Schedules

Mirunalini.P

SSNCE

May 20, 2022

Table of Contents

Session Objective

- Schedules
- Schedules based on recoverability
- Schedules based on Serializability

Session Outcome

At the end of this session, participants will be able to

- Understand the scheduling
- Understand the schedule based on recoverability
- Understand the schedule based on Serializability

Schedules

Schedule or History: A schedule (or history) S of n transactions $T_1, T_2, ..., T_n$ is an **ordering of the operations** of the transactions subject to the constraint that:

- For each transaction T_i that participates in S, the operation of T_i in S must appear in the same order in which they occur in T_i
- Operations from different transactions can be interleaved in the schedule (such that the operations from transactions T_j can be interleaved with the operations of T_i in S).

The order of operations in S is considered to be **total ordering**.

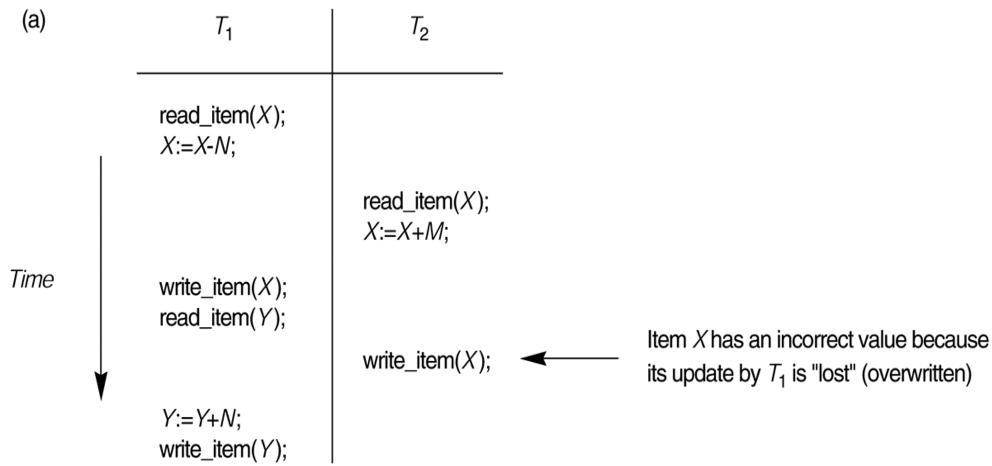
- For any two operations in the schedule, one must occur before the other.

Schedules

- Schedules can also be displayed in more compact notation
- Order of operations from left to right
- Include only read (r) and write (w) operations, with transaction id (1, 2, ...) and database item name (X, Y, ...)
- Can also include other operations such as b (begin), e (end), c (commit), a (abort)

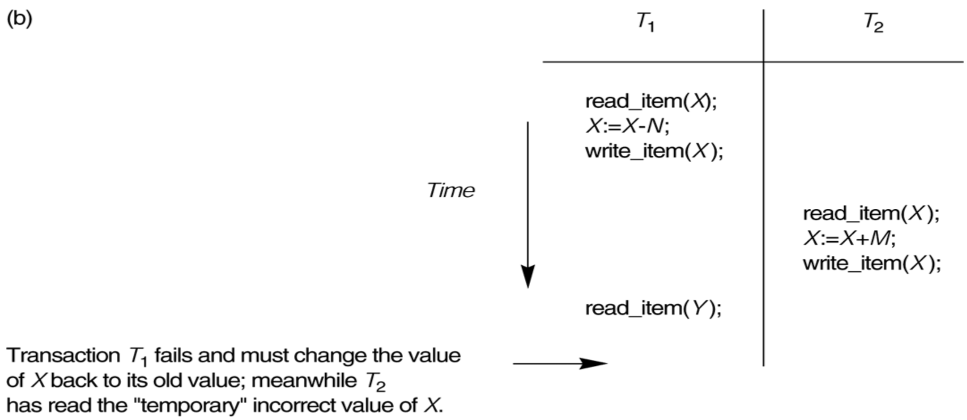
Eg: $\mathrm{S}_a : r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);$

