0:00:00 Quiz two is this week on Friday from, I hope that's correct, 0:00:03.795 whatever's on the webpage is correct, but I think it's 0:00:07.204 lecture 9 through recitation number 15. 0:00:09.649 So, in particular, the log structure file system 0:00:12.672 paper is not on the exam. Some of you may have been told 0:00:16.21 otherwise. What we're going to do today is 0:00:18.847 continue our discussion of atomicity. 0:00:22 0:00:27 Which, as you'll recall, is two properties that have to 0:00:31.429 hold in order for atomicity to hold. 0:00:34.3 And the first property that we talked about was recoverability. 0:00:39.386 And the other property, which is what we're going to 0:00:43.57 spend most of our time on today, is isolation. 0:00:48 0:01:06 So, before we get into isolation, I just want to wrap 0:01:09.714 up the discussion of recoverability because we left 0:01:13.285 it a little bit of three-quarters and didn't quite 0:01:16.785 finish it. So the story here was what we 0:01:19.571 did first was talked about how to achieve a recoverable sector 0:01:23.928 using two copies of the data and the chooser sector to choose 0:01:28.214 between them. We talked about one way of 0:01:32.158 achieving recoverability using version histories. 0:01:36.143 And we did complete that discussion. 0:01:39.049 And then we started talking about a more efficient way of 0:01:43.698 achieving recoverability in situations where you cared a lot 0:01:48.596 about performance using logging. 0:01:52 0:01:57 And the main in logging is this protocol for when you decide to 0:02:01.072 write the log called the write ahead logging (WAL) protocol. 0:02:04.948 And the idea is very simple. Always write to the log before 0:02:08.759 you update cell store. And that's the main discipline 0:02:12.175 that if you always follow then you'll get things right. 0:02:15.722 But one of the consequences of logging, you get two things from 0:02:19.795 having this log. The first is when an action 0:02:22.62 aborts. Independent of failure, 0:02:25.65 when you're running an atomic action and the program calls 0:02:29.739 abort or the system aborts that action, what the log allows you 0:02:34.186 to do is go back through the log, scan the log backwards, 0:02:38.204 look at all of the steps that that action took. 0:02:41.504 And that action didn't quite commit so you have to back those 0:02:45.808 changes out, and the log helps you unroll backward. 0:02:49.395 So primarily what you need with a logging scheme, 0:02:52.839 in or to do aborts, is the ability to undo. 0:02:57 Which means that what you need in the log, whenever you update 0:03:01.906 a cell store to something else, you need to keep track of the 0:03:06.732 old value. And this is what we called in 0:03:09.869 the log as an undo step. But now failures also happen in 0:03:14.293 addition to aborts, and we need a little bit more 0:03:18.154 mechanism in addition to just the ability to undo. 0:03:22.095 And to understand why, let's take a specific example 0:03:26.197 where you have an application and it is writing data to a 0:03:30.702 database that is on disk. And what we said was there was 0:03:35.941 a log and the log is stored on disk as well. 0:03:38.725 And let's assume that the log is actually stored on a 0:03:42.091 different disk. Now, the write ahead log 0:03:44.616 protocol basically says that whenever you update, 0:03:47.723 so you have an action which has things like reads and writes of 0:03:51.736 cell store. And whenever there's a write to 0:03:54.455 a cell store, the write ahead log protocol 0:03:57.109 says write to the log before you write to the database or to the 0:04:01.187 cell store. So two things can happen. 0:04:04.579 The first thing, the simplest model here is that 0:04:07.55 all writes are synchronous. What this means is that if you 0:04:11.152 see a write statement in an atomic action, 0:04:13.743 the first thing you have to do is to write to the log. 0:04:17.092 And then that returns. And then you write to the 0:04:20.063 database. And then you run the next 0:04:22.211 action, the next step of the action. 0:04:24.423 So clearly by the time in this action, if you ever get to the 0:04:28.215 commit point and you're about to exit your commit, 0:04:31.312 it means that all of the data has already been written to the 0:04:35.104 database. Because, by assumption, 0:04:38.132 we assume that all of the writes are synchronous and you 0:04:40.841 write to the database and only then do you go to the next step. 0:04:43.895 And, assuming there's only one thread of execution, 0:04:46.357 then in this simple model, you always write to the 0:04:48.77 database, you first write to the log and then you write to this 0:04:51.824 self-store in the database. But by construction, 0:04:54.139 when you get to the commit point, you know for sure that 0:04:56.847 all of the data has been written to the database. 0:05:00 So if a failure happens now and you didn't get to commit and the 0:05:04.655 system failed before the commit ran any time in the middle of 0:05:09.088 this action, the only thing you really need to do is to roll 0:05:13.448 back all of the changes made by actions that didn't get to 0:05:17.66 commit, which means that in this model that I've described so 0:05:22.093 far, the only thing you need to do is to undo actions that 0:05:26.305 respond to actions that didn't commit. 0:05:30 Any action that committed by construction must have written 0:05:33.124 all of its data and installed that data into cell store. 0:05:36.087 Because if an action gets to the statement to run commit and 0:05:39.265 then finishes commit, when it got to commit, 0:05:41.582 you know that because of all the writes being synchronous to 0:05:44.76 the cell store, you know that that write got 0:05:47.077 written. And, by the write protocol's 0:05:49.016 definition, you know the log got written before that. 0:05:51.818 So you don't actually have to do anything. 0:05:54.026 Even though the log contains the results of these actions 0:05:57.043 that committed, you don't have to do anything 0:05:59.414 for the committed actions. So, in fact, 0:06:02.733 this undo log is enough to handle the simple database as 0:06:06.096 well where all of the writes are being done synchronously in this 0:06:10.008 application. But there are a few things that 0:06:12.637 can happen. One of the reasons we threw out 0:06:15.205 version histories or discarded the idea to go to this logging 0:06:18.873 oriented model is for higher performance. 