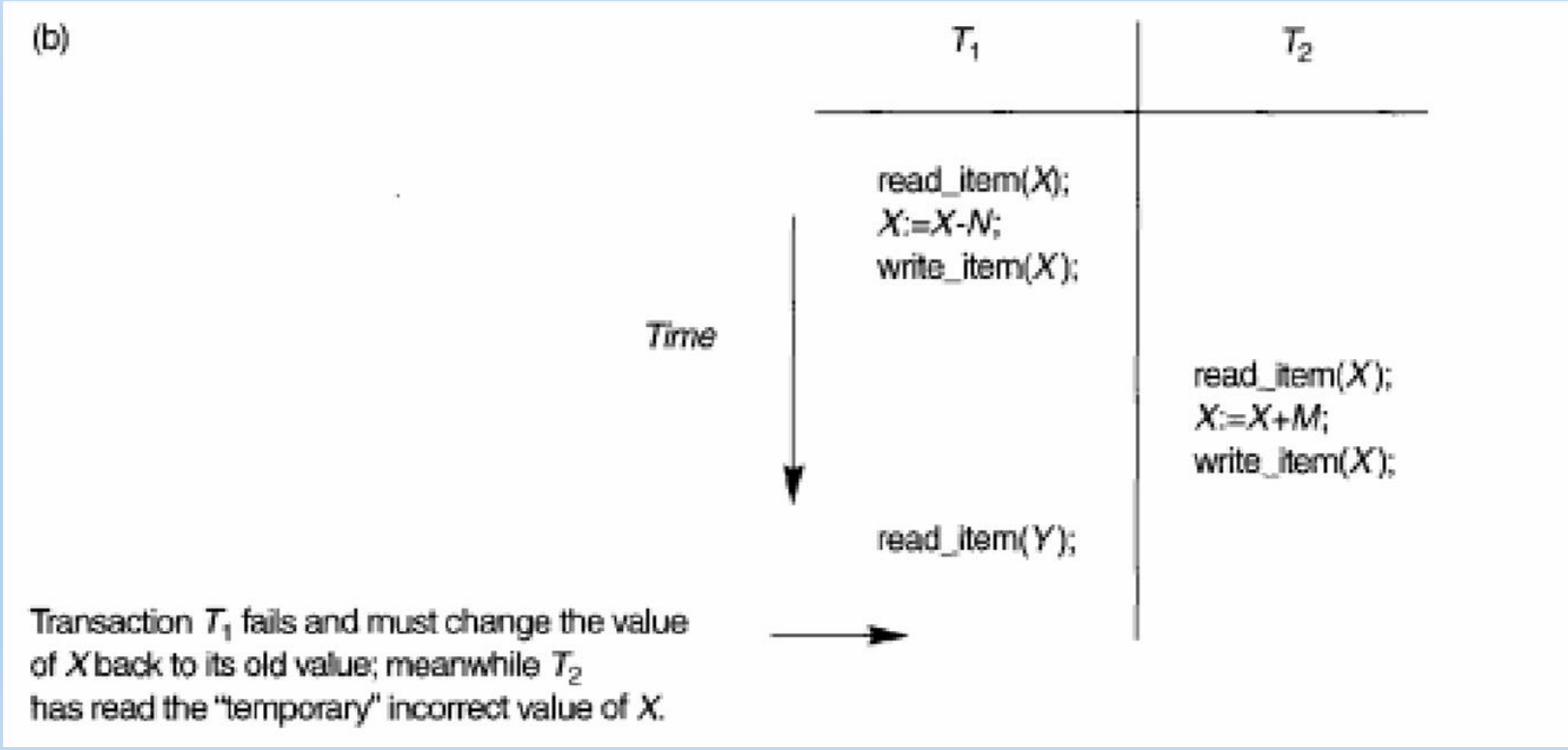
CONCURRENCY CONTROL
IN
TRANASACTION MANAGEMENT

Types of problems in concurrency

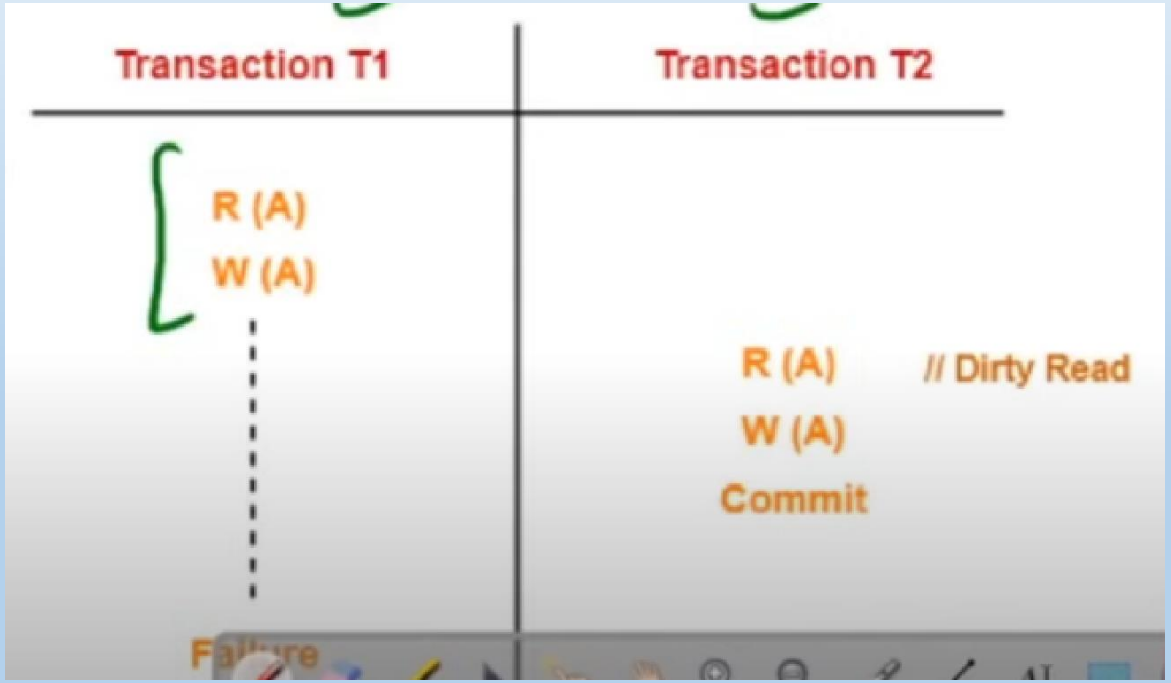
- Dirty read
- Incorrect summary
- Lost update
- Unrepeatable read
- Phantom read

Dirty Read or temporary read problem

This problem occurs when one transaction updates a database item and then the transaction fails for some reason .The updated item is accessed by another transaction before it is changed back to its original value.



Dirty Read or Uncommited Read or RAW



The Incorrect Summary Problem.

one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.(Eg. Seeing different remaing available seats in train reservation by different users)

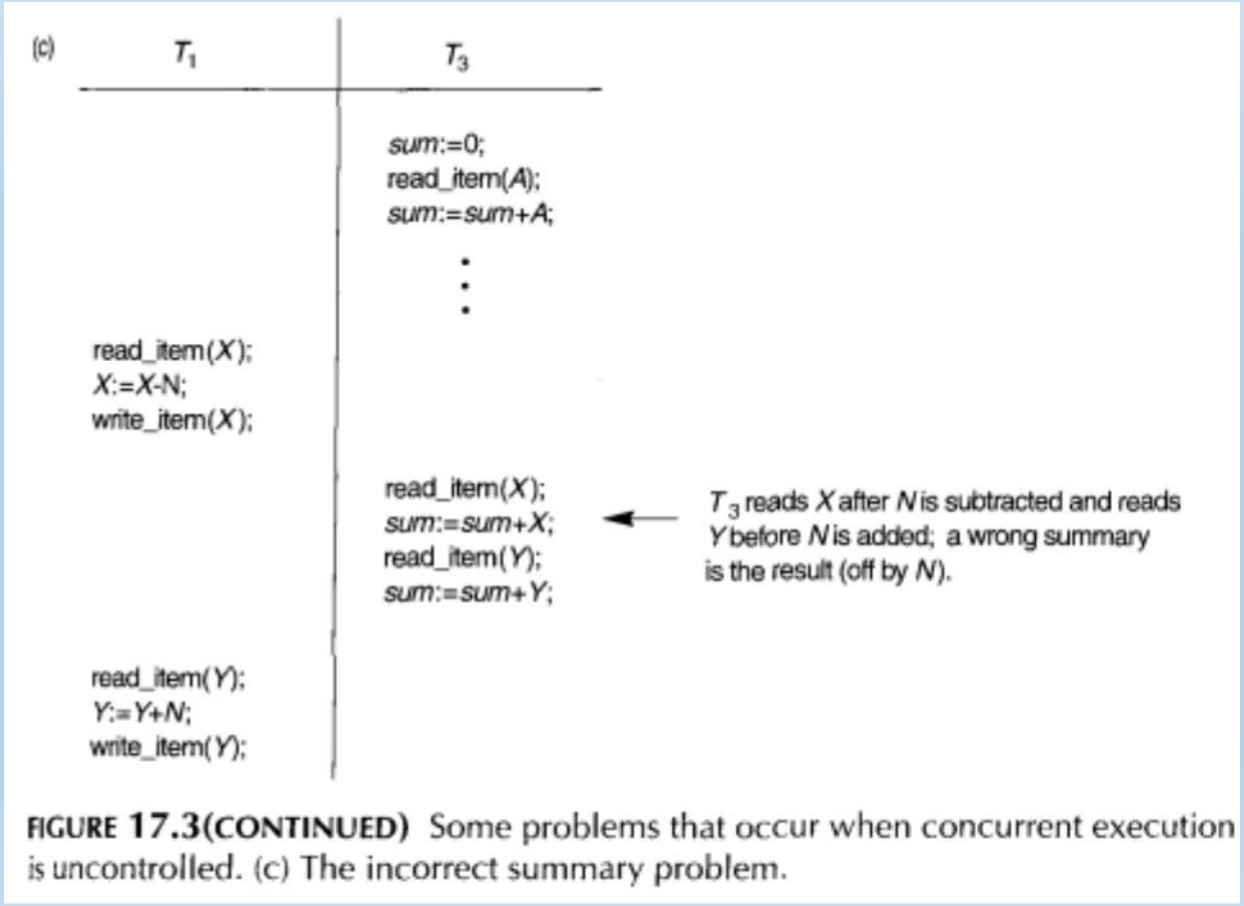


FIGURE 17.3(CONTINUED) Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.

- For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing.
- If the interleaving of operations shown in Figure occurs, the result of T3 will be off by an amount N because T3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it

Incorrect summary

Transaction T1	Transaction T2	A = 1000, B = 1000, C = 1000

	sum = 0;
	avg = 0;
	read(C);
	sum := sum + C;
		sum = 0
		avg = 0
		T2 read: C = 1000
		sum = 1000
		T1 read: A = 1000

read(A);
A := A – 50;
write(A);
		T1 write: A = 950

	read(A); *sum := sum + A;* *read(B);* *sum := sum + B;* *avg := sum/3;* *commit;*	T2 read: A = 950 sum = 1950 t2 read: B = 1000 sum = 2950 avg = 983.33

read(B);
B := B + 50;
write(B);
commit;
		T2 read: B = 1000
		T2 write: B = 1050

Incorrect summary

Transaction T1	Transaction T2	A = 1000, B = 1000, C = 1000

	sum = 0;
	avg = 0;
	read(C);
	sum := sum + C;
		sum = 0
		avg = 0
		T2 read: C = 1000
		sum = 1000
		T1 read: A = 1000

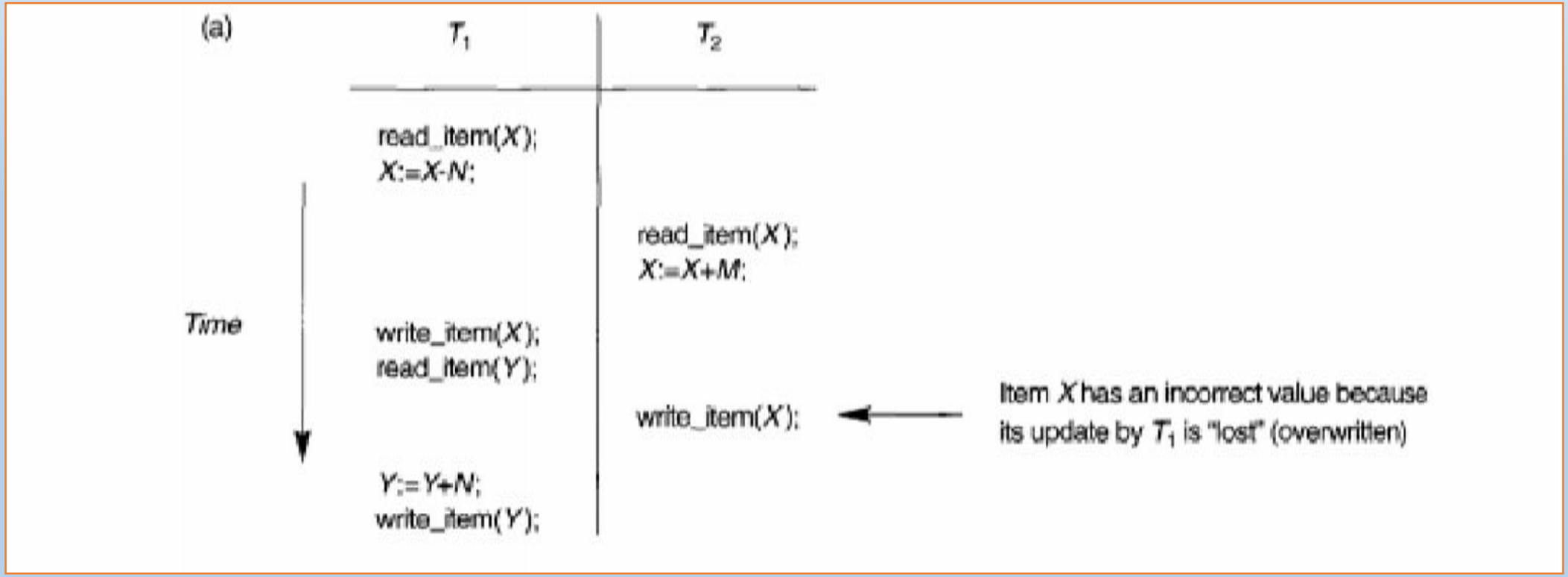
read(A);
A := A – 50;
write(A);
		T1 write: A = 950

	read(A); *sum := sum + A;* *read(B);* *sum := sum + B;* *avg := sum/3;* *commit;*	T2 read: A = 950 sum = 1950 2 read: B = 1000 sum = 2950 vg = 983.33

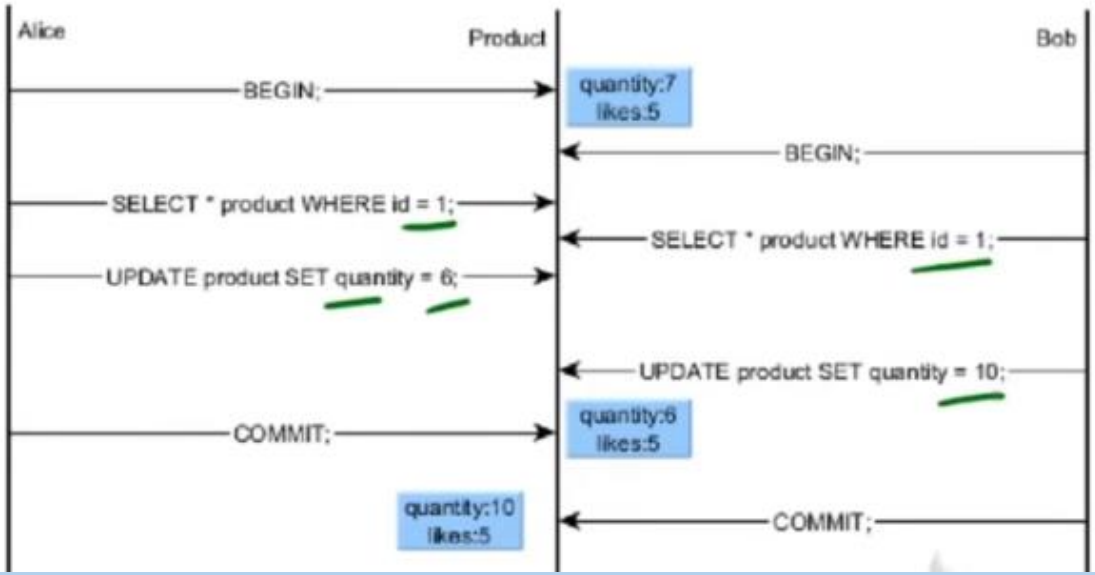
read(B);
B := B + 50;
write(B);
commit;
		T2 read: B = 1000
		T2 write: B = 1050

The lost update problem.

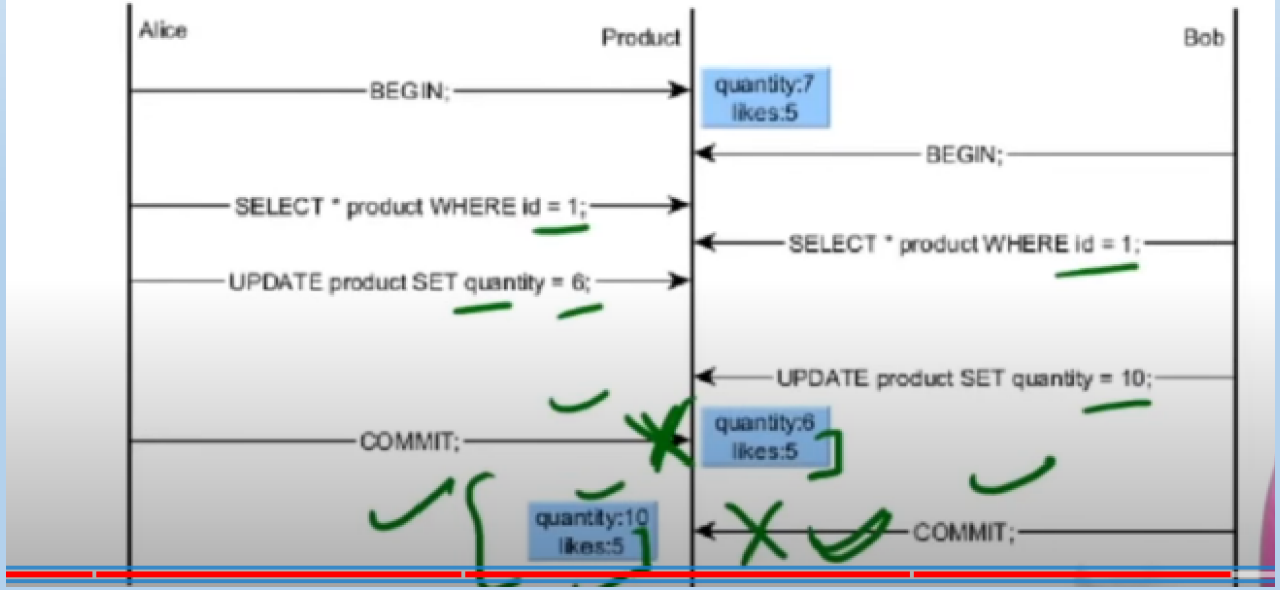
The Lost Update Problem. This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.



Lost Update



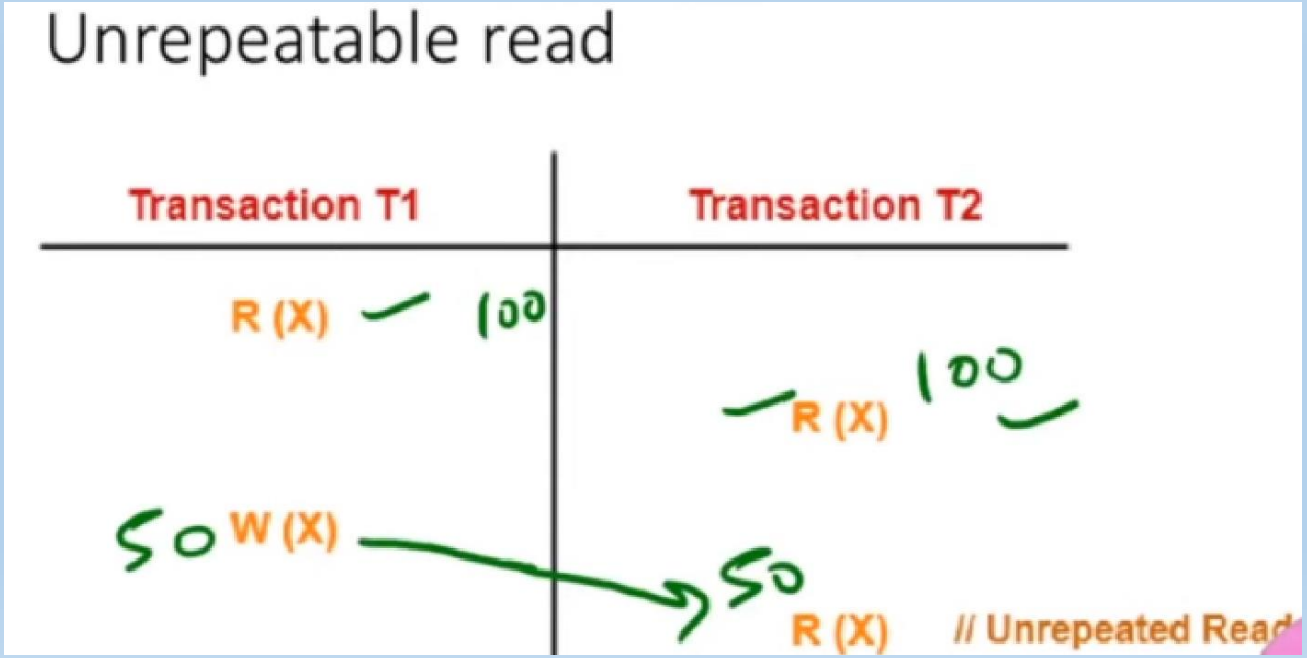
Lost Update



Unrepeatable read

A transaction T reads an item twice and the item is changed by another transaction T' between the two reads. Hence, T receives different values for its two reads of the same item.

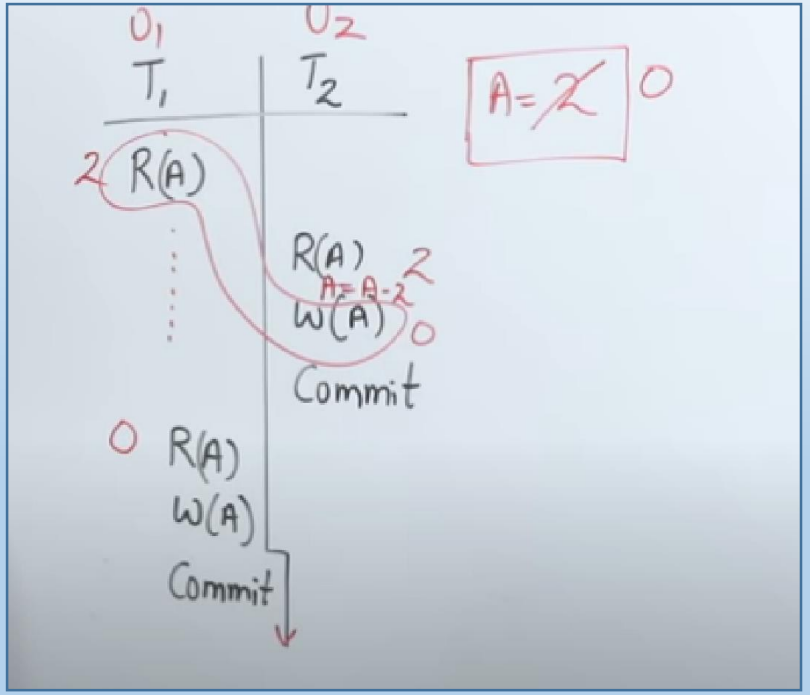
This may occur, for example, if during an airline reservation transaction, a customer is inquiring about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation.



Read - Write Conflict or unrepeatable read

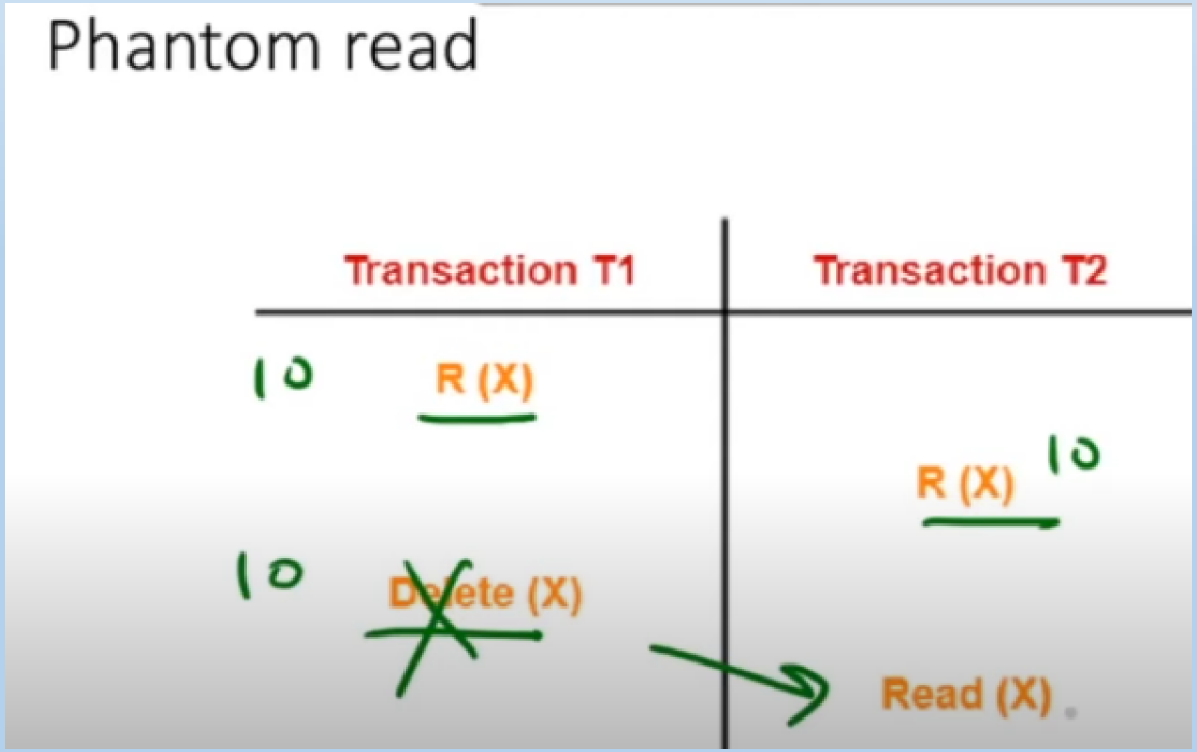
Same Data	
R	R
R	W
W	R
W	W

T_1	T_2
R(A)	
	R(A)
	W(A)
	Commit
R(A)	
W(A)	
Commit	



Phantom Read

A transaction TI may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction Tz inserts a new row that also satisfies the WHERE-clause condition used in TI , into the table used by TI' If TI is repeated, then TI will see a phantom, a row that previously did not exist.



CONCURRENCY CONTROL TECHNIQUES

1. Locks
2. Binary Locks
3. Shared/Exclusive locks
4. 2 – Phase Locking protocol
5. Time-stamp Ordering protocol

1.LOCKS

Definition :A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.

- Generally, there is one lock for each data item in the database.
- Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Types of Locks

1. **Binary locks**
2. **Shared/Exclusive locks**

2. Binary Locks

- A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity).
- A distinct lock is associated with each database item X.
- If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item.
- If the value of the lock on X is 0, the item can be accessed when requested.
- current value (or state) of the lock associated with item X is LOCK(X).
- Two operations, lock_item and unlock_item, are used with binary locking.

lock_item(X)

A transaction requests access to an item X by first issuing a lock_item(X) operation.

- If LOCK(X) = 1, the transaction is forced to wait.
- If LOCK(X) = 0, it is set to 1 (the transaction locks the item) and the transaction is allowed to access item X.

Lock_tem(X):

```
B:ifLOCK(X)=O (* item is unlocked *)
            then LOCK(X)□1 (*lock the item*)
Else
            begin
            wait (until lock(X)=O and the lock manager wakes up the transaction);
            Goto B
end;
```

unlock_item(X)

When the transaction is through using the item, it issues an un1ock_item(X) operation, which sets

LOCK(X) to 0 (unlocks the item) so that X may be accessed by other transactions

unlock_item (X):

LOCK (X)□O; (* unlock the item*)

If any transactions are waiting then wakeup one of the waiting transactions;

- Hence, a binary lock enforces mutual exclusion on the data item.

- Binary lock can be implemented easily using a binary valued variable, LOCK, associated with each data item X in the database.
- Each lock can be a record with three fields: **<data item name, LOCK, locking transaction>** plus a queue for transactions that are waiting to access the item.
- The system needs to maintain only these records for the items that are currently locked in a lock table.
- Items not in the lock table are considered to be unlocked.
- The DBMS lock manager subsystem keep track of and control access to locks.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction T must issue the operation lock_i tem(X) before any read_i tem(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unl ock_i tem(X) after all read_i tem(X) and wri te_i tem(X) operations are completed in T.
3. A transaction T will not issue a 1ock_i tem(X) operation if it already holds the lock on item X.
4. A transaction T will not issue an unl ock_i tern(X) operation unless it already holds the lock on item X.

3. Shared/Exclusive (or Read/Write) Locks.

- Binary locking scheme is too restrictive for database items, because at most one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for reading purposes only.
- However, if a transaction is to write an item X, it must have exclusive access to X.

For this purpose, a different type of lock called a multiple mode lock is used.

- In this scheme-called shared/exclusive or read/write locks-there

are **three locking operations: read_lock(X), write_lock(X), and unlock(X).**

- For item X, LOCK(X), now has three possible states: "read-locked," "write locked," or "unlocked."
- A read-locked item is also called share-locked, because other transactions are allowed to read the item.
- A write-locked item is called exclusive-locked, because a single transaction exclusively holds the lock on the item.

