Transactions and Concurrency Control

- Transaction model
- ACID properties
- Concurrency control
- Serializability
- Lock-based protocol
- Graph-based protocols
- Deadlock handling

Transactions

- What is a transaction ?
  - a sequence of operations
  - typically changes data under certain satisfied conditions
- E.g. transfer $50 from account A to account B

  read(A)
  A:= A – 50
  write(A)
  read(B)
  B:= B+ 50
  write(B)
- Challenge → make sure transactions are performed **properly**

Transactions (Cont.)

- E.g. transfer $50 from account A to account B

  read(A)
  A:= A – 50
  write(A)
  read(B)
  B:= B+ 50
  write(B)

- Issues
  - Failures
  - Concurrent execution of multiple transactions

Database Systems

- **Transaction Processing (TP)**
  - many queries
  - many updates
  - small queries
- **On-Line Analytical Processing (OLAP)**
  - few queries
  - few updates
  - large queries
- **Often specialized systems**

Applications of Transaction Processing

- Retail
- Banking
- Electronic Trading
- Credit Cards
- Telephony
  - Phone cards
  - 1-800 services
  - Lucent / AT&T

ACID properties

- **(A)**tomicity
    - No partial transactions (all-or-none)
- **(C)**onsistency
    - Consistency before and after execution
- **(I)**solation
    - Execution of multiple transactions (no interference)
- **(D)**urability
    - Protection against system crashes

- Transactions requirements for **proper** execution

ACID properties in practice

- E.g. transfer $50 from account A to account B

    read(A)
    A:= A – 50
    write(A)
    read(B)
    B:= B+ 50
    write(B)

Transaction processing

- **Concurrency control**
  - make sure no interferences
  - typically uses "**locks**"
- **Crash recovery (crash do not destroy data)**
  - assure atomicity + durability
  - solution → use a "**log**"
  - logging based recovery
    - write a log entry for each transaction **before each commit**
    - after crash → analyze, redo, undo
  - performance → durability needs a write to disk for each commit
    - write all data to disk (possibly several writes per transaction)
    - append a log entry (one write for several transactions)

Concurrency control

- Support multiple transactions (parallelism)
    - hardware
    - software
- Advantages
    - improved throughput
    - resource utilization
    - reduced average response time
- Disadvantages
    - complicates the logic and the implementation

Concurrency control (Cont.)

- Concurrency control schemes
  - control the interaction among concurrent transactions to **prevent** them from destroying the **consistency** of the database (ACID)
- Serial execution of transactions preserves consistency
- **Schedule**
  - an ordering of the operations of several transactions
  - captures the main actions of transactions affecting concurrent execution
  - consider only read/write operations for simplicity

Schedules

Serial schedules

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

Schedules (Cont.)

- Not serial schedule but equivalent to first schedule
- In the three Schedules A+B is preserved, but the fourth

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- Basic assumption: each transaction preserves consistency
- Thus serial execution of a set of transactions preserves consistency
- A (possibly concurrent) **schedule** is **serializable** if it is equivalent to a serial schedule.
- Different forms of schedule equivalence
  - **conflict serializability** (easy to handle, but restrictive)
  - **view serializability** (hard to check and enforce, more general, not used much)
  - both look only at read/writes, not details of operations

Conflict serializability

- Conflict → any **two** operations in two **different** transactions that access the **same** item, and where at least **one** is a **write** operation
- Intuitively, a conflict between two operations forces a (logical) temporal order between them
- In case the two operations are consecutive in a schedule and they do not conflict, the results would remain the same even if they had been interchanged in the schedule

Conflict serializability (Cont.)

- **Conflict serializable** if it can be transformed into serial schedules without changing order of conflicting operation
- A schedule is conflict serializable if it is conflict equivalent to a serial schedule
- Schedule not conflict serializable since we can not swap the operations to obtain a serial schedule

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

Conflict serializability (Cont.)

- First schedule can be transformed into the second, a serial schedule where T2 follows T1, by a series of swaps of non-conflicting operations.
- First schedule → conflict serializable

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

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

View serializability

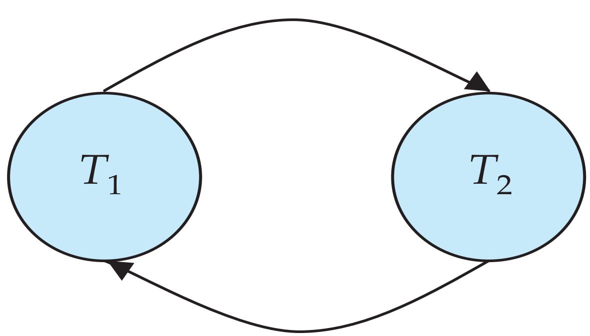
- Two schedules S and S' are view equivalent if the following three conditions are met for each data item Q
  - any transaction that reads the initial value in S also reads the initial value in S' (initial value read)
  - if Ti reads a value produced by Tj in S, it also does so in S' (producer/consumer – preserve order)
  - if Ti writes the final value of Q in S, it also does so in S' (last write)

View serializability (Cont.)

- A schedule is view serializable if it is view equivalent to a serial schedule
- Every conflict serializable schedule is also view serializable (but not the opposite)
- View equivalence between two schedules is easy to check
- View equivalence with a given serial schedule is also easy to check
- There are k! different serial schedules of k transactions
- In fact, problem is NP-complete

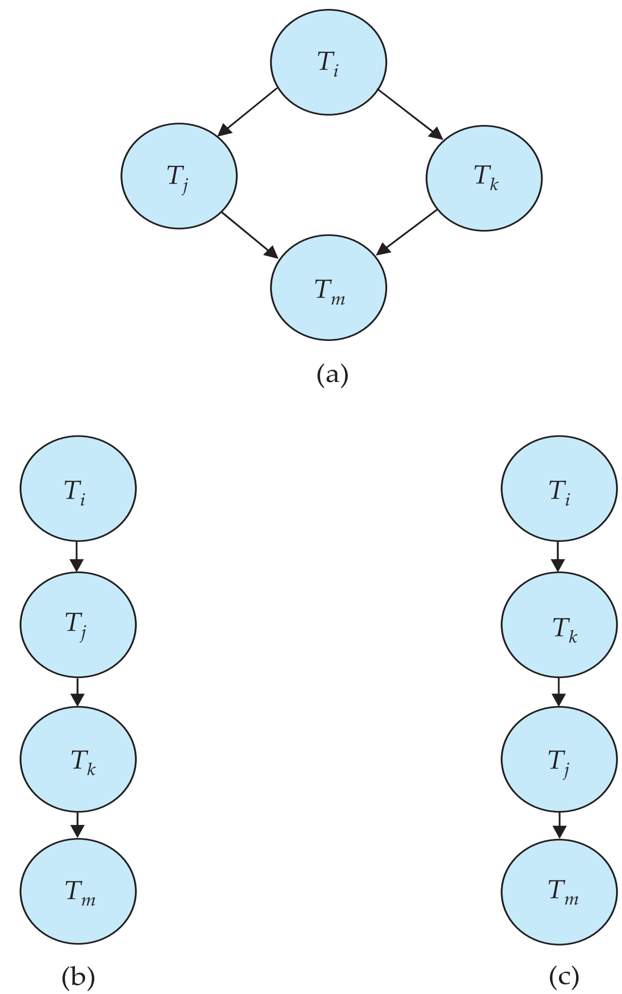
Testing serializability

- Consider some schedule of a set of transactions Ti
- Precedence graph → directed graph where vertices are the transactions
- There is an edge between Ti and Tj, if the two transactions conflict and Ti accessed the data item on which the conflict arose earlier
- Example



Test for conflict serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic
- Cycle detection algorithms takes O(n^2), where n the number of vertices in the graph
- In case of acyclic graph, the serializability order is obtained by a topological sorting of the graph

