Transactions

Operating Systems

Transactions

- Motivation
  - Provide atomic operations at servers that maintain shared data for clients
  - Provide recoverability from server crashes
- Properties
  - Atomicity, Consistency, Isolation, Durability (ACID)
- Concepts: commit, abort

Operations of the *Account* interface

deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() -> amount
return the balance of the account
setBalance(amount)
set the balance of the account to amount

Operations of the Branch interface

create(name) -> account
create a new account with a given name
lookUp(name) -> account
return a reference to the account with the given name
branchTotal() -> amount
return the total of all the balances at the branch

A client's banking transaction

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

Operations in *Coordinator* interface

openTransaction() -> trans;

starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans) -> (commit, abort);

ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);

aborts the transaction.

Transaction life histories

Successful	*Aborted by client*	*Aborted by server*	
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
•	•	server aborts transaction →	•
•	•		•
operation	*operation*		*operation ERROR reported to client*
closeTransaction	*abortTransaction*		

Concurrency control

- Motivation: without concurrency control, we have lost updates, inconsistent retrievals, dirty reads, etc. (see following slides)
- Concurrency control schemes are designed to allow two or more transactions to be executed correctly while maintaining serial equivalence
  - Serial Equivalence is correctness criterion
    - Schedule produced by concurrency control scheme should be equivalent to a serial schedule in which transactions are executed one after the other
- Schemes: locking, optimistic concurrency control, time-stamp based concurrency control
  - We will only study locking in this class

The lost update problem

Transaction *T*: *balance = b.getBalance(); b.setBalance(balance*1.1); a.withdraw(balance/10)*		**Transaction *U*:** *balance = b.getBalance(); b.setBalance(balance*1.1); c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

The inconsistent retrievals problem

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

A serially equivalent interleaving of *T* and *U*

Transaction **T**: *balance = b.getBalance()* *b.setBalance(balance*1.1)* *a.withdraw(balance/10)*		*Transaction* **U**: *balance = b.getBalance()* *b.setBalance(balance*1.1)* *c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

A serially equivalent interleaving of *V* and *W*

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

Read and *write* operation conflict rules

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

A non-serially equivalent interleaving of operations of transactions *T* and *U*

Transaction *T*:	**Transaction *U*:**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

A dirty read when transaction *T* aborts

Transaction *T*: *a.getBalance()* *a.setBalance(balance + 10)*		**Transaction *U*:** *a.getBalance()* *a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

Transactions *T* and *U* with exclusive locks

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *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 15

Lock compatibility

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

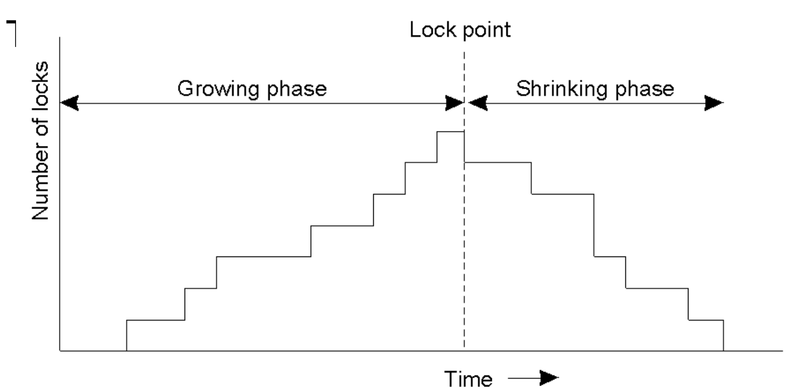
Transactions 16

Use of locks in strict two-phase locking

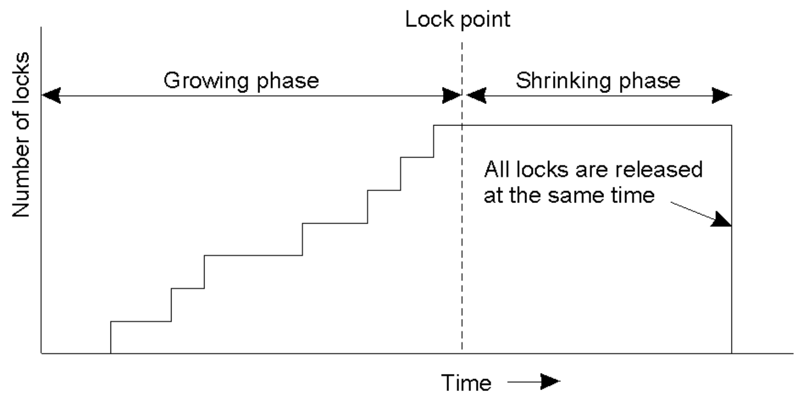
1. When an operation accesses an object within a transaction:
   - (a) If the object is not already locked, it is locked and the operation proceeds.
   - (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   - (c) If the object has a non-conflicting 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.

Two-Phase Locking (1)

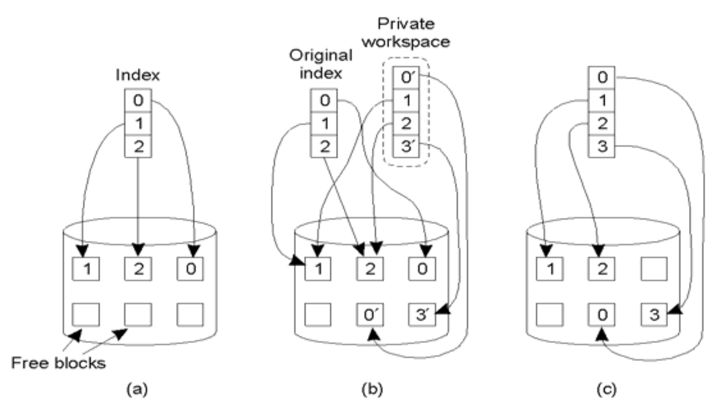
❒



Strict Two-Phase Locking (2)



Implementing Transactions: Private Workspace



a) The file index and disk blocks for a three-block file
b) The situation after a transaction has modified block 0 and appended block 3
c) After committing

Implementing Transactions: Writeahead Log

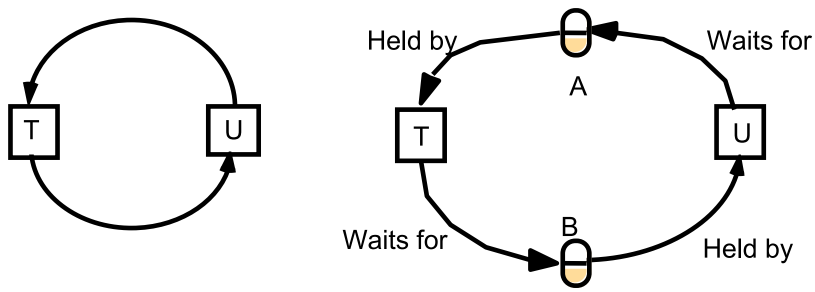
	Log	Log	Log
x = 0;			
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

- a) A transaction
- b) – d) The log before each statement is executed

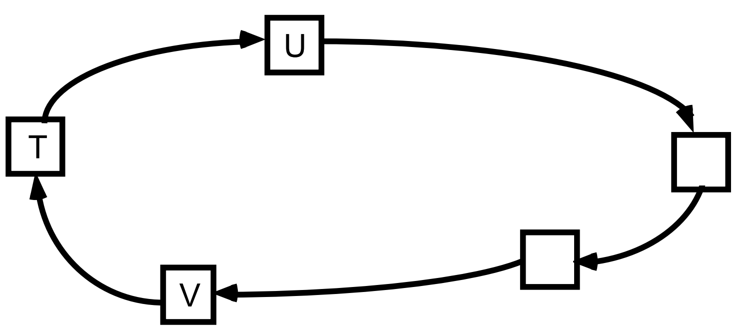
Deadlock with write locks

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*
•••		•••	
•••		•••	
		•••	

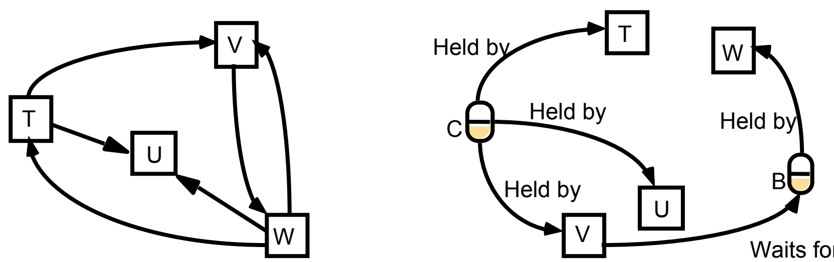
The wait-for graph



A cycle in a wait-for graph



Another wait-for graph



Resolution of deadlock

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*