Locks and Deadlocks

Reference: George Coulouris, Jean Dollimore and Tim Kindberg, “Distributed Systems Concepts and Design”, Fifth Edition, Pearson Education, 2012

Concurrency Control Protocols

- Locks
- Optimistic control
- Timestamp Ordering

Concurrency Control Protocols

- Lock an object which is used by many transactions to avoid lost updates and Dirty reads.
- Server can lock any object that is about to be used by a client.
- If another client wants to access the same object, it has to wait until the object is unlocked in the end.

Lock Compatibility

For one object		Lock requested	
		read	write
Lock already set	none	OK	OK
	read	OK	wait
	write	wait	wait

Transactions *T* and *U* with exclusive locks

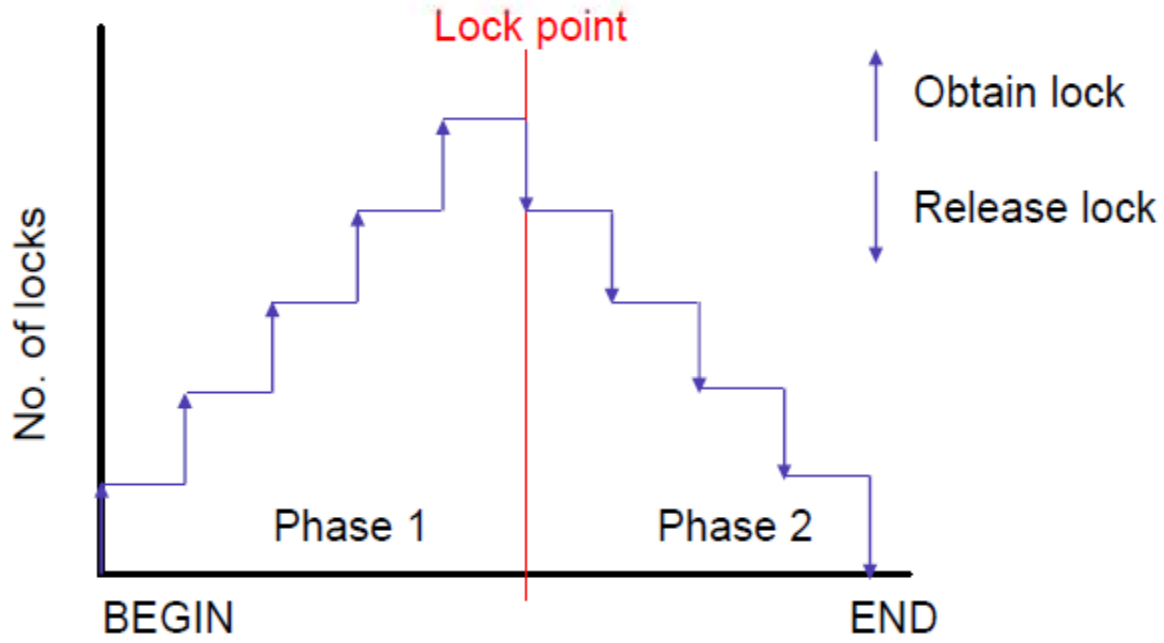
Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(bal*1.1)*		*b.setBalance(bal*1.1)*	
a.withdraw(bal/10)		*c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

Transactions *T* and *U* with exclusive locks

- All pairs of conflicting operations of two transactions should be executed in the same order.
- To ensure this, a transaction is not allowed any new locks after it has released a lock.
- The first phase of each transaction is a ‘growing phase’, during which new locks are acquired.
- In the second phase, the locks are released (a ‘shrinking phase’).
- This is called as **Two – Phase Locking.**
- **Strict Two Phase Locking**
- A transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted.

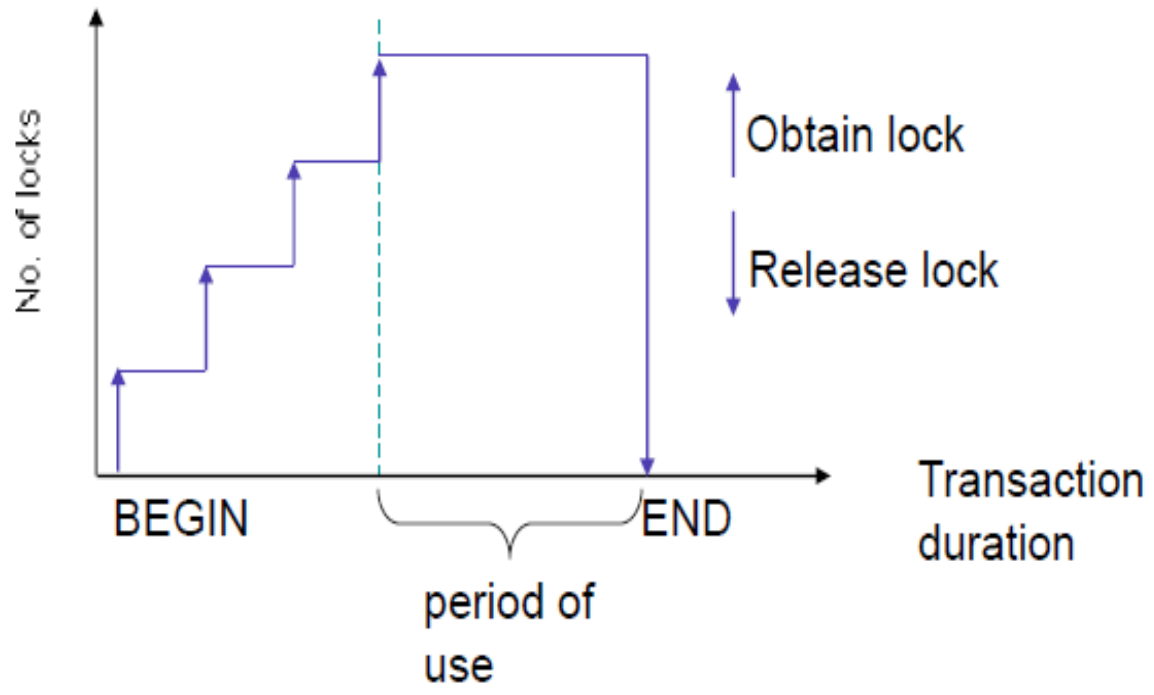
Two Phase Locking

- A transaction locks an object before using it.
- When an object is locked by another transaction, the requesting transaction must wait.
- When a transaction releases a lock, it may not request another lock.



Strict Two Phase Locking

Hold locks until the end.



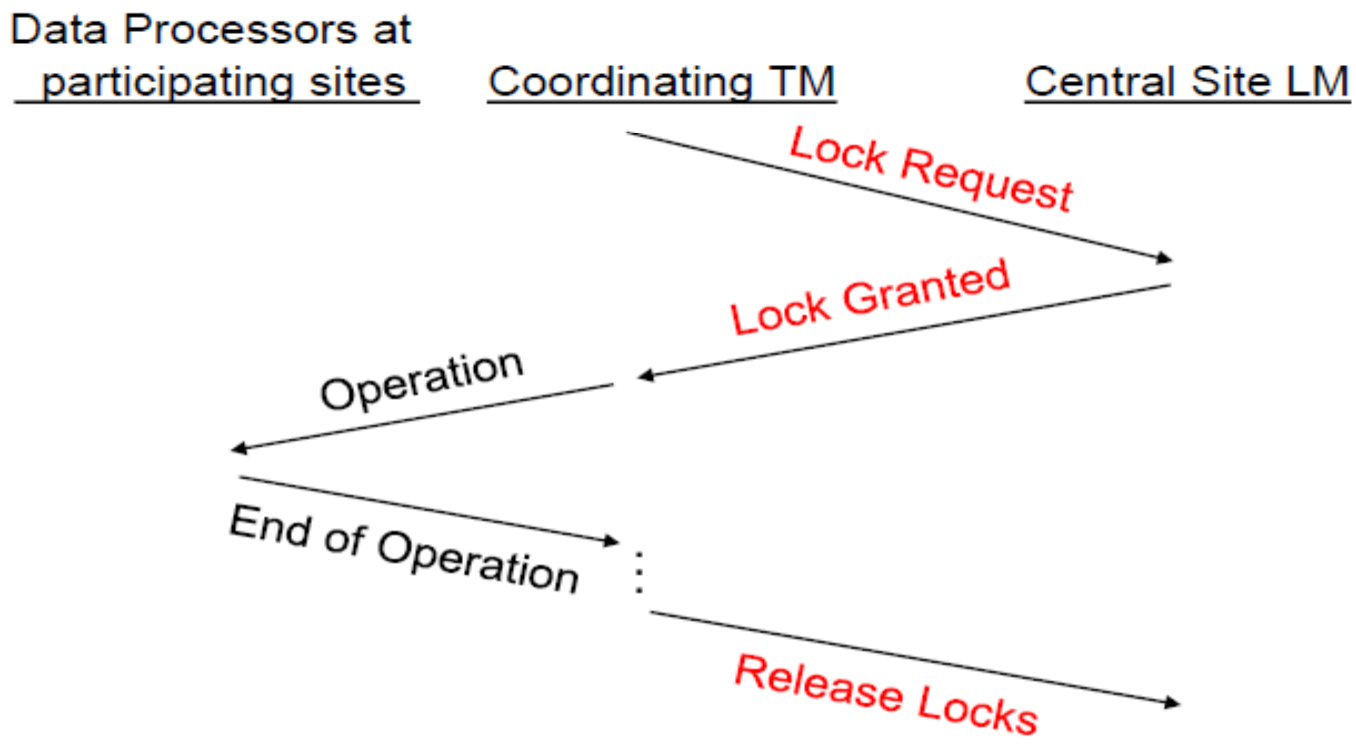
Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

Centralized Two Phase Locking

There is only one 2PL scheduler in the distributed system.

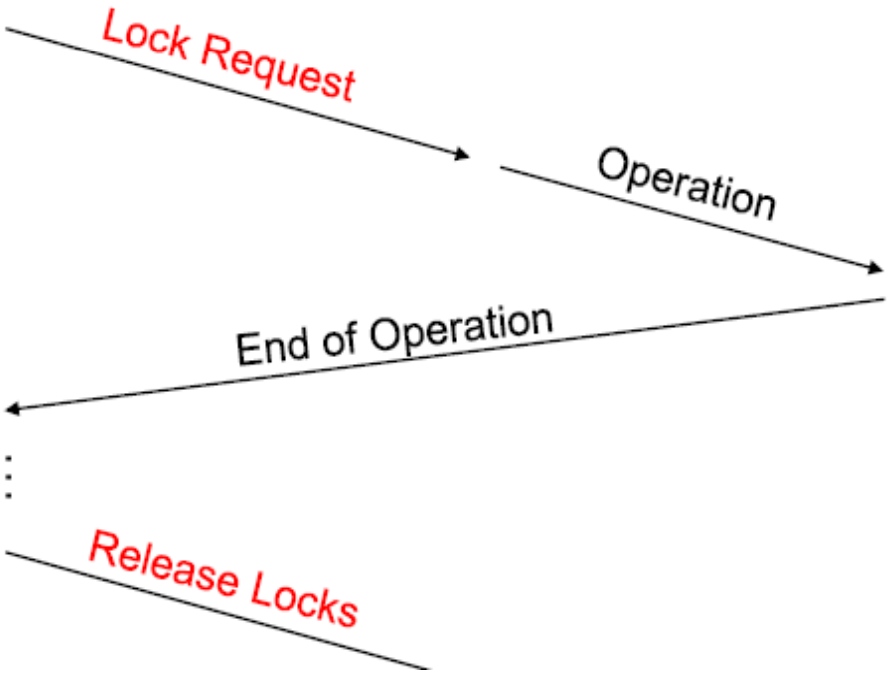
Lock requests are issued to the central scheduler.



Distributed Two Phase Locking

2PL schedulers are placed at each site. Each scheduler handles lock requests for objects at that site.

Coordinating TM Participating LMs Participating DPs



Two Version Locking

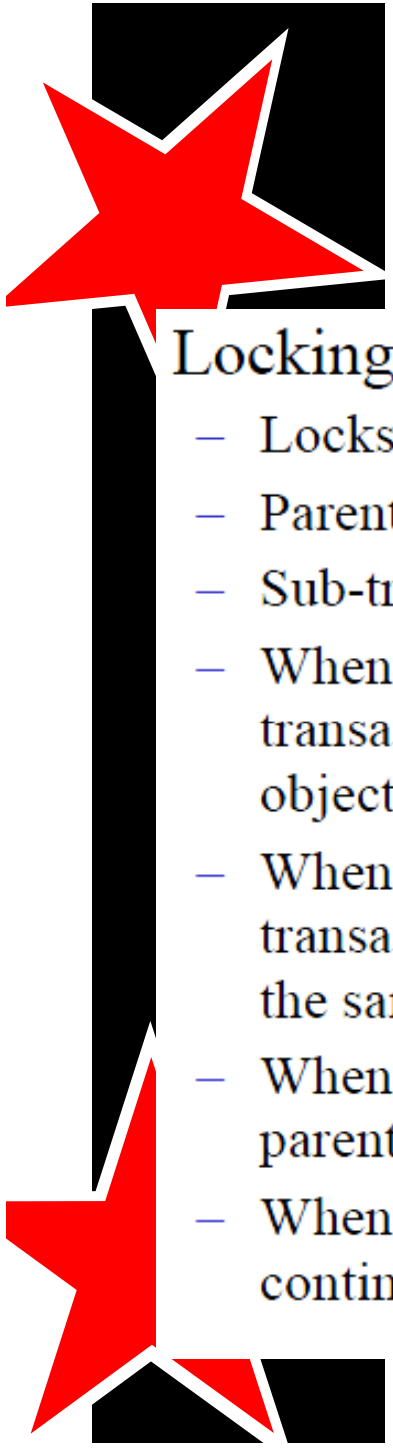
Allows one transaction to write tentative versions of objects while other transactions read from the committed versions of the same objects.

For one object		*Lock to be set*		
		read	*write*	*commit*
Lock already set	*none*	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	–
	commit	wait	wait	–

Hierarchic Locks

The granularity suitable for one operation is not appropriate for another operation.
In our banking example, the majority of the operations require locking at the granularity of an account
Intent locks are compatible with intent locks

For one object		*Lock to be set*			
		read	*write*	*I-read*	*I-write*
Lock already set	*none*	OK	OK	OK	OK
	read	OK	wait	OK	wait
	write	wait	wait	wait	wait
	I-read	OK	wait	OK	OK
	I-write	wait	wait	OK	OK



Locking rules for Nested Transactions

Locking rules for nested transactions

- Locks set by children are inherited by their parents
- Parents are not allowed to run concurrently with their children
- Sub-transactions at the same level are allowed to run concurrently
- When a subtransaction acquires a read lock on an object, no other transaction except only its parent can get a write lock on the same object
- When a subtransaction acquires a write lock on an object, no other transaction except only its parent can get a read or write lock on the same object
- When a subtransaction commits, its locks are inherited by its parent
- When a subtransaction aborts, its locks are discarded but its parent continue to retain the locks if the parent already has them

Lock Problem

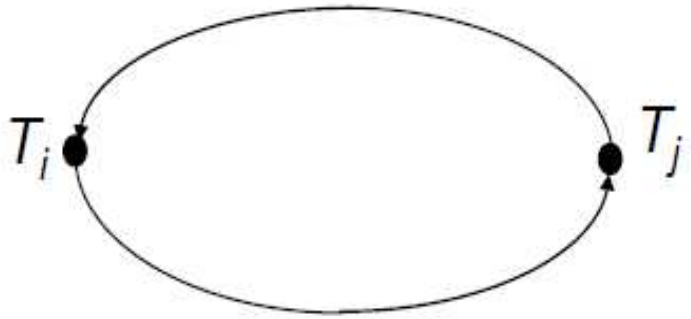
- Locks lead to Deadlock

Deadlock

- Deadlock is a state in which each member of a group of transactions is waiting for some other member in the same group to release a lock.

Deadlock

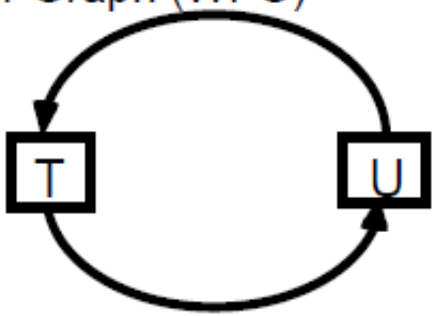
- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- Wait-for graph
  - If transaction T_i waits for another transaction T_j to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG.

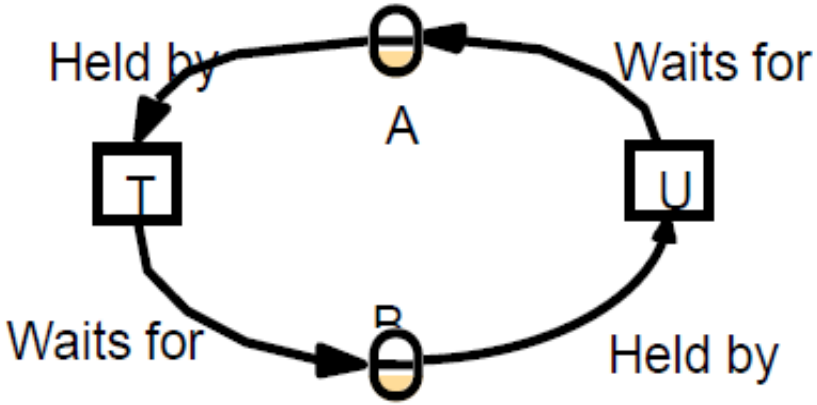


Deadlock due to Locking

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
		•••	
•••		•••	

Wait-for Graph (WFG)





Deadlocks Prevention

- One way to prevent deadlock is to obtain locks on all the objects by the transaction before it starts.

Deadlock Recovery

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* unlock *A*, *B*