# Concurrency Control

- **Concurrency Control**
- Lock-Based Concurrency Control Protocols
- Two-Phase Locking Protocol
- Deadlock Handling

# Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- **Goal** – to develop concurrency control protocols that will assure serializability.
- Concurrency-control protocols tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.

# Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Testing a schedule for serializability *after* it has executed is a little too late!
  - Instead concurrency control protocols are more efficient by imposing a discipline that avoids non-serializable schedules.
- Tests for serializability help us understand why a concurrency control protocol is correct.

# Concurrency Control

- Concurrency Control
- **Lock-Based Concurrency Control Protocols**
- Two-Phase Locking Protocol
- Deadlock Handling

# Lock-Based Protocols

- It is required that data items are accessed in a mutually exclusive manner, i.e., while one transaction is accessing a data item, no other transaction can modify that data item (***isolation***).
- A lock is a mechanism to control concurrent access to a data item, i.e., a transaction is allowed to access a data item only if it is currently holding a lock on that 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.

# 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 lock 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.

# Schedule With Lock Grants

- Assume grant happens just before the next instruction following lock request
  - Grants omitted in rest of chapter
- This schedule is not serializable (why?)
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols enforce serializability by restricting the set of possible schedules.

| $T_1$ | $T_2$ | concurrency-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($A$) | | |

# Deadlock

- 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.

# Deadlock (Cont.)

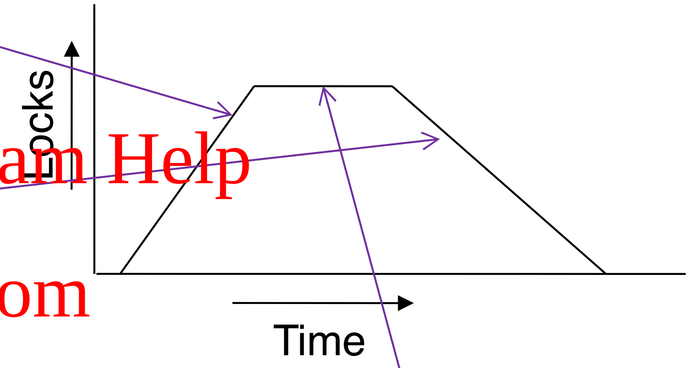
- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and 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.

# Concurrency Control

- Concurrency Control
- Lock-Based Concurrency Control Protocols
- **Two-Phase Locking Protocol**
- Deadlock Handling

# The Two-Phase Locking Protocol

- 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).

# The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure recoverability or freedom from cascading roll-back
  - **Strict two-phase locking:** a transaction must hold all its exclusive locks till it commits/aborts.
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous two-phase locking**: a transaction must hold *all* locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
- Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*

# The Two-Phase Locking Protocol (Cont.)

- Two-phase locking is not a necessary condition for serializability
  - There are conflict serializable schedules that cannot be obtained if the two-phase locking protocol is used.
- In the absence of extra information (e.g., ordering of access to data), two-phase locking is necessary for conflict serializability *in the following sense:*
  - *Given a transaction $T_i$ that does not follow two-phase locking, we can find a transaction $T_j$ that uses two-phase locking, and a schedule for $T_i$ and $T_j$ that is not conflict serializable.*

| $T_1$ | $T_2$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| unlock($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | unlock($A$) |
| | lock-S($B$) |
| | read($B$) |
| | unlock($B$) |
| | display($A + B$) |
| lock-X($A$) | |
| read($A$) | |
| $A := A + 50$ | |
| write($A$) | |
| unlock($A$) | |

# Locking Protocols

- Given a locking protocol (such as 2PL)
  - A schedule S is **legal** under a locking protocol if it can be generated by a set of transactions that follow the protocol
  - A protocol **ensures** serializability if all legal schedules under that protocol are serializable

# Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests as messages.
- 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 a deadlock).



- The requesting transaction waits until its request is answered.

