Comp 5311 Database Management Systems

14. Transactions-2PL

Transaction Concept

- A ***transaction*** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

To preserve integrity of data, the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- Transaction to transfer \$50 from account *A* to account *B*:
  - 1. **read**(*A*)
  - 2. $A := A - 50$
  - 3. **write**(*A*)
  - 4. **read**(*B*)
  - 5. $B := B + 50$
  - 6. **write**(*B*)
- Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- Consistency requirement – the sum of *A* and *B* is unchanged by the execution of the transaction.

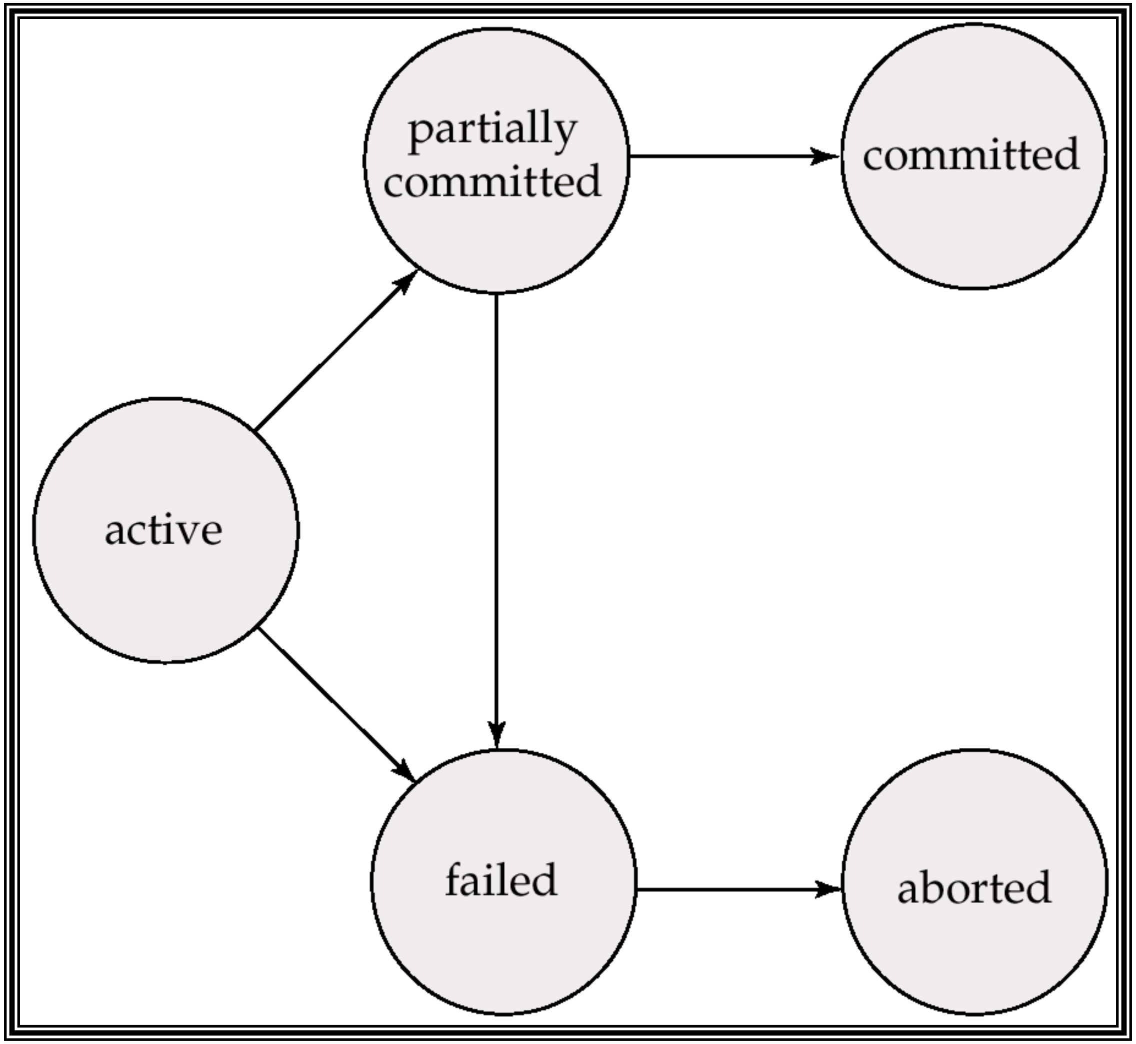
Example of Fund Transfer (Cont.)

- Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).
  Can be ensured trivially by running transactions ***serially***, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.
- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

Transaction State

- **Active,** the initial state; the transaction stays in this state while it is executing
- **Partially committed,** after the final statement has been executed.
- **Failed,** after the discovery that normal execution can no longer proceed.
- **Aborted,** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction – only if no internal logical error
  - kill the transaction
- **Committed,** after *successful completion*.

