Concurrency Control

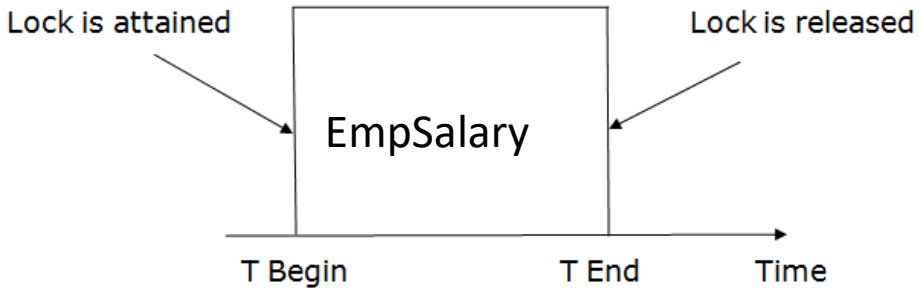
Concurrency Control

- One of the fundamental properties of a transaction is isolation.
- When several transactions execute concurrently in the database, to ensure that it is the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called *concurrency control* schemes.
- There are a variety of concurrency-control schemes.
- No one scheme is clearly the best; each one has advantages.
- In practice, the most frequently used schemes are *two-phase locking* and *snapshot isolation*.

Concurrency Control

- **Lock-Based Protocols**
- One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item.

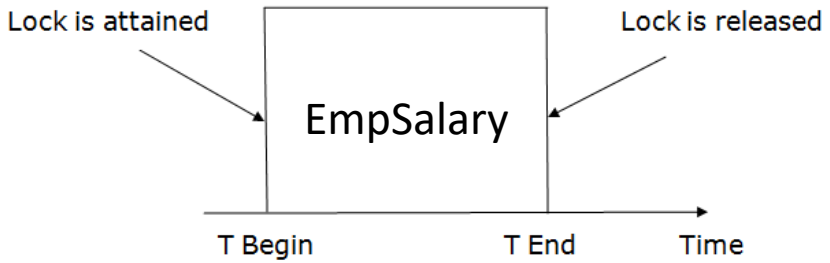
- The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item.



Concurrency Control

Locks

There are various modes in which a data item may be locked. We restrict our attention to two modes:



1. **Shared**. If a transaction *Ti* has obtained a **shared-mode lock** (denoted by S) on item *Q*, then *Ti* can read, but cannot write, *Q*.

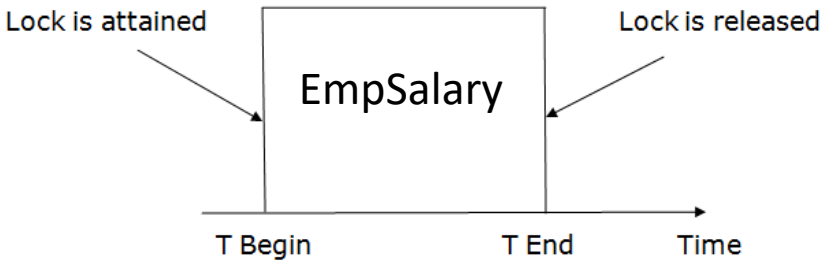
2. **Exclusive**. If a transaction *Ti* has obtained an **exclusive-mode lock** (denoted by X) on item *Q*, then *Ti* can both read and write *Q*.

- Each Transaction (T) **request** a lock in an appropriate mode on data item *Q*, depending on the types of operations that it will perform on *Q*.
- The transaction makes the request to the concurrency-control manager.
- The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction.
- The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

Concurrency Control

	S	X
S	true	false
X	false	false

Lock-compatibility matrix comp.



Given a set of lock modes, we can define a **compatibility function** on them as follows:

(1) Let *A* and *B* represent arbitrary lock modes.

(2) Suppose that a transaction *Ti* requests a lock of mode *A* on item *Q* on which transaction *Tj* currently holds a lock of mode *B*.

(3) If transaction *Ti* can be granted a lock on *Q* immediately, in spite of the presence of the mode *B* lock, then we say mode *A* is **compatible** with mode *B*.

