Chapter 16 : 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. 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

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>X</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

- 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 a transaction performing locking:

\[ T_2 : \text{lock-S}(A); \]

\[ \text{read } (A); \]

\[ \text{unlock}(A); \]

\[ \text{lock-S}(B); \]

\[ \text{read } (B); \]

\[ \text{unlock}(B); \]

\[ \text{display}(A+B) \]

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

Consider the partial schedule

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X($B$)</td>
<td>lock-S($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$B := B - 50$</td>
<td>lock-S($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td></td>
</tr>
</tbody>
</table>

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.
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

- 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).
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.

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.
There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.

However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed 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.
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)

- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.
Automatic Acquisition of Locks

- A transaction $T_i$ issues the standard read/write instruction, without explicit locking calls.
- The operation $\text{read}(D)$ is processed as:
  
  if $T_i$ has a lock on $D$
  
  then
  
  read($D$)
  
  else begin
  
  if necessary wait until no other transaction has a lock-X on $D$
  
  grant $T_i$ a lock-S on $D$;
  
  read($D$)
  
  end
Automatic Acquisition of Locks (Cont.)

- **write**\((D)\) is processed as:
  
  \[
  \text{if } T_i \text{ has a lock-X on } D \\
  \text{then} \\
  \quad \text{write}(D) \\
  \text{else begin} \\
  \quad \text{if necessary wait until no other trans. has any lock on } D, \\
  \quad \text{if } T_i \text{ has a lock-S on } D \\
  \quad \text{then} \\
  \quad \quad \text{upgrade lock on } D \text{ to lock-X} \\
  \quad \text{else} \\
  \quad \quad \text{grant } T_i \text{ a lock-X on } D \\
  \quad \text{write}(D) \\
  \text{end;}
  \]

- All locks are released after commit or abort
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.
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.
Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set $D = \{d_1, d_2, \ldots, d_h\}$ 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.
1. Only exclusive locks are allowed.
2. The first lock by $T_i$ may be on any data item. Subsequently, a data $Q$ can be locked by $T_i$ only if the parent of $Q$ is currently locked by $T_i$.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by $T_i$ cannot subsequently be relocked by $T_i$. 
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

- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do 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.
Consider the following two transactions:

\[ T_1: \text{write (X)} \quad T_2: \text{write (Y)} \]

\[ \text{write (Y)} \quad \text{write (X)} \]

Schedule with deadlock

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lock-X</strong> on X</td>
<td><strong>lock-X</strong> on Y</td>
</tr>
<tr>
<td>write (X)</td>
<td>write (X)</td>
</tr>
<tr>
<td>wait for <strong>lock-X</strong> on Y</td>
<td>wait for <strong>lock-X</strong> on X</td>
</tr>
</tbody>
</table>
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 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

- 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 \textit{wait-die} and in \textit{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.
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 \( T_i \rightarrow T_j \) 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 (Cont.)

- Wait-for graph without a cycle
- Wait-for graph with a cycle
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.