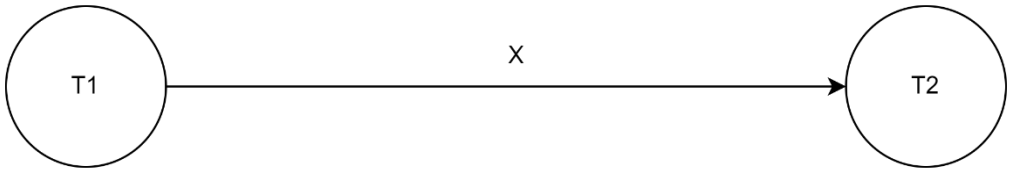
Homework-10

By: Sai Rashwant Venkataraman Sundaram

1. Schedule 1:

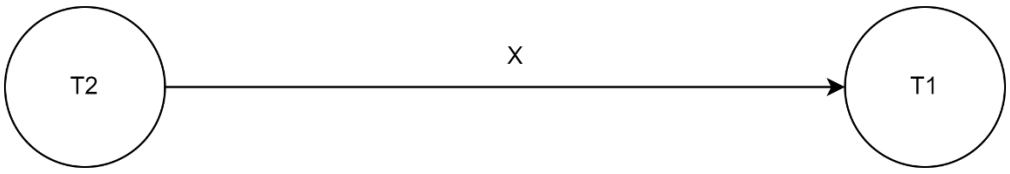
	T1	T2
t1	Start	
t2	Read(x)	
t3	x=x-n	
t4	Write(x)	
t5	Commit	
t6		Start
t7		Read(x)
t8		x=x+m
t9		write(x)
t10		commit



Its conflict serializable as there is no cycle present.

Schedule 2:

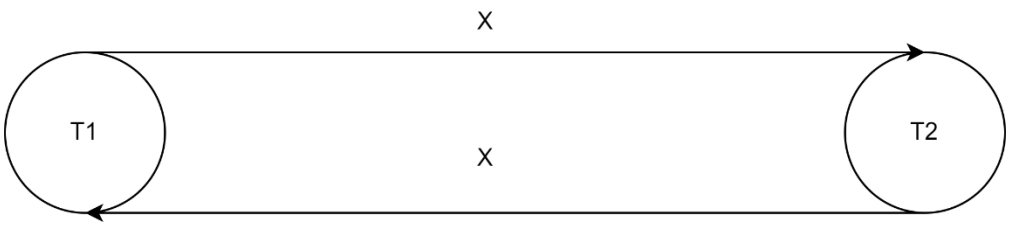
	T1	T2
t1		Start
t2		Read(x)
t3		x=x+m
t4		write(x)
t5		commit
t6	Start	
t7	Read(x)	
t8	x=x-n	
t9	Write(x)	
t10	Commit	



It is conflict serializable.

Schedule 3:

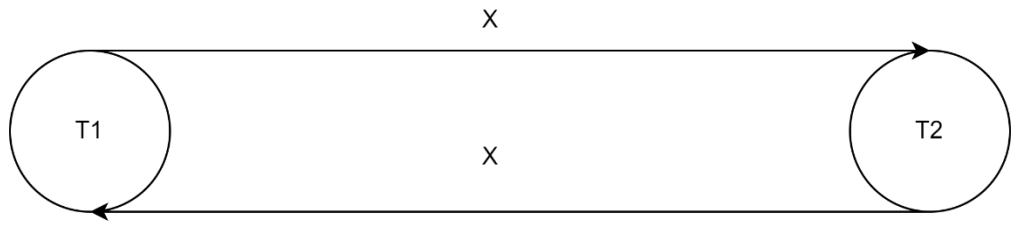
	T1	T2
t1	Start	
t2	Read(x)	
t3		Start
t4		Read(x)
t5	x=x-n	
t6	Write(x)	
t7		x=x+m
t8		write(x)
t9	Commit	
t10		Commit



Not conflict serializable due to the loop.

Schedule 4:

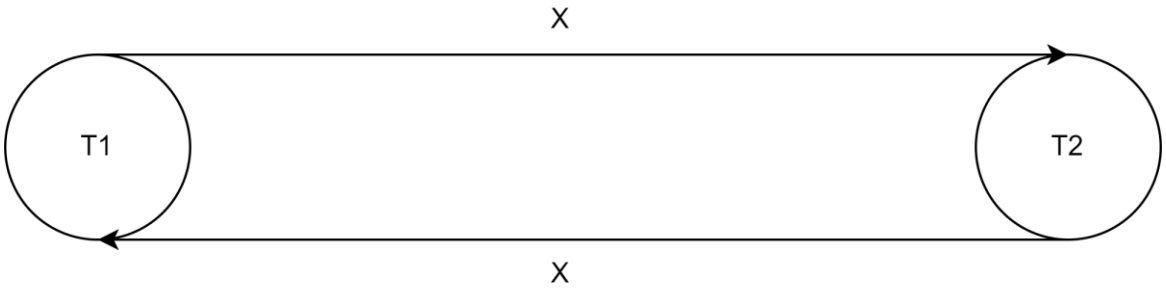
	T1	T2
t1		Start
t2		Read(x)
t3	Start	
t4	Read(x)	
t5		x=x+m
t6		write(x)
t7	x=x-n	
t8	Write(x)	
t9		Commit
t10	Commit	



Not conflict serializable as no loop exists.

