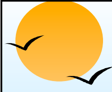
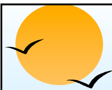


Chapter 16: Concurrency Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Deadlock Handling
- Insert and Delete Operations

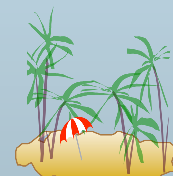


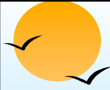
Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes: the write=>exclusive and the read => shared.
  1. *exclusive* *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared* *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

Lock compatibility Matrix:

	S	X
S	true	false
X	false	false



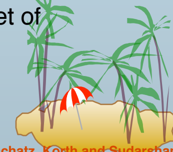


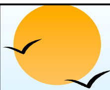
Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

  T_2: **lock-S**(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A+B*)

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.



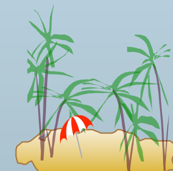


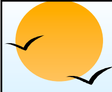
Pitfalls of Lock-Based Protocols

- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

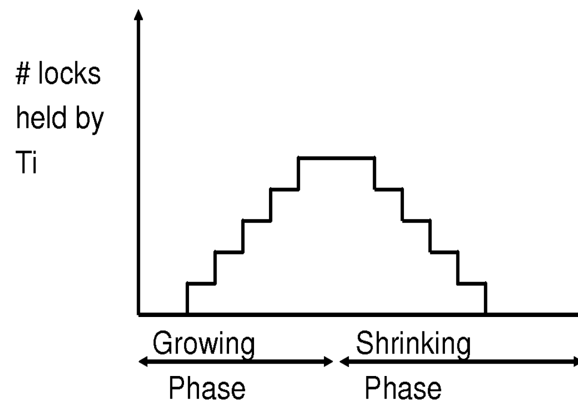


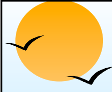


The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

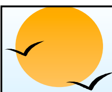
Two-Phase Locking





The Two-Phase Locking Protocol

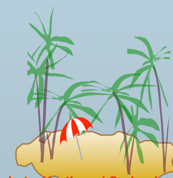
- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

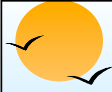


Pitfalls of 2PL

Conflict serializability achieved but:

1. Dirty reads are possible: for cascadeless use **Rigorous** 2PL.
2. Deadlock is possible (No transaction makes any progress)
   - Conservative 2PL,
   - deadlock detection
   - deadlock prevention.
3. Starvation: Some transaction makes no progress
4. Phantoms can still appear.





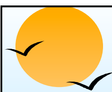
Deadlock Handling

- Consider the following two transactions:

  T_1: write (A) T_2: write(B)
  write(B) write(A)

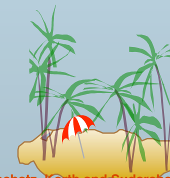
- Schedule with deadlock

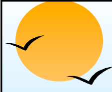
T_1	T_2
lock-X on A write (A)	
	lock-X on B write (B) wait for **lock-X** on A
wait for **lock-X** on B	



Deadlock Handling

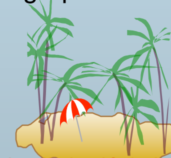
- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock Detection and recovery***
- ***Deadlock prevention:*** protocols ensure that the system will *never* enter into a deadlock state. Some non-optimal strategies :
  - Require that each transaction locks all its data items before it begins execution (conservative 2PL)
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol)—used in OS. Why not in DBs?

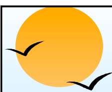




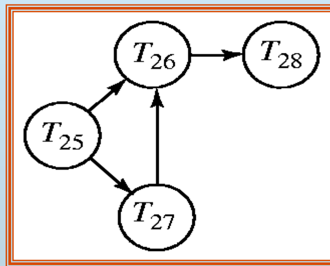
Deadlock Detection Protocols

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - ★ V is a set of vertices (all the transactions in the system)
  - ★ E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge T_i T_j is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

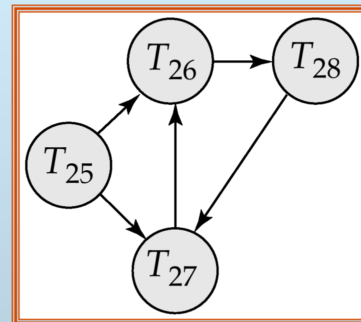




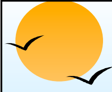
Deadlock Detection (Cont.)



Wait-for graph without a cycle

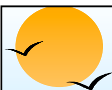


Wait-for graph with a cycle



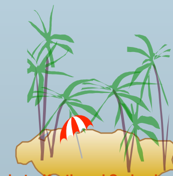
Deadlock Recovery

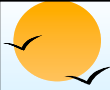
- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation



Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **wait-die** scheme — non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never waits for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - Less prone to rollbacks than *wait-die* scheme.



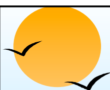


Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

Timeout-Based Schemes:

- a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
- thus deadlocks are not possible
- simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.



A schedule with no locks

T0	T1	T2
write(A)		
	read(B)	
Read(B)		
		write(C)
	read(A) write(A)	
		write(A)
read(C)		
	write(C)	
read(D)		

Now use a rigorous 2PL with locks issued just before use ...