BITS, PILANI – K. K. BIRLA GOA CAMPUS

Database Systems

(CS F212)

by

Dr. Mrs. Shubhangi Gawali

Dept. of CS and IS

Concurrency Control

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :

  1. *exclusive (X) mode*.
     - Data item can be both read as well as written.
     - Exclusive locks are placed on resources whenever Write operations(INSERT,UPDATE and DELETE) are performed.
     - Only one exclusive lock can be placed on a resource at a time i.e; the first user who acquires an exclusive lock will continue to have the sole ownership of the resource, and no other user can acquire an exclusive lock on that resource.

  2. *shared (S) mode*.
     - Data item can only be read.
     - Shared locks are placed on resources whenever a read operation (select) is performed.
     - Multiple shared locks can be simultaneously set on a resource.

Syntax

lock table <tablename> [,<tablename>]…

in {row share|row exclusive|share update|

share|share row exclusive|exclusive}

- Lock requests are made to concurrency - control manager. Transaction can proceed only after request is granted.

- Locks are released when transaction is commited or rolled back.

Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.

- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

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*)

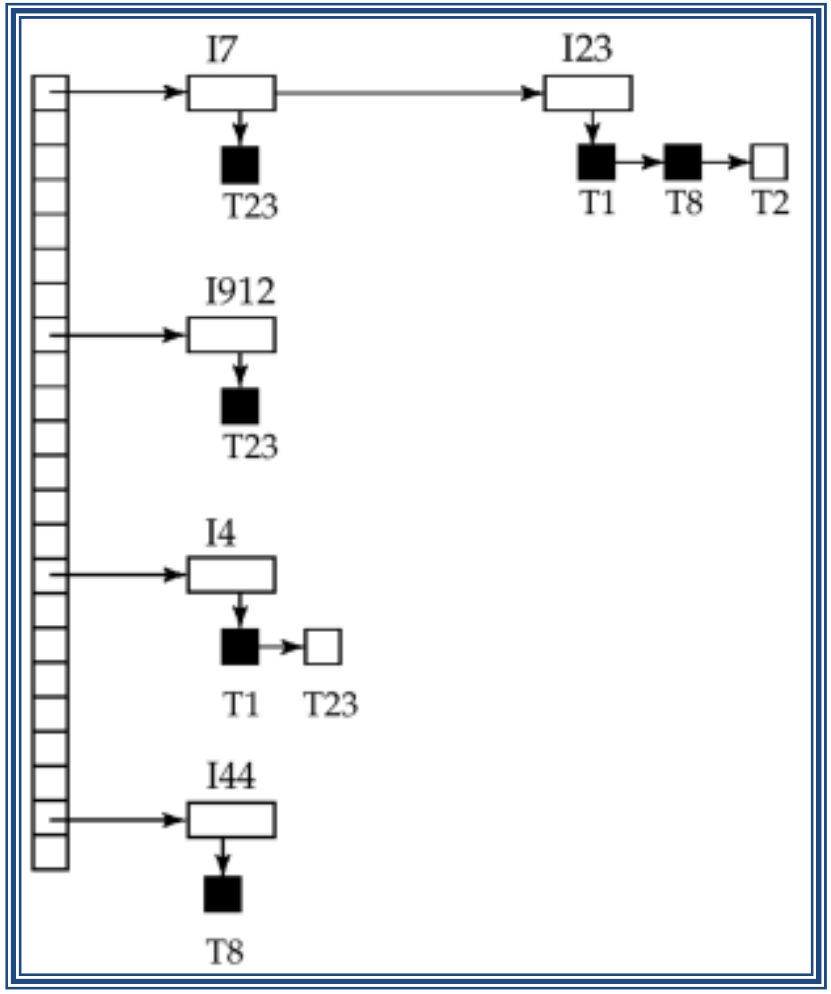
- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

- 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.

Lock Management

- Lock and unlock requests are handled by the lock manager.
- A **Lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a datastructure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations.

Lock Table



- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

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.

Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

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
- This 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).

