Today's topic is one of the most important concepts in this area, and it is called atomicity. And what we are going to do is spend time understanding what this is as a concept and then understanding how to achieve atomicity in systems. And recall that the main goal is to handle "failures", and that is what we talked about the last time. And we came up with a bunch of different ways of thinking about failures and how to cope with it. And one idea that we saw the last time was an idea involving replicating a component, let's say a disk or any component whose failure you wish to cope with and vote on the results. And so the idea is that if you are not exactly sure what the right answer should be-- If you are not sure whether any given component is working correctly or not, replicate that component and then give them all the same input, see what output appears and then vote on the results. And we did see that these things are pretty sophisticated, but the main problem with replicate plus vote is that often it is extremely expensive to build and very, very hard to get right. And, second, it often does not actually work. For example, if you just take a software program, a software module and you make 100 copies or 95 copies of that software module and give them all the same input and then vote on the output, if you have a bug in one of the modules and it is a bug that is actually replicated in all of the modules then all of the replicas are going to give you the same wrong answer. So the key assumption behind replicating and voting is that the replicas are independent of each other and have independent modes of failure. And that may not be true in all of your modules. And so the way we are going to deal with this problem, and even though it is possible to design software systems where the replicas are, in fact, independent of each other, it will turn out that it is quite expensive to do in many cases. So what we are going to do, to relax this assumption of having a system which handles failures by giving the same input to multiple outputs and then voting on it, we are going to relax that and instead look at a different concept called "recoverability". And the idea here is rather is rather than to try to replicate modules so that to the higher layers it looks as if the underlying module has never failed because you have replicated it, the idea here is to allow the underlying module to fail. But have it fail, typically in a fail fast manner so that you can detect the failure, and then arrange for that module to be restarted. And when it restarts the idea is to make it so that the module does something such that in the end the state of the system, after it does that thing, usually some kind of recovery procedure is that you can get back to using that module. So it is a little bit like rather than try to build, you know, the analogy might be something like this. You might imagine, let's say there is a little child who is learning to walk. One approach for nature to have adopted would have been to try to make it so the child never falls. And there is a lot of complexity associated with always keeping that child walking. Or, alternatively, you could have a story or a method by which every once in a while the child falls but then has a plan to get up from that fall and then restart. So that is the plan that we are going to adopt. And this notion here is called recoverability. And the general plan is going to be that if you have a module M1 which invokes another module M2 and M2 were to fail then the idea is that M2 fails and then it recovers and you restart the module. And you want to make sure that M2 is left in a situation, once it recovers, where there is no partial state. And I will define that more precisely as we go along today. But the main idea is going to be to insure that there is no vestige of previous computations that are in the middle of being run. So the state of the system, when it recovers, is at a well-understood point so that M1 can continue to use that. So there is no "partial" state where partial is in quotes here. And we will talk about what it means for something to be in a partial state. The idea is to prevent that from happening. So we are going to do this by starting with an example, and the same example that I mentioned the last time which was a transfer of money from one bank account to another. There is a "from" account, there is a "to" account and some dollar "amount". And you want to transfer money from "from" to "to" and it is whatever the "amount" is. And the problem here is, of course, that in the middle of transfer this procedure might fail, the system might crash and you might be left in a situation where a part of this transfer has already run. To take a specific example, here is an example of what the transfer procedure might look like. It takes a "from" and a "to" and an "amount". And the first thing it does is to read. Assume that all of this data is stored on disk. It reads from the "from" account and then it reduces, it debits the amount from the "account" and then writes back. And it does the same thing to the "to" account. So in the end, if this procedure completely ran, then "from" account would be reduced by "amount" and "to" account would be enhanced by "amount". Of course, the problem is you might have a failure anywhere in the middle. And, as a concrete example, if a crash were to happen after the first three lines shown above, if you owned this account you would not be very happy because you just lost some money from an account and nothing happened. No other account got money added to it, and