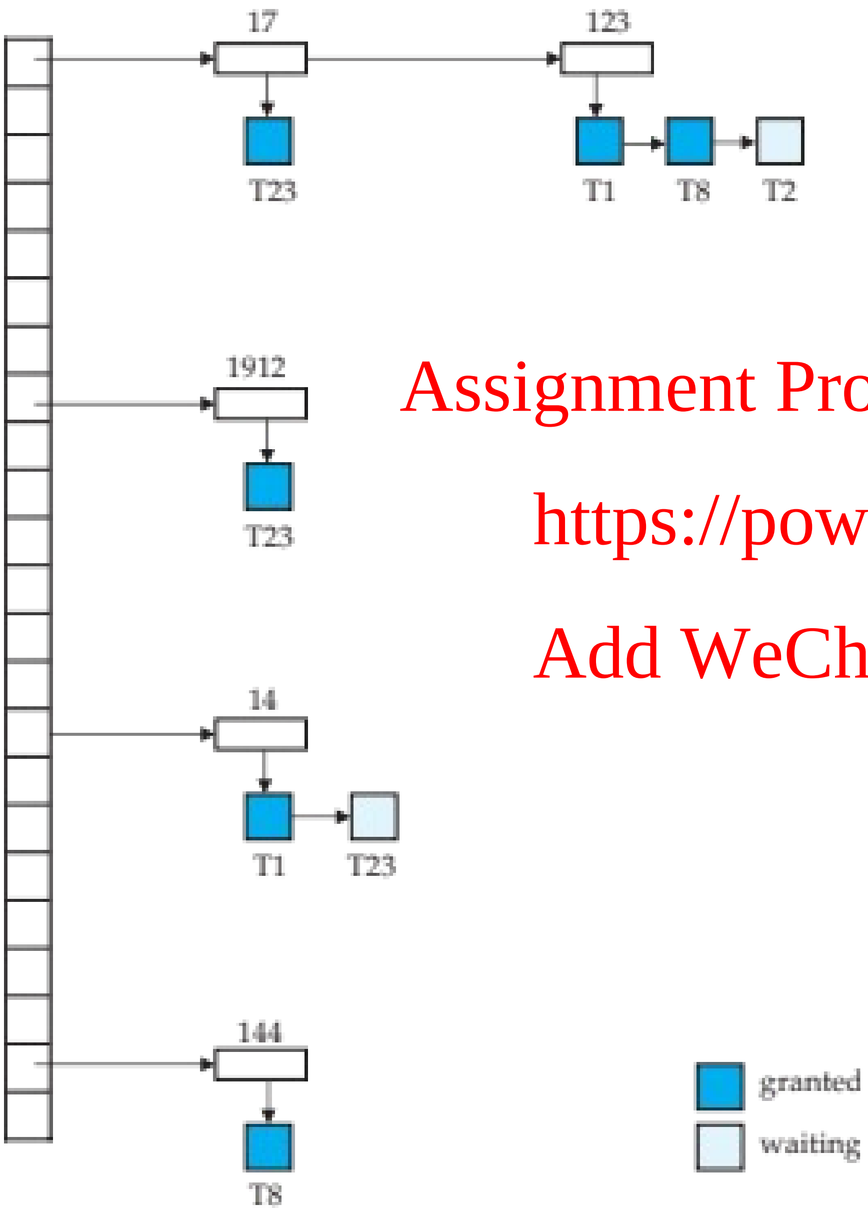


- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests.



- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.

# Lock Table




- Dark blue rectangles indicate granted locks, light 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 the data item is not currently locked or 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

# Transactions and Concurrency Control

- Concurrency Control
- Lock-Based Concurrency Control Protocols
- Two-Phase Locking Protocol
- **Deadlock Handling**

# 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.

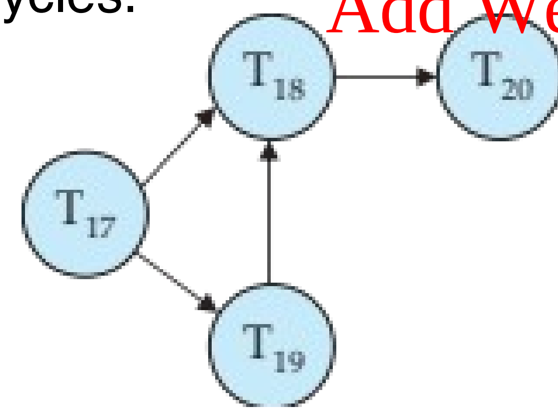


| $T_3$ | $T_4$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | lock-S($B$) |
| lock-X($A$) | |

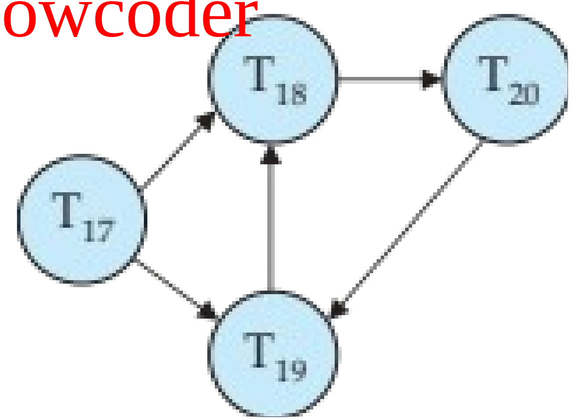
# Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from* $T_i \rightarrow T_j$. : if $T_i$ is waiting for a lock held in conflicting mode by $T_j$
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.






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 cycle.
    - 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.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim