Let's get going here. So this is a lecture that's actually appropriate for Halloween, because it's a scary topic. Non-deterministic programming. So we've been looking mostly at deterministic programs. So a program is deterministic on a given input if every memory location is updated with the same sequence of values in every execution.

So if you look at the memory of the machine, you can view that as, essentially, the state of the machine. And if you're always updating every memory location with exactly the same sequence of values, then the program is deterministic. Now it may be that two memory locations may be updated in a different order.

So you may have one location which is updated first in one execution, and another that's second, and then in a different execution, they may be a different order. That's OK, generally. The issue is whether or not every memory location sees the same order. And if they do, then it's for every execution, then it's a deterministic program.

So what's the advantage of having a deterministic program? Yeah?

AUDIENCE: It always runs the same way [INAUDIBLE].

PROFESSOR: It always runs the same way. So what? What's that good for?

AUDIENCE: So you can find bugs easier.

PROFESSOR: Yeah, debugging. It's really easy to find bugs if every time you run it it does the same thing. It's much harder to find bugs if, when you run it, it might do something different.

So that leads to our first major rule of thumb about determinism, which is you should always write deterministic programs. Don't write

non-deterministic programs. And the only problem is, boy is that poor quality there. So basically, it says, always write non-deterministic programs unless you can't. So sometimes, the only way to get performance is to do something non-deterministic. So this lecture is basically about some of the ways of doing non-deterministic programming. So it's appropriate that we say this is not for those who are faint of heart. We are treading into dangerous territory here. So the basic rule is, as I say, any time you can, make your program deterministic.

So we're going to talk about the number one way that people introduce non-determinism into programs, which is via mutual exclusion and mutexes, which are a type of lock, and then look at some of the anomalies that you get. Besides just things being non-deterministic, you can also get some very, very weird behavior sometimes for the execution.

So we'll start out with mutual exclusion. So let's take a look, for example, suppose I'm implementing a hash table as a set of bins. And I'm resolving collisions with chaining. So here, each slot of my hash table has a chain of all the values that resolve to that slot. And if I have a value x, let's say it has key 81, and I want to insert x into the table, I first compute a hash of x. And let's say it hashes to this particular list here.

And then what I do is I say, OK, let me insert x into the table. So I make the next pointer of x point to whatever is the head of the table. And then I make the table 0.2x. And that effectively inserts x into the hash table. Fairly straightforward piece of code. I would expect that most of you could write that even on an exam and get it right.

But what happens when we say, oh, let's have some concurrency. Let's have the ability to look up things in a hash table in different parallel branches of a parallel program. So here, we have a concurrent hash table now where I've got two values, and I'm going to have two different threads inserting x and y. So one of them is

going to do this one, and one of them is going to do this one. So let's just see how this can screw up.

So first, we hash x, and it hashes to this particular slot. So then we do, just as we're doing before, making its next pointer point to the beginning of the array. Then y gets in the picture, and it decides oh, I'm going to hash. And oh, it hashes to exactly the same slot.

And then y makes its next pointer point to the same to the head of the list. And then it sets the head of the list to point to y. So now y is in the list. Whoops, now x puts itself in the list, effectively taking y out of the list. So rather than x and y both being in the list, we have a concurrency bug.

So this is clearly a race. So it's a determinacy race, because we have two parallel instructions accessing essentially the same location, at least one of which-- in this case both of them-- performing a store to that location. So that's a determinacy race. And how things are going to work out depends upon which one of these guys goes first. Notice, as with most race bugs, that if this code all executed before this code, we're OK. Or if this code all executed before this code, we're OK. So the bug occurs when they happen to execute at essentially the same time and their instructions interleave.

So this is a race bug. So one of the classic ways of fixing this kind of race bug is to insist on some kind of mutual exclusion. So a critical section is a piece of code that is going to access shared data that must not be executed by two threads at the same time. So it shouldn't be accessed by two threads at the same time. So it's mutual exclusion. So that's what a critical section is.

And we have a mechanism that operating systems typically provide-- as well as runtime systems, but you can build your own-- called "mutexes," or "mutex locks," or sometimes just "locks." So a mutex is an object that has a lock and unlock member function. And any attempt by a thread to lock an already locked mutex causes that thread to block.

And "block" is, by the way, a hugely overused word in computer science. In this case, by "block," they mean "wait." It waits until the mutex is unlocked. So whenever you have something that's locked, somebody else comes and tries to grab the lock. The mutex mechanism only allows one thread to access it. The other one waits until the lock is freed. Then this other one can go access it.

So what we can do is build a concurrent hash table by modifying each slot in the table to have both a mutex, L, and a pointer called "head" to the slot contents. And then the idea is that what we'll do is hash the value to a slot. But before we access the elements of the slot, we're going to grab the lock on the slot. So every slot in the table has a lock here. Now, I could have a lock on the whole table. What's the problem with that? Sure.

AUDIENCE: [INAUDIBLE] basically can't do anything. You can't read. You couldn't be reading from the table.

PROFESSOR: Yeah, so if you have a lock on the whole table--

AUDIENCE: You would defeat the purpose [INAUDIBLE]

PROFESSOR: You defeat the purpose of trying to have a concurrent hash table, right? Because only one thread can actually access the hash table at a time. So in this case, what we'll do is we'll lock each slot of the hash table. And there are actually mechanisms where you can lock each element of the hash table or a constant number of elements. But basically, what we're trying to do is make it so that the odds are that if you have a big enough table and relatively few processors you're running on, the odds that they'll conflict are going to be very low.

So what we do is we grab a lock on the slot, and then we play the same game of inserting ourselves at the head. And then we unlock the slot. So what that does is it means that only one of the two threads in the previous example can actually execute this code at a time. And so it guarantees that the two regions of code will either execute in this order or in this order, and you'll never get the instructions interleaved.

Now, this is introducing non-determinism. Why is this going to be non-deterministic? Yes?

AUDIENCE: [INAUDIBLE] lock first, it'll be [INAUDIBLE].

