Practice Exercises on Transactions

Solutions

Serializable, Conflict Serializable, Recoverable, and Cascadeless Schedules

Recall that:

- A schedule is *serializable* if its effects are equivalent to the effects of some serial schedule.
- A schedule is *conflict serializable* if it can be transformed into a serial schedule by swapping pairs of non-conflicting actions. Two action *conflict* if they involve the same data item and at least one of them is a write.
- A schedule is *recoverable* if every transaction commits only after all transactions whose changes they read have committed.
- A schedule is *cascadeless* when it avoids *dirty reads*: reads of data that have been modified by uncommitted transactions.

In the exercises below, determine whether or not the provided schedule has each of the above properties.

1. **Schedule A:**

T1	**T2**
read(X)	
	read(X)
write(Y)	
	write(Y)
commit	
	commit

   - Is this schedule serializable?
     Yes, because its effects are equivalent to those of the serial schedule T1;T2 -- in both schedules, T1 and T2 read the initial value of X, neither transaction reads a value written by the other, and T2 performs the final write of Y.

   - ...conflict serializable?
     Yes, it's equivalent to the serial schedule in which T1 performs all of its operations and then T2 performs all of its operations. To see this, we note that T2's read of X does not conflict with T1's write of Y, because the two operations involve different data items. So, these two operations can be swapped to give us the serial schedule T1;T2 (note that the commits do not matter when it comes to determining conflict serializability).

- ...recoverable?
  Yes, because there are no dirty reads - i.e., there are no situations in which one transaction reads a value that has been written by another transaction that has yet to commit. Therefore, there's no need to worry about the order in which the transactions commit.

- ...cascadeless?
  Yes, because there are no dirty reads.

2. **Schedule B:**

T1	**T2**
read(X)	
	write(X)
read(X)	
write(Y)	
commit	
	commit

- Is this schedule serializable?
  No. This schedule is not equivalent to either of the two possible serial schedules T1;T2 or T2;T1. It's not equivalent to T1;T2 because in our schedule T1's second read of X reads the value written by T2, but in the serial schedule T1;T2 it would read the initial value of X. It's not equivalent to T2;T1 because in our schedule T1's first read of X reads the initial value of X, but in the schedule T2;T1 it would read the value written by T2.

- ...conflict serializable?
  No. We can't use swaps of consecutive non-conflicting operations to transform this schedule into a serial schedule. There are two different pairs of operations that conflict and so cannot be swapped: (1) T1:read(X) (the first time)/T2:write(X), and (2) T2:write(X)/T1:read(X) (the second time). Since we cannot move T2's write(X) either before or after all of T1's operations, there's no way to produce a conflict-equivalent serial schedule.

- ...recoverable?
  No, because T1 performs a dirty read (reading a value of X written by T2 before T2 commits) **and** T1 commits before T2 does. If the system were to crash after T1 committed but before T2 committed - or if T2 were to abort after T1 committed - the system could be put into an inconsistent state, because the rollback of T2 should undo T2's write of X, and yet T1 could have performed an action based on that write.

- ...cascadeless?
  No, because T1 performs a dirty read.

3. **Schedule C:**

T1	**T2**
read(X)	
	write(X)
read(X)	
write(Y)	
	commit
commit	

   - Is this schedule serializable?
     No. Again, apart from the order of the commits this is the same schedule as schedule B, so it's still not equivalent to either of the two possible serial schedules.

   - ...conflict serializable?
     No. Aside from the ordering of the commits, this is the same schedule as schedule B, so we still can't perform swaps of conflicting actions to get a serial schedule.

   - ...recoverable?
     Yes. Although T1 still performs a dirty read in this schedule, it now commits **after** T2, which satisfies the requirement for a recoverable schedule. If T2's write is undone by a crash or abort, T1 can also be rolled back, since it won't have committed yet. While such a cascading rollback is undesirable, it prevents us from a possible inconsistent state.

   - ...cascadeless?
     No, because T1 still performs a dirty read.

4. **Schedule D:**

T1	**T2**	**T3**
	read(A)	
read(A)		
	read(B)	
write(A)		
		read(B)
	write(A)	
		write(A)
commit		
	commit	

		commit

- Is this schedule serializable?
  No. In our schedule, T2's r(A) reads the initial value of A, but in any serial schedule in which T1 comes before T2, it would read the value of A written by T1, so it can't be equivalent to any serial schedule in which T1 comes before T2. Similarly, in our schedule, T1's r(A) reads the initial value of A, but in any serial schedule in which T2 comes before T1, it would read the value of A written by T2, so it can't be equivalent to any serial schedule in which T2 comes before T1. Since in every possible serial schedule T1 must either come before or after T2, this schedule is not view equivalent to any possible serial schedule.

- ...conflict serializable?
  No. We can see this by noting that we can't move all of T1's operations *after* all of T2's operations, since we can't swap T1's w(A) with T2's w(A) because they conflict, and we *also* can't move all of T1's operations *before* all of T2's operations, since T1's w(A) conflicts with T2's r(A). This means that T1 can neither come before nor after T2, which means a conflict-equivalent serial schedule is impossible here.

- ...recoverable?
  Yes, because there are no dirty reads - i.e., there are no situations in which one transaction reads a value that has been written by another transaction that has yet to commit. Therefore, there's no need to worry about the order in which the transactions commit.

- ...cascadeless?
  Yes, because there are no dirty reads.

5. **Schedule E:**

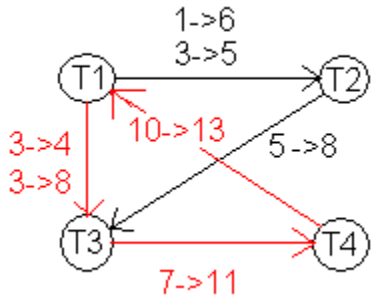
T1	**T2**	**T3**	**T4**
read(X) (1)			
	read(X) (2)		
write(Y) (3)			
		read(Y) (4)	
	read(Y) (5)		
	write(X) (6)		
		read(W) (7)	
		write(Y) (8)	
			read(W) (9)
			read(Z) (10)

			write(W) (11)
read(Z) (12)			
write(Z) (13)			

- Draw a precedence graph to determine if this schedule is conflict serializable.
  I've numbered the operations above to make the graph easier to draw. We add an edge from transaction A to transaction B in the graph if transaction A would have to come *before* transaction B in a serial schedule because there is an operation in transaction A that comes before a conflicting operation in transaction B. For example: transaction A: W(X) transaction B: R(X) -- there's no way for us to swap these operations so that transaction B comes first.

  If we end up with a cycle in the graph, it means that we have an impossible situation (eg., transaction A comes before transaction B, and transaction B comes before transaction A) and the schedule is not conflict serializable.

  Here is the precedence graph:



  As you can see, there are cycles in the graph: T1->T3->T4->T1 and T1->T2->T3->T4->T1. This means that the schedule is not conflict serializable.