0:06:21.318 And one of the ways we got higher performance is we didn't 0:06:24.803 have to do these link list reversals in order to read and 0:06:28.227 write items. But it's also going to be 0:06:31.842 tempting, and you've seen this before, to not want to do 0:06:35.896 synchronous writes to cell store on a database in order to get 0:06:40.393 high performance. You might have asynchronous 0:06:43.636 writes happening to the database and you don't know when those 0:06:48.132 writes complete. And what could happen, 0:06:50.933 as a result of that, is you'd issue some writes to 0:06:54.545 the database and then you go ahead and you commit the action. 0:07:00 But what could happen is the data that was supposed to be 0:07:03.3 written to this database as store may not actually have 0:07:06.483 gotten written because there's some kind of a cache in memory 0:07:10.019

that actually returned to you from the write, 0:07:12.612 but the write actually never made it to the database. 0:07:15.677 So this system, for higher performance, 0:07:17.917 if you stick the cache in here, what could happen is that you 0:07:21.453 might reach the commit point and finish commit but not be 0:07:24.754 guaranteed that the cell store data actually got written to the 0:07:28.408 database. And if you fail now after 0:07:31.766 commit, you have to go through the log and redo the actions for 0:07:36.146 all of those actions that committed because some of those 0:07:40.103 actions may not have made it into the cell storage in the 0:07:44.059 database. So what this means is that in 0:07:46.744 general, when you put a cash here, any memory cash here or if 0:07:50.983 you have any situation where the writes are asynchronously being 0:07:55.434 done to cell store, you need both an undo log in 0:07:58.755 order to handle aborts and to handle the simple case and you 0:08:02.923 need the ability to redo the results of certain actions from 0:08:07.092 the log. And both of these are done, 0:08:10.762 so the system fails and then, when you recover, 0:08:13.765 before you allow other actions that might be ready to go to 0:08:17.55 take over, the system has to recover running the recovery 0:08:21.206 process. And the recovery happens from 0:08:23.621 the log. And the way that recovery works 0:08:26.167 is, and we went through this the last time, you scan the log 0:08:30.018 backwards from the most recent entry in the log. 0:08:34 0:08:40 And, while scanning the log backwards, you make two lists. 0:08:43.435 You make a list of winners and a list of losers. 0:08:46.267 And the winners are all of those actions that either 0:08:49.341 committed or aborted. And there's one important point 0:08:52.475 in the abort step, which is that when the system 0:08:55.308 calls abort or when the application calls abort and 0:08:58.321 abort returns, what the system does is goes 0:09:00.852 through the log and looks at all the changes made by that action 0:09:04.649 and undoes all of them. And then it writes the record, 0:09:08.628 called the abort record, onto the log. 0:09:10.636 So when you recovery and you see an abort record, 0:09:13.242 you already know that those changes made by an aborted 0:09:16.118 action have been undone. When you compose a list of 0:09:18.832 winners that are committed actions and aborted actions, 0:09:21.764 the only things you really need to redo are the steps 0:09:24.586 corresponding to the committed actions. 0:09:26.649 You don't have to undo the steps corresponding to the 0:09:29.471 aborted actions because they already got undone. 0:09:33 Because only after they got undone that the abort entry was 0:09:37.716 written to the log. In addition to these committed 0:09:41.701 and aborted actions that are winners, there are all other 0:09:46.256 actions that were pending or active at the time of the cache, 0:09:51.135 and those are losers. And so what you have to do to 0:09:55.201 the losers is to undo the actions done by losers. 0:10:00 So the actual recovery step runs after composing these 0:10:05.335 winners and losers and it corresponds to redoing the 0:10:10.469 committed winners and undoing the losers. 0:10:15 0:10:28 And independent of crashes, independent of failures, 0:10:31 when you just have actions that might abort, the only thing you 0:10:34.647 really need is undo. You don't need to redo 0:10:37.117 anything. If you don't ever have any 0:10:39.176 failures but just actions aborting, the only thing you 0:10:42.294 need to do with the log is to undo the results of uncommitted 0:10:45.823 actions before abort returns. Now, this procedure is OK. 0:10:49.058 Failures happen rarely and you're willing to take a 0:10:52 substantial amount of time to recover from a failure then the 0:10:55.529 log might be quite long. If a system fails once a week, 0:10:59.686 your log might be quite long. And if you're willing to take 0:11:02.947 the time to scan through the log entirely and build up these 0:11:06.265 winners and losers list then this is fine, 0:11:08.57 this approach works just fine. But people are often interested 0:11:12 in optimizing the time it takes to recover from a crash. 0:11:15.092 And a common optimization that's done is called a 0:11:17.791 checkpoint. And I believe you've seen this 0:11:20.096 in System R. And, if you haven't seen it, 0:11:22.345 you'll probably see it in tomorrow's recitation. 