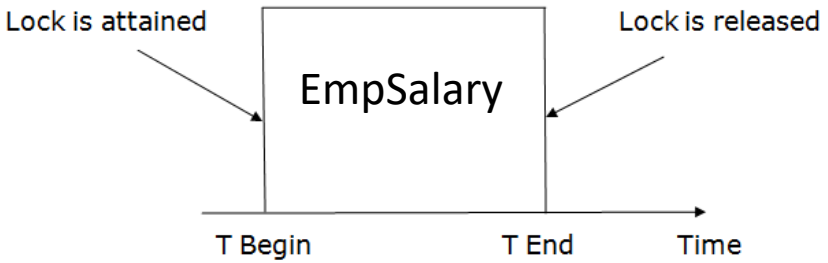
Such a function can be represented conveniently by a matrix.

An element comp(*A, B*) of the matrix has the value *true* if and only if mode *A* is compatible with mode *B*.

Concurrency Control

	S	X
S	true	false
X	false	false

Lock-compatibility matrix comp.



Note that shared mode is compatible with shared mode, but not with exclusive mode.

At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item.

A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

- A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction.
- Similarly, a transaction requests an exclusive lock through the lock-X(Q) instruction.
- A transaction can unlock a data item Q by the unlock(Q) instruction.

Concurrency Control

- To access a data item, transaction *Ti* must first lock that item.
- If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released.
- Thus, *Ti* is made to **wait** until all incompatible locks held by other transactions have been released.
- Transaction *Ti* may unlock a data item that it had locked at some earlier point.
- Note that a transaction must hold a lock on a data item as long as it accesses that item.
- Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured.

banking example

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

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

Concurrency Control

- Let A and B be two accounts that are accessed by transactions $T1$ and $T2$.
- Transaction $T1$ transfers \$50 from account B to account A.
- Transaction $T2$ displays the total amount of money in accounts A and B—that is, the sum $A + B$
- Suppose that the values of accounts A and B are \$100 and \$200, respectively. If these two transactions are executed serially, either in the order $T1$, $T2$ or the order $T2$, $T1$, then transaction $T2$ will display the value \$300.
- If, however, these transactions are executed concurrently, as shown in schedule 1, transaction **$T2$ displays \$250**, which is incorrect.
- The reason for this mistake is that the transaction **$T1$ unlocked data item B too early**, as a result of which $T2$ saw an inconsistent state.

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

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

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Concurrency Control

- Suppose now that unlocking is delayed to the end of the transaction.
- Transaction $T3$ corresponds to $T1$ with unlocking delayed.
- Transaction $T4$ corresponds to $T2$ with unlocking delayed.

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

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

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

Transaction T_3 (transaction T_1 with unlocking delayed).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

Transaction T_4 (transaction T_2 with unlocking delayed).

- Now you can verify that if the schedule is prepared with T3 and T4, then T4 will never display incorrect result.

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Concurrency Control

- Unfortunately, locking can lead to an undesirable situation.
- Consider the partial schedule for $T3$ and $T4$.
- T3 is holding an exclusive mode lock on B and $T4$ is requesting a shared-mode lock on B, $T4$ is waiting for $T3$ to unlock B. Similarly, since $T4$ is holding a shared-mode lock on A and $T3$ is requesting an exclusive-mode lock on A, $T3$ is waiting for $T4$ to unlock A.
- Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution.
- This situation is called **deadlock**.
- When deadlock occurs, the system must roll back one of the two transactions.
- Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked.
- These data items are then available to the other transaction, which can continue with its execution.

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)	

- If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states.
- On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur.
- **Deadlocks are a necessary evil** associated with locking, if we want to avoid inconsistent states.

Concurrency Control

- To take care of deadlock situations
- We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items.

- we introduce some terminology
- Let $\{T0, T1, \ldots, Tn\}$ be a set of transactions participating in a schedule S.

- **We say that *Ti* precedes *Tj* in *S*, written *Ti* → *Tj*, if there exists a data item *Q* such that *Ti* has held lock mode *A* on *Q*, and *Tj* has held lock mode *B* on *Q* later, and comp(*A*,*B*) = false.**

- If $Ti \rightarrow Tj$, then that precedence implies that in any equivalent serial schedule, *Ti* must appear before *Tj* .

