UNIT 5

Introduction to Transaction Processing Concepts and Theory

5th Edition

Elmasri / Navathe

Introduction to Transaction Processing

- **Single-User Database System**:
  - At most one user at a time can use the system.
  - Example: Mostly a Personal computer database

- **Multiiuser Database System**:
  - Many users can access the system concurrently.
  - Here multiple transactions are submitted simultaneously to the Database.
  - Example: Airline reservation, Banking sys, Stock exchange, Shopping sites etc

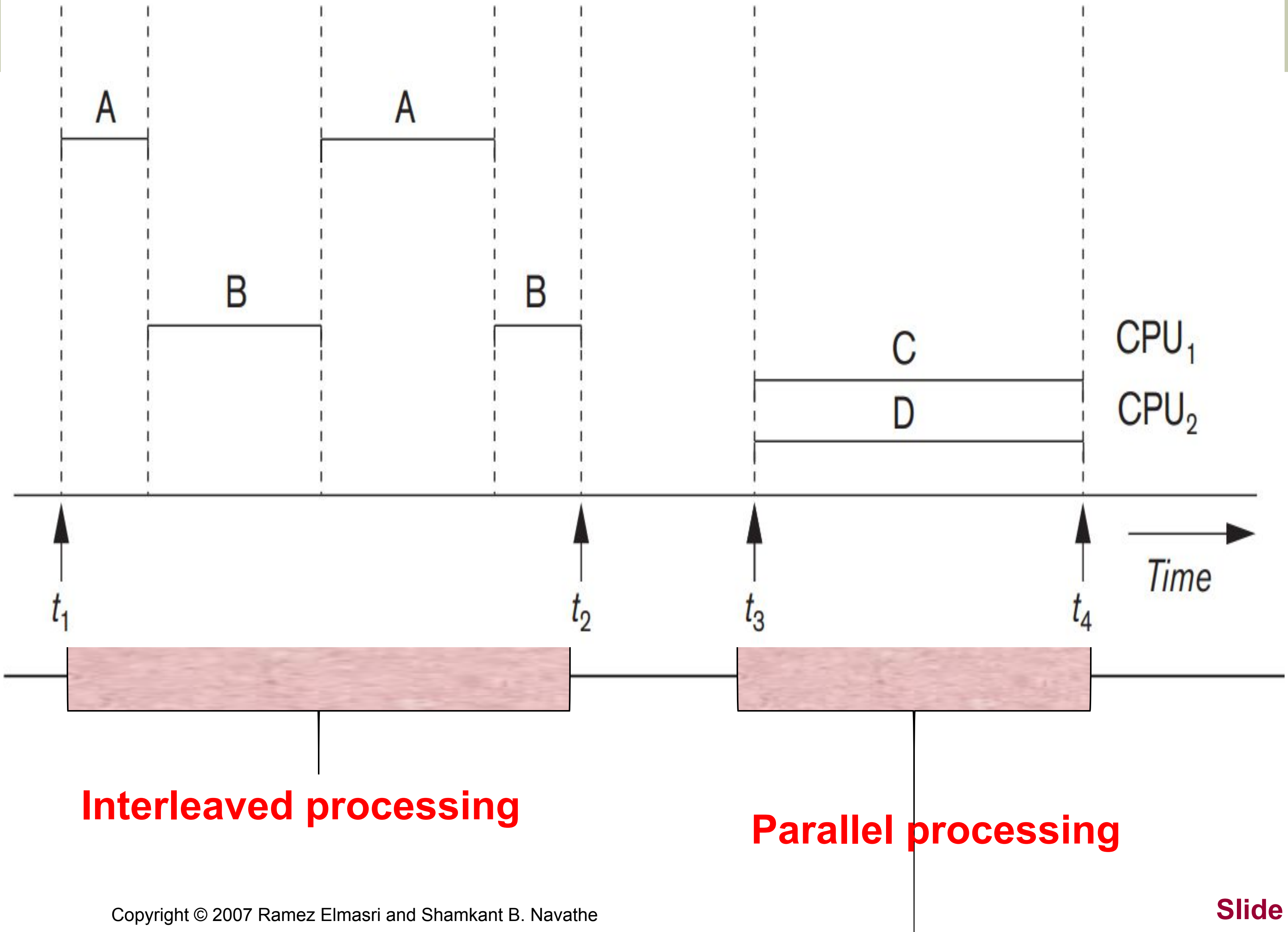
- **Concurrency -** allows the operating system of the computer to execute multiple programs—or processes—at the same time.

  1) Interleaved processing or multiprogramming:

  - Concurrent execution of processes is interleaved in a single CPU
  - i.e. execute some commands from one process, then suspend that process and execute some commands from the next process and so on.

  2) Parallel processing:

  - Processes are concurrently executed in multiple CPUs.



Interleaved processing

Parallel processing

- **A Transaction:**
  - ***A transaction is a executing program that forms a Logical unit of work in database processing.***

    It consists of a sequence of operations (like read - retrieval, write - insert or update and delete) that transforms a consistent state of database into another consistent state.

- A transaction may be
  - stand-alone □ specified in a high level language like SQL
  - embedded within a program.

- An **application program** may contain several transactions separated by the **Begin and End** transaction boundaries.

