**Transactions and Concurrency Control**

**Concurrency in a Single Object**

- Requirements:
  - Atomicity/Consistency
  - Client Cooperation
- Control Mechanisms
  - Thread synchronization (synchronized methods)
  - Thread communication (wait and notify)
- Desired Property: Fairness

**Transactions (on Multiple Objects)**

- Basic requirement: a sequence of client requests performed as an *indivisible* unit
- Properties:
  - Atomicity: all or nothing
  - Consistency
  - Isolation: partial effects not visible
  - Durability: effects saved in permanent storage
- Means to maximize concurrency
  - Serializable interleavings

* Will assume all objects reside on a single server …

**Operations of the Account Interface**

```plaintext
deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() → amount
return the balance of the account
setBalance(amount)
set the balance of the account to amount
```

**Operations of the Branch Interface**

```plaintext
create(name) → account
create a new account with a given name
lookUp(name) → account
return a reference to the account with the given name
branchTotal() → amount
return the total of all balances at the branch
```

**A Client’s Bank Transaction**

Transaction $T$:

```plaintext
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);
```
The Usual Failure Model

- Writes to permanent storage may fail
- Processors may crash
- Messages may be delayed or even lost

The faults are assumed to be detectable.

Operations in the Coordinator Interface

```
openTransaction() -> trans;
starts a new transaction and delivers a unique THD trans. This identifier will be used in the other operations in the transaction.
closeTransaction() = (commit, abort);
ends a transaction: a commit return value indicates that the transaction has committed; an abort return value indicates that it has aborted.
abortTransaction(trans);
aborts the transaction.
```

Transaction Life Histories

The Lost Update Problem

<table>
<thead>
<tr>
<th>Transaction F:</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance = b.getBalance();</td>
<td>balance = b.getBalance();</td>
</tr>
<tr>
<td>b.setBalance(balance + $100);</td>
<td>b.setBalance(balance + $100);</td>
</tr>
<tr>
<td>a.withdraw(balance/10);</td>
<td>a.withdraw(balance/10);</td>
</tr>
<tr>
<td>$100</td>
<td>$100</td>
</tr>
<tr>
<td>total = a.getBalance();</td>
<td>total = total + b.getBalance();</td>
</tr>
<tr>
<td>$300</td>
<td>$300</td>
</tr>
<tr>
<td>b.deposit(100);</td>
<td>c.withdraw(balance/10);</td>
</tr>
<tr>
<td>$300</td>
<td>$200</td>
</tr>
</tbody>
</table>

The Inconsistent Retrievals Problem

Serializable Interleavings

```
Transaction F: | Transaction U: |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>balance = b.getBalance();</td>
<td>balance = b.getBalance();</td>
</tr>
<tr>
<td>b.setBalance(balance + $100);</td>
<td>b.setBalance(balance + $100);</td>
</tr>
<tr>
<td>a.withdraw(balance/10);</td>
<td>c.withdraw(balance/10);</td>
</tr>
<tr>
<td>$100</td>
<td>$200</td>
</tr>
<tr>
<td>balance = b.getBalance();</td>
<td>balance = b.getBalance();</td>
</tr>
<tr>
<td>b.setBalance(balance + $100);</td>
<td>b.setBalance(balance + $100);</td>
</tr>
<tr>
<td>a.withdraw(balance/10);</td>
<td>c.withdraw(balance/10);</td>
</tr>
<tr>
<td>$300</td>
<td>$220</td>
</tr>
<tr>
<td>total = a.getBalance();</td>
<td>total = total + b.getBalance();</td>
</tr>
<tr>
<td>$300</td>
<td>$320</td>
</tr>
<tr>
<td>b.deposit(100);</td>
<td>c.withdraw(balance/10);</td>
</tr>
<tr>
<td>$300</td>
<td>$78;</td>
</tr>
</tbody>
</table>
```
Ensuring Serializability

All pairs of conflicting operations of two transactions should be executed in the same order to ensure serializability.

(Two operations conflict if their combined effect depends on the order in which they are executed.)

Conflict Rules for Read and Write

A Non-Serializable Interleaving

Recoverability from Aborts

- Strict executions of transactions
  - Dirty reads (reading uncommitted values)
  - Cascading aborts
  - Premature writes (overwriting uncommitted values)
- Use of tentative versions

A Dirty Read
Overwriting Uncommitted Values

<table>
<thead>
<tr>
<th>Transaction F:</th>
<th>Transaction G:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.setBalance(105)</td>
<td>a.setBalance(110)</td>
</tr>
<tr>
<td>$100</td>
<td>$105</td>
</tr>
<tr>
<td>a.setBalance(110)</td>
<td></td>
</tr>
<tr>
<td>$110</td>
<td></td>
</tr>
</tbody>
</table>


Nested Transactions

Methods of Concurrency Control

- **Locking**: locks are used to order transactions according to the order of arrival of their operations at the same object
- **Optimistic concurrency control**: transactions are allowed to proceed until they are ready to commit, whereupon a check is made to see whether they have performed conflicting operations
- **Timestamp ordering**: timestamps are used to order transactions according to their starting times

Advantages of Nested Transactions

- Subtransactions at the same level may run concurrently
- Subtransactions can commit or abort independently

Methods of Concurrency Control

Exclusive Locks

Exclusive locks are a simple serializing mechanism:

- Two-phase locking: all locks are acquired in the first phase and released in the second
- Strict two-phase locking: locks are held until the transaction commits or aborts.

