CSE 486/586 Distributed Systems

Concurrency Control (part 2)

Slides by Steve Ko
Computer Sciences and Engineering
University at Buffalo

Recap

- Transactions need to provide ACID
- Serial equivalence defines correctness of executing concurrent transactions
- It is handled by ordering conflicting operations

Handling Abort()

- What can go wrong?

Transaction *V*: *a.withdraw(100); b.deposit(100)*		**Transaction *W*:** *aBranch.branchTotal()*	
a.withdraw(100);	\$100		
		total = a.getBalance()	\$100
b.deposit(100)	\$300		
		total = total+b.getBalance()	\$400
		total = total+c.getBalance() ...	

Strict Executions of Transactions

- Interleaving interacts with abort(), causing problems
  - Intermediate state is visible to other transactions; other transactions may have already used some (now non-final!) results.
- For abort(), transactions should delay both their read and write operations on an object (until commit time)
  - Until all transactions that have written that object have either committed or aborted
  - This is called strict execution, and avoids making intermediate states visible before commit, just in case we need to abort.
- This further restricts which interleavings of transactions are allowed.
  - Serial equivalence
  - Strict execution

Story Thus Far

- How can we support transactions with shared data
- First strategy: Complete serialization
  - One transaction at a time with one big lock
  - Correct, but at the cost of performance
- How can we improve performance?
  - Interleave different transactions
- Problem: Not all interleavings are correct
  - Serial equivalence and strict execution must be met.
- How do we meet these 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() — **Lock B**

b.setBalance = (balance*1.1)

a.withdraw(balance* 0.1) — **Lock A**

commit() — **UnLock B**, **UnLock A**

Transaction T2

begin()

balance = b.getBalance() — **WAIT on B**

…

…

b.setBalance = (balance*1.1) — **Lock B**

c.withdraw(balance*0.1) — **Lock C**

commit() — **UnLock B**, **UnLock C**

How to Acquire/Release Locks

- Can’t do it naively

Transaction T1		Transaction T2	
x= a.read()	Lock A	y = b.read()	Lock B
a.write(20)	UnLock A	b.write(30)	UnLock B
b.write(x)	Lock B	z = a.read()	Lock A
	UnLock B		UnLock A

- Serially equivalent?
- Strict execution?

Using Exclusive Locks

- Two phase locking
  - To satisfy serial equivalence
  - First (growing) phase: new locks are acquired
  - Second (shrinking) phase: locks are only released
  - A transaction is not allowed to acquire any new lock, once it has released any lock
- Strict two phase locking
  - To satisfy strict execution, *i.e.*, to handle abort() and failures
  - Locks are released only 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.

Can We Do Better?

- We have considered only exclusive locks.
- Non-exclusive locks break a lock into a read lock and a write lock
- Allows more concurrency
  - Read locks can be shared (read-read is not a conflict)
  - Write locks must 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 is promoted to a write lock when the transaction needs write access to a read locked object.
- A read lock already shared with other transactions' read locks cannot be promoted. The transaction must wait for other read locks to be released.
- Cannot demote a write lock to read lock during a transaction – violates the 2-phase principle

Example: Non-Exclusive Locks

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

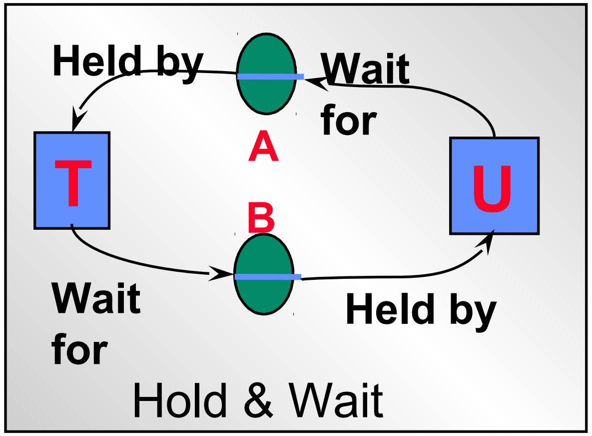
2PL: a Problem

- What happens in the example below?

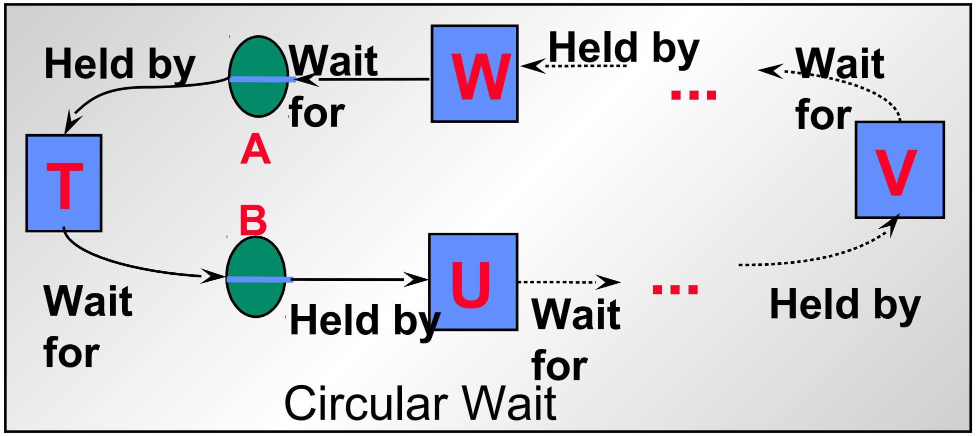
Transaction T1	Transaction T2
begin()	
balance = b.getBalance()	begin()
R-Lock B	balance = b.getBalance() **R-Lock B**
	b.setBalance =balance*1.1
	Cannot Promote lock on B, Wait
b.setBalance=balance*1.1	
Cannot Promote lock on B, Wait	
…	…

Deadlock Conditions

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



Hold & Wait



Circular Wait

Preventing Deadlocks

- Acquire all locks at once
- Acquire 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
  - Let other transactions read from the previously-committed version
- At commit():
  - Promote all write locks in the transaction to commit locks
  - If any objects have outstanding read locks, the committing transaction must wait until those transactions release their locks (complete)
- Allow different transactions to simultaneously take locks
  - Unlike non-exclusive locks
  - Write locks remain exclusive with other write locks
- Delay commits until all readers using the previously-committed version have committed.

Extracting Even More Concurrency

- This allows for more concurrency than read-write locks.
- Writing transactions risk waiting on commit
- Read operations wait only if another transaction is currently committing the same object
- Read operations of one transaction can cause a delay in the commit (or even abort, in the case of deadlock) of other transactions
- This can be extended even farther, to allow conflicting write locks at the risk of aborting conflicting writers [2]

Summary

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

References

[1] Textbook sections 16.1-16.5. **Required Reading.**

[2] H.T. Kung and J.T. Robinson. *On Optimistic Methods for Concurrency Control.* ACM Transactions on Database Systems, Vol. 6 No. 2. June 1981. pp.213-226 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.3052&rep=rep1&type=pdf

Acknowledgements

- These slides by Steve Ko, lightly modified and used with permission by Ethan Blanton
- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).