Simple Transaction Example : you are transferring money from your bank account to your friend's account, the set of operations would be like this:

- 1. Read your account balance
- 2. Deduct the amount from your balance
- 3. Write the remaining balance to your account
- 4. Read your friend's account balance
- 5. Add the amount to his account balance
- 6. Write the new updated balance to his account

SIMPLE MODEL OF A DATABASE to Understand Transactions

- **A Database** is a collection of named data items
- **Granularity** means **size of data item** - a field, a record , or a whole disk block
- Each data item has a unique name that identifies it

- Basic database access operations are **read** and **write**
  - **read_item(X)**: Reads a database item named X into a program variable X.
  - **write_item(X)**: Writes the value of program variable X into the database item named X.

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to main memory is **one block**.
- Further, a data item (read or written) will be the field of some record in the database

- **read_item(X)** command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.

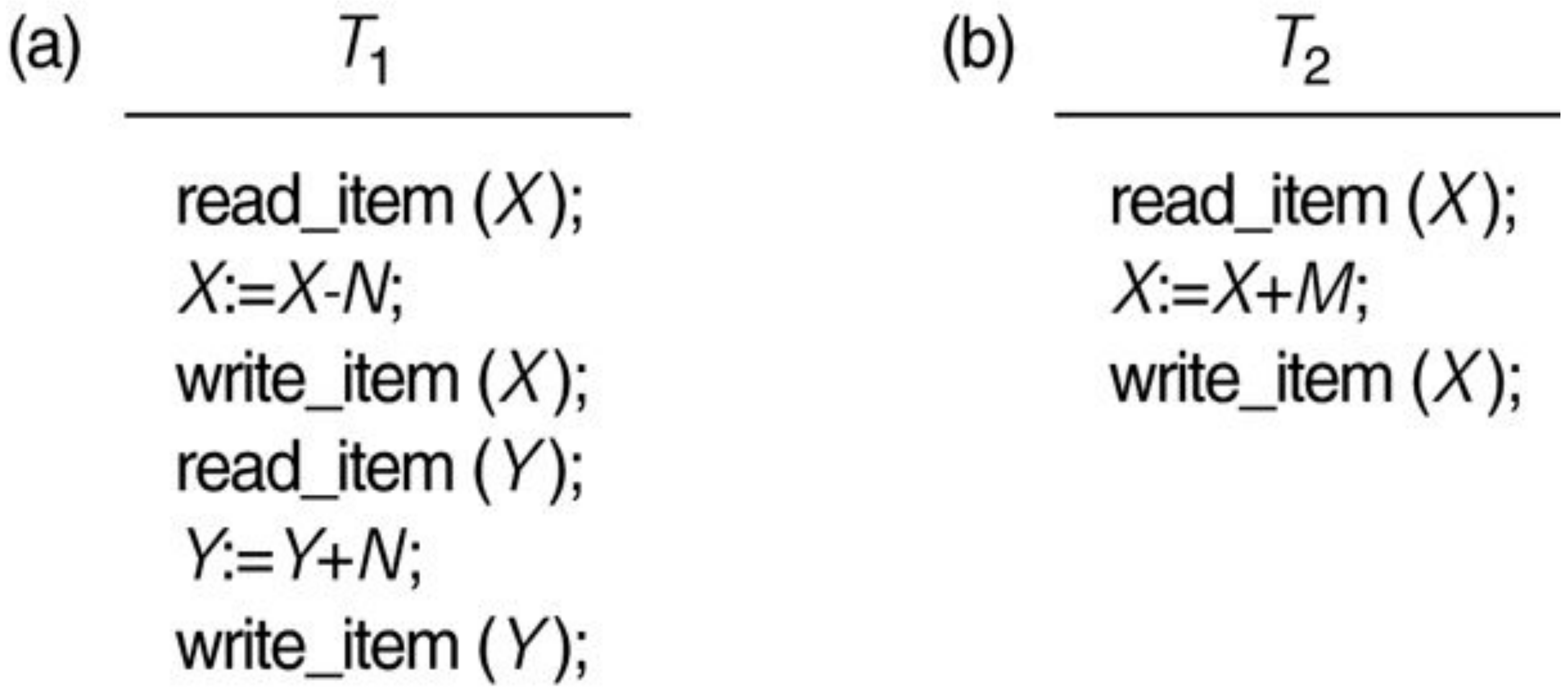
READ AND WRITE OPERATIONS (contd.):

- **write_item(X)** command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the program variable named X into its correct location in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).

- Usually, the decision about when to store a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system

Two sample transactions

- Consider two sample transactions:
  - (a) Transaction T1
  - (b) Transaction T2



(a) T_1

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

(b) T_2

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

Consider X = 80 , N= 5, M= 4 , Y= 50

Transaction T1

- Read (X) = (80)
- X = X- N = 80 – 5= 75
- Write (X) = 75
- Read (Y) = 50
- Y =Y+N = 50 + 5 = 55
- Write (Y) = 55

Transaction T2

- Read (X) = (75)
- X = X + M = 75+ 4= 79
- Write (X) = 79

- The **read-set** of a transaction is the set of all items that the transaction reads
- the **write-set** is the set of all items that the transaction writes.
- For example, in previous slide,

  the read-set of T1 is {X, Y} and write-set is also {X, Y}.

  the read-set of T2 is {X} and write-set is also {X}.

Case study Example:

- Consider an airline reservations database in which a record is stored for each airline flight.
- Each record includes the number of reserved seats on that flight.
- When a database access program is written, it has the flight number, flight date, and the number of seats to be booked as parameters;
- For concurrency control purposes, a transaction is a particular execution of a program on a specific date, flight, and number of seats.