- Each record in the lock table will have four fields: <data item name, LOCK, no_of_reads, locking transactions>
- If LOCK(X)=write-locked, the value of locking transactions is a single transaction that holds the exclusive (write) lock on X.
- If LOCK(X)=read-locked, the value of locking transactions is a list of one or more transactions that hold the shared (read) lock on X.
- The three operations read_lock(X), write_lock(X), and unlock(X)

```
read_lock (X):
  B: if LOCK (X)="unlocked"
     then begin LOCK (X)← "read-locked";
                no_of_reads(X)← 1
          end
     else if LOCK(X)="read-locked"
          then no_of_reads(X)← no_of _reads(X) + 1
          else begin wait (until LOCK (X)="unlocked" and
                    the lock manager wakes up the transaction);
               go to B
               end;
```

```
write_lock (X):
  B: if LOCK (X)="unlocked"
     then LOCK (X)← "write-locked"
     else begin
          wait (until LOCK(X)="unlocked" and
            the lock manager wakes up the transaction);
          go to B
          end;
```

```
unlock (X):
  if LOCK (X)="write-locked"
     then begin LOCK (X)← "unlocked;"
          wakeup one of the waiting transactions, if any
          end
     else if LOCK(X)="read-locked"
          then begin
               no_of_reads(X)← no_of_reads(X) – 1;
               if no_of_reads(X)=0
                 then begin LOCK (X)="unlocked";
                      wakeup one of the waiting transactions, if any
                      end
               end;
```

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

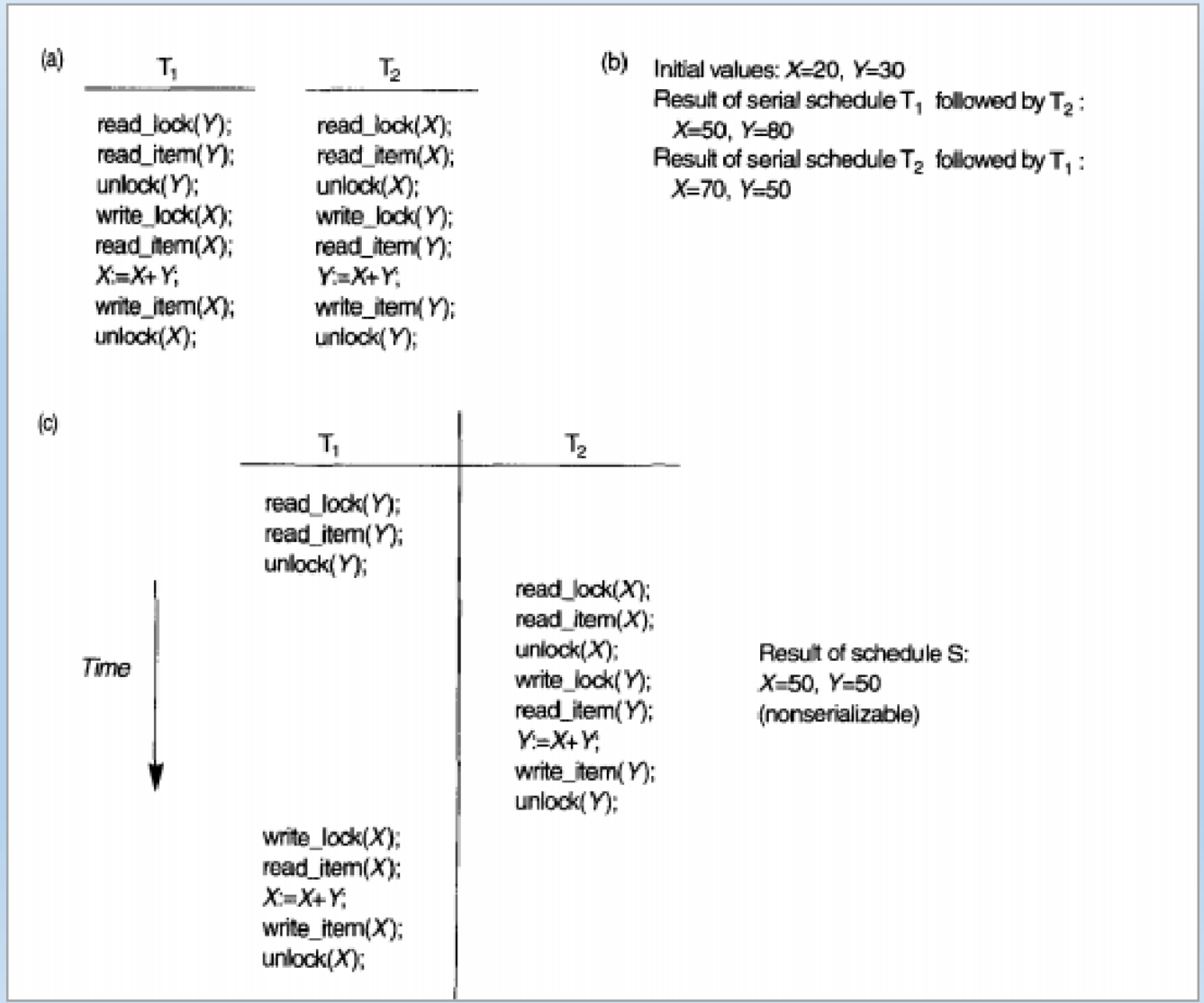
1. A transaction T must issue the operation `read_lock(X)` or `write_lock(X)` before any `read_item(X)` operation is performed in T.
2. A transaction T must issue the operation `write_lock(X)` before any `write_item(X)` operation is performed in T.
3. A transaction T must issue the operation `unlock(X)` after all `read_item(X)` and `write_item(X)` operations are completed in T.[3]
4. A transaction T will not issue a `read_lock(X)` operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed, as
5. A transaction T will not issue a `write_lock(X)` operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may be relaxed, as we discuss shortly.
6. A transaction T will not issue an `unlock(X)` operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks

- Sometimes it is desirable to relax conditions 4 and 5 to allow lock conversion; that is, a transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another.
- It is possible for a transaction T to issue a read_lock(X) and then later on to upgrade the lock by issuing a write_l ock(X) operation. If T is the only transaction holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded otherwise, the transaction must wait.
- It is also possible for a transaction T to issue a wri te_l ock(X) and then later on to downgrade the lock by issuing a read_lock(X) operation.
- The descriptions of the read_lock(X) and wri te_l ock(X) operations must be changed appropriately.

Problems of Locks

- Using binary locks or read/write locks in transactions, **does not guarantee serializability of schedules** on its own.



This is because in Figure a the items Yin T1and X in T 2 were unlocked too early

To guarantee serializability, we must properly position the locking and unlocking operations in every transaction.

Transactions that do not obey two-phase locking. (a) Two transactions T1 and T2. (b) Results of possible serial schedules of T1 and T2. (c) A nonserializable schedule S that uses locks

'Shared - Exclusive Locking'

→ Shared Lock (S) => if transaction locked data item in Shared mode then allowed to read only.

→ Exclusive Lock(X) => if transaction locked data item in Exclusive mode then allowed to Read and write both.

* Problems in S/X locking

1) May not sufficient to produce only Serializable schedule.
2) May not free from Irrecoverability
3) May not free from deadlock.
4) May not free from Starvation.

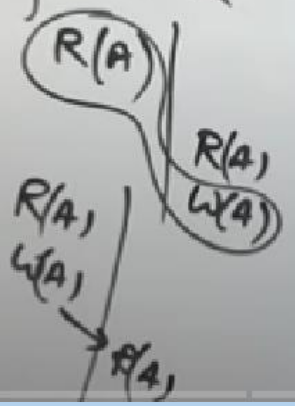
T_1	T_2
S(A)	X(A)
R(A)	R(A)
U(A)	W(A)
	U(A)

(A)

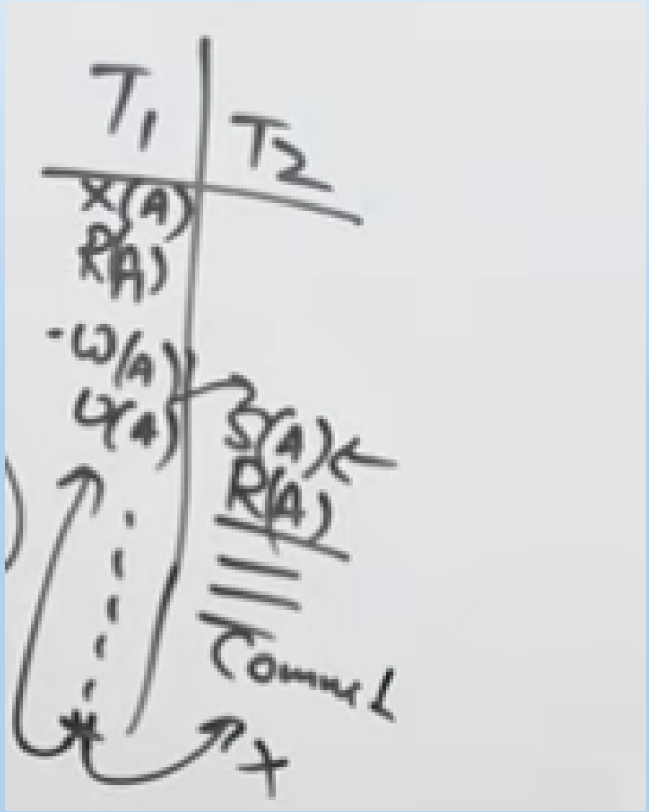
→ Request

↓ grant	S	X
S	✓ Yes	No
X	No	No

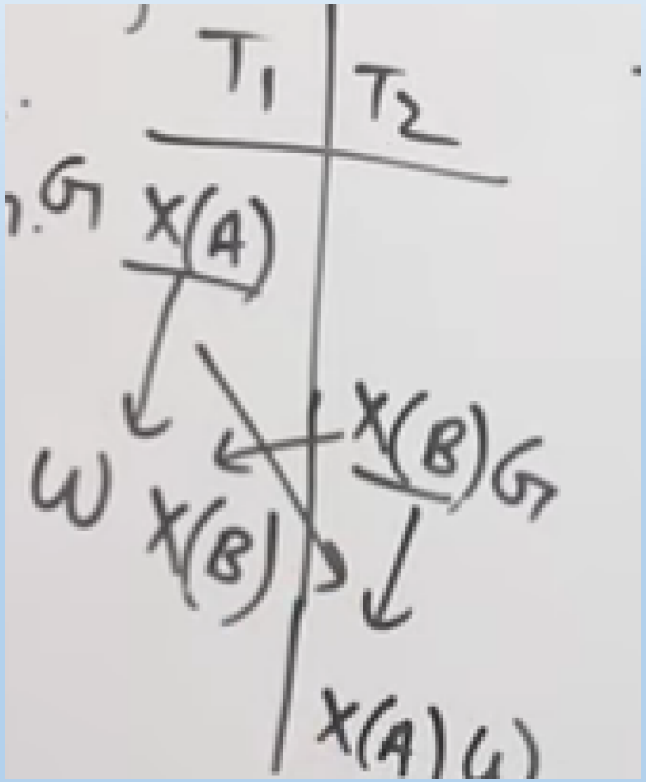
R-R



Irrecoverable Schedule
(Dirty read problem exists)

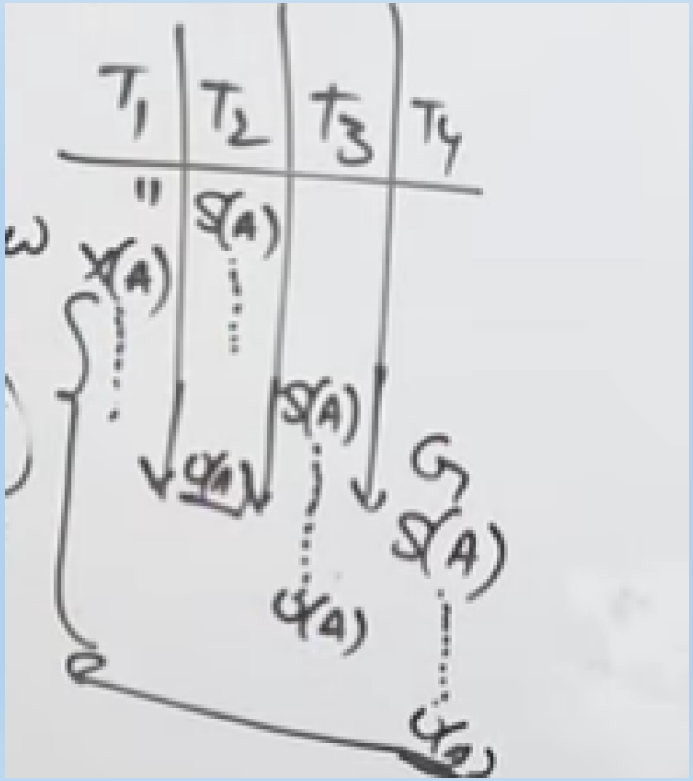


Deadlock exists
Infinite waiting of T1 and T2 on locks



Starvation :

1. T2 locks in S mode.
2. T1 wants lock in X mode and waits
3. T3 locks in S mode(granted)
4. T2 unlocks and T3 uses but T1 waits
5. T4 locks in S mode(granted)
6. T3 unlocks but T1 waits as T4 locked it
7. ……
8. **T1 waits indefinitely(Starvation)**



4. Two-Phase locking (2PL)protocol

A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.

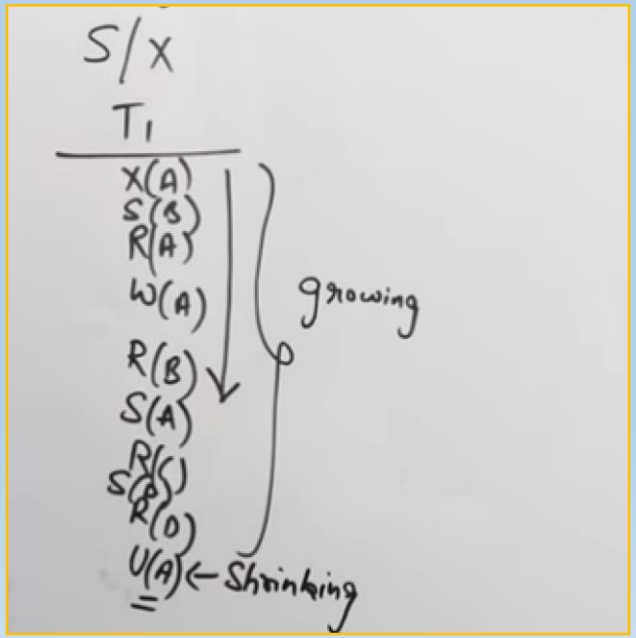
- Such a transaction can be divided into two phases:

  1. An expanding or growing (first) phase

     New locks on items can be acquired but none can be released;

  2. A shrinking (second) phase

     locks can be released but no new locks can be acquired.



T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Does not follow 2PL

T_1'	T_2'
read_lock (Y);	read_lock (X);
read_item (Y);	read_item (X);
write_lock (X);	write_lock (Y);
unlock (Y);	unlock (X);
read_item (X);	read_item (Y);
X:=X+Y;	Y:=X+Y;
write_item (X);	write_item (Y);
unlock (X);	unlock (Y);

FIGURE 18.4 Transactions T_1' and T_2', which are the same as T_1 and T_2 of Figure 18.3 but which follow the two-phase locking protocol. Note that they can produce a deadlock.

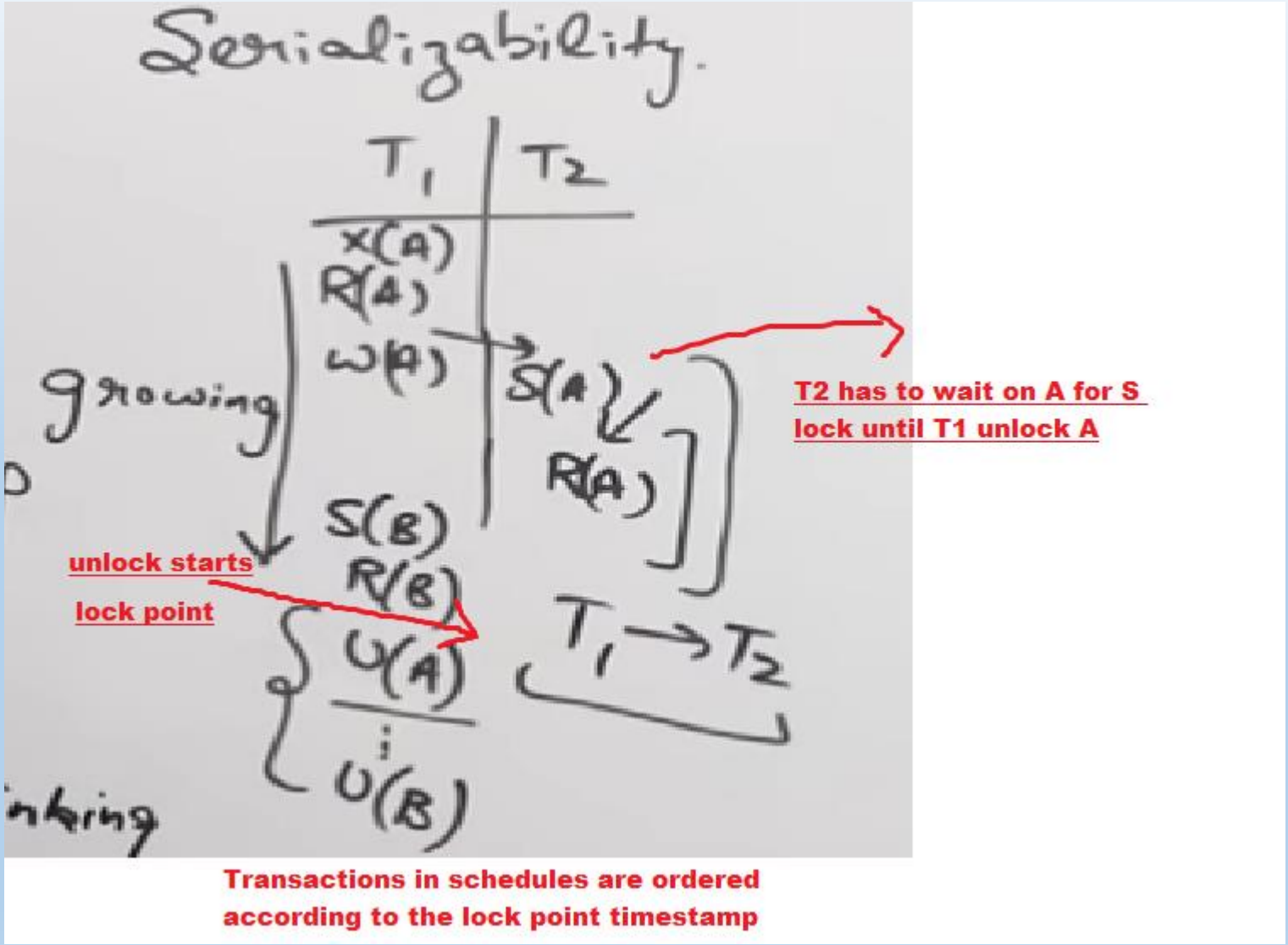
- It can be proved that, if every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable.
- Two-phase locking may reduce the amount of concurrency that can occur in a schedule.

