Transactions

- A transaction is a set of operations on objects to be performed as an indivisible unit by the servers managing those objects.
- Banking example: Three accounts A:$100, B:$200, C:$300
  - Client 1: transfer $4 from A to B
  - Client 2: transfer $3 from C to B
- Result can be inconsistent unless certain properties are imposed on the accesses

ACID Properties of Transactions

- **Atomic**: Either all of the operations in the transaction succeed or none of the operations persist.
- **Consistent**: If the data are consistent before the transaction begins, then they will be consistent after the transaction finishes.
  - It is the responsibility of the programmers to ensure consistency.
- **Isolated**: Each transaction must be performed without interference from other transactions, i.e., the intermediate effects of a transaction must not be visible to other transactions.
- **Durable**: After a transaction has completed successfully, all its effects are saved in permanent storage.
Transaction Primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Make the start of a transaction</td>
</tr>
<tr>
<td>END_TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, etc.</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, etc.</td>
</tr>
</tbody>
</table>

An airline reservation example:
BEGIN_TRANSACTION
if(reserve(NY, Paris)==full) ABORT_TRANSACTION
if(reserve(Paris, Athens)==full) ABORT_TRANSACTION
if(reserve(Athens, Hong Kong)==full) ABORT_TRANSACTION
END_TRANSACTION

Distributed Transactions

- A distributed transaction is a transaction that accesses objects managed by multiple servers
  - All servers involved must agree whether to commit or abort
- A distributed transaction can be structured in two ways:
  - In a flat transaction, operations are invoked sequentially
  - In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting
    - Each subtransaction can independently commit or abort
    - Subtransactions at the same level can run concurrently
Examples

BEGIN_TRANSACTION
  A.withdraw(10);
  B.deposit(10);
END TRANSACTION

BEGIN_TRANSACTION
  BEGIN_SUB
    A.withdraw(10);
  END_SUB
  BEGIN_SUB
    B.deposit(10);
  END_SUB
END TRANSACTION

A flat transaction
A nested transaction

Achieving Atomicity: Private Workspace

• Each transaction gets its own copy of all objects
  – Changes are made to the copy, keep the original intact
• Commit: copy changed data to original
• Abort: discard the copy
• Optimizations
  – Not making copies for reads
  – Copying only what is required for writes
Private Workspace Example

(a) The file index and disk blocks for a three-block file
(b) The situation after a transaction has modified block 0 and appended block 3
(c) After committing

Achieving Atomicity: Write-Ahead Logs

• A transaction makes changes directly to objects
• Prior to making change, transaction writes to log on stable storage
  – A log record is a 4-tuple <transaction ID, object ID, original value, new value>
• If transaction commits, leave the changes
• If transaction aborts, read log records and undo changes (rollback)
Write-Ahead Log Example

```
x = 0;
y = 0;
BEGIN_TRANSACTION;
x = x + 1;
y = y + 2
x = y * y;
END_TRANSACTION;
```

<table>
<thead>
<tr>
<th></th>
<th>Log</th>
<th>Log</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>[x = 0 / 1]</td>
<td>[x = 0 / 1]</td>
<td>[x = 0 / 1]</td>
</tr>
<tr>
<td>(b)</td>
<td>[y = 0 / 2]</td>
<td>[y = 0 / 2]</td>
<td>[y = 0 / 2]</td>
</tr>
<tr>
<td>(c)</td>
<td>[x = 1 / 4]</td>
<td>[x = 1 / 4]</td>
<td>[x = 1 / 4]</td>
</tr>
<tr>
<td>(d)</td>
<td>[x = 1 / 4]</td>
<td>[x = 1 / 4]</td>
<td>[x = 1 / 4]</td>
</tr>
</tbody>
</table>

a) A transaction. b)-d) The log before each statement is executed.