From the previous slide, consider a airline database with two flights F1 and F2

- X = no. of seats reserved on F1
- Y = no. of seats reserved on F2
- N= number of seats to be transferred from F1 to F2

Hence

- transaction T1 □ transfers N seats from F1 to F2
- transaction T2 □ reserves new seats “M” to flight F1

Why concurrency and recovery are needed?

- Transactions submitted by the various users may execute concurrently and may access and update the same database items.
- If this concurrent execution is uncontrolled, it may lead to problems, such as an inconsistent database.
- Hence , Concurrency control and recovery mechanisms are required which are mainly concerned with the database commands in a transaction.

Why Concurrency Control is needed:

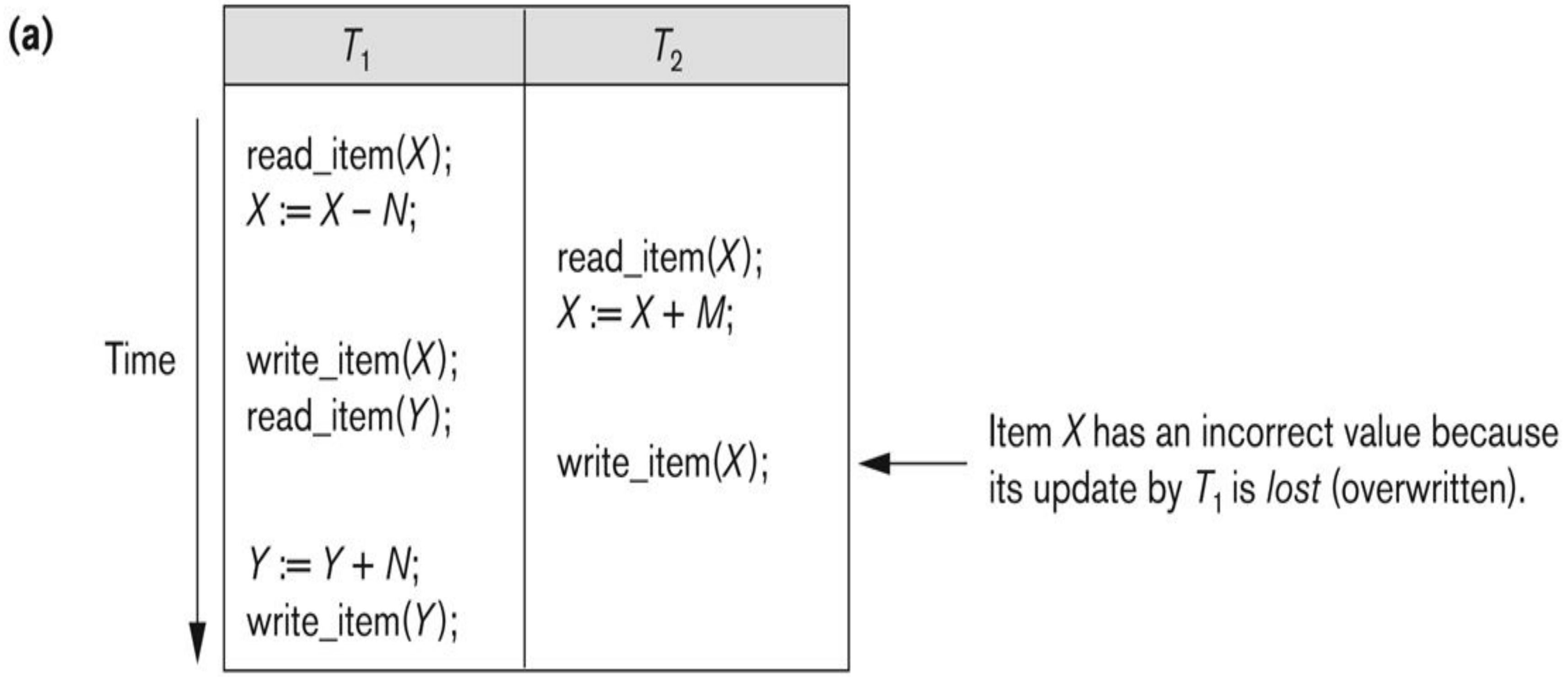
1) The Lost Update Problem

- *This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect*.
- ex: consider the transactions T1 and T2 are submitted at same time and suppose their operations are interleaved.
- Then the final value of X is incorrect bcoz T2 reads the value of X before T1 updates it and hence the updated value of X is lost.
- (refer next slide) Consider the initial value of X=80, N=5 and M=4 and solve the example.

Concurrent execution is uncontrolled: (a) The lost update problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(a)



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);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

Consider X = 80 , N= 5, M= 4 , Y= 50 and the transactions are Interleaved

Transaction T1

- Read (X) = (80)
  X = X - N = 80 – 5= 75
- Write (X) = 84
  Read (Y) = 50
- Y =Y+N = 50 + 5 = 55
  Write (Y) = 55

Transaction T2

- Read (X) = (80)
  X = X + M = 80+ 4= 84
- Write (X) = 84

X contains wrong value 84 instead of correct value 79 which was updated by T1

Why Concurrency Control is needed:

2) The Temporary Update (or Dirty Read) Problem

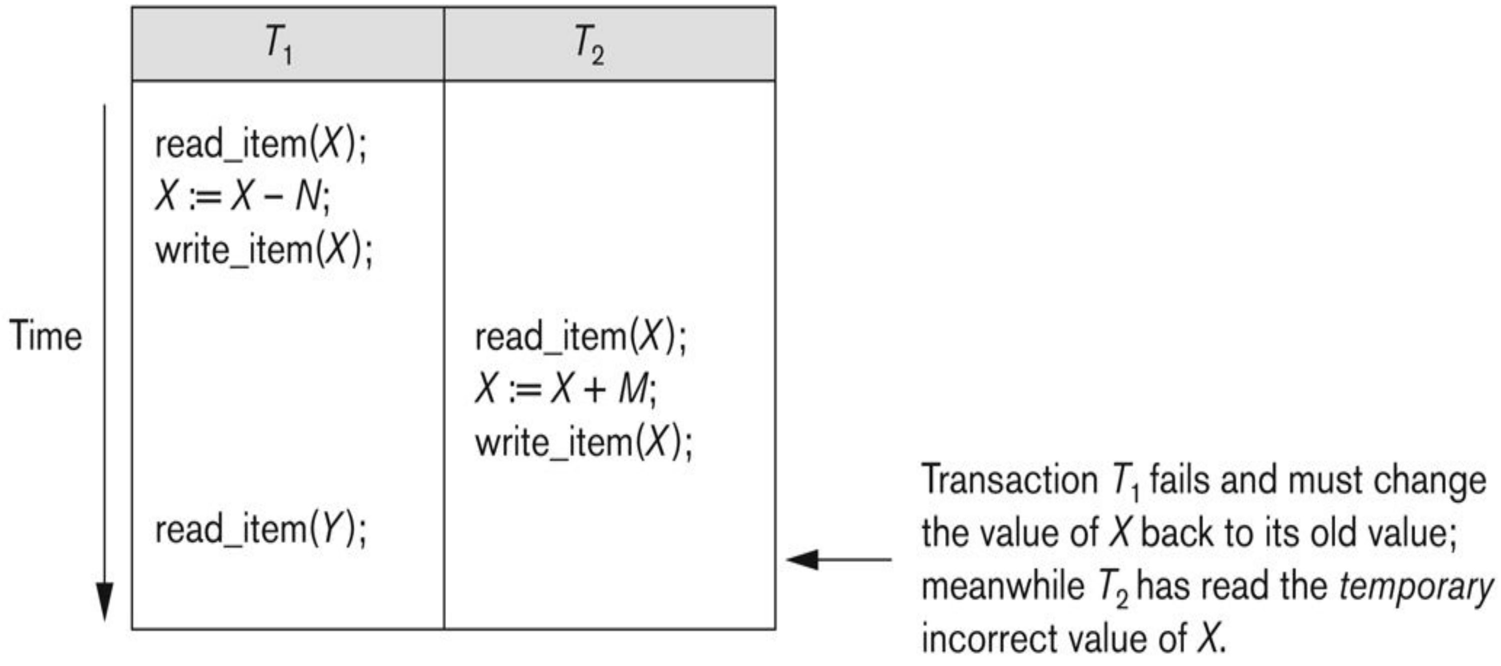
- This occurs when one transaction updates a database item and then the transaction fails for some reason
- Meanwhile, The updated item is accessed by another transaction before it is changed back to its original value by the first transaction.
- the temporary data read by another transaction is called dirty data because it is created by a transaction that is not completed and committed yet.

- consider next slide (17.3 b) where T1 updates value of X but fails before completing the transaction T1.
- hence it has to restore the old value of X. but before restoring, T2 reads temporary value of X which is incorrect (dirty data).

Concurrent execution is uncontrolled: (b) The temporary update problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(b)



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);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

Why Recovery Is Needed

- Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that **either all the operations** in the transaction are completed successfully and their effect is recorded permanently in the database, or the transaction **has no effect** whatsoever on the database or on any other transactions.
- The DBMS must not permit some operations of a transaction *T* to be applied to the database while other operations of T are not applied.

Types of failures:
a) transaction b) system c) media

(What causes a Transaction to fail)

1. A computer failure (system crash):

   A **hardware crash, software or network error** occurs in the computer system during transaction execution.

2. A transaction or system error:

   Some operation in the transaction may cause it to fail, such as **integer overflow , division by zero,** erroneous parameter values, **user interrupt** or because of a logical programming error.

Why **recovery** is needed (Contd.):

3. Local errors or exception conditions detected by the transaction:

Certain conditions may require cancellation of the transaction. For example, **data for the transaction may not be found**, **insufficient account balance** in a banking database

4. Concurrency control enforcement:

The concurrency control method may decide to abort the transaction, it **violates serializability** or because **transactions are in a state of deadlock**

Why **recovery** is needed (contd.):

5. Disk failure:

During a read or a write operation of the transaction, some disk blocks may lose their data because of a read or write malfunction or a **disk read/write head crash.**

6. Physical problems and catastrophes:

This includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Transaction and System Concepts

- A **transaction** is an **atomic** unit of work that is either completed in its **entirety** or not done at all.
  - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

Transaction and System Concepts (2)

- Recovery manager keeps track of the following operations:
  - **begin_transaction**: This marks the beginning of transaction execution.
  - **read** or **write**: These specify read or write operations on the database items that are executed as part of a transaction.
  - **end_transaction**: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
    - At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

Transaction and System Concepts (3)

- **commit_transaction**: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- **rollback** (or **abort**): This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

Transaction and System Concepts (4)

