Concurrency, 2PL, and 2PC

Today's Topics

- Continue concurrency
  - Review two-phase locking (2PL)
  - 2PL with shared locks
  - Deadlocks
  - Timestamped concurrency
- Implementing distributed transactions
  - Transaction manager
  - Two-phase commit (2PC)

Recap from last class

Serial Equivalence: combined effect of two (or more) tx is equivalent to a serial execution

T1	T2
read X	
read Y	
write Z	read A
	write Y
write W	
	write Z

Recap from last class

Serial Equivalence: combined effect of two (or more) tx is equivalent to a serial execution

Conflicts are operations in two tx on same data whose combined effect depends on order

T1	T2
read X	
read Y	
write Z	read A
	write Y
write W	
	write Z

Recap from last class

Serial Equivalence: combined effect of two (or more) tx is equivalent to a serial execution

Conflicts are operations in two tx on same data whose combined effect depends on order

- Read/write or write/write

T1	T2
read X	
read Y	
write Z	read A
	write Y
write W	
	write Z

Recap from last class

Serial Equivalence: combined effect of two (or more) tx is equivalent to a serial execution

Conflicts are operations in two tx on same data whose combined effect depends on order

If all conflicts follow transaction ordering, execution is serially equivalent

T1	T2
read X	
read Y	
write Z	read A
	write Y
write W	
	write Z

Recap from last class

Serial Equivalence: combined effect of two (or more) tx is equivalent to a serial execution

Conflicts are operations in two tx on same data whose combined effect depends on order

If all conflicts follow transaction ordering, execution is serially equivalent

T1	T2
read X	
read Y	
write Z	
write W	
	read A
	write Y
	write Z

Recap from last class

Serial Equivalence: combined effect of two (or more) tx is equivalent to a serial execution

Conflicts are operations in two tx on same data whose combined effect depends on order

If all conflicts follow transaction ordering, execution is serially equivalent

Two-phase locking (2PL): lock variable before access, unlock only a commit/abort time

T1	T2
lock X; read X	
lock Y; read Y	
lock Z; write Z	lock A; read A try lock Y…
lock W; write W	
commit; unlock all	
	lock Y; write Y
	lock Z; write Z

Exclusive locks: missed parallelism

T1

read A

read B

write C

T2

read A

read B

write D

2P Locking: Non-exclusive lock (per object)

- 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

Lock set	Lock requested	Action
Read	Read	OK
Read	Write	Wait
Write	Read	Wait
Write	Write	Wait

Locking Procedure in 2P Locking

- When an operation accesses an object:
  - if the object is not already locked, lock the object in the lowest appropriate mode & proceed.
  - if the object has a conflicting lock by another transaction, wait until object has been unlocked.
  - if the object has a non-conflicting lock by another transaction, share the lock & proceed.
  - if the object has a lower lock by the same transaction,
    - if the lock is not shared, promote the lock & proceed
    - else, wait until all shared locks are released, then lock & proceed
- When a transaction commits or aborts:
  - release all locks that were set by the transactions

R/W 2PL

T1: add 1% dividend to C based on balances of A and B

Read Lock A

```
x := A.getBalance()
```

Read Lock B

```
y := B.getBalance()
```

Read Lock C

```
z := C.getBalance()
```

Promote C to write

```
C.setBalance((x+y)*0.01+z)
```

Unlock A, B, C

T2: transfer 100 from A to B

Read Lock A

```
t := A.getBalance()
```

Read Lock B

```
u := B.getBalance()
```

Try to promote A to write lock

Promote A to write lock

```
A.setBalance(t-100)
```

Promote B to write lock

```
B.setBalance(u+100)
```

Unlock A, B

Why lock promotion is necessary

T1: add 1% dividend to C based on balances of A and B

Read Lock A

```
x := A.getBalance()
```

Read Lock B

```
y := B.getBalance()
```

Read Lock C

```
z := C.getBalance()
```

Promote C to write

```
C.setBalance((x+y)*0.01+z)
```

Unlock A, B, C

T2: transfer 100 from A to B

Read Lock A

```
t := A.getBalance()
```

Read Lock B

```
u := B.getBalance()
```

Try to promote A to write lock

Promote A to write lock

```
A.setBalance(t-100)
```

Promote B to write lock

```
B.setBalance(u+100)
```

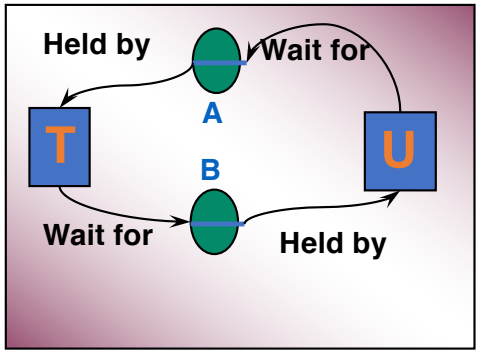
Unlock A, B

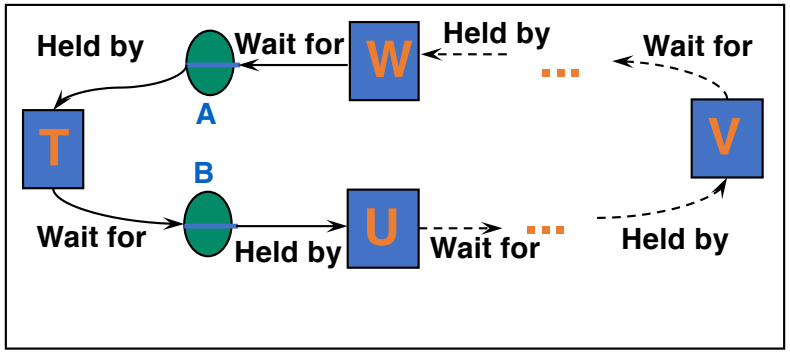
Why we need lock promotion

T1:	T2:
acquire R-lock on a	
a.read()	
	acquire R-lock on a
	a.read()
	release R-lock on a
release R-lock on a	
acquire W-lock on a	
a.write()	
commit	
release W-lock on a	
	acquire W-lock on a
	a.write()
	commit
	release W-lock on a

Deadlocks

- Necessary conditions for deadlocks
  - Non-shareable resources (locked objects)
  - No preemption on locks
  - Hold & Wait
  - Circular Wait (Wait-for graph)





Deadlock Resolution Using Timeout

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *a*		
		b.deposit(200)	write lock *b*
b.withdraw(100)			
•••	waits for *U*'s lock on *b*	*a.withdraw(200);*	waits for *T*'s lock on *a*
	(timeout elapses)	•••	
T's lock on *A* becomes vulnerable, unlock *a*, abort T		•••	
		a.withdraw(200);	write locks *a*
		commit	unlock *a*, *b*

Deadlock Strategies

- Timeout: how to set value?
  - Too large -> long delays
  - Too small -> false positives
- Deadlock prevention
  - Lock all objects at transaction start
  - Use lock ordering
- Deadlock Detection (later)
  - Maintain wait-for graph, look for cycle
  - Abort one transaction in cycle

Identify all conflicts

T1	T2
read X	read X
write Y	write Y
read Z	read A
read W	read B
write V	write C

Timestamp Ordering

- Assign each transaction a unique *timestamp (ts)*
  - Serialize transactions according to timestamps
- Keep track of timestamp last transaction to read and write an object
- Maintain two invariants:
  - If T writes O, last read and write timestamp must be lower than T's
  - If T reads O, last write timestamp must be lower than T
- If T tries to read/write object with higher timestamp, abort and rollback

```
T (1)                U (2)                V (3)
read X (X.rts=1)
write Y(Y.wts=1)
                     read X (X.rts=2)
                                          read Y (Y.rts = 3)
                                          write X (X.wts=3)
                     read Y (Y.rts=3)
write X: abort!
```