[1] We have done a lot of programming and proving without concurrency. In this lecture, I want to show you how concurrency can be introduced automatically. We can write our programs as usual, and let the compiler transform them for concurrent execution. The purpose is to speed up the execution without changing the results. [2] Here's an example. Three assignments, in sequence, with no concurrency. Let me [3] draw the execution like this. The first two assignments both assign to x, so they cannot be executed concurrently. But the last 2 assignments work with different variables, so we can [4] make them concurrent. [5] Here's the picture. Now the first assignment, x gets y, is followed immediately by both of the other assignments. In particular, it's followed immediately by z gets y. So we have those 2 assignments in sequence, and they can be concurrent. So [6] here's the picture. And as a program it [7] looks like this. The point is that if you have 2 things in sequence next to each other that can be made parallel, when you put them in parallel, you might enable further changes from sequential to parallel execution.

I said this transformation can be done automatically, so [8] here are the rules. [9] Whenever two programs occur in sequence, and neither assigns to a variable appearing in the other, they can be placed in parallel. For example, x gets z followed by y gets z. The first assigns to x and there's no x in the second. The second assigns to y and there's no y in the first. So put them in parallel. It doesn't matter that z appears in both because it's not being assigned to. [10] Here's a more general rule. Whenever two programs occur in sequence, and neither assigns to a variable *assigned* in the other, and no variable assigned in the first appears in the second, they can be placed in parallel. In the example, x gets y and then y gets z. They assign to different variables. And no variable assigned in the first, that's x, appears in the second. So they can be put in parallel. But secretly, [11] a copy must be made of the initial value of any variable appearing in the first and assigned in the second, that's y. This looks like a stupid thing to do because it doesn't improve the execution time in this example. But you have to imagine, instead of 2 simple assignments, you have two complicated programs. Then you might really save a lot of time by making a copy of some variables and then running the two programs concurrently.

Let's look at a better example. [12] The buffer. There are two programs, produce and consume. Produce might do lots of things, but the important thing for us is that it assigns something to variable b, which is the buffer. And consume might do lots, but for us the main thing is it uses the value of variable b. I've written it as simply as I can, but imagine that b is some big data structure, and maybe produce reads a lot of files and fills in values in the data structure. And consume uses the values in the data structure to calculate things and maybe print out a report. They work under [13] control of a program called control, which says first produce, which fills the buffer, then consume, which uses the buffer, then repeat, forever. [14] Here's the picture, with P standing for produce and C standing for consume. Produce and consume might each take a long time, and we want to save time by running them concurrently. Well we can't, because the rule said "no variable assigned in the first appears in the second". Obviously consume can't use b until after b is assigned. So here's what we do. [15] We [16] unroll the loop once. We start with produce, and then newcontrol starts with consume, followed by produce, and then repeat. Now it follows the rule, so we can [17] put them in parallel. The implementation has to make a copy of b, and consume uses the copy. Or, we could [18] do that ourselves. -- [19] Here's the picture. I've used the letter big B for the copying assignment c gets b. Now all the produces and consumes are in parallel, except for the first produce. If produce takes longer than consume, then this computation just takes the time to produce, and the consumes are free. If consume takes longer, then the produces are free.

The program we started with had a one-place buffer, variable b. This program has a 2 place buffer, variables b and c. The [20] next one is an infinite buffer. This time, variable b is an array. And we have two index variables, r and w, for read and write. Both w and r have

to be initialized to 0. Produce writes into the buffer at position w and then increases w. Consume reads from the buffer at position r and then increases r. The execution can be like [21] this. The [22] first produce has to precede the first consume because the first produce writes to b 0 and the first consume reads b 0. But the [23] second produce and the first consume can be in parallel. As a matter of fact, [24] any other produce can be in parallel with the first consume. The producer can get arbitrarily far ahead of the consumer. Whenever w and r are unequal, produce and consume can be in parallel. Whenever consume catches up to produce, w and r are equal, and they cannot be in parallel. That's the diagonal arrows in the picture, and that prevents consume from overtaking produce.

That was an infinite buffer. A [25] finite buffer is the same, except that adding 1 to w and r is done modulo the buffer size. If consume catches up to produce, then w equals r and that prevents concurrency, the same as before. But now, produce can get only n steps ahead of consume, where n is the buffer size, and then again w equals r and that prevents concurrency. The [26] picture is almost the same as before, but the new arrows show that when the buffer is full, produce must wait for consume. This waiting, this synchronization, does not need any programming on our part. It's already there in the program. It's the fact that when w equals r, produce assigns to a variable that consume uses. It's easy analysis for a compiler. Synchronization is just what's left of the sequencing in the program after the opportunities for concurrency have been exploited.

By the way, I'd like to mention that large buffers are not as useful as some people seem to think. If consume is the faster process, then it always has to wait for the next produce, and the whole program runs at the speed of produce, just like the two-place buffer. If produce is faster, then it fills up the buffer, and after that it has to wait for consume, and the whole program runs at the speed of consume, just like the two-place buffer. The only way a buffer larger than 2 places helps is if consume and produce run at the same speed on average, but with some variation. Then the buffer smoothes out the variation. That's not a lot of help, and it doesn't happen very often.