Schedule 5:

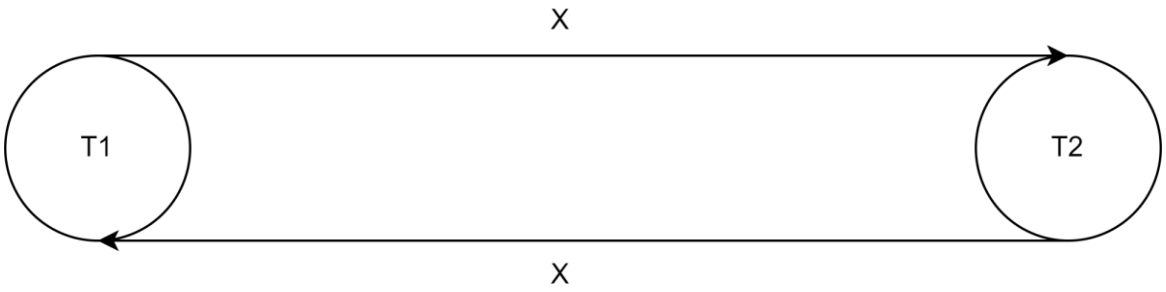
	T1	T2
t1	Start	
t2	Read(x)	
t3		Start
t4		Read(x)
t5		x=x+m
t6		write(x)
t7	x=x-n	
t8	Write(x)	
t9	Commit	
t10		Commit



Not conflict serializable as loop exists between 1 and 2.

Schedule 6:

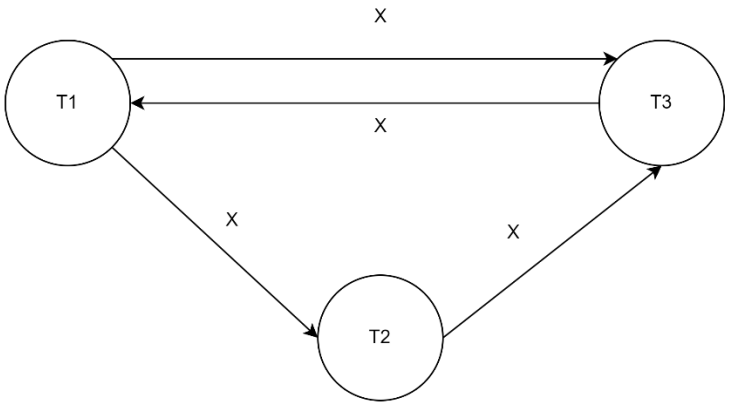
	T1	T2
t1		Start
t2		Read(x)
t3	Start	
t4	Read(x)	
t5	x=x-n	
t6	Write(x)	
t7		x=x+m
t8		write(x)
t9	Commit	
t10		Commit



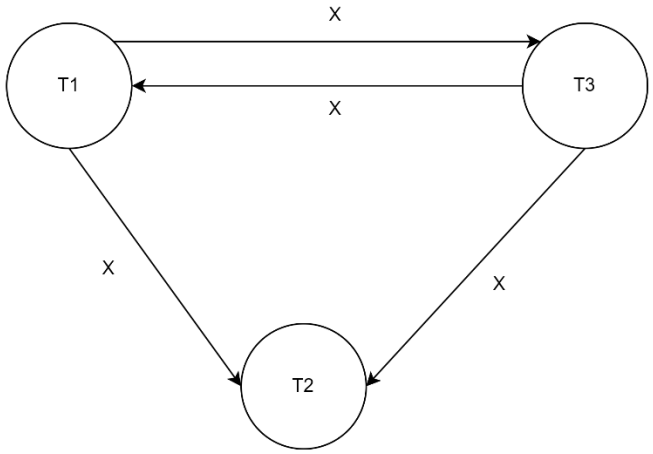
Not conflict serializable because of existence of loop.

2.

