MODULE 5
Transactions, Concurrency and Recovery, Recent Topics

Sindhu Jose, CSE Dept, VJCET

SYLLABUS

- Transaction Processing Concepts - overview of concurrency control, Transaction Model, Significance of concurrency Control & Recovery, Transaction States, System Log, Desirable Properties of transactions. Serial schedules, Concurrent and Serializable Schedules, Conflict equivalence and conflict serializability, Recoverable and cascade-less schedules, Locking, Two-phase locking and its variations. Log-based recovery, Deferred database modification, check-pointing.
- Introduction to NoSQL Databases, Main characteristics of Key-value DB (examples from: Redis), Document DB (examples from: MongoDB) ,Main characteristics of Column - Family DB (examples from: Cassandra) and Graph DB (examples from : ArangoDB)

Transaction Processing Concepts: overview of concurrency control and recovery

Single-User versus Multiuser Systems

- One criterion for classifying a database system is according to the number of users who can use the system concurrently.
- A DBMS is **single-user** if at most one user at a time can use the system,
- **Multiuser** if many users can use the system and hence access the database concurrently.
- Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser.
- Multiple users can access databases and use computer systems simultaneously.
- Concept of multiprogramming allows the operating system of the computer to execute multiple programs or processes at the same time.

- A single central processing unit (CPU) can only execute at most one process at a time.
- A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again.
- Hence, concurrent execution of processes is actually interleaved, as illustrated in Figure(next slide) shows two processes, A and B, executing concurrently in an interleaved fashion.
- Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk.
- Interleaving also prevents a long process from delaying other processes.

- If the computer system has multiple hardware processors (CPUs), parallel processing of multiple processes is possible, as illustrated by processes C and D in Figure .
- Most of the theory concerning <u>concurrency control</u> in databases is developed in terms of <u>interleaved concurrency</u>

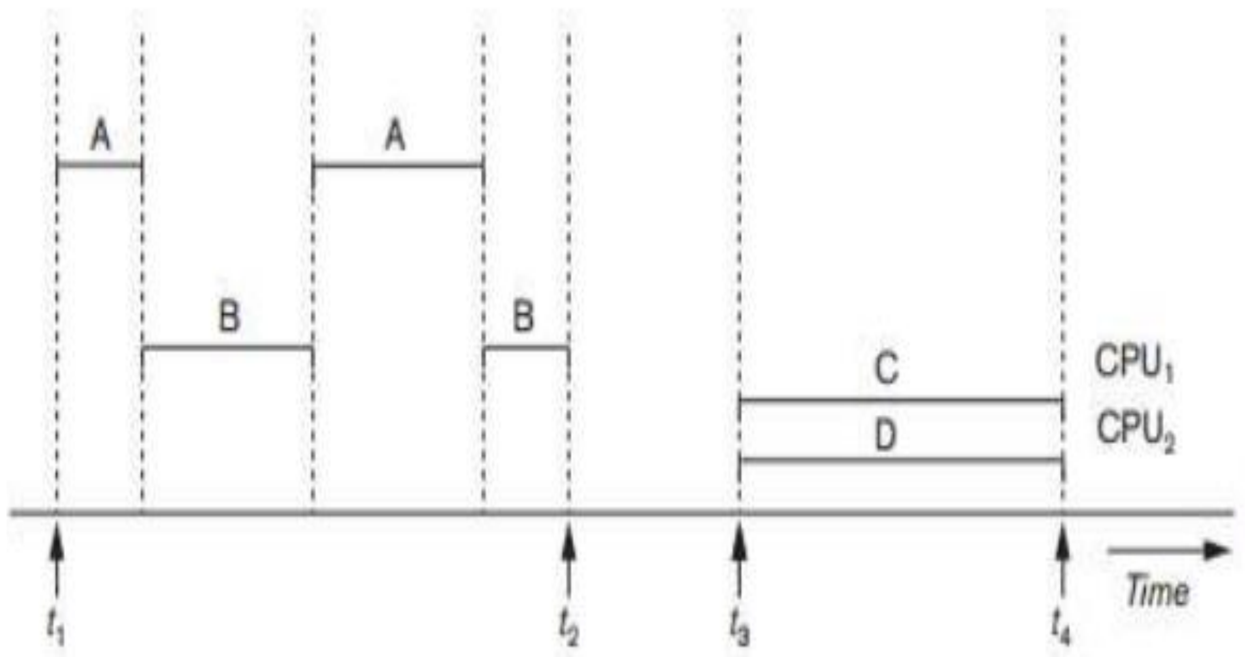


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

Transactions, Database Items, Read and Write Operations

- A **transaction** is an executing program that forms a logical unit of database processing.
- A transaction includes one or more database access operations → These can include insertion, deletion, modification, or retrieval operations.
- One way of specifying the transaction boundaries is by specifying explicit begin transaction end transaction statements in an application program
- A database is basically represented as a collection of named **data items**.
- The size of a data item is called its **granularity**. Fine granularity refers to small Item sizes, where as Coarse granularity refers to large item sizes.
- The **basic unit of data transfer** from disk to main memory is **one block.**

- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a read-only transaction
- Otherwise it is known as a read-write transaction.
- The basic database access operations that a transaction can include are as follows:
- **read_item(X)** - Reads a database item named X into a program variable.
- Executing a read_item(X) command includes the following steps:

  1. Find the address of the disk block that contains item X.
  2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  3. Copy item X from the buffer to the program variable named X.

- **write_item(X)** - Writes the value of program variable X into the database item named X.
- Executing a write_item(X) command includes the following steps:

  1. Find the address of the disk block that contains item X.
  2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  3. Copy item X from the program variable named X into its correct location in the buffer.
  4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Serial and Concurrent transaction

- **Serial transaction:** Transaction one after the other or one at a time in some order

  Concurrent transaction: More then one transactions performing simultaneously

  Eg: Railway reservation

Why Concurrency Control Is Needed

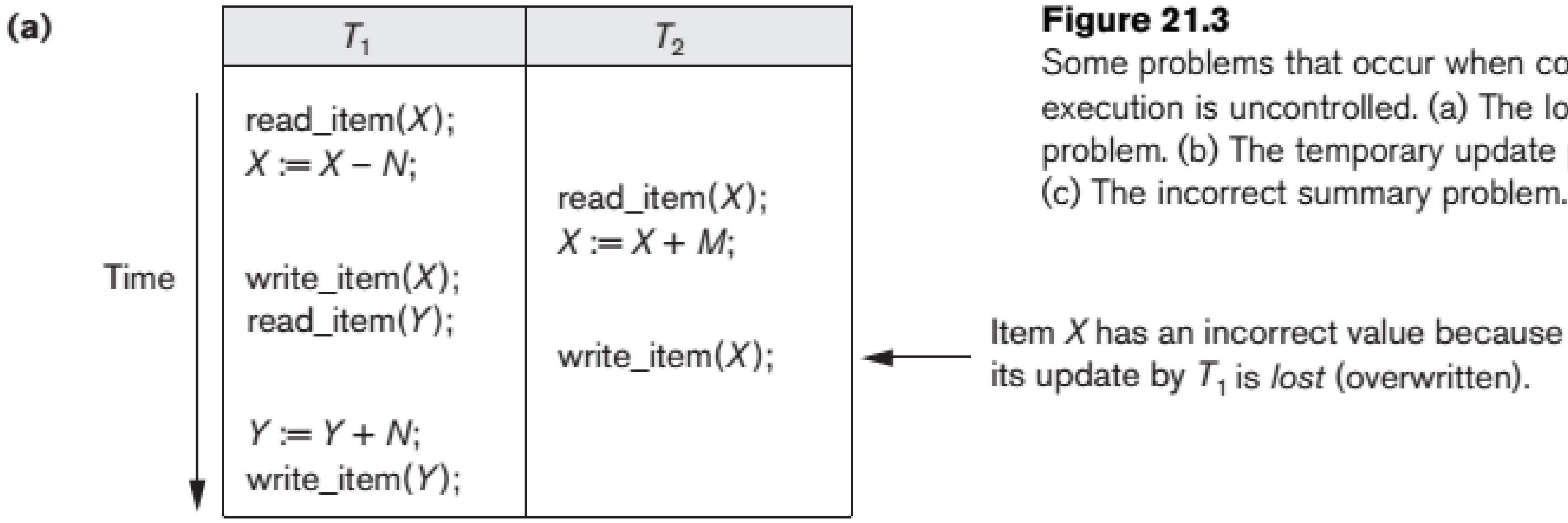
- Several problems can occur when concurrent transactions execute in an uncontrolled manner.

Types of Problems On Concurrent Transactions

1. The Lost Update Problem (Write - Write Conflict)
2. The inconsistent retrievals problem
3. Dirty read (Temporary Update/ Write - Read conflict) problem
4. The Unrepeatable Read Problem (Read - Write Conflict)

The lost update problem (WW Conflict)

- 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.
- Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved.


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

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);	

Figure 21.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

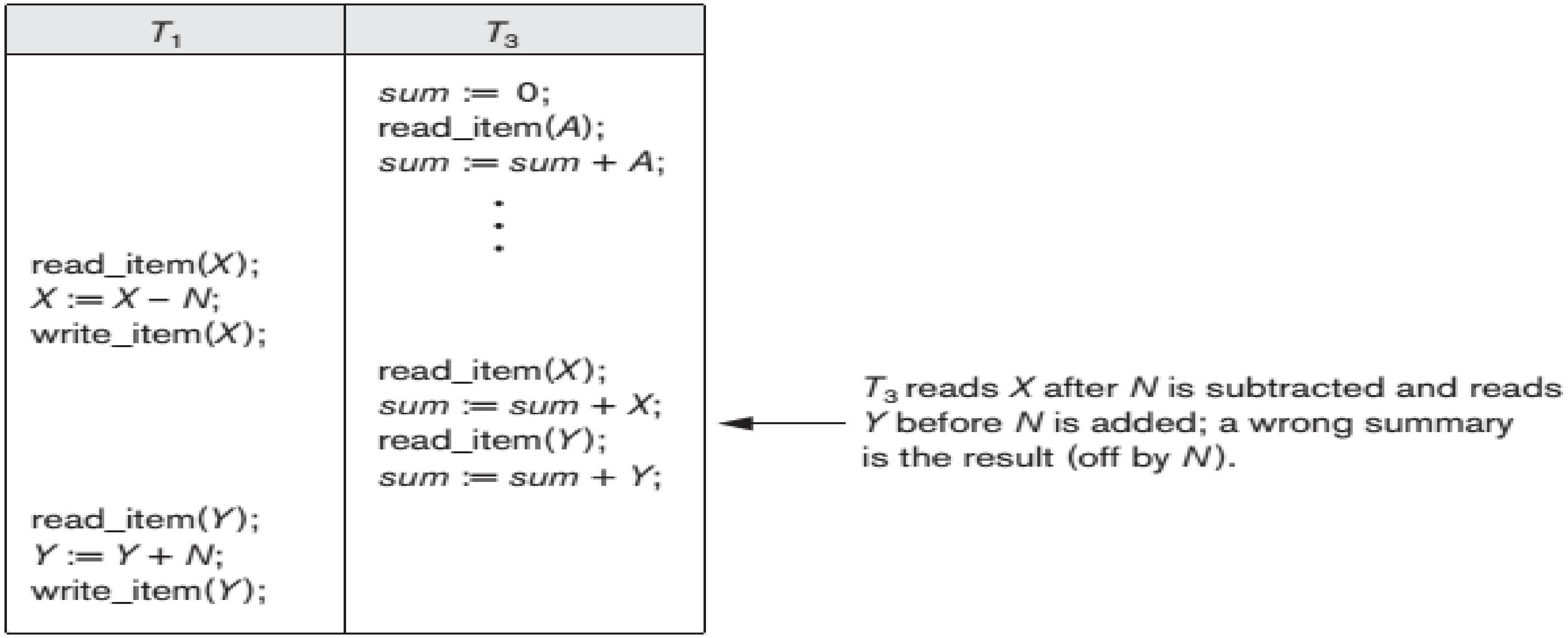
- Then the final value of item X is incorrect because T2 reads the value of X before T1 changes it in the database, and hence the updated value resulting from T1 is lost.