this is the problem that we want to avoid. If you think about this for a moment, what you would like intuitively is that if a crash like this were to happen and the system were to recover and come back up, there are really only two states that the system should be in for the system to really be correct and to meet what your intuition might expect. Either this procedure must completely be finished, that is the state of the system must be the same as if this procedure completely ran and finished, or the state of the system must be such that the procedure never ran at all. It is not at all OK to let the state of the system be equal to whatever the state was, in this example, at the time the crash happened. What you want is a kind of all or nothing behavior. And, of course, if the crash happened as I have shown here, there is no way for you to have prevented those lines of code from being wrong. Those lines of code ran and then the crash happened. So what you really need is a way by which you can back out of these changes. What the system needs is a way by which when the system crashes and then recovers from the crash, during failure recovery the system has to have a way to back out of whatever changes it has made. In other words, what we want is a concept called recoverability. So a more precise definition of recoverability is shown on this slide, and let me just read it out. A composite sequence of steps, which we are also going to use the word "action" for, an action is recoverable if, from the point of view of the module that invokes this action, this sequence either always completes or aborts. That is if it fails and then backs out, aborts in a way such that it appears that the sequence had never started to begin with. And, in particular, what this means is that if a failure were to happen in the middle when the system recovers, it better have a plan of backing out the

changes. In other words, of aborting this action. The way you think about recoverability, the simple way to think about it is do it all or not at all. And our goal is to try to somehow come up with a way to achieve this goal. And before we get into a solution to this problem there are a few other concepts to discuss, and they will turn out to be very related to each other. And the second concept after recoverability that is very closely related to this idea has to do with concurrent actions. Imagine for a moment that you had the same transfer procedure as in this example but you had two transfers running at the same time and they happened to act on the same data items like that. Let's say that the first transfer moved from a savings account to a checking account, it moved $100. And the second one moved from savings to checking, it moved $200. And let's say at the beginning S was $1,000. And, of course as you recall from several lectures ago, when you have these interleave sequences, these two threads running the steps that these threads are made of might be interleave in arbitrary order if you don't have a plan to isolate them. And, in particular, you might have many results that show up. And one result that might show up is both of these transfers running concurrently read $1,000 from "from" account and then both of them debit by $100 and $200 respectively. So at the end of it you might be left with either $800 or $900 left in the account when the right answer is to have been left intuitively, if you ran both these transfers you would like to see $700 left in that account. So what you intuitively want here is if this is the first action, A1, and this is the second action, A2, what you would like to see is a sequence-- You don't actually care what the order is between these two transfers. I mean you are transferring money from one account to another and you are doing two of these. You do not actually care in this example, and it will turn out all the examples that we are going to be talking about with this notion that you are not really going to care what the order is. Either order is perfectly fine, but the order should be as if it is equivalent to either A1 before A2 or A2 before A1. And that is what we would like. And, of course, some naÄ«ve way to achieve this is to insure that exactly one action runs at a time, it finishes and then the second one runs, but that is kind of going to be no fun for us to do. It is the right simplest solution, but we are going to want to improve concurrency as we had wanted to several lectures ago. So we are going to come up with ways of getting higher performance than running one after the other. But the net effect is if you run it in some serial order, in some sequential order of the actions. That is the result of running concurrent action has to be the same as some serial ordering of the individual actions. And this idea of A1 before A2 or A2 before A1 has a name. It is called "isolation". And you should distinguish that in your mind clearly from recoverability. So a more precise definition of isolation is essentially what I said before. The composite sequence of steps is isolated if its effect from the point of view of its invoker is the same as if the action occurred either completely before or completely after every other isolated action. And the simple way to understand this is you either do it all before or do it all after. That is the net effect has to be the same as doing it all before or doing it all after. And it is different from recoverability which is really do it all or not at all. Now, when you have a system that satisfies both recoverability and isolations-- The way to understand this is both of these