- Observe that this concept is similar to the precedence graph that we used to test for conflict serializability.
- Conflicts between instructions correspond to noncompatibility of lock modes.

Concurrency Control

- We say that a schedule S is **legal** under a given locking protocol if S is a possible schedule for a set of transactions that follows the rules of the locking protocol.

- We say that a locking protocol **ensures** conflict serializability if and only if all legal schedules are conflict serializable;

Granting of Locks

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.

However, care must be taken to avoid the following scenario. Suppose a transaction $T2$ has a shared-mode lock on a data item, and another transaction $T1$ requests an exclusive-mode lock on the data item.
Clearly, $T1$ has to wait for $T2$ to release the shared-mode lock.

Meanwhile, a transaction $T3$ may request a shared-mode lock ….$T3$ is granted .. it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item ….$T1$ never gets the exclusive-mode lock on the data item.

The transaction $T1$ may never make progress, and is said to be **starved**.

Concurrency Control

Granting of Locks – (1)

- We can avoid starvation of transactions by granting locks in the following manner: When a transaction *Ti* requests a lock on a data item *Q* in a particular mode *M*, the concurrency-control manager grants the lock provided that:

(1) There is no other transaction holding a lock on *Q* in a mode that conflicts with *M*.

(2) There is no other transaction that is waiting for a lock on *Q* and that made its lock request before *Ti* .

Thus, a lock request will never get blocked by a lock request that is made later.

Granting of Locks – (2)

- **The Two-Phase Locking Protocol**

- This protocol requires that each transaction issue lock and unlock requests in two phases:
  1. **Growing phase**. A transaction may obtain locks, but may not release any lock.
  2. **Shrinking phase**. A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

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

(not following two-phase locking)

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

(following two-phase locking)

Concurrency Control

- The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction.

```
T3: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

- Now, transactions can be ordered according to their lock points— this ordering is, in fact, a serializability ordering for the transactions.

- Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions *T*3 and *T*4 are two phase, but, in schedule 2 they are deadlocked.

Concurrency Control

- **Strict two-phase locking protocol**
- Cascading rollbacks is avoided.
- all exclusive-mode locks taken by a transaction be held until that transaction commits, preventing any other transaction from reading the data.

- **Rigorous two-phase locking protocol**
- all locks be held until the transaction commits

Concurrency Control

- Consider these two transactions, for which we have shown only some of the significant read and write operations.
- If we employ the two-phase locking protocol, then $T8$ must lock $a1$ in exclusive mode.
- Therefore, any concurrent execution of both transactions amounts to a serial execution.
- Notice, however, that $T8$ needs an exclusive lock on $a1$ only at the end of its execution, when it writes $a1$.
- Thus, if $T8$ could initially lock $a1$ in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since $T8$ and $T9$ could access $a1$ and $a2$ simultaneously.

T_8: read(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

lock conversions - a mechanism for upgrading a shared lock to an exclusive lock (upgrading), and downgrading an exclusive lock to a shared lock (downgrading)

Uupgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

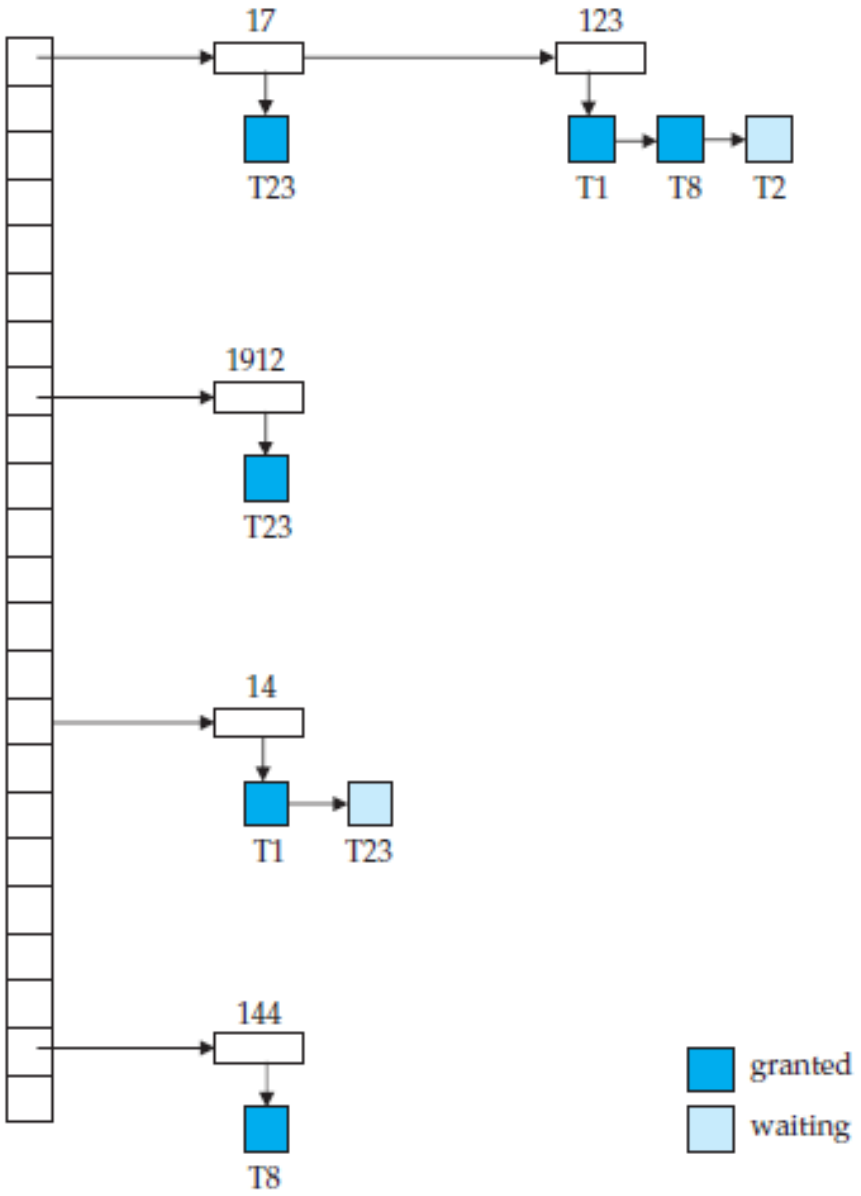
Concurrency Control

- **Implementation of Locking**

- A **lock manager** can be implemented as a process that receives messages from transactions and sends messages in reply.
  - The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks).
  - Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.

- Lock Manager's DS
  (1) For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived.
  (2) It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the **lock table**.
  (3) Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested.
  (4) The record also notes if the request has currently been granted.

Concurrency Control

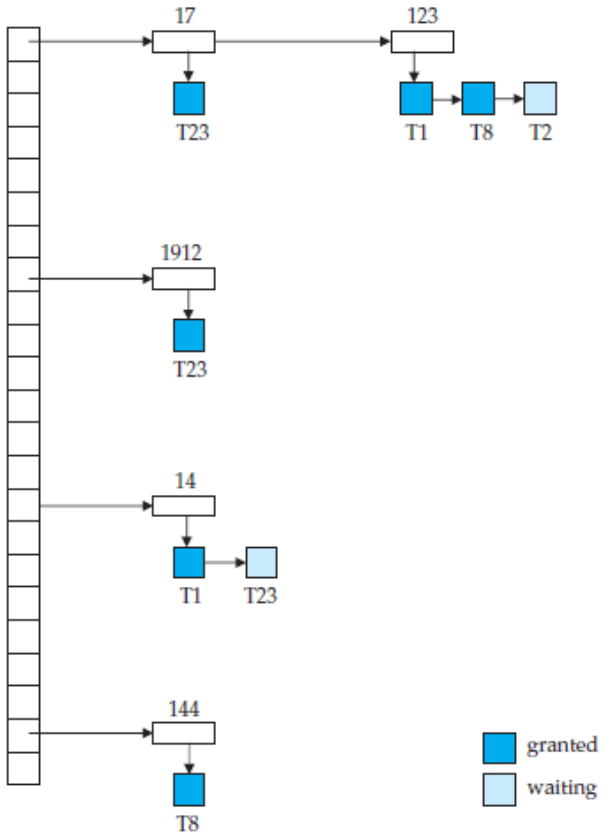


- This is an example of a lock table.
- The table contains locks for five different data items, I4, I7, I23, I44, and I912.
- The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table.
- There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items.

(omitted the lock mode to keep the figure simple).

Example, T23 has been granted locks on I912 and I7, and is waiting for a lock on I4.

Concurrency Control



- When **a lock request message arrives**, it adds a record to the end of the linked list for the data item, if the linked list is present.
- Otherwise it creates a new linked list, containing only the record for the request.
- It always grants a lock request on a data item that is not currently locked.
- But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already.
- Otherwise the request has to wait.

- When the lock manager **receives an unlock message** from a transaction, it deletes the record.
- It tests the record that follows, if any, to see if that request can now be granted.
- If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.

- If **a transaction aborts**, the lock manager deletes any waiting request made by the transaction.

- Once the database system has taken appropriate actions to undo the transaction, it releases all locks held by the aborted transaction.

Deadlock Handling

