Distributed Systems

CS425/ECE428

Acknowledgements for the materials: Indy Gupta and Radhika Mittal

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

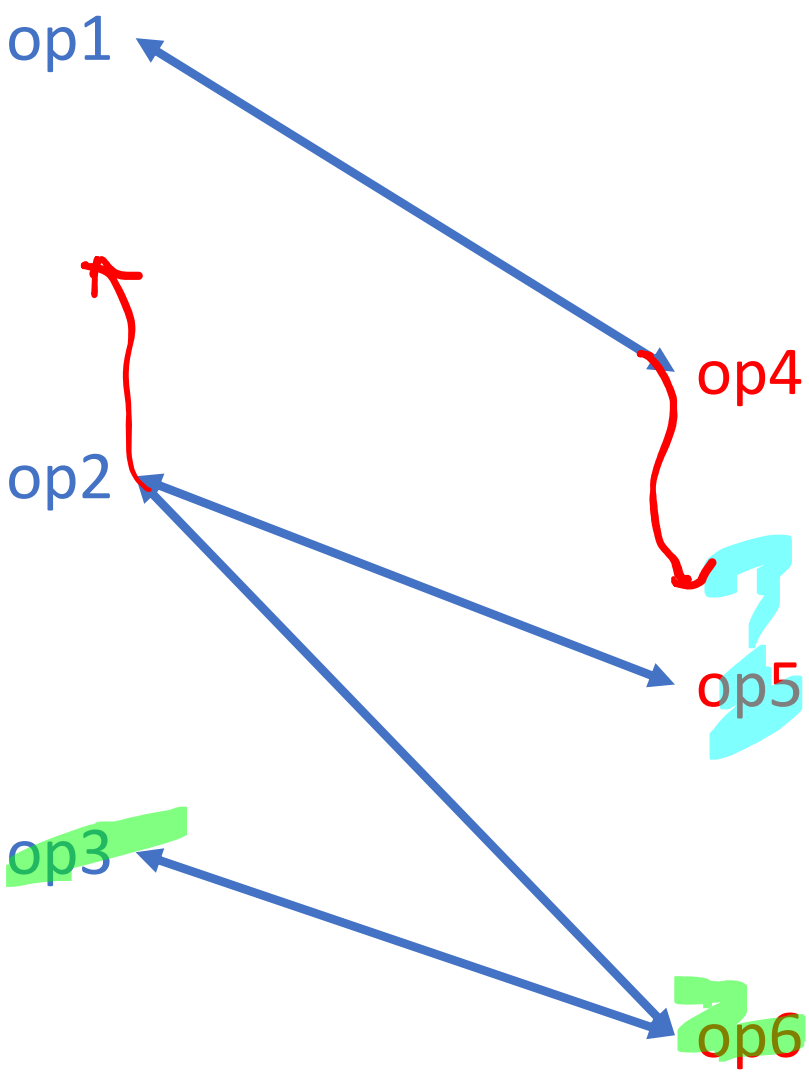
- Transaction Processing and Concurrency Control
  - Chapter 16
    - Isolation and serial equivalence
    - Conflicting operations
    - Two-phase locking
    - Deadlocks
    - Timestamped ordering

