

MIT OpenCourseWare  
<http://ocw.mit.edu>

Multicore Programming Primer, IAP 2007

Please use the following citation format:

Rabbah, Rodric and Saman Amarasinghe, *Multicore Programming Primer, IAP 2007*. (Massachusetts Institute of Technology: MIT OpenCourseWare).  
<http://ocw.mit.edu> (accessed MM DD, YYYY). License: Creative Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit:  
<http://ocw.mit.edu/terms>



MIT OpenCourseWare  
<http://ocw.mit.edu>

Multicore Programming Primer, IAP 2007  
Transcript – Lecture 4

The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at [ocw.mit.edu](http://ocw.mit.edu).

PROFESSOR: So, the next part, today's going to be about concurrent programming. So in this lecture we are going to study concurrent programming with the emphasis for correctness of programs. Because parallel programs will have the same correctness issues. So, if you want to get parallel, you'd better get the concurrency right first.

What we're also going to do is start with a much simpler machine model. In a program where we are going to use Java, because I think a lot of people understand Java. Also, we are going to do some very simple shared [UNINTELLIGIBLE] machine abstraction. I'm not going to even talk about any machine anymore. I'm just going to talk about concurrent programming here. You need to get through this one before you can start to dig in deep into the next level.

So in the next lecture, we will switch from Java to C-C , I guess using MPI primitives in here. We'll start moving into parallelism with emphasis on performance. And, of course, you have to get correctness, that's given, but we'll start looking at performance in there. And we'll start using the distributive memory machine, all the notions and details of Cell, so we'll just kind of go down and down in that direction.

So, what's concurrency? Sequential program is -- because sequential program opposite. It's basically single thread of execution, with is a good one. Finish that, go to the next, go to the next. That's a very simple abstract model that for about 35 years, 40 years, none of the machines were actually following, that they had things in the back that actually had some parallelism.

A concurrent program is the [UNINTELLIGIBLE PHRASE] because it's a collection of autonomous sequential threads executing logically in parallel.

So you can execute this thing either multi-programming, so we can multiplex different parts on multiprocessing. Well, multiprocessing basically has [UNINTELLIGIBLE] starting on different machines. You can distribute, you can actually send it to different places. Of course, you have to deal with memory issues.

So, concurrency's not only parallel systems. So you can do interleaved concurrency. You can have logically parallel, but you run Thread A for a while, contacts with Thread B for a while, contacts with Thread C, so you can have multiple threads on the same machine running. Or you can actually have running parallel. You can have three different machines running, A, B and C all the time. So you can have both in there. But logically you should not see a difference except for performance and stuff like that.



So what I'm going to do is do a bunch of examples. Can you read this? Let's start with a bank. So you have a bank account. So in Java you just basically have ID, password, and balance, and you have some way to construct this object in here. And you can ask, and see the password is correct. You can get the balance. And you can post the balance. So that's a very simple account object. If you have a bank, you have a bunch of accounts in a hash map, and you create the hash map in here. Then you can basically [? figure out ?] the bank, you actually create a bank in here, and you can get an account, given an ID.

Now, assume you want to build an ATM. How do you build an ATM? So, you have a bank -- you need a bank in here, and here's some input and output streams in here. When you start the ATM, you will set up these input and output streams in here. In the main function, what you'll do is, you get a bank, create where the input streams are coming from. Create output goes standard -- system output goes there. And create an ATM in here, and you will make the ATM run. So how do you run the ATM?

So, what happens in run is, you run forever? ATM doesn't stop any time. What you can do is you can ask when somebody walks into the ATM, you can say what's the account ID. Type the account ID. You can get that account, so of course, if the account already is wrong it says, throw exception. You can say OK, what's the password, get the password. You take the password, if it's wrong you throw exception. Then you can say, here's your balance today. What do you want to do? If you want to withdraw or deposit. If you want to withdraw you can do a minus number, if you want to deposit it will be a plus number. Then you can post that into your balance.

Everybody got the thing for ATMs? So, assume activity trace. So somebody comes and gives the account ID, at least gives the password, and say that they have \$100,000 and say how much you withdraw, \$200 withdraw. And you get the balance in here. Looks nice. It works.