- If X = 80 at the start, N = 5 and M = 4 , then the final result should be X = 79.
- However, in the interleaving of operations shown in Figure 21.3(a), it is X = 84 because the update in T1that reduced 5 from X was lost.

The inconsistent retrievals problem

- If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.

(c)



T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	⋮

read_item(*X*);
X := *X* − *N*;
write_item(*X*);	

	read_item(*X*);
	sum := *sum* + *X*;
	read_item(*Y*);
	sum := *sum* + *Y*;

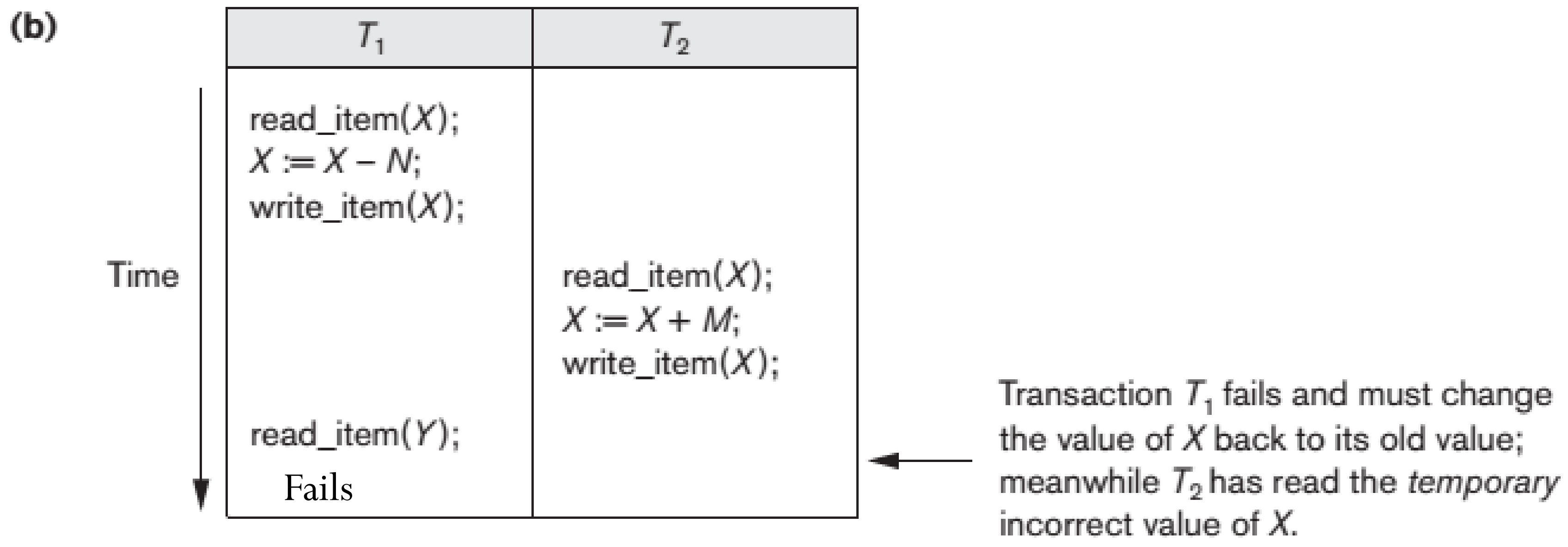
read_item(*Y*);
Y := *Y* + *N*;
write_item(*Y*);	

T_3 reads *X* after *N* is subtracted and reads *Y* before *N* is added; a wrong summary is the result (off by *N*).

- If the interleaving of operations occurs, the result of T3 will be off by an amount N because T3 reads the value of X after N is subtracted from it but reads the value of Y before N is added to it.

Dirty read (Temporary Update/ WR conflict) problem

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



(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); Fails	

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.

- T1 updates item X and then fails before completion, so the system must change X back to its original value.
- Before it can do so, however, transaction T2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T1.
- The value of item X that is read by T2 is called **dirty data** because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the **dirty read problem.**

The Unrepeatable Read Problem (RW Conflict)

- Another problem that may occur is called unrepeatable read, where a transaction T reads the same 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 inquiries 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, and it may end up reading a different value for the item.

Transaction States and Additional Operations

- A transaction is an atomic unit of work that should either be completed in its entirety or not done at all.

- For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits or aborts .

- Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

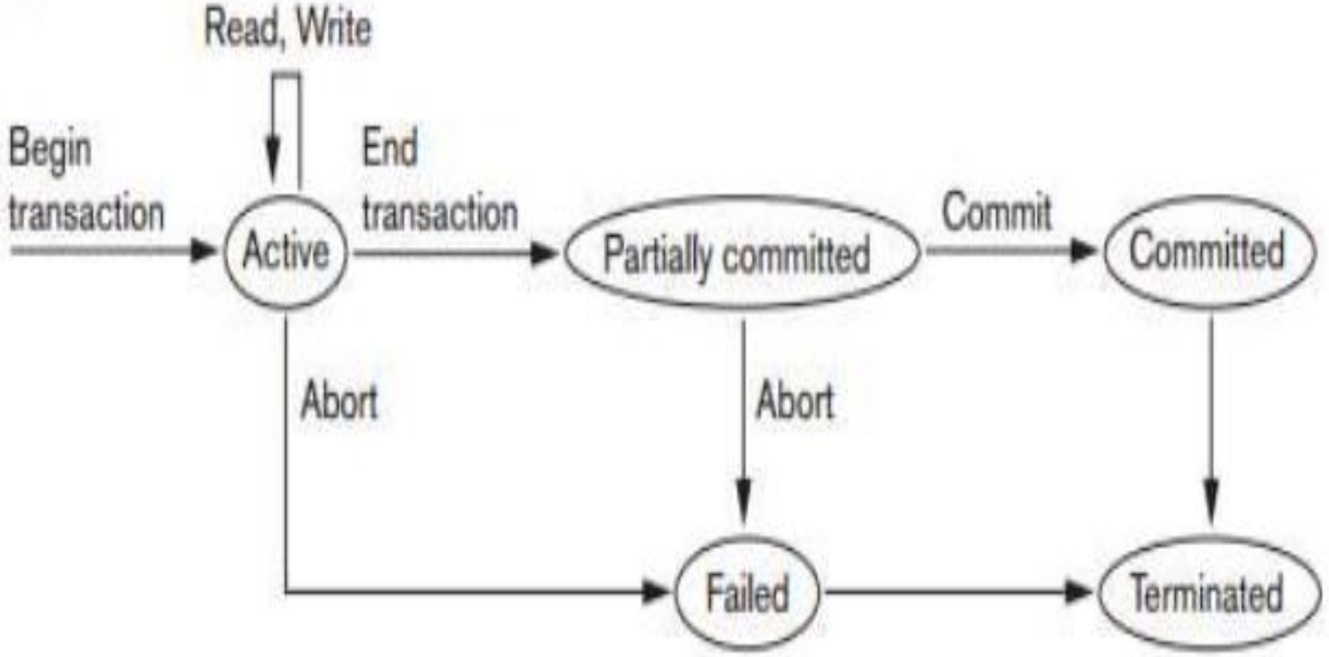
1. BEGIN_TRANSACTION → This marks the beginning of transaction execution.
2. READ or WRITE → These specify read or write operations on the database items that are executed as part of a transaction.

3. END_TRANSACTION → This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution.
   - Temporary. Committed or not, Permanent

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.

4. 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.

Figure 21.4
State transition diagram illustrating the states for transaction execution.



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**
  - That the transaction does not have any effect on the database or any other transactions.

- In the first case, the transaction is said to be **committed**, whereas in the second case, the transaction is **aborted**.

- If a transaction fails after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Types of failures

1. **A computer failure (system crash):** A hardware or software error occurs in the computer system during transaction execution.
   - If the hardware crashes, the contents of the computer's internal memory may be lost.
2. **A transaction or system error**: Some operation in the transaction may cause it to fail, such as integer overflow or division by zero.
   - Transaction failure may also occur because of erroneous parameter values or because of a logical programming error.
   - In addition, the user may interrupt the transaction during its execution.
3. **Local errors or exception conditions detected by the transaction**: Certain conditions necessitate cancellation of the transaction.
   - For example, data for the transaction may not be found.
   - A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled

4. **Concurrency control enforcement:**
   - The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

5. **Disk failure:**
   - Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.
   - This may happen during a read or a write operation of the transaction.

6. **Physical problems and catastrophes:**
   - This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

ACID Properties

- Transactions should possess several properties, often called the ACID properties
- They should be enforced by the <u>concurrency control and recovery methods</u> of the DBMS.
- The following are the ACID properties:

1. Atomicity
2. Consistency
3. Isolation
4. Durability

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

- All or Nothing

Consistency preservation: A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

Isolation: A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently.

- That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

Durability or permanency: The changes applied to the database by a committed transaction must persist in the database.

- These changes must not be lost because of any failure.

- 2 accounts A & B.   A initially has 1000 Rs, B has 500 Rs
- A → B

Read (A)

A = A – 100

Write (A)   → 900

Commit

Read (B)

B = B + 100

Write (B)   → 600

Commit

Finally   A=900 and B=600

The System Log (Journal)

- To be able to recover from failures that affect transactions, the system maintains
  - A log to keep track of all transaction operations that affect the values of database items, and
  - Other transaction information that may be needed to permit recovery from failures.

