Last Class

- Leader election
- Distributed mutual exclusion

Transactions

- Transactions provide higher level mechanism for *atomicity* of processing in distributed systems
  - Have their origins in databases
- 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

<table>
<thead>
<tr>
<th>Access</th>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read A</td>
<td>$100</td>
<td></td>
</tr>
<tr>
<td>Write A</td>
<td>$96</td>
<td></td>
</tr>
<tr>
<td>Read B</td>
<td>$200</td>
<td></td>
</tr>
<tr>
<td>Read B</td>
<td>$200</td>
<td></td>
</tr>
<tr>
<td>Write B</td>
<td>$203</td>
<td>$204</td>
</tr>
<tr>
<td>Read C</td>
<td></td>
<td>$300</td>
</tr>
<tr>
<td>Write C</td>
<td></td>
<td>$297</td>
</tr>
</tbody>
</table>
ACID Properties

- **Atomic**: all or nothing
- **Consistent**: transaction takes system from one consistent state to another
- **Isolated**: Immediate effects are not visible to other (serializable)
- **Durable**: Changes are permanent once transaction completes (commits)

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read A: $100</td>
<td></td>
</tr>
<tr>
<td>Write A: $96</td>
<td></td>
</tr>
<tr>
<td>Read B: $200</td>
<td>Read C: $300</td>
</tr>
<tr>
<td>Write B: $204</td>
<td>Write C: $297</td>
</tr>
<tr>
<td></td>
<td>Read B: $204</td>
</tr>
<tr>
<td></td>
<td>Write B: $207</td>
</tr>
</tbody>
</table>

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, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>

Example: airline reservation

```
Begin_transaction
    if(reserve(NY,Paris)==full) Abort_transaction
    if(reserve(Paris,Athens)==full)Abort_transaction
    if(reserve(Athens,Delhi)==full) Abort_transaction
End_transaction
```
Distributed Transactions

(a) Nested transaction
- Subtransaction
  - Airline database
  - Hotel database
- Two different (independent) databases

(b) Distributed transaction
- Subtransaction
  - Distributed database
- Two physically separated parts of the same database

Implementation: Private Workspace

- Each transaction get copies of all files, objects
- Can optimize for reads by not making copies
- Can optimize for writes by copying only what is required
- Commit requires making local workspace global
Option 2: Write-ahead Logs

- **In-place updates**: transaction makes changes directly to all files/objects
- **Write-ahead log**: prior to making change, transaction writes to log on stable storage
  - Transaction ID, block number, original value, new value
- Force logs on commit
- If abort, read log records and undo changes [*rollback*]
- Log can be used to rerun transaction after failure

- Both workspaces and logs work for distributed transactions
- Commit needs to be *atomic* [will return to this issue in Ch. 7]

---

Writeahead 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>[x = 1/4]</td>
</tr>
</tbody>
</table>

- a) A transaction
- b) – d) The log before each statement is executed
Concurrency Control

• Goal: Allow several transactions to be executing simultaneously such that
  – Collection of manipulated data item is left in a consistent state
• Achieve consistency by ensuring data items are accessed in an specific order
  – Final result should be same as if each transaction ran sequentially

• Concurrency control can implemented in a *layered* fashion

Concurrency Control Implementation

- General organization of managers for handling transactions.
Distributed Concurrency Control

- General organization of managers for handling distributed transactions.

