Concurrency Control

- Concurrency Control
  - Ensures interleaving of operations amongst concurrent transactions result in serializable schedules

- How?
  - transaction operations interleaved following a protocol
How to enforce serializable schedules?

Prevent P(S) cycles from occurring using a concurrency control manager: ensures interleaving of operations amongst concurrent transactions only result in serializable schedules.

\[ T_1 \ T_2 \ \ldots \ \ldots \ T_n \]

CC Scheduler

DB
Concurrent Via Locks

Idea:
- Data items modified by one transaction at a time

Locks
- Control access to a resource
  - Can block a transaction until lock granted
- Two modes:
  - Shared (read only)
  - eXclusive (read & write)
Granting Locks

■ Requesting locks
  ▪ Must request before accessing a data item

■ Granting Locks
  ▪ No lock on data item? Grant
  ▪ Existing lock on data item?
    ▪ Check compatibility:
      ▪ Compatible? Grant
      ▪ Not? Block transaction

<table>
<thead>
<tr>
<th></th>
<th>shared</th>
<th>exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>exclusive</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</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>unlock(A)</td>
</tr>
<tr>
<td>write(B)</td>
<td>lock-S(B)</td>
</tr>
<tr>
<td>unlock(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>lock-X(A)</td>
<td>unlock(B)</td>
</tr>
<tr>
<td>read(A)</td>
<td>display(A+B)</td>
</tr>
<tr>
<td>A ← A + 50</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>unlock(A)</td>
<td></td>
</tr>
</tbody>
</table>
Locking Issues

- **Starvation**
  - T1 holds shared lock on Q
  - T2 requests exclusive lock on Q: blocks
  - T3, T4, ..., Tn request shared locks: granted
  - T2 is starved!

- **Solution?**

  Do not grant locks if older transaction is waiting
No transaction proceeds:

Deadlock
- T1 waits for T2 to unlock A
- T2 waits for T1 to unlock B

Rollback transactions
Can be costly...
Locking Issues

- Locks do not ensure serializability by themselves:

  T1
  lock-X(B)
  read(B)
  B ← B-50
  write(B)
  unlock(B)

  T2
  lock-S(A)
  read(A)
  unlock(A)
  lock-S(B)
  read(B)
  unlock(B)
  display(A+B)

  T2 displays 50 less!!
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). Locks can be either X, or S/X.
Example: T1 in 2PL

Growing phase
- lock-X(B)
- read(B)
- $B \leftarrow B - 50$
- write(B)
- lock-X(A)
- read(A)
- $A \leftarrow A - 50$
- write(A)

Shrinking phase
- unlock(B)
- unlock(A)
2PL & Serializability

- Recall: Precedence Graph

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read(Q)</td>
<td>write(Q)</td>
<td>write(R)</td>
</tr>
<tr>
<td></td>
<td>read(R)</td>
<td>read(R)</td>
<td>read(S)</td>
</tr>
</tbody>
</table>

![Graph Diagram](image)
## 2PL & Serializability

- **Recall:** Precedence Graph

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(Q)</td>
<td></td>
<td>write(Q)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(S)</td>
<td>read(R)</td>
<td>write(R)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>read(S)</td>
</tr>
</tbody>
</table>

Cycle $\rightarrow$ Non-serializable
2PL & Serializability

Relation between Growing & Shrinking phase:

\[ T_1 G < T_1 S \]
\[ T_2 G < T_2 S \]
\[ T_3 G < T_3 S \]

T1 must release locks for other to proceed

\[ T_1 S < T_2 G \]
\[ T_2 S < T_3 G \]
\[ T_3 S < T_1 G \]

\[ T_1 G < T_1 S < T_2 G < T_2 S < T_3 G < T_3 S < T_1 G \]

Not Possible under 2PL!

It can be generalized for any set of transactions...
2PL Issues

- As observed earlier, 2PL does not prevent deadlock.
- > 2 transactions involved?
  - Rollbacks expensive.
- We will revisit later.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</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>write(B)</td>
</tr>
<tr>
<td>lock-X(A)</td>
<td>lock-S(B)</td>
</tr>
</tbody>
</table>
2PL Variants

Strict two phase locking

- Exclusive locks must be held until transaction commits

- Ensures data written by transaction can’t be read by others

- Prevents cascading rollbacks
### Strict 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A)</td>
<td>lock-X(A)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(A)</td>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>lock-S(B)</td>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>unlock(A)</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock(A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<.action fails>
Strict 2PL & Cascading Rollbacks

- Ensures any data written by uncommitted transaction not read by another

- Strict 2PL would prevent T2 and T3 from reading A
  - T2 & T3 wouldn’t rollback if T1 does
Deadlock Handling

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></th>
<th>(T_1)</th>
<th>(T_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lock-X on A</td>
<td>lock-X on B</td>
</tr>
<tr>
<td></td>
<td>write (A)</td>
<td>write (B)</td>
</tr>
<tr>
<td></td>
<td>wait for lock-X on B</td>
<td>wait for lock-X on A</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.
## Deadlock Prevention

<table>
<thead>
<tr>
<th></th>
<th>Wait / Die</th>
<th>Wound / Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>O Needs a resource held by Y</td>
<td>O Waits</td>
<td>Y Dies</td>
</tr>
<tr>
<td>Y needs a resource held by O</td>
<td>Y Dies</td>
<td>Y Waits</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Wait**
  - Req. by Old
  - Locked by Young

- **DIE**
  - Req by Young
  - Locked by Old

- **WAIT**
  - Req by Young
  - Locked by Young

- **WOUND**
  - Req by Young
  - Locked by Old

**WAIT / DIE**

**WOUND / WAIT**
Dealing with Deadlocks

How do you detect a deadlock?
- **Wait-for graph**
- Directed edge from Ti to Tj
  - If Ti waiting for Tj

Suppose T4 requests lock-S(Z)....
Detecting Deadlocks

- Wait-for graph has a cycle → deadlock

T2, T3, T4 are deadlocked

- Build wait-for graph, check for cycle

- How often?
  - Tunable
    - IF expect many deadlocks or many transactions involved
      run often to reduce aborts
    - ELSE run less often to reduce overhead
Recovering from Deadlocks

- Rollback one or more transaction
  - Which one?
    - Rollback the cheapest ones
    - Cheapest ill-defined
      - Was it almost done?
      - How much will it have to redo?
      - Will it cause other rollbacks?
  - How far?
    - May only need a partial rollback
  - Avoid starvation
    - Ensure same xction not always chosen to break deadlock
Timestamp-Based Protocols

Idea:
- Decide in advance ordering of transactions.
- Ensure concurrent schedule serializes to that serial order.

 Timestamps
1. TS(T_i) is time T_i entered the system
2. Data item timestamps:
   1. W-TS(Q): Largest timestamp of any action that wrote Q
   2. R-TS(Q): Largest timestamp of any action that read Q

 Timestamps -> serializability order
Idea: If action $p_i$ of Xact $T_i$ conflicts with action $q_j$ of Xact $T_j$, and $TS(T_i) < TS(T_j)$, then $p_i$ must occur before $q_j$. Otherwise, restart violating Xact.
When Xact T wants to read Object O

- If TS(T) < W-TS(O), this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again!)

- If TS(T) > W-TS(O):
  - Allow T to read O.
  - Reset R-TS(O) to max(R-TS(O), TS(T))

- Change to R-TS(O) on reads must be written to disk! This and restarts represent overhead.

---

U writes O

T reads O

T start  U start
When Xact T wants to Write Object O

- If $TS(T) < R-TS(O)$, then the value of $O$ that $T$ is producing was needed previously, and the system assumed that that value would never be produced. write rejected, $T$ is rolled back.

- If $TS(T) < W-TS(O)$, then $T$ is attempting to write an obsolete value of $O$. Hence, this write operation is rejected, and $T$ is rolled back.

- Otherwise, the write operation is executed, and $W-TS(O)$ is set to $TS(T)$.
Timestamp-Ordering Protocol

- Rollbacks still present
  - On rollback, new timestamp & restart

T1 rollback since $TS(T1) < W-TS(O)=TS(T2)$

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write(O)</td>
<td>Write(O)</td>
</tr>
<tr>
<td>Write</td>
<td>Write(O)</td>
<td>Write(O)</td>
</tr>
</tbody>
</table>

Can reduce one rollback situation
When transaction writes an obsolete value, ignore it:
Thomas’ write-rule does not rollback T1
Example Use of the Protocol

A partial schedule for several data items for transactions with initial
timestamps 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₅</th>
<th>T₆</th>
<th>T₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(Y)</td>
<td>read(Y)</td>
<td>write(Y)</td>
<td>write(Z)</td>
<td>write(X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(X) abort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(Q) abort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(Z abort)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(Z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>write(Y)</td>
<td>write(Z)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>read(X)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

  ![Diagram with two circles connected by an arrow from 'transaction with smaller timestamp' to 'transaction with larger timestamp'.]

  Thus, there will be no cycles in the precedence graph.

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.