Unit VII
Transaction Processing Concepts

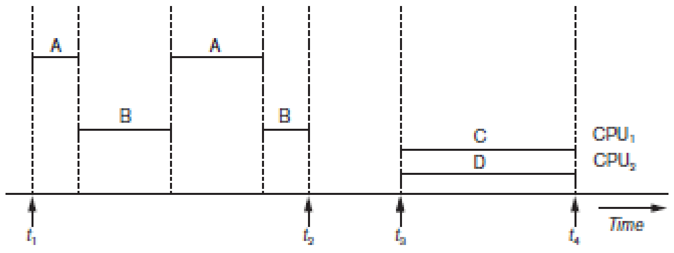
Introduction to transaction processing:

A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further. 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 executing database transactions. Examples of such systems include airline reservation, banking, credit card processing, stock markets, and so on.

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, and it is **multiuser** if many users can use the system at a time. Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently.

Multiple users can access databases simultaneously because of the concept of **multiprogramming**, which 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 execute at most one process at a time. However, **multiprogramming operating systems** execute some commands from one process, then suspend that process and execute some commands from 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 processes is actually **interleaved,** as illustrated in figure below, which shows two processes A and B executing concurrently in an interleaved fashion. If the computer system has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in the figure given below.



Transactions, 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. Every database operation involves two major operations called **read** and **write.** The DBMS will generally maintain a number of buffers in main memory that hold database disk blocks containing the database items being processed.

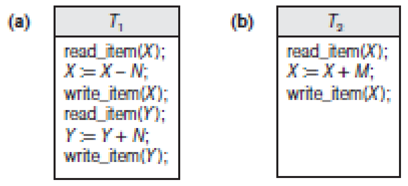
- read_item(X): Reads a database item named X into a program variable.
- write_item(X): Writes the value of program variable X into the database item named X.

Steps involved in read_item(X):

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.

Steps involved in write_item(X):

1. Find the address of the disk block that contains item X.
2. Copy the 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.



(a)

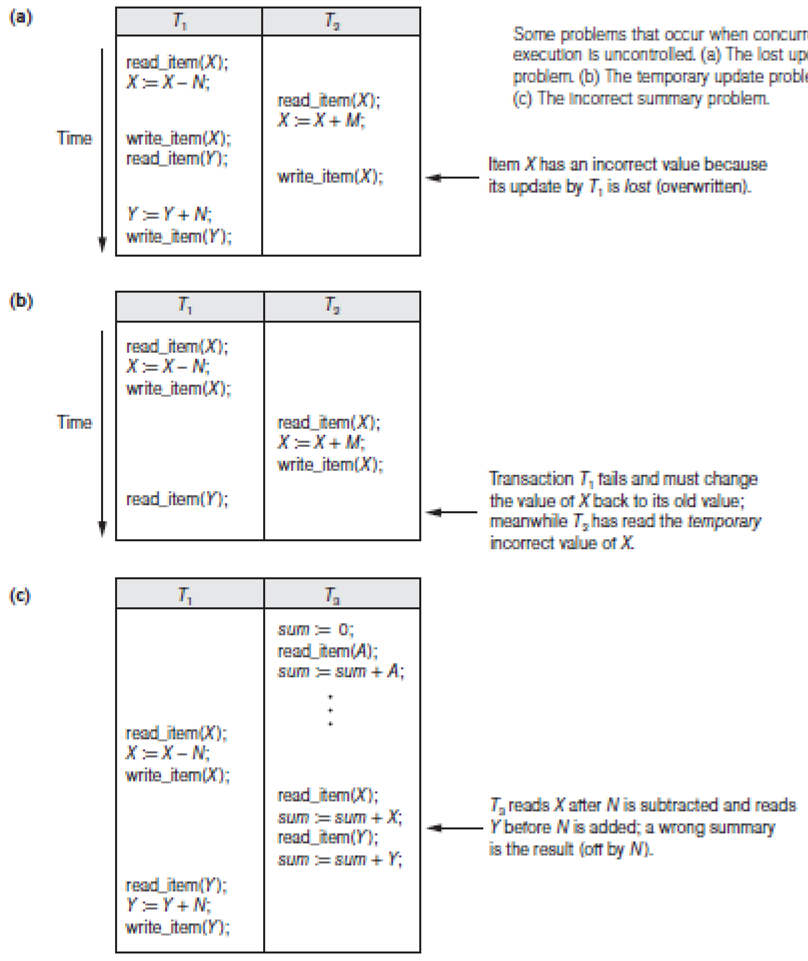
T_1
read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);

