Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta and Nikita Borisov

Logistics

- Midterm assessments are visible.
- HW2 solutions will be released shortly.
- HW3 is due next week.
- Hope all 4 credits students have started work on MP2.

Agenda for the next 2-3 classes

- Transaction Processing and Concurrency Control
    - Chapter 16
        - Transaction semantics: ACID
        - Isolation and serial equivalence
        - Conflicting operations
        - Two-phase locking
        - Deadlocks
        - Timestamped ordering

- First focus on transactions executed on a single server.
- Look into distributed transactions later (Chapter 17)

Today's Agenda

- Transaction Processing and Concurrency Control
  - Chapter 16
    - Transaction semantics: ACID
    - Isolation and serial equivalence
    - Conflicting operations
    - Two-phase locking
    - Deadlocks
    - Timestamped ordering

- First focus on transactions executed on a single server.
- Look into distributed transactions later (Chapter 17)

Transaction

- Series of *operations* executed by a client on a server (or a set of servers).

- Example: Switch from T4 to T3 section:
  rosters.remove("ece428", "t4", student.name)
  student.schedule.remove("ece428", "t4")
  student.schedule.add("ece428", "t3")
  rosters.add("ece428", "t3", student.name)

Transaction

- Another example:

Client code:

```
int transaction_id = openTransaction();
x = server.getFlightAvailability(ABC123, date);     // read(ABC123, date)
if (x > 0)
    y = server.bookTicket(ABC123, date);            // write(ABC123, date)
closeTransaction(transaction_id);
```

Transaction Properties

- Atomic: all-or-nothing
  - Transaction either executes completely or not at all
- Consistent: required rules are maintained
- Isolation: multiple transactions do not interfere with each other
  - Equivalent to running transactions in isolation
- Durability: values preserved even after crashes

Transaction Properties

- Atomic: all-or-nothing
  - Transaction either executes completely or not at all
- Consistent: rules maintained
- Isolation: multiple transactions do not interfere with each other
  - Equivalent to running transactions in isolation
- Durability: values preserved even after crashes

ACID properties

Atomicity

- All-or-nothing
  - Transaction either executes completely or not at all
- What can happen after partial execution?

```
rosters.remove(“ece428”, “t4”, student.name)
student.schedule.remove(“ece428”, “t4”)
student.schedule.add(“ece428”, “t3”)
rosters.add(“ece428”, “t3”, student.name)
```

Atomicity

- All-or-nothing
  - Transaction either executes completely or not at all
- Make tentative updates to data.
- **Commit** transaction to make tentative updates permanent.
- **Abort** transaction to roll back to previous values.

Consistency

Various rules about state of objects must be maintained:

- E.g. class enrollment limit, schedule can’t conflict
- Account balances have to stay positive
- Consistency must be maintained at end of transaction.
- Checked at commit time, abort if not satisfied

```
rosters.remove(“ece428”, “t4”, student.name)
student.schedule.remove(“ece428”, “t4”)
student.schedule.add(“ece428”, “t3”)
rosters.add(“ece428”, “t3”, student.name)
```

Durability

- Committed transactions must persist:
  - Client crashes
  - Server crashes
- How do we ensure this?
  - Permanent storage
  - Replication

Isolation

Multiple clients may execute transactions concurrently on one server.

What could go wrong?

What could go wrong?

Transaction T1	**Transaction T2**	At Server: seats = 10
x = getSeats(ABC123);		
// x = 10	x = getSeats(ABC123);	
if(x > 1)	if(x > 1) **// x = 10**	
x = x – 1;		
write(x, ABC123);		seats = 9
	x = x – 1;	
	write(x, ABC123);	seats = 9
commit		
	commit	

1. Lost Update Problem

Transaction T1	**Transaction T2**	At Server: seats = 10
x = getSeats(ABC123);		
// x = 10	x = getSeats(ABC123);	
if(x > 1)	if(x > 1) **// x = 10**	
x = x – 1;		
write(x, ABC123);		seats = 9
	x = x – 1;	
	write(x, ABC123);	seats = 9
commit		
	commit	

T1’s or T2’s update was lost!