- **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
- **redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

State transition diagram illustrating the states for transaction execution

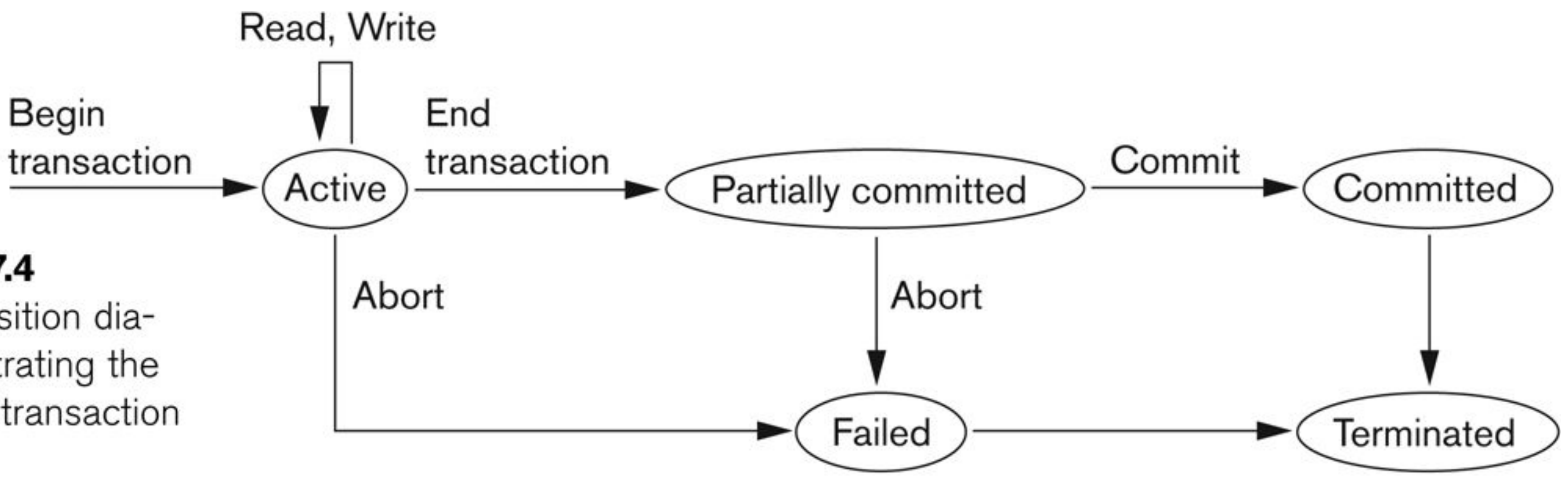


Figure 17.4
State transition diagram illustrating the states for transaction execution.

- A transaction goes into **an active state** immediately after it starts execution, where it can execute its READ and WRITE operations. When the transaction ends, it moves to the **partially committed state.** At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently
- Once this check is successful, the transaction is said to have reached its commit point and enters **the committed state**.
- a transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state..
- The **terminated state** corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates.

Desirable Properties of Transactions

ACID (Atomicity, Consistency , Isolation, Durability) properties:

- **Atomicity**: *A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.*

  the transaction recovery subsystem ensures atomicity. And if any transaction fails to complete then the effects of the transaction on the database must be undone.

- **Consistency preservation**: *A correct execution of the transaction must take the database from one consistent state to another.*

  It is responsibility of programmer or DBMS module that enforces integrity constraints to ensure consistency.

ACID properties continued.

- **Isolation**: *A transaction should appear as though it is being executed in isolation from other transactions.*
  A transaction **should not make its updates visible** to other transactions until it is committed; Isolation is enforced by **concurrency control subsystem of DBMS.**

- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.
  It is enforced by recovery subsystem of DBMS.

CHAPTER
CONCURRENCY CONTROL TECHNIQUES

Database Concurrency Control

- Purpose of Concurrency Control
  - To **enforce Isolation** when there is conflict among transactions.
  - To **preserve database consistency**
  - To **resolve read-write and write-write conflicts**.

- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the access to A and if the other transaction is rolled-back or waits.

serializability

- serializability is a property of a system describing how different processes operate on shared data.
- A system is serializable if its result is the same as if the operations were executed in some sequential order, meaning there is no overlap or interleaving in execution of transactions.
- serializability can be accomplished in DBMS by locking data so that no other process can access it while it is being read or written.

Locks:

- ***A lock is a variable associated with a data item that describes the status of the item w.r.t possible operations that can be applied to it.***
- This is a procedure used to **control concurrent access** to data.
- When one transaction is accessing that database, a lock may deny access to other transactions to prevent incorrect results.

Locking

- **Locking** methods are the most widely used approach to ensure **serializability** of concurrent transactions.
- Data items of various sizes, ranging from the entire database to a field, may be locked.
- ***Read lock*** *- If a transaction has a read lock on a data item, it can read the item but not update it.*
- ***Write lock*** *– If a transaction has a write lock on a data item, it can both read and update the item.*
- The fundamental principle is that a transaction must **claim a read (shared) or write (exclusive) lock on a data item** before the corresponding database read or write operation.

  The lock prevents another transaction from modifying the item or even reading it, in the case of a write lock.

Locks are used in the following way:

- Any transaction that needs to access a data item must **first lock the item**, requesting a **read lock** for read only access or a **write lock** for both read and write access.
- If the **item is not already locked** by another transaction, the **lock is granted**.
- If the item is **currently locked**, the DBMS determines whether the **request is compatible** with the existing lock.