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

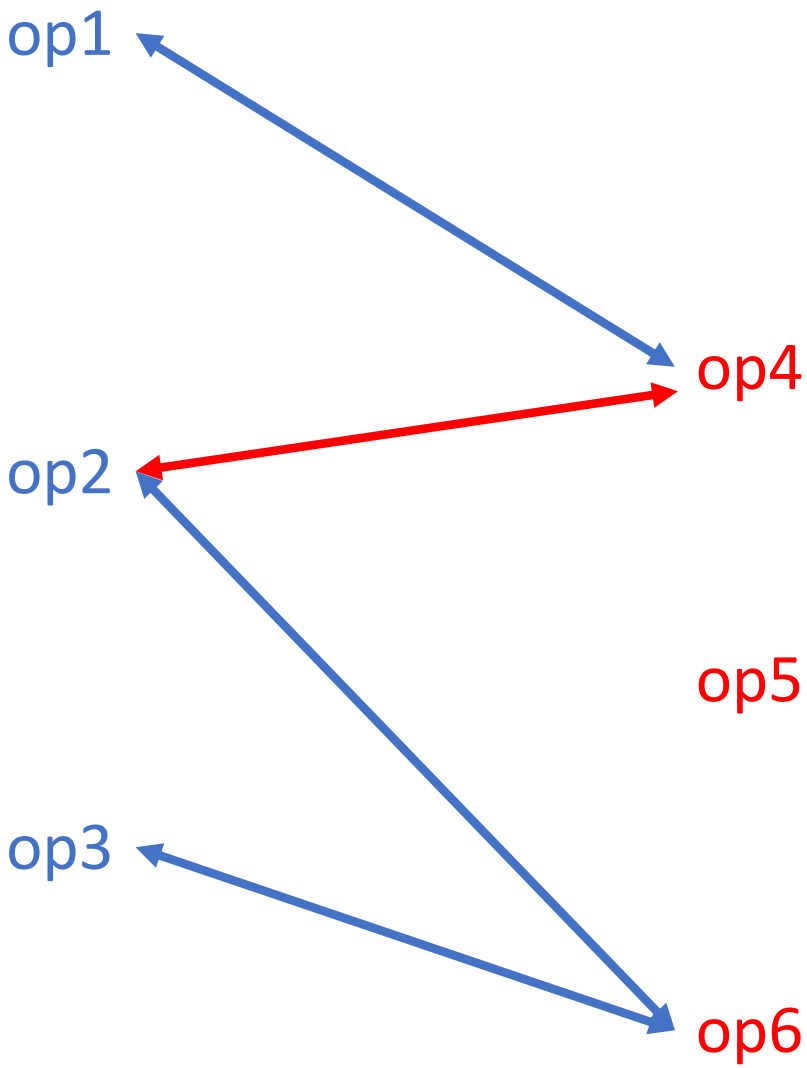
Serial Equivalence Recap

- A *serially equivalent* interleaving is equivalent in terms of its *combined effect* to executing transactions serially (T1;T2 or T2;T1)
- Two operations *conflict* if their combined effect depends on their order
  - Read/write or write/write of the same variable
- An interleaving is serially equivalent if and only if *every* pair of conflicting operations occurs in a fixed transaction order (T1 then T2 or T2 then T1)

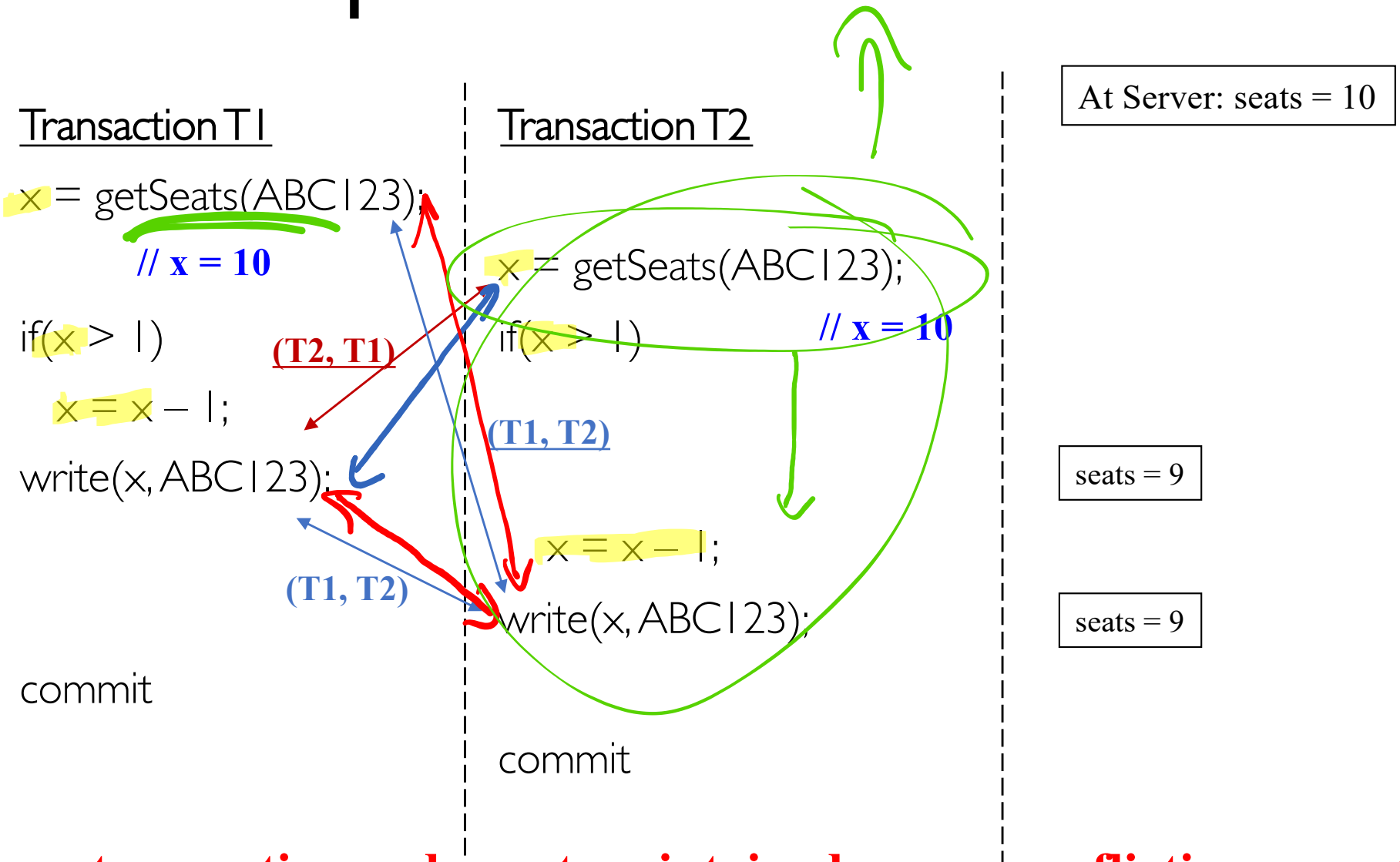
Transaction Reordering



Transaction Reordering



1. Lost Update Problem



Same transaction order not maintained across conflicting operations.
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);
	// Prints "Total: 20"
commit	
	commit

(T1, T2)

(T2, T1)

At Server:
ABC123 = 10
ABC789 = 15

Same transaction order not maintained across conflicting operations.
Not serially equivalent.

How do we handle such conflicts?

- 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 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!

xfer T4 → T3 → lock(T4.roster)
lock(T3.roster)
lock(student.schedule)

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)

Another approach: Read-Write Locks

- 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)

unlock(A)

allow promotion

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?

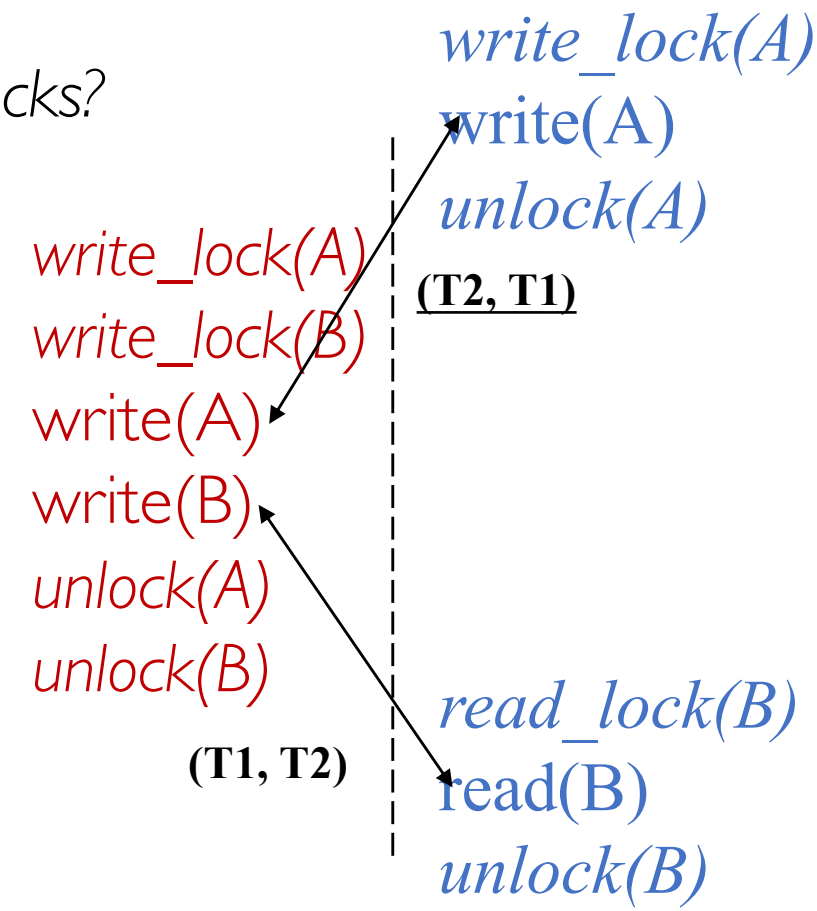
- We can have per-object locks in two modes to increase concurrency.
- Grab the object's lock in the appropriate mode when trying to access an object.
- *When to release locks?*

	write_lock(A)
write_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?

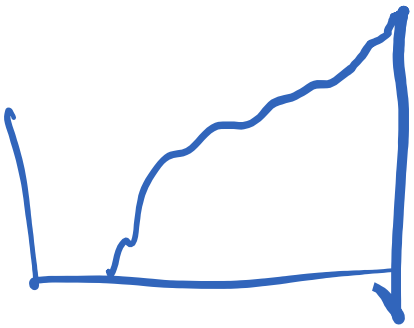
- We can have per-object locks in two modes to increase concurrency.
- Grab the object's lock in the appropriate mode when trying to access an object.
- *When to release locks?*



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)

release

Not allowed with
two-phase locking

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

acquire

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~~) ABC123		Deadlock!
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	

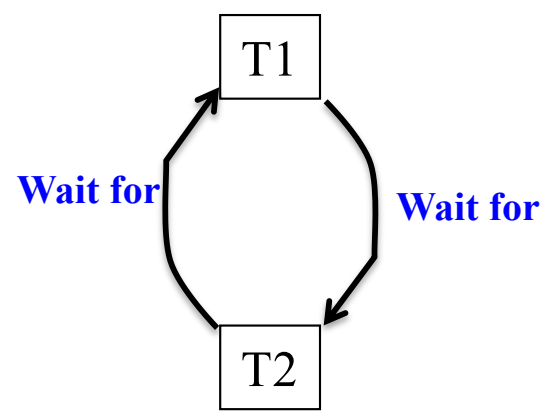
Downside of Locking

- Deadlock!

Deadlock Example

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);
unlock(x)	
commit	
	unlock(x)
	commit

Deadlock!

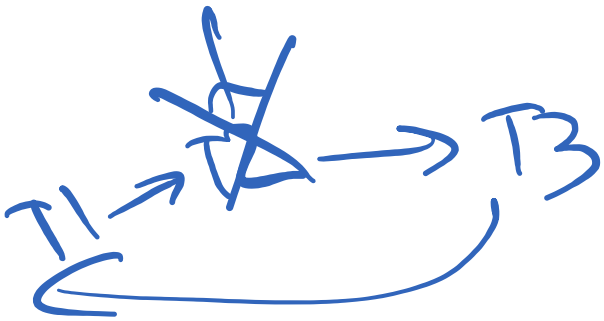


When do deadlocks occur?

- 3 <u>necessary</u> 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

Optimistic Concurrency Control

- Increases concurrency more than pessimistic concurrency control
- Used in Dropbox, Google apps, Wikipedia, key-value stores like Cassandra, Riak, and Amazon's Dynamo
- Preferable than pessimistic when conflicts are *expected to be* rare
    - But still need to ensure conflicts are caught!

First cut approach

- Most basic approach
  - Write and read objects at will
  - Check for serial equivalence at commit time
  - If abort, roll back updates made
  - An abort may result in other transactions that read dirty data, also being aborted
    - Any transactions that read from *those* transactions also now need to be aborted

    ☹ *Cascading aborts*

Timestamped ordering

- Assign each transaction an id
- Transaction id determines its position in serialization order.
- Ensure that for a transaction T, both are true:
  1. T's write to object O allowed only if transactions that have read or written O had lower ids than T.
  2. T's read to object O is allowed only if O was last written by a transaction with a lower id than T.
- Implemented by maintaining read and write timestamps for the object
- If rule violated, abort!
- Never results in a deadlock! Older transaction never waits on newer ones.

Timestamped ordering: per-object state

- Committed value.
- Transaction id (timestamp) that wrote the committed value.
- Read timestamps (RTS): List of transaction ids (timestamps) that have read the committed value.
- Tentative writes (TW): List of tentative writes sorted by the corresponding transaction ids (timestamps).
    - Timestamped *versions* of the object.

Timestamped ordering rules

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$ This requires that $T_c \geq$ the maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$ This requires that $T_c >$ write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$ This requires that $T_c >$ write timestamp of the committed object.

Timestamped ordering: write rule

Transaction T_c requests a write operation on object D

if (Tc ≥ max. read timestamp on D

&& Tc > write timestamp on committed version of D)

Perform a tentative write on D:

If T_c already has an entry in the TW list for D, update it.

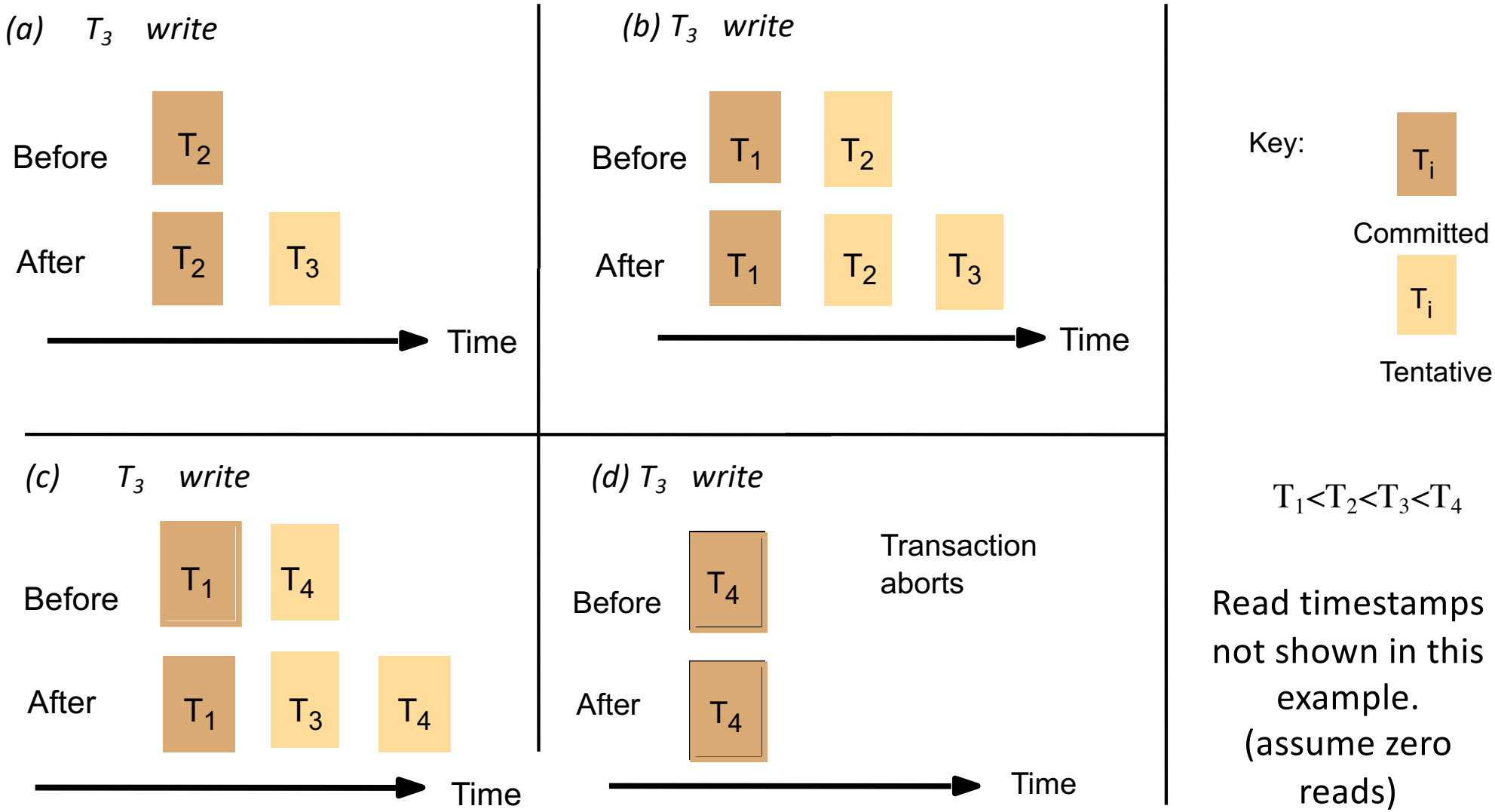
Else, add T_c and its write value to the TW list.

else

abort transaction T_c

//too late; a transaction with later timestamp has already read or written the object.