Granularity is an important issue.

Using Exclusive Locks

**Locking**

- A pair of read operations on the same object do not conflict.
- Exclusive locks reduce concurrency more than is necessary.
- The “many reader/single write” scheme distinguishes two types of lock: read (shared) locks and write locks.
- Two-phase locking or strict two-phase locking still applies for ensuring serializability.

**Lock Compatibility**

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read</td>
</tr>
<tr>
<td>lock already set</td>
<td>none</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td>write</td>
<td>wait</td>
</tr>
</tbody>
</table>

**Lock Management**


**A Lock Implementation**

```java
class Lock {
    private Object object; // The object being protected by the lock
    private Vector holders; // The list of current holders
    private LockType lockType; // The current type

    public synchronized void acquire(Transaction tranId, LockType lockType) {
        if (lockType == LockType.WRITE) {
            // If the transaction is a writer, add it to the list
            holders.addElement(tranId);
            lockType = LockType.WRITE;
        } else if (lockType == LockType.READ) {
            // If the transaction is a reader, check if it has a conflicting lock
            if (holders.contains(tranId)) {
                // If it has a conflicting lock, it cannot acquire the lock
                throw new LockException();
            } else {
                // Add the transaction to the list of holders
                holders.addElement(tranId);
                lockType = LockType.READ;
            }
        }
    }

    public synchronized void release(Transaction tranId) {
        // Remove the transaction from the list of holders
        holders.removeElement(tranId);
        lockType = LockType.NONE;
    }
}
```

**A Lock Manager Implementation**

```java
class LockManager {
    private HashTable locks;

    public synchronized void release(Transaction tranId, LockType lockType) {
        if (lockType == LockType.WRITE) {
            // If the transaction is a writer, remove it from the list
            locks.remove(tranId);
        } else if (lockType == LockType.READ) {
            // If the transaction is a reader, check if it has a conflicting lock
            if (locks.contains(tranId)) {
                // If it has a conflicting lock, it cannot release the lock
                throw new LockException();
            } else {
                // Remove the transaction from the list of holders
                locks.remove(tranId);
            }
        }
    }
}
```
Deadlocks

- The use of locks may lead to deadlock.
- Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.
- A wait-for graph can be used to represent the waiting relationships between concurrent transactions at a server.

A Deadlock

<table>
<thead>
<tr>
<th>Transaction F</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>Locks</td>
</tr>
<tr>
<td>a.deposit(00)</td>
<td>write lock A</td>
</tr>
<tr>
<td>b.withdraw(100)</td>
<td>waits for U’s</td>
</tr>
<tr>
<td>---</td>
<td>lock on B</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

A Wait-For Graph

[Diagram of a wait-for graph showing transactions A, B, and U waiting for each other]

A Cycle in a Wait-For Graph

[Diagram of a cycle in a wait-for graph showing transactions U, V, and W waiting in a cycle]

Another Wait-For Graph

[Diagram of another wait-for graph showing transactions A, B, C, D, and E waiting for each other]

Deadlock Prevention Techniques

- Lock (in one atomic step) all objects used by a transaction when it starts.
- Every transaction requests locks on objects in a predefined order.
- Each lock is given a limited period, after which it becomes vulnerable.
Deadlock Detection

- Deadlocks may be detected by finding cycles in the wait-for graph.
- Two design issues:
  - Frequency of checking the existence of a cycle
  - Choice of transactions to be aborted

Increasing Concurrency

- **Two-version locking**: allows one transaction to write tentative versions of objects while other transactions read from the committed version of the same object; read operations wait only if another transaction is currently committing the same object
- **Hierarchic locks**: at each level, the setting of a parent lock has the same effect as setting all the equivalent child locks

Resolving a Deadlock

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>Operations</td>
</tr>
<tr>
<td>a.deposit(100)</td>
<td>h.deposit(200)</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td></td>
</tr>
<tr>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>waits for U's</td>
<td>waits for T's</td>
</tr>
<tr>
<td>lock on B</td>
<td>lock on A</td>
</tr>
<tr>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>(timeout elapses)</td>
<td></td>
</tr>
<tr>
<td>P's lock on A becomes vulnerable, unlock A, abort T</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(200)</td>
<td>write lock A</td>
</tr>
<tr>
<td></td>
<td>unlock A, abort T</td>
</tr>
</tbody>
</table>