(b)

T_2
read_item(X); $X := X + M$; write_item(X);

Why Concurrency Control Is Needed:

Several problems can occur when concurrent transactions execute in an uncontrolled manner. The types of problems we may encounter with these transactions if they run concurrently.



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

(a)

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

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

(b)

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

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

(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).

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 the transactions T_1 and T_2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in the above figure (a); then the final value of item X is incorrect because T_2 reads the value of X before T_1 changes it in the database, and hence the updated value resulting from T_1 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 the above figure (a), it is X=84 because the update in T_1, that removed the five seats from X was lost.

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. The above figure (b) shows an example where T_1 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 T_2 reads the *temporary* value of X, which will not be recorded permanently in the database because of the failure of T_1. This type of problem is known as *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 T_3 is calculating the **total number** of reservations on **all** the flights; meanwhile, transaction T_1 is executing. If the interleaving of operations shown in the above figure (c) occurs, the result of T_3 will be off by an amount N because T_3 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.

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:

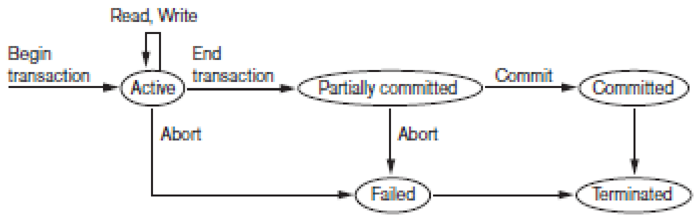
1. **A computer failure (system crash):** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures – for example, main memory failure.
2. **A transaction or system error:** Some operations in the transaction may cause 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.
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, insufficient balance in bank account, etc.
4. **Concurrency control environment:** The concurrency control method may decide to abort the transaction, to be restarted later, 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 of because of a disk read/write head crash.
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, etc.

Transaction States:
A transaction is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts. Therefore, the recovery manager keeps 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.
4. **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.
5. **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.

State transition diagram illustrating the states for transaction execution.



Desirable Properties (ACID properties) of Transactions:
Transaction should posses several properties, often called the ACID properties. They should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

1. **Atomicity:** A transaction is an atomic unit of processing. It is either performed in its entirety or not performed at all.
2. **Consistency preservation:** A transaction is consistency preservation if its complete execution take(s) the database from one consistent state to another.
3. **Isolation:** 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 any other transaction executing concurrently.
4. **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.

Concurrency Control Technique:

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. 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 concurrency transactions to the database items.

Types of locks:

Several types of locks are used in concurrency control such as *binary locks and shared/exclusive locks.*

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

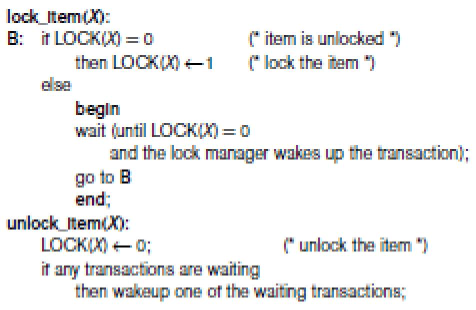
Two operations, *lock_item* and *unlock_item*, are used with binary locking.

Lock_item(X):

A transaction requests access to an item X by first issuing a lock_item(X) operation. If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction locks the item) and the transaction is allowed to access item X.

Unlock_item (X):

When the transaction is through using the item, it issues an unlock_item(X) operation, which sets LOCK(X) 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; i.e., at a time only one transaction can hold a lock.



```
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;
```

Lock and unlock operations for binary locks.