Schedules - Example1



$$S_a : r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

Schedules - Example2



Assume the transaction T_1 is aborted

$S_b : r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1$

Conflicting Operations in a Schedule

Two operations in a schedule are said to be **conflict** if they satisfy all the three conditions:

- if they belong to different transactions
- if they access the same item X
- At least one of the operations is a write_item(X)

Conflicting and Non_Conflicting Operations in a Schedule

$S_a : r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$

Conflicting Operations:

- $r_1(X)$ and $w_2(X)$
- $r_2(X)$ and $w_1(X)$
- $w_1(X)$ and $w_2(X)$

Non_Conflicting Operations:

- $r_1(X)$ and $r_2(X)$
- $w_2(X)$ and $w_1(Y)$
- $r_1(X)$ and $w_1(X)$

Conflicting Operations in a Schedule

Two operations conflict if changing their order results in a different outcome

- **Read_Write Conflict** : Changing $r_1(X); w_2(X)$ to $w_2(X); r_1(X)$ means that T1 will read a different value for X
- **Write_Write Conflict**: Changing $w_1(Y); w_2(Y)$ to $w_2(Y); w_1(Y)$ means that the final value for Y in the database can be different
- **Read operations are not conflicting**: Changing $r_1(Z); r_2(Z)$ to $r_2(Z); r_1(Z)$ does not change the outcome

Complete Schedule

A schedule S of n transactions $T_1, T_2, T_3.......T_n$ is said to be **complete schedule** if the following conditions hold:

- The operation in S are exactly those operations in $T_1, T_2, T_3.......T_n$ including commit or abort operation as the last operation
- For any pair of operations from the same transaction T_i, their relative order of appearance in S is the same as their order of appearance in T_i
- For any two conflicting operations, one of the two must occur before the other in the schedule

Two non-conflicting operations to occur in the schedule without defining which occurs first leads to the **partial order** of the operations in the “n” transactions.

Characterizing Schedules Based on Recoverability

Schedules are characterized as follows:

- **Recoverable schedule:**
  - Once a transaction T is committed, it should never be necessary to roll back (T)
  - This ensures that the durability property of transactions is not violated.
  - A schedule S is **recoverable** if no transaction T in S commits until all transactions T' that have written some item X that T reads have committed.
- **Non-Recoverable schedule:**
  - A schedule where a committed transaction may have to be rolled back during recovery.
  - This violates Durability from ACID properties (a committed transaction cannot be rolled back) and so non-recoverable schedules should not be allowed.

Characterizing Schedules Based on Recoverability

Consider two schedules:

$Sc : r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$

$Sd : r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$

Sc is not recoverable, because T2 reads item X from T1, and then T2 commits before T1 commits.
If T1 aborts after the C2;

Then the value of X that T2 read is no longer valid and T2 must be aborted after it had already committed $--->$ **schedule not recoverable.**

For the schedule to be **recoverable** the c_2 operation is postponed until after T_1 commits

Sd is recoverable

Cascading Rollback

An uncommitted transaction has to be rolled back because it read an item from a transaction that failed. This phenomenon is known as **Cascading rollback**

$Se : r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$

The transaction T_2 has to be rolled back because it read item X from T_1 which is failed transactions and T_1 then aborted

Cascadless

A schedule is said to be a **Cascadless schedule**, or to avoid cascading rollback, if every transaction in the schedule **reads only items** that were written by committed transactions.

$Se : r_1(X);\ w_1(X); r_2(X);\ r_1(Y); w_2(X); w_1(Y); a_1; a_2;$

To avoid rollback :
Then $r_2(X)$ must be postponed untill after T_1 has committed (aborted), thus delaying T_2.