0:11:26 And the main idea in the checkpoint is it's an 0:11:28.44 optimization that allows this recovery process not to have to 0:11:31.694 scan the log all the way back to time zero. 0:11:33.972 That the system periodically, while it's normally operating, 0:11:37.172 takes this checkpoint, which is to write a special 0:11:39.83 record into the log. And that record basically says 0:11:42.542 at this point in time here are all of the actions that have 0:11:45.688 committed and whose results have already been installed into the 0:11:49.105 cell store. In other words, 0:11:50.515 they've committed and already been installed so when you 0:11:53.498 recover you don't have to go and scan the log all the way back. 0:11:58 You just have to go back enough so you find all of the actions 0:12:01.562 whose results haven't yet been installed completely which have 0:12:05.125 been committed. And the checkpoint also 0:12:07.345 contains a list of all the actions that are currently 0:12:10.382 active. So once you write a checkpoint 0:12:12.543 record to the log, during recovery you don't have 0:12:15.346 to go all the way back in time. There are a few other 0:12:18.384 optimizations that you can do with checkpoints that are not 0:12:21.771 that interesting to get into here, but the primary 0:12:24.633 optimization that's done to speed up the recovery process is 0:12:28.079 to use this checkpoint record. And most database systems, 0:12:32.535 the checkpoint record is pretty small. 0:12:34.72 It's not like you're check pointing the entire state of the 0:12:38.146 database. The main thing you're doing is 0:12:40.45 it's a pretty small amount of state that you're using to just 0:12:43.993 speed up recovery. So you shouldn't be thinking 0:12:46.71 that the checkpoint is something where you take the entire 0:12:50.077 database and copy it over. That's not what goes on. 0:12:53.03 It's a pretty lightweight, small amount of state rather 0:12:56.219 than the size of all of the data in the system. 0:13:00 So that's the story behind recoverability. 0:13:03.129 And we're actually going to come back to a piece of the 0:13:07.251 story after we talk about isolation now because it will 0:13:11.374 turn out that the mechanisms for isolation and the mechanisms for 0:13:16.259 recoverability using logs interact in certain ways, 0:13:20.076 so we have to come back to this either later today or on 0:13:24.274 Wednesday. So now we're going to start 0:13:27.099 talking about isolation. If you remember, 0:13:31.132 the idea behind isolation is when you have a set of actions 0:13:35.235 that run concurrently, what you would like is an 0:13:38.561 equivalent ordering of the steps of the actions running 0:13:42.382 concurrently such that the results are equivalent to some 0:13:46.344 serial ordering of the actions. The simple way of describing 0:13:50.518 isolation is do it all before or do it all after. 0:13:53.915 If you have actions, let's say T1, 0:13:56.25 T2 and so on running at the same time. 0:14:00 And these actions might operate on some data that is in common. 0:14:03.712 they might act on data that's not at all uncommon, 0:14:06.646 what you would like is an equivalent of the state of the 0:14:09.94 system after running these concurrent actions should be 0:14:13.173 equivalent to some serial ordering of the actions. 0:14:16.107 And it will turn out that what's tricky about isolation is 0:14:19.52 that if you want isolation and you don't care about performance 0:14:23.233 it's very, very easy to do. So it's very easy to get 0:14:26.287 isolation if you don't care about performance. 0:14:30 It's very easy to run fast if you don't care about 0:14:32.903 correctness. So, you know, 0:14:34.385 fast and correct is the hard problem. 0:14:36.518 Fast and not correct is trivial and correct and slow is also 0:14:40.014 trivial. I mean correct and slow is very 0:14:42.325 easy because you could take all of these concurrent actions and 0:14:46 just run them one after the other so you don't actually take 0:14:49.496 advantage of any potential concurrency that might be 0:14:52.518 possible. So suddenly slow and correct is 0:14:54.888 very easy to do. And we'll actually start out 0:14:57.496 with slow and correct and then optimize a simple scheme. 0:15:02 There has also been a huge amount of work that's been done 0:15:05.613 on fast and correct schemes. And, in the end, 0:15:08.402 they all boil down to this one basic idea that we'll talk about 0:15:12.333 toward the end of lecture today. So let's take an example. 0:15:15.946 Let's say you have two actions. I'm going to call them with Ts 0:15:19.813 because very often these are intended to be transactions 0:15:23.3 which are consistency and durability in addition to 0:15:26.47 isolation and recoverability. So I'm just going to use the 0:15:30.083 word T to represent an action. Let's say you have an action 0:15:34.826 that does read of some variable x and it does write of a 0:15:38.413 variable y and you have transaction T2 that does write x 0:15:42 and it does write y. So we're going to take a few 0:15:45.13 examples like this in order to understand what it means for 0:15:48.913 actions to run such that they're steps equivalent to some serial 0:15:53.021 order. So we're going to spend some 0:15:55.239 time really understanding that. And then, once we understand 0:15:59.086 what we want, it will turn out to be 0:16:01.369 relatively easy to come up with schemes to achieve it. 0:16:06 Let's say that what happens here is that the system gets 0:16:09.519 presented with these concurrent actions. 0:16:12.015 And let's assume each of these is an atomic step. 0:16:15.087 So there are four steps that can be interleaving in arbitrary 0:16:18.927 ways, in any number of ways. Let's say that what happens is 0:16:22.639 you run that first and then you run that second and then you run 0:16:26.672 this third and then you run this fourth. 0:16:30 So what you get is that I'm going to introduce a little bit 0:16:36.083 of a notation here. I'm going to write these steps 0:16:41.223 as r1(x), w2(x), w1(y) and w2(y). 0:16:44.58 What the r represents is a read, w represents a write, 0:16:50.139 what the subscripts represent is the identifier of the action 0:16:56.433 doing the read or write. And what's in parenthesis is 0:17:01.975 the variable that's being written. 0:17:04.304 So this says that action one does a read of x, 0:17:07.479 action two does a write of x, action one does a write of y 0:17:11.501 and action two does a write of y. 0:17:13.759 Now, if you look at these different actions, 0:17:16.793 there are a few steps that conflict with each other and a 0:17:20.744 few that don't. If you look at the read of x 0:17:23.778 and the write of y, they're independent of 0:17:26.671 everything else. If you just have read x here 0:17:30.624 and write y here, they don't conflict with each 0:17:33.291 other because those can happen in any order and the results are 0:17:36.887 exactly the same. Similarly, the write y and the 0:17:39.614 write x don't conflict with each other. 