- The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.

- Typically, one (or more) main memory buffers hold the last part of the log file, so that log entries are first added to the main memory buffer.

- In addition, the log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures.

The following are the types of entries — called **log records**

1. [start_transaction, T] → Indicates that transaction T has started execution.

2. [write_item, T, X, old_value, new_value] → Indicates that transaction T has changed the value of database item X from old_value to new_value.

3. [read_item, T, X] → Indicates that transaction T has read the value of database item X.

4. [commit, T] → Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.

5. [abort, T] → Indicates that transaction T has been aborted.

- In these entries, T refers to a unique transaction-id that is generated automatically by the system for each transaction and that is used to identify each transaction.

- Because the log contains a record of every WRITE operation that changes the value of some database item,
  - It is possible to **undo** the effect of these WRITE operations of a transaction T by tracing backward through the log and resetting all items changed by a WRITE operation of T to their old_values.

- **Redo** of an operation may also be necessary if a transaction has its updates recorded in the log but a failure occurs before the system can be sure that all these new_value shave been written to the actual database on disk from the main memory buffers.

Commit Point of a Transaction

- A transaction T reaches its commit point when
  - All its operations that access the database have been executed successfully and
  - The effect of all the transaction operations on the database have been recorded in the log.
- Beyond the commit point:
  - Its effect must be permanently recorded in the database.
  - The transaction then writes a commit record [commit, T] into the log.

- If a system failure occurs, we can search back in the log for all transactions T that have written a [start_transaction, T] record into the log but have not written their [commit, T] record yet
  - These transactions may have to be rolled back to undo their effect on the database during the recovery process.

- It is common to keep one or more blocks of the log file in main memory buffers, called the log buffer, until they are filled with log entries and then to write them back to disk only once, rather than writing to disk every time a log entry is added.
- This saves the overhead of multiple disk writes of the same log file buffer.
- At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process because the contents of main memory may be lost.
- Hence, before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk.
- This process is called **force-writing the log buffer** before committing a transaction.

Schedules (Histories) of Transactions

- A **schedule (or history**) S of n transactions T1,T2, …,Tn is an ordering of the operations of the transactions.
- Operations from different transactions can be interleaved in the schedule S.
- However, for each transaction Ti that participates in the schedule S, the operations of Ti in S must appear in the same order in which they occur in Ti.
- The order of operations in S is considered to be a total ordering, meaning that for any two operations in the schedule, one must occur before the other.

- A shorthand notation for describing a schedule uses the symbols:
  - b, r, w, e, c, and a : - The operations begin_transaction, read_item, write_item, end_transaction, commit, and abort, respectively,
  - and appends as a subscript the transaction id (transaction number) to each operation in the schedule.
  - In this notation, the database item **X** that is read or written follows the **r** and **w** operations in parentheses.

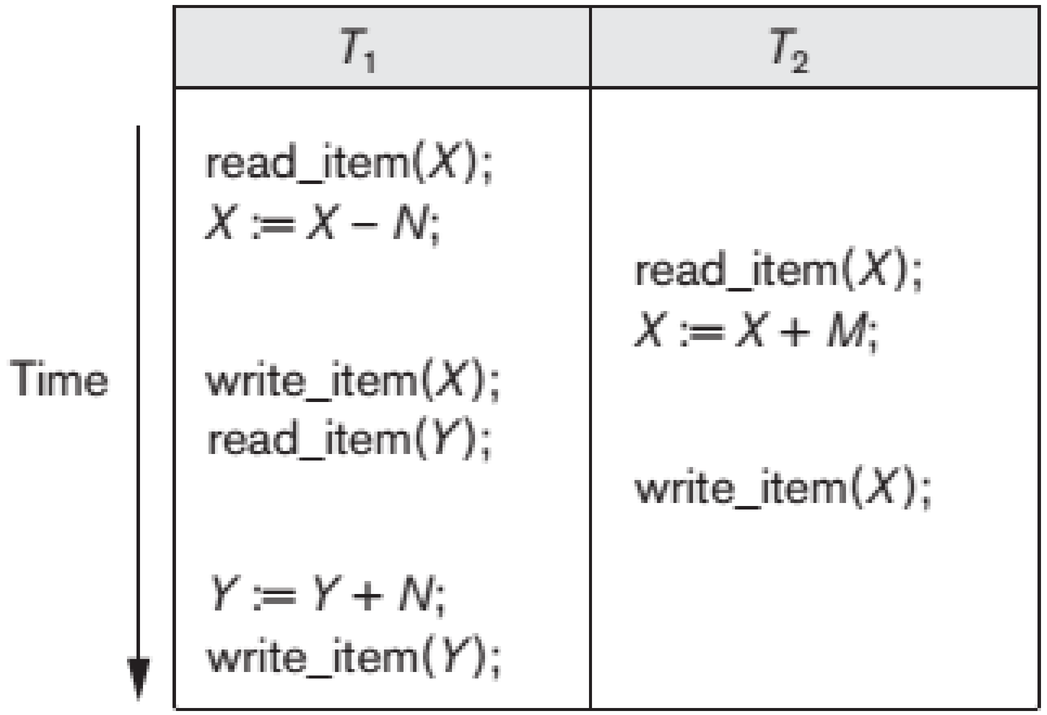
- Two operations in a schedule are said to conflict if they satisfy all three of the following conditions:

1. They belong to different transactions;
2. They access the same item X; and
3. At least one of the operations is a write_item(X).

Example

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

(a)



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);	

Contd...

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

  (1) they belong to different transactions;

  (2) they access the same item X; and

  (3) at least one of the operations is a write_item(X).

$$S_a: r_1(X); r_2(X); \overset{\downarrow}{w_1}(X); r_1(Y); w_2(\overset{\downarrow}{X}); w_1(Y);$$

- For example, in schedule S_a, the operations r1(X) and w2(X) conflict, as do the operations r2(X) and w1(X), and the operations w1(X) and w2(X).
- The operations r1(X) and r2(X) do not conflict, since they are both read operations; the operations w2(X) and w1(Y) do not conflict because they operate on distinct data items X and Y; and the operations r1(X) and w1(X) do not conflict because they belong to the same transaction.

read-write conflict

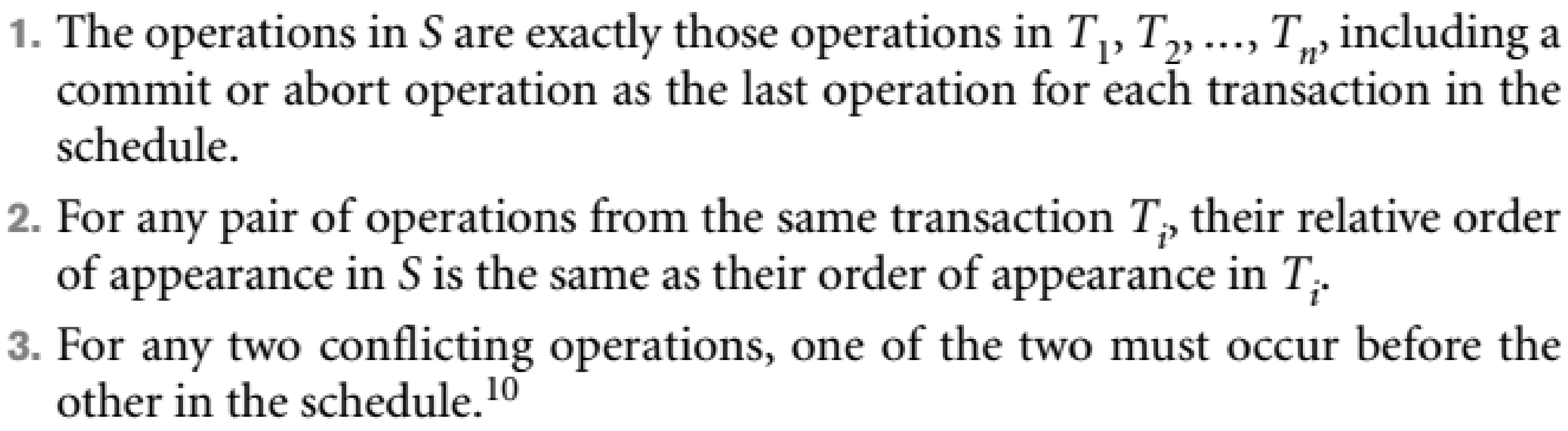
- Two operations are conflicting if changing their order can result in a different outcome.
- For example, if we change the order of the two operations r1(X);w2(X) to w2(X); r1(X), then the value of X that is read by transaction T1 changes, because in the second order the value of X is changed by w2(X) before it is read by *r*1(*X*), whereas in the first order the value is read before it is changed. This is called a **read-write conflict**.

write-write conflict

- In **write-write conflict** we change the order of two operations such as w1(X); w2(X) to w2(X); w1(X).
- For a write-write conflict, the last value of X will differ because in one case it is written by T2 and in the other case by T1.

Complete schedule

- A schedule S of n transactions T_1, T_2, …, T_n is said to be a **complete schedule** if the following conditions hold:



1. The operations in *S* are exactly those operations in $T_1, T_2, \ldots, T_n$, including a commit or abort operation as the last operation for each transaction in the schedule.
2. 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.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.[10]

Committed projection

- **Committed projection** C(S) of a schedule S, which includes only the operations in S that belong to committed transactions—that is, transactions T_i whose commit operation c_i is in S.

Characterizing Schedules Based on Recoverability

- For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be quite involved.
- In some cases, it is even not possible to recover correctly after a failure.
- Hence, it is important to characterize the types of schedules for which recovery is possible, as well as those for which recovery is relatively simple.
- The definition of **recoverable schedule** is as follows:
  - 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.
  - A transaction T reads from transaction T' in a schedule S if some item X is first written by T' and later read by T.
  - In addition, T' should not have been aborted before T reads item X, and there should be no transactions that write X after T' writes it and before T reads it.

Consider a schedule Sa'

Sa ' = r1(X);r2(X);w1(X);r1(Y);w2(X);c2; w1(Y);c1;

- Sa ' is **recoverable**, even though it suffers from the lost update problem.
- Consider the two (partial) schedules S_c and S_d.

Sc:	r1(X);w1(X);r2(X);r1(Y);w2(X);c2;a1;
Sd:	r1(X);w1(X);r2(X);r1(Y);w2(X);w1(Y);c1;c2;
Se:	r1(X);w1(X);r2(X);r1(Y);w2(X);w1(Y);a1;a2;

- **Sc** is **not recoverable** because T2 reads item X from T1 ,but T2 commits before T1 commits.
- The problem occurs if T1 aborts after the c2 operation in Sc , then the value of X that T2 read is no longer valid and T2 must be aborted after it is committed, leading to a schedule that is not recoverable.