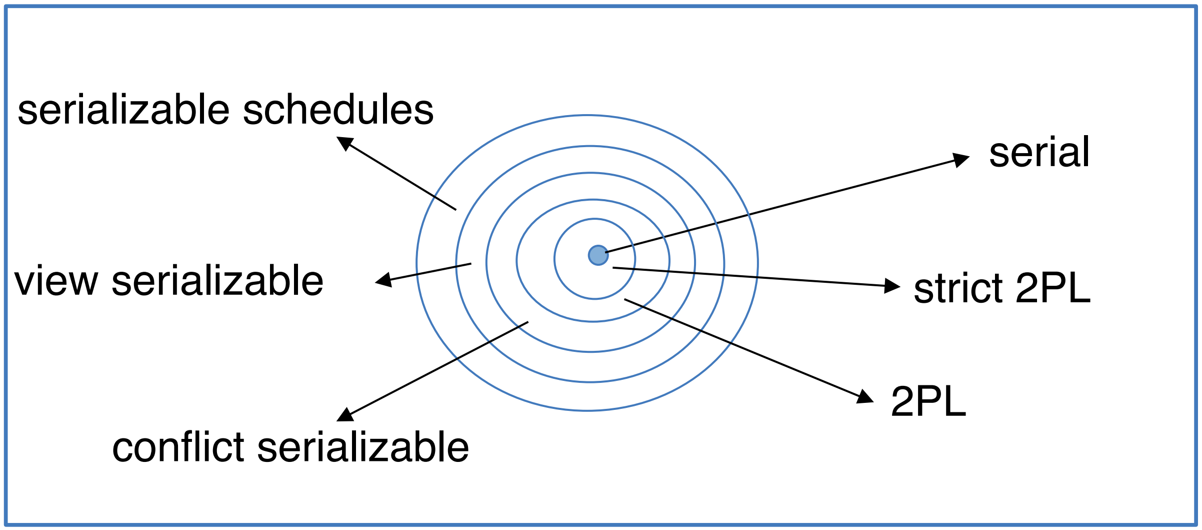


Test for view serializability

- The precedence graph test for conflict serializability can not be used directly to test for view serializability
  - Extension for view serializability has exponential cost in the size of the graph
  - Problem of checking if a schedule is view serializable falls in the class of NP-complete problems
    - Existence of an efficient algorithm is extremely unlikely
    - However practical algorithms that just check some sufficient conditions for view serializability can still be used

Relation between levels of serializability

- Suppose set of k transactions
- Set of all possible schedules



Concurrency control overview

- Serial execution
  - inefficient
- Strict 2-phase locking
  - lock based
  - most commonly used
- 2-Phase locking
  - lock based
  - may result in abort of committed transactions
- Optimistic concurrency control
  - timestamp based in distributed databases
- Concurrency protocols
  - enforce some form of serializability

conservative

aggressive

Lock-based protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes
    - **exclusive** (X) mode. Data item can be both **read** as well as **written** and the X-lock is requested with the **lock-X** instruction.
    - **shared** (S) mode. Data item can only be **read** and S-lock is requested using the **lock-S** instruction
- Concurrency-control manager grants locks
- Transactions proceed only when requested lock is granted

Lock-based protocols (Cont.)

- Lock-compatibility matrix
- 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

	S	X
S	true	false
X	false	false

- Any number of transactions can hold shared locks on an item **BUT** if any transaction holds an exclusive lock on the item **NO** other transaction may hold any lock on the item
- If a lock can not be granted, the requesting transaction waits until all incompatible locks held by other transactions have been released.

Lock-based protocols (Cont.)

- Example of transaction performing locking

    T2: **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.
- Display result is wrong if A, B gets updated in-between their reads
- 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 in Lock-based protocols

- Partial schedule
- Neither of transactions proceed
  - lock-S(B) causes the right transaction to wait to release its lock on B
  - lock-x(A) causes the left transaction to wait for the release of lock on A
- **Deadlock**
  - handling of deadlocks → one of the transactions must be rolled back and its locks released

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)	

Pitfalls in Lock-based protocols (Cont.)