Strict Schedule

- A schedule in which a transaction $T2$ can either **read or write** an item X until the transaction $T1$ that last wrote X has committed
- Simplify the recovery process by simply undoing a write_item(X) of an aborted transaction to restore the before image of data item X.
- **Sf:W1(X,5);W2(X,8);a1**
- if T1 aborts, the recovery procedure recovers the before image which was orginally 9, even it has been changed to 8 by transaction T2.
- S_f is not **strict scheule** since it permits T2 to write item X even though the last wrote X had not commited

Schedule C below is **cascadeless and also strict**

Schedule D is cascadeless, but not strict (which writes the value of X before T1 commits)

Schedule E is cascadeless and strict, to make it strict and cascadless, w2(X) and r2(x) must be delayed until after T3 commits

Schedule C: $r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; r_2(X); w_2(X);$

Schedule D: $r_1(X); w_1(X); w_3(X); r_1(Y); w_1(Y); c_1; r_2(X); w_2(X);$

Schedule E: $r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; w_3(X); r_2(X); w_2(X);$

Summary

- The set of all possible schedules can be partitioned into two subsets: **recoverable and non-recoverable**
- A **cascadeless** schedule will be the subset of the **recoverable** schedules.
- A **strict schedules** will be the subset of **cascadless** schedules.
- All strict schedules are cascadless and all cascadless schedules are recoverable.

Serializability of Schedules

- Consider two transactions T1 and T2 which is submitted at the same time.
  If no interleaving is permitted then there are two possible ways:
  - Execute all the operations of T1 and then T2
  - Execute all the operations of T2 and then T1
- **Serializability theory**, attempts to determine which schedules are "correct" and which are not and to develop techniques that allow only correct schedules

Serial Schedules

(a)

Time	T_1	T_2

↓	read_item(X);
	$X := X - N$;
	write_item(X);
	read_item(Y);
	$Y := Y + N$;
	write_item(Y);	

		read_item(X);
		$X := X + M$;
		write_item(X);

Schedule A

(b)

Time	T_1	T_2

↓		read_item(X);
		$X := X + M$;
		write_item(X);

	read_item(X);
	$X := X - N$;
	write_item(X);
	read_item(Y);
	$Y := Y + N$;
	write_item(Y);	

Schedule B

Schedule A and B are called as Serial Schedule

Serial Schedules

- Schedules A and B are called serial schedules because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction.
- In a serial schedule:
  - Transactions are performed in serial order
  - Only one transaction is active at any time
  - Commit or abort of the active transactions initiates execution of next transactions
  - Since transactions are independent every serial schedule produce correct result

Serial Schedules: Disadvantage

- Serial schedules are not feasible for performance reasons:No interleaving of operations.
- Long transactions force other transactions to wait.
- System cannot switch to other transaction when a transaction is waiting for disk I/O or any other event.
- Need to allow concurrency with interleaving without sacrificing correctness

Non - Serial Schedules

Time

T_1	T_2
read_item(X); X:=X-N;	
	read_item(X); X:=X+M;
write_item(X); read_item(Y);	
	write_item(X);
Y:=Y+N; write_item(Y);	

Schedule C

T_1	T_2

read_item(X);
X:=X-N;
write_item(X);	

	read_item(X);
	X:=X+M;
	write_item(X);

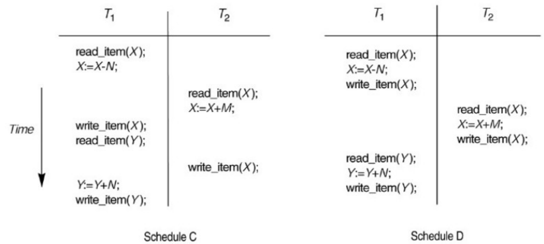
read_item(Y);
Y:=Y+N;
write_item(Y);	

Schedule D

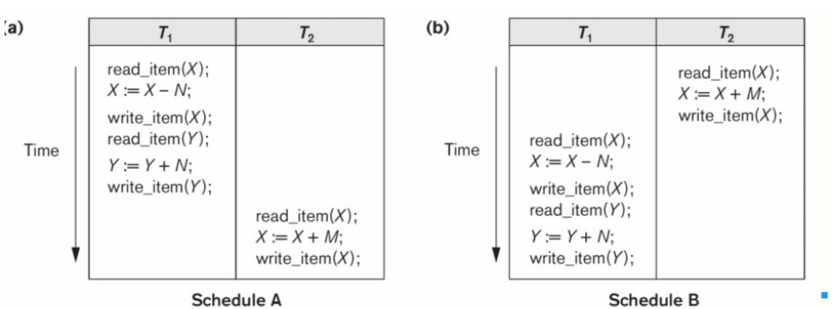
Two nonserial schedules C and D with interleaving of operations

Serial & Non - Serial Schedules

X=90 and Y=90, N=3 and M=2 serial Schedule A, B expect database values X=89 and Y=93. C produces the value X=92 and Y=93



Two nonserial schedules C and D with interleaving of operations



Characterizing Schedule based on Serializability

Schedules C and D are nonserial because of interleaving operations from two transactions.

Some non-serial schedules give correct results.

We have to determine which of the non-serial schedule always give a correct result.

A schedule S of n transactions is **serializable** if it is equivalent to some serial schedule of the same n transactions.

Question is : When are two schedules considered "equivalent" ?

- Result equivalent
- Conflict equivalent - Conflict Serializable
- View equivalent - View Serializable

Result Equivalent

- Two schedules are called result equivalent if they produce the same final state of the database.
- Disadvantages:
  - Not all the result equivalent schedules produces same final state.
  - Cannot be used to define equivalence of schedules.

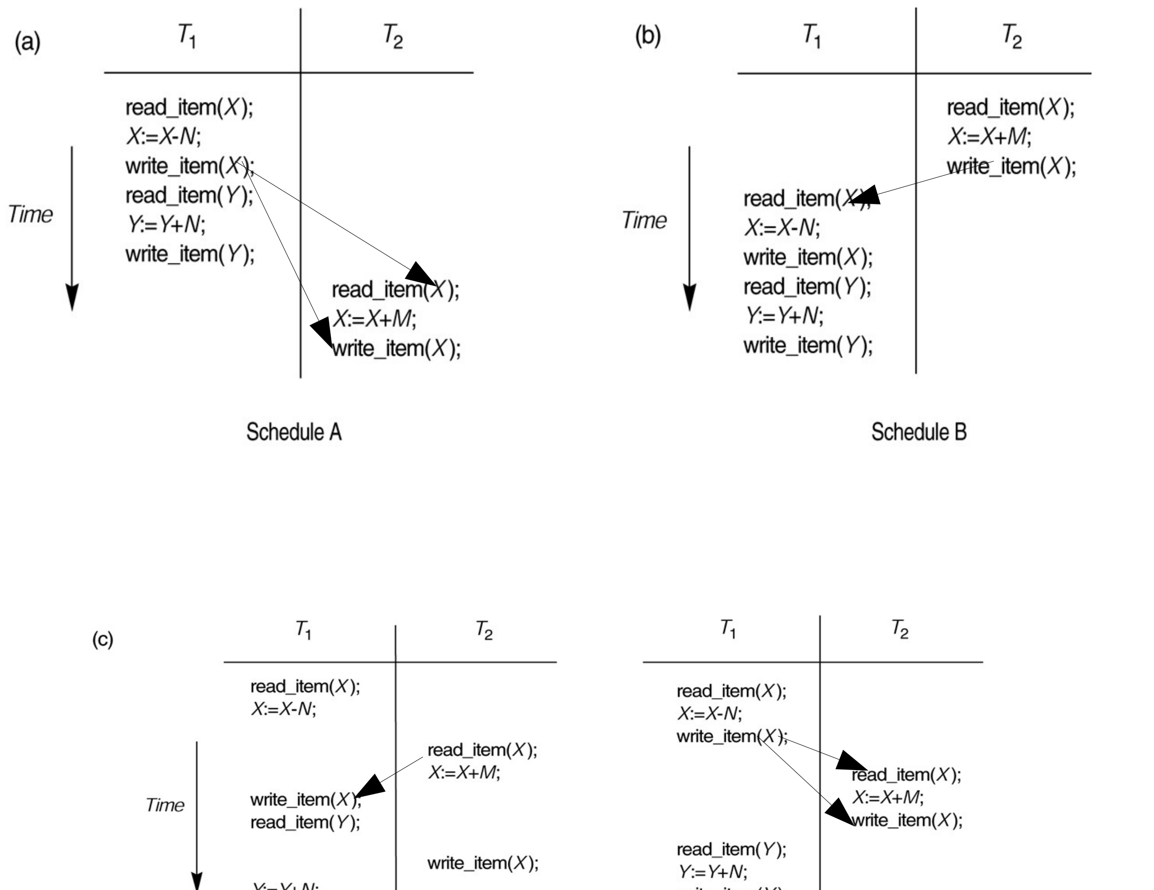
S_1	S_2
read_item(X);	read_item(X);
X:=X+10;	X:=X*1.1;
write_item(X);	write_item(X);

Two schedules that are result equivalent for the initial value of X = 100 but are not result equivalent in general.

Conflict Equivalent

- Two schedules are said to be **conflict equivalent** if the order of any two conflicting operations is the same in both schedules.
- According to this, schedule D is equivalent to the serial schedule A. Since A is serial schedule and D is equivalent to A, **D is serializable schedule.**
- We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule, hence D is conflict serializable.
- Conflict equivalence can be obtained using
  - Ordering of conflict operations
  - Swapping of non-conflict operations
  - Precedence Graph

Conflict Equivalent - Ordering of conflict operations



Conflict Equivalent - Ordering of conflict operations

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

S_1

S_1

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

S_2

Conflict Equivalent - Swapping of non-conflict operations

If a schedule S can be transformed into an equivalent schedule S' by a series of swaps of non-conflicting instructions, we saythat S and S' are conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Conflict Equivalent

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

equivalent to S_1

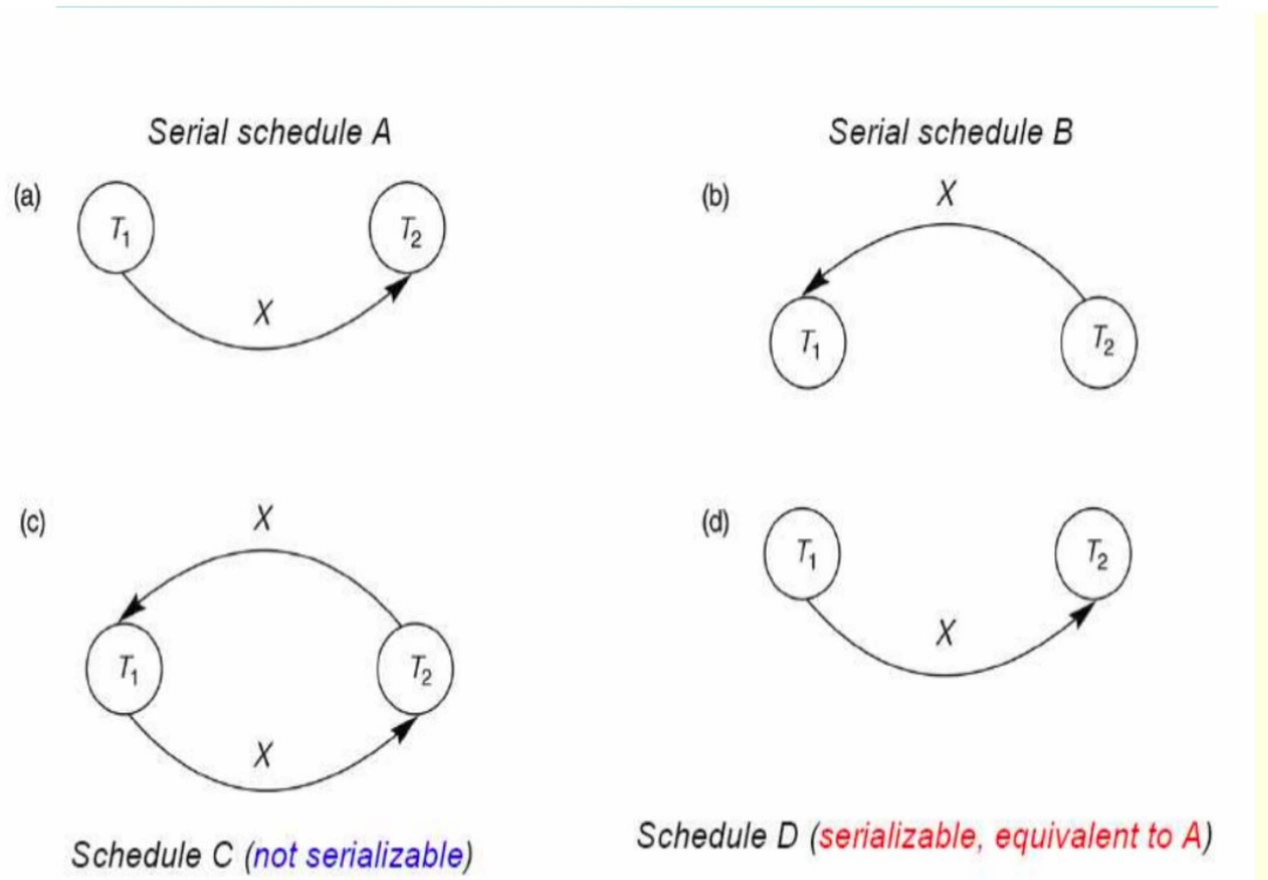
Conflict Equivalent - Precedence Graph

- The algorithm looks at only the read_item and write_item operations in a schedule to construct a precedence graph (or serialization graph),
- The graph is a directed graph $G = (N, E)$ that consists of a set of nodes $N = T_1, T_2, ..., T_n$ and a set of directed edges $E = e_1, e_2, ..., e_m$.
- There is one node in the graph for each transaction T_i in the schedule.
- Each edge e_i in the graph is of the form $(T_j \rightarrow T_k)$, where T_j is the starting node of e_i and T_k is the ending node of e_i.