- For the schedule to be recoverable, the c2 operation in Sc must be postponed until after T1 commits, as shown in Sd .
- If T1 aborts instead of committing, then T2 should also abort as shown in Se, because the value of X it read is no longer valid.
- In Se , aborting T2 is acceptable since it has not committed yet, which is not the case for the non-recoverable schedule Sc.

$$S_d: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$$
$$S_e: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$$

Cascading Rollback

Cascading rollback(or **cascading abort**): A phenomenon occurring in some recoverable schedules, where an uncommitted transaction has to be rolled back because it read an item from a transaction that failed.

- Example:
  - Se: r1(X); w1(X);r2(X); r1(Y); w2(X); w1(Y); a1; a2;
  - In schedule Se, where transaction T2 has to be rolled back because it read item X from T1, and T1 then aborted.

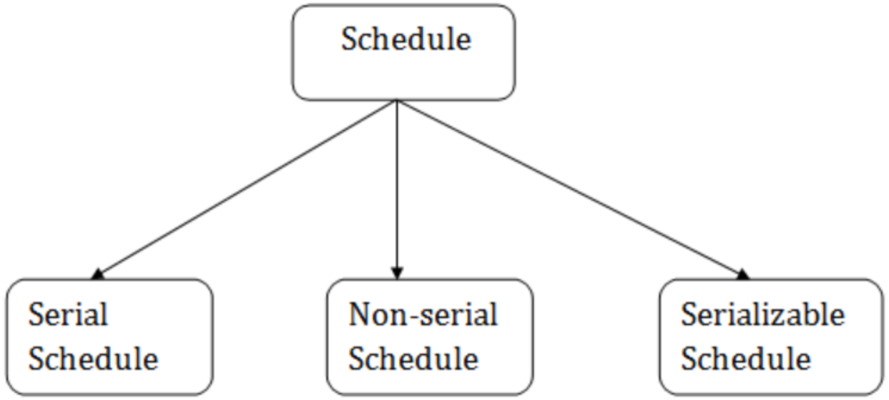
$$S_e: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$$

Cascadeless

- A schedule is said to be **cascade-less**, or to **avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions.
- In this case, all items read will not be discarded, so no cascading rollback will occur.

- There is a third, more restrictive type of schedule, called a **strict schedule**
- A **Strict Schedule**, is in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted).
- All strict schedules are cascadeless, and all cascadeless schedules are recoverable.

Serial, Non-serial, and Conflict-Serializable Schedules



1. Serial Schedule

The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.

- **For example:** Suppose there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible outcomes:
- Execute all the operations of T1 which was followed by all the operations of T2.
- Execute all the operations of T2 which was followed by all the operations of T1.

These two schedules are called **serial schedules.**

Non-serial Schedule

If interleaving of operations is allowed, then there will be non-serial schedule.
It contains many possible orders in which the system can execute the individual operations of the transactions.

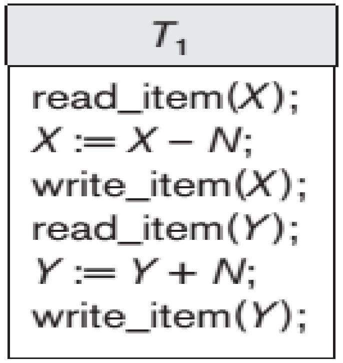
Serializable schedule

- The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
- It identifies which schedules are correct when executions of the transaction have interleaving of their operations.
- A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially.

Characterizing Schedules Based on Serializability

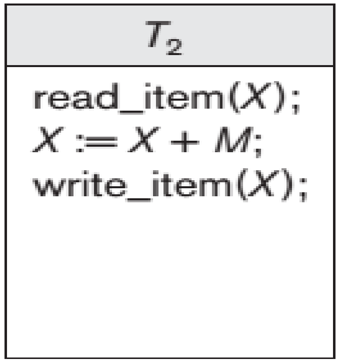
- Suppose that two users—for example, two airline reservations agents—submit to the DBMS, transactions T1 and T2 at approximately the same time.

(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);

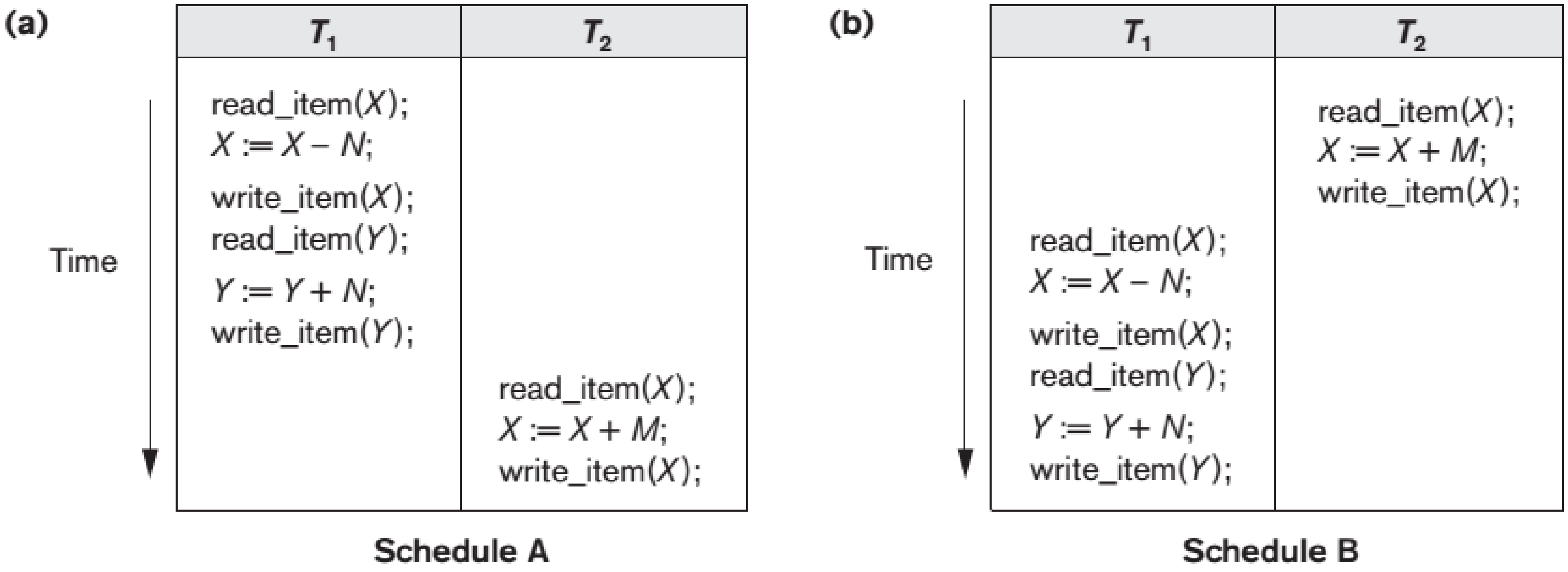
- If no interleaving of operations is permitted, there are only two possible outcomes:

  1. Execute all the operations of transaction T1 (in sequence) followed by all the operations of transaction T2 (in sequence).

  2. Execute all the operations of transaction T2 (in sequence) followed by all the operations of transaction T1 (in sequence).

- These two schedules are called **serial schedules.**

Figure 21.5
Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1.



(a)

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)

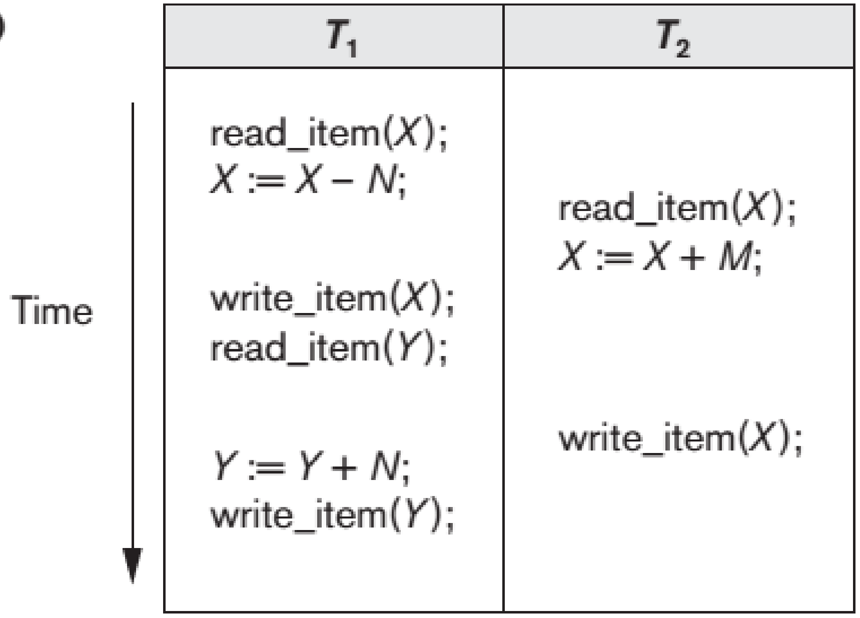
T_1	T_2
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule B

Contd...

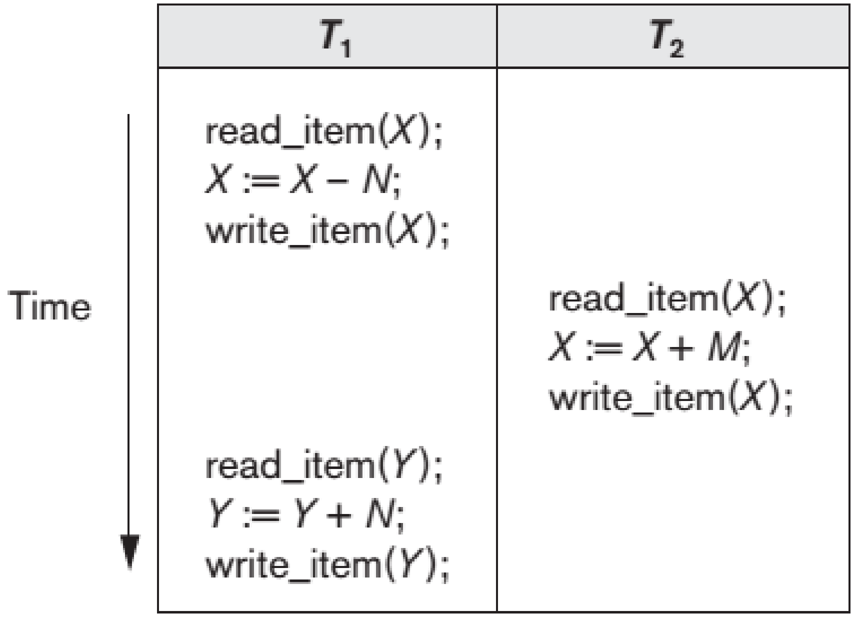
- If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions.