- If a read lock is requested on an item that already has a read lock on it, the request will be granted; **otherwise, the transaction must wait** until the existing lock is released.
- A transaction continues to hold a lock **until it explicitly releases it** either during execution or **when it terminates** (aborts or commits).
- However, It is only when the **write lock has been released** that the **effects of the write operation will be made visible** to other transactions.

- Some systems also permit a transaction to **issue a read lock** on an item and then **later upgrade the lock to a write lock.**
- This effectively allows a transaction to examine the data first and then decide whether it wishes to update it.
- If **upgrading is not supported**, a transaction must hold **write locks on all data items** that it may update at some time during the execution of the transactions, thereby potentially reducing the level of concurrency in the system.

- For the same reason, some systems also permit a transaction to **issue a write lock and then later to down grade the lock to a read lock.**
- Using locks in transactions does not guarantee serializability of schedules by themselves.
- To guarantee serializability, we must follow an additional protocol concerning the positioning of the lock and unlock operation in every transaction. The best-known protocol is **two-phase locking (2PL).**

Types of Locks

1) Binary Locks

- A **binary lock** can have two **states** or **values:**
  locked (1) and unlocked (0).
- 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.
- the current value (or state) of the lock associated with item X is referred as **LOCK(*X*).**
- This is not very practical type of lock

Operations used with binary locking

- **lock_item and unlock_item**, are used with binary locking.
- A transaction requests access to an item *X* by first checking 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*.
- Hence, a binary lock enforces mutual exclusion on the data item

Implementation of a binary lock

- It needs 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 has a **lock manager sub system** to keep track of and control access to locks.

```
lock_item(X):
B:  if LOCK(X) = 0                      (* item is unlocked *)
          then LOCK(X) ←1               (* lock the item *)
    else
          begin
          wait (until LOCK(X) = 0
                and the lock manager wakes up the transaction);
          go to B
          end;
unlock_item(X):
    LOCK(X) ← 0;                              (* unlock the item *)
    if any transactions are waiting
          then wakeup one of the waiting transactions;
```

2) Shared/Exclusive (or Read/Write) Locks

- The **binary locking scheme is too restrictive** for database items, because at most one transaction can hold a lock on a given item.
- We can 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*).

- Hence , A lock associated with an 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_transaction(s)>

- Again, to save space, the system need maintain lock records only for locked items in the lock table.
- The value (state) of LOCK is either read-locked or write-locked, suitably coded

	S	X
S	✔	✗
X	✗	✗

- If the transaction T1 is holding a shared lock in data item A, then the control manager can grant the shared lock to transaction T2 as compatibility is TRUE, but it cannot grant the exclusive lock as compatibility is FALSE.
- Similarly if an exclusive lock (i.e. lock for read and write operations) is hold on the data item in some transaction then no other transaction can acquire Shared or Exclusive lock as the compatibility function denoted FALSE.

```
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 it 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;
```

Conversion of Locks

- Sometimes it is desirable 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.
- For example, 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_lock(*X*) operation.
- If T is the only transaction holding a read lock on *X* at the time, then it issues the write_lock(*X*) operation, i.e the lock can be **upgraded**; otherwise, the transaction must wait.
- It is also possible for a transaction T to issue a write_lock(*X*) and then later on to **downgrade** the lock by issuing a read_lock(*X*) operation.
- When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock to store the information about on which transactions hold locks on the item.

Two-phase locking (2PL)