➢ This is because a transaction T may not be able to release an item X after it has finished using it if T must lock an item Y later on. Or ,T must lock the additional item Y before it needs it so that it can release X

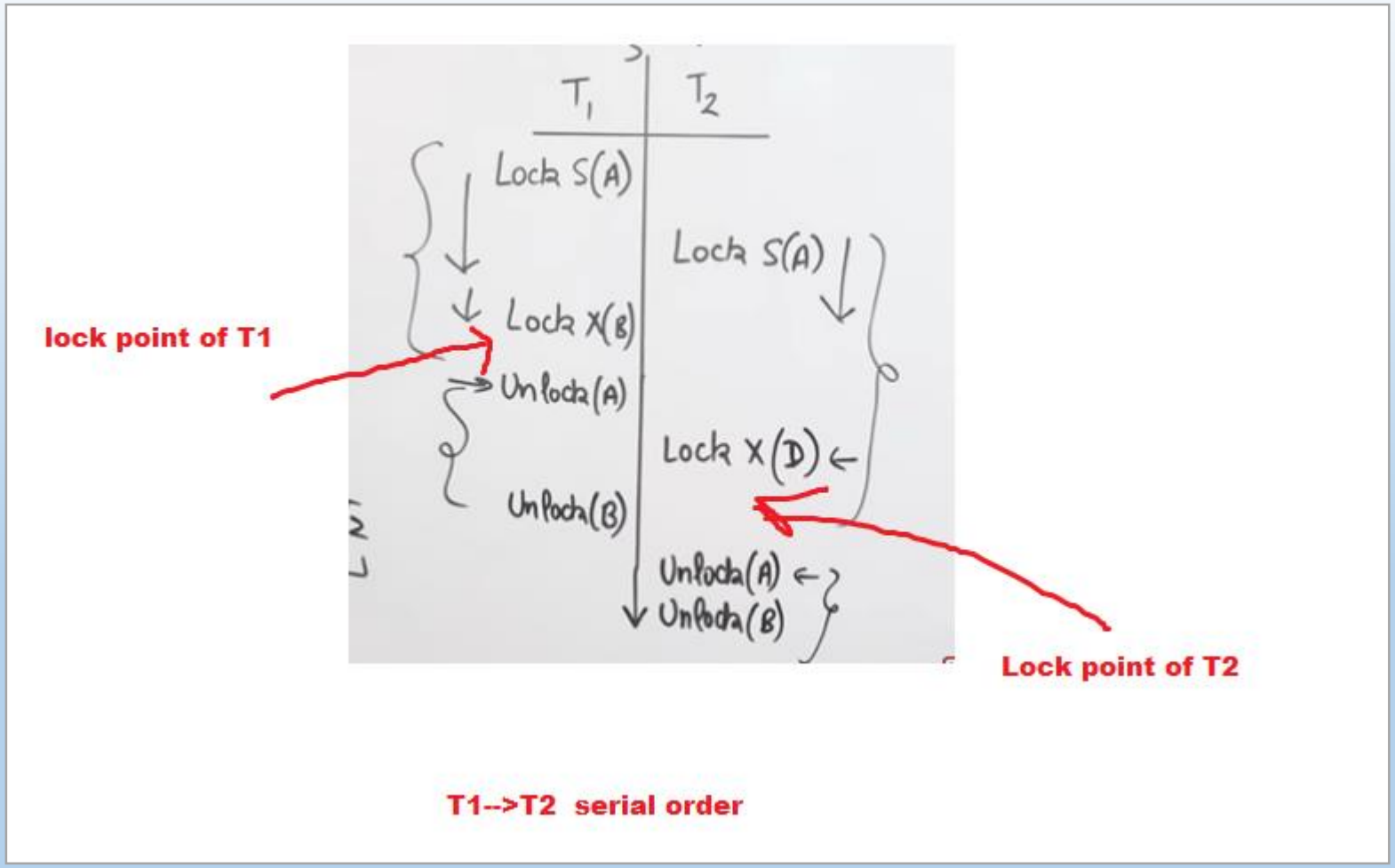
➢ Hence, X must remain locked by T until all items that the transaction needs to read or write have been locked; only then can X be released by T

➢ Meanwhile, another transaction seeking to access X may be forced to wait, even though T is done with X;

➢ Conversely, if Y is locked earlier than it is needed, another transaction seeking to access Y is forced to wait even though T is not using Y yet.



Lock point: A point where a transaction acquire last lock or where it first starts un locking



Although the two-phase locking protocol guarantees serializabiliry (that is, every schedule that is permitted is serializable), it does not permit all possible serializable schedules (that is, some serializable schedules will be prohibited by the protocol)

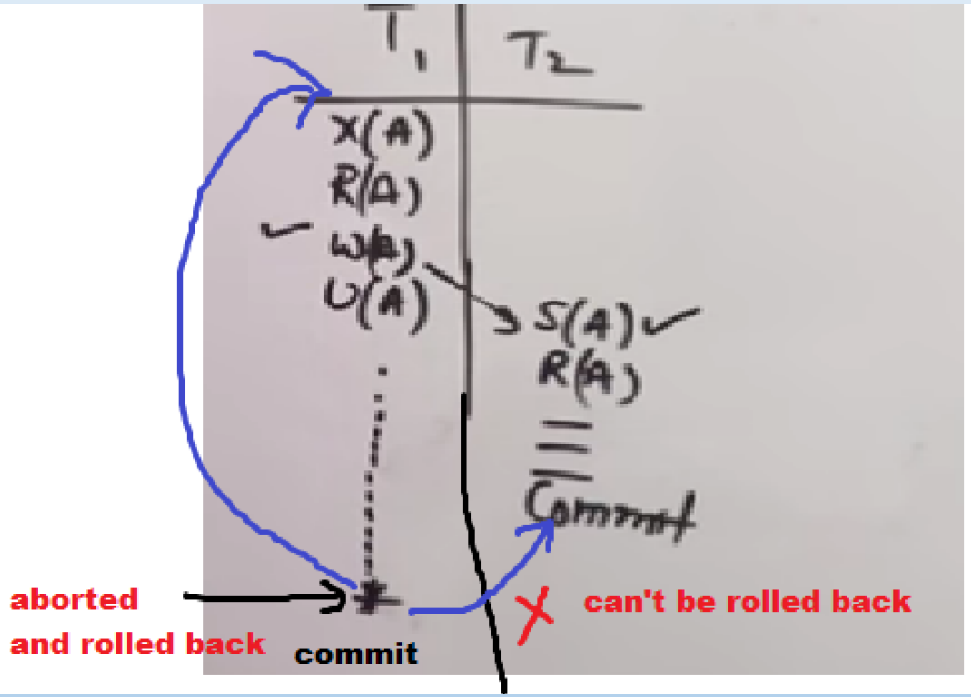
Drawbacks of 2PL

2PL (2 phase locking)

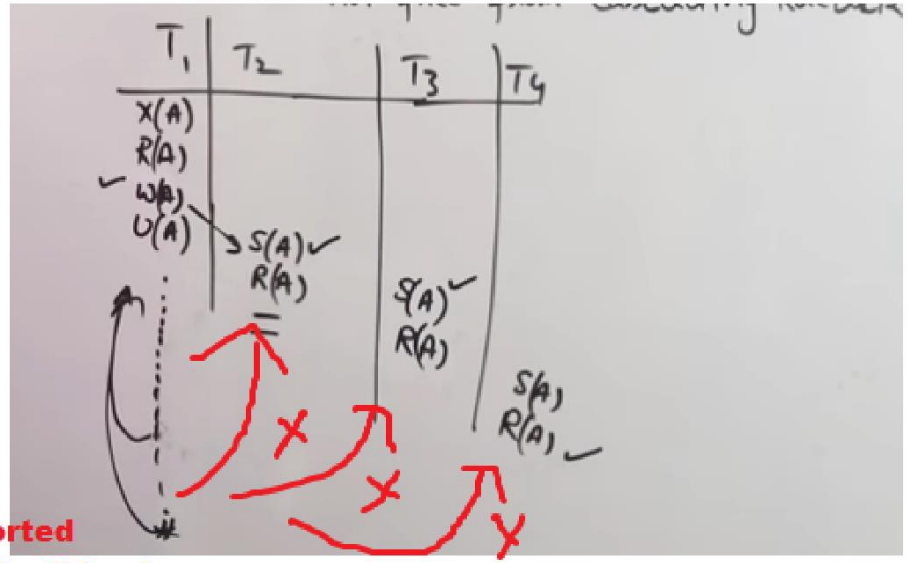
Advantages: Always ensures Serializability

Drawbacks: May not free from irrecoverability
Not free from deadlocks
Not free from starvation
Not free from Cascading Rollback

Irrecoverable Schedule

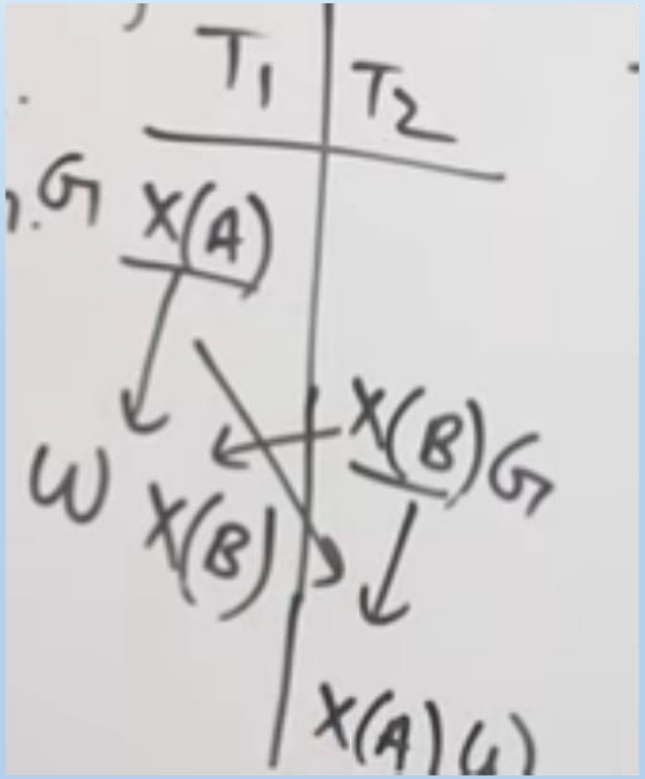


Cascading Rollback



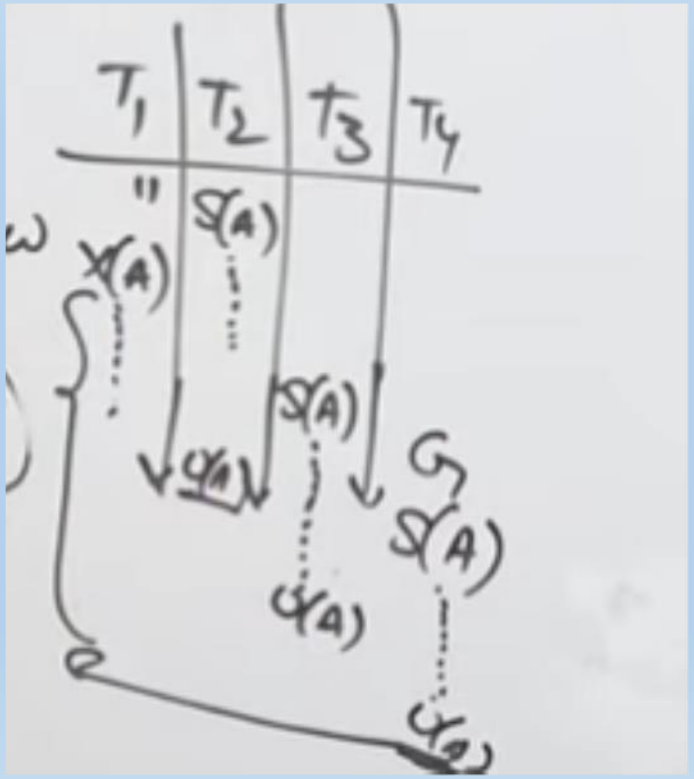
Deadlock exists

Infinite waiting of T1 and T2 on locks



Starvation :

1. T2 locks in S mode.
2. T1 wants lock in X mode and waits
3. T3 locks in S mode(granted)
4. T2 unlocks and T3 uses but T1 waits
5. T4 locks in S mode(granted)
6. T3 unlocks but T1 waits as T4 locked it
7. .....
8. **T1 waits indefinitely(Starvation)**



Basic, Conservative, Strict, and Rigorous Two-Phase Locking

Basic 2PL The technique just described is known as basic 2pL

Conservative 2PL A transaction lock all the items it accesses before the transaction begins execution, by predeclaring its readset and write-set.

read-set of a transaction is the set of all items that the transaction reads.

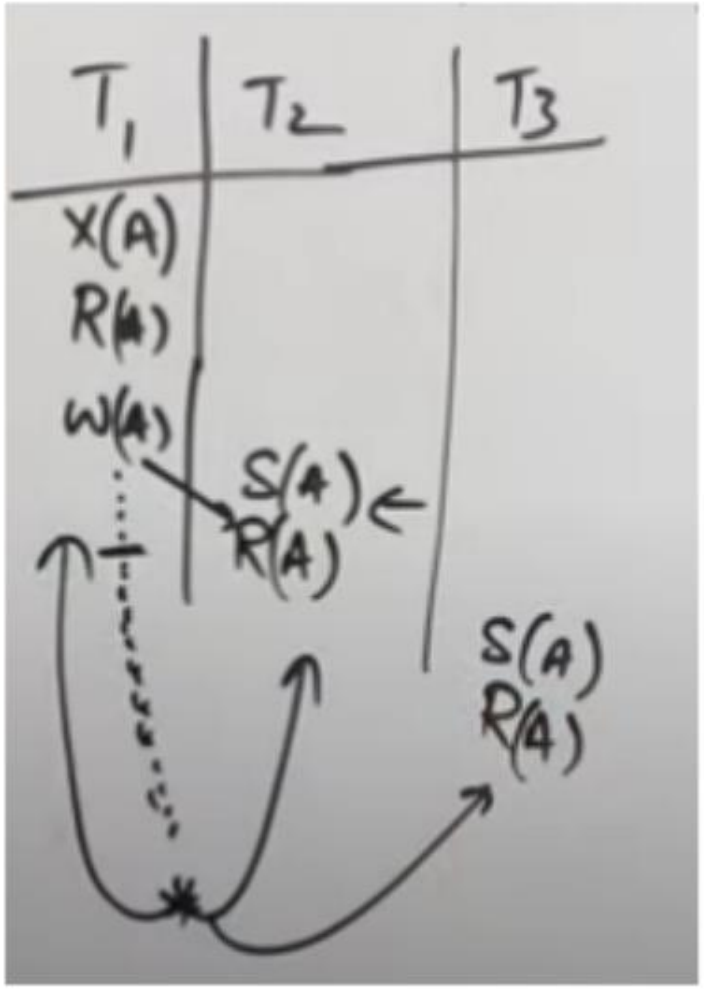
write-set is the set of all items that it writes.