PROFESSOR: Yeah, depending upon which one gets the lock first, the length list in there will have the elements in a different order. So a program that depends on the order of that list is going to behave differently from run to run. So let's recall the definition of a determinacy race. It occurs when two logically parallel instructions access the same memory location, and at least one of the instructions performs a write.

So that is, we do have a determinacy race when we introduce locks. Locks are essentially, we're going to have an intentional determinacy race. So a program execution with no determinacy races means the program is deterministic on that input.

So if there are no determinacy races, then although individual locations may be updated in a different order in a parallel execution, every memory location will be updated by exactly the same thing at the same time. The order will be of update of operations on any given location will be the same always. So that's actually a theorem, which we're not going to prove.

But I think if you think about it, it's fairly straightforward. If you never have two guys in parallel that could possibly affect the same location, then the behavior always is going to be the same thing. Things are going to get written in the same order.

So the program in that case always behaves the same on that given input, no matter how it's scheduled and executed. We'll always have essentially the same behavior, even though it may get scheduled one way or another. And one of the nice things that we have in our race detection tool Cilkscreen is that if we do have determinacy races that exist in an ostensibly deterministic program-- that is, a program with no mutexes. If basically it just reads and writes on locations and so forth, then Cilkscreen guarantees to find such a race. So It's nice that we get a guarantee out of Cilkscreen.

So this is all beautiful, elegant, everything works out great if there are no determinacy races. But when we do something like a concurrent hash table, we're intentionally putting in a determinacy area. So that asks sort of a natural question. Why would I want to have a concurrent hash table? Why not make it so that my program is deterministic?

Why might a concurrent hash table be an advantageous thing to have in a program that you wanted to go fast? Some ideas? Where might you want to use it? Yeah?

AUDIENCE: Speed?

PROFESSOR: Yeah, speed. But I mean, what's an application? What's a use case, as the entrepreneurs would ask you? Where is it that you would really want to use a concurrent hash table to give you speed? Yeah?

AUDIENCE: If you started using it [INAUDIBLE] along with your system [INAUDIBLE] values.

PROFESSOR: Yeah, it could be that there's some sort of global table that you want a lot of people to be able to access at one time. So if you lock down and only had one thread accessing at a time, you reduce how much concurrency that you could have. That's a good one. Yeah?

AUDIENCE: Perhaps most of the time, people are just reading. So if you had something concurrent, your reading should be fine. So in that case, a lot more reading high performance [INAUDIBLE]

PROFESSOR: Yeah, so in fact, there's a type of lock called a reader-writer lock, which allows one writer to operate, but many readers. So that's another type of concurrency control. So just another place, a common place that you use it, is when you're memoizing. Meaning I do a computation, I want to remember the results so that if I see it again, I can look it up rather than having to compute it again from scratch.

So you might keep all those values in a hash table. Well, if I go in the hash table, now I'm going to have concurrent accesses to that hash table if I've got a parallel program that wants to do memorizing. And there are a bunch of other cases.

So we have determinacy races. And we have a great guarantee that if there is a race, we guarantee to find it. Now, there's another type of race, and in fact, you'll hear more about this type of race if you read the literature than you hear about determinacy races. So a data race occurs when you have two logically parallel instructions holding no locks in common. And they access the same location, and at least one of the instructions performs a write.

So this is saying that I've got accesses. And if they have no locks in common-- so it could be that you have a problem where one of them holds a lock L, and another one holds L prime. And then they access the location, that's going to be a data race, because they don't hold locks in common.

But if I have L and L being the locks that the two threads hold, and they access the same location, that's not a data race now. It is a determinacy race, because it's going to matter which order it is, but it's not a data race, because the locks, in some sense, are protecting access.

So Cilkscreen, in fact, understands locks and will not report a determinacy race unless it is also a data race. However, since codes that use locks are non-deterministic by intention, they actually weaken Cilkscreen's guarantee. And in particular, in its execution that it does, if it finds a data race, it's going to say, I'm going to ignore that data race.

But now it is only going to follow one of the two paths that might arise from that data race. In other words, it doesn't follow both paths. If you could think about it, when one of them wins-- so you have a race between two critical sections.

When one of them wins, you can imagine that's one possible outcome of the computation. When the other wins, it's another path. And what Cilkscreen does is it picks one path. In fact, it picks the path which is the one that would occur in the cereal execution. So there's a whole path there that you're not exploring. So Cilkscreen's guarantee is not going to be strong there.

However, if the critical sections, in fact, commute-- that is, they do exactly the same

thing, no matter what the order. So for example, if they're both incrementing a value, then the result, after doing one versus after the other is the same value, then you get a guarantee out of Cilkscreen. So Cilkscreen could still be very helpful for finding bugs, because typically, when you organize your computation, if it occurs in this order, there's typically some execution or input where you can make things occur in the other order.

So you can actually cover more races than you might imagine on first blush. But it is a danger. But what we're talking about today is dangerous programming, non-deterministic programming. So when you start using mutexes, some of the guarantees and so forth get much dicier. Any questions about that?

Now, if you have no data races in your code, that doesn't mean that you have no bugs. So for example, here's a way somebody might fix that insertion code. So we hash the key, we grab a lock, we set x next to be whatever is the head of the list, and then we do an unlock. And now we lock it again. Now we follow the head to set x-- sorry, we set x to be the head of the list and then unlock again.

And now notice that in this case, technically, there is no data race if I have two concurrent threads trying to access these at a time, because all the axis I'm doing, I'm holding lock L. Nevertheless, I can get that same interleaving of code that causes the bug. So just because you don't have a data race doesn't mean that you don't have a bug in your code. As I say, this is dangerous programming.

However, typically, if you have mutexes and no data races, usually it means that you went through and thought about this code. And if you were thinking about this code, you would say, gee, really I'm trying to make these two instructions be the critical section. Why would I unlock and lock again? So most of the time, as a practical matter, if you don't have data races, it probably means you did the right thing in terms of identifying the critical sections that needed to be locked and not unlocking things in the middle of them.

So as a practical matter, no data races usually means it's unlikely you have bugs. But no guarantees. As I say, dangerous programming. Non-deterministic

programming is dangerous program. Any questions about that? Anybody scared off yet? Yeah?

AUDIENCE: So what you can do is the opposite. You don't have any bugs, but you made the critical distinction to [INAUDIBLE]

PROFESSOR: Yes, so certainly from a performance point of view, one of the problems with locking is that-- and we'll talk about this a little bit later-- with locking is that if you have a large section that you decide to lock, it means other threads can't do work on that section. So they're spinning, wasting cycles. So generally, you want to try to lock things as small as possible.

The other problem is, it turns out that there's overhead associated with these locks. So if there's overhead associated with the locks, that's problematic as well, because now you may be slowing down. If this is in an inner loop, notice that we've now, even if I just have the lock and unlock without these two spurious ones here, we may be more than doubling the overhead. In fact, locking instructions tend to be much more expensive than register operations. They usually cost something on the order of going to L2 cache as a minimum. So it's not even L1 cache. It's like going out to L2 cache.

Now, it turns out there are some times where you have data races. So we say if there are no data races, then you have no guarantee there's no bugs. If there are data races, your program still may be correct. Here's an example of a code where you might want to allow a benign data race.

So here we have, let's say, an array A that has these elements in it. And we want to find, what is the set of digits in the array? So these are all going to be values between 0 and 9. And I want to know which ones are present of the digits 0 to 9. Which ones are not present?

So I can write a little code for that. Let me initialize an array called "digits" to have all-zero entries. And now let me go through all the elements of A and set digits of whatever the digit is to be 1. So set at 1 if that digit is present. And I can do that in

parallel, even.

So what can happen here is I can have, if I've done this in parallel, this particular update of digits of 6 will be set to 1 when this one is being sent to 1. Is that a problem? In some sense, no. They're both being set to 1. Who cares? But there is a race there. There is a race, but it's a benign race. Well, it may or may not be benign.

So there's a gotcha on this one. So this code only works correctly if the hardware writes the array elements atomically. So for example, not on the x86-64 architecture we're using. But on some architectures, you cannot write a byte value. You cannot write a byte value as an atomic operation. It implements a right to a byte by reading a word, masking out things, changing the field, masking again, and then writing it back out.

So you can have a race on a byte value. In particular, even if I were going to do this with bits, I could have a race on bits, although C doesn't let me access bits directly. The smallest unit I can access is a byte.

So you have to worry about what's the level of atomicity provided by your architecture? So the x86 architecture, the grain size of atomic update is you can do a single-byte write, and it will do the right, proper thing-- do the right thing on the write. So we have both things. No bugs. No data races doesn't mean no bugs. Presence of data races doesn't mean you have bugs. But generally, they're fairly well overlapped.

Now, why would I not want to put in a lock and unlock here just to get rid of the race? If I run Cilkscreen on this, it's going to complain. It's going to say, you've got a race here. Why would I not want to put a lock on here, for example?

AUDIENCE: Because then we don't have parallelism anymore?

PROFESSOR: No, well, I'd have parallelism maybe up to 10, for example, right? Because I have 10 different things that could be going on at a time. But that's one reason. That is one reason. What's another reason why I might not want to put locks in here?

AUDIENCE: [INAUDIBLE]

PROFESSOR: It could be that all the numbers-- that's a case where it doesn't get me much speedup. But what's another reason I might want to do this?

AUDIENCE: [INAUDIBLE]

PROFESSOR: I think you're on the right track. Overhead. Yeah. Overhead. This is my inner loop. So if I'm locking and unlocking, all this is doing is just doing a memory [? dereference ?] and an assignment.

And that may be fairly cheap, whereas if I grab a lock and then release the lock, those operations may be much, much more expensive. So I may be slowing down the execution of this loop by more than I'm going to gain out of the parallelism of this. So I may say, I may reason, hey, there is a good reason why not have a data race there.

So I may want to have a data race, and I may want to say that's OK. And if that happens, however, you're now going to get warnings out of Cilkscreen. And I generally recommend that you have no warnings on Cilkscreen when you run your code. So the Cilk environment provides a mechanism called "fake locks." So a fake lock allows you to communicate to Cilkscreen that a race is intentional.

So what you then do is you put a fake lock in around this access here. And what happens is when Cilkscreen runs, it says, oh, you grabbed this lock, so I shouldn't report a race. But during execution, no lock is actually grabbed, because it's a fake one. So it doesn't slow you down it all at runtime, but Cilkscreen still thinks that a lock is being acquired.

Questions about that? So this is if you want to have an intentional race, this is a way you can quiet Cilkscreen. Of course, it's dangerous, right? It's yet another example of what's dangerous here. Because what happens if you did it wrong? What happens if there really is a bug there?

You're now telling it to ignore that bug. So one way that you can make your code-- if you put in fake locks everywhere, you could make it so, oh, Cilkscreen runs just great, and have your code full of race bugs. So if you use fake locks, you should document very carefully that you're doing so and why that's going to be a safe thing to do. Any questions about that?

By the way, one of the nice things about some of the concurrency platforms like Cilk is that they provide a layer of abstraction where generally, you don't have to do very much locking. If you program with Pthreads, for example, you're locking all the time. So you're writing non-deterministic programs all the time, and you're debugging non-deterministic programs all the time. Whereas Cilk provides a layer of programming where you can do most of your programming in a deterministic fashion.

And occasionally, you may want to have some non-determinism here or there. But hopefully you can manage that if you do it judiciously. Any questions about mutexes and uses for them and so forth? Good.

So let's talk about how they get implemented. Because as with all these things, we want to understand not just what the abstractions is but how it is that you actually implement these things so that you can reason about them more cogently. So there's typically three major properties of mutexes when you look at them.

And when you see documentation for mutexes, you should understand what the difference is of these things. The first is whether it's a yielding mutex or a spinning mutex. So a yielding mutex, when you spin, it returns control to the operating system. And why might you want to do that? Whereas a spinning one just consumes processor cycles. Why would you want to do that? Yeah.

