Recap

- Question: How to support transactions?
  - Multiple transactions share data.

- Complete serialization is correct
  - Use a lock to serialize transactions.

- But performance and abort are two issues.
  - For performance: Interleaving transactions

Handling Abort()

- For serialized transactions, abort() can be done if we use temporary memory.
- When commit() is invoked at the end of each transaction, we make the final outcomes permanent and visible to other transactions.

```
Transaction T1 | Transaction T2
begin() | begin()
balance = b.getBalance() | bal = b.getBalance()
b.setBalance(b*1.1) | b.setBalance(b*1.1)
a.withdraw(balance*0.1) | c.withdraw(bal*0.1)
commit() | commit()
```

Handling Abort() with Interleaving

- What can go wrong?

```
Transaction V:
  a.withdraw(100);
  b.deposit(100)

Transaction W:
  aBranch.branchTotal()
  a.withdraw(100);
  b.deposit(100)
  total = a.getBalance()
  total = total + b.getBalance()
  total = total + c.getBalance()
```

Strict Executions of Transactions

- Problem of interleaving for abort():
  - Other transactions could have used intermediate results.
- In order to handle abort(), we need to avoid making intermediate states visible before commit, just in case we need to abort.
  - This means that transactions should delay their read and write operations on a shared object.
- Until all transactions that previously wrote to that object have either committed or aborted
  - This is called strict execution.
- 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.
Using Exclusive Locks

- Exclusive Locks (Avoiding One Big Lock)

**Transaction T1**
- `begin`
- `balance = b.getBalance()`
- `b.setBalance = (balance*.11)`
- `a.withdraw(balance*.11)`
- `commit`

**Transaction T2**
- `begin`
- `balance = b.getBalance()`
- `b.setBalance = (balance*.11)`
- `c.withdraw(balance*.11)`
- `commit`

---

How to Acquire/Release Locks

- Can’t do it naively

**Transaction T1**
- `x = a.read()`
- `a.write(20)`
- `b.write(x)`
- `a.write(10)`

**Transaction T2**
- `y = b.read()`
- `b.write(30)`
- `a.write(10)`

---

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 further 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 first example shown before does both. But the second example does neither.

---

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

---

CSE 486/586 Administrivia

- Midterm re-grading: This Friday 4 pm – 6 pm during my office hours
Non-Exclusive Locks

- A read lock said to be promoted to a write lock when the transaction needs write access.
- 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 strict 2P principle

<table>
<thead>
<tr>
<th>Lock already</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>read</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>write</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>write</td>
</tr>
</tbody>
</table>

Example: Non-Exclusive Locks

Transaction T1 | Transaction T2
---|---
begin()        | begin()
balance = b.getBalance() | balance = b.getBalance()

- Cannot promote lock on B, wait
- Commit

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.

Even More: Two-Version Locking

- Three types of locks: read lock, write lock, commit lock
  - Acquiring a commit lock only happens at commit().
  - Transaction cannot get a read or write lock if there is a commit lock
  - Read and write (from different transactions) can go concurrently

- Lock compatibility

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

- What can go wrong with this?
  - Read-write conflicts (but no write conflict)

A Problem

- What happens in the example below?

Transaction T1 | Transaction T2
---|---
begin() | begin()
balance = b.getBalance() | balance = b.getBalance()

- Cannot promote lock on B, wait
- Commit

...
Two-Version Locking

- Allow writing **tentative versions** of objects
  - Letting other transactions read from the previously committed version
  - Optimistic writes: this works well if there’s little chance of read-write conflicts.
- At commit(),
  - Promote all the write locks of the transaction 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

This allows for more concurrency than read-write locks.
- Writing transactions risk waiting 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).