Transaction State (Cont.)



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes* – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database (problems occur when concurrent transactions access the same items).

Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

Serial Schedule

Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B. The following is a serial schedule, in which T_1 is followed by T_2.

Lets start with A=100 B=200

	T_1	T_2	
A=100	read(A)		
	$A := A - 50$		
A=50	write (A)		
B=200	read(B)		
	$B := B + 50$		
B=250	write(B)		
		read(A)	A=50
		$temp := A * 0.1$	*temp*=5
		$A := A - temp$	
		write(A)	A=45
		read(B)	B=250
		$B := B + temp$	
		write(B)	B=255

We end with A=45 B=255

Not Serial but Correct Schedule
equivalent to the previous serial schedule

Lets start with A=100 B=200

	T_1	T_2	
A=100	read(A)		
	$A := A - 50$		
A=50	write(A)		
		read(A)	A=50
		$temp := A * 0.1$	*temp*=5
		$A := A - temp$	
		write(A)	A=45
B=200	read(B)		
	$B := B + 50$		
B=250	write(B)		
		read(B)	B=250
		$B := B + temp$	
		write(B)	B=255

We end again with A=45 B=255 In both Schedules, the sum A + B is preserved.

Not Serial and Incorrect Schedule

The following concurrent schedule does not preserve the value of the the sum $A + B$ – The schedule is wrong and should not be allowed.

	T_1	T_2	
A=100	read(A)		
	$A := A - 50$		
		read(A)	A=100
		$temp := A * 0.1$	*temp*=10
		$A := A - temp$	
		write(A)	A=90
		read(B)	B=200
A=50	write(A)		
B=200	read(B)		
	$B := B + 50$		
B=250	write(B)		
		$B := B + temp$	
		write(B)	B=210

We end with A=50 B=210

Serializability

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is ***equivalent*** to a serial schedule.
- We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

Conflict Serializability

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions writes Q.
  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
  2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
  3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  4. l_i = **write**(Q), l_j = **write**(Q). They conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them. If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability (Cont.)

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

we are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Conflict Serializable Schedule

The following schedule is equivalent to a serial schedule where T_2 follows T_1, by a series of swaps of non-conflicting instructions.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Recoverability

- **Recoverable schedule** — if a transaction T_j reads data items previously written by a transaction T_i, the commit operation of T_i must appear before the commit operation of T_j.

If T_9 commits immediately after the read, the following schedule is not recoverable and the durability property is violated. If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. All schedules must be recoverable.

T_8	T_9
Read(A)	
Write(A)	
	Read(A)
	Commit
Read (B)	

Cascading Rollback

- The following schedule is recoverable because every transaction T_i commits after all transactions that wrote items which T_i read.

T_{10}	T_{11}	T_{12}
Read(A)		
Write(A)		
	Read(A)	
	Write(A)	
		Read(A)
Commit		
	Commit	
		Commit

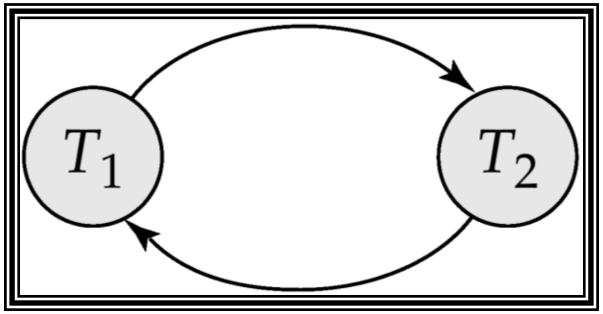
- *Cascading Rollback*: when a single transaction failure leads to a series of transaction rollbacks. If T_{10} fails, T_{11} and T_{12} must also be rolled back. This can lead to the undoing of a significant amount of work
  - How would you put the commit statements to make the schedule cascadeless?

Cascadeless Schedules

- Schedules where cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Testing for Serializability

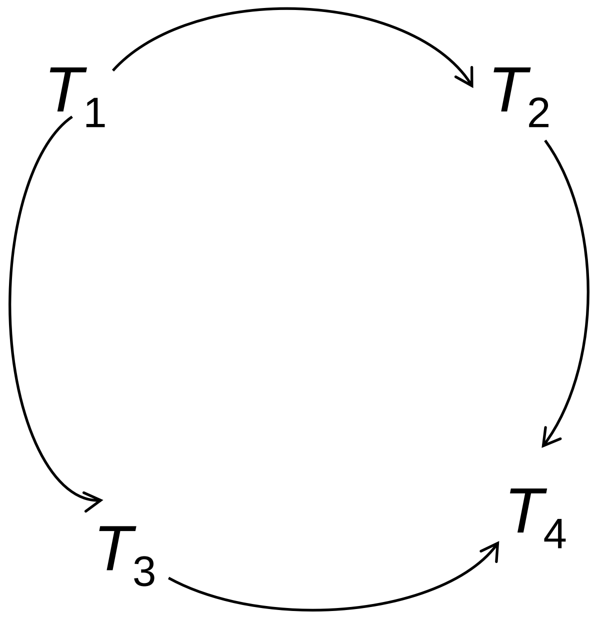
- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.