(Concurrency Control – Transaction Management)

Deadlock Handling

- A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.

- More precisely, there exists a set of waiting transactions $\{T0, T1, \ldots, Tn\}$ such that $T0$ is waiting for a data item that $T1$ holds, and $T1$ is waiting for a data item that $T2$ holds, and . . . , and $Tn-1$ is waiting for a data item that Tn holds, and Tn is waiting for a data item that $T0$ holds.

- None of the transactions can make progress in such a situation.
- The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock.

- Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.

Deadlock Handling

- There are two principal methods for dealing with the deadlock problem.

- We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state.

- Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme.

Deadlock Handling

Deadlock Prevention

- There are two approaches to deadlock prevention.

(1) One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together.

(2) The other approach is closer to deadlock recovery, and performs transaction rollback instead of waiting for a lock, whenever the wait could potentially result in a deadlock.

The **simplest scheme** under the first approach requires that each transaction locks all its data items before it begins execution.

Moreover, either all are locked in one step or none are locked.

There are two main disadvantages to this protocol:

(1) it is often hard to predict, before the transaction begins, what data items need to be locked;

(2) data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

Deadlock Handling

Deadlock Prevention

- Another approach for preventing deadlocks is to **impose an ordering of all data items**, and to require that a transaction lock data items only in a sequence consistent with the ordering.

- Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering.

- This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution.

Deadlock Handling

Deadlock Prevention

- Another approach for preventing deadlocks is to use preemption and transaction rollbacks.
- In preemption, when a transaction *Tj* requests a lock that transaction *Ti* holds, the lock granted to *Ti* may be **preempted** by rolling back of *Ti* , and granting of the lock to *Tj* .
- To control the preemption, we assign a unique timestamp, based on a counter or on the system clock, to each transaction when it begins.
- The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control.
- If a transaction is rolled back, it retains its *old* timestamp when restarted.

Deadlock Handling

- Two different deadlock-prevention schemes using timestamps have been proposed: **wait–die and wound–wait**

- The **wait–die** scheme is a **nonpreemptive technique**. When transaction *Ti* requests a data item currently held by *Tj* , *Ti* is allowed to **wait** only if it has a timestamp smaller than that of *Tj* (that is, *Ti* is older than *Tj*).
- Otherwise, *Ti* is rolled back (**dies**).
- For example, suppose that transactions *T*14, *T*15, and *T*16 have timestamps 5, 10, and 15, respectively. If *T*14 requests a data item held by *T*15, then *T*14 will wait. If *T*24 requests a data item held by *T*15, then *T*16 will be rolled back.

- The **wound–wait** scheme is **a preemptive technique**. It is a counterpart to the wait–die scheme. When transaction *Ti* requests a data item currently held by *Tj* , *Ti* is allowed to wait only if it has a timestamp larger than that of *Tj* (that is, *Ti* is younger than *Tj*). Otherwise, *Tj* is rolled back (*Tj* is *wounded* by *Ti*).