- Deadlocks exist in most locking protocols
- **Starvation** is also possible if concurrency control manager is badly designed
    - a transaction may be waiting for an X-lock on an item, while a sequence of other transactions request are granted an S-lock on the same item
    - the same transaction (waiting for an X-lock) is repeatedly rolled back due to deadlocks
- Concurrency control manager can be designed to prevent starvation

The Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules
- Phase 1: Growing Phase
    - transaction may obtain locks, but may not release locks
- Phase 2: Shrinking Phase
    - transaction may release locks, but may not obtain any new locks
- The protocol assures serializibility
- It can be proved that the transactions can be serialized in the order of their lock points (when a transactions acquired its final lock)

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking does not ensure freedom from deadlocks
- Cascading roll-back is possible under this protocol, but can be avoided with a modified protocol called **Strict two-phase locking**
- Strict two-phase locking → a transaction must hold all its exclusive locks until it commits/aborts
- **Rigorous two-phase locking** is even stricter → all locks are held until commit/abort

Lock Conversions

- Two-phase locking with conversions
- First Phase
    - can acquire a lock-S
    - can acquire a lock-X
    - 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)
- This protocol assures serializibility, but still relies on the programmer to insert the various locking instructions

Automatic acquisition of Locks

- A transaction T issues the standard read/write operations **without** explicit locking calls
- The read(D) is processed as follows
  - if T has a lock on D
    - read(D)
  - else
    - if necessary wait until no other transaction has a lock-X on D
    - grant T a lock-S on D
    - read(D)

Automatic acquisition of Locks (Cont.)

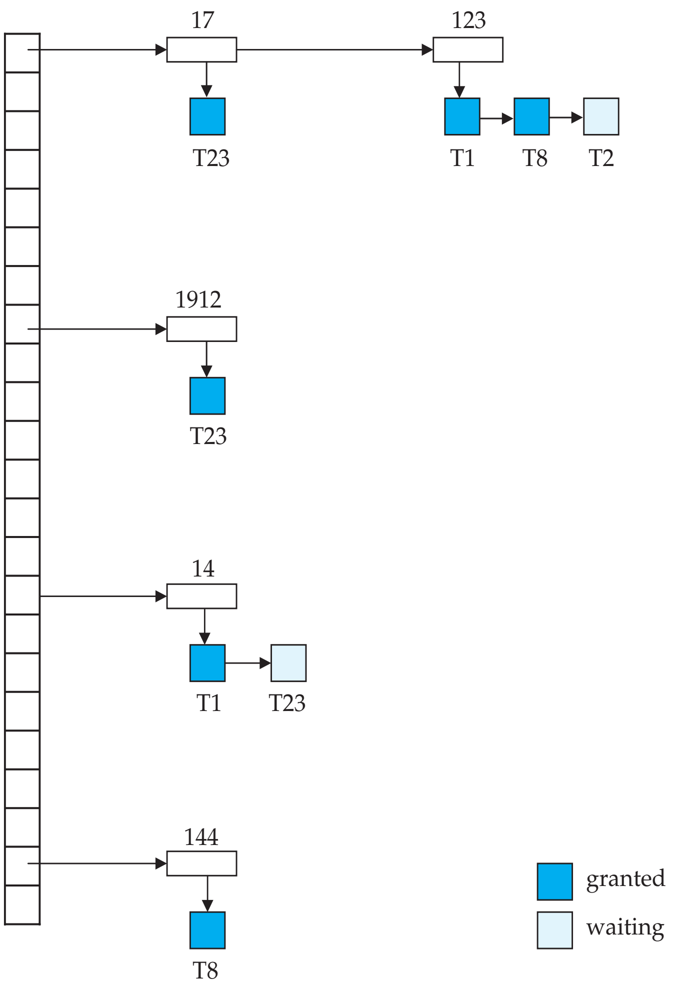
- The write(D) is processed as follows
  - if T has a lock-X on D
    - write(D)
  - else
    - if necessary wait until no other trans. has any lock on D
    - if T has a lock-S on D
      - upgrade lock on D to lock-X
    - else
      - grant T a lock-X on D
    - write(D)
- All locks are released after commit/abort

Lock implementation

- A lock manager can be implemented as a separate process to which transactions send lock/unlock requests
- The lock manager replies to a lock request by sending a lock grant message (or a message asking the transaction to roll-back in case of deadlock)
- The requesting transaction waits until its request is replied
- 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 being locked

Lock table

- Dark blue rectangle → granted lock
- Light blue rectangle → waiting request
- Lock table records type of lock
- New request is added at the end of the queue if it is compatible with all the 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 transactions are deleted (list of lock of each trans.)

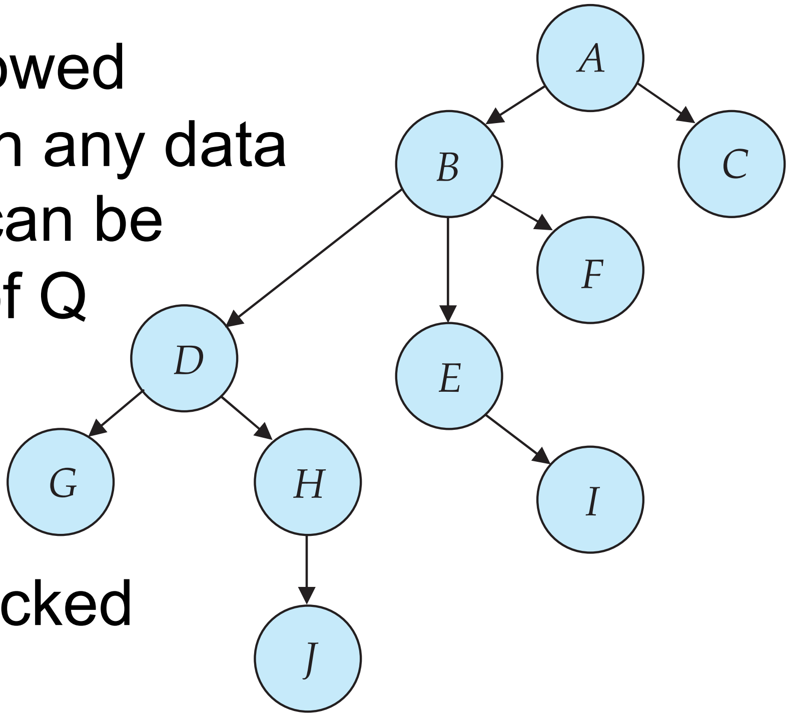


Graph-based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering on the set of all data items
  - if d → d' then any transaction accessing both d and d' must access d before accessing d'
  - implies that the entire set of all data items D may now be viewed as a directed acyclic graph, called a database graph
- The tree-protocol is a simple kind of graph protocol

Tree protocol

- Only exclusive locks are allowed
- The first lock by T may be on any data item. Subsequently, a data Q can be locked by T only if the parent of Q is currently locked by T
- Data items may be unlocked at any time
- A data item that has been locked and unlocked by T cannot subsequently be relocked by T



Tree protocol (Cont.)

- The tree protocol ensures conflict serializability as well as deadlock freedom
- Unlocking may occur earlier in this protocol compared to the two-phase locking
    - shorter waiting times and increase in concurrency
    - no roll-backs are required and protocol is deadlock-free
- Drawbacks
    - protocol does no guarantee recoverability or cascade freedom → need to ensure commit dependencies to ensure recoverability
    - transactions may have to lock data items that they do not access → increased locking overhead and additional waiting time and potential decrease in concurrency

Deadlock handling

- Consider the following two transactions

  T: write(A)   T': 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 (Cont.)