Example Schedule (Schedule A)

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

Precedence Graph for Schedule A



Concurrency Control vs. Serializability Tests

- Testing a schedule for serializability *after* it has executed is too late!
- Goal – to develop concurrency control protocols that will assure serializability. They will generally not examine the precedence graph as it is being created; instead a protocol will impose a discipline that avoids nonseralizable schedules.

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. 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 requests are made to concurrency-control manager. Transaction can proceed only after request is granted.
- Should only allow conflict-serializable schedules.

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.

Example of Lock-Based Protocol

A transaction that displays *A+B*:

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.

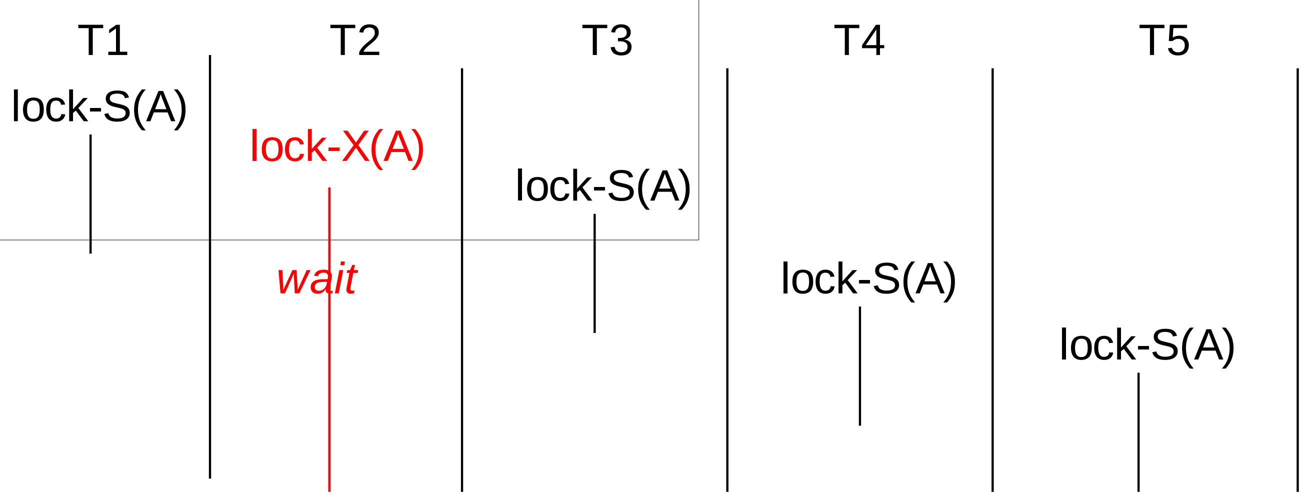
Pitfalls of Lock-Based Protocols-Deadlock

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 - Starvation

- 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 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).
- If a schedule is executed by 2PL it must be conflict serializable.
- If a schedule is conflict serializable it may or may not be executed by 2PL.

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**)

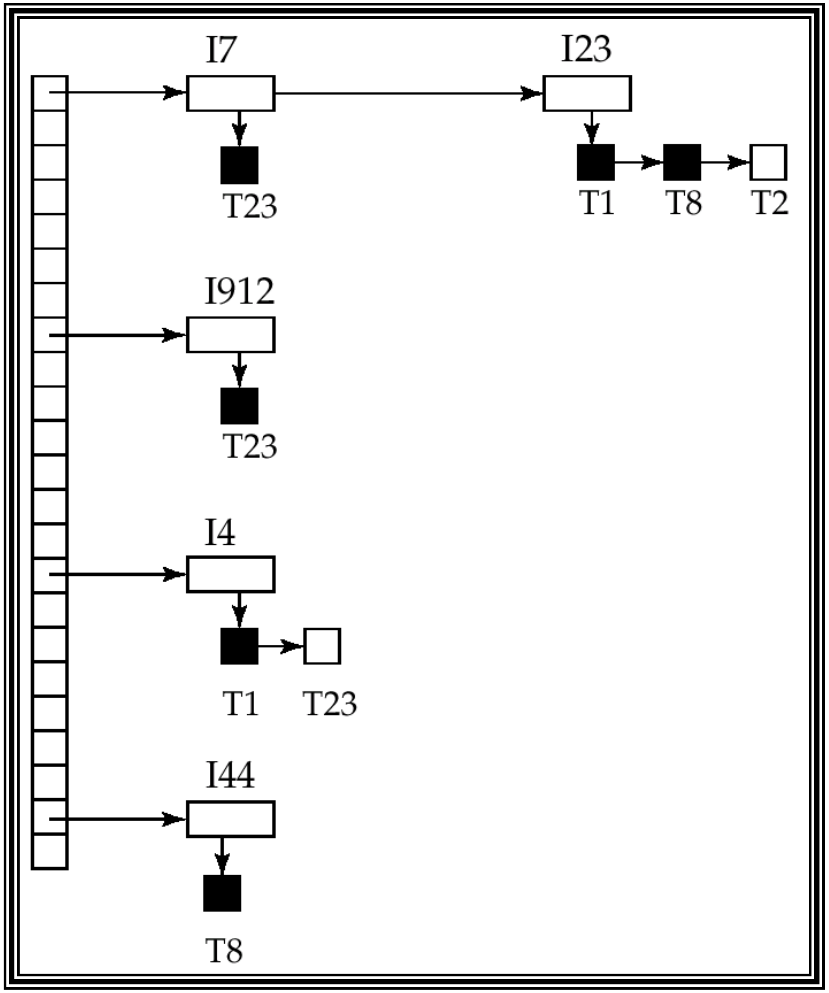
Second Phase:

- can release a **lock-S**
- can release a **lock-X**
- can convert a **lock-X** to a **lock-S** **(downgrade**)

Implementation of Locking

- 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 data structure 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



- 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

Strict 2PL

- Cascading roll-back is the situation where the failure of a transaction T_i may lead to failures of other transactions (because they read items written by T_i before its commitment)
- 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.

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. 2PL permits deadlocks.
- ***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).
  - 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).

More Deadlock Prevention Strategies

- The 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
- **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.

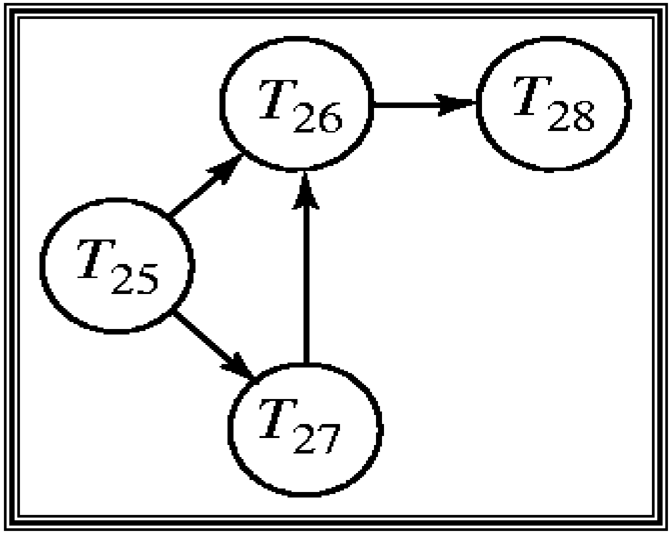
Deadlock prevention (Cont.)

- 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 a pre-defined waiting period, the transaction is rolled back.
  - Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

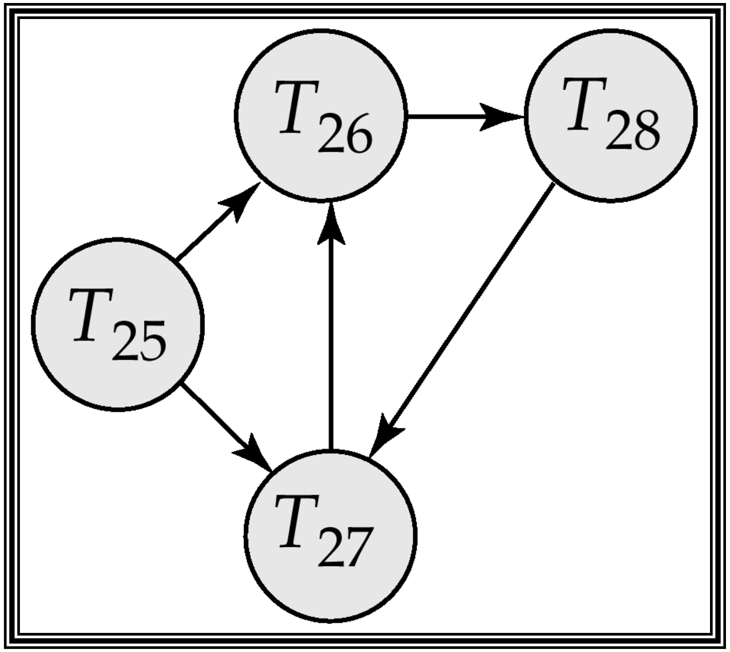
Deadlock Detection

- 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.

Deadlock Detection Examples



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