Lock Conversions

- Two-phase locking with lock conversions:
- **First Phase:**
  - can acquire a **lock-S** on item
  - can acquire a **lock-X** on item
  - can convert a **lock-S** to a **lock-X** (**upgrade**)
    - Eg: update with where clause
    - Favours concurrency but does not prevent deadlocks
- **Second Phase:**
  - can release a **lock-S**
  - can release a **lock-X**
  - can convert a **lock-X** to a **lock-S (downgrade**)
    - Reduces concurrency
    - Improves throughput by reducing deadlocks
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

The Two-Phase Locking Protocol (Cont.)

- 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. (dynamic databases and phantom problem: index locking)
- **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.

Dynamic databases and phantom problem: index locking

- T1: find oldest sailor in each rating level
- T2: insert sailor having age=96 and rating =1

  : and delete oldest sailor having rating=2

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking.
- Impose a partial ordering $\rightarrow$ on the set **D** = {d_1, d_2 ,..., d_h} of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

Graph-Based Protocols (Cont.)

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
  - the abort of a transaction can still lead to cascading rollbacks.
- However, in the tree-locking protocol, a transaction may have to lock data items that it does not access.
  - increased locking overhead, and additional waiting time
  - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.

Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

Suppose a transaction T_i issues a **read**(Q)

1. If TS(T_i) < **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the **read** operation is rejected, and T_i is rolled back.

2. If TS(T_i)> **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

Timestamp-Based Protocols (Cont.)

Suppose that transaction T_i issues **write**(Q).

- If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the **write** operation is rejected, and T_i is rolled back.
- If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this **write** operation is rejected, and T_i is rolled back.
- Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}. Hence, rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency. Unlike previous protocols, it allows some view-serializable schedules that are not conflict-serializable.

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

  Thus, there will be no cycles in the precedence graph
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by T_j must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks

- Solution:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp

Exercise

Time	T19	T20	T21
t1	begin		
t2	R(balx)		
t3	balx=balx+10		
t4	W(balx)	begin	
t5		R(baly)	
t6		baly=baly+20	begin
t7			R(baly)
t8		W(baly)	
t9			baly=baly+30
t10			W(baly)
t11			balz=100
t12			W(balz)
t13	balz=50		commit
t14	W(balz)	begin	
t15	commit	R(baly)	
t16		baly=baly+20	
t17		W(baly)	
t18		commit	

Answer

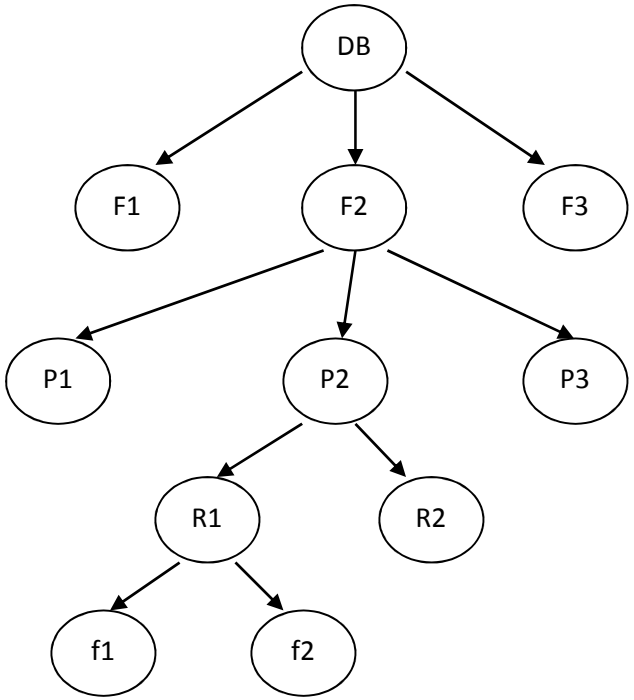
- At time t8, the write by transaction T20 violates the first timestamping write rule and hence must be aborted and restarted at time t14.

- At time t14, the write transaction T19 can safely be ignored as it would have been overwritten by the write of transaction T21 at time t12.

Multiple Granularity

- The size of data items chosen as the unit of protection by a concurrency control protocol is called as granularity.
- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.
- Can be represented graphically as a tree.
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - *fine granularity* (lower in tree): high concurrency, high locking overhead
  - *coarse granularity* (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy



The highest level in the example hierarchy is the entire database.

The levels below are of type *file, page(area), record* and field in that order.

Granularity of locking

- Whenever a node is locked, all its descendants are also locked.
- If another transaction requests a lock on any of the descendants of the locked node, the DBMS checks the hierarchical path from root to the requested node to determine any of its ancestors are locked before deciding whether to grant the lock. If it is locked then it denies the request.
- A transaction may lock on a node if the descendant of the node is already locked.

Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:
- Allow transactions to lock at each level, but with a special protocol using new “intention” locks.
- Before locking an item, transaction must set “intention locks” on all its ancestors.
- For unlock, go from specific to general (i.e., bottom-up).

	--	IS	IX	S	X
--	y	y	y	y	y
IS	y	y	y	y	n
IX	y	y	y	n	n
S	y	y	n	y	n
X	y	n	n	n	n

Two-phase locking protocol with new compatibility matrix

To ensure serializability with locking levels, a two-phase locking protocol is used as follows:

- No lock can be granted once any node has been unlocked.
- No node may be locked until its parent is locked by an intention lock.
- No node may be unlocked until all its descendants are unlocked.

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

Deadlock Handling

- Consider the following two transactions:

T_1:	write (X)	T_2:	write(Y)
	write(Y)		write(X)

- Schedule with deadlock

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

Deadlock Handling

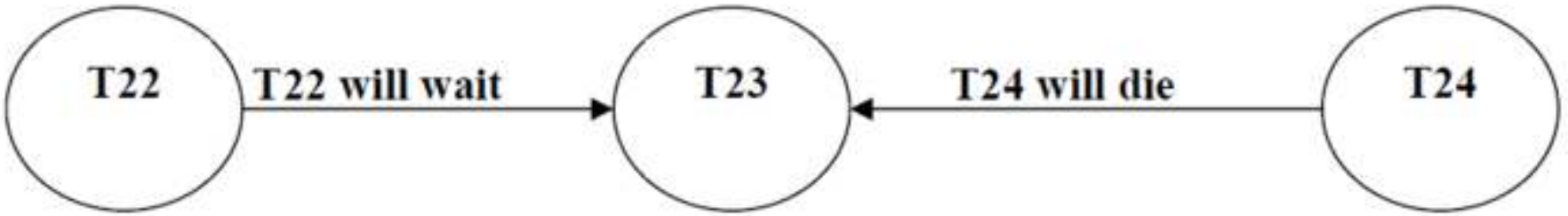
- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- **Deadlock:** Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection and Deadlock recovery
- **Deadlock prevention** protocols ensure that the system will *never* enter into a deadlock state.
- Some prevention strategies require that each transaction locks all its data items before it begins execution (predeclaration).

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 wait for older ones; they are rolled back instead.
  - a transaction may **die** several times before acquiring needed data item

wait-die scheme Example

- See explanation in notes below

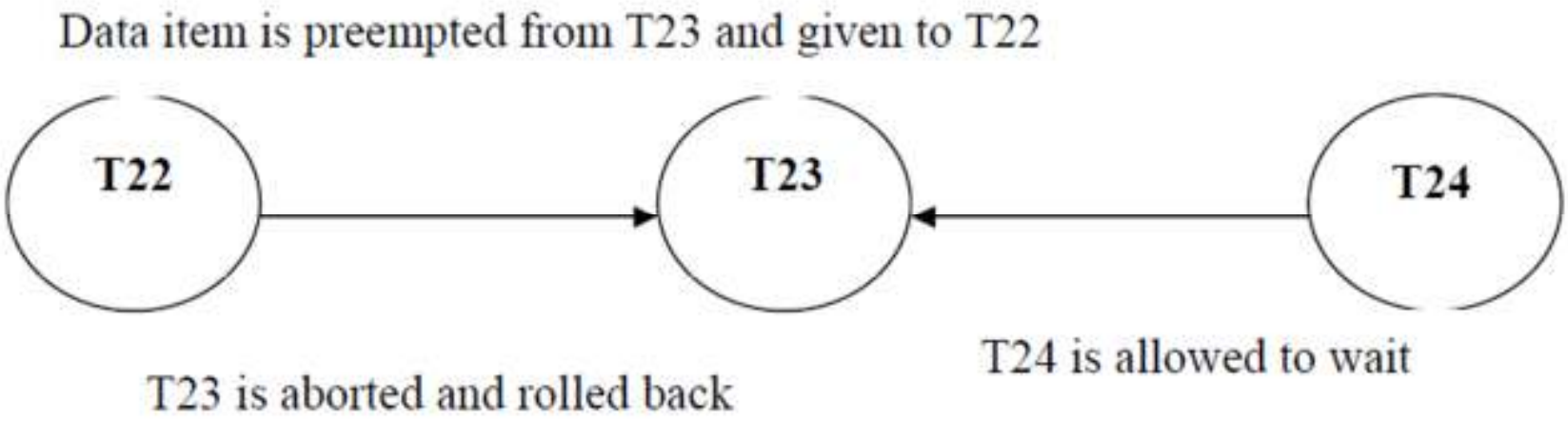


Deadlock Prevention Strategies contd…

- **wound-wait** scheme — preemptive
  - older transaction ***wounds*** (forces rollback) of younger transaction instead of waiting for it. Younger transactions may **wait** for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

wound-wait scheme Example

- See explanation in notes below



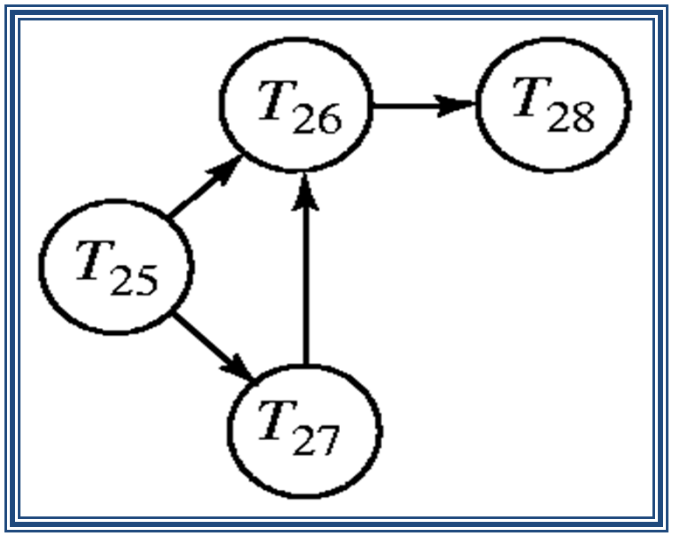
Deadlock prevention (Cont.)

- 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.

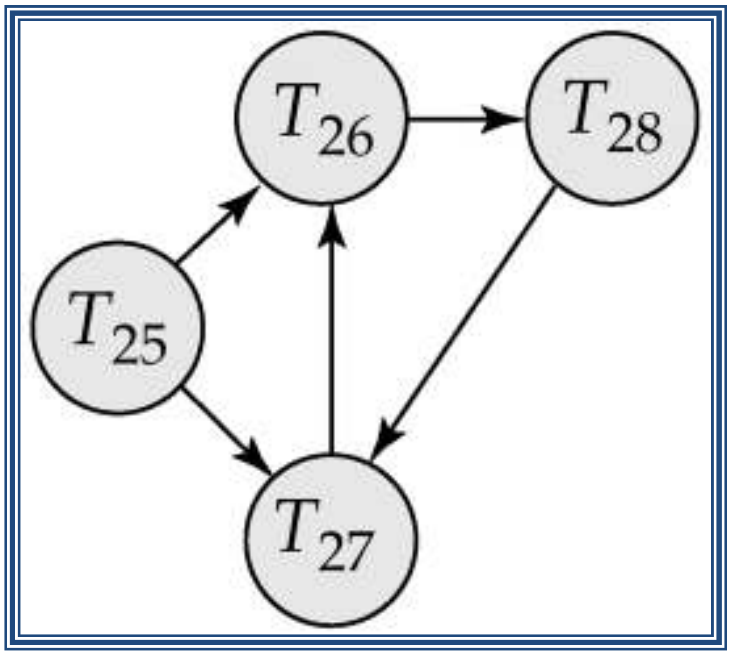
Deadlock Detection

- 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.

Example: Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

When deadlock is detected :

1. Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.

2. Rollback : determine how far to roll back transaction
   - **Total rollback** : Abort the transaction and then restart it.
   - **Partial rollback** :More effective to roll back transaction only as far as necessary to break deadlock.

3. Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation.