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

Steve Ko
Computer Sciences and Engineering
University at Buffalo

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

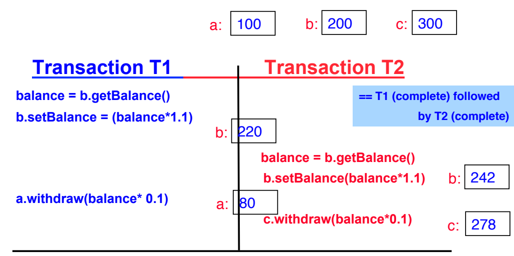
Recap: Conflicting Operations

- 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.

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

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).

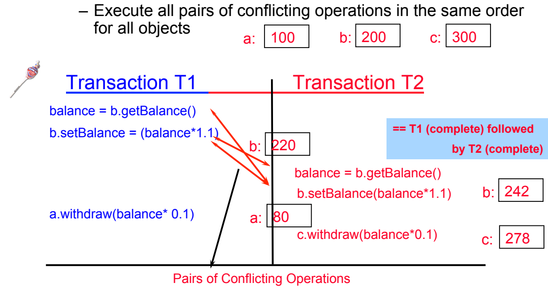


Recap: Serial Equivalence

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

Recap: Serial Equivalence

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



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

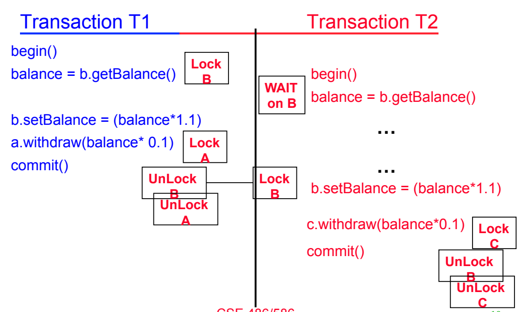
- Problem of interleaving for abort()
  - Intermediate state visible to other transactions, i.e., other transactions could have used some results already.
- For abort(), transactions should *delay both their read and write operations* on an object (until commit time)
  - Until all transactions that previously 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.

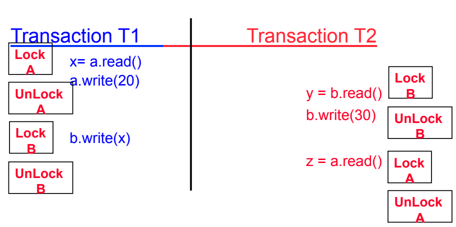
Using Exclusive Locks

- 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 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

- PA3 deadline: 4/8 (Friday)

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 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	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
	…

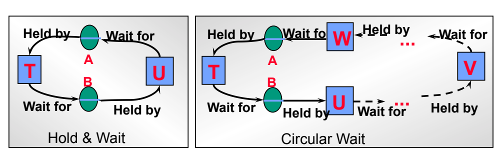
2PL: a Problem

- What happens in the example below?

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
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().

lock compatibility

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

- What can go wrong with this?

Extracting Even More Concurrency

- Allow writing *tentative versions* of objects
  - Letting other transactions read from the previously committed version
- 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
- Allow read and write locks to be set together by different transactions
  - Unlike non-exclusive locks
- Disallow commit if other uncompleted transactions have read the objects
  - These transactions must wait until the reading transactions have committed

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).