Recap: Conflicting Operations

- Two operations are said to be in conflict if their combined effect depends on the order they are executed, e.g., read-write, write-read, write-write (all on same variables). NOT read-read, not on different variables.

<table>
<thead>
<tr>
<th>Operations of different transactions</th>
<th>Conflict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>read read</td>
<td>No</td>
<td>Because the effect of a pair of read operations does not depend on the order in which they are executed.</td>
</tr>
<tr>
<td>read write</td>
<td>Yes</td>
<td>Because the effect of a read and a write operation depends on the order of their execution.</td>
</tr>
<tr>
<td>write write</td>
<td>Yes</td>
<td>Because the effect of a pair of write operations depends on the order of their execution.</td>
</tr>
</tbody>
</table>

Recap: Serial Equivalence

- An interleaving of the operations of 2 or more transactions is said to be serially equivalent if the combined effect is the same as if these transactions had been performed sequentially (in some order).

Transaction T1: balance = b.getBalance();
balance = balance * 1.1;
a.withdraw(balance * 0.1);

Transaction T2: balance = b.getBalance();
b.setBalance(balance * 1.1);
c.withdraw(balance * 0.1);

Pairs of Conflicting Operations:
- Transaction T1 read write Transaction T2
- Transaction T1 write write Transaction T2
- Transaction T2 read write Transaction T1
- Transaction T2 write write Transaction T1

Recap: Serial Equivalence

- How to provide serial equivalence with conflicting operations?
  - Execute all pairs of conflicting operations in the same order for all objects.

Implementing Transactions

- Two things we wanted to take care of (from the last lecture):
  - Performance: interleaving of operations
  - Failure: intentional (abort()), unintentional (e.g., process failure)

- Interleaving must satisfy serial equivalence
- What about failures?
  - Should be able to rollback as if no transaction has happened.
Handling Abort()

- What can go wrong?

<table>
<thead>
<tr>
<th>Transaction 1:</th>
<th>Transaction 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100);</td>
<td>allBranch.branchTotal();</td>
</tr>
<tr>
<td>b.deposit(100);</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(100);</td>
<td>$100</td>
</tr>
<tr>
<td>b.deposit(100);</td>
<td>$300</td>
</tr>
</tbody>
</table>

Strict Executions of Transactions

- Transactions should **delay both their read and write operations** on an object
  - Until all transactions that previously wrote that object have either committed or aborted
  - This is called **strict executions**.
- How do we implement serial equivalence & strict executions? Many ways
- We’ll see how to do this with locks

Using Exclusive Locks

- Exclusive Locks

Using Exclusive Locks

- Two phase locking
  - To satisfy serial equivalence
  - First phase (growing phase): new locks are acquired
  - Second phase (shrinking phase): locks are only released
  - A transaction is not allowed to acquire any new lock, once it has released any one lock
- Strict two phase locking
  - To handle abort() (failures)
  - Locks are only released at the end of the transaction, either at commit() or abort()
**Can We Do Better?**

- What we saw was "exclusive" locks.
- Non-exclusive locks: break a lock into a read lock and a write lock
  - Allows more concurrency
    - Read locks can be shared (no harm to share)
    - Write locks should be exclusive

---

**Non-Exclusive Locks**

<table>
<thead>
<tr>
<th>non-exclusive lock compatibility</th>
<th>Lock already</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>none</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
<td>WAIT</td>
</tr>
<tr>
<td>write</td>
<td>WAIT</td>
<td>WAIT</td>
</tr>
</tbody>
</table>

- A read lock is promoted to a write lock when the transaction needs write access to the same object.
- A read lock shared with other transactions’ read lock(s) cannot be promoted. Transaction waits for other read locks to be released.
- Cannot demote a write lock to read lock during transaction — violates the 2P principle

---

**Example: Non-Exclusive Locks**

**Transaction T1**

- OpenTransaction()
- balance = b.getBalance()
- Commit

**Transaction T2**

- OpenTransaction()
- balance = b.getBalance()
- b.setBalance(balance*1.1)
- Cannot Promote lock on B. Wait

Promote lock on B

---

**2PL: a Problem**

- What happens in the example below?

**Transaction T1**

- OpenTransaction()
- balance = b.getBalance()
- b.setBalance(balance*1.1)

**Transaction T2**

- OpenTransaction()
- balance = b.getBalance()
- b.setBalance(balance*1.1)

- Cannot Promote lock on B. Wait

---

**Deadlock Conditions**

- Necessary conditions
  - Non-sharable resources (locked objects)
  - No lock preemption
  - Hold & wait or circular wait

---

**Preventing Deadlocks**

- Acquiring all locks at once
- Acquiring locks in a predefined order
- Not always practical:
  - Transactions might not know which locks they will need in the future
- One strategy: timeout
  - If we design each transaction to be short and fast, then we can abort() after some period of time.
Extracting Even More Concurrency

- Allow writing tentative versions of objects
  - Letting other transactions read from the previously committed version
- Allow read and write locks to be set together by different transactions
  - Unlike non-exclusive locks
- Read operations wait only if another transaction is committing the same object
-Disallow commit if other uncompleted transactions have read the objects
  - These transactions must wait until the reading transactions have committed
- This allows for more concurrency than read-write locks
  - Writing transactions risk waiting or rejection when commit

Two-Version Locking

- Three types of locks: read lock, write lock, commit lock
  - Transaction cannot get a read or write lock if there is a commit lock
- When the transaction coordinator receives a request to commit
  - Converts all that transaction’s write locks into commit locks
  - If any objects have outstanding read locks, transaction must wait until the transactions that set these locks have completed and locks are released
- Compare with read/write locks:
  - Read operations are delayed only while transactions are committed
  - Read operations of one transaction can cause a delay in the committing of other transactions

<table>
<thead>
<tr>
<th>lock compatibility</th>
<th>Lock already set</th>
<th>Lock requested read</th>
<th>write</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
<td>OK</td>
<td>WAIT</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>OK</td>
<td>WAIT</td>
<td>WAIT</td>
<td></td>
</tr>
</tbody>
</table>

Summary

- Strict Execution
  - Delaying both their read and write operations on an object until all transactions that previously wrote that object have either committed or aborted
- Strict execution with exclusive locks
  - Strict 2PL
- Increasing concurrency
  - Non-exclusive locks
  - Two-version locks
  - Hierarchical locks

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).