UNIT – V chapter 1

Transaction:

A **transaction** is a **unit** of program execution that accesses and possibly updates various data items.

Usually, a transaction is initiated by a user program written in a high-level data-manipulation language (typically SQL), or programming language with embedded database accesses in JDBC or ODBC.

A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**.

The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

A transaction is action, or series of actions, carried out by user or application, which accesses or updates contents of database.

It Transforms database from one consistent state to another, although consistency may be violated during transaction.

The concept of transaction provides a mechanism for describing logical units of database processing.

Transaction processing systems are systems with large databases and hundreds of concurrent users that are executing database transactions.

Examples of such systems include systems for reservations, banking, stock markets, super markets and other similar systems.

They require high availability and fast response time for hundreds of concurrent users.

Single User Vs. Multi User Systems:

- A DBMS is a single user if at most one user at a time can use the system.
- A DBMS is a multi user if many users can use the system and hence access the database concurrently.

- Multiple users can access databases and use the computer systems simultaneously because of the concept of Multiprogramming.
- Multiprogramming allows the computer to execute multiple programs or processes at the same time.
- If only a single central processing unit(CPU) exists, it can actually executes at most one process at a time.
- However multiprogramming operating systems executes some actions from one process then suspend that process and execute some actions of the next process,and so on.
- 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 process is actually interleaved as illustrated in the following figure, which shows two processes A and B executing concurrently in an interleaved fashion.

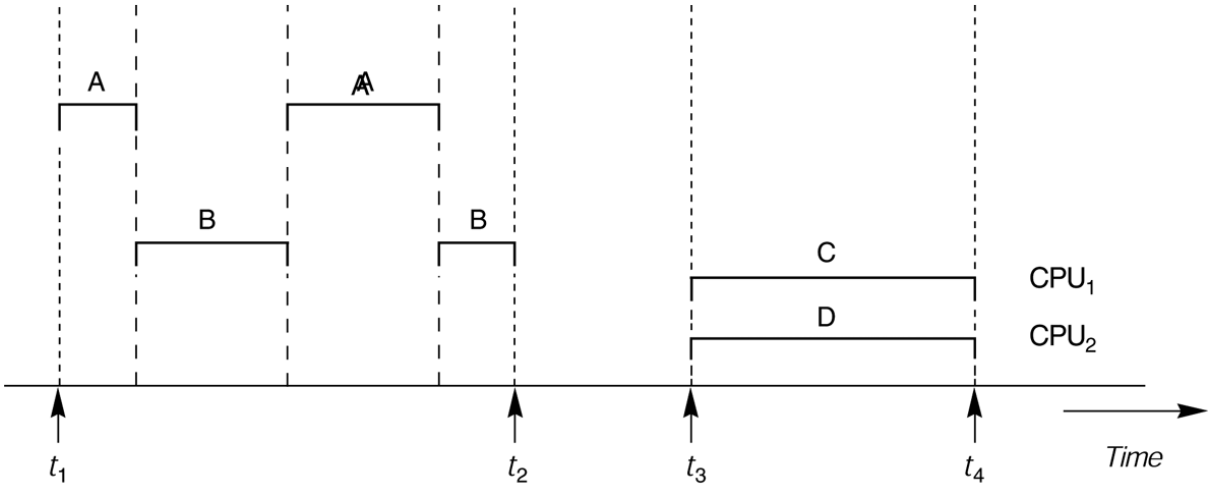


Fig. 4.1 Interleaved processing Vs. Parallel Processing of concurrent transactions.

Interleaving also prevents the long process from delaying other processes.

If the computer system has multiple hardware processors(CPUs), parallel processing of multiple processing is possible as illustrated the process C and D in the figure.

A transaction Can have one of two outcomes:

Success - transaction *commits* and database reaches a new consistent state.
Failure - transaction *aborts*, and database must be restored to consistent state before it started.. Such a transaction is *rolled back* or *undone*.
Committed transaction cannot be aborted.
Aborted transaction that is rolled back can be restarted later.

Transactions, Read and Write Operations, and DBMS Buffers

A **transaction** is an executing program that forms a logical unit of database processing.
A **transaction** includes one or more database access operations such as insertion, deletion, modification, or retrieval operations.
The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.
One way of specifying the transaction boundaries is by specifying explicit begin transaction and end transaction statements in an application program; in this case, all database access operations between the two are considered as forming one transaction.
A single application program may contain more than one transaction if it contains several transactions boundaries.
read-only transaction:
If the database operations in a transaction do not update the database but
Only retrieve data, the transaction is called a **read-only transaction.**
A database is basically represented as a collection of named data items.
Granularity:
The size of a data item is called its granularity, and it can be a field of some record in the database, or it may be a larger unit such as a record or even a whole disk block,
Basic database access operations:

read_item(X): Reads a database item named X into a program variable. To simplify our notation, we assume that *the program variable is also named* X.
write_item(X): Writes the value of program variable X into the database item named X.

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
3. Copy item X from the buffer to the program variable 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
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

Step 4 is the one that actually updates the database on disk. In some cases the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer.
Usually, the decision about when to store back a modified disk block that is in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system.
The DBMS will generally maintain a number of buffers in main memory that hold database disk blocks containing the database items being processed
A transaction includes read_item and wri te_item operations to access and update the database. Figure 4.2 shows examples of two very simple transactions.
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 the transaction writes.