What else could go wrong?

At Server:
ABC123 = 10
ABC789 = 15

Transaction T1	**Transaction T2**
x = getSeats(ABC123);	
y = getSeats(ABC789);	
write(x-5, ABC123);	
// ABC123 = 5 now	x = getSeats(ABC123);
	y = getSeats(ABC789);
write(y+5, ABC789);	**// x = 5, y = 15**
	print(“Total:” x+y);
commit	**// Prints “Total: 20”**
	commit

2. Inconsistent Retrieval Problem

Transaction T1

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);
   // ABC123 = 5 now

write(y+5, ABC789);


commit
```

Transaction T2

```
x = getSeats(ABC123);
y = getSeats(ABC789);
     // x = 5, y = 15

print("Total:" x+y);
       // Prints "Total: 20"
commit
```

At Server:
ABC123 = 10
ABC789 = 15

T2's sum is the wrong value! Should have been "Total: 25"

Isolation

Multiple clients executing transactions concurrently on one server.

What could go wrong?

- Lost Update Problem
- Inconsistent Retrieval Problem

How to prevent transactions from affecting each other?

Isolation

How to prevent transactions from affecting each other?

- Option 1: Execute them serially at the server (one at a time).
    - Grab a global lock before executing any transaction, release the lock after the transaction has committed (or aborted).
- But this reduces number of concurrent transactions
    - *Transactions per second* directly related to revenue of companies
        - This metric needs to be maximized

Goal: increase concurrency while maintaining correctness (ACID).

Concurrent Transactions

Goal: increase concurrency while maintaining correctness (ACID).

- How do we increase concurrency?
  - Instead of targeting strict serial execution, target **serial equivalence**.

Interleaving

- An ordered sequence of the operations across multiple transactions, where each transaction's operations follows the order defined by the transaction.

- E.g., if $T_1 = \{op_1, op_2, op_3\}$ and $T_2 = \{op^4, op^5, op^6\}$

then $O = \{op_1, op_2, op^4, op_3, op^5, op^6\}$ is a sample interleaving.

Interleaving: Another example

Transaction T1

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);



write(y+5, ABC789);



commit
```

Transaction T2

```



x = getSeats(ABC123);
y = getSeats(ABC789);



print(“Total:” x+y);

commit
```

Interleaving

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);
x = getSeats(ABC123);
y = getSeats(ABC789);
write(y+5, ABC789);

print(“Total:” x+y);
commit
commit
```

Concurrent Transactions

- Allowing transaction operations to be interleaved with one-another increases concurrency.
- To avoid transactions from affecting one another, the interleaving of operations across transactions must be *serially equivalent.*

Serial Equivalence

- An interleaving (say O) of transaction operations is serially equivalent iff (if and only if):
  - There is some ordering (O') of those transactions, one at a time,
  - where the operations of each transaction occur consecutively (in a batch)
  - which gives the same end-result (for all objects and transactions) as the original interleaving O
- Says: Cannot distinguish end-result of real operation O from (fake) serial transaction order O'
- E.g., if $T_1 = \{op_1, op_2, op_3\}$ and $T_2 = \{op^4, op^5, op^6\}$

  then $O = \{op_1, op_2, op^4, op_3, op^5, op^6\}$ is a serially equivalent, if:
  - end result of O is same as $\{op_1, op_2, op_{3,} op^4, op^5, op^6\}$
  - **Or** end result of O is same as $\{op^4, op^5, op^6, op_1, op_2, op_{3,} \}$

1. Lost Update Problem

Transaction T1	**Transaction T2**	At Server: seats = 10
x = getSeats(ABC123);		
// x = 10	x = getSeats(ABC123);	
if(x > 1)	if(x > 1) // x = 10	
x = x – 1;		
write(x, ABC123);		seats = 9
	x = x – 1;	
	write(x, ABC123);	seats = 9
commit		
	commit	

T1's or T2's update was lost!
Not serially equivalent.

2. Inconsistent Retrieval Problem

Transaction T1	**Transaction T2**
x = getSeats(ABC123);	
y = getSeats(ABC789);	
write(x-5, ABC123);	
// ABC123 = 5 now	x = getSeats(ABC123);
	y = getSeats(ABC789);
write(y+5, ABC789);	**// x = 5, y = 15**
	print(“Total:” x+y);
commit	
	// Prints “Total: 20”
	commit

At Server:
ABC123 = 10
ABC789 = 15

T2’s sum is the wrong value! Should have been “Total: 25”

Not serially equivalent.

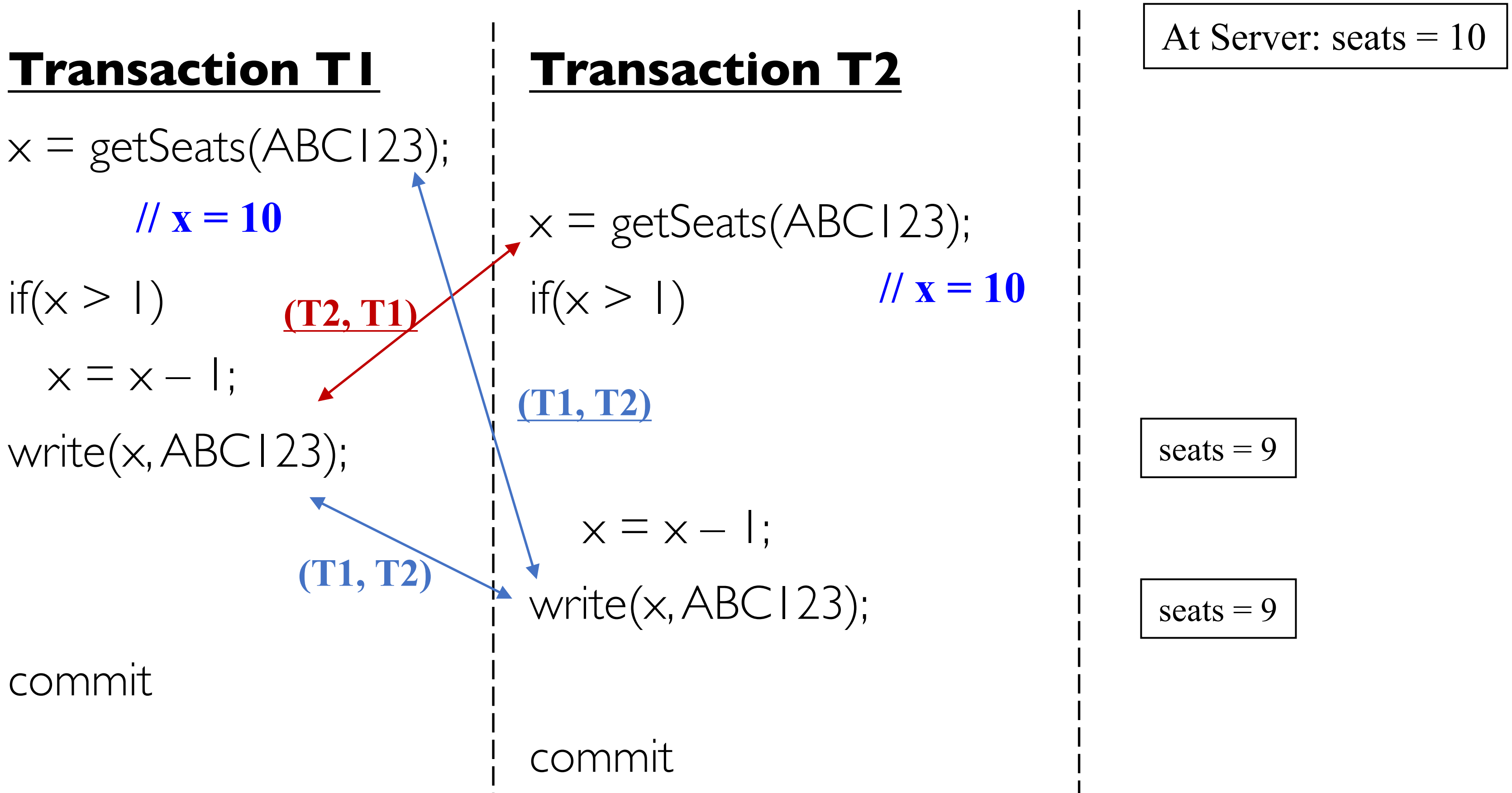
Checking for Serial Equivalence

- An operation has an effect on
  - The server object if it is a write
  - The client (returned value) if it is a read
- Two operations are said to be conflicting operations, if their *combined effect* depends on the order they are executed
  - read(x) and write(x):
  - write(x) and read(x):
  - write(x) and write(x):
  - read(x) and read(x):
    - swapping them doesn't change their effects
  - read/write(x) and read/write(y):
    - ok to swap them as they access different objects.

Checking for Serial Equivalence (cont.)

- *Two transactions are serially equivalent if and only if all pairs of conflicting operations (pair containing one operation from each transaction) are executed in the same order (transaction order) for all objects (data) they both access.*
    - Take all pairs of conflict operations, one from T1 and one from T2
    - If the T1 operation was reflected first on the server, mark the pair as "(T1,T2)", otherwise mark it as "(T2,T1)"
    - All pairs should be marked as either "(T1,T2)" or all pairs should be marked as "(T2,T1)".

1. Lost Update Problem



Transaction T1

x = getSeats(ABC123);

// x = 10

if(x > 1)

x = x – 1;

write(x, ABC123);

commit

Transaction T2

x = getSeats(ABC123);

if(x > 1)

// x = 10

x = x – 1;

write(x, ABC123);

commit

At Server: seats = 10

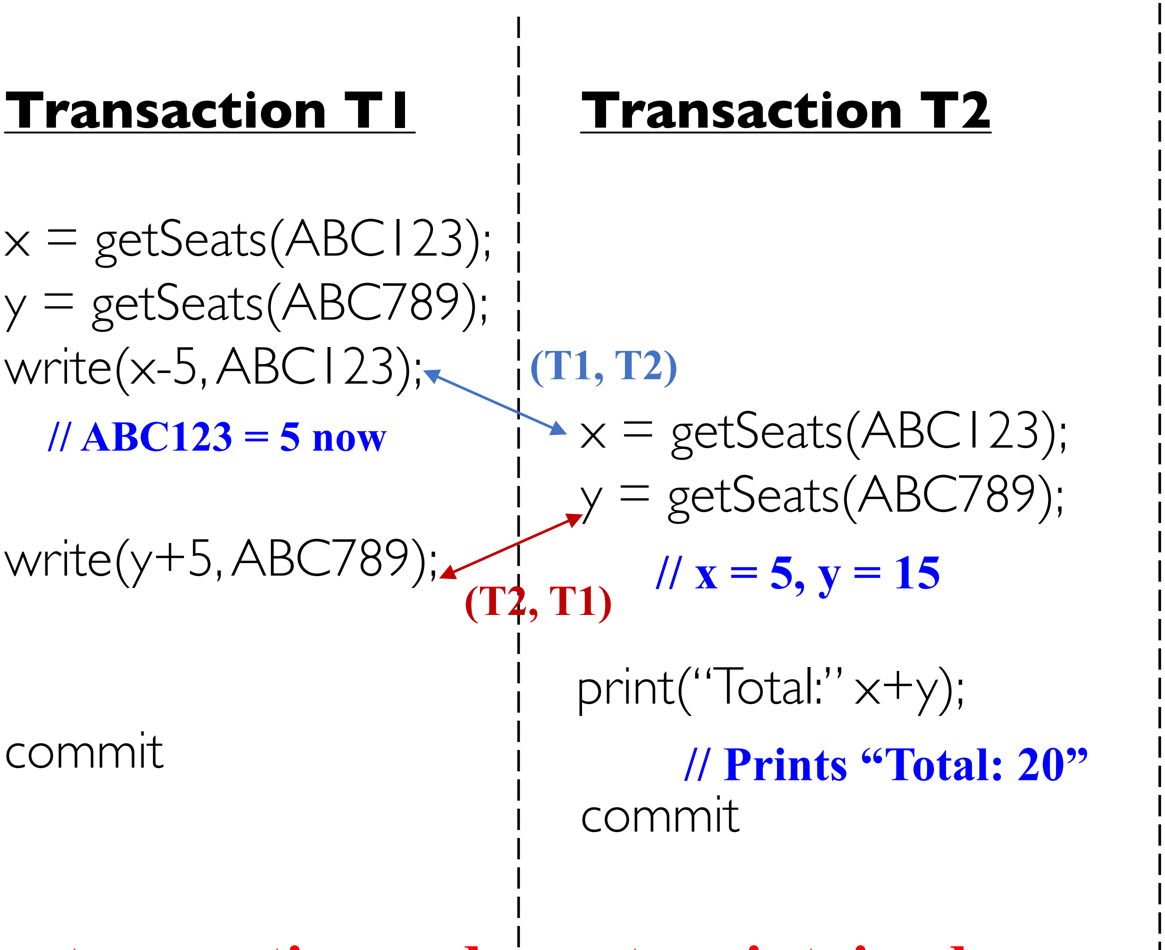
seats = 9

seats = 9

Same transaction order not maintained across conflicting operations.

Not serially equivalent.

2. Inconsistent Retrieval Problem



Transaction T1

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);
   // ABC123 = 5 now

write(y+5, ABC789);

commit
```