So I need to run multiple ATMs. Assume I am in a place that I actually want to put two ATMs or four ATMs next to each other. So how am I going to do that?

So, in order to do that, there's concurrency in Java. So one way to get Java concurrency is you can extend this class thread and define a method run. Or you have interface called [? Runnable, ?] that you can basically use that interface and has estimated run. Then when you have made that run and when [UNINTELLIGIBLE PHRASE] start, that will get started. Very simple way to do that. Let me give you an example.

Little bit of a digression. Why do you want concurrent programming? A lot of times, natural application structure is not sequential. The world is not sequential. And then try to sequentialize the world sometimes means it's much more complicated [UNINTELLIGIBLE] So sometimes it's natural to do things in parallel. A lot of times the sequentiality's an artifact of the programming language, because we use a language like that. Sometimes doing things in parallel ways, you can really improve things like throughput and responsiveness. If you are doing IO, if you're doing sequential programming [UNINTELLIGIBLE PHRASE] you're just twiddling your thumb waiting for the IO to come back. In parallel things actually, you can do parallel IO and you can do a lot cool stuff in here.



Of course, in this class, if you are multicore and multiprocessor multicore, you can get parallel executions. So there are more than one [UNINTELLIGIBLE PHRASE]. Also, if you are building a very large distributed system, concurrent programming is, you had to deal with, especially dealing with things like client-server type of applications.

So here's our original ATMs. So to go to multiple ATMs, I am doing a few changes. I'll go back and forth a few times. So the first thing I have done is I have sett here number of ATMs to be four. Can you really read it from back there?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: OK, good. Then what I have done is, in here, I did four ATMs here, and then I put it in a loop to create this ATM, so I created four ATMs in here and start four ATMs, basically. Then of course I extended these ATMs so now we will extend [? up a thread. ?] And I haven't started that. And the ATMs [UNINTELLIGIBLE] ATMs, so it's great. So now what happens is I assume now there's two guys going, both ATMs, at least [UNINTELLIGIBLE] been. Then enter the account and [UNINTELLIGIBLE PHRASE], that's works really well. No problem. So we have two ATMs, two people actually went on parallel. One then deposited some money, other one took money, great.

Now, as MIT students, they want to do something, they can hack it. So, [UNINTELLIGIBLE] basically [UNINTELLIGIBLE] went [UNINTELLIGIBLE], and basically what [UNINTELLIGIBLE PHRASE] enter the password, and they said I want to get \$100. I would get \$90, basically. So he had \$100 in his account. Then what he got was, so he actually managed to get \$180 out of an account that had \$100. This is not a good ATM, at least from the bank's perspective. So what went wrong?

If you look at what happened in activity trace, so we print 100 in here. And then you said, you want to read this value, you both entered 90, right here. And this account balance [UNINTELLIGIBLE PHRASE] because the account balance was 100. You saw also the account balance [UNINTELLIGIBLE PHRASE], yes, it is [UNINTELLIGIBLE]. Then each went post, it went to 10 -- this also did a post of the same time, result came both 10. How could this happen?

So that way it can happen is, so in the ATM, the [UNINTELLIGIBLE PHRASE], what happens is  $v$  is minus 90, and this post [UNINTELLIGIBLE] also when you start a  $v$  it's minus 90. Then you treat the balance as 100. So in this interleaving, and so it is the plus, you get 10. Also, before you write it out, you read the balance in the other interleaving, you've got the balance as 100, and you do the plus as 10. So it destroyed the balance, now balance became 10, and also this guy also wrote the balance -- it doesn't matter, it got 10 updated twice, and that's it. So you can have interleaving in here, that actually did something that's not a signature program. And you're in big trouble. So in order to get out of that, problem is all interleaving of threads are not acceptable and current. What you want is some kind of a sequential-looking performance, even though you'd get parallel, you don't want to do all these interleavings in here.

So in order to do that, Java provides this synchronization mechanism. That's just strict interleaving. So, what synchronizations do is, they ensure safety for shared updates. So if you're sharing something, so it avoids races, basically. It avoids this old interleaving ordering here. Also, it allows you to coordinate actions among shared



space, basically. Because at some point people have to coordinate and take that parallel computation. With notification you can do that.

So, when multiple threads access the shared resource, simultaneously, it's safe only if all accesses have no effect on the resource. Basically, we're reading variables. But everybody can read the same variable, because you're not changing anything. I can do that. Or all accesses are idempotent. So you can say, we can do that. Or only one access at a time. Which is called mutual exclusion. So in this case we are changing something. It's not that important. So we have to actually do mutual exclusion.

So here's a way to look at safety problems. Here might be an algorithm that you and your roommate have. So you arrive home, look in the fridge, no milk. Leave for grocery, arrive at grocery, buy milk and arrive at home. The minute you leave for grocery, your roommate arrives and do this. Then what do you have is you have too much milk.

So here's the problem in a little bit more abstract sense. And you need a way to synchronize this. So how about this. No milk and no note. So you leave a note before you actually leave the house and buy milk, and then you come back and remove the note. Does this work?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: I mean here also you can do that, no milk and no note. So both are started. We would leave a note, and these things can happen at the same time. There's a little bit of things saying OK, why didn't you see your roommate. They go buy milk, he goes to buy milk and you have too much milk too.

So the way to do this in Java is this notion of critical section. Critical section is where only one thread can be in it at a given time. The way you can do it with Java is, you can put synchronized in front of the method. And you do that method, only one person can be executing that method at any one time. So in here I would say get balance and post so you can synchronize.

So when you do that, what happens is -- so in here you read. No problem, you can do this parallel. You can do this in parallel. And then you say, first take out post in here. And this [? takes out ?] post. Because of synchronization these things can't have an order, because this has to happen in some order. Either this happens first and this has to finish before this one. At that point you can actually -- what happened now? Are we happy?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. At least banks realize -- bank's book is correct. Because it realizes, here is more money. But actually it let you take more money than your account had. So at least it got that value right. But what happened was, why is this happening now?

AUDIENCE: You want the check covered.

PROFESSOR: OK, you want to check also. Good. So the key thing is, here we didn't check. So you have a negative bank balance happening. So this is a problem with atomacity. Because synchronized methods execute the body at atomic units. So



when that happens, the entire thing of body happens without anybody else [? modifying. ?] That's the only thing that's happening at any given time. The code read [UNINTELLIGIBLE] you chose is probably too small in this case. What we need to do is, we need to basically have synchronizing not on the method but a lot more [UNINTELLIGIBLE] in there, so you have to do block synchronization. So synchronized keywords actually work like this too. You can say instead of doing a method, you can just synchronize account and all those things happen synchronously within that block.

So, now what we have done is we have built a bigger atomic unit. So here's the programming here. So here's the synchronized unit in here. So what we did was, we say instead of these synchronized and these synchronize separately, both of these computations have to happen atomically. So when I check our bank balance, we can't do anything else.

So now what happens? So yeah, in this situation you're reading, reading, and you get synchronized account in here, and I do account balance plus [? well ?] and post the account. So in here I go to 10, I do that. If I start the other one here, I have to wait till that entire synchronization is over before I do that. Of course, I don't have enough balance, so I throw exception.

Are we still happy? Is there issue on this one? I mean, I guess -- assume you can do something clever, but I haven't done that. But there's one issue in here. Which, when you start it's just, balance is 100. So in this one, say balance is 100. You go type it and then voila, you type it and then, sorry, I don't have money. So that's not nice, because if you've got the balance you should be able to get that. So how we deal with that?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: So that's probably the best solution to that because we can only log into one. But in this example I assume what we are doing. How can we deal with this one?

AUDIENCE: There are two ways of doing it. One is to put the whole thing [INAUDIBLE PHRASE] section. The other way is to notify somebody that the [UNINTELLIGIBLE PHRASE].

PROFESSOR: [UNINTELLIGIBLE PHRASE]. So what I can do is I can say OK, wait a minute. I am actually going to make the critical section even bigger. So now I print the balance before I do that. So the entire thing is critical section. I print the balance off and then go ahead and withdraw that. So what might happen in this case?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: Yeah. That's the issue of a little bit of waiting. So what happens is, in here. You do this one, and you do synchronized account. And you put the balance and other one do synchronized and you ask the question. In here. And you start thinking. We can start thinking that my machine is not responsive, it's just waiting for the critical section started. [UNINTELLIGIBLE] and you have this [? IOUN ?] sitting in the middle. That's not good either. So he has a performance issue. So that's not a good way of doing that. So you don't get any response in here. So you



can make this atomic [? radius ?] but there's a price you'll pay by making it [UNINTELLIGIBLE PHRASE] large.

So here's another thing we want to do. I want to do something that can transfer account balance from one account to another. So I might do that if I have a method in here to transfer [UNINTELLIGIBLE] account, this amount. So what I do is, I synchronize from account. I say, I get balance in here. If the balance is available, I can synchronize the two accounts and force it there. See any problems?

So let's see what happens. So assume I want to transfer 10 to Ben's account and Ben wants to transfer 20 to Alyssa's account. So what happens is, this goes -- get the value in here, and you synchronize to two and say OK, great. Now what happens is, in here, in from, I am holding a Alyssa's account. There I am holding Ben's account. Now, inside I want to synchronize for Alyssa. And I'm still [UNINTELLIGIBLE] when I -- wait until Alyssa got released. And he says I want to wait till Ben got released. And nobody's going to release, and you're hung. You are in what you call a deadlock situation. That's a deadlock. So you have to be very careful when you do synchronizing. If you do multiple synchronization, the easiest thing you can do is, you do it in some order. And end up in a deadlock situation. This is a very common way of parallel programs doing that.

So how to avoid deadlock? Because deadlock is, there's a cycle in locking graph. So somebody's going to lock somebody, he's going to lock that person, and we have a cycle. You can end up in deadlock situation.

So standard solution for that is, you take locks in some kind of canonical order. You don't take in arbitrary order. So it's some kind of a -- you have some base in, OK, if you are taking this lock, you have to have, after that, you can take a higher order lock. So you have to have some kind of order in here. Acquire in increasing order and release in decreasing order. So you have some kind of force in here.

This ensures deadlock freedom most of the time, but it's not that easy to do a lot of the time. Because your program might not fit into this nice ordering a lot of times, and then sometimes you realize that you had locked something and at that time it's too late when you realize it. And then it has a different order. So this is, you have to sometime do some changes to basically make the program work like this.

So in here, what you can do is, in the program you can associate some kind of a rank, and when you put in account, you put the rank to the account number. So you have some kind of ordering in here. Then what you have is, you always get the first, highest rank one before you go the next one. So there's some ordering in here. So at least then we'll be at least forced into some ordering in here.

AUDIENCE: Is there a way of [UNINTELLIGIBLE PHRASE] deadlock [INAUDIBLE PHRASE].

PROFESSOR: Not statically. Because most of the time that means you have to know all the possible control profile, to do that. And, for example, there are some tools that can -- because you might know that, for example, assume you are trying to enforce some ordering of locks. But it's not the software, it's the locking software, that doesn't know about those. You can actually write a locking software that will tell you, like look, you are trying to acquire locking out of order, out of this locking



order. Most of the time you might be OK because it might not hit, but [? we assume ?] that if you are doing unsafe thing that might work, so.

So you can put some dynamic checks that might warn you that you might be in a situation, but it doesn't guarantee you. So deadlock is something, you have to basically -- there's no nice tools for. Basically, it's almost a software [? methodology. ?] So, for example, you can impose a software methodology to say, I'm following this convention and that will guarantee me deadlock freedom. So one good convention is this, basically some order in here.

So, another interesting thing and hard thing is race conditions. These are non-deterministic timing dependent, and cause data corruption, crashes that are impossible to detect. So the problem with race conditions is the minute you put your debug, or put any debugging things, race conditions goes away. It comes back when you are in it all and you're debugging [UNINTELLIGIBLE PHRASE]. It happens again because it's basically an independent thing. In fact, I have this interesting experience with myself.

A long time ago I was working at Microsoft and I worked two summers. In one summer I was working on their LAN manager and network manager, and there's a bug that after you run the network manager for some time it just freezes. That's not a nice behavior to have if you are running your network. That bug lasted the entire year. And at the end they had, I think, a \$2,000 bounty on that bug. Because the minute you do any instrumentation, the [? bug isn't ?] [UNINTELLIGIBLE] When you have more instrumentation and have 100 machines running, heavily, hitting another machine. Once in a while voila. It freezes. And you have no idea why it happened. That was so hard to debug because there's nothing you could do, because any time you do any changes, the bug goes away. You had to be very careful because these things are not easy to find, and happen intermittently. And very hard to debug. So having good discipline and good design really helps to get rid of it. These are not something you can go through like program debugs, it [UNINTELLIGIBLE] cycle. If you read that cycle, it's a very slow cycle. The best way to do that is get the design right first.

So what's a data race? So I assume I had this program like that. So I read [? hit ?] in there, and then I modify and write in this. This doesn't have to be in two statements. If we [UNINTELLIGIBLE] same statement, the compiler might put it in register, read, update and modify and write. So it might just look hits equals hits plus 1 and hits equals hits plus 1 on the cycle. Doesn't have to [? have temporary. ?] and in your call. Because the compiler puts a [? temporary ?] in there.

And if you execute like this you're happy. But if you get excluded in this order, I don't get at it two times, I only get it because I read hit the order given values. This adds once and writes. And this also adds one to [? the ordinary value ?] and write. So I only get it increased by one and you are in a bad situation.

The problems with data races is this non-determinism. We ensured [UNINTELLIGIBLE] that this mutual exclusion. So if you have same data access, make sure that they are in the mutual exclude region. You can basically see that it has access to old objects.

Before you go there, one interesting thing is this is just a problem with all parallel programs. So at the beginning you say OK, I'm going to have this nice mutual



excluded, lock ordered program. You write this. It worked correctly, beautifully, but run dog slow because now we are huge critical sections. Everybody's waiting in data and then someone says I want to run fast. I think I don't need this lock.

It doesn't seem to be, so keep removing locks, making critical sections smaller and stuff like that. That's where all the problems start cropping up, because all this nice design goes to the dogs when you have performance issues. So when you realize that, you want to write this nice program, nice large critical sections, stuff like that. The programs will work correctly. But run like a dog because now it's sequential in many cases because you are doing this. Then you go and say OK, I want to run parallel. Eh, this is OK. That's when problems start creeping up. So make sure that when you get a discipline, as you can go into the performance improvement but you still maintain at least some part of discipline. That's the hard thing.

So I want to switch gears a little bit to talk about a classic problem. It's called dining philosophers problem. So, there are five philosophers sitting around a table. Between each of the philosophers there's a chopstick. So each philosopher do two things. He thinks -- he or she thinks or he or she eats. So the philosopher thinks for a while. And then the philosopher is hungry. She stops thinking and she picks up a left and right chopstick, eats, and puts the chopsticks down. He cannot eat until they have both chopsticks right in hand because you can't eat with one chopstick. So you have to wait until you get both chopsticks. When you are done, you put the chopsticks down. Then after you're done, you go back to thinking again for a while and come back to eating.

That's the classic problem. So how to write that, record that? You can have philosopher extend thread, and [UNINTELLIGIBLE PHRASE] philosopher you have a chopstick in here, and instead of philosopher buy left and right chopstick. Then what you do is you create a number of philosophers and you get a new chopstick and start to the left and you go to the other philosophers assigning left and right chopsticks in here, and then you start the philosophers going. So you just set up a chopstick [? on it, ?] and then you share the chopstick and do that.

So here is what a [UNINTELLIGIBLE] philosopher does. So I am here, I'm taking my left chopstick, I'm taking my right chopstick and I'm going to eat and I'm done eating and I'm putting down there. What will happen in this one?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: In what situation, [UNINTELLIGIBLE PHRASE]. [UNINTELLIGIBLE PHRASE], but right technical though is different.

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: You end up in a deadlock because [UNINTELLIGIBLE PHRASE] we will pick up the left chopstick suddenly, and they all try to take the right chopstick. There's no right chopstick and nobody has right chopstick and everybody waiting for somebody to drop the chopstick, that's not going to happen. So you have a problem. Second way to solve that is this, and you say OK. The problem is everybody trying to pick up this chopstick. I will put unique variable table, unique object table. If anybody want to eat, I need to own the table. What will this do?

AUDIENCE: [INAUDIBLE PHRASE].



AUDIENCE: It prevents two people who wouldn't normally interact from eating at the same table.

PROFESSOR: Yes. So what happens is only one person can eat at a time. Which works perfectly, beautifully, sequential. So, you wonder if one philosopher eating, the person or [? posit ?] can eat. But you're not allowed to because the chopstick is there. So one way of doing that is sequentialized large regions, with putting these critical sections in there. This works. Not greatly, but it will work.

Another thing is, of course, what I point out to have some kind of ordering. So you put some position ordering and saying if you are sitting in even position, you're the first to pick the left one, if you are putting an odd position, you're supposed to pick the right one. So in some sense, it got [UNINTELLIGIBLE PHRASE] go here, the person go here, so only one can get that so you don't have ordering. At least between those two you can maintain that. So you can do something but you have to figure out what the right ordering is here. This is not a linear list, linear ordering for this circuit. But you can copy ordering and say OK, look, if you do that this new way, you can run into a deadlock situation.

There are a lot of types of synchronizations in Java, and then tomorrow you learn more different type of synchronization with available using [UNINTELLIGIBLE PHRASE], so using MPI. But there are a lot of potential problems you are worried about. Deadlock you have to worry about. Two or more threads stop, wait for each other forever. Livelock. What livelock means is two or more threads basically trying to do something but never made progress. So good example.

So assume I go -- it's like sometimes you try to cross each other on the road and you go into them and say oops, or you both say oops, sorry, [UNINTELLIGIBLE] You get into a situation that you try to go something, both you start to move a little bit and then do that and you keep doing that forever and ever, doing it, right. So that can happen. If you program right, you can actually try to avoid deadlock by doing that, but both no one making forward progress, so that's called livelock.

So another thing that's called starvation. So the ordering is a very good example. So ordering says the higher order guy always gets the lock for the lower guy. So assume you have thousands of things and everybody's trying to do something. If you have an lower number, you probably never get to around to get picked up because always if higher order person has, that person will get the lock. So if they're ordering constraint in there, you can be in situation that some people always get and others never get -- there's no fairness in that, because when you some ordering constraints. So again, lack of fairness.

Of course, race conditions. So you didn't realize that the same object is accessed by multiple people without being in a particular section. That's the key -- I mean don't try to do fancy things by letting multiple people have access to same thing. This is not much of issue on distributed memory machines because there's only you access to your memory. But the problem there is if you keep values, you suddenly start giving it to everybody and say go play, assuming that only one person have access to it. So multiple people might be modifying it and then what are you going to do. So that issue is there. So when you are doing, using data, you got to be very careful who holds it and at what time.



So, concurrency and parallelism are important concepts in comparison beyond what we are doing in here. Concurrency can simplify programming beyond anything. It's very hard to understand and debug concurrent programs. That's the entire reason that we are still doing sequential programming and this is entire reason that multiple people are looking at it in a very -- people are scared because writing and getting concurrent program right is probably an order of magnitude harder than trying to get sequential programs right. This is issue.

Parallelism is critical for high performance. I mean, it was huge for supercomputers in national labs and now it's becoming everybody's issue because of multicore. Basically, you need to understand concurrent and concurrence issues, it's the basis of writing parallel programs. So, you will run into all these issues, deadlock, you can deadlock on limited access on Cell, you can deadlock on messages. So everybody is waiting for somebody else to send you a message and nobody's sending a message because that other guy will send you a message. You can [? easier ?] do that in a message in there, and a lot of times you can deadlock in that.

So this lecture we kind of did concurrent programming, how to write a concurrent program. We are going to switch gears and start going into parallelism next. But keep these issues in my mind when you are writing parallel programs. Have things like 617 we had very good discipline on testing and methodology of development. You probably won't have kind of discipline on how to do parallelism. So there are many ways -- next few lectures we'll cover many different ways of doing parallelism. Parallelism's a very powerful tool, but if you don't use it in a disciplined way, you will not be able to debug these [UNINTELLIGIBLE] I mean you run into bugs that are so subtle, so difficult it's very hard to find. You don't want in that situation. So having a good design, good disciplining programming will actually get you working correct program. Good. That's all I have for today. You can spend some time filling out this one. Just put your name down and you're done.