really, although they are talking about different concepts, this is saying all or nothing and this is saying all before or all after, both of these are getting at the same intuitive idea which is that somehow there is a sequence of steps, for example, in this transfer procedure there will be sequences of steps. And somehow you want to make it look as if, for each action, the sequence of steps is not visible to somebody invoking the action because you do not want the person invoking this action for recoverability. You do not want him to know that it is build out of a sequence of steps. And if a failure happens in the middle, you do not want the invoker of that action to see some partial state. Likewise, when you have concurrent actions running together, you do not want the different invokers of that action to somehow see this muddled result of the interleaving. You want them to only see the results of running these actions one after the other. What you really trying to achieve for both of these concepts, although they are distinct concepts, is to hide the fact that this action is a composite sequence of steps. You want to make it look as if it is quite [UNINTELLIGIBLE]. And this idea of wanting something to look [UNINTELLIGIBLE] is called "atomicity". And we are going to be basically hiding the fact that it is composite. So more precisely for this course, we are going to use the word "atomic" to mean recoverable and isolated. And I am going to say for this course because these terms have been used in various different ways for at least probably more than 30 years and I think it is about time we made these precise. In the literature, you will see the word atomic to often mean recoverable. And sometimes, and this is unfortunate, you will see the word consistent to mean isolated. And, in particular, you will run into this confusion when you read the paper for recitation on Thursday, the System R paper. The problem is those terms used historically have not been used in a very precise way so we will define it precisely. When we say something is atomic, in general we mean both recoverable and isolated. When we mean only one of them, we will say atomic with respect to recoverability or recoverable, atomic with respect to isolation or isolated. And, like I said, atomic means recoverable and isolated. The general plan is to hide the fact that an action is built out of composite sequence of steps. Now, to add to this confusion of terminology, there are actually two other terms or two other properties that you often want from actions in addition to recoverability and isolation. And these two other properties are provided by many database systems which are one of the most common users of these concepts. The most common system that provides atomicity, one example is a database system. Now, many, many systems provide atomicity. For example, every computer does it in its instruction set. You often want your instructions, from the point of view of the invoker of the instruction, to be atomic. So we are going to be designing techniques that, in general, operate across the whole range of systems. But database systems are of particular interest because they are very common and they exercise these concepts to a high degree. And two other concepts that many systems provide, the first one is "consistency". And it is unfortunate that the word consistency was previously used, to some extent, to mean isolated. So it is important not to get into that confusion. In some old papers when you see consistency, you should realize that what they really are talking about isolated, A1 before A2 or A2 before A1. But we will mean by consistency, and we will get into this next week, is that there is some invariant for the application that is often using atomicity that is maintained. For example, in a banking application, if you take the transfer examples, isolated means that you want the result to be as if the transfers ran in some serial order. Consistent means that there might be a high level notion that the

as if the transactions ran in some serial order. Consistent means that there might be a higher notion that the designer of this banking application might have wanted, such as a bank might have a rule that says that at the end of each day every checking account should have an amount that is at least 10% of the corresponding savings account. Now, during the middle of the day there might be individual actions that transiently violate that rule. But, at various points, the designer might wish to insure that a rule is the checking account must have at least a certain amount of money, some fraction of the savings account. Or in some payroll application for a company, they are modifying the payroll and giving raises to various people, but they might have a rule that says you could give whatever raise you want but every manager must make at least 5% more than all of his or her direct reports. You might have a rule like that. All of these are applications of an invariant that correspond to the consistency of the data that is being maintained in this example in a database. And you can use database systems to provide these consistency rules. But that is different from isolation. Isolation just says that there has to be some equivalent serial ordering in which things run. And the fourth property after recoverability, isolation and consistency is "durability". Durability basically says that the data should last for as long as-- It's an application-specific concept, but what it says is the data must last for as long as some pre-defined duration. For example, you might store data in a database. And, in many databases, you really want it to last "forever". But in reality it is very hard to make things last forever so you might define that the data in this database must last for three years, and you work hard to preserve that. Or you might have an application that as long as the thread is running you want the data to last, but after the thread is terminated you do not actually care about the data. And that is a different notion of durability. But both of these have talked about the lifetime with which you want to preserve data. Now, when you have a system that provides recoverability and isolation, that is atomicity, consistency and durability, then we are going to call that a transaction. A set of actions, each of which is recoverable, that are isolated from each other, that has a notion of consistency and can achieve it and where the data has durability, those actions are called transactions. And many database systems work hard to provide transactions, which means they provide all of these features. But it is certainly possible, and we will look at many examples where you can just design systems that have just recoverability and isolation. And we will not even worry about these other notions. That is what we will start with. We do not want to solve all of the problems at once. We will start with the easier set of problems and then build from there. Today, and on Wednesday, our plan is to come up with ways of achieving recoverability. So that is what we are going to start doing. The general approach for how we are going to achieve recoverability of modules is, and recall that the problem here is M2 fails and then M1 somehow discovers its failure and then when it restarts you do not want any partial state to be kept. The general plan is to design modules to be failed fast. You need a way to discover that things are not working, and that is the scope of the kinds of systems we are going to be dealing with. And then once the system's failure is detected and then you restart the system or it recovers, you run some kind of a repair procedure. This is in general you run some kind of repair procedure that allows that failed module to recover and then it restarts where restarts means it allows, M1 in this case, allows invokers to start running on that system, on that module. We are going to do this in three steps. The first thing we are going to do is to look at a very specific special case of this problem which is realize that all of these having to do with partial state occur because there is some state, once a module has crashed there is some state that it has remaining. So if it just recovered and started running again without doing something then that partial state is visible to the invoker of that module. Now, if the state were all a volatile state like in just RAM, for example, and a thread crashed, if it was in its virtual memory and the thread crashed and it recovered then you do not really have to worry about this because all of the state anywhere has gone away. Primarily, we were worried about state that lasts across failures. And an example of that is the state that is maintained on this, just as a concrete example. We are going to start first by obtaining a recoverable sector. Basically coming up with the scheme that allows us to do reads and writes of a single sector of a disk in a recoverable way. So we are going to define two procedures, a recoverable "put" that allows you to put stuff, write stuff onto a single sector of a disk and the recoverable "get" that allows you to read stuff of a single sector of a disk in a way that is recoverable. And the hard problem here is going to be that as the system is crashing, for a variety of reasons, bad data might get written to a sector. If you just took a regular sector of your disk, let's say that the operating system is trying to write something into a disk sector, somebody turns off the power and random stuff might get written out onto the disk. And so when the system comes back up, the reader of that sector might get some garbage value, a result of some partial write. So that is what we are going to try to avoid. So we will do that first. And that is for next time, to complete the recoverability story. We are going to use this solution as a building-block for a more general solution because it is not going to be enough for us to just be able to read and write single sectors in a recoverable way because how many applications use only one sector of a disk? What you would like to do is to make sure that you have a general solution that works across all of the data that is being written and read. We are going to use that to come up with two schemes. The first scheme uses an idea called a "version history". And a second scheme uses an idea called "logging" using logs. And both of these schemes will turn out to be very general and useful and work, but both of these schemes basically will use this technique as a bootstrapping technique. And so we need a solution here anyway because we are going to build on that to develop a more sophisticated solution for the general case. And so today we are going to start with a special case. A, because it is a building block, and, B, because it will turn out to show us a rule that we are going to religiously following in coming up with systematic solutions to work in a more general case when you have more than one sector being read. So let's write out the assumptions in the model here for this solution. The first assumption we are going to make, since we are dealing with recoverability and not with isolation. We are going to deal with isolation next week. The first assumption we will make is that there is no concurrency, and we will come up with different solutions for dealing with people concurrently trying to write the same sector. And this is an assumption we will revisit in a couple of weeks to show you how to actually achieve this goal. But we will assume that there are no hardware failures, no hardware errors. For example, the appendix to Chapter 8, which we have assigned for reading later on in the semester, actually shows two methods, "careful put" and "careful get" that actually deal with a variety of hardware problems. For example, every sector has a disk "checks-