0:17:41.817 The only things that really conflict with each other are the 0:17:45.24 read x and the write x in this example because the results 0:17:48.546 depend on which goes first. Write y conflicts with this 0:17:51.678 write y, and those are basically the two things that conflict 0:17:55.157 with each other in this example. Generally, if you have two 0:17:59.89 actions conflict with one another, if they both contain a 0:18:03.963 read and a write of the same variable or a write and a write 0:18:08.254 of the same variable. So there are basically three 0:18:11.818 things that conflict. If you have some variable, 0:18:15.236 call it z, if you have a read of z and a write of z in one 0:18:19.381 action or the other, if you have write of z and read 0:18:23.09 of z or if you have write of z and write of z and those 0:18:27.018 conflict with one another. Now a read and a read don't 0:18:32.307 conflict. Because it doesn't matter what 0:18:35.641 order they run in, they are going to give you the 0:18:39.743 same answer. If you look at this ordering of 0:18:43.418 r1(x), w2(x), w1(y) and w2(y), 0:18:45.897 the things that conflict are these two and these two. 0:18:50.341 Now, if you look at the two things that conflict and you 0:18:55.042 draw arrows from which one happens before the other, 0:18:59.401 what you'll find is that for this x conflict one runs before 0:19:04.444 two and for this y conflict one runs before two. 0:19:10 So the arrows, if I were to draw them in time 0:19:13.025 order, would point this way. And this ordering is basically 0:19:17.014 the same as the same ordering you would get, 0:19:19.971 even though the steps run in different order, 0:19:22.997 it's exactly the same as if you ran T1 completely before T2. 0:19:27.054 Because running T1 completely before T2 says the ordering is 0:19:31.111 r1(x), w1(y), w2(x) and w2(y). 0:19:34 But the results are exactly the same, this different 0:19:37.013 interleaving, which does one, 0:19:38.668 two, three, four. So what this says is that this 0:19:41.445 trace that you get, we're going to use the word 0:19:44.163 trace for this, you present this concurrent 0:19:46.645 actions each of which has one or more steps and the system runs 0:19:50.309 them. And then the order in which the 0:19:52.436 individual steps run produces a trace. 0:19:54.622 And then the question is whether that trace is what is 0:19:57.754 called serializable. A trace is serializable if the 0:20:01.972 trace's results are identical to running the actions in some 0:20:05.852 serial order one after the other. 0:20:07.956 And so what we're going to be trying to do is, 0:20:10.916 what we're going to do today is to come up with schemes which 0:20:14.861 take these different steps corresponding to the action that 0:20:18.675 produces an order that turns out to be a serializable order. 0:20:22.555 And the challenge is to do it in a way that allows you to get 0:20:26.501 reasonable performance. To give you an example of a 0:20:32.716 nonserializable order, if we had the following trace, 0:20:39.874 r1(x) followed by w2(x) followed by w2(y) followed by 0:20:47.032 w1(y). What happens here is that these 0:20:52.125 two guys conflict so you've got an arrow from one to two going 0:21:00.522 this way. And, similarly, 0:21:03.933 these two guys conflict but the arrow in time goes from two to 0:21:07.728 one, which means that if you drew the arrow one to two this 0:21:11.337 way, you would have an arrow going the opposite direction. 0:21:14.884 So this doesn't correspond to any serial order because, 0:21:18.244 as far as this conflict is concerned, this trace says that 0:21:21.791 action one should run before action two. 0:21:24.217 And, as far as this conflict is concerned, it says that action 0:21:28.013 two should run before action one. 0:21:31 Which means you're in trouble because there is really no 0:21:34.196 serial ordering here. This trace does not correspond 0:21:37.16 to either T1 before T2 or T2 before T1 which means this 0:21:40.298 trace, if you have a scheme that runs your actions, 0:21:43.204 the steps of your action that produces the stress it means 0:21:46.517 that that scheme does not provide isolation. 0:21:49.016 Notice that we don't actually care whether T1 runs before T2 0:21:52.444 or T2 runs before T1. We're not worried about that. 0:21:55.35 We're just worried about producing some serial equivalent 0:21:58.605 order. 0:22:00 0:22:08 So that's the definition of the property that we want, 0:22:13.252 serializability. And what it says is a trace 0:22:17.513 whose conflict arrows, these are these conflict 0:22:22.072 arrows, are equivalent to some serial ordering of the steps of 0:22:28.117 the action. What we want is a trace 0:22:34.015 conflict that should be in the same order as some serial 0:22:43.619 schedule or some serial order of the actions. 0:22:52 0:22:58 So what we're going to do is in three parts. 0:23:00.617 The first part is we're going to look at one of these traces. 0:23:04.269 And given a trace, the first problem is to figure 0:23:07.191 out whether that trace corresponds to some serial 0:23:10.113 order. And then we're going to derive 0:23:12.304 a property that guarantees that if a sudden property holds we 0:23:15.956 would be assured that a trace is in serial order. 0:23:18.878 And then the second part is we are going to come up with 0:23:22.226 various schemes for achieving serializability. 0:23:26 And the third part is in order to prove that those schemes are 0:23:30.442 correct, what we are going to do is to prove that this property, 0:23:35.031 that all serial orderings should satisfy holes for the 0:23:38.891 protocol or for the algorithm that we design. 0:23:42.096 That is the plan for the rest of today. 0:23:44.864 This property for serializability is going to use 0:23:48.36 data structure or a construction called an action graph. 0:23:52.366 And it turns out what we are going to do, given one of these 0:23:56.663 traces, is produce a graph out of those traces called the 0:24:00.742 action graph. And then we are going to look 0:24:04.825 to see whether a certain property holds for that action 0:24:08.223 graph. Let me show you what this 0:24:10.174 action graph is by example. The graph itself consists of 0:24:13.636 nodes and edges And it's a directed graph. 0:24:15.65 And the nodes are not these r1s and w2s. 0:24:18.104 What the nodes are, are the actions themselves. 0:24:21 So, if you have four actions running, you have four nodes on 0:24:24.713 the graph. And then there are edges 0:24:26.853 between these nodes on the graph. 0:24:30 Let's do it by an example. Let's say you have first action 0:24:36.123 T1 which has read of x and write of y. 0:24:40.099 And just so we are sure that it is action one, 0:24:44.933 I'm going to draw one underneath as a subscript 0:24:49.876 because they're just reads. Action two has a write of x and 0:24:56.107 a write of y. Action three has a read of y 0:25:00.512 and a write of another variable z. 0:25:04.057 And action four has a read of x. 0:25:09 0:25:17 First of all, given these actions, 0:25:20.225 which of the actions conflict with each other? 0:25:24.624 Let's first write for T1. Does T2 conflict with T1? 0:25:30 Yes it does because the read x, I mean that's the same as that 0:25:34.048 example, so suddenly T2 conflicts with T1. 0:25:36.769 What about T3? Does T3 conflict with T1? 0:25:39.358 What that means is the interleaving of the individual 0:25:42.809 steps matter as far as the final answer is concerned. 0:25:46.261 Yes, it does because the write of y and the read of y conflict, 0:25:50.376 so you've got T1 and T3 that you have to worry about. 0:25:53.827 Does T1 conflict with T4? No it doesn't. 0:25:56.415 The read and the read do not conflict. 0:26:00 Does T2 conflict with T3? Yes, it does because it has got 0:26:05.106 the y. Does T2 conflict with T4? 0:26:07.933 It does. And does T3 conflict with T4? 0:26:11.308 It does not. There's nothing that's shared. 0:26:15.138 Out of the six possible, or whatever, 0:26:18.421 four, choose two possible conflicts, for conflicts you've 0:26:23.528 got four of them that you've got to worry about. 0:26:29 Now we're going to draw this graph that has T1, 0:26:32.197 T2, T3 and T4, and we're going to call this 0:26:35.116 the action graph. And what it's going to do is to 0:26:38.452 draw arrows between actions that conflict with one another. 0:26:42.484 But of course the answer, the arrows depend on the order 0:26:46.307 in which these individual steps get scheduled by the system 0:26:50.338 running these actions. We need an actual example for 0:26:53.883 that. Let's say that what happens is 0:26:56.316 you present these concurrent actions. 0:27:00 And what happens is this guy runs first and then two, 0:27:06.145 three, four, five, six and seven. 0:27:09.927 That's the order in which the system runs the individual steps 0:27:17.136 of this action. Now we're going to draw these 0:27:22.336 arrows between actions. If two actions share a 0:27:27.867 conflicting operation and in the first action the conflicting 0:27:32.962 operation occurs before the second action then you're going 0:27:37.886 to draw an arrow from the first action to the second action. 0:27:42.896 So more generally there's an arrow from PI to PJ. 0:27:46.971 If I and J have a conflicting operation that individual step 0:27:51.981 is run by the system for I first before J. 0:27:55.462 If you look here at T1 to T2 there's an arrow between T1 and 0:28:00.471 T2. Because it ran r1(x) before it 0:28:04.681 run w2(x). If you look at T1 and T3, 0:28:07.81 this is a little subtle because if ran read of x before it ran 0:28:13.262 r3(y), but that doesn't matter because read of x there and read 0:28:18.804 of y there don't conflict. But then it ran w1(y) after it 0:28:23.81 ran read 3y which means that the conflict is equivalent to 0:28:28.905 running that step of T3 before the step of T1. 0:28:34 So you actually have an arrow going back from T3 to T1. 0:28:38.618 Now what about T2 and T3? T2 and T3, the same story, 0:28:42.98 w2(x) and r3(y) don't conflict. But r3(y) before w2(y), 0:28:47.598 that's the conflict, so you have an arrow going this 0:28:51.96 way. So we've got three of them, 0:28:54.611 you need a fourth one, and that is between T2 and T4. 0:29:00 Between T2 and T4, w2(x) runs before r4(x) which 0:29:03.606 means you have an arrow going this way. 0:29:06.522 If you actually look at this picture, and we'll come up with 0:29:11.049 a method to systematically argue this point, but if you look at 0:29:15.806 that schedule, as shown here, 0:29:17.955 where the system runs the individual steps in that order, 0:29:22.252 this is actually equivalent to T3 running and then T1 running 0:29:26.856 and then T2 running and then T4 running. 0:29:31 The order of interleaving these different steps in the way shown 0:29:34.732 in that picture, one, two, three, 0:29:36.627 four, five, six, seven is actually equivalent to 0:29:39.412 the same result, it's the same result that you 0:29:42.078 get if you run T3 completely and then T1 and then T2 and then T4. 0:29:45.869 In fact, if you think about it, it's also equivalent to the 0:29:49.305 same ordering that you get if you run T3 and then you run T1 0:29:52.8 and then you run T4 and then you run T2. 0:29:56 That just says that for the exact same scheduling of 0:30:00.206 individual steps in the concurrent actions you might 0:30:04.412 find multiple equivalent serial orders that all give you the 0:30:09.278 same answer. Is that clear? 0:30:12 0:31:25 The arrow from two to four is correct. 0:31:27.932 Write effects runs before read effects. 0:31:30.943 So I think this is correct. Is there a problem? 0:31:34.588 All right. So, in this example, 0:31:36.966 it isn't multiple serial orders. 0:31:39.422 But in general it appears that there are multiple serial orders 0:31:44.335 possible. 0:31:46 0:31:53 What does this action graph got to do with anything? 0:31:57.434 It turns out, and we'll prove this, 0:32:00.391 that if the action graph-- 0:32:03 0:32:09 --for any ordering of steps within an action, 0:32:12.46 if the action graph does not have a cycle-- 0:32:16 0:32:21 --then the corresponding trace from which the action graph was 0:32:25.908 derived is serializable. 0:32:28 0:32:36 What that means is that what you have to do is, 0:32:39.118 given a certain ordering of steps, you construct this action 0:32:43.118 graph, you look to see if it has any cycles. 0:32:46.033 And, if it doesn't have any cycles, then you know that the 0:32:49.898 order is equivalent to some serial order of the steps of the 0:32:53.898 individual actions. And it actually turns out the 0:32:57.152 result is a bit more powerful. The converse is also true that 0:33:02.105 if you have a serializable trace then the corresponding action 0:33:06.385 graph has no cycles. Now, to turn out, 0:33:08.982 the more interesting result for us is going to be the following 0:33:13.333 direction where if the graph is acyclic then the trace is 0:33:17.263 serializable. Because what we're going to end 0:33:20.35 up doing is inventing one or two protocols for achieving 0:33:24.21 isolation, for achieving serializability. 0:33:28 And we're going to prove that those protocols are correct by 0:33:31.433 proving that the corresponding action graph, 0:33:33.935 all of the possible action graphs produced by those 0:33:36.844 protocols all have no cycles. So this direction is the 0:33:39.928 direction that's actually more important for us, 0:33:42.663 but the opposite is also true and not that hard to prove. 0:33:45.922 So what is the intuition behind why, if you have an action graph 0:33:49.588 that's serializable, sorry, that doesn't have cycles 0:33:52.556 the trace is serializable? Well, notice one thing about 0:33:57.102 this which is to draw a little bit of intuition. 0:34:00.632 Suppose, in fact, what happened here was we 0:34:03.786 didn't execute the actions, the steps in that order, 0:34:07.616 but what we did was to run this at step five and that at step 0:34:12.122 six. What would happen with the 0:34:14.375 resulting action graph is that you would actually have an arch 0:34:18.956 from T2 to T1 going the other way as well. 0:34:22.035 Because what this says is between T1 and T2 there is one 0:34:26.166 conflicting operation that goes this way where action one runs 0:34:30.747 before two. And these two guys conflict 0:34:34.977 where the step in two runs before the step in one and those 0:34:39.224 two steps conflict with one another. 0:34:41.787 And this is actually the cycle here that causes the whole 0:34:45.887 scheme to be not serializable anymore. 0:34:48.597 That's a little bit of intuition as to why this acyclic 0:34:52.551 property is important. But to really prove this notice 0:34:56.432 that if you have a directed acyclic graph you could do 0:35:00.313 something called a topological sort on the graph. 0:35:05 How many people know what a topological sort is? 0:35:08 OK. For those who don't, 0:35:09.468 the idea is very simple. In any directed acyclic graph 0:35:12.851 there is going to be at least one node that has no arrows 0:35:16.425 coming into it. All of the arrows going out are 0:35:19.361 only going out of the node. And you can actually prove that 0:35:23.063 by arguing the contradiction. If it turns out that every node 0:35:26.893 has an arch coming in and an arch going out then by 0:35:30.085 traversing that chain of pointers you'll end up with a 0:35:33.468 cycle. So it's pretty easy to see that 0:35:36.96 in any directed acyclic graph you're going to have some node 0:35:40.69 that has no arrows coming into it and only arrows going out of 0:35:44.548 it. So find that action, 0:35:46.002 in this picture it is T3, and take that action and put it 0:35:49.543 in first. That's the first action that 0:35:51.882 you run. Because it has no arrows coming 0:35:54.348 into it and only arrows going out, what that means is that no 0:35:58.142 other action in a serial order runs before it. 0:36:02 Or at least there's no reason for any action to run before it 0:36:05.514 because there are no arrows from any other action coming into 0:36:09.029 this action. So you put that in first. 0:36:11.196 Now, remove that node from the entire graph. 0:36:13.715 The resulting graph is acyclic, right? 0:36:15.882 You cannot manufacture a cycle by removing a node so you 0:36:19.104 recursively apply the same idea, find some other node which has 0:36:22.736 no other things coming into it and only things going out of it. 0:36:26.368 And if there are ties just pick one at random. 0:36:30 And, therefore, construct an order. 0:36:32.506 And all such orders that you construct are topological sort 0:36:36.781 orders. This topological sort order, 0:36:39.361 by construction, is a serial order of the 0:36:42.309 actions. And this topological sort, 0:36:44.815 if you now draw the arrows in the topological sort, 0:36:48.501 they're going to be the same arrows as in the original 0:36:52.407 directed graph, it's exactly the same graph, 0:36:55.577 and now you have an equivalent serial order. 0:37:00 That's the reason why the cyclic property is actually 0:37:04.543 important as far as serializability is concerned. 0:37:09 0:37:18 Now we can look at schemes that actually guaranty 0:37:20.914 serializability. And the schemes we're going to 0:37:23.708 discuss all are in this system where you have cell storage and 0:37:27.412 where you have logs for recovery. 0:37:29.356 The logs are not going to matter, for the most part. 0:37:33 You can also do isolation in version histories, 0:37:36.128 and one of the sections of the notes deals with that at length. 0:37:40.345 I personally think it's not that important, 0:37:43.201 but that doesn't mean it's not on the quiz. 0:37:46.057 I think you get a little bit more intuition reading that in 0:37:50.002 addition to this discussion. And a bulk of this discussion 0:37:53.879 is actually not in the notes. It's just another way of 0:37:57.483 looking at the problem. The mechanism we're going to 0:38:01.94 build on is, in fact, described in the notes as well. 0:38:05.544 It's a mechanism you've seen before, and it is called locks. 0:38:09.633 As you recall, if you have variable x or any 0:38:12.613 chunk of data you can protect it using two calls, 0:38:15.94 acquire and release. So you could do things like 0:38:19.198 acquire lock of x and release lock of x. 0:38:21.9 And the lock protocol is that only one person can acquire a 0:38:25.92 lock at a time. And all of the people wanting 0:38:29.776 all other actions wishing to acquire the same lock will wait 0:38:33.39 until the lock is released and then they fight to acquire it. 0:38:37.066 And ultimately at the lowest level the lock has to be 0:38:40.252 implemented with some low level atomic instruction. 