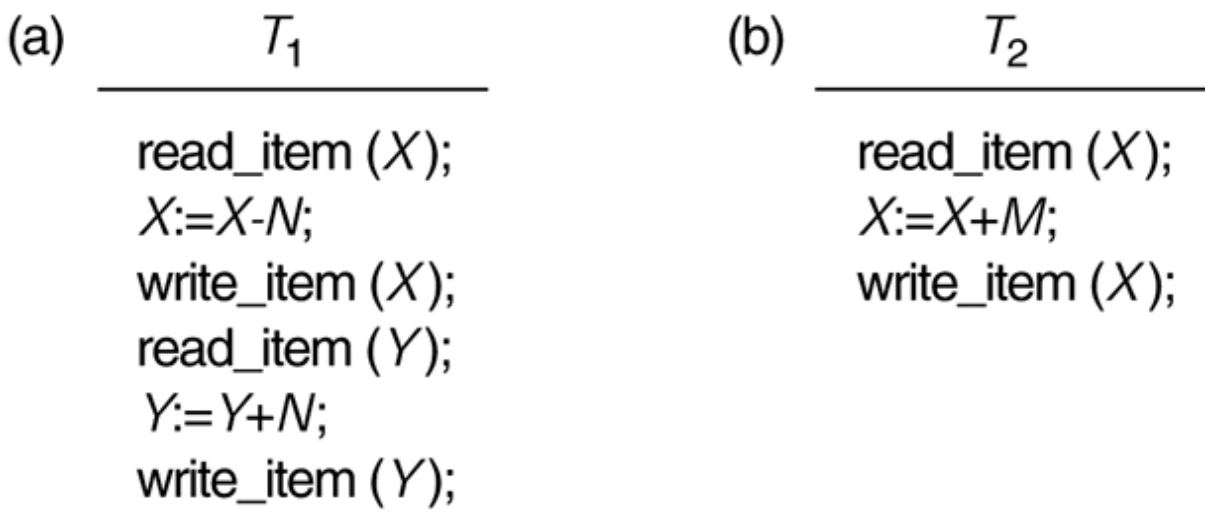
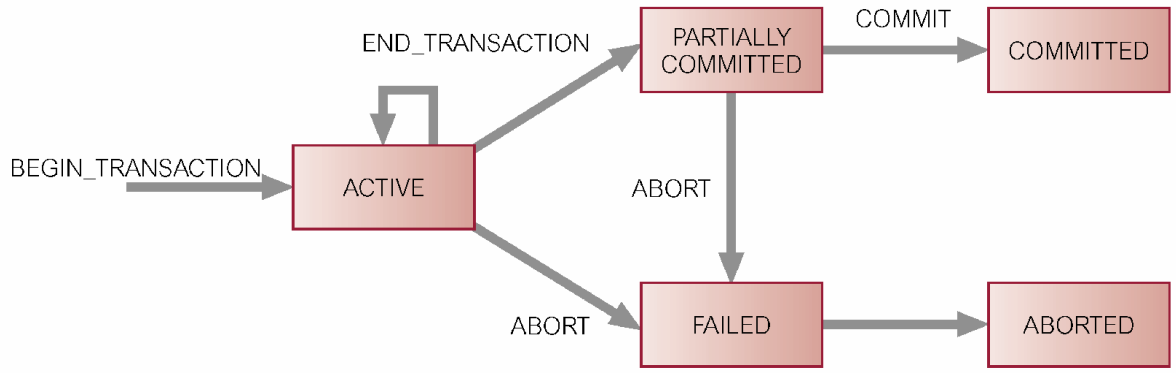


Fig 4.2: Two sample transactions. (a) Transaction T_1 . (b) Transaction T_2

Transaction States or State Transition Diagram and Additional Operations



A transaction is an atomic unit of work that is either completed 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.

Hence, the recovery manager keeps track of the following operations:

BEGIN_TRANSACTION: This marks the beginning of transaction execution.

READ DR 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 of transaction execution. However, at this point it may be necessary to check whether the changes introduced by

the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.

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.*

Figure 17.4 shows a state transition diagram that describes how a transaction moves through its execution states.

Active state: A transaction goes into an active state immediately after it , where it can issue READ and WRITE operations.

Partially committed state: 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

Committed state:

Once check in partially committed state is successful, the transaction is said to have reached its commit point and enters the committed state.

Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.

Failed 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 transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.

Terminated 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.

ACID Properties or DESIRABLE PROPERTIES OF TRANSACTIONS

In DBMS **ACID** (*Atomicity*, *Consistency*, *Isolation*, *Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

Jim Gray defined these properties of a reliable transaction system in the late 1970s and developed technologies to achieve them automatically

Atomicity:

Atomicity refers to the ability of the DBMS to guarantee that either all of the operations of a transaction are performed or none of them are. Database modifications must follow an all or nothing rule. Each transaction is said to be atomic if when one part of the transaction fails, the entire transaction fails.

The atomicity property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity.

If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.

Consistency:

The consistency property ensures that the database remains in a consistent state before the start of the transaction and after the transaction is over (whether successful or not).

The preservation of consistency is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints.