- Edge from node T_j to node T_k is created by the algorithm if a pair of conflicting operations exist in T_j and T_k
- The conflicting operation in T_j appears in the schedule before the conflicting operation in T_k .

Testing for conflict serializability

- For each transaction T_i participating in s Schedule S, create a node labeled T_i in the precedence graph.
- An edge is created from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
- The schedule is **serializable** if and only if the **precedence graph** has no cycles.

Precedence Graph



View Equivalent

- A less restrictive definition of equivalence of schedules is called view equivalence.
- A schedule is view serializable if it is view equivalent to a serial schedule.
- Schedules are said to be view equivalent if the following three conditions hold:
  - The same set of transactions participates in S and S' and S and S' includes same operations of those transactions.
  - For any operation, $R_i(X)$ of T_i in S, if the value of X read was written by an operation W_j(X) of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation R_i(X) of Ti in S'. (Initial read and update read)
  - If the operation $W_k(Y)$ of T_k is the last operation to write item Y in S, then $W_k(Y)$ of T_k must also be the last operation to write item Y in S'. **(Final write)**

View Serializable

- **Constrained writes**: The value written by w(X) in Ti depends only on the value of X read by r1(X) in Ti
- **Blind write**: The value written by w(X) in Ti is independent of its old value, so it is not preceded by a read of X in the transaction T

T1	T2	T3
r1(x)		
w1(x)		
	$w2(X) -> Blindwrite$	
		$w3(X) -> Blindwrite$

Table 1: Example -Blind Write

Reference

Fundamentals of Database systems 7^{th} Edition by Ramez Elmasri.