- If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking.
- Conservative 2PL is a deadlock-free protocol

Strict 2PL

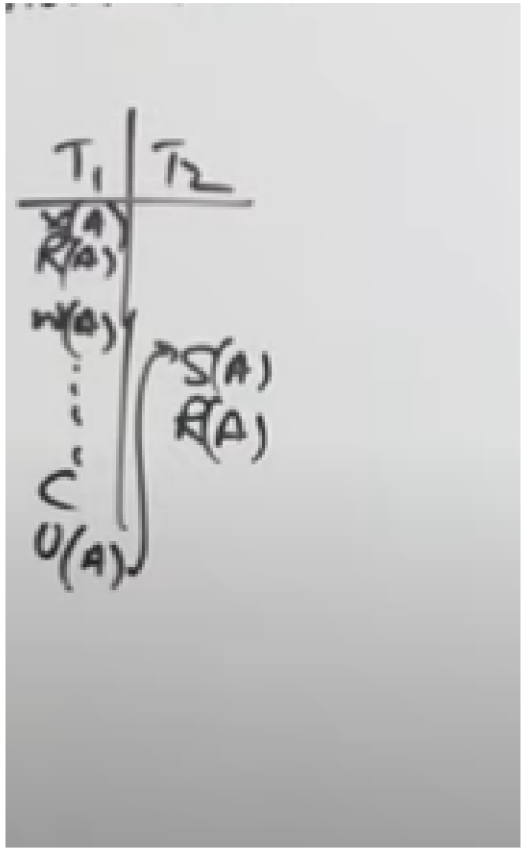
guarantees **strict schedules (**in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted).

- A transaction T does not release any of its exclusive (write) locks until after it commits or aborts.
- Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability.
- Strict 2PL is not deadlock-free.
- It is produces strict, Recoverable, cascade less schedule



Irrecoverable and cascading rollback

Basic 2PL



Recoverable and cascadeless as T2 reads after commit of T1

Strict 2PL

Rigorous 2PL A more restrictive variation ,guarantees **strict schedules**.

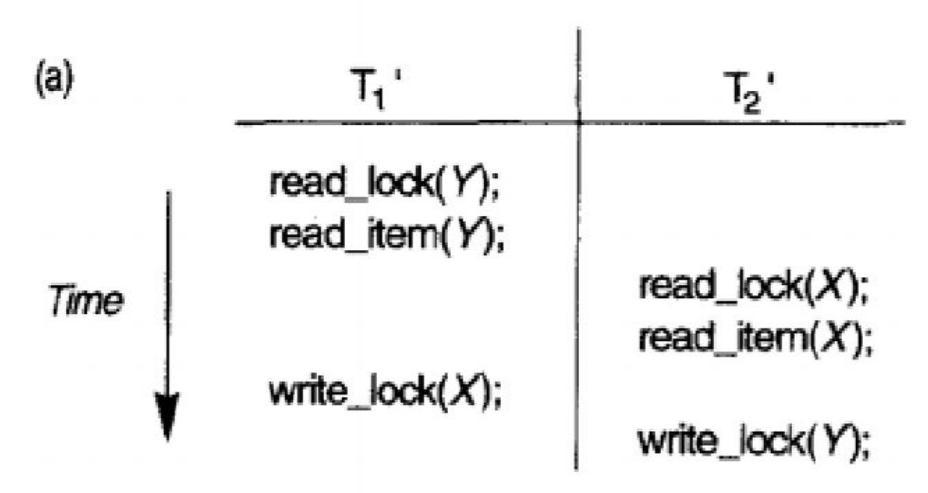
A transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2pL.

Difference between conservative and rigorous 2PL;

- **Conservative 2PL** must lock all its items before it starts so once the transaction starts it is in its shrinking phase,
- whereas the **Rigorous 2PL** does not unlock any of its items until after it terminates (by committing or aborting) so the transaction is in its expanding phase until it ends.

Deadlock and Starvation

- Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T' in the set.
- Hence, each transaction in the set is on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.



- T I' is on the waiting queue for X, which is locked by T2" while T2' is on the waiting queue for Y, which is locked by TI '
- Meanwhile, neither T 1' nor T2' nor any other transaction can access items X and Y.

Deadlock Prevention

- A technique in which the system is designed in such a way that deadlock never occurs.
- If transactions are long and each transaction uses many items, it may be advantageous to use a deadlock prevention scheme.

Protocol-1

- One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock all the items it needs before it starts (which is generally not a practical assumption)-
- If any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then

Protocol-2

- It limits concurrency, involves ordering all the items in the database and making sure that a transaction that needs several items will lock them according to that order.
- This requires that the programmer (or the system) be aware of the chosen order of the items, (not practical)

Timestamp based protocol1

Deals with : What to do with a transaction involved in a possible deadlock situation?

1. Should it be blocked and made to wait?
2. or should it be aborted,
3. or should the transaction preempt and abort another transaction?

- These techniques use the concept of transaction timestamp TS(T),

A unique identifier assigned to each transaction when it starts.

- if transaction T1 starts before transaction T2, then TS(T1) < TS(T2).
- older transaction has the smaller timestamp value.

1. Wait-die:

Suppose that transaction T1 tries to lock an item X but is not able to because X is locked by some other transaction T2 with a conflicting lock.

Wait-die:

If TS(T1) < TS (T2) , then (T1 older than T2)

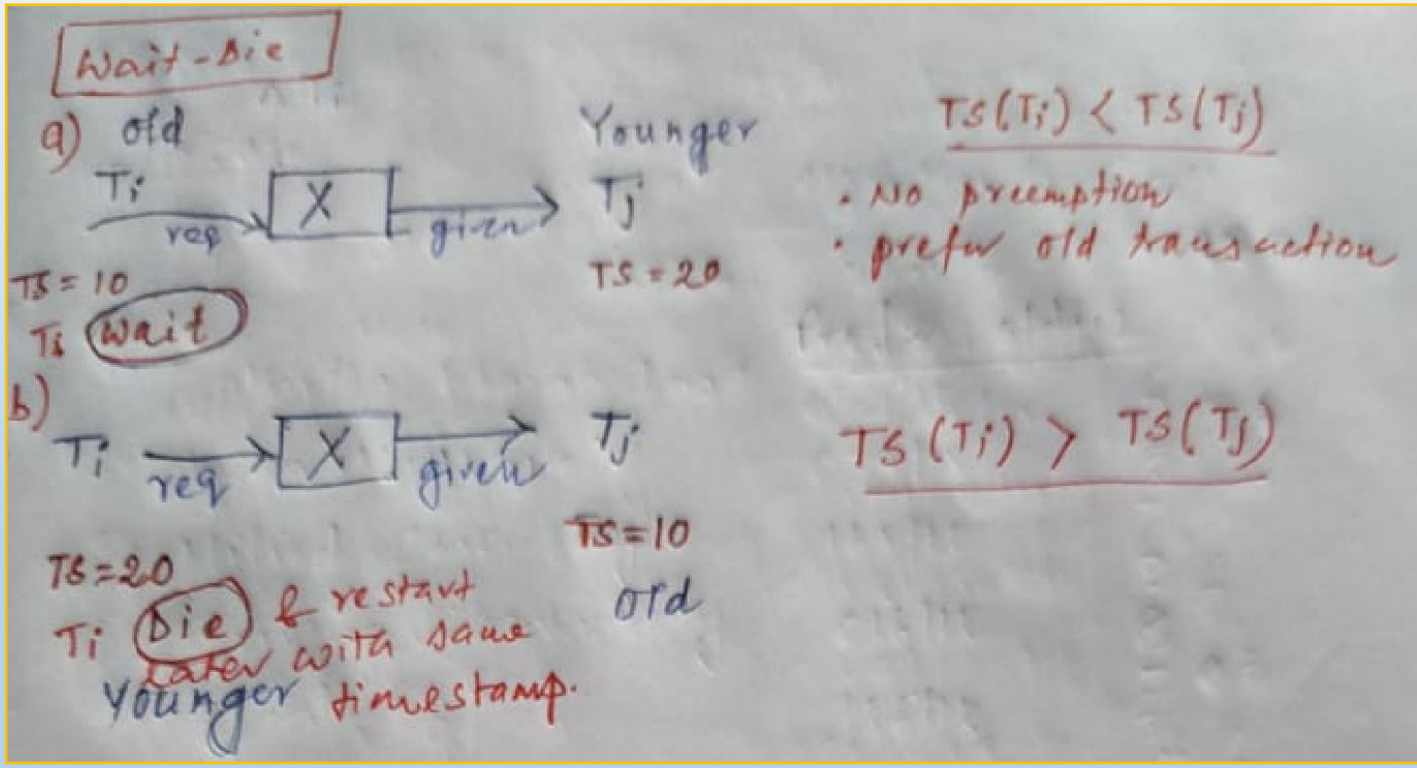
T1 is allowed to wait;

else (T1 younger than T2)

abort T1 (T1 dies) and

restart it later with the same timestamp.

End



1. An older transaction is allowed to wait on a younger transaction.
2. A younger transaction requesting an item held by an older transaction is aborted and restarted.

2. Wound-wait:

Wound-wait

If TS(T1) < TS(T2) , then (T1older than T2)

abort T2 (T1 wounds T2)

and restart it later with the same timestamp;

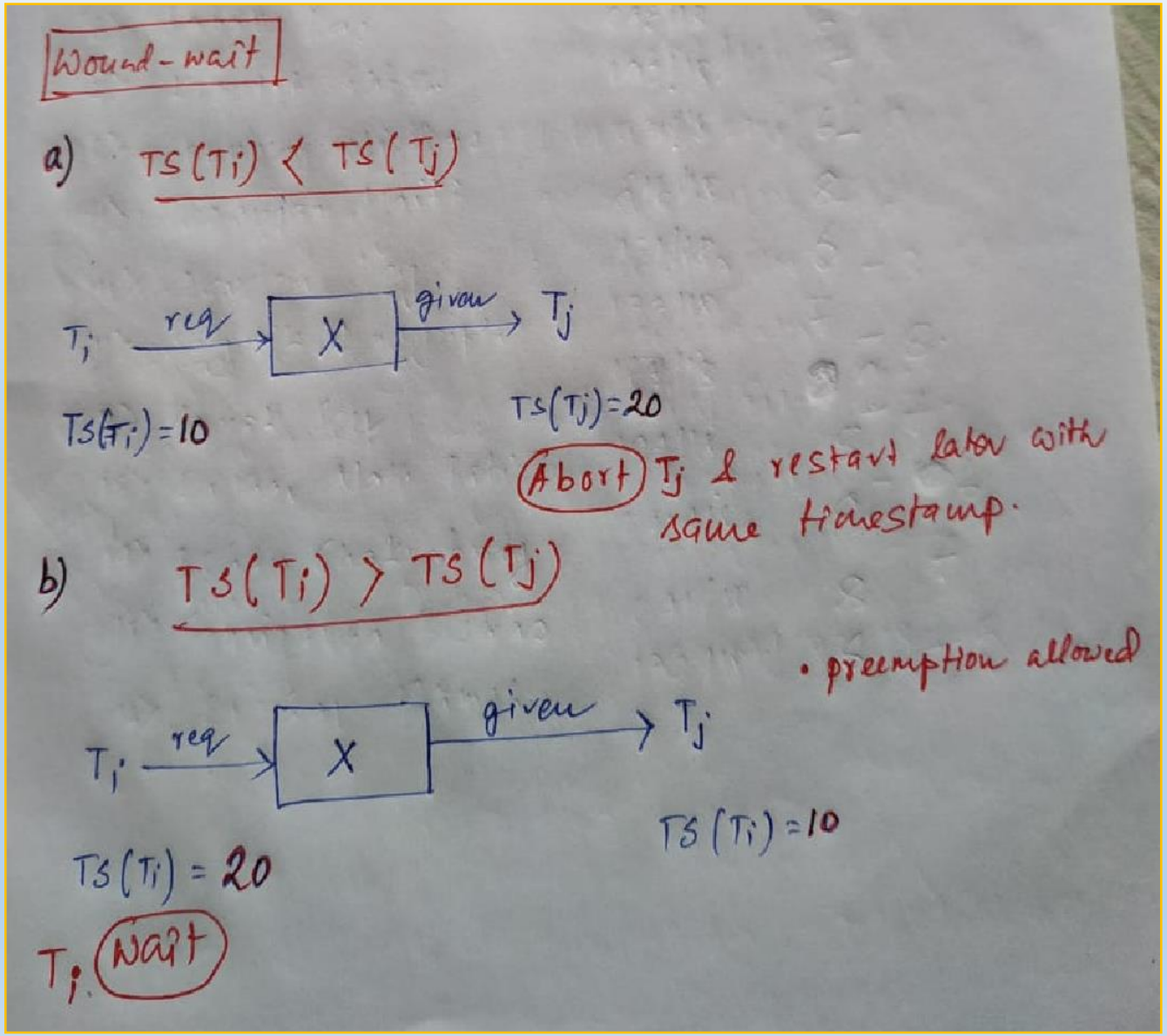
else (T1 younger than T2)

T1 is allowed to wait.

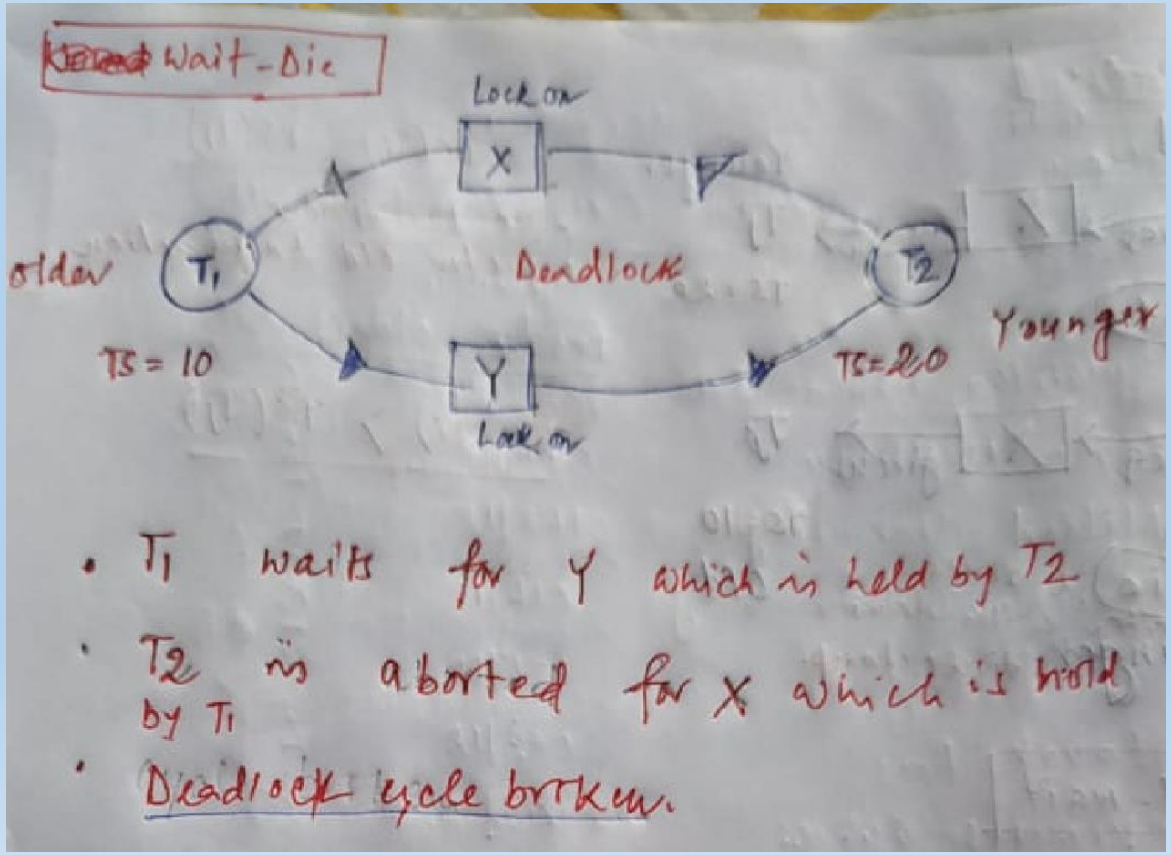
End

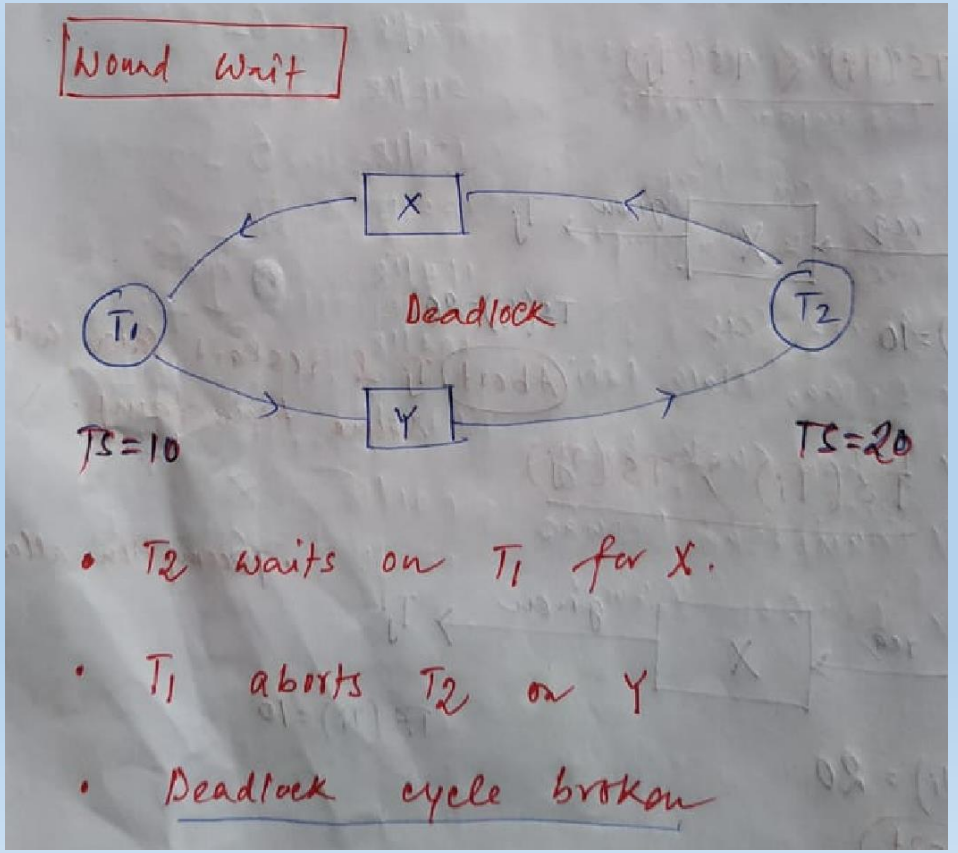
- **The wound-wait approach does the opposite:**

1. A younger transaction is allowed to wait on an older one,
2. An older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it.



- Both schemes end up aborting the younger of the two transactions that may be involved in a deadlock.
- two techniques are deadlock-free.
- In wait-die, transactions only wait on younger transactions so no cycle is created.
- In wound-wait, transactions only wait on older transactions so no cycle is created.
- However, both techniques may cause some transactions to be aborted and restarted needlessly, even though those transactions may never actually cause deadlock





Timestamp based protocol 2

1) No waiting algorithm:

If a transaction is unable to obtain a lock, then

- It is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not.

- It cause transactions to abort and restart needlessly, the cautious waiting algorithm was proposed to try to reduce the number of needless aborts/restarts

2) Cautious waiting algorithm

- Was proposed to try to reduce the number of needless aborts/restarts
- Suppose that transaction Ti tries to lock an item X which is locked by some other transaction Tj with a conflicting lock
- If Tj is not blocked (not waiting for some other locked item), then

  Ti is blocked and allowed to wait;

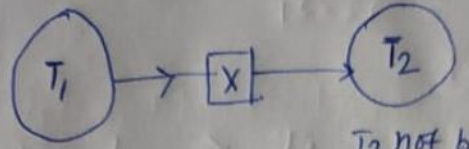
  otherwise

  abort Ti

- Cautious waiting is deadlock-free.
- B(T)=Blocked time of T, i.e. time at which T was blocked
- If the two transactions T1and T2 above both become blocked, and T1 is waiting on T2 , then b(T1) < b(T2), since T1 can only wait on T2 at a time when T2 is not blocked.

- Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

Cautious Waiting



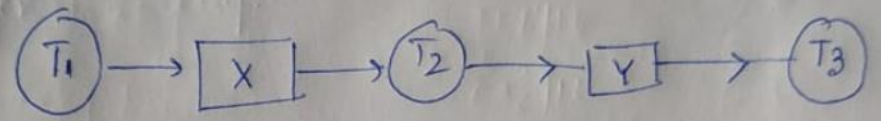
$b(T)$ = Block time

T_2 not blocked.

block T_1

$B(T_1) = 10$

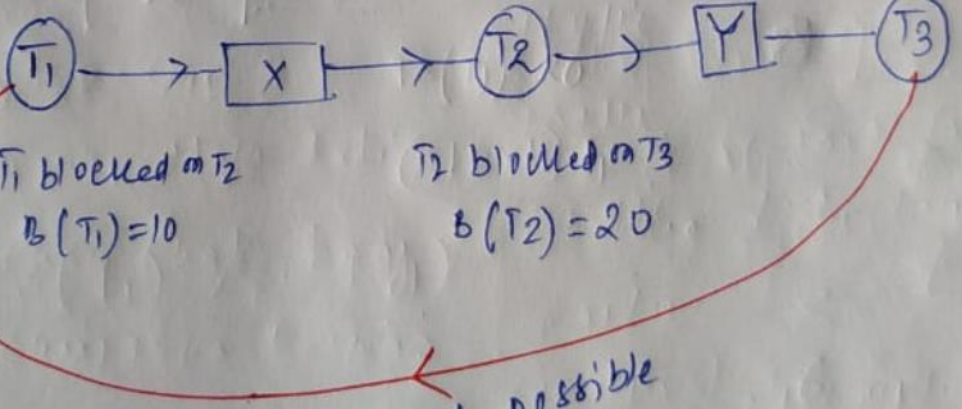
T_1 again after 10 msec.



T_1 blocked. T_2 blocked T_3 not blocked

$B(T_1) = 10$ $B(T_2) = 20$

So $B(T_1) < B(T_2)$



T_1 blocked on T_2 $B(T_1) = 10$

T_2 blocked on T_3 $B(T_2) = 20$

- not possible as T_1 is blocke
- so abort T3
- Hence deadlock cycle broken

Deadlock Detection

- **Deadlock detection :** The system checks if a state of deadlock actually exists.
- Useful ------- if the transactions are short and each transaction locks only a few items

Method-1: wait-for graph.

- System construct and maintain a **wait-for graph.**
- One node is created in the wait-for graph for each transaction that is currently executing.
- Whenever a transaction Ti is waiting to lock an item X that is currently locked by a transaction Tj , a directed edge (Ti Tj) is created in the wait-for graph.
- When Tj releases the locks on the items that Ti was waiting for, the directed edge is dropped from the wait-for graph
- Deadlock occurs if wait-for graph contains a cycle

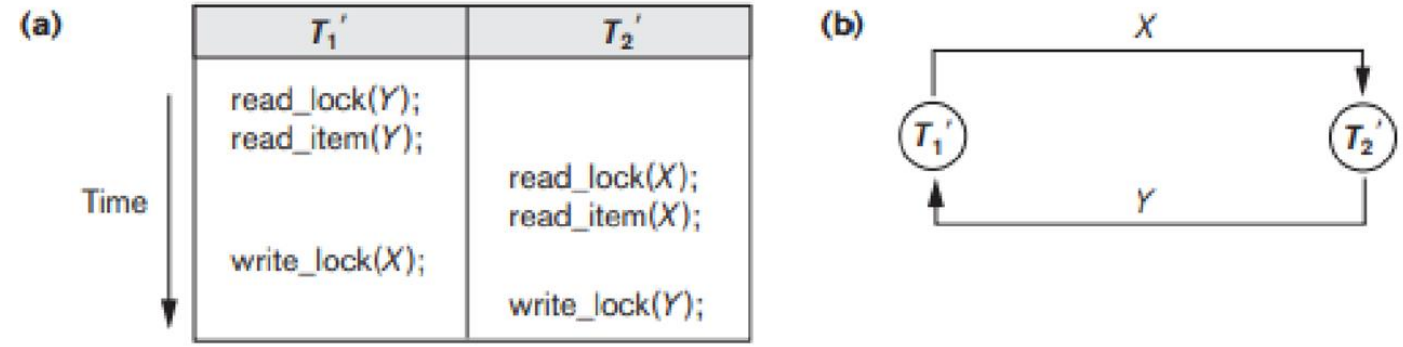


Figure 22.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

- **Problem : Suitable time for cycle detection ?**
- Solution-1 : Check for a cycle every time an edge is added to the wait-for graph, Problem: excessive computation overhead.
- Solution-2: Number of currently executing transactions .
- Solution-3 Lock waiting time of several transactions have been waiting to lock items may be used.

Victim selection:

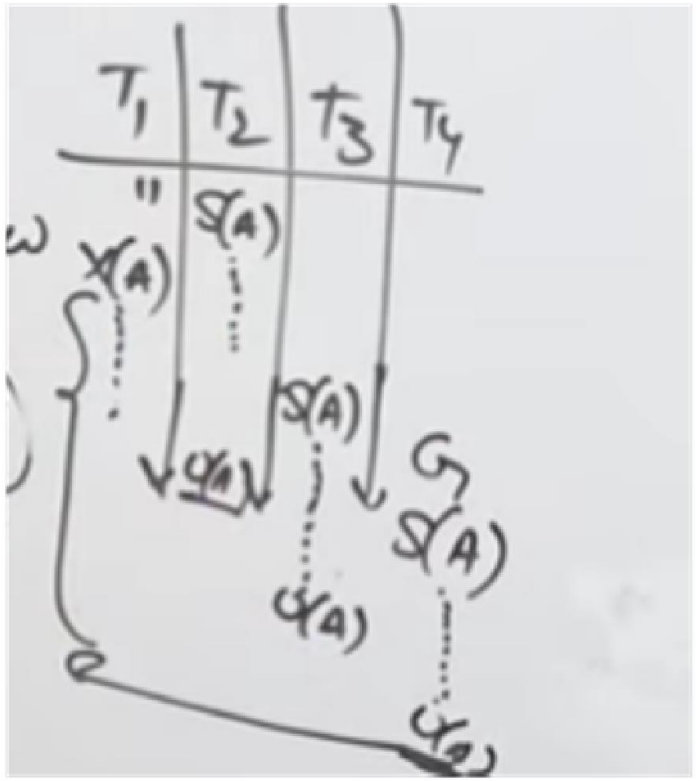
- If deadlock is detected some of the transactions causing the deadlock must be aborted.
- Choosing which transactions to abort is known as victim selection
- Avoid selecting transactions that have been running for a long time(Older) and that have performed many updates.
- Rather choose younger transactions

Timeouts

- If a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists or not.
- This method is practical because of its low overhead and simplicity.

Starvation.

- Another problem that may occur when we use locking is starvation, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally



Starvation :

1. T2 locks in S mode.
2. T1 wants lock in X mode and waits
3. T3 locks in S mode(granted)
4. T2 unlocks and T3 uses but T1 waits
5. T4 locks in S mode(granted)
6. T3 unlocks but T1 waits as T4 locked it
7. .....
8. **T1 waits indefinitely(Starvation)**

- One solution for starvation is to have a fair waiting scheme, such as using a first-come-first-served queue;
- Transactions are enabled to lock an item in the order in which they originally requested the lock.
- Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.
- Starvation can also occur if victim selection is wrong: if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution.
- The wait-die and wound-wait schemes avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

Time Stamp ordering protocol

Timestamps TS(T): A unique identifier of a transaction T, created by the DBMS to identify a start time of T.

- So a timestamp can be thought of as the transaction start time.

So if TS(T1)=10 and TS(T2)=20 then T2 is younger transaction and T1 is older transaction.

Timestamps Generation:

Method-1 :Use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, ... in this scheme.

Method-2 : Use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

Timestamp Ordering Algorithm

- Principle: Order the transactions based on their timestamps.
- A schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamp values.
- Schedule is equivalent to the particular serial order corresponding to the order of the transaction timestamps called timestamp ordering (TO).
- For each item accessed by conflicting operations in the schedule, the order in which the item is accessed does not violate the timestamp order.

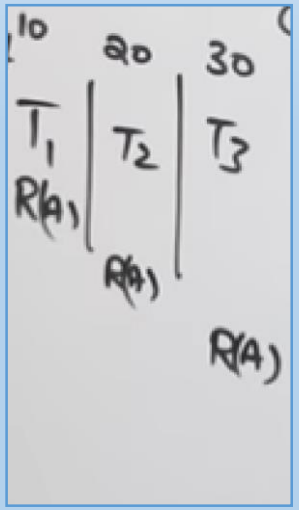
Each database item X has two timestamp (TS) values:

1. **Read_TS(X):** Read timestamp of X.

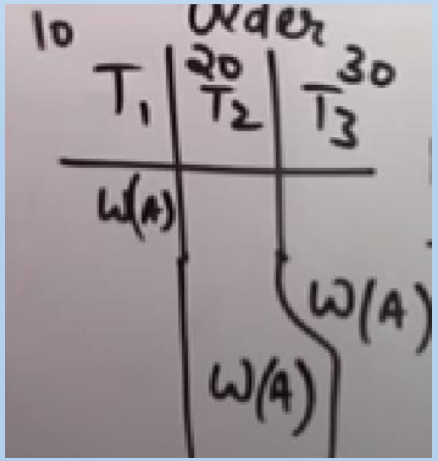
Timestamp of the transaction that have performed last read on X successfully.

2. **Write_TS(X):** Write timestamp of X.

Timestamp of the transaction that have performed last write on X successfully.



Read_TS(A)=30



Write_TS(A)=20

Basic Timestamp Ordering (TO).

- Whenever some transaction T tries to issue a read_item(X) or a write_item(X) operation, the basic TO algorithm compares the timestamp of T with read_TS(X) and write_TS(X) to ensure that the timestamp order of transaction execution is not violated.
- Whenever the basic TO algorithm detects two conflicting operations that occur in the incorrect order(not in , it rejects the later of the two operations by aborting the transaction that issued it .
- If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a new timestamp.
- If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back. Similarly, any transaction T2 that may have used a value written by T1 must also be rolled back, and so on. This effect is **known as cascading rollback** and is one of the problems associated with basic TO. Schedules produced are not guaranteed to be recoverable.
- Additional protocol must be enforced to ensure that the schedules are recoverable, cascadeless, or strict.

1. Whenever a transaction T issues a write_item(X) operation

a) **If read_TS(X) > TS(T) or if write_TS(X) > TS(T)** then

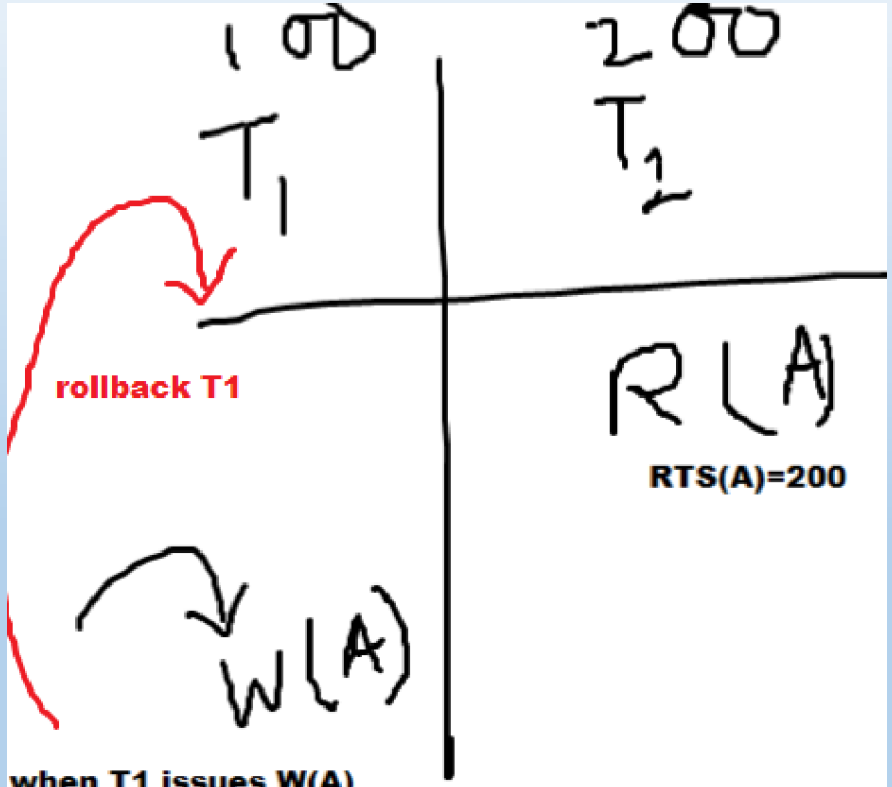
- **Abort and roll back T and reject the operation.**

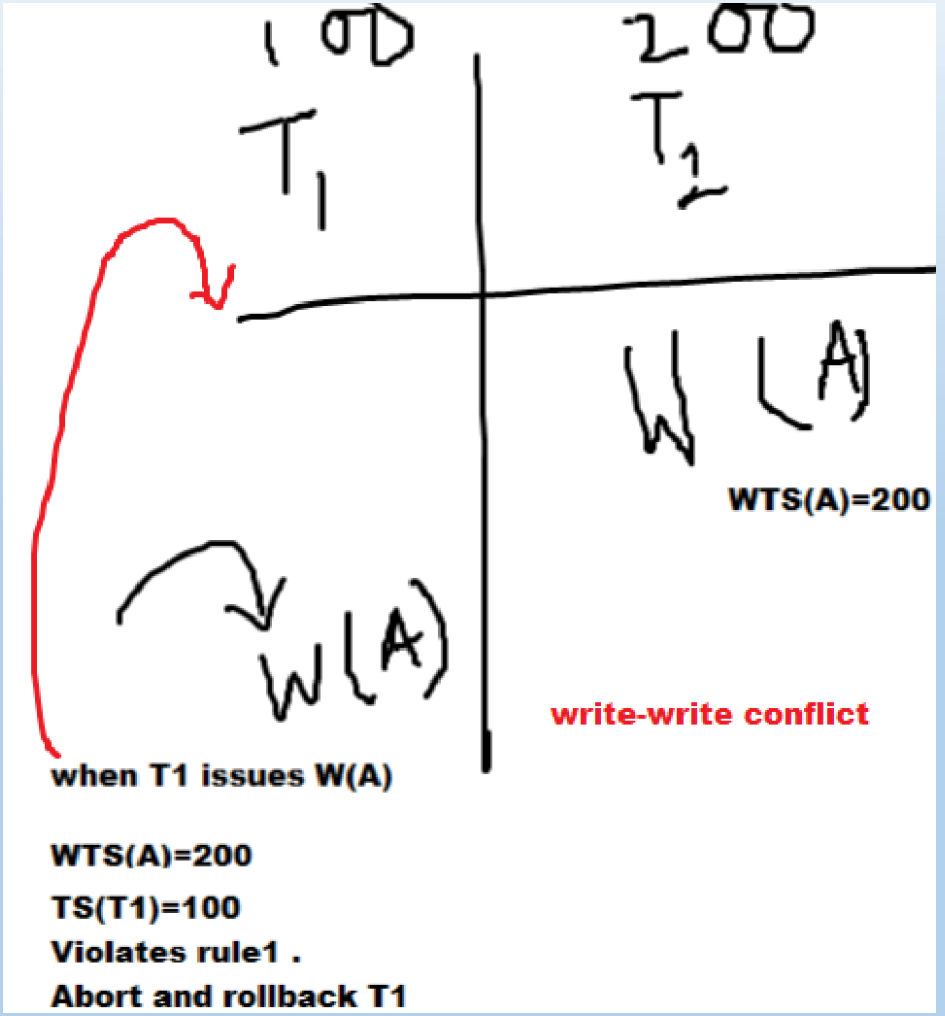
This should be done because some younger transaction with a timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.

b)else

Execute the write_item(X) operation of T

and set write_TS(X) =TS(T).


100
200
T1
T2
rollback T1
R(A)
RTS(A)=200
W(A)
when T1 issues W(A)
RTS(A)=200
TS(T1)=100
Violates rule1 .
Abort and rollback T1
Read-Write conflict


100
200
T1
T2
W(A)
WTS(A)=200
W(A)
write-write conflict
when T1 issues W(A)
WTS(A)=200
TS(T1)=100
Violates rule1 .
Abort and rollback T1

Whenever a transaction T issues a read_item(X) operation

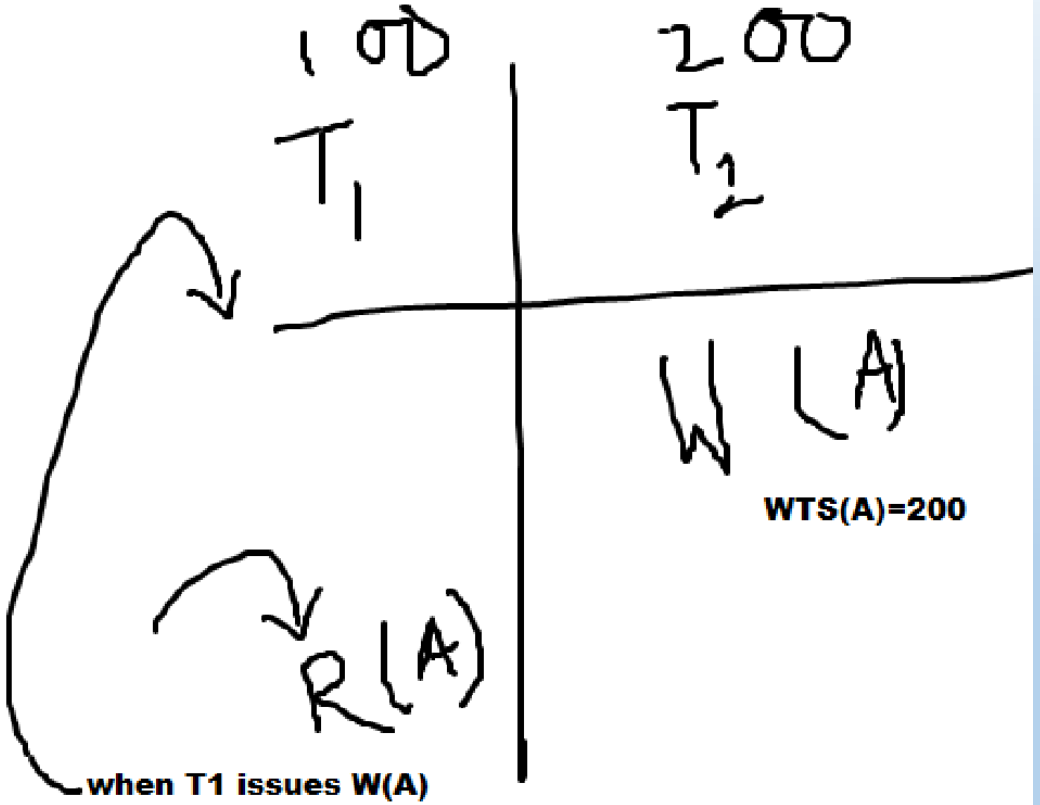
If write_TS(X) > TS(T), then

- Abort and roll back T and reject the operation.

This should be done because some younger transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of item X before T had a chance to read X.

Else If write_TS(X) ≤ TS(T), then

- Execute the read_item(X) operation of T and set read_TS(X) =Max(TS(T), read_TS(X))



WTS(A)=200

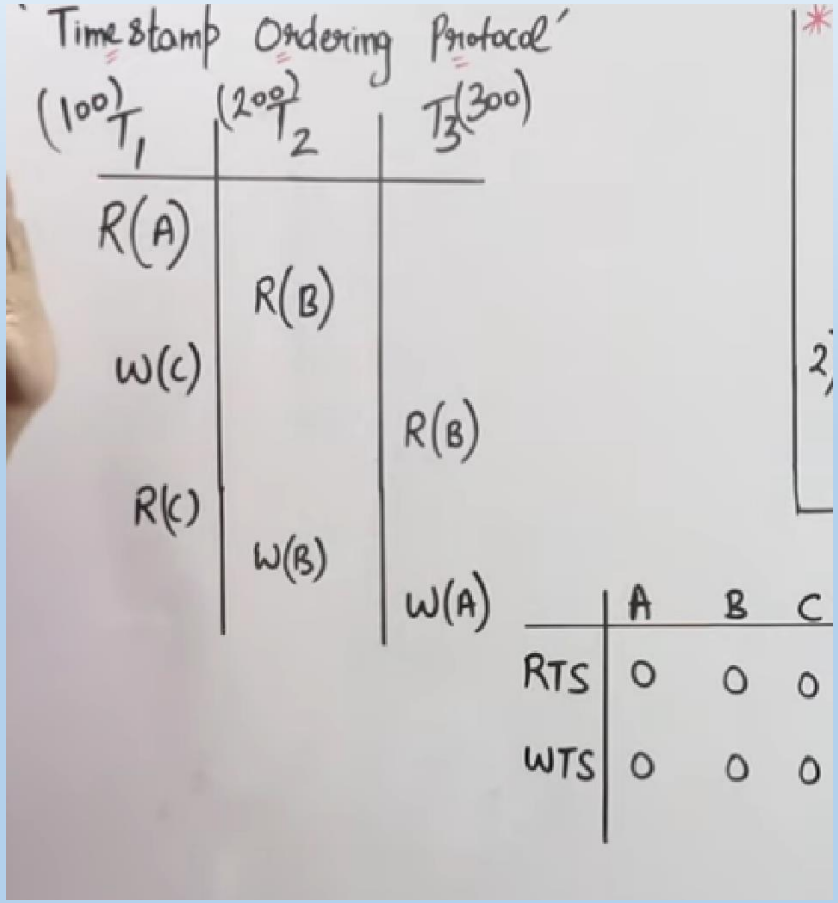
TS(T1)=100

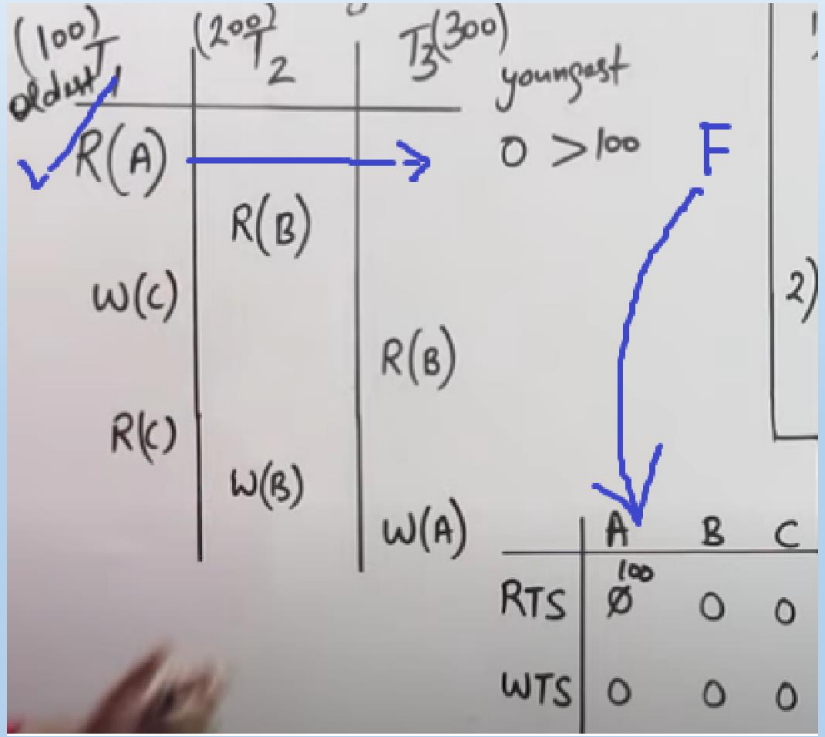
Violates rule1 .

Abort and rollback T1

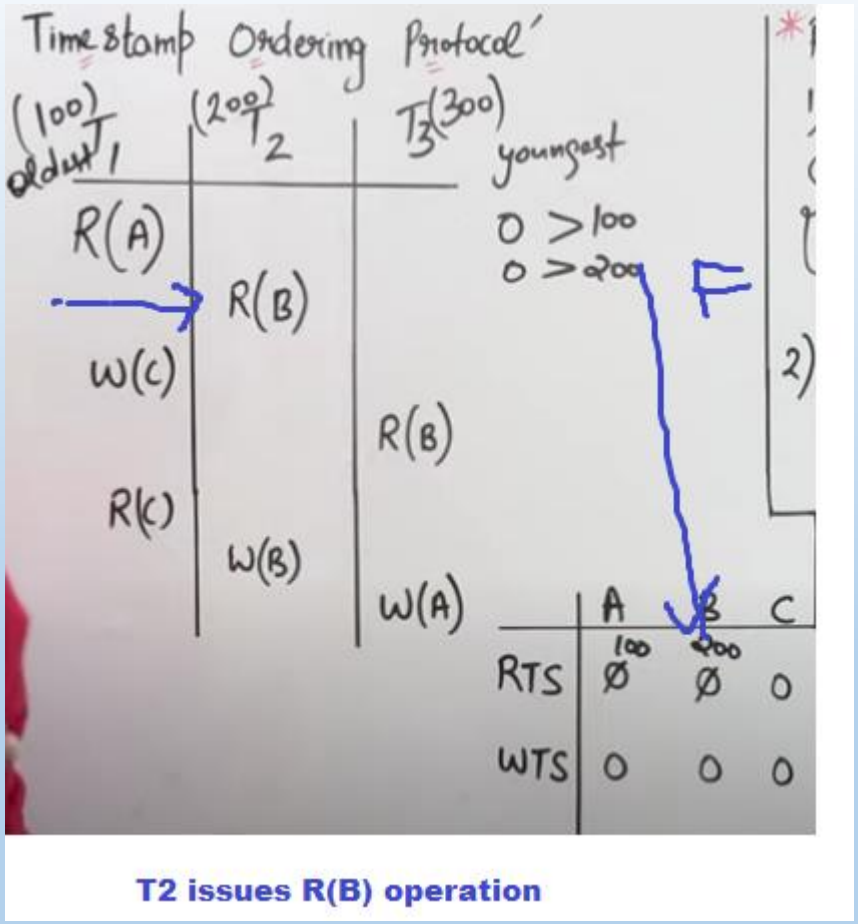
- The schedules produced by basic TO are hence guaranteed to be conflict serializable, like the 2PL protocol.
- It is deadlock free but ,Cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

Example on Basic TS ordering

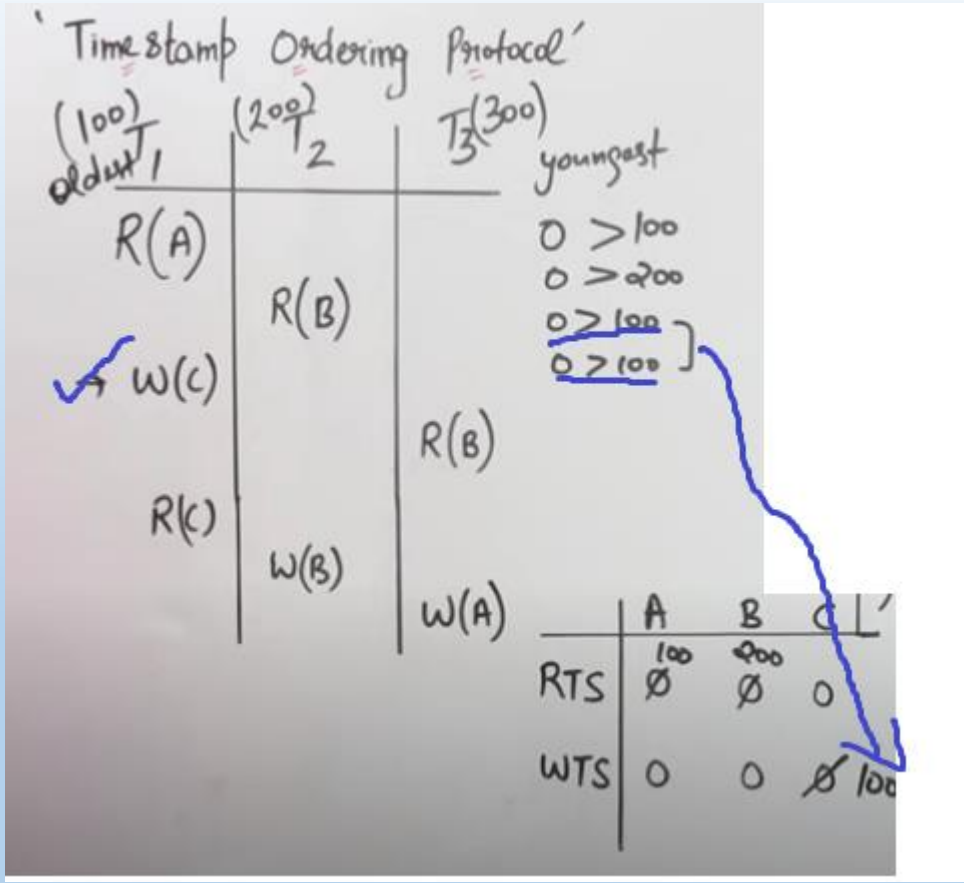




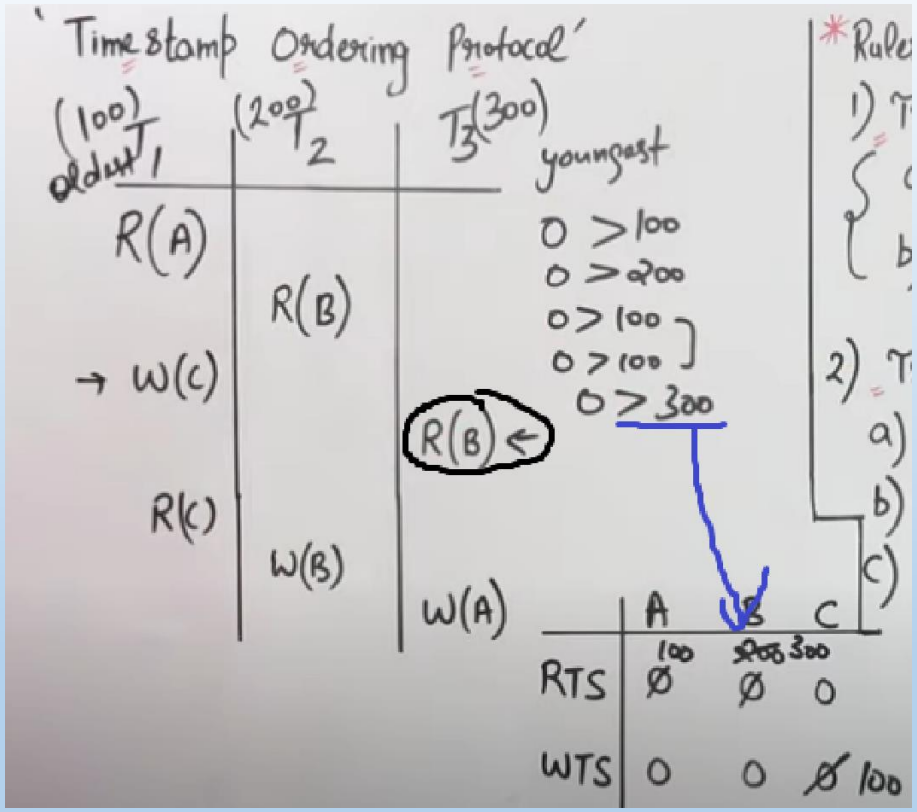
T1 issues R(A) operation

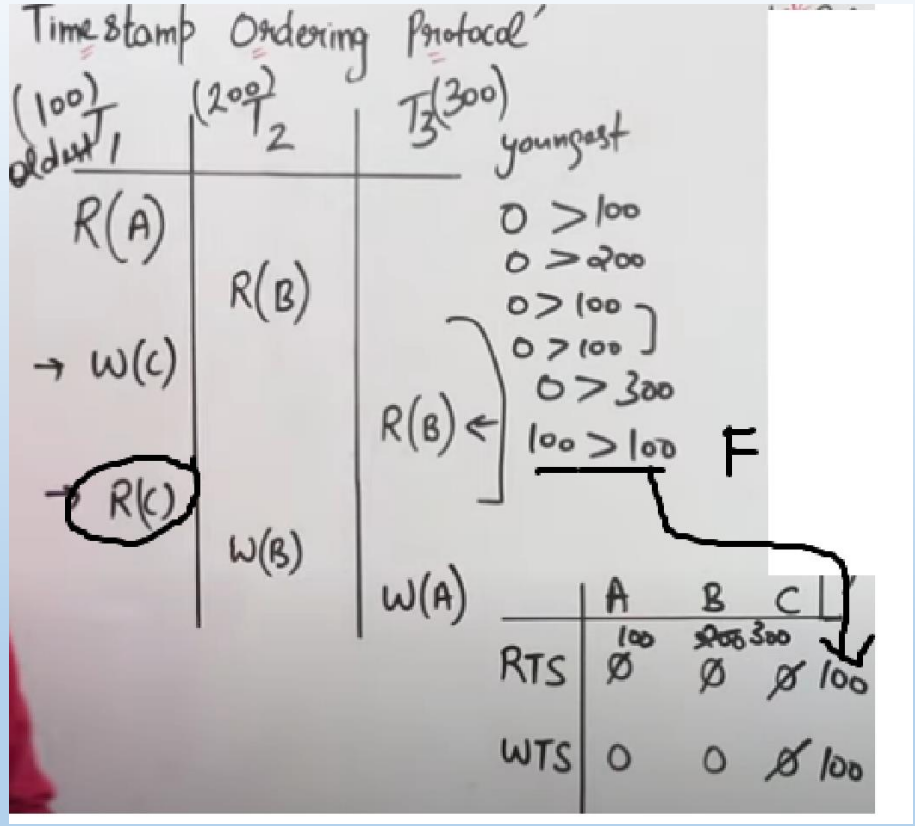


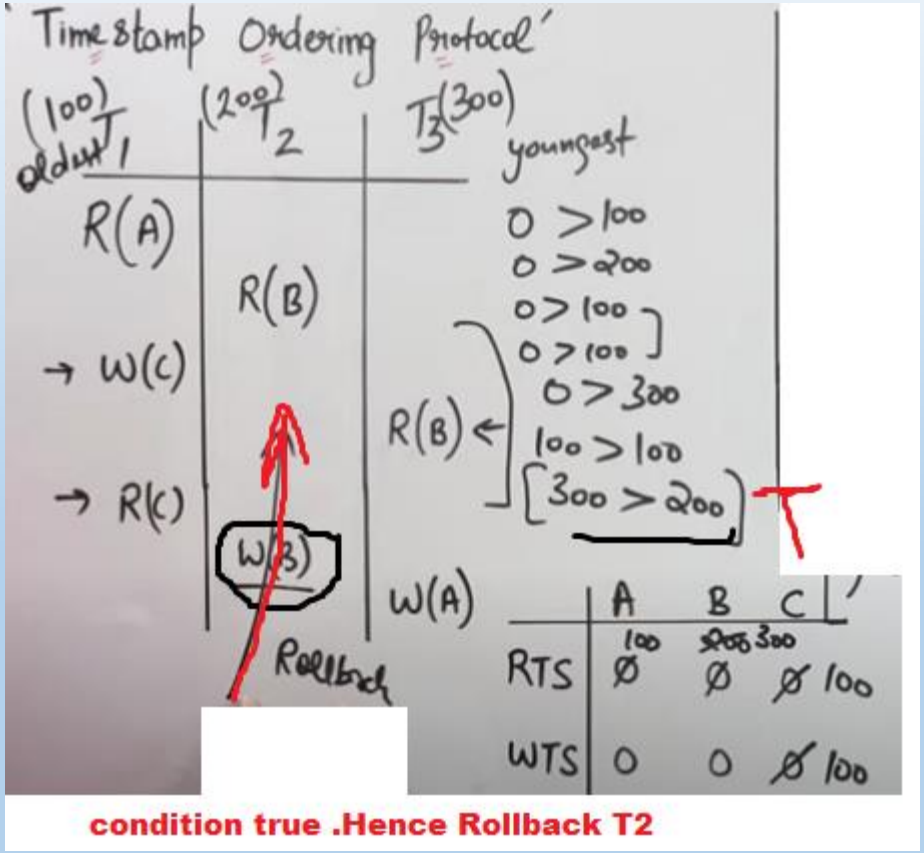
T2 issues R(B) operation

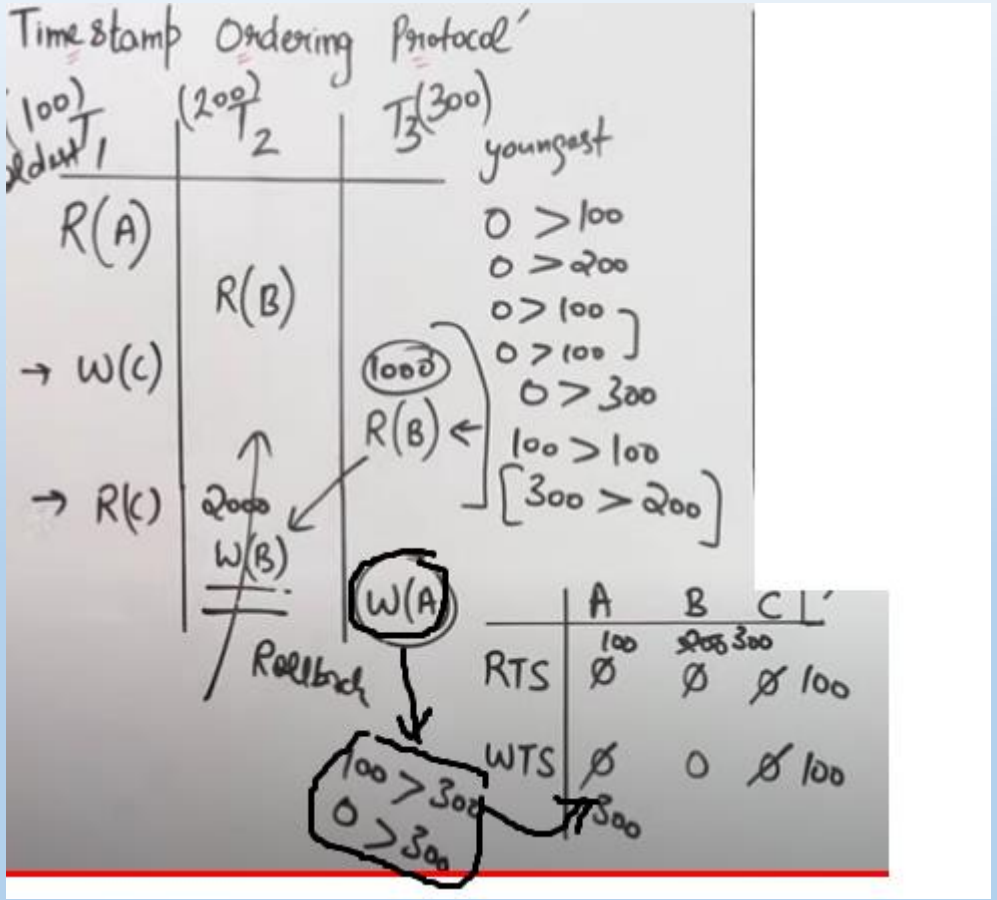


T1 issues W(C) operation


Timestamp Ordering Protocol
(100) T1
(200) T2
T3 (300)
oldest
youngest
R(A)
R(B)
→ W(C)
R(C)
W(B)
W(A)
R(B) ←
0 > 100
0 > 200
0 > 100
0 > 100
0 > 300
A B C
100 200 300
RTS Ø Ø 0
WTS 0 0 Ø 100


Timestamp Ordering Protocol
(100) T1
(200) T2
T3 (300)
oldest
youngest
R(A)
R(B)
→ W(C)
R(C)
W(B)
W(A)
R(B) ←
0 > 100
0 > 200
0 > 100
0 > 100
0 > 300
100 > 100
F
A B C
100 200 300
RTS Ø Ø Ø 100
WTS 0 0 Ø 100


Timestamp Ordering Protocol
(100) T1
(200) T2
T3 (300)
youngest
R(A)
R(B)
W(C)
R(C)
R(B)
W(B)
W(A)
Rollback
A B C
RTS
WTS
condition true .Hence Rollback T2

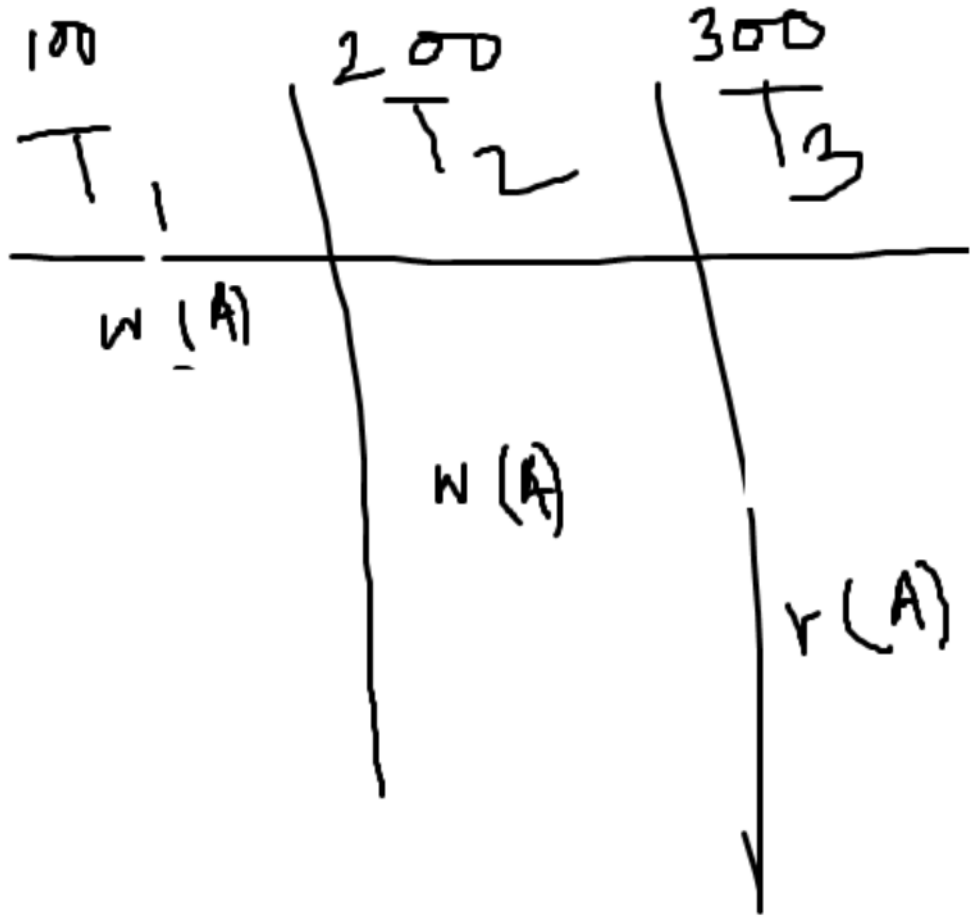

Timestamp Ordering Protocol
(100) T1
(200) T2
T3 (300)
youngest
R(A)
R(B)
W(C)
R(C)
R(B)
W(B)
W(A)
Rollback
A B C
RTS
WTS

Strict Timestamp Ordering (TO).

- Ensures that the schedules are both **strict (for easy recoverability) and (conflict) serializable**.
- A transaction T that issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X)

has its read or write operation delayed until the transaction T' that wrote the value of X (hence TS(T) = write_TS(X)) has committed or aborted.

- To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T 'until T' is either committed or aborted. This algorithm does not cause deadlock, since T waits for T' only if TS(T) > TS(T').



1. WTS(A)=0 < TS(T1)=100 hence operation successful and WTS(A)=100

2. WTS(A)=100 < TS(T2)=200;

T2 has to wait until T1 commits

3. WTS(A)=200 < TS(T3)=300

T3 has to wait until T2 commit

hence it is cascadeless

Thomas's Write Rule. A modification of the basic TO algorithm, known as **Thomas's write rule**, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows:

1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of X. Thus, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).