0:38:43.315 For example, something like a test and set 0:38:45.827 lock instruction. It's the same story as before. 0:38:48.706 Now, there are a few things to worry about with locks. 0:38:51.953 The first one is the granularity of the lock. 0:38:54.648 You could be very, very conservative and decide 0:38:57.466 that the granularity of the lock is your entire system. 0:39:02 So all of the data on the system gets protected with one 0:39:05.473 lock, which means that you're running this very slow scheme 0:39:09.136 because what you're guaranteeing is, in fact, isolation but 0:39:12.8 you're running essentially one action at a time and you have no 0:39:16.715 concurrency at all. The other extreme, 0:39:19.052 you could be very, very aggressive and every 0:39:21.768 individual cell stored item is protected with the lock. 0:39:26 And now you're trying to max out the degree of concurrency, 0:39:31.165 which means that although things could be fast you have to 0:39:36.241 be a lot more careful if you want things to be correct. 0:39:41.05 And correct here means that you have an order that is equivalent 0:39:46.661 to some serial order. Why does the locking protocol 0:39:51.114 actually matter? Well, let's go back to these 0:39:55.033 two action examples of r1(x) w2(x) and w1(y) and w2(y). 0:40:01 And let's just throw in an acquire lock of x here and an 0:40:08.137 acquire lock of y here and a release lock of x here and in 0:40:15.534 between here you have the read and here you do the release of 0:40:23.32 the lock of y and similarly you do an acquire lx here and you do 0:40:31.496 an acquire ly here. And then you release the locks 0:40:37.211 at the end here. So acquire lock x, 0:40:39.717 read x, release lock x, acquire lock y, 0:40:42.518 write y, release lock y, and acquire right, 0:40:45.614 acquire right, release, release. 0:40:47.899 And, on the face of it, this kind of thing is actually 0:40:51.805 reasonable for the old style synchronization, 0:40:55.049 some of the old style synchronization things that we 0:40:58.808 wanted because we didn't actual care about atomicity of full 0:41:03.157 actions. If you look at what happens 0:41:07.634 with this kind of locking, as shown here, 0:41:11.536 you might be a little bit in trouble because it could be that 0:41:17.39 these three steps happen first and then this whole set of steps 0:41:23.439 up to here happen second, actually, up to the end. 0:41:29 And then this chunk happens third. 0:41:31.368 Now you're in trouble because read x happens before write x 0:41:35.53 and the write y happens before this write y. 0:41:38.615 And now you have not achieved isolation because, 0:41:41.988 if you draw the conflict graph, you'll get an arrow from T1 to 0:41:46.365 T2 and an arrow coming back from T2 to T1 and you have a cycle, 0:41:50.814 which means that just throwing in the right acquires and 0:41:54.761 releases isn't going to be sufficient to achieve isolation. 0:42:00 So we're going to need a much better set of skills, 0:42:03.896 a better scheme than just this throw in the right acquires and 0:42:08.649 releases. The first scheme we're going to 0:42:11.766 see is a simple scheme. It's called simple locking. 0:42:15.662 The idea in simple locking is that every action knows 0:42:19.714 beforehand all of the data items that it wants to read or write, 0:42:24.623 and it acquires all of the locks before it does anything. 0:42:30 Before doing any reads or writes it acquires all of the 0:42:33.74 locks. The idea would be here you 0:42:35.956 would acquire the lock of x and acquire the lock of y, 0:42:39.627 and then you run the steps. Similarly, the other action 0:42:43.367 does the same thing. And by construction, 0:42:46.138 if you have actions that conflict with one another and 0:42:49.809 one of the actions reaches the point where all of the locks it 0:42:54.034 needs have been acquired then by construction no other 0:42:57.705 conflicting action could be in the same state. 0:43:02 Because if you have another action that conflicts then it 0:43:05.181 means that there's at least one data item common to them which 0:43:08.647 means that only one of them could have reached this point 0:43:11.829 because they're both trying to acquire. 0:43:13.988 This protocol where every action acquires all of the locks 0:43:17.227 before running any step of the action will guaranty isolation. 0:43:20.693 And the isolation order, the equivalent serial order 0:43:23.59 that it guarantees is the same as the order in which the 0:43:26.715 different actions reach this point where they have acquired 0:43:30.011 all of the locks. And that point is also called 0:43:33.646 the lock point. The lock point of an action is 0:43:36.2 defined as the point where all of the locks that it needs or it 0:43:39.719 wants to acquire have been acquired. 0:43:41.706 And, because in this discipline where no action does any 0:43:44.828 operation until all of the locks it needs have been acquired, 0:43:48.233 you're guaranteed that this serial order of every action 0:43:51.355 following this protocol, independent of knowledge of any 0:43:54.477 other actions, everybody just blindly follows 0:43:56.974 this protocol, the serial order is the same as 0:43:59.529 if the actions had run in the order of acquiring the lock 0:44:02.707 points, whatever that order might be. 0:44:06 This is called simple locking, and it does provide isolation. 0:44:11.165 But the problem with simple locking where you acquire all of 0:44:16.245 the locks before running anything is two-fold. 0:44:20.119 The first problem is that this dictates that all of the actions 0:44:25.456 know the different items that they want to read or write 0:44:30.192 beforehand. Which means that if you're deep 0:44:33.595 inside some action and there are all sorts of conditional 0:44:36.477 statements you kind of have to know beforehand all of the data 0:44:39.617 items you might be reading and writing. 0:44:41.573 And that could be pretty tricky. 