(c)



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

Serial, Non-serial, and Conflict-Serializable Schedules

- Formally, a schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule; otherwise, the schedule is called **non-serial**.
- Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction.
- No interleaving occurs in a serial schedule.

- The problem with serial schedules is that they limit concurrency by prohibiting interleaving of operations.
- In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time.
- Additionally, if some transaction T is quite long, the other transactions must wait for T to complete all its operations before starting.
- Hence, serial schedules are considered unacceptable in practice.

- To illustrate our discussion, consider the schedules in Figure 21.5, and assume that the initial values of database items are
  - X= 90 and Y= 90 and that N= 3 and M= 2.
- After executing transactions T1 and T2 , we would expect the database values to be
  - X= 89 and Y= 93, according to the meaning of the transactions.

Figure 21.5
Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

(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

(c)

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

Time ↓

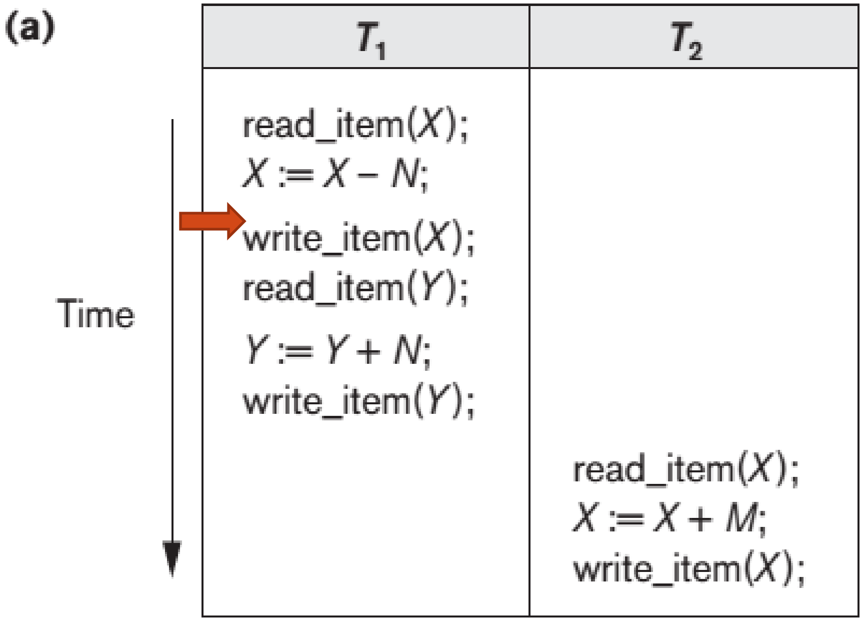
T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule D

Example

- Assume that the initial values of database items are X = 90 and Y = 90 and that N = 3 and M = 2.

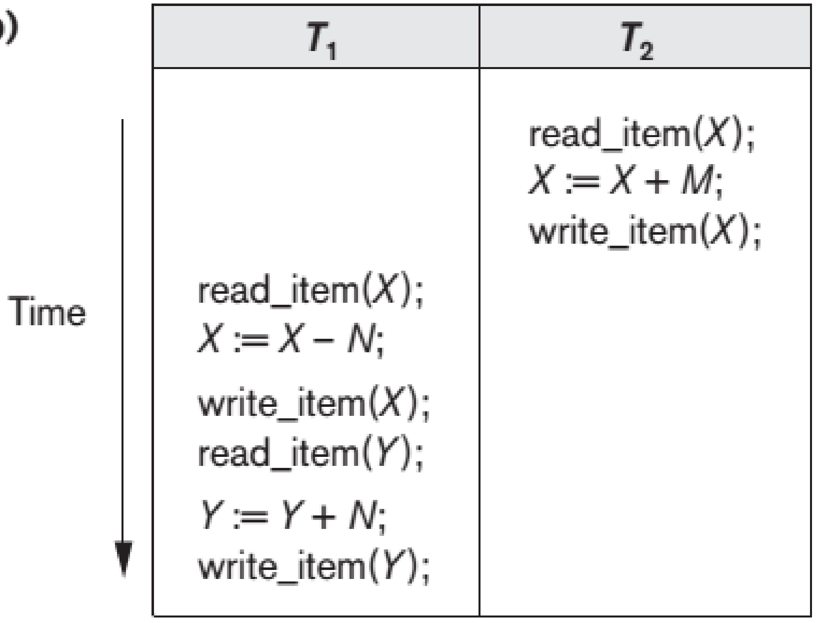
(a)



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)



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

- Finally , X=89, Y=93 for both A and B

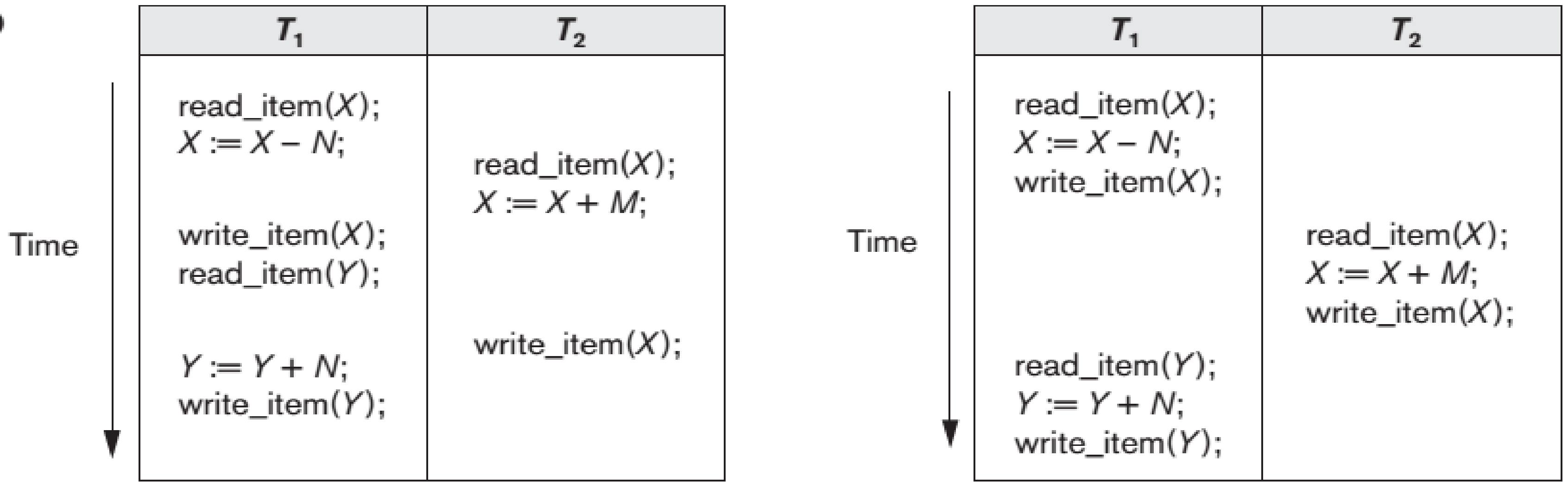
- Executing either of the serial schedules A or B gives the correct results(same results).

- Now consider the non-serial schedules C and D.
- Schedule C gives the
  - X= 92 andY= 93,
  - in which the X value is erroneous, whereas **schedule D gives the correct results**(ie X=89 and Y= 93, same as that of Serial schedules A and B).
- Schedule C gives an erroneous result because of the lost update problem.

Example

- Assume that the initial values of database items are X = 90 and Y = 90 and that N = 3 and M = 2.

(c)



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

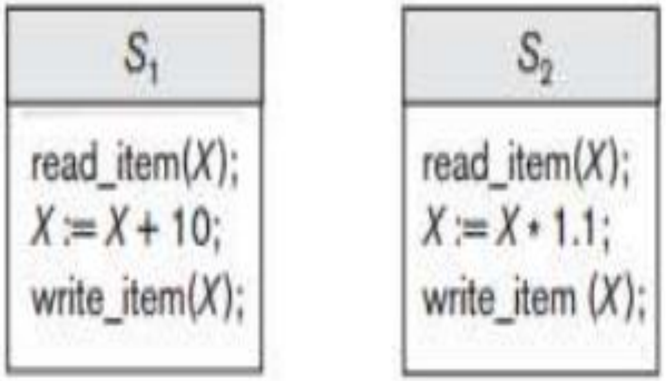
- Schedule C:X=92,Y=93, Schedule D: X=89,Y=93 which is Correct

Serializable schedule

- **Definition:** A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.
- There are several ways to define schedule equivalence.
- The simplest but least satisfactory definition involves comparing the effects of the schedules on the database.
- Two schedules are called **result equivalent** if they produce the same final state of the database.
- However, two different schedules may accidentally produce the same final state.
- For example, in Figure 21.6, schedules S1 and S2 will produce the same final database state if they execute on a database with an initial value of X= 100; however, for other initial values of X, the schedules are not result equivalent.

- Additionally, these schedules execute different transactions, so they definitely should not be considered equivalent.
- Hence, result equivalence alone cannot be used to define equivalence of schedules.