- Returning to our example, with transactions *T*14, *T*15, and *T*16, if *T*14 requests a data item held by *T*15, then the data item will be preempted from *T*15, and *T*15 will be rolled back. If *T*16 requests a data item held by *T*15, then *T*16 will wait.

- Another simple approach to deadlock prevention is based on **lock timeouts**.

- In this approach, a transaction that has requested a lock waits for at most a specified amount of time.

- If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts.

Deadlock Handling

- **Deadlock Detection and Recovery**

- An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock.

- To do so, the system must:

1. Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.

2. Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.

3. Recover from the deadlock when the detection algorithm determines that a deadlock exists.

Deadlock Handling

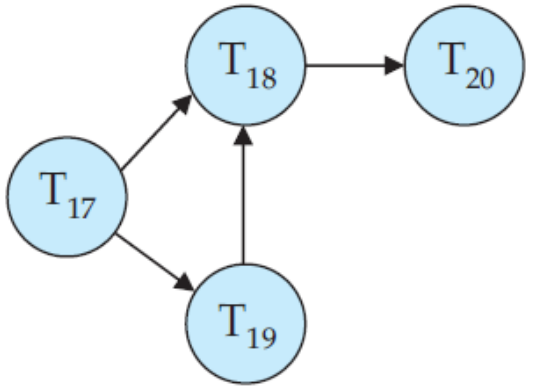
Deadlock Detection

- Deadlocks can be described precisely in terms of a directed graph called a **wait for graph**.
  - This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges.
  - The set of vertices consists of all the transactions in the system.
  - Each element in the set E of edges is an ordered pair $Ti \rightarrow Tj$. If $Ti \rightarrow Tj$ is in E, then there is a directed edge from transaction Ti to Tj , implying that transaction Ti is waiting for transaction Tj to release a data item that it needs.
- When transaction Ti requests a data item currently being held by transaction Tj , then the edge $Ti \rightarrow Tj$ is inserted in the wait-for graph. This edge is removed only when transaction Tj is no longer holding a data item needed by transaction Ti .
- A deadlock exists in the system if and only if the wait-for graph contains a cycle.

Deadlock Handling

- To illustrate these concepts, consider the wait-for graph given here, which depicts the following situation:
  - Transaction *T*17 is waiting for transactions *T*18 and *T*19.
  - Transaction *T*19 is waiting for transaction *T*18.
  - Transaction *T*18 is waiting for transaction *T*20.

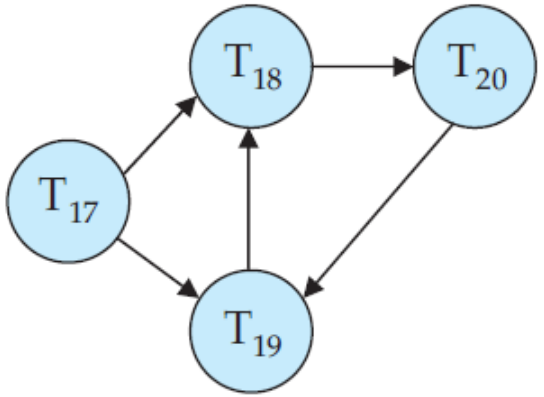
Since the graph has no cycle, the system is not in a deadlock state.



Wait-for graph with no cycle.

Suppose now that transaction *T*20 is requesting an item held by *T*19.

- The edge *T*20 → *T*19 is added to the wait-for graph, resulting in the new system state as given in the figure.
- This time, the **graph contains the cycle**: *T*18 →*T*20 →*T*19 →*T*18 implying that transactions *T*18, *T*19, and *T*20 are all deadlocked.



Wait-for graph with a cycle.

Deadlock Handling

- **Recovery from Deadlock**

- The most common solution is to roll back one or more transactions to break the deadlock.

- Three actions need to be taken:

(1) Selection of a victim. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back.

- We should roll back those transactions that will incur the minimum cost. Many factors may determine the cost of a rollback, including:

a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.

b. How many data items the transaction has used.

c. How many more data items the transaction needs for it to complete.

d. How many transactions will be involved in the rollback.

Deadlock Handling

(2) Rollback. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.

- The simplest solution is a **total rollback**: Abort the transaction and then restart it.
- However, it is more effective to roll back the transaction only as far as necessary to break the deadlock.
- Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions.

(3) Starvation. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**.

We must ensure that a transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

The End…