careful get that actually deal with a variety of hardware problems. For example, every sector has a disk checks them" on it. If you wrote bad data and something happened in the middle of that write and then someone went back and read that sector, they would discover that it is bad because the checks-them would not match. Now, the appendix to this chapter, 9B, has a more careful description of how you deal with a variety of errors so that you can achieve this careful put and careful get of a disk sector. Assume for now that there are no hardware errors, there is no decay of data on the disk and so on. It will turn out the problem is still interesting, that it is not easy to achieve a recoverable put and get even though the hardware is fine. And that is because there are software errors. And, in particular, the model here is that you have some application and then you have the operating system. And the operating system has a buffer here of data that it is waiting to write onto disk. Then you have a disk and that is a disk sector. The problem might be that as a failure occurs there is something that happens, an error or something that gets triggered in the operating system so the buffer gets corrupted and then there is some bad data that gets written out onto the sector. That is the kind of problem that we want to protect against. The fact that your hardware is perfect does not actually solve this problem because this buffer itself has been corrupted or something happens during the process of writing this buffer to the sector so the data itself is bad, and that is what we want to protect against. We are going to build on something that I have already talked about. We are going to build on two procedures, careful put that puts to a sector, it puts some data, and the corresponding careful get which reads from a sector and returns the data that is on that sector. And the assumption is that careful put and get, once you give it some data there are no hardware failures for you to worry about anymore. The solution we are going to take to this problem is to realize that when a failure happens, for example, somebody turns off the power switch and this buffer gets corrupted, when the operating systems does a write to that sector, the sector might be left in a state that does not actually correspond to the data that was intended to put onto that sector. And so when the system recovers you are sort of stuck because this data in the sector contains some values in it that do not actually correspond to any actual intended put of the data, any intended write of the data. What this suggests is that a solution to this problem must involve a copy of some kind. You must make sure that if you have just one copy of the data and you write to it and something fails in the middle and you do not have a plan to back out to an earlier working version that was correct you are stuck. That suggests that we better have a solution that involves a copy of data. Later on we will see how to systematically [develop a rule?] based on this. The idea here is very simple. The way we are going to achieve a "recoverable get of a sector" is actually to build a single sector, a recoverable sector out of three sectors. The first sector here is going to have one copy of the data, the second sector is going to have another copy of the data and we are going to have a third sector which is going to act as a flag that allows us to choose one version or the other version. Let me call this D0, let me call this D1 and let me call this the "chooser". Assume that at some point in time D0 has proper data on it. The idea now is going to be that anybody reading it, the chooser is going to contain the value zero in it. Now, anybody reading is going to read from D0. Anybody writing in recoverable put is not allowed to write to D0 because that is what people are reading from. Instead, they will write to D1. When the chooser value is zero, they will start writing into D1. The plan is going to be that if that write succeeds properly then what we will do is go ahead and change the chooser from zero to a one, and then people will start reading from one. But if that write were to fail in the middle, if the power fails or something like that, D1 will be left in sort of a weird intermediate state. But that is OK because nobody is really going to be reading from D1. They are all going to be reading from D0 because the chooser has not yet been changed. The only other thing we have to worry about is now we are OK, as long as the failure happens, if the failure happens in the middle here somewhere where we are writing D1 we are OK because we have not touched the chooser. If the failure happens at the end of writing D1-- So we have written D1 and then we have not yet started writing the chooser and a failure happens here, we are still OK because everybody will be reading from zero. And that is not going to have garbage in it. It is not going to have the latest value in it. But that is OK. We never said that we should see the latest value for recoverability to hold. It is going to be OK for us to be reading from D0 and continue to read from D0. And really the correctness of this boils down to understanding what will happen when a failure happens during the middle of writing this sector. You are starting to write the chooser sector and the system fails. And we do not have to worry about that because now we have written D1 completely and a failure happened in the middle of that. To understand that, we will get back to understanding the correctness of it, but it helps to see what pseudo code looks like. So that is what put looks like. To do a put, you first read the chooser sector and then you put into the other place. This which here is the thing that tells you what the value of the chooser sector is. It tells you which of the two copies to write into. And then after you do the careful put, if which is zero, you put it to one, if which is one, you put it to zero. After that you twiddle a bit and then you do a put onto the chooser sector. The get is actually easier. You just look at what the value is of the chooser sector and then get it from the corresponding place. Now, there is a line here, the second line of this pseudo code which says status "not-OK". So status not-OK is the key thing. If status not-OK is what happens when a failure happens in the middle of writing the chooser sector. Let's say a failure happens on this pseudo code, I already explained why there is no problem if a failure happens until you get to the last line, until you get to the careful put of the chooser sector. Until that line is executed nobody sees the new data. Everybody doing a get is continuing to see the old data, not the new data that just got written with careful put. After this careful put executes and returns then everybody is going to see the new data because the chooser sector has been correctly changed. The only tricky part to worry about, we have reduced this problem of the slightly more general case of writing these sectors and switching between then to this specific problem of figuring out what happens if a failure occurs in the middle of the chooser sector's write. If a failure happens here, one of the common things that could happen is that this particular sector's checks-them does not match the data that is written here. So when you do a get of that sector here, in the first line up there, when you do a careful get of that, you will find that the checks-them does not match so it returns a status of not-OK. If the status is not OK, you will have to figure out which of the two copies to put. Now, the reason you can pick either and you can arbitrarily pick read the data from sector zero. But you could pick either of these. And the reason is it OK to pick either is you know for sure that the failure must have happened

other of these. And the reason is it OK to pick either is you know for sure that the failure must have happened here while writing this chooser sector. And because there are no concurrent threads going on, you are assured that there is no failure that happened here while writing D0, nor was there any failure that occurred here while writing D1 because the assumption we have made is that there is no concurrency. A system crashes and recovers and discovers that there is a failure, or the careful-get of the chooser sector did not quite work out, did not give you a status of OK, that it was not OK then you know the failure happened while writing here. And what that means is it is perfectly OK for you to read from either version. Both of those correspond to a write to that individual sector that did not fail in the middle. And it does not matter which of the two you pick. That is the reason why this approach basically works. And if you look at this solution, this copy idea is actually a pretty critical idea for all of our solutions to achieving recoverability. And it is going to lead to a rule that we are going to call the "Golden Rule of Recoverability". The rule says never modify the only copy. If you were asked to come up with a way to achieve something that is recoverable, one guideline, this is unfortunately not a sufficient condition. But a necessary condition is that if you have something, and you only have one copy of that which you end up writing, then chances are that if a failure happens in the middle of writing that one copy you cannot back out of it so your scheme would not work. So never modify the only copy of anything, that is the general rule. Now, there is another point to observe about this recoverable disk write. And that has to do with that careful put line. Write before that line, everybody else reading this recoverable sector using recoverable get sees the old version of data. Right after that line has finished, everybody reading it sees the new data. That line is an example of something that we will repeatedly visit and use called a "commit point". The successful completion of that line insures that everybody else following doing gets will see the data that was written by this put. And before that line is run, everybody else following will see the older version of the data. Now, if a failure occurs in the middle of that line then the answer depends on what the recovery procedure does. And one approach might be that the invoker of this module, the person who originally did the disk write-- If a failure happens in the middle of the write, one plan might be that the invoker of that disk write, upon recovery, tries the write again, tries the put again. And the way he tries the put is he first does a get and sees what answers return. If the answer is the new answer then he says OK everything is fine. If the answer is the old answer then he says I am going to retry the put. And this is an example of something we saw the last time which is "temporal redundancy". You can retry things. Not only can you replicate in space, but you can retry things in time which is the idea here for achieving fault-tolerance. An example of this idea called a commit point is that careful put line. And, in general, a commit point is a point in a recoverable action, in this case. And it will turn out to be an idea that is useful for isolated actions and for transactions more generally. But a commit point is a point where before the commit point other people do not see the results of your action. And after the commit point successfully finishes everybody sees the results of your action, and that is the definition of a commit point. Now we have to generalize this idea because what we have seen is a scheme. By the way, is this clear to everybody? Do you have any questions about recoverable put and get? What does that mean? No questions or not clear? All right. Good. Now we have to generalize this idea because the class of programs where you could just sort of read and write from one sector is quite limited. And so to generalize this idea of what we are going to do is to change the programming model for writing recoverable actions a little bit. Ideally, what you would like to be able to do, the model we are going to try to get at is to be able to take a procedure and begin recoverable action in front of that procedure, write code for that procedure and just say end recoverable action and sort of magically end up with a model where the set of steps in that action becomes recoverable. And it will turn out we have come very, very close to achieving this very general model by making some slight assumptions, or requiring the programmer to make some small assumptions in the way they write their programs. And this generalization to more general actions that are recoverable, generalizing from a single sector uses this idea of a commit point. The way this is going to work out is the programmer, for any recoverable action, he or she is going to end up writing this special function call called begin recoverable action and then writing the code for that recoverable action. And then at some point in the middle of this code calling a function called "commit". And the idea is that until this commit is called nobody else sees the results of this action. Which means that if a failure happened, upon crash recovery or once the system restarts, the result would be as if none of the steps of this action ever happened. So they are called commit. And then once commit finished then no matter what happens, a failure could happen and the system restarts, but once commit is called and it returns then you are guaranteed that all other actions see the state changes made by this action. So this is a special call. And then after commit they might have some other lines that they write and then they end the recoverable action. Now, in many, many cases, the very last thing that is done before the end recoverable action is the commit. But, in general, you might have other things here. And it will turn out that you cannot do arbitrary things here. For example, you cannot do disk writes that you want to make recoverable over here because the moment you do that, by definition, if a crash happens after a commit, we do not have a plan to back out of it. Because the semantics were that once a commit is done then no matter what happens the state of the system is as if all of the things in this action finished. The discipline is going to be, this thing is called the "pre-commit phase" and this thing here is called the "post-commit phase". And so the idea is that in the pre-commit phase you should always be prepared to back out. Because, by definition, if the failure occurs before commit is called the result is going to be as if nothing ever happened, which means that any change you make here you better religiously follow that never modify the only copy rule and be prepared to back out. In the post-commit phase, conversely, you don't have the option to back out so you better make sure that once you get here you just run to completion. If a failure occurs out here and you restart, you better make sure that you can run to completion. In fact, there are a few other restrictions out in the post-commit phase. Let me do this by an example. In the pre-commit phase, because you have to be prepared to back out, it often means in practice that you cannot be sending messages out onto the network. You can maintain your local state but you have a way to back out of that. But if you are sending messages out onto the network and you do not have a bigger story to deal with it-- We will talk later about nesting atomic actions within one another or nesting recoverable actions within one another in a few lectures from now. But, in the simple model, if you do anything that

recoverable actions within one another. In a few lectures from now. But, in the simple model, if you do anything that you cannot back out of such as sending a network packet then you are stuck. So all of that stuff like printing out checks or firing a bullet or things like that, that you cannot back out of, you better put out here. All the things that you can back out of go here. Likewise, nothing you can back out of can go here. Because, once you reach here and a failure happens, you have to continue to completion. What that means is in the first commit phase, really, you cannot do very many things. I mean you can do things that do not really have, for example, you can do things that are OK to keep doing. For example, you can do item put and operation so that if a failure happens here and you recover then you know that you are out at this point so you could keep retrying those actions over and over again until you insure that it completes. But those are the only rules. There is a pre-commit phase and a post-commit phase. There is a commit that is explicitly called. Now, in addition there is another call that a programmer can make or that the system can invoke automatically and that is called "abort". For example, when you are moving money from savings account to checking account in that transfer example, if you discover in the middle here that you do not have enough funds to cover that transfer, you could just decide to abort the recoverable action. And what that means is that abort automatically will insure that the state of the system is at the point right before the start of the recoverable action. Whatever changes were made in the middle until abort was called end up backing out. Now, abort might also be invoked by the system. In a database, there is somebody booking airline tickets, car reservations and all of that, and you discover in the middle that you are not actually able to find a hotel for the same dates. So you might just abort the whole process, control C the thread you are running, which means that all of the work that has been done has to be backed out. And so the system would normally implement that by aborting all of the changes that you have made so far. It will back out of your car reservation, back out of your airline reservation and so on. So abort is called in a few different contexts. Sometimes by the program itself, sometimes by the system to free up resources, sometimes by the user of your, say, transaction system. I am not going to get into how we implement recoverable action today, but this programming model is important to understand. I do want to mention one thing going back to this idea of isolation that we talked about. If you recall, isolation is this idea that you have two actions or multiple actions whose net effect is as if they ran in some sequential order, some serial order, A1 before A2 or A2 before A1 for all implantation of A1, A2, A3, etc. Now, this idea is actually very closely related but not the same as stuff we have seen before. Earlier in the semester we looked at ways in which you have multiple threads that need to be synchronized with each other. And we actually did look at isolation as a concept then but we specifically focused on things like sequence coordination where you want to have one thread run before the other thread or one thread run off of the other thread. For example, in a producer-consumer relationship. The point is that in one significant respect, achieving this idea of isolation for actions is harder than achieving sequence coordination. And the reason it is harder is that everybody who writes an isolated action, in general, does not know, any given isolated action does not know what other actions there are in the system. So you might have 25 different actions all of which are touching the same data, but no single action is aware of all of these other actions. That is very different from sequence coordination. In sequence coordination, there is one or two or a small number of threads that are actually aware of each other. And there is a single programmer that is actually designing these things to specifically interact with each other in some fashion, so this thread runs and then this other one runs after the data has been produced and so on. In that sense, this kind of isolation is harder to achieve because each individual action does not know which other action there are. But, yet, you want to achieve this sequential goal. Now, in one other respect, actually isolated actions are easier than sequence coordination. And the significant way in which they are easier is they are easier for programmers. Because we are not worried about coordinating different actions with each other, once you design a system that inside the system deals with ways of achieving isolation, the programmers do not have to think about locks and unlocks and acquiring and releasing locks or other ways in which they control access to variables that might be shared. What this means is that if we can design isolated actions right and we do not worry about any serial order, A1 can run before A2 or A2 before A1, then it makes life a lot easier for a programmer. And our goal is to come up with ways of achieving recoverability and isolation that require very little from a programmer that wants these properties. It is a little bit like pixy dust. You might write a general program and come in and just put a begin recoverable action, end recoverable action and make a few changes to your program. Or you might just say begin isolated action, end isolated action, and magically the system achieves isolation or recoverability for you. It can make life much easier for a programmer but it is a harder problem for us because no single action is aware of all of the other actions in the system. Next time we will see how to achieve recoverability and then isolation and transactions. Design Project 2 is out on the website now. And the main thing for you to make sure you do this week is get project partners and send a list of team members to your teaching assistant by Thursday's recitation. Thanks.