Achieving Isolation

- The goal of **concurrency control** is to control the execution of concurrent transactions such that the result is the same as if they executed in some sequential order.
- A **schedule** of transactions is an interleaving of the operations of concurrent transactions.
- A **legal schedule** is one that provides results that are the same as though the transactions were serialized (i.e., performed one after another).
- A **scheduler** employs a concurrency control algorithm to produce legal schedules.
An Example

Serialization of Conflicting Operations

- **Conflicting operations** are those operations that operate on the same data item and whose combined effects depend on the order they are executed in.
  - A pair of operations conflicts if at least one operation is write: read-write conflict, write-write conflict
- A schedule is **serially equivalent** if all pairs of conflicting operations are performed in the same order on all data items
- A **serially equivalent schedule is a legal schedule**
An Example

T         U
-------------  -------------
x=read(i);
write(i, 55);
write(j, 44);
write(j, 66);

A serially equivalent schedule

T         U
-------------  -------------
x=read(i);
write(i, 55);
write(j, 66);
write(j,44);

A non-serially equivalent schedule

Approaches to Concurrency Control

• Optimistic concurrency control
• Two-phase locking
• Timestamp ordering
Optimistic Concurrency Control

- A transaction T does what it wants and validates changes prior to commit
  - Check if any object accessed by T has been modified by transactions that completed after T started
    - Yes – abort the transaction
    - No – commit the transaction
  - Insight: conflicts are rare, so works well most of the time
- Advantages:
  - Deadlock free (no locks are used)
  - Maximum parallelism
- Disadvantages:
  - Rerun transaction if aborts
  - Probability of conflict rises substantially at high loads
- Not widely used

Two-Phase Locking (2PL)

- 2PL is a widely used concurrency control technique
- Each transaction obtains a lock from scheduler before performing a read or write operation
- A lock for a data item is granted to a process if no conflicting locks are held by other processes
- A process releases a lock on a data item when the operation on the data item has been completed
- A transaction acquires all necessary locks in growing phase, releases locks in shrinking phase (i.e., once a transaction has released a lock, it cannot request any new locks)
  - This guarantees legal schedules
- 2PL can lead to deadlock
  - Solution: adding timeouts to locks. When a lock times out, the transaction holding the lock is aborted.
Ta
\[\begin{align*}
x &= 0 \\
x &= x + 1
\end{align*}\]

\text{Acquire x.lock}
\text{w(x)0}
\text{Release x.lock}

\text{x = 1 or 2 in a legal schedule}

Tb
\[\begin{align*}
x &= 0 \\
x &= x + 2
\end{align*}\]

\text{Acquire x.lock}
\text{w(x)0}
\text{Release x.lock}

\text{This schedule is illegal (x = 3).}

This schedule is illegal (x = 3).

2PL produces a legal schedule (x = 2)
Two-Phase Locking

A transaction executes in two phases

Strict Two-Phase Locking

- 2PL can lead to cascading aborts
  - Suppose T1 reads the result of a write of T2, if T2 is aborted then T1 must be aborted
- In strict 2PL, locks can only be released when the transaction commits or aborts
  - This avoids cascading aborts
Timestamp-Based Concurrency Control

• Basic idea: if two transactions perform a pair of conflicting operations, the transaction that started earlier should perform its operation first
  – This produces serially equivalent schedules
• Each transaction $T_i$ is given timestamp $ts(T_i)$ when it starts
• Two values are maintained for each data item $x$
  – $max-rts(x)$: max timestamp of a transaction that read $x$
  – $max-wts(x)$: max timestamp of a transaction that wrote $x$
• When transaction $T_i$ performs $Read_i(x)$
  – If $ts(T_i) < max-wts(x)$ then Abort $T_i$
  – Else perform $Read_i(x)$, $max-rts(x) \leftarrow \text{MAX}(max-rts(x), ts(T_i))$
• When transaction $T_i$ performs $Write_i(x)$
  – If $ts(T_i) < max-rts(x)$ or $ts(T_i) < max-wts(x)$ then Abort $T_i$
  – Else perform $Write_i(x)$, $max-wts(x) \leftarrow ts(T_i)$