Transaction T2

```
x = getSeats(ABC123);
y = getSeats(ABC789);
    // x = 5, y = 15

print("Total:" x+y);
    // Prints "Total: 20"
commit
```

At Server:
ABC123 = 10
ABC789 = 15

Same transaction order not maintained across conflicting operations.

Not serially equivalent.

How do we handle such conflicts?

- Option 1: Execute them serially at the server (one at a time).
- Option 2:
  - At commit point of a transaction T, check for serial equivalence with all other transactions
    - Can limit to transactions that overlapped in time with T
  - If not serially equivalent
    - Abort T
    - Roll back (undo) any writes that T did to server objects
- Aborting all such transactions => wasted work.
  - Can we do better?
  - Can we prevent violations from occurring?

Two Approaches

- Preventing isolation from being violated can be done in two ways
  1. Pessimistic concurrency control
  2. Optimistic concurrency control

Pessimistic vs. Optimistic

- Pessimistic: assume the worst, prevent transactions from accessing the same object
    - E.g., Locking

- Optimistic: assume the best, allow transactions to write, but check later
    - E.g., Check at commit time

Pessimistic vs. Optimistic

- Pessimistic: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking

- Optimistic: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time

Pessimistic: Exclusive Locking

- Grabbing a global lock is wasteful
    - what if no two transactions access the same object?
- Each object has a lock
    - At most one transaction can be inside lock
    - Before reading or writing object O, transaction T must call lock(O)
        - Blocks if another transaction already inside lock
    - After entering lock(O), T can read and write O multiple times
    - When done (or at commit point), T calls unlock(O)
        - If other transactions waiting at lock(O), allows one of them in
- Sound familiar?
    - This is Mutual Exclusion!

Can we improve concurrency?

- More concurrency => more transactions per second => more revenue ($$$)

- Real-life workloads have a lot of read-only or read-mostly transactions
  - Exclusive per-object locking reduces concurrency
  - Ok to allow two transactions to concurrently read an object, since read-read is not a conflicting pair

Another approach: Read-Write Locks

- Each object has a lock that can be held in one of two modes
  - Read mode: multiple transactions allowed in
  - Write mode: exclusive lock
- Before first reading O, transaction T calls read_lock(O)
  - T allowed in (does not wait on the lock) only if *all* transactions inside lock for O all entered via read mode
  - Not allowed (i.e. must wait) if *any* transaction inside lock for O entered via write mode

