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

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

Recap

- Question: How to support transactions (with locks)?
  - Multiple transactions share data.

- Complete serialization is correct, but performance and abort are two issues.

- Interleaving transactions for performance
  - Problem: Not all interleavings produce a correct outcome.
Handling Abort()

- What can go wrong?

<table>
<thead>
<tr>
<th>Transaction V:</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100);</td>
<td>aBranch.branchTotal();</td>
</tr>
<tr>
<td>b.deposit(100);</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(100);</td>
<td>total = a.getBalance(); $100</td>
</tr>
<tr>
<td>b.deposit(100);</td>
<td>total = total+b.getBalance(); $400</td>
</tr>
</tbody>
</table>

Strict Executions of Transactions

- Problem of interleaving for abort()
  - Intermediate state visible to other transactions, i.e., other transactions could have used some results already.
  - For abort(), transactions should delay both their read and write operations on an object (until commit time)
    - Until all transactions that previously wrote that object have either committed or aborted
      - This way, we avoid making intermediate states visible before commit, just in case we need to abort.
      - This is called strict executions.
    - This further restricts which interleavings of transactions are allowed.
    - Thus, correctness criteria for transactions:
      - Serial equivalence
      - Strict execution

Story Thus Far

- Question: How to support transactions?
  - With multiple transactions sharing data
- First strategy: Complete serialization
  - One transaction at a time with one big lock
  - Correct, but at the cost of performance
- How to improve performance?
  - Let’s see if we can interleave different transactions.
- Problem: Not all interleavings produce a correct outcome
  - Serial equivalence & strict execution must be met.
- Now, how do we meet the requirements?
  - Overall strategy: using more and more fine-grained locking
  - No silver bullet. Fine-grained locks have their own implications.

How to Acquire/Release Locks

- Can’t do it naively

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = a.read();</td>
<td>y = b.read();</td>
</tr>
<tr>
<td>a.write(20);</td>
<td>b.write(30);</td>
</tr>
<tr>
<td>b.write(x);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>z = a.read();</td>
</tr>
</tbody>
</table>

  - Serially equivalent?
  - Strict execution?

Using Exclusive Locks

- Exclusive Locks (Avoiding One Big Lock)

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin()</td>
<td>begin()</td>
</tr>
<tr>
<td>balance = b.getBalance()</td>
<td>balance = b.getBalance()</td>
</tr>
<tr>
<td>b.setBalance = (balance*1.1)</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(balance*0.1)</td>
<td>c.withdraw(balance*0.1)</td>
</tr>
<tr>
<td>commit()</td>
<td>commit()</td>
</tr>
</tbody>
</table>

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 satisfy strict execution, i.e., to handle abort() & failures
      - Locks are only released at the end of the transaction, either at commit() or abort(), i.e., the second phase is only executed at commit() or abort().
      - The example shown before does both.
CSE 486/586 Administrivia

- Midterm grades will be posted today.

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

non-exclusive lock compatibility

<table>
<thead>
<tr>
<th>Lock already set</th>
<th>Lock requested read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<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

begin()
balance = b.getBalance()
...
b.setBalance = balance*1.1
...
Commit

Transaction T2

begin()
balance = b.getBalance()
...
b.setBalance = balance*1.1

Cannot Promote lock on B, Wait

2PL: a Problem

What happens in the example below?

Transaction T1

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

Transaction T2

begin()
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.

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
  - Read and write (from different transactions) can go together.
  - Acquiring a commit lock only happens at commit().

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

Extracting Even More Concurrency

- This allows for more concurrency than read-write locks.
- Writing transactions risk waiting or rejection when commit
- Read operations wait only if another transaction is committing the same object
- Read operations of one transaction can cause a delay in the committing of other transactions

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
  - Etc.
Acknowledgements

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