AUDIENCE: [INAUDIBLE] allow other threads.

PROFESSOR: Yeah, it can allow other threads or other jobs that could be running to use the processor while you're waiting. What's the downside of that? To speak to the-- either one. Go ahead.

AUDIENCE: It might be possible that whatever you're trying to do is essential, and you really want to get that done [UNINTELLIGIBLE] everything else executes. So you really want [INAUDIBLE]

PROFESSOR: Yeah, context switching a thread out is a heavyweight operation. And It may be, if you end up context switching out, it may be you only had to wait for a half a dozen cycles and you'd have the lock. But instead, now you're going and you're doing a context switch and may not get access to the machine for another hundredth of a second or something. So it may be on the order of 10 to the 6th-- a million instructions before you get access again, rather than just a few.

The second property of mutexes is whether they're reentrant or not. So a reenttrant mutex allows a thread that's holding a lock to acquire it again. So I may hold the lock, and then I may try to acquire the lock again. Java is full of reentrant locks, reentrant mutexes.

So why is this a positive or negative? What are the pros and cons of this one? Why might reentrancy be a good thing to want? Why would I bother doing-- why would I grab a lock that I already have?

AUDIENCE: It'd be too easy to do a check [INAUDIBLE].

PROFESSOR: It lets you do what?

AUDIENCE: It lets you not have to worry about locking when you already have a lock.

PROFESSOR: It lets you not worry about it. That's right. But why is that valuable?

AUDIENCE: It saves you one line in an If statment to check if you have a lock or not.

PROFESSOR: That could be. Basically, the If statement is embedded in there. But why would I care? Why would I want to be acquiring something that I already have? In what programming situation might that arise? This seems kind of weird, right? Could be recursion. Yeah.

So usually, what it comes from is when you have objects, and you have several

methods on the object. And what you'd like to do is, if somebody's calling the method from the outside, you would like to be able to execute that particular-- I guess in C++ they don't call them "methods." They call them "member functions." "Member functions," they call them. In Java, they call them "methods," and in C++, they call them "member functions." Doesn't matter. It's the same thing.

So when you access one of these, normally, from the outside, you want to make sure you grab the lock associated with the object. However, it may be that what you're doing inside the object is you want to be able-- one of the operations may be a more complex operation that wants to use one of its own implementations. So rather than implementing it twice-- once in the locked form, once without getting the lock-- you just implement it once, and you use reentrant locks. And that way, you don't have to worry about, in coding those things, whether or not you've already got it. So that's probably the most common place that I know that people want reentrant locks.

Naturally, to acquire a reentrant lock, you have to do some kind of If statement, which is a conditional. And as you know, if it's an unpredictable branch, that's going to be very expensive. So generally, there is a cost to making it reentrant.

The third property is whether the lock is fair or unfair. So a fair mutex puts block threads essentially into a FIFO queue. And the unlock operation unblocks the thread that has been waiting the longest. So it makes it so that if you try to acquire a lock, you don't have some other thread coming in and trying to access that lock and getting ahead of you. It puts you in a queue.

So an unfair mutex lets any blocked thread go next. So the cheapest thing to implement is a spinning, non-reentrant, unfair lock-- mutex. Those are the cheapest ones to implement. Very lightweight, very easy to use.

The heavyweight ones are a yielding, reentrant, fair lock. And of course, you can have combinations, because all of these have, as you can see, different properties in terms of convenience of use and so forth, as well as different overheads. So there's some cases where the overhead isn't a big deal because it's not in the inner

loop of a program or a heavily executed statement.

So let's take a look at one of the simplest locks, which is a simple spinning mutex. This is the x86 code for how to acquire a lock. So let's run through this. So we start out at the top. And I check to see if the mutex is 0, which is basically, it's going to be 0 if it's free and 1 if it has been acquired.

So we compare it. If it's free, then I jump to try to get the mutex. Otherwise, I execute this PAUSE instruction, and this turns out to be a-- it's humorous. It's x86 hack to un-confuse the pipeline.

So it turns out that in this case, if you don't have a pause here-- which is no-op and does nothing-- x86 mispredicts something or whatever, and it's more time consuming than if it doesn't have that there. The manual explains very little about this hardware bug except to say, put in the pause. So if you didn't get it, then you jump to spin mutex, and try again, check to see if it's free.

Now, notice that we're going to spin until it's free, and then we're going to try to get it. Why not just try to get it first? Well, think about that while we go through how to get it, and then I'll ask it again. Think about why it is that you might want to get it first.

So if I want to get the mutex, I first get a value of 1 in my register. And then I compute this exchange operation, which exchanges the value of the mutex with the value of the-- with the one that I have. So it exchanges the memory location with the register.

Now, this is an expensive operation-- exchange-- because it's an atomic exchange, and it typically has to go at least out to L2 to do this. So it's an expensive operation, because it's a read-modify-write operation. I'm swapping my register value with a value that's in the mutex.

So it turns out that if it's 0, then it means I got it. So I compare it with 0, and if it's equal to 0, I go onto the critical section. When I'm done with the critical section, I release the mutex by basically storing 0 in there, because I'm the only one who

accesses the mutex at this point.

If I didn't get it, if the value is 1, notice that because I'm swapping a 1 in, even though the 1 got swapped in, well, there was a 1 there before. So it basically did not affect the value of the mutex. But I discover, oh, I don't have it. Then we go all the way back up there to spin mutex.

So here's the question. Why do I need all this preamble code? Why not just go straight to Get_Mutex, make the spin mutex here be a jump to Get_Mutex? Yeah?

AUDIENCE: Maybe it's because the exchange is expensive.

PROFESSOR: Excuse me?

AUDIENCE: The exchange is--

PROFESSOR: Yeah, because the exchange is expensive. Exactly. So this code here, I can compare. And as long as nobody's touching anything, this becomes just L1 memory accesses. Whereas here, it's going to be at least L2 to do the exchange operation. So rather than doing that-- moreover, this one actually changes the value.

So what happens when I change the value of the mutex? Even though I change it to the same value, what happens in order to do that exchange? Remember from several lectures ago. What's going to happen when I make an exchange there? What does the hardware have to do? What's the hardware going to do on any store to a shared memory location, to a memory location in shared memory that is actually shared? Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, it's got to invalidate all the other copies. So if everybody spinning here-- imagine that you have five guys spinning, doing exchanges-- they're all creating all this traffic of invalidations, what's called an "invalidation storm." So they create an invalidation storm as they all are invalidating each other so that they can get access to it so that they can change the value themselves.

But up here, all I'm doing is looking at the value. All I'm doing is looking at the value to see if it's free. And it's not until the guy actually frees the value that it actually-- actually, this is interesting. I think I wrote this with Intel syntax, rather than AT&T, didn't I? The MOV mutex, 0 moves 0 into the mutex, which is Intel syntax.

I probably should have converted this to AT&T, because that's what we're generally using in the class. I'll fix that before I put the slides up. Basically, I pulled this out of the Intel manual.

So any questions about this code? Everybody see how it works? It relies on this atomic exchange operation. And I'm going to end up sitting here spinning until maybe I can get access to it. When I have a chance to get access to it, I try to get it. If I don't get it, I go back to spinning.

How do I convert this to a yielding mutex?

AUDIENCE: Instead of having that spinning mutex, you should-- you shouldn't have that. You should just have something that allows you to just [INAUDIBLE].

PROFESSOR: Yeah, so actually, the way you do it is you replace the PAUSE instruction. Exactly what you're saying. You've got the right place in the code. We basically call a yield. And you can use, for example, pthread_yield. What it tells the operating system is, give up on this quantum. You can schedule me out. Somebody else can be scheduled. Now, if nobody else is there to be scheduled, often you'll just get control back, and you'll jump again and give the operating system another time.

Now, one of the things I've seen in computer benchmarks that use locking is that they all use spin locks. They never use the yielding, because if you yield, then when the lock comes free, you're not going to be ready to come back in. You may be switched out.

So a common thing that all these companies do when they're vying for who's got the fastest on this benchmark or fastest on that benchmark is they go through and they convert all their yielding mutexes into spinning mutexes, then take their measurements, when in fact, as a practical matter, they can't actually ship code that

way. So you'll see this kind of game played where people try to get the best performance they can in some kind of laboratory setting. It's not the same as when you're actually doing a real thing.

So you have a choice here. There's kind of a tension here. You'd like to claim the mutex soon after it's released. And you're not going to get that if you yield. At the same time, you want to behave nicely and waste few cycles. So what's the strategy for being able to accomplish both of these goals?

So one of these goals is the spinning mutex does a great job of claiming the mutex soon after it's released. The yielding mutex behaves nicely and wastes few cycles. Is there the best of both worlds? There's certainly the worst of both worlds, right?

What's the best of both worlds? How might we accomplish both of these goals with small modification to the locking code? So it turns out you can get within a factor of two of optimal. How might you do that while wasting few cycles?

So here's the idea. Spin for a little while, and then, if after a little while you didn't manage to access the mutex, then yield. So that if the new mutex was right there available to be accessed, you could access it, but you don't spin for an indefinite amount of time. So the question is, how long do you spin? So we're going to spin for a little while and then yield. Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, exactly. So what you do is you spin for basically as long as a context switch takes. So if you spin for as long as it takes to do a context switch and then do a context switch, if the mutex became immediately available, well, you're only going to wait double what you would have waited. And if in the meantime during that first part where you're spinning it becomes available, you're not waiting at all any longer than you actually have to.

So in both cases, you're waiting at most a factor of two. In one case, you're waiting exactly the right. The other, you can actually wait a factor of two. So this is a classic

amortized kind of argument, that you can amortize the cost of the spinning to the context switch.

So spin until you spend as much time as it would cost for a context switch. Then do the context switch. Yet another voodoo parameter. Yeah, so if the mutex is released while spinning, that's optimal. If the mutex is released after the yield, you're within twice optimal.

Turns out that 2 is not the optimal value. There's a randomized algorithm that makes it e over e minus 1 competitive where e is the base of the natural logarithm. So 2.7 divided by 1.7, which is what? Who's got a calculator? 2.7 divided by 1.7 is-- I should have calculated this out.

AUDIENCE: [INAUDIBLE]

PROFESSOR: It's about 1.6. Good. So it's better than 2. People analyze these things, right? So any questions about implementation of locks? There are many other ways of implementing locks. There are other instructions that people use. They do things like compare-and-swap is another operation that's used.

There are some machines have an operation called load-linked/store-conditional, which is not on the x86 architecture, but it is on other architectures. You'll see a lot of other things of doing some kind of atomic operation to implement a lock. Uniformly, they're expensive compared to register operations in particular or even L1 accesses, typically, in particular. Any questions?

So now that we've decided that we're going to use mutexes, and we understand we're writing non-deterministic code and so forth, well, it turns out there are a host of other system anomalies that occur. So locks are like, they're this really evil mechanism that works really well. It feels so good that nobody wants to stop using it, even though-- but nobody has better ideas.

One of the most interesting ideas in recent memory is the idea of using what's called "transactional memory," which is basically where memory operates like a database transaction. And it's allowed to abort, in which case you roll it back and

retry it. Yet, transactional memory has a host of issues with it, and still people use locks.

So let's talk about some of the bad things that happen when you start doing locks. I'm going to talk about three of them, deadlock, convoying, and contention. So deadlock is probably the most important one, because it has to do with correctness. So you can have coded-- in fact, I've seen people with very fast code that has deadlock potential in it. It's like, if you deadlock, then your average running time is infinite if there's a possibility of a deadlock, right? Because you're averaging infinity with everything else that you might run.

So it's not good to have deadlock in your code, regardless. It's kind of like your code seg faulting. No decent code should seg fault. It should always catch its own errors and terminate gracefully. It shouldn't just seg fault in some circumstance. Similarly, your code should not deadlock.

So here's sort of a classical instance of deadlock. And deadlock typically occurs when you hold more than one lock at a time. So here, this guy is going to grab a lock A, going to grab a lock B, then unlock B, unlock A, and in there do a critical section. Why might I grab two locks? What's the circumstance where I might have code that looked very similar to this? Use case.

AUDIENCE: Two objects?

PROFESSOR: sorry?

AUDIENCE: You need two objects.

PROFESSOR: You need two objects. When might that occur?

AUDIENCE: Account transactions.

PROFESSOR: Yeah, account transactions. That's the classic one. You want to move something from this bank account to that bank account. And you want to make sure that as you're updating it, nothing else is occurring. Another place this comes up all the time

is when you do graph algorithms. You always want to grab the edge and have the two vertices on each end of the edge not move while you do something across the edge. So lots of cases there.

It turns out the order in which you unlock things doesn't matter, because you can always unlock something. You never hold up for unlocking. The problem with deadlock is generally how you acquire locks. So in this example, Thread 2 grabs Lock B, then grabs Lock A. So it might be, for example, that you have some random process that's at the node of a graph.

And now it's going to grab a lock on the other end of an edge. But you might have the guy at the other end grabbing that vertex and then grabbing the one on your end. And that's basically the situation. So what happens is Thread 1 acquires a lock here. Thread 2 acquires that lock. And now which one can go? Neither of them. You've got a deadlock. Ultimate loss of performance.

So it's really a correctness issue. But you can view it, if you really say, oh, correctness, that's for sissies. We do performance. Well, it's still a performance issue, because it's the ultimate loss of performance. In fact, that's probably true of any correctness issue. No, that's not true. Sometimes you just get the wrong number. Here is a correctness issue that your code stops operating.

So there are three conditions that are usually pointed to that you need for deadlock. The first is mutual exclusion. Each thread claims exclusive control over the resources that it holds, in this case, the resources being the locks. So there's got to be some resource that you're grabbing, and that you're the only one who gets to have it. So in this case, it would be the locks.

The second is non-preemption. You don't let go of your resources until you complete your use of them. So that means you can't let go of a lock in a situation. If you're actually able to preempt-- so this piece of code over there has grabbed locks, and now I can come in and take them away, then you may not have a deadlock potential. You may have other issues, but you won't have a deadlock potential.

And the third one is circular waiting. You have a cycle of threads in which each thread is blocked waiting for resources held by the next thread in the cycle. So let me illustrate this with a very famous story that some of you may have seen, because it is so famous. It's the dining philosophers problem.

It's an illustrative story a deadlock that was originally told by Tony Hoare, based on an examination question by Edsger Dijkstra. And the story has been embellished over the years by many retellers. It's one of these things that if you're a computer scientist, you should know this story just because everybody knows this story.

So here's how the story goes, at least my version of it. I get to retell it now. So each of n philosophers needs the two chopsticks on either side of his or her plate to eat the noodles on the plate. So they're not worried about germs here, by the way.

So you have five philosophers in this case sitting around the table. There are five chopsticks between them. In order to eat, they need to grab the two chopsticks on either side. Then they can eat. Then they put them down. So here's what philosopher i does. So in an infinite loop, the philosopher does thinking, because that's what philosophers do.

Then it grabs the lock of chopstick i and grabs the lock of chopstick i plus 1. That's the 1. So if we index them, say, to the left of the plate, this is grabbing the chopstick to the left of your plate. This is grabbing the chopstick to the right of your plate. Then you can eat. Then you release your two chopsticks.

So here, that's the code. And then you go back to thinking. I guess they have no other bodily functions. So the problem is, one day they all pick up their left chopsticks simultaneously. Now they go to look for their right chopstick. It's not there. So what happens? They starve because their code doesn't let them release--there's no preemption, so they can't release the chopstick they've already got.

And we have a circular waiting. They have mutual exclusion. Only one of them can have a chopstick at a time. And we have a circular waiting thing, because everyone is waiting for the philosopher on the right. Is that clear to everybody? That's the

dining philosophers problem.

How do you fix this problem? What are solutions to fixing this problem? The problem being that you'd like them to eat indefinitely.

AUDIENCE: You can index the chopstick and say that [INAUDIBLE].

PROFESSOR: Yeah, you can pick the smaller index first. So in general, that means everybody would grab the one on their left, then the one on their right, except for the guy who's going between 0 and n minus 1. They would do n minus 1 and then 0. They would do n minus 1 first, and then 0. Sorry. They would do 0 first, and then n minus 1. [INAUDIBLE] Let me say that more precisely.

So this is a classic way to prevent deadlock. Suppose that we can linearly order the mutexes in some order so that whenever a thread that holds a mutex Li and attempts to lock another mutex Lj, we have it that Li goes before Lj in the ordering. Then no deadlock can occur.

So always grab the resource so if they can all order the resources-- so they're always grabbing them in some subsequence of this order, so they're always grabbing one that's larger and larger and larger, and you're never going back and grabbing one smaller, than you have no deadlock. Here's why. Suppose you have a cycle of waiting. You have a deadlock has occurred.

Let's look at the thread in the cycle that holds the largest mutex that's called Lmax in the ordering. So whatever is in the ordering. And suppose that it's waiting on a mutex L held by the next threat in the cycle. That's the condition.

Well, then it must be that Lmax falls before L, because we're gathering them always in an increasing order. But that contradicts the fact that Lmax is the largest. So a deadlock cannot occur. Questions? Is this clear?

Who's seen this before? A few people. OK. Is this clear? So if you grab them in increasing order, then there's always some guy that has the largest one, and nobody is holding one larger. So he can always grab the next one.

So in this case of the dining philosophers, what we can do is grab the minimum of i and i plus 1 mod n and then the maximum of i and i plus 1 mod n. That gives us the same two chopsticks. And in fact, for most of the philosophers, it's exactly the same order. But for one guy, it's a different order. It ends up being the guy who would normally have done n minus 1 and 0.

Instead, he does 0, n minus 1. So in some sense, it's like having a left-handed person at the table. You grab your left, then your right, except for one guy does right and then left. And that fixes it, OK? That fixes it. Good.

So that's basically the dining philosophers problem. That's one way of fixing it. There are actually other ways of doing it. One of the problems with this particular solution is you still can have a long chain of waiting.

So there are other schemes that you can use where, for example, if every other one grabs left and then right and then right and then left and then left and then right and then right and left and so forth, you can end up making it so that nobody has to wait to go all the way around the circle. Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Well, that would be a preemption type of thing, where I grab one, and if I didn't get it in time, I release it and then try again. When you have something like that, there's an issue. It's, how do you set the timeout amount? And the second issue that you get into when you do timeouts is, how do you know you don't then repeat exactly the same thing and convert a deadlock situation into a livelock situation?

So a livelock situation is where they're not making progress, but they're all busily working, thinking they're making progress. So you timeout. Let's try again. What makes you think that the guys that are deadlocking aren't going to do exactly the same thing.

AUDIENCE: [INAUDIBLE]

PROFESSOR: And exactly. And in fact, that's actually a workable scheme. And there are schemes

that do it. Now, that's much more complicated. Sometimes has more overhead, especially because things become available. And it's like, no, you're busy raiding some random amount of time before you try again.

So this is, by the way, the protocol that is used on the Ethernet for doing contention resolution. It's what's called "exponential backoff." And various backoff schemes are used in order to allow multiple things acquire mutually-exclusive access to something without having to have a definite ordering. So there are solutions, but they definitely get more heavyweight. It's not lightweight.

Whereas if you can prevent deadlock, that's really good, because you just simply do the natural thing. And that tends to be pretty quick. But yeah, all I'm doing is sort of covering the introduction to all these things. There are books written on this type of subject. Any other questions about dining philosophers and deadlock and so forth?

Now let me tell you how to deadlock Cilk++. So here's a code that will deadlock Cilk++, or has the potential. You might run it a bunch of times, it looks fine. Here's what we've done is main routine spawns foo. Here's foo down here. All foo does is grab a lock and then unlocks it. Empty critical section. It could do something in there. It doesn't matter.

Then the main grabs a lock, does a cilk_sync and then unlocks it. So what can go wrong here? Notice, by the way, this is only one lock, L. There's not two locks. So you can deadlock Cilk by just introducing one lock. So here's sort of what's going on. Let's let this be the main thread and this be foo. And this will represent a lock acquire, and this is a lock release.

So what happens is we perform the lock acquire here in the parent. First, we spawned here, then we acquire the lock here. And now foo tries to get access to the lock, and it can't because why? The main routine has the lock.

Now what happens? The main routine proceeds to the sync, and what does it do at the sync? It waits for all children to be done. And notice now we've created a cycle of waiting, even though we didn't use a lock. Main waits, but foo is never going to

complete, because it's waiting for the main thread to release it, the main strand here to release it, the main function here. Is that clear?

So you can deadlock Cilk too by doing non-deterministic programming. So here's the methodology that will help you not do that. So what's bad here? What's bad is holding the lock across the sync. That's bad. So don't do that. Doctor, my head hurts. Well, stop hitting it. So don't hold mutexes across Cilk syncs. Hold mutexes only within strands, only with serially-executing pieces of code.

Now, it turns out that you can hold it across syncs and so forth, but you have to be careful. And I'm not going to get into the details of how you can do that. If you want to figure that out on your own, that's fine. And then you're welcome to try to do that without deadlocking something. Turns out, basically, if you grab the lock before you do any spawns, and then released it after the Cilk sync, you're OK. You're generally, in that case, OK.

So as always, try to avoid using mutexes, but that's not always possible. In other words, try to do deterministic programming. That helps too. And on your homework, you had an example of where it is that deterministic programming can actually do a pretty good job.

The next anomaly I want to talk about is convoying. Once again, another thing that can happen. This one is actually quite an embarrassment, because the original MIT Cilk system that we built had this bug in it. So we had this bug. So let me show you what it is. So here's the idea. We're using random work-stealing where each thief grabs a mutex on its victim's deck.

So in order to steal from a victim, it grabs a mutex on the victim. And now, once it's got the mutex, it now is in a position to migrate the work that's on that victim to actually steal the work. And you want to do that atomically. You don't want two guys getting in there trying to steal from each other.

So if the victim's deck is empty, the thief releases the mutex and tries again at random. That makes sense. If there's nothing there to be stolen, then just released

the mutex and move on. If the victim's deck contains work, the thief then steals the topmost frame and then releases the mutex. Where's the performance bug here?

AUDIENCE: [INAUDIBLE] trying to steal from each other. Like A steals from B, B steals from C, C steals from D, and they all have locks on each other, and then--

PROFESSOR: No, because in that case, they'll each grab the deck from each other, discover it's empty, and release it. OK, let me show the bug. It is very subtle. As I say, we didn't realize we had this bug until we noticed some codes on which we weren't getting the speedups we were expecting. Let me show you where this bug comes from.

Here's the problem. At the startup, most thieves will quickly converge on the worker P0 containing the initial strand, creating a convoy. So let me show you how that happens. So here we have the startup of our Cilk system where one guy has work, and all these are workers that have no work to do. So what happens? They all try to steal at random.

In this case, we have this guy tries to steal from this fellow, this guy tries to steal from this fellow, et cetera. So of these, this guy, this guy, and that guy all are going to discover there's nothing there to be stolen, and they're going to repeat the process. This guy and this guy, there's going to be some arbitration. And one of them is going to get the lock. Let's assume it's this one here.

So what happens is, this guy gets the lock. What does this guy do? He's going to wait. Because he's trying to acquire the lock. He can't acquire the lock, so he waits.

So then what happens? This guy now wants to steal the work from this fellow. So he steals a little bit of work. Then these guys now, what do they do? They try again. So this guy tries to steal from there, this guy tries to steal from there, this one happens to try to steal there. This one sees there's work there to be done, so what does it do? It waits.

But these guys then try again. Maybe a little bit more stuff is moved. They try again. A little bit more stuff. They try again. But every time one tries and gets stuck on P0 while we're doing that whole transfer, they all are ending up getting stuck waiting for

this guy to finish.