A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints that should hold on the database. A

database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction,

Isolation:

The isolation portion of the ACID Properties is needed when there are concurrent transactions. Concurrent transactions are transactions that occur at the same time, such as shared multiple users accessing shared objects.
Although multiple transactions may execute concurrently, each transaction must be independent of other concurrently executing transactions. A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Durability:

Maintaining updates of committed transactions is critical. These updates must never be lost. The ACID property of durability addresses this need. Durability refers to the ability of the system to recover committed transaction updates if either the system or the storage media fails. Features to consider for durability:

- recovery to the most recent successful commit after a database software failure
- recovery to the most recent successful commit after an application software failure
- recovery to the most recent successful commit after a CPU failure

- recovery to the most recent successful backup after a disk failure
- recovery to the most recent successful commit after a data disk failure

The System Log

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.

This information may be needed to permit recovery from failures.

The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.

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

We now list the types of entries-called log records-that are written to the log and the action each performs.

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

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.

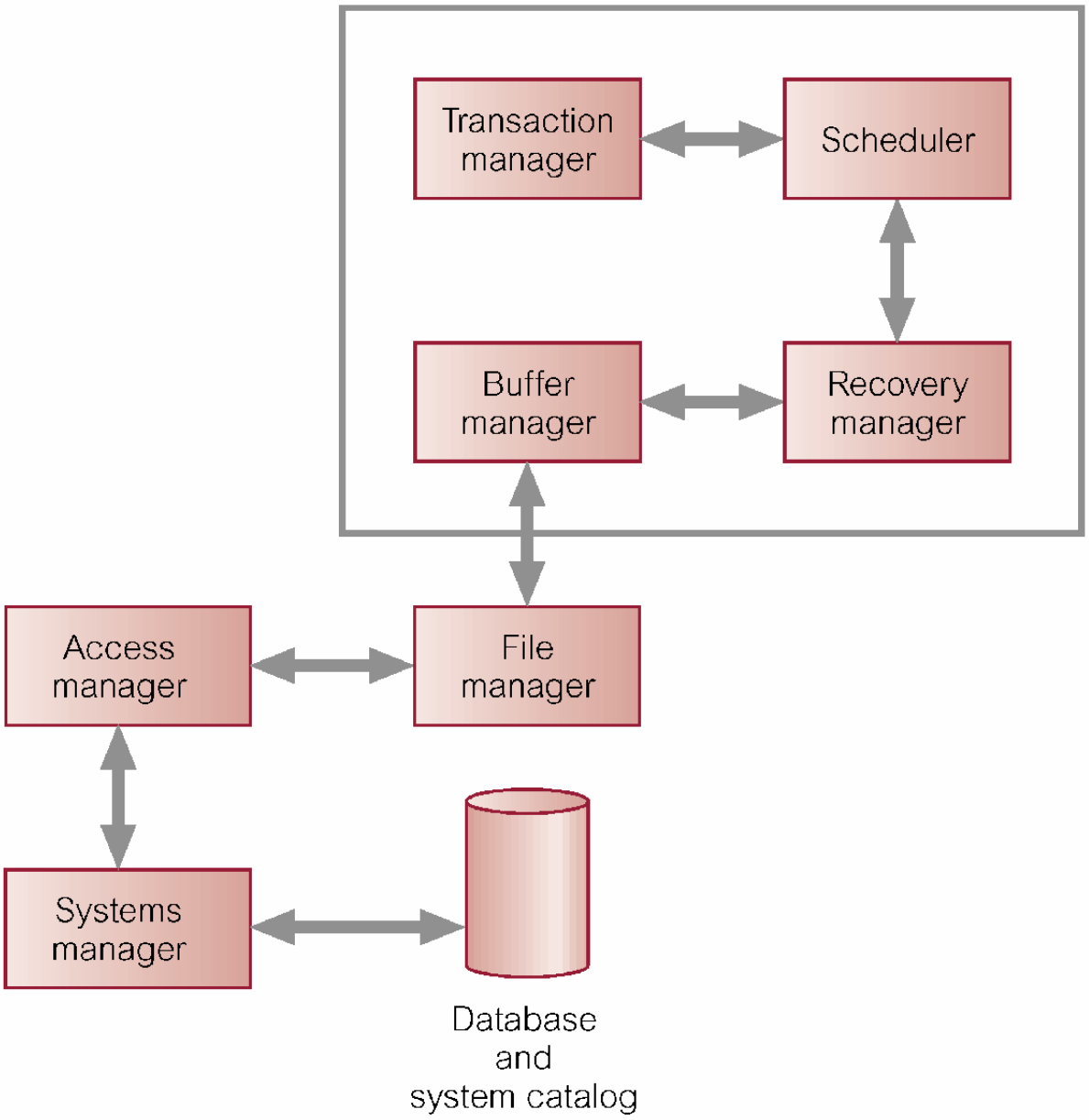
5. [abort.T]: Indicates that transaction T has been aborted.

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, the transaction is said to be **committed,** and its effect is assumed to be *permanently recorded* in the database.

- The transaction then writes a commit record [commit,T] into the log.
- If a system failure occurs, we 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. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone*from the log records.
- The log file must be kept on disk. Updating a disk file involves copying the appropriate block of the file from disk to a buffer in main memory, updating the buffer in main memory, and copying the buffer to disk.
- It is common to keep one or more blocks of the log file in main memory buffers 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 block.
- 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 file before committing a transaction



Concurrency control:

Processes of managing simultaneous operations on the database without having them interfere with one another.

- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.

- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Why Concurrency Control Is Needed

Concurrency control and recovery mechanisms are mainly concerned with the database access commands in a transaction. Transactions submitted by the various users mayexecute 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. Several problems can occur when concurrent transactions execute in an uncontrolled manner.

These problems are

1. Lost update problem
2. The temporary update or Dirty Read Problem.
3. Incorrect summary 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.

Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in the figure a, then the final value of 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.

For example, if X = 80 at the start, N =5 and M = 4 the final result should be X =79; but in the interleaving of operations shown in Figure a, it is X = 84 because the update in T1 that removed the five from X was *lost.*

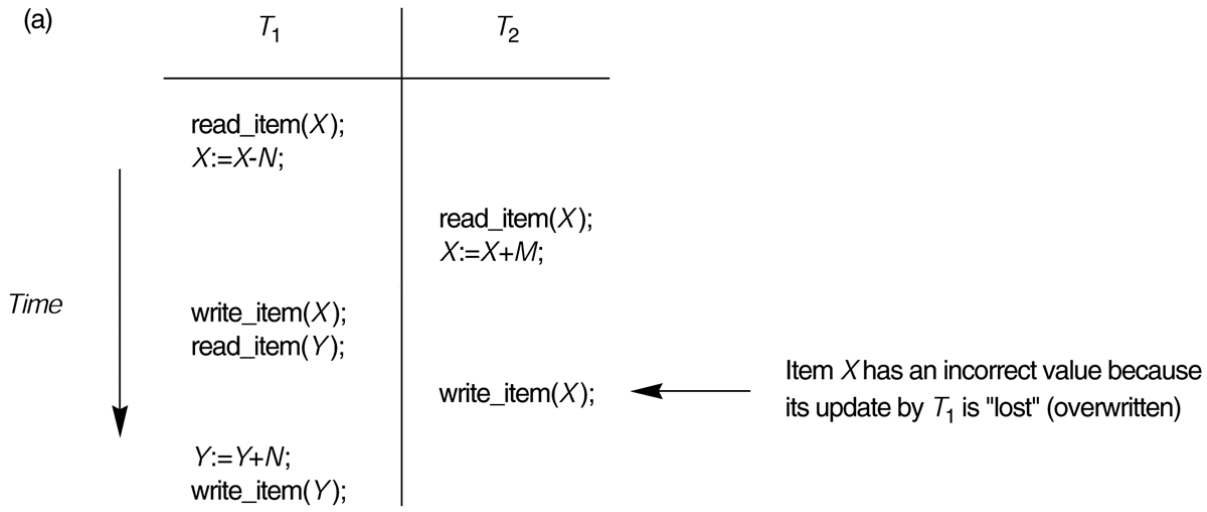
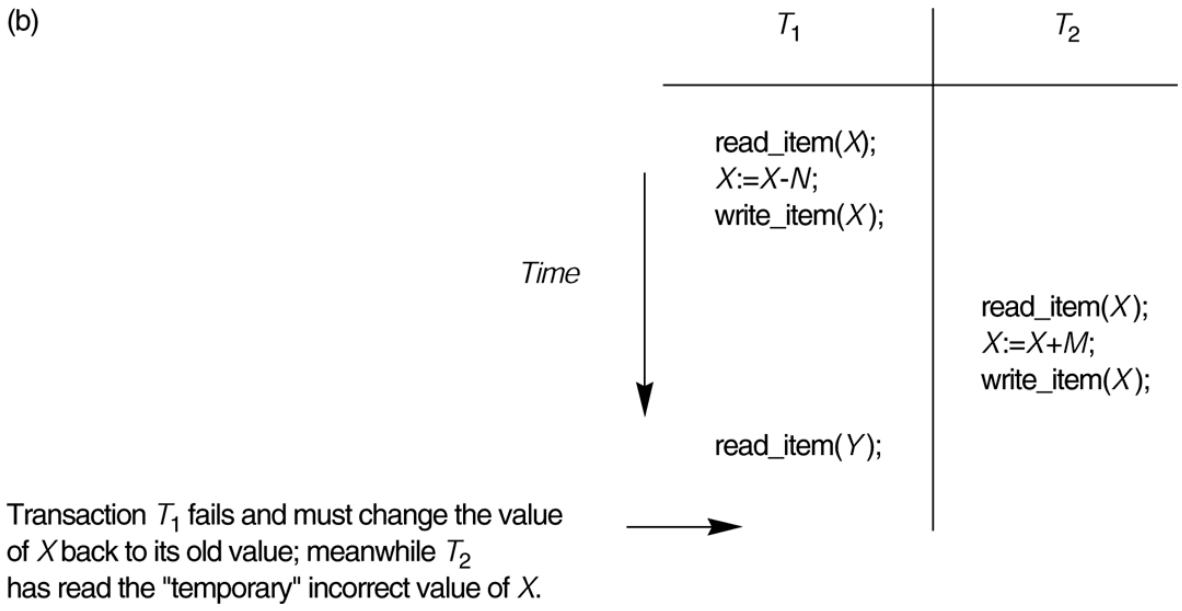


Fig a: The lost update problem.

The Temporary Update (or Dirty Read) Problem

This problem occurs when one transaction updates a database item and then the transaction fails for some reason.

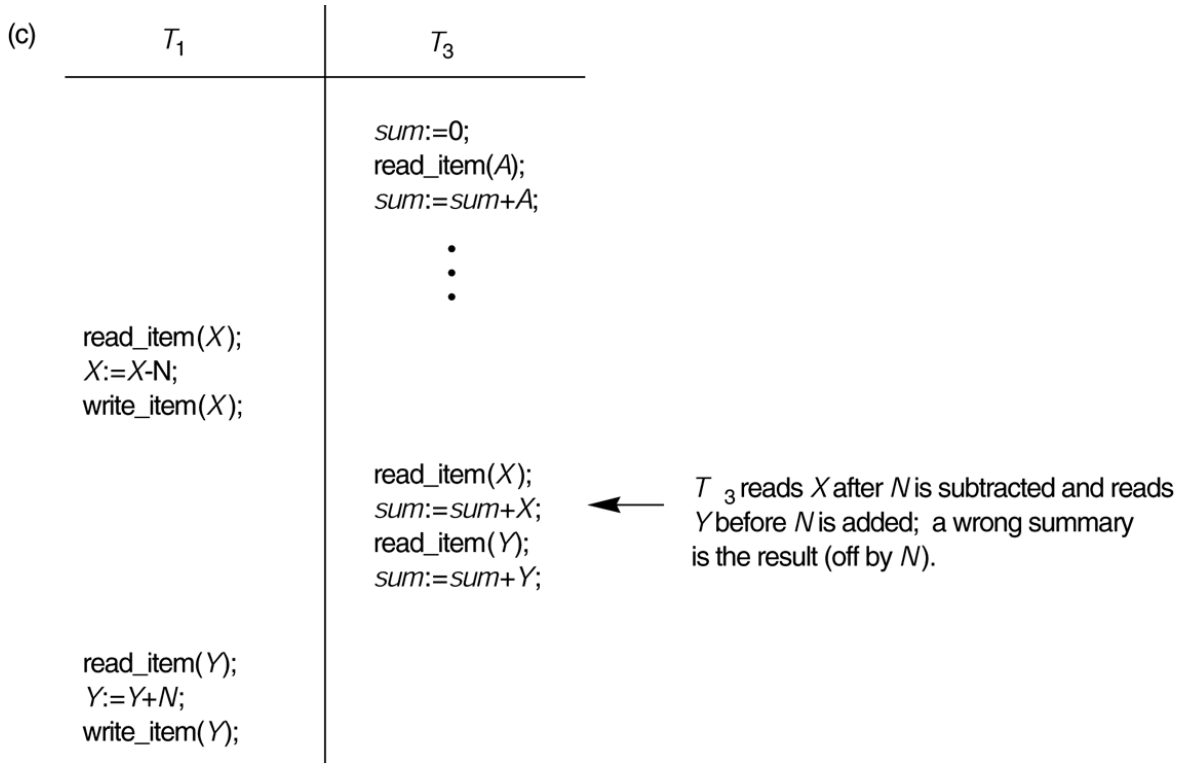
The updated item is accessed by another transaction before it is changed back to its original value. Figure b shows an example where T1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T r-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 Incorrect Summary Problem

If 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. For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1is executing. If the interleaving of operations shown in Figure c 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.

Another problem that may occur is called unrepeatable read, where 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.



Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either (1) all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or (2) 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.

This may happen if a transaction fails after executing some of its operations but before executing all of them.

Types of Failures

Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution

1. A *computer failure (system crash):* A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures-for example, main memory failure.

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: During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found.

Notice that an exception condition," such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be cancelled.

This exception should be programmed in the transaction itself, and hence would not be considered a failure.

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, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Schedule

When transactions are executing concurrently in an interleaved fashion, then the order Of execution of operations from the various transactions is known as a schedule.

A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed

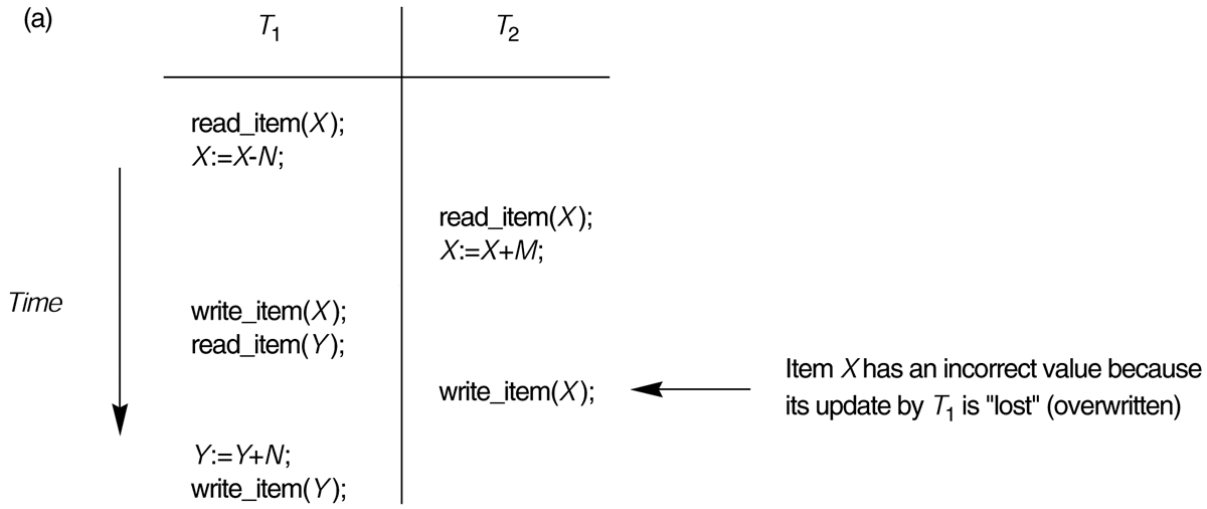
Schedules (Histories) of Transactions

A schedule (or history) S of n transactions T1 , T2, ... , Tn is an ordering of the operations of the transactions subject to the constraint that, for each transaction *Ti* that participates in S, the operations of *Ti,* in S must appear in the same order in which they occur in *Ti.*

That means a schedule for a set of transactions must consist of all instructions of those transactions.

For example, the schedule of below Figure which we shall call S_a can be written as follows in this notation:

S_a: r1(X); r2(X); *W1(X); r1(Y); ω2(X); W1(Y);*



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

(l) 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).

For example, in schedule *Sa* the operations r l(X) and *w2(X)* , the operations r2(X)and W1(X), and the operations w1(X) and *W2(X) are conflict.* However, the operations r l(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 rl(X)and *W1* (X) do not conflict, because they belong to the same transaction.

A schedule S of n transactions *T1, T2,* ••• , *Tn* is said to be a **complete schedule** if the following conditions hold:

1. The operations in S are exactly those operations in *T1,T2,* •.• , *Tn* 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 *Ti,* their order of appearance in S is the same as their order of appearance in *T;*
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

Characterizing Schedules Based on Recoverability

For some schedules it is easy to recover from transaction failures, whereas for other schedules the recovery process can be quite involved.

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. These characterizations do not actually provide the recovery algorithm but instead only attempt to theoretically characterize the different types of schedules.

Following are the schedules classified based on recoverability

1.Recoverable schedule
2.Cascadeless schedule
3.Strict schedules

1. **Recoverable schedule**

A schedule S is recoverable if no transaction in T1 in S commits until all Transactions of T2 that have written an item that T1 reads have committed.

A recoverable schedule is one where, for each pair of Transaction Ti and Tj such that Tj reads data item previously written by Ti then the commit operation of Ti appears before the commit operation Tj.

A transaction T1 reads from transaction T2 in a schedule S if some item X is first written by T2 and later read by T1.

Recoverable schedules require a complex recovery process

Sa: T1(X); T2(X); *W1*(X); *T1(Y); W2(X);* C2; *ω1(Y);* c1;

Sa is recoverable, even though it suffers from the lost update problem.

However,

consider the two (schedules Sc and Sd that follow:

Sc: rl(X); *W1*(X); r2(X); *r1(Y); ω2(X);* C2; al;

Sd: rl (X); *W1* (X); r2(X); rl*(Y); ω2(X); W1(Y);* C1; C2;

Se: rl(X); *W1*(X); T2(X); rl(Y); *W2(X); W1(Y);* a1; a2;

Sc is not recoverable, because T2 reads item X from T1 , and then T2 commits before *T1* commits. If TI 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 had been 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 T I commits, as shown in *Sd;* if T I aborts instead of committing, then

T2 should also abort as shown in *Se'* because the value of X it read is no longer valid.

2. **Cascadeless schedule**

A schedule is said to be cascadeless, if every transaction in the schedule reads only items that were written by committed transactions.
To satisfy this criterion, the *r2(X)* command in schedules *Sd* and *Se* must be postponed until after T1 has committed or aborted.

3. **Strict schedules**

   A schedule, called a strict schedule, in which transactions can *neither read nor write* an item X until the last transaction that wrote X has committed (or aborted). Strict schedules simplify the recovery process.

Schedules classified on Serializability:-

Serial schedule:

- 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 schedule.

Serializable schedule:

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

Assumption: Every serial schedule is correct

Goal: Like to find *non-serial* schedules which are also correct, because in serial schedules one transaction have to wait for another transaction to complete, Hence serial schedules are unacceptable in practice.

Result equivalent:

Two schedules are called result equivalent if they produce the same final state of the database.

Problem: May produce same result by accident!

S1	S2
read_item(X);	read_item(X);
X:=X+10;	X:=X*1.1;
write_item(X);	write_item(X);

Conflict equivalent:

Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

Conflict serializable:

A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.
Can reorder the *non-conflicting* operations to improve efficiency

Non-conflicting operations:

- Reads and writes from same transaction
- Reads from different transactions
- Reads and writes from different transactions on different data items

Conflicting operations:

- Reads and writes from different transactions on same data item

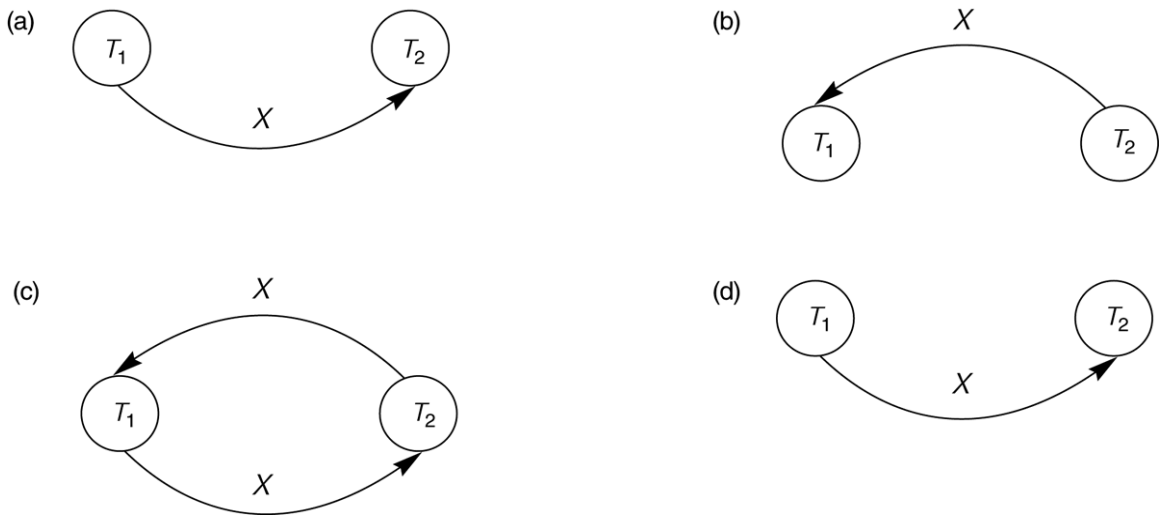
Test for Serializability:-

- Construct a directed graph, *precedence graph,* G = (V, E)
  - V: set of all transactions participating in schedule

- E: set of edges Ti → Tj for which one of the following holds:
  - Ti executes a write_item(X) before Tj executes read_item(X)
  - Ti executes a read_item(X) before Tj executes write_item(X)
  - Ti executes a write_item(X) before Tj executes write_item(X)
- An edge Ti → Tj means that in any serial schedule equivalent to S, Ti must come before Tj
- If G has a cycle, than S is not conflict serializable
- If not, use topological sort to obtain serialiazable schedule (linear order consistent with precedence order of graph)

FIGURE 17.7 Constructing the precedence graphs for schedules A and D from Figure 17.5 to test for conflict serializability.

- Precedence graph for serial schedule A.
- Precedence graph for serial schedule B.
- Precedence graph for schedule C (not serializable).
- Precedence graph for schedule D (serializable, equivalent to schedule A).



View equivalence:

A less restrictive definition of equivalence of schedules

View serializability:

definition of serializability based on view equivalence. A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

Two schedules are said to be **view equivalent** if the following three conditions hold:

1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
2. For any operation Ri(X) of Ti in S, if the value of X read by the operation has been written by an operation Wj(X) of Tj (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation Ri(X) of Ti in S'.
3. If the operation Wk(Y) of Tk is the last operation to write item Y in S, then Wk(Y) of Tk must also be the last operation to write item Y in S'.

The premise behind view equivalence:

- As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction musr produce the same results.
- **"The view"**: the read operations are said to see the *the same view* in both schedules.

Relationship between view and conflict equivalence:

- The two are same under **constrained write assumption** which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)
- Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serialiable.
- Any conflict serializable schedule is also view serializable, but not vice versa.

Consider the following schedule of three transactions

T1: r1(X), w1(X); T2: w2(X); and T3: w3(X):

Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;

In Sa, the operations w2(X) and w3(X) are blind writes, since T1 and T3 do not read the value of X.

Sa is **view serializable**, since it is view equivalent to the serial schedule T1, T2, T3. However, Sa is **not conflict serializable**, since it is not conflict equivalent to any serial schedule.

Introduction to Concurrency

What is concurrency?

Concurrency in terms of databases means allowing multiple users to access the data contained within a database at the same time. If concurrent access is not managed by the Database Management System (DBMS) so that simultaneous operations don't interfere with one another problems can occur when various transactions interleave, resulting in an inconsistent database.

Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions. Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins. Concurrent execution of user programs is essential for good DBMS performance. Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently. Interleaving actions of different user programs can lead to inconsistency: e.g., check is cleared while account balance is being computed. DBMS ensures such problems don't arise: users can pretend they are using a single-user system.

Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- 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 A and if the other transaction is rolled-back or waits.

LOCK

Definition : Lock is a variable associated with data item which gives the status whether the possible operations can be applied on it or not.

Two-Phase Locking Techniques:

Binary locks: Locked/unlocked

The simplest kind of lock is a binary on/off lock. This can be created by storing a lock bit with each database item. If the lock bit is on (e.g. = 1) then the item cannot be accessed by any transaction either for reading or writing, if it is off (e.g. = 0) then the item is available. Enforces mutual exclusion

Binary locks are:

- Simple but are restrictive.
- Transactions must lock every data item that is read or written
- No data item can be accessed concurrently

  Locking is an operation which secures

  (a) permission to Read

  (b) permission to Write a data item for a transaction.

  Example: Lock (X). Data item X is locked in behalf of the requesting transaction.

Unlocking is an operation which removes these permissions from the data item.

Example:Unlock (X): Data item X is made available to all other transactions.

- Lock and Unlock are Atomic operations.
- Lock Manager:

  Managing locks on data items.

- Lock table:

  Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode . One simple way to implement a lock table is through linked list.

< locking_transaction ,data item, LOCK >

The following code performs the lock operation:

```
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);
   goto B
   end;
```

The following code performs the unlock operation:

```
LOCK (X) ← 0 (*unlock the item*)
if any transactions are waiting then
     wake up one of the waiting the transactions;
```

Multiple-mode locks: Read/write

- a.k.a. Shared/Exclusive

- Three operations
  - read_lock(X)
  - write_lock(X)
  - unlock(X)
- Each data item can be in one of three lock states
  - Three locks modes:
    - (a) shared (read) (b) exclusive (write) (c) unlock(release)
  - **Shared mode**: shared lock (X)

More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.

– **Exclusive mode**: Write lock (X)

Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

_ **Unlock mode**: Unlock(X)

After reading or writing the corresponding transaction releases by issuing this.

The rules for multiple-mode locking schemes are a transaction T:

- Issue a **read_lock(X)** or a **write_lock(X)** before **read(X)**
- Issue a **write_lock(X)** before **write(X)**
- Issue an **unlock(X)** after all **read(X)** and **write(X)** are finished

The transaction T

- Will not issue **read_lock (X)** or **write_lock(X)** if it already holds a lock on **X**
- Will not issue **unlock(X)** unless it already holds a lock on X

Lock table:

Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and no of transactions that are currently reading the data item . It looks like as below

<data item,read_ LOCK,nooftransactions,transaction id >

This protocol isn't enough to guarantee serializability. If locks are released too early, you can create problems. This usually happens when a lock is released before another lock is acquired.

The following code performs the **read operation**:

```
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;
```

The following code performs the **write lock operation**:

```
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;
```

The following code performs the **unlock operation**:

```
    if LOCK (X) = "write-locked" then
begin LOCK (X) ← "unlocked";
     wakes up one of the 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";
      wake up one of the transactions, if any
       end
    end;
```

Lock conversion:-----

Lock upgrade: existing read lock to write lock

if Ti has a read-lock (X) and Tj has no read-lock (X) ($i \neq j$) then

convert read-lock (X) to write-lock (X)

else

force Ti to wait until Tj unlocks X

Lock downgrade: existing write lock to read lock

Ti has a write-lock (X) (*no transaction can have any lock on X*)

convert write-lock (X) to read-lock (X)

Two-Phase Locking Techniques: The algorithm

The timing of locks is also important in avoiding concurrency problems. A simple requirement to ensure transactions are serializable is that all read and write locks in a transaction are issued before the first unlock operation known as a two-phase locking protocol.

Transaction divided into 2 phases:

- *growing* - new locks acquired but none released
- *shrinking* - existing locks released but no new ones acquired

During the shrinking phase no new locks can be acquired!

- Downgrading ok
- Upgrading is not

Rules of 2PL are as follows:

- If T wants to read an object it needs a read_lock
- If T wants to write an object, it needs a write_lock
- Once a lock is released, no new ones can be acquired.

The 2PL protocol guarantees serializability

- Any schedule of transactions that follow 2PL will be serializable
- We therefore do not need to test a schedule for serializability

But, it may limit the amount of concurrency since transactions may have to hold onto locks longer than needed, creating the new problem of deadlocks.

Two-Phase Locking Techniques: The algorithm

Here is a example without 2PL:-

T1	**T2**	**Result**
read_lock (Y);	read_lock (X);	Initial values: X=20; Y=30
read_item (Y);	read_item (X);	Result of serial execution
unlock (Y);	unlock (X);	T1 followed by T2
write_lock (X);	Write_lock (Y);	X=50, Y=80.
read_item (X);	read_item (Y);	Result of serial execution
X:=X+Y;	Y:=X+Y;	T2 followed by T1
write_item (X);	write_item (Y);	X=70, Y=50
unlock (X);	unlock (Y);	

T1	**T2**	**Result**
read_lock (Y);		X=50; Y=50
read_item (Y);		Nonserializable because it.
unlock (Y);		violated two-phase policy.
	read_lock (X);	
	read_item (X);	
	unlock (X);	
	write_lock (Y);	
	read_item (Y);	
	Y:=X+Y;	
	write_item (Y);	
	unlock (Y);	
write_lock (X);		
read_item (X);		
X:=X+Y;		
write_item (X);		
unlock (X);		

Here is a example with 2PL:-

T'1	**T'2**	**Result**

read_lock (Y);	read_lock (X);	T1 and T2 follow two-phase
read_item (Y);	read_item (X);	policy but they are subject to
write_lock (X);	Write_lock (Y);	deadlock, which must be
unlock (Y);	unlock (X);	dealt with.
read_item (X);	read_item (Y);	
X:=X+Y;	Y:=X+Y;	
write_item (X);	write_item (Y);	
unlock (X);	unlock (Y);	

Two-phase policy generates four locking algorithms:-

1. BASIC
2. CONSERVATIVE
3. STRICT
4. RIGOUROUS

- Previous technique knows as *basic 2PL*
- ***Conservative* 2PL (static) 2PL**: Lock all items needed BEFORE execution begins by predeclaring its read and write set
  - If any of the items in read or write set is already locked (by other transactions), transaction waits (does not acquire any locks)
  - Deadlock free but not very realistic
- ***Strict* 2PL**: Transaction does not release its write locks until AFTER it aborts/commits

  - Not deadlock free but guarantees recoverable schedules (strict schedule: transaction can neither read/write X until last transaction that wrote X has committed/aborted)
  - Most popular variation of 2PL
- ***Rigorous* 2PL**: No lock is released until after abort/commit
  - Transaction is in its expanding phase until it ends