The [27] next example is insertion sort. Define sort, with parameter n, to mean that list L is sorted up to index n. For all i and j from 0 to n, if i is less than or equal to j, then L i is less than or equal to L j. So the problem is [28] sort prime of length of L. In the end, we want L to be sorted right to its end. And we can refine that as [29] sort 0 implies sort prime of length of L, because sort 0 is trivially true. And this implication is [30] exactly the right form for refinement by a for-loop. And this makes the problem easier because [31] all we have to do now is get from sort n to sort prime of n plus 1. Given that the list is sorted up to index n, make it be sorted up to index n plus 1. Maybe a [32] picture will help here. [33] If n equals 0, meaning that the list is sorted up to [34] here, we want it to be sorted to [35] here. Well, a one item list cannot be out of order, so there's [36] nothing to be done. [37] Else it's already sorted up to somewhere, and we have to include one more item. Well [38] maybe the next item just happens to be in order, so there's nothing to be done. If not [39], well, in the picture n is 3. So we can't just move the arrow over L 3 because L 3 isn't the largest item so far. That means L 2 is the largest item so far, and it should be where L 3 is. So [40] swap L 2 and L 3, where [41] swap is defined as usual. Now L 3 is right, but L 2 might not be. The list is still sorted up to index 2, but L 2 might not be the next bigger item. The problem is [42] given the list is sorted to n minus 1, make it be sorted to n. And that's the specification we are refining, with a change of parameter. So we're done. But the point of the program in this lecture is to see what concurrency we can get. [43] Here's the execution without any concurrency, in the worst case where the list was sorted backwards to begin with. Each C is a comparison of 2 adjacent items, and each S is a swap of 2 adjacent items. And you can see from the triangular area being filled, that the execution time is approximately n squared over 2, where n is the length of the list. We give the program to our clever compiler, and it notices that [44] two swaps can be done concurrently if the items being swapped by one

don't overlap with the items being swapped by the other. Likewise a [45] swap and a comparison can be done concurrently if neither of the items being swapped is an item being compared. And [46] Any 2 comparisons can be done concurrently because that doesn't involve any assignments. The picture can be transformed, little by little, to [47] this picture. Its execution time is approximately 2 times n. We write an n squared sorting program, and a clever compiler turns it into a linear sorting program.

The [48] last example of this lecture is the dining philosophers. It's a standard example in operating system textbooks. In case you don't know the problem, I'll explain it. There are 5 philosophers sitting around a round table. They are the yellow circles. In the center of the table is a large bowl of noodles. Actually, it has to be an infinite bowl of noodles. In between each 2 philosophers there's a chopstick. Those are the little lines. Philosophers think a lot. Sometimes a philosopher gets hungry, and then picks up the chopsticks on each side, and then eats for a while. When the philosopher has eaten enough, the philosopher puts down the chopsticks, one on each side for the neighboring philosophers to use, and then thinks for a while, before becoming hungry again. Obviously these philosophers are not too concerned with hygiene. Sometimes two or more philosophers might get hungry at the same time, or their eating might overlap. If one of the chopsticks needed is in use by the neighboring philosopher, they just have to wait until it's free again. There's a small chance that they might all get hungry at the same time, and they all pick up their left chopstick, and they look for their right chopstick, but it isn't there, so they all decide to hang on to their left chopstick and wait for their right chopstick. That's called deadlock, and we want to avoid that situation. There's also the possibility that a philosopher gets hungry, and notices that their left chopstick is in use, and decides not to pick up the right chopstick until the left one comes free. That should help avoid deadlock. But when the left one comes free, the right one is in use. And later when the right one is free, the left one is in use, and so on. That philosopher would starve, and we want to avoid that too. [49] Here's the usual solution you see in operating system textbooks. [50] Life is the five philosophers all living concurrently. [51] Each philosopher thinks for a while, then picks up their left and right chopsticks, then eats, then puts down their left and right chopsticks, and then repeats the cycle. I'm using a circle-plus to mean addition modulo 5. [52] Picking up a chopstick and putting down a chopstick mean assigning a value to a variable. [53] I don't quite know what to write for eating and thinking, except that eating uses those variables, and thinking doesn't. So that's the life of the philosophers. But [54] this program doesn't work. It's just plain wrong. You cannot put any pair of neighbor philosophers in parallel because they both assign and use the same chopstick variables. One philosopher might be picking up a chopstick that his neighbor is using to eat with. And this program can deadlock. These errors have to be corrected by adding mutual exclusion devices, synchronization, and deadlock avoidance or deadlock detection schemes. That's complicated and hard. But it's easy to just not make the mistake. Put in parallel only those things that follow the rules I gave earlier this lecture.

[55] Here is a program without concurrency. Well, it's not yet a program, because it starts with a [56] disjunction. It says choose any one philosopher, then repeat. Refining this disjunction is where we make sure each philosopher keeps getting turns. But since we cannot observe whether each philosopher keeps getting turns, that's a worthless requirement. Maybe we use probability distributions from Chapter 5 to decide the order in which philosophers are chosen. [57] The chosen philosopher thinks, gets hungry and eats. So the philosophers eat one at a time, in some order. So that's the life of these philosophers, eating one at a time. Now the clever compiler has a look, and it sees [58] all these facts. I'm just listing the concurrency that results from the rule that doesn't involve copying any initial values. The compiler sees [59] that any two philosophers can think at the same time. [60] And since thinking doesn't involve chopsticks, thinking can happen at the same time as

picking up or putting down a chopstick, or eating. [61] Different chopsticks can be picked up at the same time. We can apply this transformation right away. It sees [62] that picking up one chopstick and by putting down another can be transformed into a parallel composition. We can't apply this transformation right away, but as we apply some transformations, others become enabled, so we have to make a note of all possible transformations. And [63] putting down different chopsticks can be done concurrently. [64] Eating and picking up or putting down can be done at the same time as long as the chopstick being picked up or put down is not one of the chopsticks being used for eating. And finally, [65] two philosophers can eat at the same time as long as they are not immediate neighbors. With these transformations, we get all the possible concurrency there is that doesn't lead to deadlock. The compiler sees all these things just from one of the rules I gave earlier this lecture about when things can be done concurrently.