Multiple transactions are allowed to run concurrently in the system. Advantages are:

- **increased processor and disk utilization**, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk
- **reduced average response time** for transactions: short transactions need not wait behind long ones.

**Concurrency control schemes** – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

**Schedules** – sequences that indicate the chronological order in which instructions of concurrent transactions are executed

- a schedule for a set of transactions must consist of all instructions of those transactions
- must preserve the order in which the instructions appear in each individual transaction.

### Serial Schedule

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td></td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td></td>
</tr>
<tr>
<td>write($A$)</td>
<td></td>
</tr>
<tr>
<td>read($B$)</td>
<td></td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td></td>
</tr>
<tr>
<td>write($B$)</td>
<td></td>
</tr>
<tr>
<td>read($A$)</td>
<td></td>
</tr>
<tr>
<td>$temp := A \times 0.1$</td>
<td></td>
</tr>
<tr>
<td>$A := A - temp$</td>
<td></td>
</tr>
<tr>
<td>write($A$)</td>
<td></td>
</tr>
<tr>
<td>read($B$)</td>
<td></td>
</tr>
<tr>
<td>$B := B + temp$</td>
<td></td>
</tr>
<tr>
<td>write($B$)</td>
<td></td>
</tr>
</tbody>
</table>

### Equivalent Schedule

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td></td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td></td>
</tr>
<tr>
<td>write($A$)</td>
<td></td>
</tr>
<tr>
<td>read($B$)</td>
<td></td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td></td>
</tr>
<tr>
<td>write($B$)</td>
<td></td>
</tr>
<tr>
<td>read($A$)</td>
<td></td>
</tr>
<tr>
<td>$temp := A \times 0.1$</td>
<td></td>
</tr>
<tr>
<td>$A := A - temp$</td>
<td></td>
</tr>
<tr>
<td>write($A$)</td>
<td></td>
</tr>
<tr>
<td>read($B$)</td>
<td></td>
</tr>
<tr>
<td>$B := B + temp$</td>
<td></td>
</tr>
<tr>
<td>write($B$)</td>
<td></td>
</tr>
</tbody>
</table>
Inconsistent Schedule

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>$temp := A \times 0.1$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td>$A := A - temp$</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>

Serializability

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. conflict serializability
  2. view serializability
- We ignore operations other than read and write instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only read and write instructions.

Conflict Serializability
Instructions $l_i$ and $l_j$ of transactions $T_i$ and $T_j$ respectively, conflict if and only if there exists some item $Q$ accessed by both $l_i$ and $l_j$, and at least one of these instructions wrote $Q$.

1. $l_i = \text{read}(Q)$, $l_j = \text{read}(Q)$. $l_i$ and $l_j$ don’t conflict.
2. $l_i = \text{read}(Q)$, $l_j = \text{write}(Q)$. They conflict.
3. $l_i = \text{write}(Q)$, $l_j = \text{read}(Q)$. They conflict
4. $l_i = \text{write}(Q)$, $l_j = \text{write}(Q)$. They conflict

Intuitively, a conflict between $l_i$ and $l_j$ forces a (logical) temporal order between them. If $l_i$ and $l_j$ are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

If a schedule $S$ can be transformed into a schedule $S’$ by a series of swaps of non-conflicting instructions, we say that $S$ and $S’$ are conflict equivalent.

We say that a schedule $S$ is conflict serializable if it is conflict equivalent to a serial schedule.

Example of a schedule that is not conflict serializable:

\[
\begin{array}{c|c|c}
T_3 & T_4 \\
\hline
\text{read}(Q) & \text{write}(Q) \\
\text{write}(Q) & \text{read}(Q) \\
\end{array}
\]

We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

Schedule 3 below can be transformed into Schedule 1, a serial schedule where $T_2$ follows $T_1$, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{read}(A) & \text{read}(A) \\
\text{write}(A) & \text{write}(A) \\
\text{read}(B) & \text{read}(B) \\
\text{write}(B) & \text{write}(B) \\
\end{array}
\]
View Serializability

Let $S$ and $S'$ be two schedules with the same set of transactions. $S$ and $S'$ are **view equivalent** if the following three conditions are met:

1. For each data item $Q$, if transaction $T_i$ reads the initial value of $Q$ in schedule $S$, then transaction $T_i$ must, in schedule $S'$, also read the initial value of $Q$.
2. For each data item $Q$, if transaction $T_i$ executes read($Q$) in schedule $S$, and that value was produced by transaction $T_j$ (if any), then transaction $T_i$ must in schedule $S'$ also read the value of $Q$ that was produced by transaction $T_j$.
3. For each data item $Q$, the transaction (if any) that performs the final write($Q$) operation in schedule $S$ must perform the final write($Q$) operation in schedule $S'$.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

- A schedule $S$ is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Schedule 9 (from text) — a schedule which is view-serializable but not conflict serializable.
- Every view serializable schedule that is not conflict serializable has **blind writes**.

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($Q$)</td>
<td>write($Q$)</td>
<td>write($Q$)</td>
</tr>
<tr>
<td>write($Q$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lock-Based Protocols
See Chapter 16,

- 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.
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.
Neither \( T_3 \) nor \( T_4 \) can make progress — executing \textbf{lock-S}(B)\ causes \( T_4 \) to wait for \( T_3 \) to release its lock on \( B \), while executing \textbf{lock-X}(A)\ causes \( T_3 \) to wait for \( T_4 \) to release its lock on \( A \).

Such a situation is called a \textbf{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.

\textbf{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.

**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 \textbf{lock points} (i.e. the point where a transaction acquired its final lock).

Two-phase locking \textit{does not} ensure freedom from deadlocks

Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called \textbf{strict two-phase locking}. Here a transaction must hold all its exclusive locks till it commits/aborts.

\textbf{Rigorous two-phase locking} is even stricter: here \textit{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.

Can allow 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.