I'm always-- whenever I'm trying to acquire a lock, I'm only holding locks that are smaller. But each time that I restart, I have one more lock that I didn't used to have before I restart my transaction, which I've acquired in the order, in the linear order, in that ownership array from 0 to n minus 1.

And so here's the algorithm. I'll let you guys look at it in more detail, because I see our time is up. And it's actually fun to take a look at, and we'll put the paper online. There's one other topic that I wanted to go through here which you should know about, is this locking anomaly

called convoying. And this was actually a bug that we had-- a performance bug that we had in our original and MIT-Cilk. So it's kind of a neat one to see and how we resolved it. And that's it.