And now, we've got work over here, but how many guys are going to be trying to steal from this guy? None. They're all going to be trying to steal from this one, because they all have done a lock acquisition, and they're sitting there waiting. So this is called convoying, where they all pile up on one thing, and now resolving that convoy.

So this was a bug in startup. Why wasn't Cilk starting up fast? Initially, we just thought, oh, there's system kinds of things going on there. So the work now gets distributed very slowly, because each one is going to serially try to get this work, and they're not going to try to get the work from each other. What you want is that on the second phase, half the guys might start hitting this one.

So you get some kind of exponential distribution of the work in kind of a tree fashion. And that's what theory says would happen. But the theory is usually done without worrying about what happens in the implementation of the lock. What's the fix for this? Yeah?

AUDIENCE: Can you just basically shove-- when you're transferring, you should also say, I have work, so that people [INAUDIBLE] waiting for that guy to [INAUDIBLE].

PROFESSOR: You could do that, but in the meantime, it could be that the attempt to steal goes so much faster than the actual getting of the work, you're still going to get half the guys locked up on this one. And the other half might be locked up on this one. Good idea. What other things can we do?

AUDIENCE: Can you check how many people are waiting on the--

PROFESSOR: Yeah, so the idea is we don't want to use a lock operation. So here's the idea. We use a non-blocking function that's usually called "try_lock," rather than "lock." try_lock attempts to acquire the mutex. If it succeeds, great. It's got it. If it fails, it doesn't go to spin. It simply returns and say, I failed.

It doesn't go to spin or to yield or anything. It just says, oh, I failed, and tells that

back to the user. But it doesn't attempt to block. So with try_lock now, what can these other processors do? They do a try_lock-- yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Exactly. Instead of waiting there on the guy that they fail on, they pick another random one to steal from. So they'll just continually try to get it. If they get it, then they can do their operation. If they don't get it, they just look elsewhere for work. So that's what it does. It just tries to steal again at random, rather than blocking.

And that gets rid of this convoying problem. As I say, dangerous programming, because we didn't even know we had a problem. Just our code was slower than it could have been. Questions about convoying?

So try_lock is actually a very convenient thing to use. So in many cases, you may find that, hey, rather than waiting on something with nothing to do, let me go see if there's something else I can do in the meantime.

Contention. So here's an example of a code where I want to add up some function of the elements of some array. So here I've got a value of n, which is a million. And I have a type X. So we have a compute function, which takes a pointer to a-- did I do this right? To value V. So anyway, my C++ is not as good as my C, and for those who don't know, my C isn't very good.

So anyway, we have an array of type X of n elements. And what I do is I set result to be 0, and then I have a loop here which basically goes and adds into result the result of computing on each element of the array. And then it outputs the result. Does everybody understand what's going on in the code? It's basically compute on every element in the array, take the result, add all those results together.

We want to parallelize this. So let's parallelize that. What looks like the best opportunity for parallelizing? Yeah, we go after the for and make it be a cilk_for. Let's add all those guys up. And what's the problem with that? We get a race. What's the race on?

AUDIENCE: Result.

PROFESSOR: Result. They're all updating result in parallel. Oh, I know how to resolve a race. Let's just put a lock around it. So here we have the race. First, let's analyze this. So the work here is order n. What is the span?

AUDIENCE: Log n.

PROFESSOR: Yeah, the span is log n for the control of the stuff here, because this is all constant time. So the running time here is order n over P plus log n. If you remember the greedy scheduling, it's going to be something like this, because this is the work over P plus the span. So we expect that if n over P is big compared to log n, we're going to do pretty well, because we have parallelism over log n.

So let's fix this bug. So this is fast code, but it's incorrect code. So let's fix it by getting rid of this race. So what we'll do is we'll put a lock before. We'll introduce a mutex L, and we'll lock L before we add to the result, and then we'll unlock it.

So first of all, this is a bad way to do it, because what I really should do is first compute the result of my array and then lock, add it to the result, and then unlock so that we lessen the time that I'm holding the lock in each iteration. Nevertheless, this is still a lousy piece of code. Why's that?

AUDIENCE: It's still serialized.

PROFESSOR: Yeah, it's serialized. Every update to result here has to go on serially. They're n accesses. They're all going to go one at a time. So my running time, instead of being n over log n, is going to be something like order n.

Believe me, I have seen many people write code where they essentially do exactly this. They take something, they make it parallel, they have a race bug, they fix it with a mutex. Bad idea, because then we end up with contention on this mutex. What's the right way to parallelize this? Yeah?

AUDIENCE: Maybe you could have each [INAUDIBLE] have result as an array and have each [INAUDIBLE] one place [INAUDIBLE]. And then at the end, some of all the--

PROFESSOR: But won't that be n elements to sum up?

AUDIENCE: [INAUDIBLE]

AUDIENCE: So basically, have, say. Eight results, instead of having--

PROFESSOR: For each thread. Good. So that each one could keep it local to its own thread. Now, of course, that involves me knowing how many processors I'm running on. So now, if that number changes or whatever-- there's a way of doing it completely processor obliviously.

AUDIENCE: Divide and conquer.

PROFESSOR: Yeah, do divide and conquer. Add up recursively the first half of the elements, add up the second half of the elements, and add them together. Next time, we're going to see yet another mechanism for doing that, which gets the kind of performance that you're mentioning but without having to rewrite the For loop as divide and conquer. We'll see that next time.

So in this case, we have lock contention that takes away our parallelism. Unfortunately, very little is known about lock contention. The greedy scheduler, you can show that it achieves T1 over P plus T infinity plus B where B is the bondage, that is, if you add the total time of all critical sections. That's a lousy bound, because it says, even if they're locked by different locks, you still add up the total time of all the critical sections.

And generally, although you can improve this in special cases, the general theory for understanding contention is not understood very well. And this upper bound is weak, but little is known about lock contention. Very little is known about lock contention.

So to conclude, always write deterministic programs, unless you can't. Always write deterministic programs, unless you can't. Great.