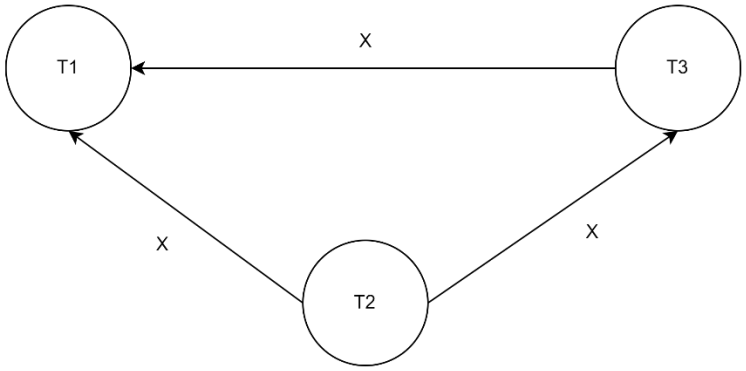
Schedule 1:



Schedule 2:



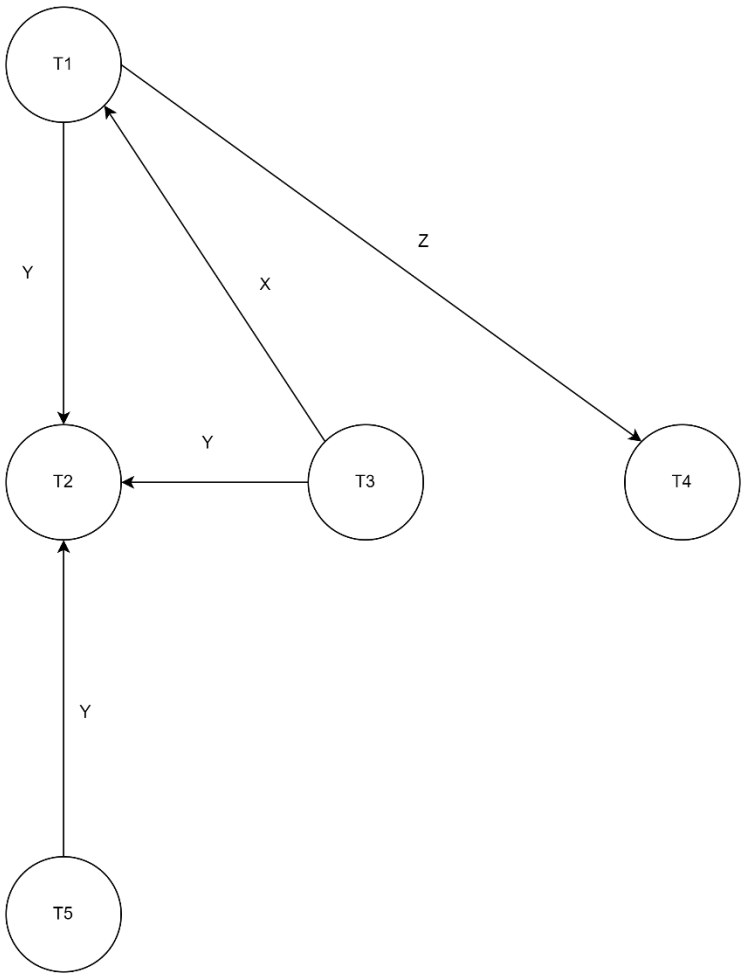
Schedule 3



3. Schedule 3 is conflict serializable as it does not have any loop in it.
   To achieve it, we can rearrange it in the following way.

   Schedule 3:
   T2 Read(X)
   T3 Read(X)
   T3 Write(X)
   T1 Read(X)
   T1 Write(X)

4.



5. A deadlock occurs when two or more transactions are waiting for each other to release locks on resources, creating a cycle of dependency where no transaction can proceed.
   Example Schedule
   Consider two transactions, T1 and T2, operating on two data items, A and B.

Time	Transaction	Operation
t1	T1	Lock(A) (Exclusive)
t2	T2	Lock(B) (Exclusive)
t3	T1	Lock(B) (Request)
t4	T2	Lock(A) (Request)

Description of the Issue

1. At t1, T1 acquires an **exclusive lock** on data item A.
2. At t2, T2 acquires an **exclusive lock** on data item B.
3. At t3, T1 requests an exclusive lock on B, which is held by T2. T1 is now waiting for T2 to release B.
4. At t4, T2 requests an exclusive lock on A, which is held by T1. T2 is now waiting for T1 to release A.
   Now:
   - T1 cannot proceed until T2 releases B.
   - T2 cannot proceed until T1 releases A.

   This creates a **circular wait**, where both transactions are waiting for the other to release locks, resulting in a **deadlock**.

Deadlock Graph Representation

The deadlock can be represented as a **wait-for graph**:
Since the graph contains a cycle, a deadlock exists.

How to Resolve or Prevent Deadlocks

1. **Deadlock Prevention**:
   - Enforce a **total order of resource allocation** (e.g., always request locks in a predefined order).
   - Use **timeouts**: Abort a transaction if it waits too long for a lock.
2. **Deadlock Detection**:

- Periodically check the wait-for graph for cycles and abort one transaction in the cycle.

3. **Deadlock Avoidance**:
   - Use techniques like the **Banker's Algorithm** to avoid unsafe states.

6.

Initial Transaction Timestamps: A:23, B:21, C:22

TIME	Transaction A	Transaction B	Transaction C	Valid/Invalid
11		READ(Z)		Valid
12		READ(Y)		Valid
13		WRITE(Y)		Valid
14			READ(Y)	Valid
15			READ(Z)	Valid
16	READ(X)			Valid
17	WRITE(X)			Valid
18			WRITE(Y)	Valid
19			WRITE(Z)	Valid
20	READ(V)			Valid
21	READ(Y)			Valid
22	WRITE(Y)			Valid

23		WRITE(V)		Invalid->Restart Transactions A, B and C
24		READ(Z)		Valid
25		READ(Y)		Valid
26		WRITE(Y)		Valid
27			READ(Y)	Valid
28			READ(Z)	Valid
29	READ(X)			Valid
30	WRITE(X)			Valid
31			WRITE(Y)	Valid
32			WRITE(Z)	Valid
33		WRITE(V)		Valid
34	READ(V)			Valid
35	READ(Y)			Valid
36	WRITE(Y)			Valid

Final Timestamp Table:

Object	Read Timestamp (RTS)	Write Timestamp (WTS)
V	26	24
X	26	26
Y	26	26
Z	25	25

Final Transaction Timestamps: A:26, B:24, C:25

7. **Two-Phase Locking Protocol (2PL)** is a concurrency control mechanism used in databases to ensure serializability (a key property for the correctness of transactions). It operates in two distinct phases:

   Growing Phase:
   A transaction can acquire locks (read or write locks) but cannot release any locks.

   Shrinking Phase:
   A transaction can release locks but cannot acquire any new locks.
   By ensuring this two-phase behaviour, 2PL prevents situations that could lead to **non-serializable** schedules.

   Types of Two-Phase Locking Protocols

   Basic Two-Phase Locking (Basic 2PL):
   In this form, the protocol ensures that all transactions follow the two-phase rule. However, it does not address issues like deadlocks.

   Strict Two-Phase Locking:
   In this variation, a transaction holds all its exclusive (write) locks until it commits or aborts. This helps prevent cascading rollbacks because no other transaction can read or write data modified by an uncommitted transaction.

Rigorous Two-Phase Locking:
In this stricter variation, a transaction holds all locks (read and write) until it commits or aborts. This provides serializability and ensures recoverability, making it easier to implement.

Conservative Two-Phase Locking (Static 2PL):
In this type, a transaction pre-acquires all the locks it will ever need at the beginning of its execution. If the required locks are not available, the transaction waits or aborts. This avoids deadlocks but can lead to reduced concurrency.

8. Cascading Rollback

A cascading rollback occurs when the failure of a transaction (and its subsequent rollback) forces other dependent transactions to also roll back. This happens when transactions read data that is later invalidated by the rollback of a preceding transaction.

Cascading rollbacks typically occur in databases that do not use strict or rigorous two-phase locking, as these allow transactions to read uncommitted data from other transactions.

Example of a Cascading Rollback
Let us consider three transactions T1, T2, and T3 operating on a database item X, with the following schedule:

Time	T1	T2	T3
t1	Write(X)		
t2		Read(X)	
t3			Read(X)
t4	Abort		

Explanation
1. At t1, T1 writes X=10 (but does not commit).
2. At t2, T2 reads X=10 a value written by T1 but not yet committed.
3. At t3, T3 also reads X=10, again an uncommitted value.
4. At t4, T1, invalidating the value X=10.

Now:

- T2 must roll back because it read the uncommitted value of X.
- T3 must also roll back because it read a value invalidated by T1's rollback.

9. The given schedule is recoverable and it is of type Cascade recoverable.

Transaction 1 reads the value x and in the next moment Transaction 3 reads x. But Transaction 3 commits only after Transaction 1 commits and both read same values of x from the database. Transaction 2 starts only after Transaction 1 commits. So, this is a recoverable schedule.
Cascade less:
Transaction 3 reads uncommitted data from Transaction 1. So, it is not cascade less.
Strict Recoverable:
Transaction 3 accesses uncommitted data from Transaction 1. So, no it is not Strict Recoverable.
Cascade Recoverable:
The schedule is recoverable and there are no cascading rollbacks if Transaction 1 commits successfully. So yes, it is Cascade recoverable.

This schedule is cascade recoverable. This is because the read in Transaction 3 reads uncommitted data from Transaction 1. This makes it a cascade recoverable schedule.