Serializability

BEGIN_TRANSACTION
x = 0;
x = x + 1;
END_TRANSACTION

BEGIN_TRANSACTION
x = 0;
x = x + 2;
END_TRANSACTION

BEGIN_TRANSACTION
x = 0;
x = x + 3;
END_TRANSACTION

(a) (b) (c)

<table>
<thead>
<tr>
<th>Schedule</th>
<th>X-values</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x = 0;  x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3</td>
<td>Legal</td>
</tr>
<tr>
<td>2</td>
<td>x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;</td>
<td>Legal</td>
</tr>
<tr>
<td>3</td>
<td>x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;</td>
<td>Illegal</td>
</tr>
</tbody>
</table>

- Key idea: properly schedule conflicting operations
- Conflict possible if at least one operation is write
  - Read-write conflict
  - Write-write conflict
Optimistic Concurrency Control

• Transaction does what it wants and *validates* changes prior to commit
  – Check if files/objects have been changed by committed transactions since they were opened
  – Insight: conflicts are rare, so works well most of the time
• Works well with private workspaces
• Advantage:
  – Deadlock free
  – Maximum parallelism
• Disadvantage:
  – Rerun transaction if aborts
  – Probability of conflict rises substantially at high loads
• Not used widely

Two-phase Locking

• Widely used concurrency control technique
• Scheduler acquires all necessary locks in growing phase, releases locks in shrinking phase
  – Check if operation on *data item* \( x \) conflicts with existing locks
    • If so, delay transaction. If not, grant a lock on \( x \)
  – Never release a lock until data manager finishes operation on \( x \)
  – One a lock is released, no further locks can be granted
• Problem: deadlock possible
  – Example: acquiring two locks in different order
• Distributed 2PL versus centralized 2PL
Two-Phase Locking

1. Two-phase locking.

Strict Two-Phase Locking

2. Strict two-phase locking.
Timestamp-based Concurrency Control

- Each transaction $T_i$ is given timestamp $ts(T_i)$
- If $T_i$ wants to do an operation that conflicts with $T_j$
  - Abort $T_i$ if $ts(T_i) < ts(T_j)$
- When a transaction aborts, it must restart with a new (larger) time stamp
- Two values for each data item $x$
  - $Max-rts(x)$: max time stamp of a transaction that read $x$
  - $Max-wts(x)$: max time stamp of a transaction that wrote $x$

Reads and Writes using Timestamps

- $Read_i(x)$
  - If $ts(T_i) < max-wts(x)$ then Abort $T_i$
  - Else
    - Perform $R_i(x)$
    - $Max-rts(x) = max(max-rts(x), ts(T_i))$
- $Write_i(x)$
  - If $ts(T_i) < max-rts(x)$ or $ts(T_i) < max-wts(x)$ then Abort $T_i$
  - Else
    - Perform $W_i(x)$
    - $Max-wts(x) = ts(T_i)$
Pessimistic Timestamp Ordering

1. Concurrency control using timestamps.

\[
\begin{align*}
&\text{ts}_{\text{RD}}(x) & \text{ts}_{\text{WR}}(x) & \text{ts}(T_2) \\
& (T_1) & (T_1) & (T_2) \\
& \text{(a)} & & \text{Do tentative write} & \text{OK} \\
&\text{ts}_{\text{WR}}(x) & \text{ts}_{\text{RD}}(x) & \text{ts}(T_2) \\
& (T_1) & (T_1) & (T_2) \\
& \text{(b)} & & \text{OK} \\
&\text{ts}(T_2) & \text{ts}_{\text{RD}}(x) \\
& (T_2) & (T_3) \\
& \text{(c)} & & \text{Abort} \\
&\text{ts}(T_2) & \text{ts}_{\text{WR}}(x) \\
& (T_2) & (T_3) \\
& \text{(d)} & & \text{OK} \\
&\text{ts}_{\text{WR}}(x) & \text{ts}_{\text{tent}}(x) & \text{ts}(T_2) \\
& (T_1) & (T_3) & (T_2) \\
& \text{(e)} & & \text{OK} \\
&\text{ts}(T_2) & \text{ts}_{\text{WR}}(x) \\
& (T_2) & (T_3) \\
& \text{(g)} & & \text{Abort} \\
&\text{ts}(T_2) & \text{ts}_{\text{tent}}(x) \\
& (T_2) & (T_3) \\
& \text{(h)} & & \text{Abort}
\end{align*}
\]