0:44:43.169 In practice, if you want to adopt simple 0:44:45.176 locking, you might have to be very conservative and sort of 0:44:48.161 try to lock everything or lock a large amount of data which 0:44:51.147 reduces performance. Second, it actually doesn't 0:44:53.566 give you enough opportunity to get high performance, 0:44:56.191 even when you do know all of the data items beforehand. 0:45:00 For example, one thing you could do is 0:45:02.926 acquire the lock of x and then read x. 0:45:05.853 And while you're off trying to read x then you might acquire a 0:45:10.677 lock of y and read y. And so, depending on how you 0:45:14.553 have structured your system scheme where you do some work, 0:45:19.062 maybe some computation as well in between the lock acquisition 0:45:23.887 steps might give you higher performance. 0:45:28 So the question is can we be a little bit more aggressive than 0:45:31.828 the simple locking scheme in order to get isolation as well 0:45:35.468 as a little bit higher performance? 0:45:37.602 And the answer is that there is such a scheme and it's called 0:45:41.368 two-phase locking. 0:45:43 0:45:50 And although a lot of work has happened for maybe a couple of 0:45:53.284 decades on very high performance locking schemes, 0:45:55.912 it turns out in the end they all, or at least for a large 0:45:58.978 class of schemes that use this kind of locking they all boil 0:46:02.208 down to some variable of two-phase locking. 0:46:05 And the idea is very simple. The two-phase locking idea says 0:46:11.413 there should be no release before all the acquires are 0:46:17.173 done. So do not release any lock 0:46:20.543 until all of the locks that you need have been applied.

0:46:26.413 That's what happens here. This schemes violates two-phase 0:46:32.104 locking because you actually release lock x before you 0:46:35.701 acquire lock y, and that violated two-phase 0:46:38.552 locking. And this idea that you don't 0:46:40.995 release before all the acquires, there is a little bit of a 0:46:44.932 subtlety that happens because you want recoverability which is 0:46:49.072 that the action could at any stage abort. 0:46:51.787 And, in order to abort an action, you need to go back and 0:46:55.588 undo that variable, the value to a previous value. 0:47:00 Which means that in order to abort you need to hold onto the 0:47:03.85 lock. You better make sure that an 0:47:06.004 action in order to abort has the locks for all of the data items 0:47:10.115 whose values it wishes to change to the original value. 0:47:13.639 Which means that in practice, at least for all the items that 0:47:17.555 you're writing, the locks that you hold should 0:47:20.492 not be released until the commit point, until the place where the 0:47:24.669 action calls commit. So no release before all 0:47:27.541 acquires is basically equivalent to-- 0:47:31 There should be no acquire statement after any release 0:47:34.856 statement. The moment you see a release 0:47:37.622 statement and then an acquire after that of anything, 0:47:41.406 then you know that this violates two-phase locking. 0:47:45.044 It turns out that two-phase locking is correct that it 0:47:48.901 provides isolation. And to see why let's look at a 0:47:52.467 picture. We're going to go back to this 0:47:55.232 action graph idea and look at a picture of what happens with 0:47:59.525 two-phase locking. What we're going to prove is 0:48:03.798 that if you use two-phase locking and you construct an 0:48:07.203 action graph of what you get from running a sequence of steps 0:48:11.057 that action graph has no cycles. And we know that if you have no 0:48:15.104 cycles in the action graph you're guaranteed that it is 0:48:18.573 equivalent to some serial order. We will argue this by 0:48:21.978 contradiction. Let's say you have T1 and T2 0:48:24.676 all the way through some action Tk and you have a cycle going 0:48:28.531 back from Tk to T1 in the action graph. 0:48:32 Now, if there is an arrow from T1 to T2, it means that there is 0:48:38.287 some data item x1 in common between T1 and T2. 0:48:42.85 And you know that T2 ran after T1 which means that in T1 there 0:48:49.036 was a release done of l1 after which in the system there was an 0:48:55.323 acquire done of l1. Likewise, between T2 and T3, 0:49:00.09 there is some data item x2 such that a release was done of l2 by 0:49:06.478 T2. And after that an acquire was 0:49:11.011 done of l2 by T3. And the release has to have 0:49:15.58 been done after the acquire of l1 because we're following 0:49:21.396 two-phase locking. If you continue that down up to 0:49:26.484 here out for Tk, Tk did an acquire somewhere 0:49:30.949 later in time of lk minus one. And then it did a release of 0:49:37.723 some data item, a lock of lk where lk is 0:49:41.517 actually some data item that is shared between Tk and T1, 0:49:46.964 so there is actually some data item xk whose lock is lk. 0:49:52.315 And you know that Tk did a release of lk before T1 did an 0:49:57.762 acquire of lk. But the T1's acquire of lk must 0:50:01.667 have happened after the release for lk so it must have happened 0:50:05.001 at some point here. T1 must have done an acquire of 0:50:07.69 lk at some point at the bottom here. 0:50:09.573 Just going out in time, by two-phase locking you get 0:50:12.315 release of l1, then the other guy doesn't 0:50:14.467 acquire of l1, release of l2, 0:50:15.973 acquire of l2 all the way out, then release of lk, 0:50:18.608 and then after that in time an acquire of lk. 0:50:20.974 So that must have happened later on in time, 0:50:23.287 but now this picture here violates two-phase locking 0:50:26.03 because T1, for this cycle to hold, has to have done an 0:50:28.934 acquire of lk after release of l1. 0:50:32 But that violates two-phase locking because you're not 0:50:34.852 allowed to acquire anything after you've released something. 0:50:38.028 So two-phase locking, therefore, cannot have a cycle 0:50:40.773 in the action graph. And, from the previous story, 0:50:43.411 it means that it's equivalent to some serial order that 0:50:46.317 corresponds to the topological sort of the directed acyclic 0:50:49.439 graph. I'm going to stop here. 0:50:51 We will continue with this stuff and then talk about other 0:50:54.068 aspects of transactions next time. 0:50:55.844 And if there are any questions either send me an email or ask 0:50:59.074 me the next time.