Transaction Management

Dr. Jeevani Goonetillake

Single User Vs Multiuser Systems

- **Single user** -at most one user at a time can use the system. Restricted to some PC DBMS.
- **Multi-user** -many users can use the system concurrently (at the same time). Most DBMS are multi-user. E.g. Airline reservations systems, banks, Insurance agencies, stock exchanges are multi- user systems operated concurrently.

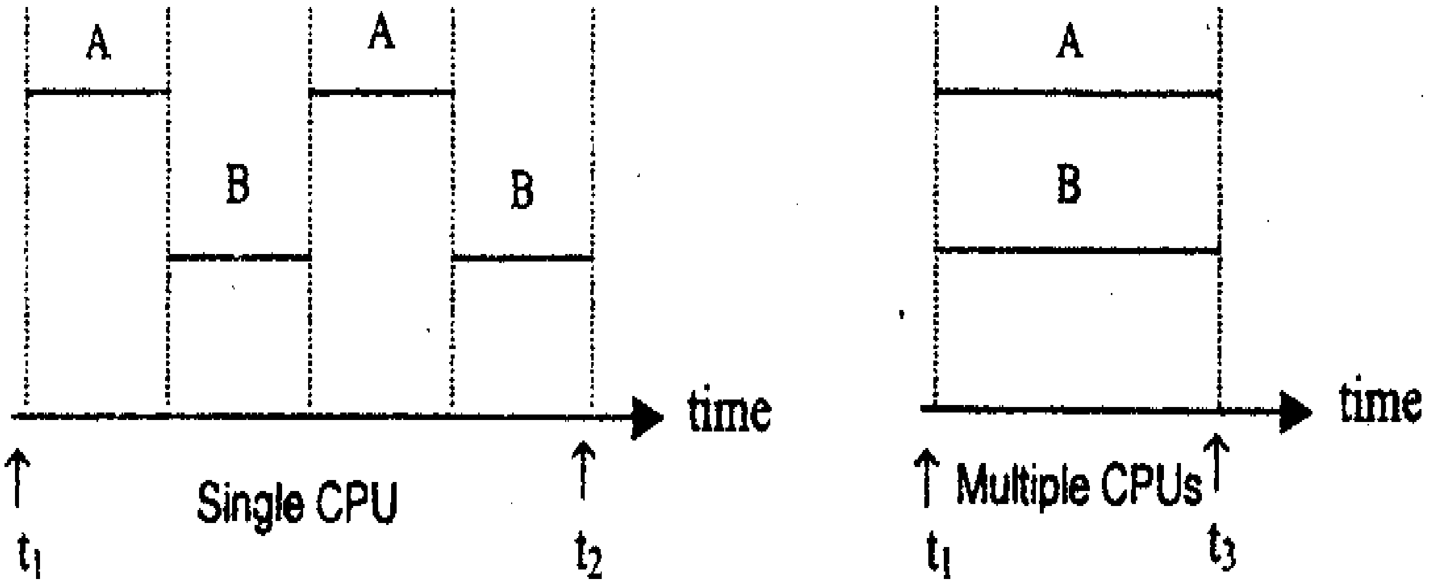
MultiProgramming

- Multiple users can use computer systems simultaneously because of the concept of **multiprogramming**.

- When only one CPU, the multiprogramming operating systems
  - execute some commands from one program,
  - then suspend that program and execute some commands from the next program and so on. A program is resumed at the point where it was suspended when it gets its turn to use the CPU again.

Interleaved Processing Vs Parallel Processing

- Hence, concurrent execution of the program is actually **interleaved**. Simultaneous processing of multiple programs are done with multiple CPUs.



Transaction Support

Transaction

Action, or series of actions, carried out by user or application, which accesses or changes contents of database.

- Logical unit of work on the database.
- Transforms database from one consistent state to another, although consistency may be violated during transaction.

Example Transaction

- E.g. Transaction Tl -No of reservations for airline A is X; No of reservation for airline B is *Y;* N reservation from A is cancelled and booked for B.
- Transaction T2 -M reservations to airline A.

T1	T2
read_item(X)	read_item(X)
X=X-N	X=X+M
write_item(X)	write_item(X)
read_item(Y)	
Y=Y+N	
write_item(Y)	

Transaction Support

- 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*.

Properties of Transactions

•Four basic *(ACID)* properties of a transaction are:

Atomicity 'All or nothing' property.

Consistency Must transform database from one consistent state to another.

Isolation Partial effects of incomplete transactions should not be visible to other transactions.

Durability Effects of a committed transaction are permanent and must not be lost because of later failure.

Transaction Support

- For recovery purpose, the system needs to keep track of when the transaction starts, terminates and commits or aborts. The recovery manager keeps track of:
  - BEGIN_TRANSACTION marks the beginning of transaction execution
  - READ or WRITE operations on the database items that are executed.
  - END_TRANSACTION specifies that READ and WRITE transaction operations have ended and mark the end of transaction execution.

Transaction Support

– COMMIT_TRANSACTION 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) 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.

Concurrency Control

Process 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.

Concurrency Control

- Three examples of potential problems caused by concurrency:
  - Lost update problem.
  - Uncommitted dependency problem.
  - Inconsistent analysis problem.

Lost Update Problem

- **This occurs when two transactions that access the same database item have their operations interleaved in a way that makes the value of some database item incorrect.**

  E.g. Originally there were 80 reservations on the flight.
  - **T_1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y.**
  - **T_2 reserves 4 seats on X.**
  - **Serially, final result of X should be 79.**

Lost Update Problem

T1	T2	
read_item(X)		X = 80, N = 5, M = 4
X = X -N		X = 75
	read_item(X)	X = 80
	X=X+M	X=84
write_item(X)		
read_item(Y)		
	write_item (X)	
Y=Y+N		
write_item(Y)		gives X = 84,
		but should be 80-5+4 = 79

- The update in T1 that removed the five seats from X was lost.

Uncommitted Dependency Problem

- Occurs when one transaction can see intermediate results of another transaction before it has committed.
- T1 cancels 5 seat reservations updating X to 75. Later T1 aborts, so X should be back at original value of 80.
- T2 has read new value of X (75) and reserves 4 seats, giving X = 79, instead of 84.

Uncommitted Dependency Problem

```
        T1                  T2
read_item(X)                                X = 80, N = 5, M = 4
X = X -N                                    X = 75
write_item(X)
                        read_item(X)        X = 75
                         X=X+M     X=79
                        write_item(X)
read_item(Y)
-abort -changes X back to its
original value gives X = 80,
                                    but should be 80+4 = 84
```

Uncommitted Dependency Problem

- 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*.

Uncommitted Dependency Problem

- This problem can be avoided by preventing T2 from reading X until after T1 commits or aborts.

Inconsistent Analysis 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.

Inconsistent Analysis Problem

```
T1                      T2
sum = 0
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+M
write_item(Y)
```

- **Problem avoided by preventing T2 from reading X and Y until after T1 completed updates.**

Serializability

- Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.

- Could run transactions serially, but this limits degree of concurrency or parallelism in system.

- Serializability identifies those executions of transactions guaranteed to ensure consistency.

Serializability

Schedule

Sequence of reads/writes by set of concurrent transactions.

Serial Schedule

Schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions.

- No guarantee that results of all serial executions of a given set of transactions will be identical.

Nonserial Schedule

- Schedule where operations from set of concurrent transactions are interleaved.
- Objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.
- In other words, want to find nonserial schedules that are equivalent to *some* serial schedule. Such a schedule is called *serializable*.

Equivalence of Schedules

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

  Two schedules can accidentally produce the same final database.

- 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.

Equivalence of Schedules

- In serializability, ordering of read/writes is important:
  - (a) If two transactions only read a data item, they do not conflict and order is not important.
  - (b) If two transactions either read or write completely separate data items, they do not conflict and order is not important.
  - (c) If one transaction writes a data item and another reads or writes same data item, order of execution is important.

Equivalence of Schedules

- Two definitions of equivalence of schedules are generally used:
  - Conflict equivalence
  - View equivalence

Conflict equivalence

- Two schedules are said to be conflict equivalent if order of any two conflicting operations is the same in both schedules.
- A schedule S is conflict serializable if it is (conflict) equivalent to some serial schedule S’.
- Conflict serializable schedule orders any conflicting operations in same way as some serial execution.

Precedence Graph

- Precedence graph is used for determining the conflict serializability of a schedule.
- Create:
  - node for each transaction;
  - a directed edge $T_i \rightarrow T_j$, if T_j reads the value of an item written by T_I;
  - a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been read by T_i.

- If precedence graph contains cycle schedule is not conflict serializable.

Example

- T9 is transferring £100 from one account with balance bal_x to another account with balance bal_y.
- T10 is increasing balance of these two accounts by 10%.

Example

Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	$bal_x = bal_x + 100$	
t_4	write(bal_x)	begin_transaction
t_5		read(bal_x)
t_6		$bal_x = bal_x * 1.1$
t_7		write(bal_x)
t_8		read(bal_y)
t_9		$bal_y = bal_y * 1.1$
t_{10}		write(bal_y)
t_{11}	read(bal_y)	commit
t_{12}	$bal_y = bal_y - 100$	
t_{13}	write(bal_y)	
t_{14}	commit	

View Serializability

- Offers less restrictive definition of schedule equivalence than conflict serializability.
- Two schedules S_1 and S_2 are view equivalent if the following three conditions hold:
  - For each data item x, if T_i reads initial value of x in S_1, T_i must also read initial value of x in S_2.

View Serializability

- For each read on x by T_i in S_1, if value read by x is written by T_j, T_i must also read value of x produced by T_j in S_2.
- For each data item x, if last write on x performed by T_i in S_1, same transaction must perform final write on x in S_2.

View Serializability

- Schedule is view serializable if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is view serializable, although converse is not true.
- It can be shown that any view serializable schedule that is not conflict serializable contains one or more blind writes.
- In general, testing whether schedule is serializable is NP-complete.

Example - View Serializable schedule

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

S1 : r1(X); w2(X); w1(X); w3(X);

w2(X) and w3(X) – blind writes

Schedule S1 is view serializable since it is equivalent to the serial schedule T1, T2, T3.