CSE 486/586 Distributed Systems
Concurrency Control --- 2

Steve Ko
Computer Sciences and Engineering
University at Buffalo

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.

a: 100 b: 200 c: 300

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

Handling Abort() with Interleaving

- What can go wrong?

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

Strict Executions of Transactions

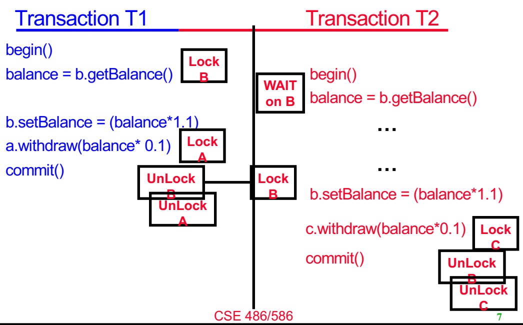
- Problem of interleaving for abort()
  - Other transactions could have used intermediate results.
- In order to handle abort(),
  - we need to avoid making intermediate states visible before commit, just in case we need to abort.
  - This means that transactions should *delay both their read and write 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.

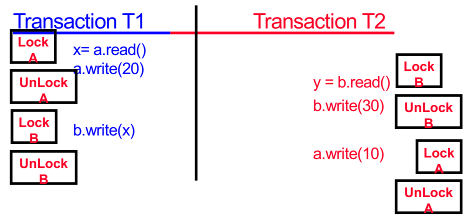
Using Exclusive Locks

- Exclusive Locks (Avoiding One Big Lock)



How to Acquire/Release Locks

- Can't do it naively



- Serially equivalent?
- Strict execution?

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.

CSE 486/586 Administrivia

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

Story Thus Far

- Question: How to support transactions?
  - With multiple transactions sharing data
  - One big lock works since it's complete serialization.
  - But performance suffers and it cannot handle abort().
- Interleaving for improved performance
  - Serial equivalence
- Abort() for interleaving
  - Strict execution
- Now, how do we meet the requirements?
  - Overall strategy: using locks
  - We looked at exclusive locks.
  - We'll look at two more schemes.

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

Lock already set	Lock requested: read	Lock requested: write
none	OK	OK
read	OK	WAIT
write	WAIT	WAIT

- 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

Example: Non-Exclusive Locks

Transaction T1	Transaction T2
begin()	
balance = b.getBalance() [R-Lock B]	begin()
…	balance = b.getBalance() [R-Lock B]
…	b.setBalance =balance*1.1
	[Cannot Promote lock on B, Wait]
Commit	
	[Promote lock on B]
	…

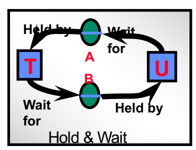
A Problem

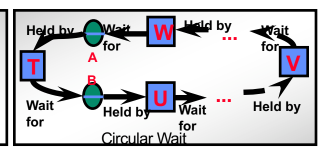
- What happens in the example below?

Transaction T1	Transaction T2
begin()	
balance = b.getBalance() [R-Lock B]	begin() [R-Lock B]
	balance = b.getBalance()
	b.setBalance =balance*1.1
	[Cannot Promote lock on B, Wait]
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.

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

Lock already set	Lock requested: read	Lock requested: write	Lock requested: commit
none	OK	OK	OK
read	OK	OK	WAIT
write	OK	WAIT	WAIT
commit	WAIT	WAIT	WAIT

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

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

Two-Version Locking

- 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).