Figure 21.6
Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

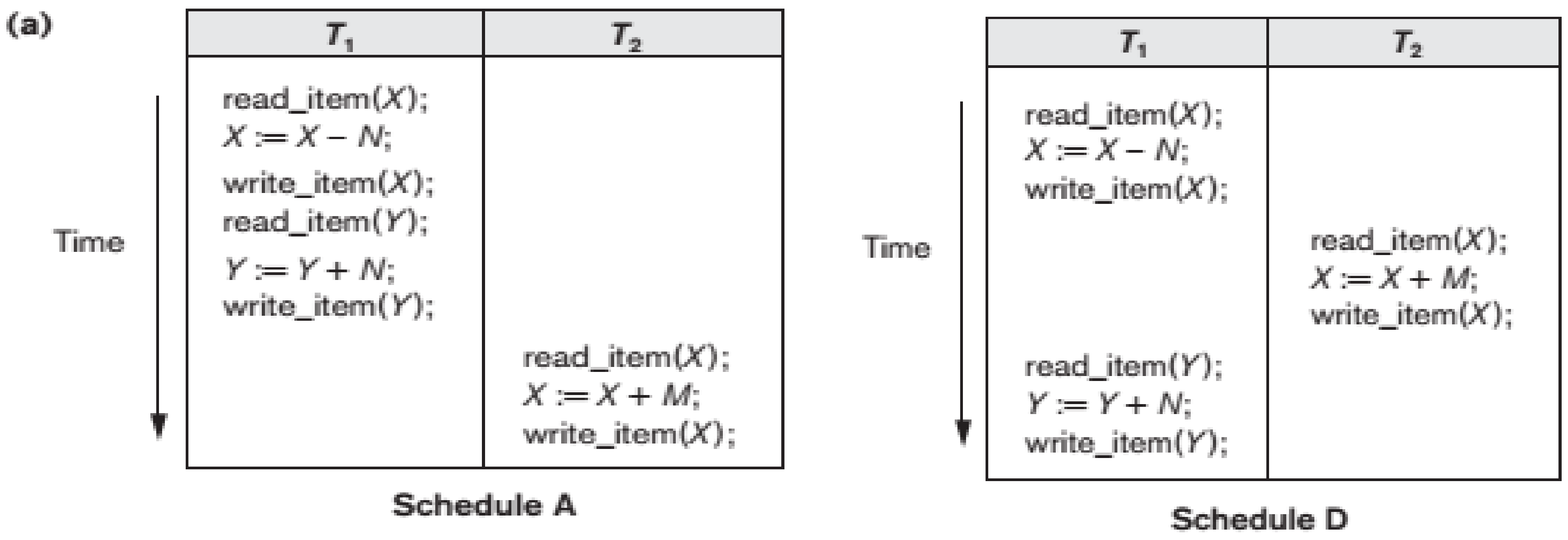


S_1	S_2

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

- For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules in the same order.
- Two definitions of equivalence of schedules are generally used: **Conflict equivalence and View equivalence.**

- The definition of **conflict equivalence** of schedules is as follows:
  - Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
  - Two operations in a schedule are said to conflict if they belong to different transactions, access the same database item, and either both are write_item operations or one is a write_item and the other a read_item.
- A schedule S to be **conflict serializable** if it is (conflict) equivalent to some serial schedule S.


(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

Time

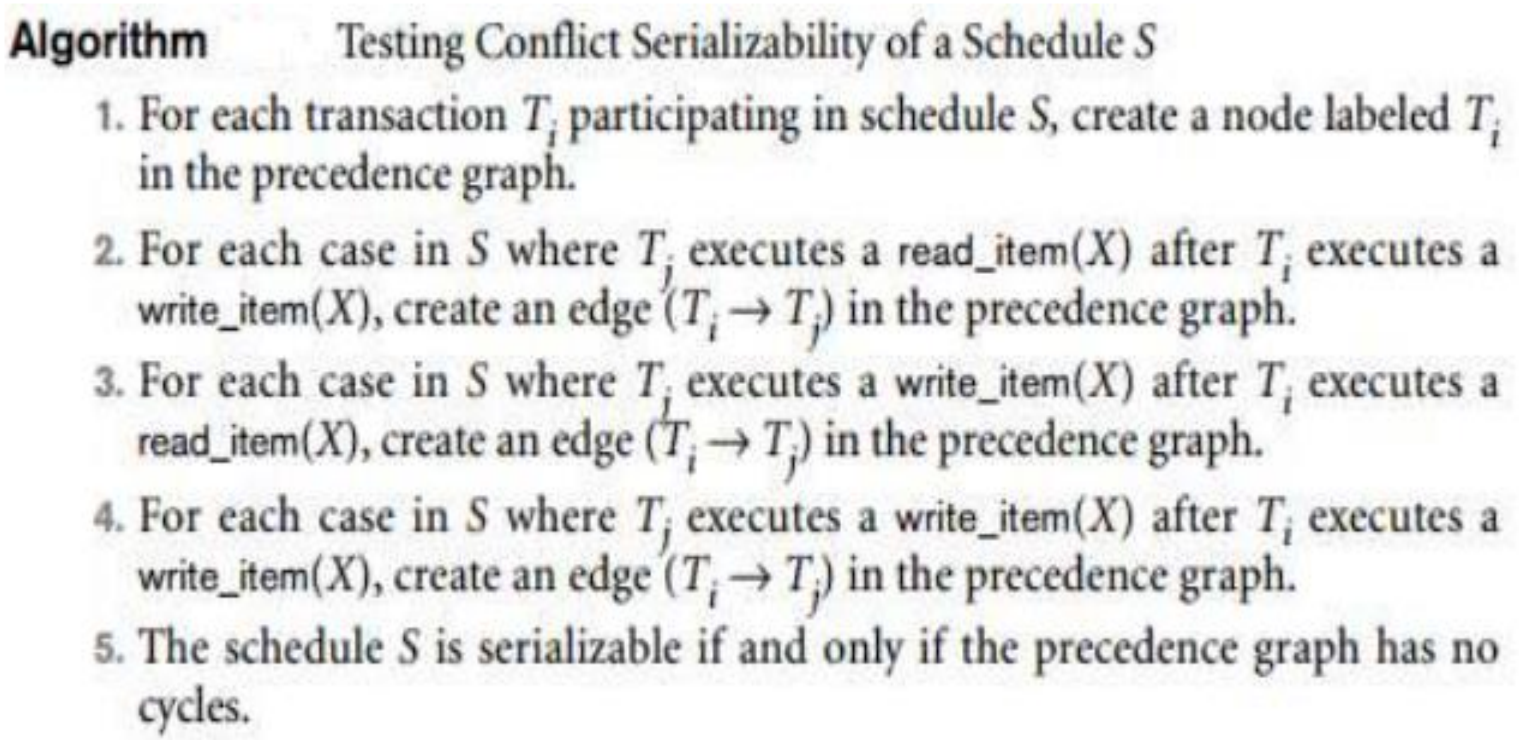
T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule D

- Schedule D is conflict equivalent to the serial schedule A.
- In both schedules, the read_item(X) of T2 reads the value of X written by T1, while the other read_item operations read the database values from the initial database state.
- Because A is a serial schedule and schedule D is equivalent to A, D is a conflict serializable schedule.

Testing for Conflict Serializability of a Schedule

- There is a simple algorithm for determining whether a particular schedule is conflict serializable or not.



Algorithm Testing Conflict Serializability of a Schedule S

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

- The algorithm looks at only the read item and write item operations in a schedule to construct a precedence graph(or serialization graph),
  - which is a directed graph G= (N,E) that consists of
    - A set of nodes N= {T1,T2, ...,Tn} and
    - A set of directed edges E= {e1,e2, ..., em }.

- The precedence graph is constructed as described in Algorithm.
- If there is a cycle in the precedence graph, schedule S is not (conflict) serializable; if there is no cycle, S is serializable.

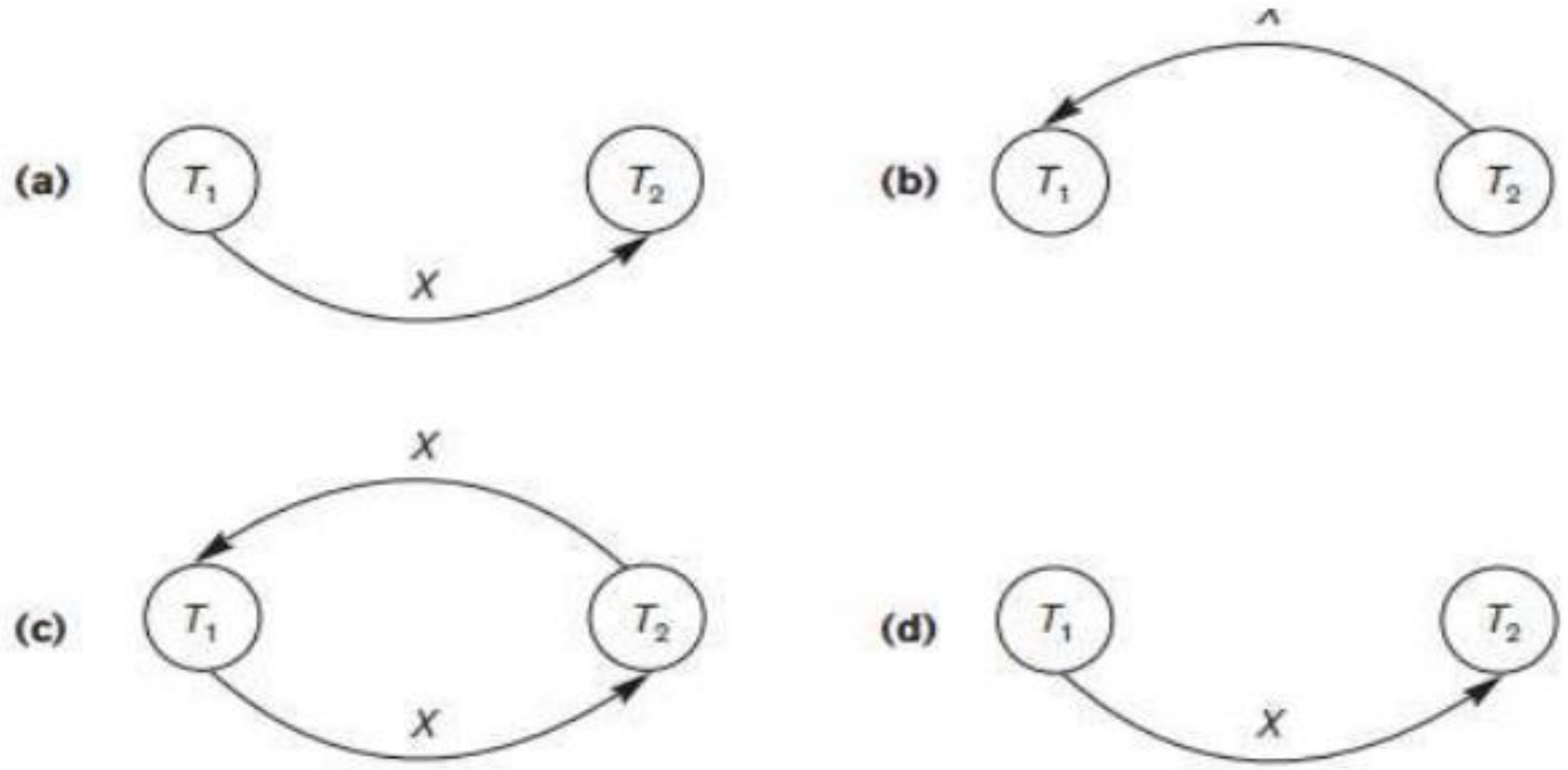


Figure
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

Concurrency control Methods

- 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

- Several types of locks are used in concurrency control.

- To introduce locking concepts gradually, first we discuss

➢ **binary locks**, which are simple, but are also too restrictive for database concurrency control purposes, and so are not used in practice.

➢ **Shared/exclusive locks** also known as **read/write locks** which provide more general locking capabilities and are used in practical database locking schemes.

➢ **Two-phase locking** : Has a 'growing' and a 'shrinking' phase

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, and the lock value is changed to 1

- We refer to the current value (or state) of the lock associated with item X as lock(X).

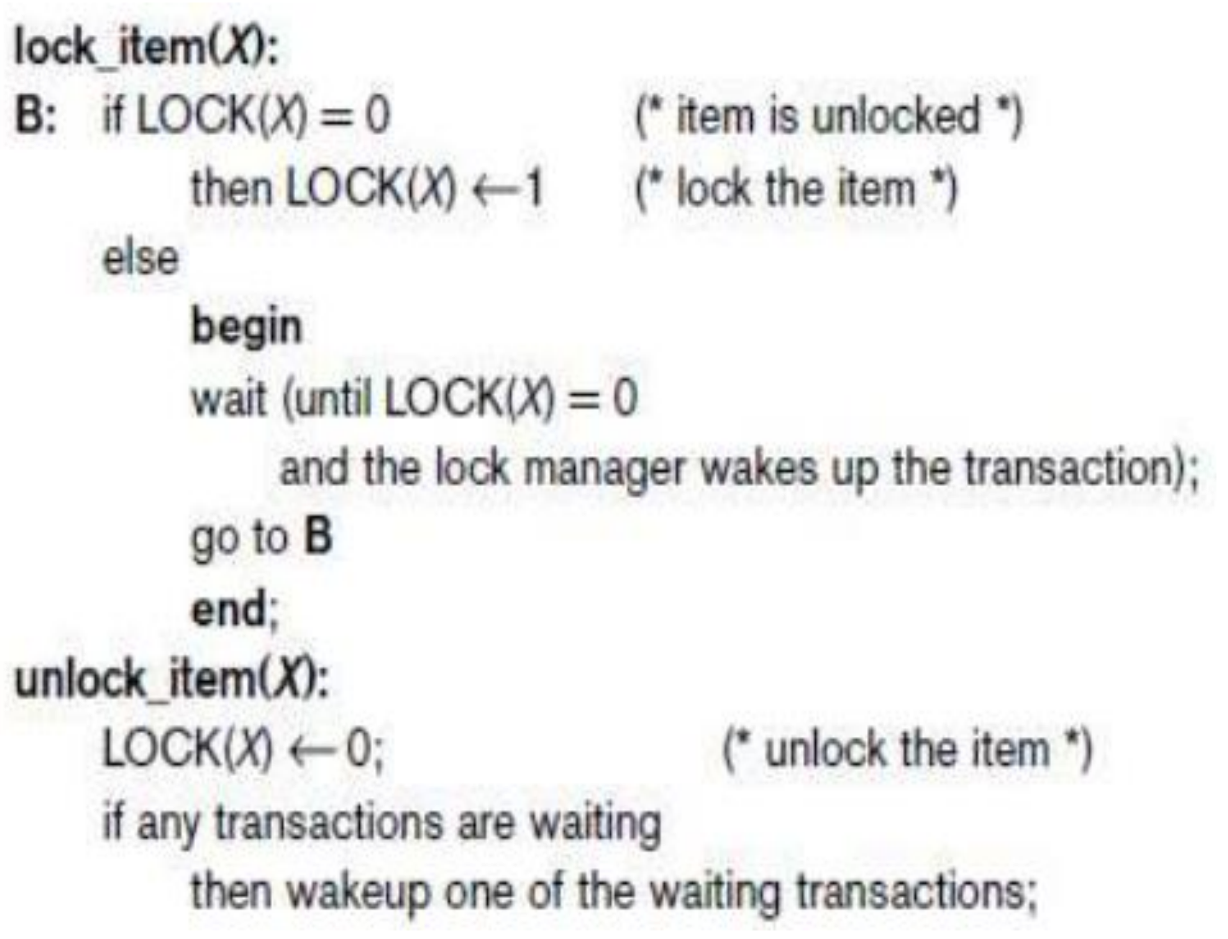
- Two operations, **lock_item** and **unlock_item**, are used with binary locking.

- 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.

- When the transaction is through using the item, it issues an unlock_item(X) operation, which sets LOCK(X) back to 0 (unlocks the item) so that X may be accessed by other transactions.

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

❑ A description of the lock_item(X) and unlock_item(X) operations is shown below:



```
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;
```

- It is quite simple to implement a binary lock; all that is needed is a binary-valued variable, **LOCK**, associated with each data item X in the database.

- In its simplest form, each lock also includes 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**, which could be organized as a **hash file** on the item name.

- Items not in the **lock table** are considered to be **unlocked**.

- The DBMS has a **lock manager subsystem** to keep track of and control access to locks.

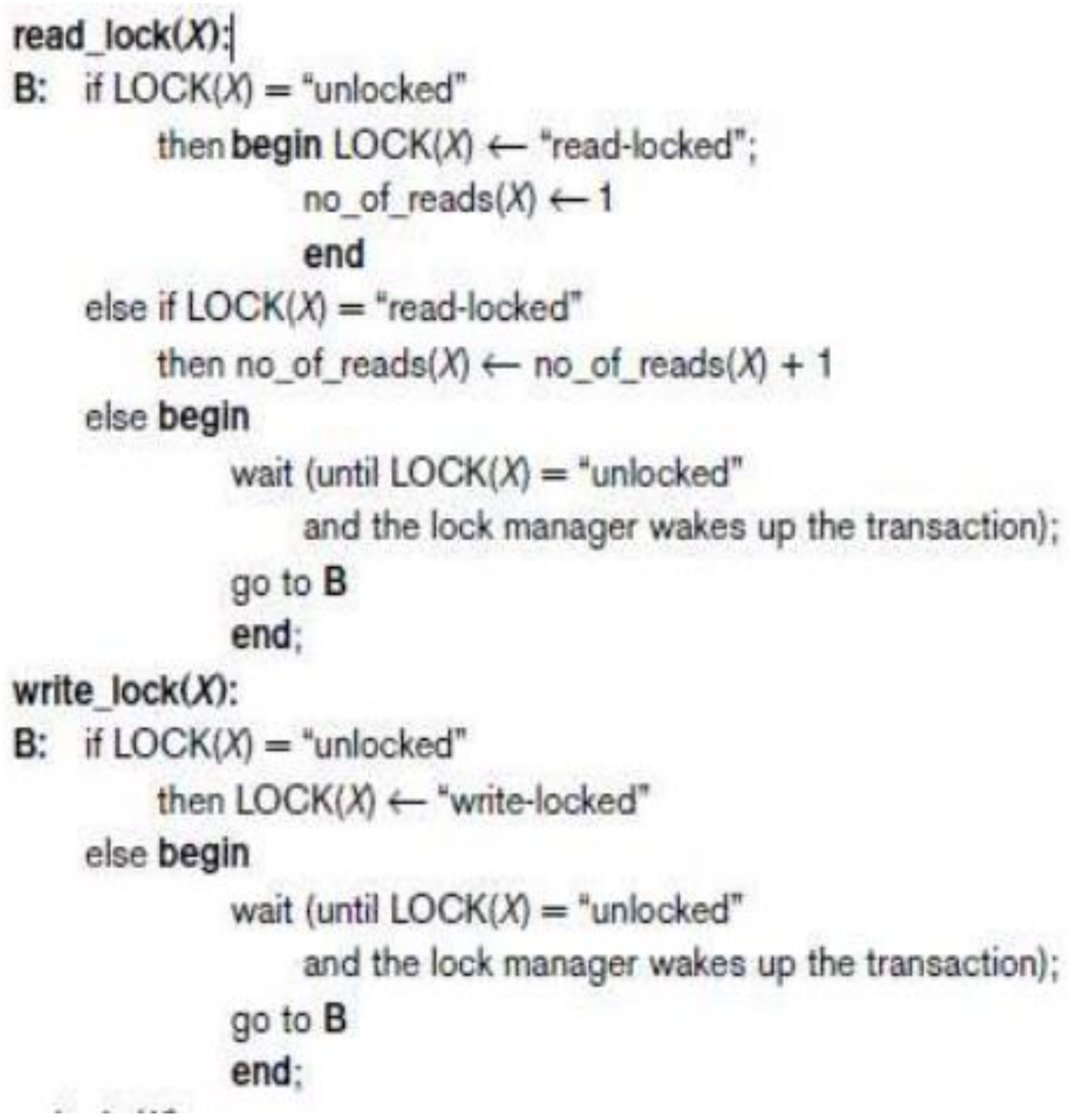
Shared/Exclusive (or Read/Write) Locks:

- The binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item.
- We should allow several transactions to access the same item X if they all access X for reading purposes only(ie not conflicting).
- 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.

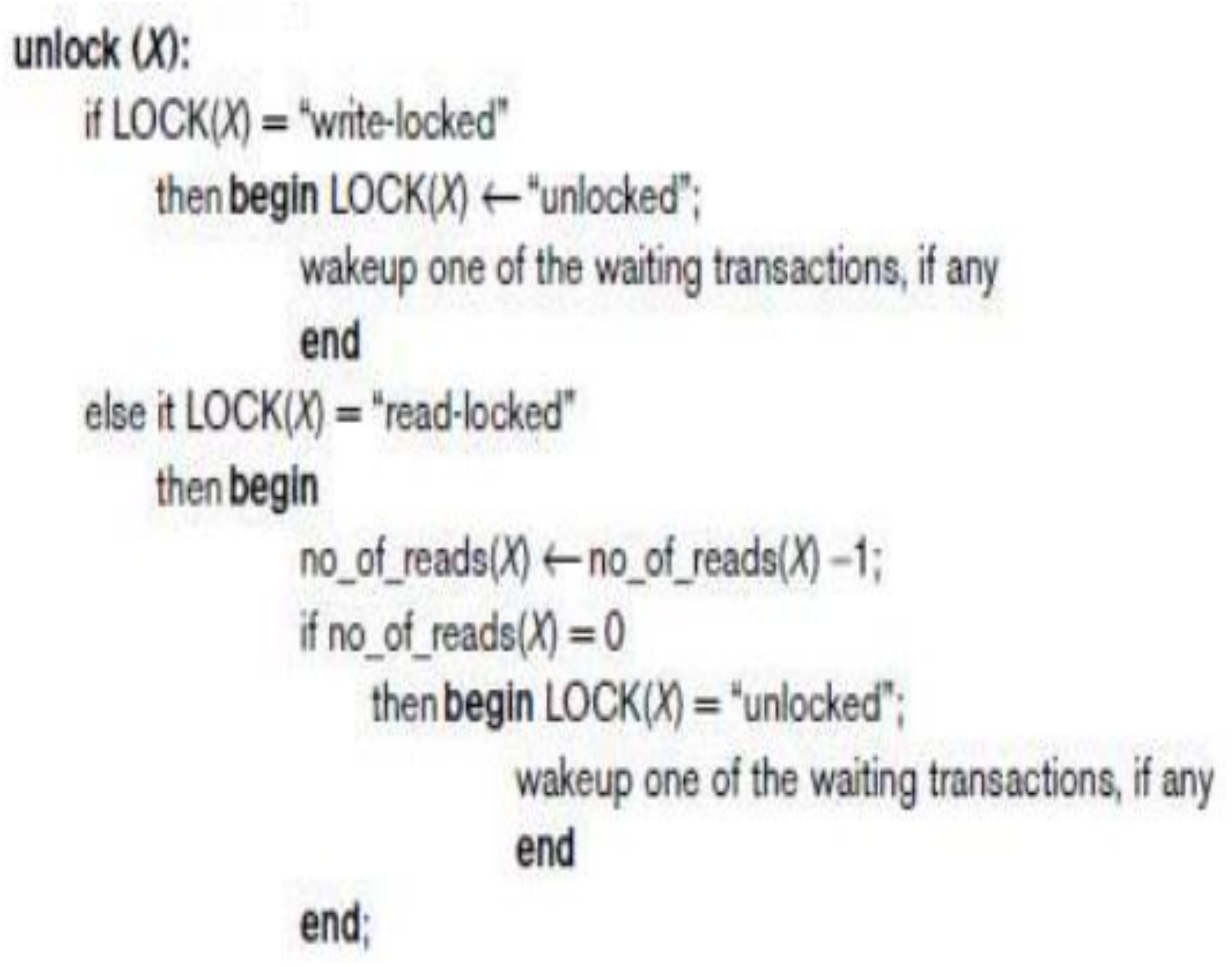
- In this scheme called **shared/exclusive** or **read/write locks**
- There are three locking operations: → **read_lock(X), write_lock(X), and unlock(X).**
- A lock associated with an item X, LOCK(X), now has three possible states → **read-locked, write-locked, or unlocked**.
- A read-locked item is also called share-locked because other transactions are allowed to read the item, whereas a write-locked item is called exclusive-locked because a single transaction exclusively holds the lock on the item.

- 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.
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.
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.
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.



```
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 relax conditions 4 and 5 in the preceding list in order to allow lock conversion
  - That is, <u>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</u>.

- For example, it is possible for a transaction T to issue a read_lock(X) and then later 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 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 write_lock(X) and then later to **downgrade** the lock by issuing a read_lock(X) operation.

Two-phase locking

- 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:
- An **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released.
- A **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired.

Guaranteeing Serializability by Two-Phase Locking

- If **lock conversion** is allowed,
  - Then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and
  - Downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.
- Hence, a read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase.

Contd…

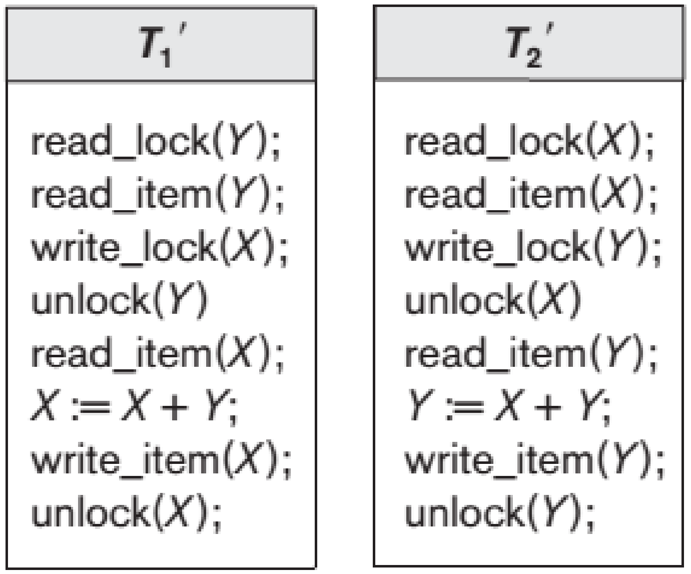
- Transactions T1 and T2 do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in T1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T2.

(a)

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);

Contd....

- If we enforce two-phase locking, the transactions can be rewritten as T_1’ and T_2’.



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 22.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

Contd...

- Two-phase locking limits the amount of concurrency that can occur in a schedule because a transaction T may not be able to release an item X after it is through using it if T must lock an additional item Y later;
- 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;
- Although the two-phase locking protocol guarantees serializability, it does not permit all possible serializable schedules.

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

- There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**.
- A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses before the transaction begins execution, by **predeclaring** its read-set and write-set.
- The **read-set** of a transaction is the set of all items that the transaction reads, and the **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.

Contd...

- **Conservative 2PL** is a deadlock-free protocol.
- However, it is difficult to use in practice because of the need to predeclare the read-set and writeset, which is not possible in many situations.

Strict 2PL

- **Strict 2PL** guarantees strict schedules.
- In this variation, 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.

Rigorous 2PL

- A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules.
- In this variation, 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.

Contd…

- 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;
- 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.

Pitfalls of lock-based protocols

- The potential for **deadlock** exists in most locking protocols.
- **Starvation** is also possible if concurrency control manager is badly designed.

Use of locks in strict two-phase locking

1. When an operation accesses a data item within a transaction:
   a) If the data item is **not** already locked, it is locked and the operation proceeds.
   b) If the data item has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   c) If the data item has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   d) If the object has already been locked in the same transaction, the lock will be promoted/converted if necessary and the operation proceeds

2. When a transaction is committed or aborted, the server unlocks all data item it locked for the transaction

Log based recovery

Using the Log to Redo and Undo Transactions

- The recovery scheme uses two recovery procedures. Both these procedures make use of the log to find the set of data items updated by each transaction Ti, and their respective old and new values.

  redo(Ti)

  - Sets the value of all data items updated by transaction Ti to the new values.
  - The order in which updates are carried out by redo is important; when recovering from a system crash, if updates to a particular data item are applied in an order different from the order in which they were applied originally, the final state of that data item will have a wrong value.
  - Most recovery algorithms do not perform redo of each transaction separately; instead they perform a single scan of the log, during which redo actions are performed for each log record as it is encountered.
  - This approach ensures the order of updates is preserved, and is more efficient since the log needs to be read only once overall, instead of once per transaction.

undo(Ti)

- Restores the value of all data items updated by transaction Ti to the old values. In the recovery scheme :
- The undo operation not only restores the data items to their old value, but also writes log records to record the updates performed as part of the undo process.
- As with the redo procedure, the order in which undo operations are performed is important
- When the undo operation for transaction Ti completes, it writes a log record, indicating that the undo has completed.

Database modification

The database can be modified using two approaches :

1) **Deferred database modification** – All logs are written on to the stable storage and the database is updated when a transaction commits.
2) **Immediate database modification** – Each log follows an actual database modification. That is, the database is modified immediately after every operation.

Deferred database modification

- The **deferred update** techniques do not physically update the database on disk **until *after* a transaction reaches its commit point**; then the updates are recorded in the database.
- **Before reaching commit**, all transaction updates are **recorded in** the local transaction workspace or in the **main memory buffers** that the DBMS maintains (the DBMS main memory cache).
- Before commit, the updates are recorded persistently in the log, and then after commit, the updates are written to the database on disk.

Contd…

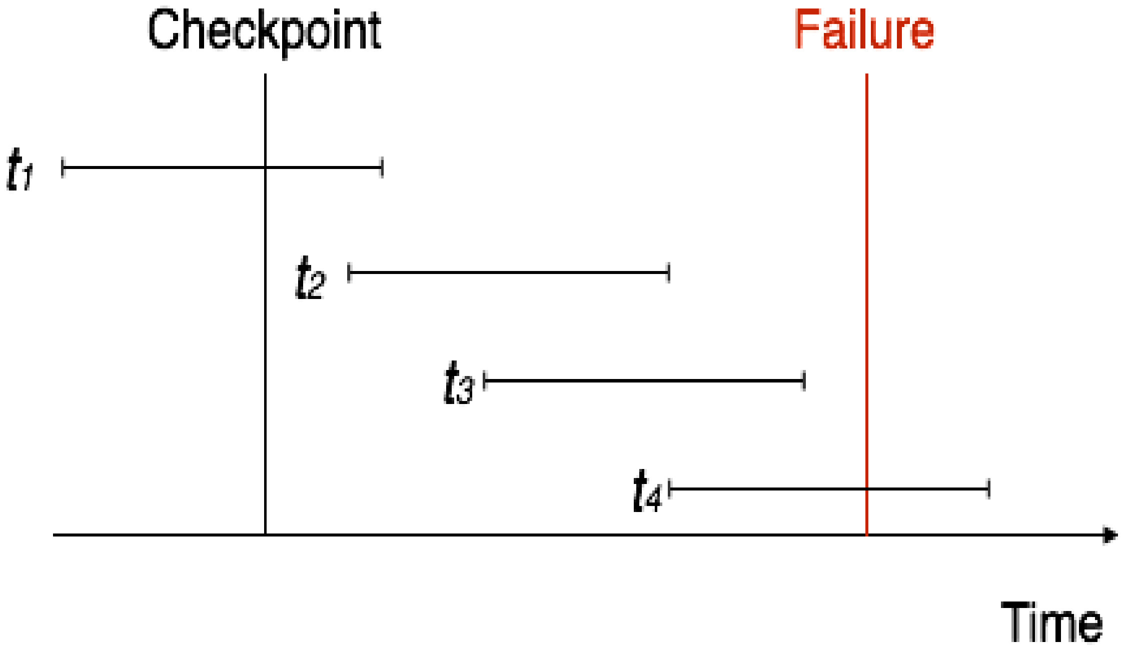
- If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed.
- It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been recorded in the database on disk.
- Hence, **deferred update** is also known as the **NO-UNDO/REDO algorithm**.

Check-pointing

- When a system crash occurs, must consult the log to determine those transactions that need to be redone and those that need to be undone.
- In principle, need to search the entire log to determine this information.
- Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all.
- Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk.
- Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.(like a bookmark)

Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −



- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an **undo-list** and a **redo-list**.
- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.
- All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

- **For example:** In the log file, transaction t2 and t3 will have <tn, Start> and <tn, Commit>. The t1 transaction will have only <tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts t1, t2 and t3 transaction into redo list.
- The transaction is put into undo state if the recovery system sees a log with <tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
- **For example:** Transaction t4 will have <tn, Start>. So t4 will be put into undo list since this transaction is not yet complete and failed.