Increasing Concurrency

- Two-version locking: allows one transaction to write tentative versions of objects while other transactions read from the committed version of the same object; read operations wait only if another transaction is currently committing the same object
- Hierarchic locks: at each level, the setting of a parent lock has the same effect as setting all the equivalent child locks

Lock Compatibility

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock to be set</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
</tr>
<tr>
<td>lock already set</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td>write</td>
<td>OK</td>
</tr>
<tr>
<td>commit</td>
<td>wait</td>
</tr>
</tbody>
</table>

Compatibility of Hierarchical Locks

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock to be set</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
</tr>
<tr>
<td>lock already set</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td>write</td>
<td>wait</td>
</tr>
<tr>
<td>I-read</td>
<td>OK</td>
</tr>
<tr>
<td>I-write</td>
<td>wait</td>
</tr>
</tbody>
</table>
### Drawbacks of Locking

- Overhead of lock maintenance: some locks may be unnecessary
- Reduced concurrency due to
  - deadlock prevention
  - holding locks until the end of a transaction (to avoid cascade aborts)

* In some applications, the likelihood of conflict is low.

### Optimistic Concurrency Control

- Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client issues a `CloseTransaction` request.
- When a conflict is detected, some transaction is aborted.

### Optimistic Concurrency Control (cont.)

Each transaction has three phases:

- **Working** phase: use a tentative version for each updated object.
- **Validation** phase: check if there is a conflict.
- **Update** phase: if validated, all tentative versions are made permanent.

### Conflict Rules for Optimistic Concurrency Control

<table>
<thead>
<tr>
<th>$T_i$</th>
<th>$T_j$</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>read</td>
<td>1. $T_j$ must not read objects written by $T_i$</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>2. $T_j$ must not read objects written by $T_i$</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>3. $T_j$ must not write objects written by $T_i$ and $T_j$ must not write objects written by $T_i$.</td>
</tr>
</tbody>
</table>


### Validation of Transactions

- For validation purposes, each transaction is assigned (in an ascending order) a **transaction number** when it enters the validation phase.
- A transaction always finishes its working phase *after* all transactions with lower numbers.
Validation of Transactions (cont.)

- Validation phases may overlap (but transaction numbers should be assigned sequentially).
- All update phases are executed sequentially according to their transaction numbers so that there is no need to check write-write conflicts.
- Reuse of transaction numbers (as suggested in the book) is not a very good idea.

Comparing Backward and Forward Validation

- **Backward**: abort the transaction being validated
- **Forward**: three options:
  * defer the validation until the conflicting (and active) transactions have finished
  * abort all the conflicting active transactions and commit the one being validated
  * abort the transaction being validated

Comparing Backward and Forward Validation (cont.)

- **Backward**: must retain the write sets of committed transactions that may conflict with active transactions
- **Forward**: must allow for new transactions to start during validation
- **Backward**: compare a possibly large read set against old write sets
- **Forward**: compare a small write set against the read sets of active transactions

Timestamp Ordering

- Each transaction is assigned a unique timestamp value when it starts.
- Every operation bears the timestamp of its issuing transaction and is validated when it is carried out.
- If the operation cannot be validated, then the transaction is aborted immediately.

Conflict Rules for Timestamp Ordering

1. **write read** $T_w$ must not write an object that has been read by any $T_r$ where $T_r > T_w$.

2. **write write** $T_w$ must not write an object that has been written by any $T_r$ where $T_r > T_w$.

3. **read read** $T_r$ must not read an object that has been read by any $T_w$ where $T_w > T_r$.
If a write is too late it can be ignored instead of aborting the transaction, because if it had arrived in time its effects would have been overwritten anyway.

However, if another transaction has read the object, the transaction with the late write fails due to the read timestamp on the object.
Multiversion Timestamp Ordering

- The server keeps old committed versions as well as tentative versions in its list of versions of objects.
- With the list, Read operations that arrive too late need not be rejected.
- A Read operation of a transaction is directed to the version with the largest write timestamp less than the transaction timestamp.

A Late Write Invalidating a Read

A Late Write Invalidating a Read

Comparison

<table>
<thead>
<tr>
<th></th>
<th>Locks</th>
<th>Timestamps</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Attitude&quot;</td>
<td>pessimistic</td>
<td>optimistic</td>
</tr>
<tr>
<td>Order Decision</td>
<td>dynamic</td>
<td>static</td>
</tr>
<tr>
<td>Benefiting Trans.</td>
<td>more writes than reads</td>
<td>read-only</td>
</tr>
<tr>
<td>Conflict Resolution</td>
<td>wait; may abort later</td>
<td>abort immediately</td>
</tr>
</tbody>
</table>