- 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 prevention protocols ensure that the system will **never** enter into a deadlock state
- Deadlock 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

- Use of transaction timestamps
- **Wait-die** scheme (non-preemptive)
  - older transaction may wait for a 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

More deadlock prevention strategies (Cont.)

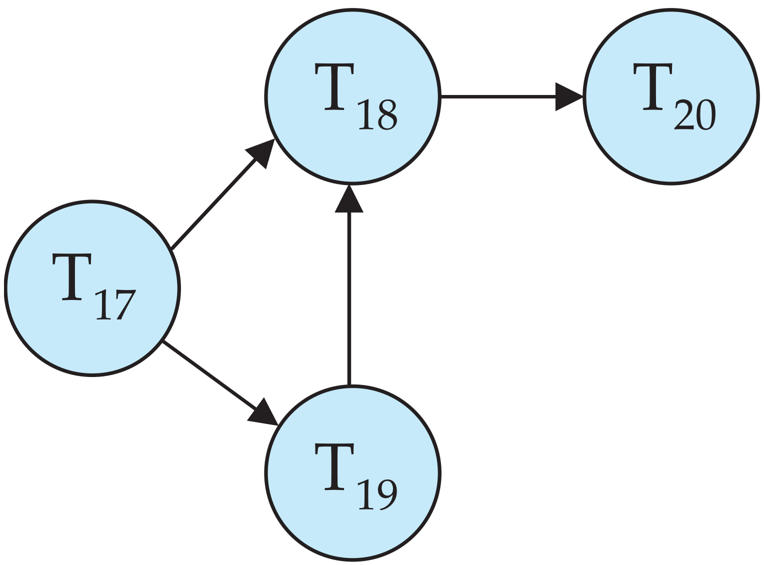
- Both in wait-die and wound-wait schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided
- Timeout-Based schemes
  - a transaction waits for a lock only for a specified amount of time. After 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

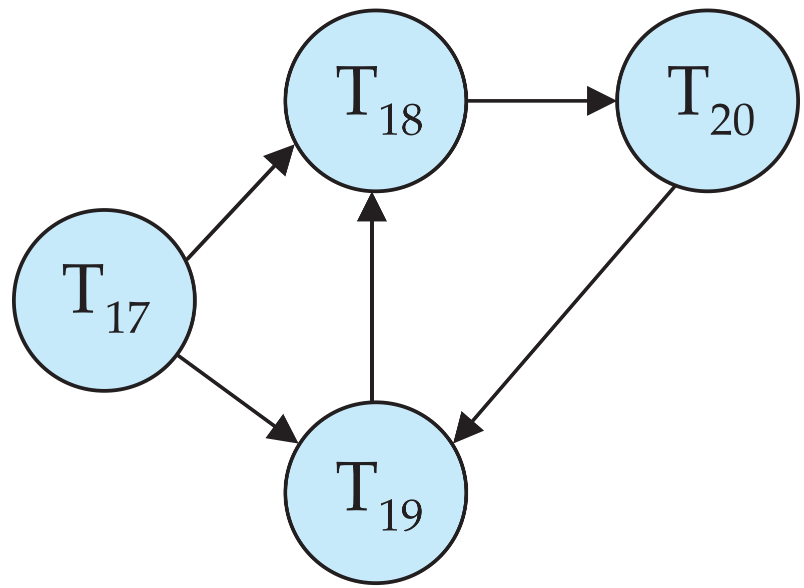
- Deadlocks can be described as a wait-for graph, which consists of a pair G=(V, E),
  - V is a set of vertices (all transactions of the system)
  - E is a set of edges, each element is an ordered pair $T \rightarrow T'$
- If $T \rightarrow T'$ is in E, then there is a directed edge from T to T', implying that T is waiting for T' to release a data item
- When T requests a data item currently being held by T', then the edge $T \rightarrow T'$ is inserted in the wait-for graph. This edge is removed only when T' is no longer holding a data item needed by T
- The system is in a deadlock state iff 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 detection (Cont.)

- 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