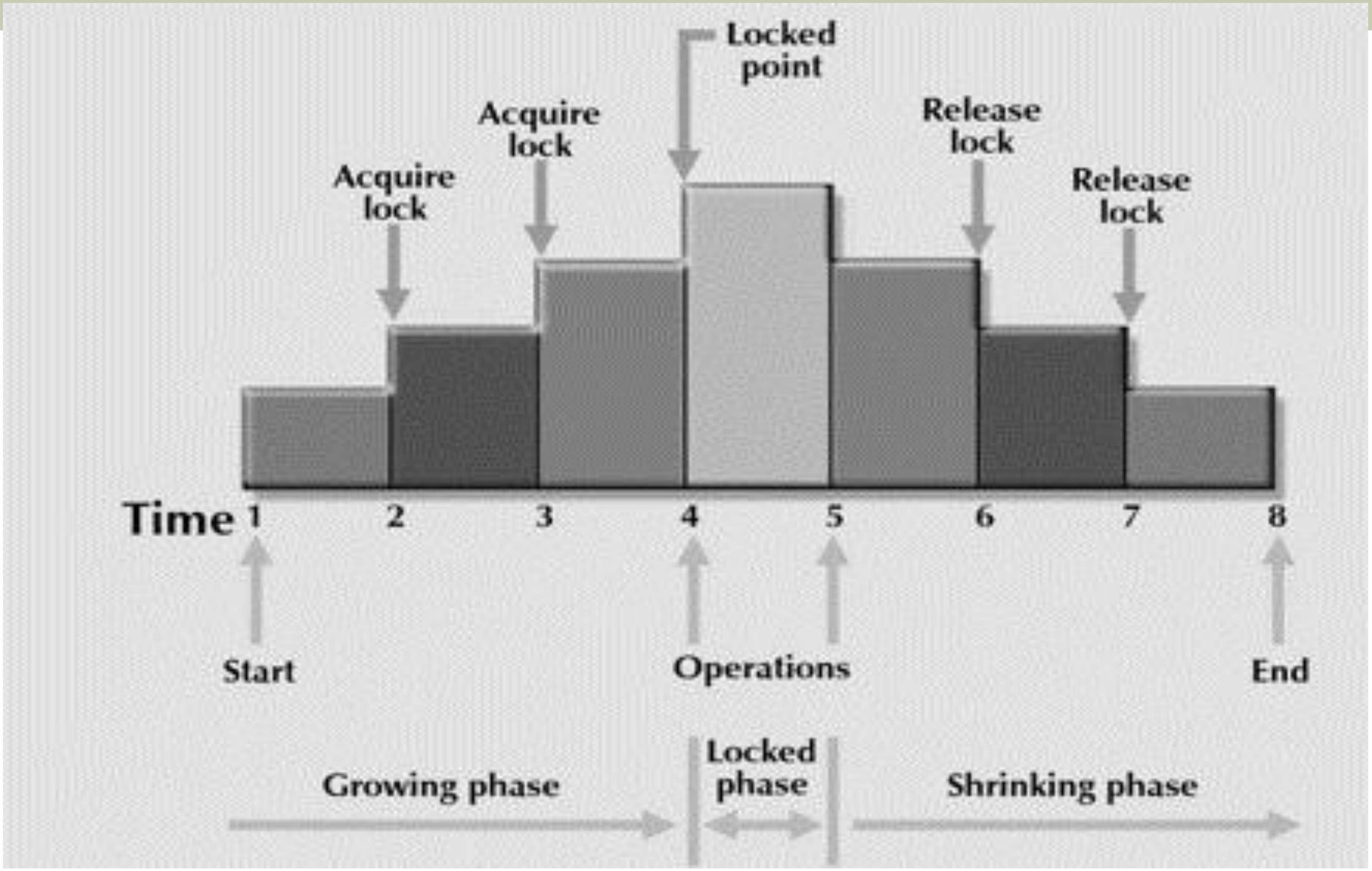
- The two-phase locking protocol divides the execution phase of the transaction into three parts.
  - **first part** when the execution of the transaction starts, it seeks permission for the lock it requires.
  - **second part** the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
  - **third part** the transaction cannot demand any new locks. It only releases the acquired locks.

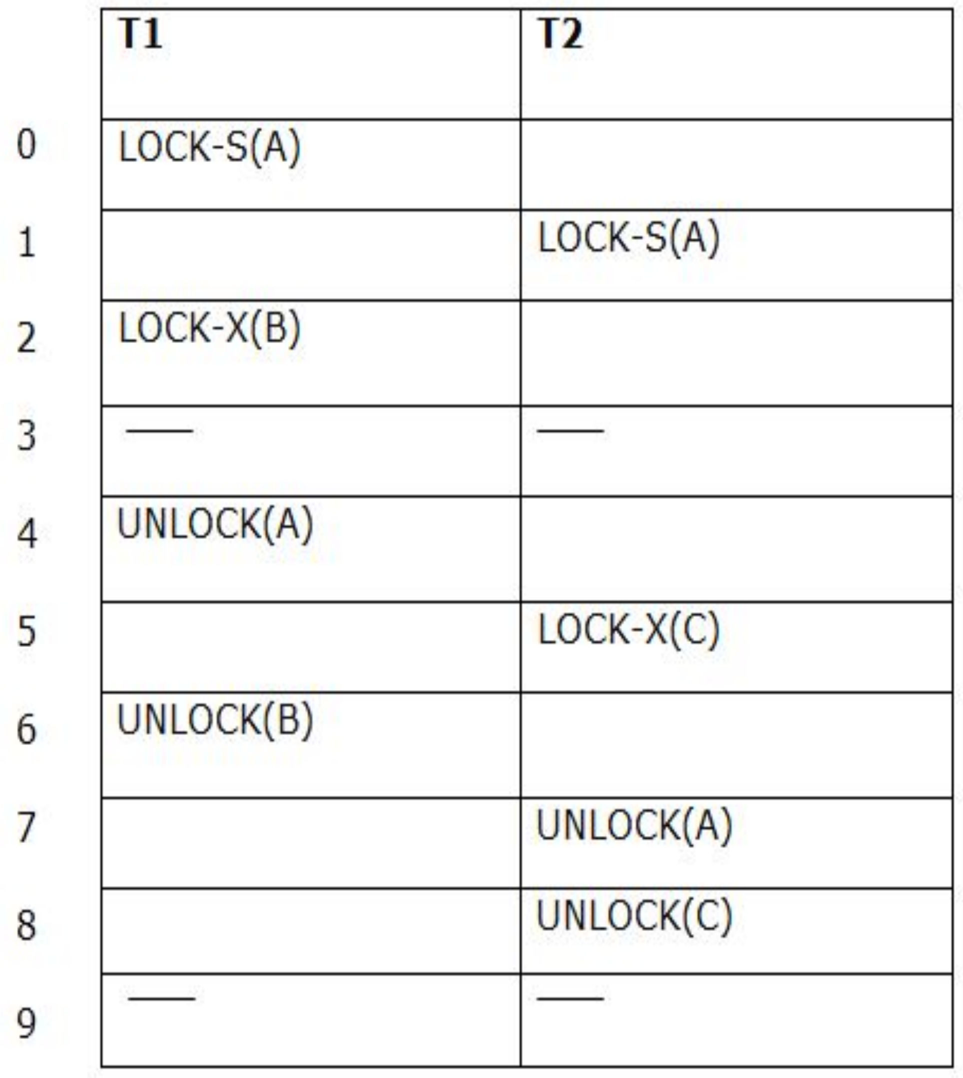
There are two phases of 2PL:

- **Growing phase:** In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.
- **Shrinking phase:** In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.
- In the growing phase transaction reaches a point where all the locks it may need has been acquired. This point is called **LOCK POINT.**

if lock conversion is allowed then the following phase can happen:

- Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
- Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.