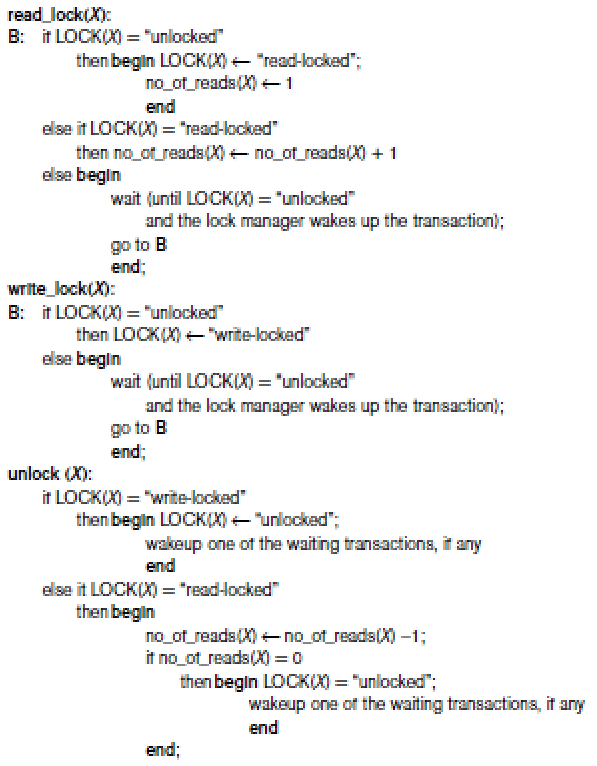
Shared/Exclusive (or Read/Write) Lock:

Shared lock:

These locks are referred to as read locks. If a transaction T has obtained Shared-lock on data item X, then T can read X, but cannot write X. Multiple Shared lock can be placed simultaneously on a data item.

Exclusive lock:

These Locks are referred to as write locks. If a transaction T has obtained Exclusive lock on data item X, then T can be read as well as write X. Only one Exclusive lock can be placed on a data item at a time. This means that a single transaction exclusively holds the lock on the item.



```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
                no_of_reads(X) ← 1
                end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else if LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) −1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

Two-Phase Locking (2PL):

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; and a **shrinking (second) phase,** during which existing locks can be released but no new locks can be acquired.

Deadlocks:

A deadlock is a condition in which two (or more) transactions in a set are waiting simultaneously for locks held by some other transaction in the set. Neither transaction can continue because each transaction in the set is on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. Thus, a deadlock is an impasse that may result when two or more transactions are each waiting for locks to be released that are held by the other. Transactions whose lock requests have been refused are queued until the lock can be granted.

A deadlock is also called a *circular waiting condition* where two transactions are waiting (directly or indirectly) for each other. Thus in a deadlock, two transactions are mutually excluded from accessing the next record required to complete their transactions.

Example:
A deadlock exists two transactions A and B exist in the following example:
Transaction A=access data items X and Y
Transaction B=access data items Y and X
Here, Transaction-A has acquired lock on X and is waiting to acquire lock on y. While, Transaction-B has acquired lock on Y and is waiting to acquire lock on X. But, none of them can execute further.

Deadlock Detection and Prevention:

Deadlock detection:

This technique allows deadlock to occur, but then, it detects it and solves it. Here, a database is periodically checked for deadlocks. If a deadlock is detected, one of the transactions, involved in deadlock cycle, is aborted. Other transactions continue their execution. An aborted transaction is rolled back and restarted.

Deadlock Prevention:

Deadlock prevention technique avoids the conditions that lead to deadlocking. It requires that every transaction lock all data items it needs in advance. If any of the items cannot be obtained, none of the items are locked. In other words, a transaction requesting a new lock is aborted if there is the possibility that a deadlock can occur. Thus, a timeout may be used to abort transactions that have been idle for too long. This is a simple but indiscriminate approach. If the transaction is aborted, all the changes made by this transaction are rolled back and all locks obtained by the transaction are released. The transaction is then rescheduled for execution. Deadlock prevention technique is used in two-phase locking.

Time-Stamp Methods for Concurrency control:

Timestamp is a unique identifier created by the DBMS to identify the relative starting time of transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system. So, a timestamp can be thought of as the transaction start time. Therefore, time stamping is a method of concurrency control in which each transaction is assigned a transaction timestamp.