Read Locks Example

read_lock(A)

read(A)

Allowed!

read_lock(A)

read(A)

write_lock(A)

write(A)

Blocked!

read_lock(A)

read(A)

Read-Write Locks (contd)

- Before first writing O, call write_lock(O)
  - Allowed in only if no other transaction inside lock
- If T already holds read_lock(O), and wants to write, call write_lock(O) to *promote* lock from read to write mode
  - Succeeds only if no other transactions in write mode or read mode
  - Otherwise, T blocks
- Unlock(O) called by transaction T releases any lock on O by T

Write Locks Example

read_lock(A)	
read(A)	
	Blocked!
	write_lock(A)
	write(A)

write_lock(A)	
write(A)	
	Blocked!
	write_lock(A)
	write(A)

Write Locks Example

Within a single transaction

read_lock(A)

read(A)

Promoted and allowed!

write_lock(A)

write(A)

Write Locks Example

read_lock(A)

read(A)

Allowed!

read_lock(A)

read(A)

Blocked!

write_lock(A)

write(A)

When to release locks?

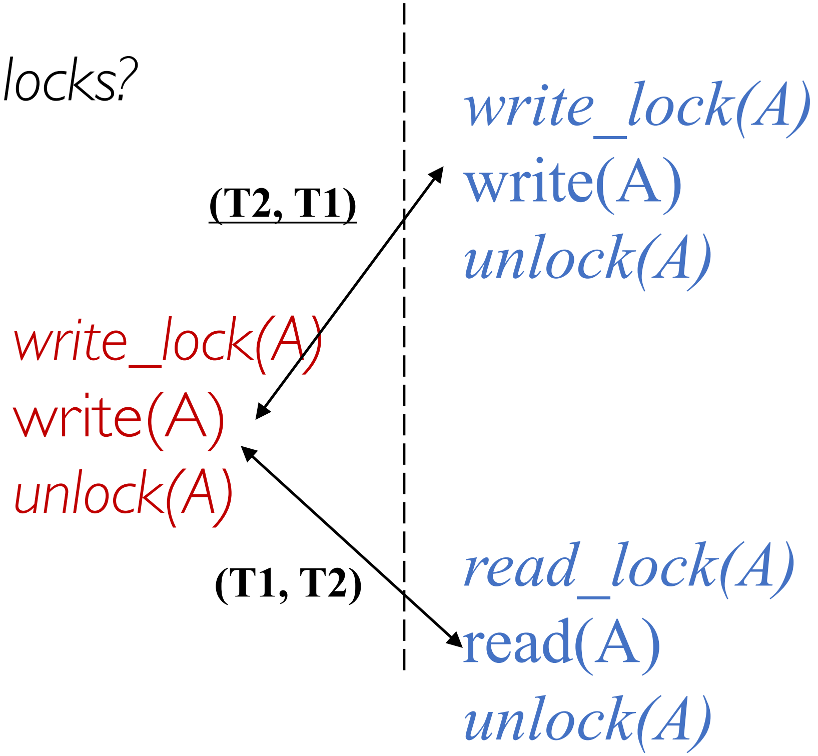
- We can have per-object locks in two modes to increase concurrency.
- Grab the object's lock in the appropriate mode when trying to access an object.
- *When to release locks?*

	write_lock(A)
write_lock(A)	write(A)
write(A)	*unlock(A)*
unlock(A)	*read_lock(A)*
	read(A)
	unlock(A)

Is this a good idea?

When to release locks?

- We can have per-object locks in two modes to increase concurrency.
- Grab the object’s lock in the appropriate mode when trying to access an object.
- *When to release locks?*



write_lock(A)
write(A)
unlock(A)

(T2, T1)

write_lock(A)
write(A)
unlock(A)

(T1, T2)

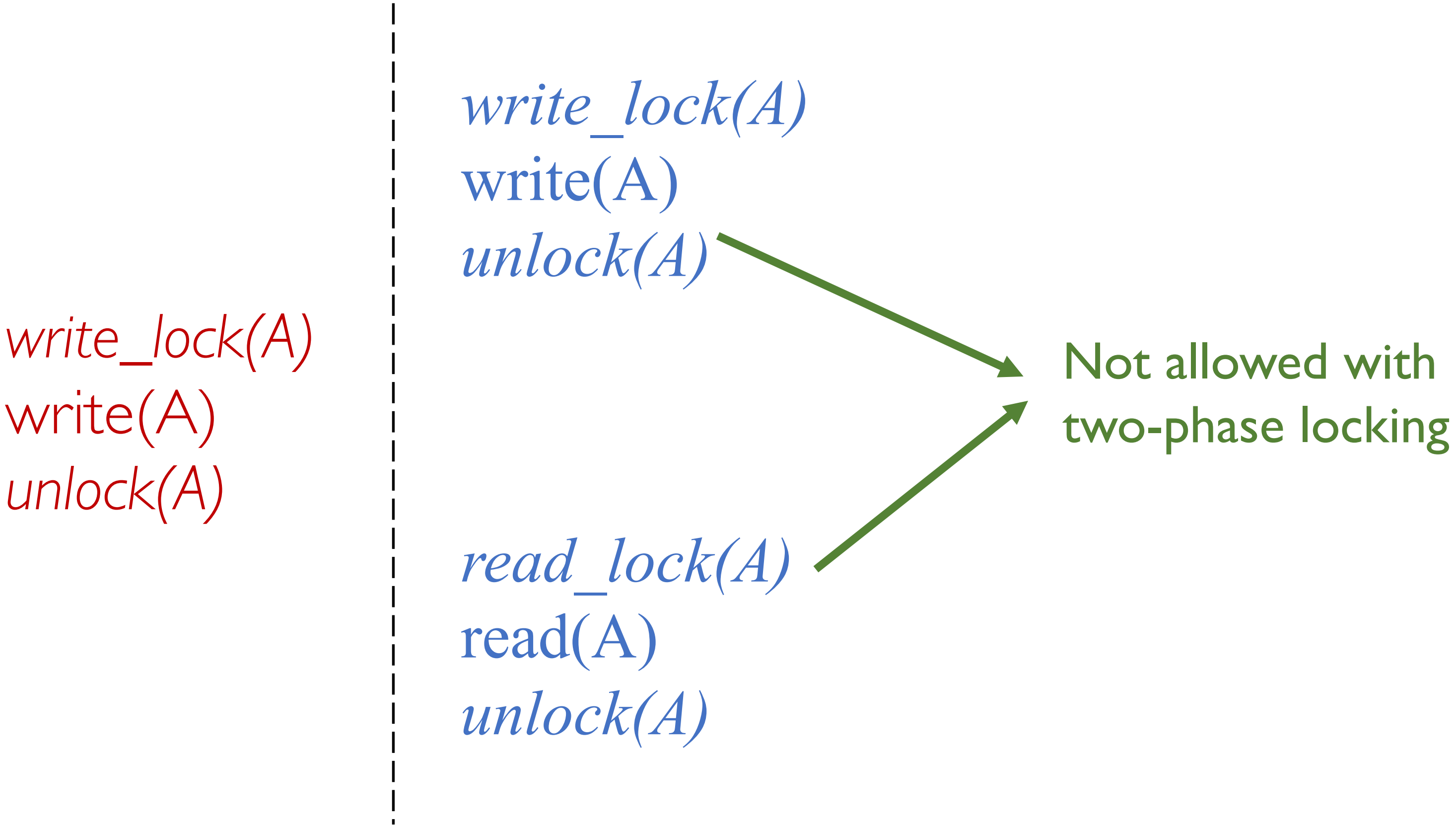
read_lock(A)
read(A)
unlock(A)

Not
serially
equivalent

Guaranteeing Serial Equivalence with Locks

- Two-phase locking
  - A transaction cannot acquire (or promote) any locks after it has started releasing locks
  - Transaction has two phases
    1. Growing phase: only acquires or promotes locks
    2. Shrinking phase: only releases locks
       - Strict two phase locking: releases locks only at commit point

Two-phase Locking



write_lock(A)
write(A)
unlock(A)

write_lock(A)
write(A)
unlock(A)

read_lock(A)
read(A)
unlock(A)

Not allowed with two-phase locking

Not serially equivalent

Two-phase Locking

	write_lock(A)
	write(A)
	~~*unlock(A)*~~
write_lock(A)	
blocked	
	~~*read_lock(A)*~~
	read(A)
	unlock(A)
write(A)	
unlock(A)	

Serially equivalent!

Why two-phase locking => Serial Equivalence?

- Proof by contradiction
- Assume two phase locking system where serial equivalence is violated for some two transactions T1, T2
- Two facts must then be true:
    - (*A*) For some object O1, there were conflicting operations in T1 and T2 such that the time ordering pair is (T1, T2)
    - (*B*) For some object O2, the conflicting operation pair is (T2, T1)
    - (*A*) => T1 released O1's lock and T2 acquired it after that

      => T1's shrinking phase is before or overlaps with T2's growing phase
- Similarly, *(B)* => T2's shrinking phase is before or overlaps with T1's growing phase
- But both these cannot be true!

Lost Update Example with 2P Locking

Transaction T1	**Transaction T2**
read_lock(x)	
x = getSeats(ABC123);	
	read_lock(x)
	x = getSeats(ABC123);
if(x > 1)	if(x > 1)
x = x – 1;	
write_lock(x) *Blocked!*	
write(x, ABC123);	
	x = x – 1;
	write_lock(x) *Blocked!*
	write(x, ABC123);
commit	
	commit

Deadlock!

Downside of Locking

- Deadlock!

Deadlock Example

Transaction T1

read_lock(x)

x = getSeats(ABC123);

if(x > 1)

  x = x – 1;

write_lock(x) *Blocked!*

write(x, ABC123);

unlock(x)
commit

Transaction T2

read_lock(x)

x = getSeats(ABC123);

if(x > 1)

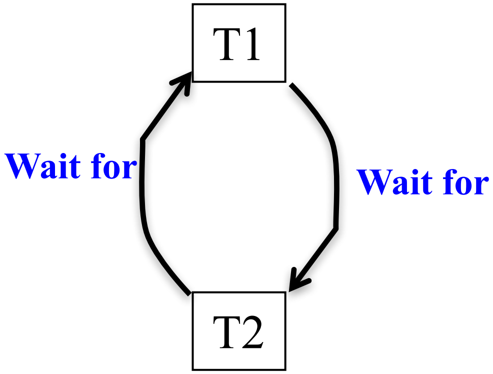
 x = x – 1;

write_lock(x) *Blocked!*

write(x, ABC123);

unlock(x)
commit

Deadlock!



When do deadlocks occur?

- 3 necessary conditions for a deadlock to occur
  1. Some objects are accessed in exclusive lock modes
  2. Transactions holding locks are not preempted
  3. There is a circular wait (cycle) in the Wait-for graph

- "Necessary" = if there's a deadlock, these conditions are all definitely true

- (Conditions not sufficient: if they're present, it doesn't imply a deadlock is present.)

Combating Deadlocks

1. Lock all objects in the beginning in a single atomic step.
   - no circular wait-for graph created (3rd deadlock condition breaks)
   - may not know of all operations a priori.
2. Lock timeout: abort transaction if lock cannot be acquired within timeout
   - (2nd deadlock condition breaks)
   - Expensive; leads to wasted work
   - How to determine the timeout value?
     - Too large: long delays
     - Too small: false positives.
3. Deadlock Detection:
   - keep track of Wait-for graph, and find cycles in it (e.g., periodically)
   - If find cycle, there's a deadlock

     ⇒Abort one or more transactions to break cycle (2nd deadlock condition breaks)

Concurrency Control: Two approaches

- Pessimistic: assume the worst, prevent transactions from accessing the same object
    - E.g., Locking

- Optimistic: assume the best, allow transactions to write, but check later
    - E.g., Check at commit time
    - Next class!