Locked point
Acquire lock
Acquire lock
Release lock
Release lock
Time 1 2 3 4 5 6 7 8
Start
Operations
End
Growing phase
Locked phase
Shrinking phase



	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	—	—
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	—	—

Transaction T1:

- **Growing phase:** from step 1-3
- **Shrinking phase:** from step 5-7
- **Lock point:** at 3

Transaction T2:

- **Growing phase:** from step 2-6
- **Shrinking phase:** from step 8-9
- **Lock point:** at 6

Types of 2PL

1) Strict 2PL

- It can release a shared lock at lock point but not exclusive locks until the transaction commits.
- This protocol creates a cascade less schedule.

2) Rigorous 2PL:

- In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts.
- It guarantees serializability

3) Conservative or static 2PL

- Here before a transaction begins, it has to lock all the items it accesses by pre-declaring its read-set and write-set.
- So once the transaction starts it is in its shrinking phase.
- This is a deadlock-free protocol

Timestamp Ordering

- **The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.**
- A timestamp is a unique identifier assigned by the DBMS to every transaction when it enters the system.
- Hence, the conflicting pair of tasks are executed according to the timestamp values of the transactions. I.e. Older transactions are executed first and then younger transaction.
- For example Assume:
- T1 first enters a system , TS(T1) = 100 (older transaction)
- Next T2 enters the system , TS(T2) = 101 (younger transaction)

- The timestamp of transaction Ti is denoted as TS(Ti).
  - Read time-stamp of data-item X it is the latest transaction number which performed read(X) successfully and is denoted by

    R-timestamp(X) or RTS(X)
  - Write time-stamp of data-item X   it is the latest transaction number which performed write(X) successfully is denoted by W-timestamp(X) or WTS(X)

- Example:

  If T1 reads a data item X at TS value 100 and then T2 reads same data item X at TS value 101.

  Then RTS(X) = T2 = 101

Basic Timestamp ordering protocol follows these rules:

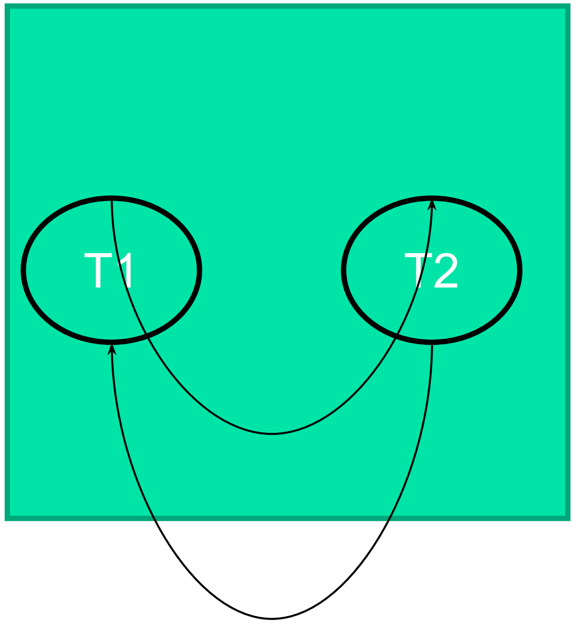
1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:
   - If W_TS(X) >TS(Ti) then the operation is rejected.
   - If W_TS(X) <= TS(Ti) then the operation is executed.
   - Timestamps of all the data items are updated.
2. Check the following condition whenever a transaction Ti issues a **Write(X)** operation:
   - If TS(Ti) < R_TS(X) then the operation is rejected.
   - If TS(Ti) < W_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

- A conflict occurs when an older transaction tries to read/write a value already read or written by a younger transaction.
- Read or write proceeds only if the last update on that data item was carried out by an older transaction.
- Otherwise, the transaction requesting read/write is restarted and given a new timestamp.
- This protocol ensures serializability.
- Here no locks are used so no deadlock.

Deadlocks in transactions

- 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 to release the lock on an item.
- A simple example is shown :
  - two transactions T1 and T2 are deadlocked in a schedule;
  - T1 is on the waiting queue for X, which is locked by T2
  - T2 is on the waiting queue for Y, which is locked by T1.
  - Meanwhile, neither T1 nor T2 nor any other transaction can access items X and Y.

T1	**T2**
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
	write_lock(Y)



- Deadlock prevention protocols ensure that the system will never enter into a deadlock state .
- Some prevention strategies are :
  - Each transactions locks all its data items before it begins execution.
  - Imposes a partial ordering of all data items that a transaction can lock

Following are the two schemes for deadlock prevention

- **Wait-Die Scheme :** Older transaction may wait for younger one to release the dataitem. Younger transactions never wait for older ones, they are rolled back instead.
- **Wound-Wait Scheme :** Older transaction forces roll back of younger transaction instead of waiting. Younger transactions may wait for older ones.

Starvation

- **Starvation occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.**
- This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others.
- One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-serve queue;**
- 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.