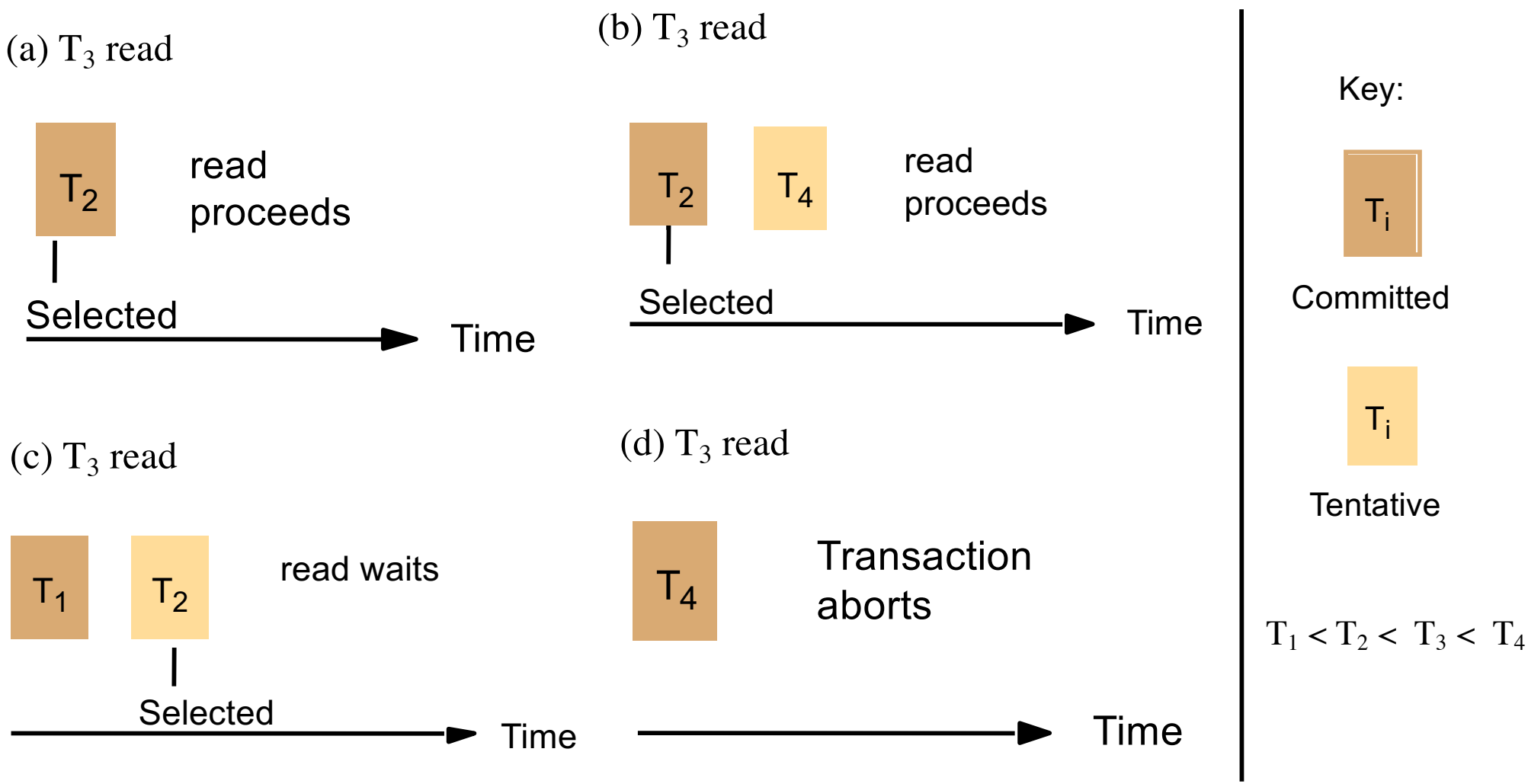
Timestamped ordering: write rule



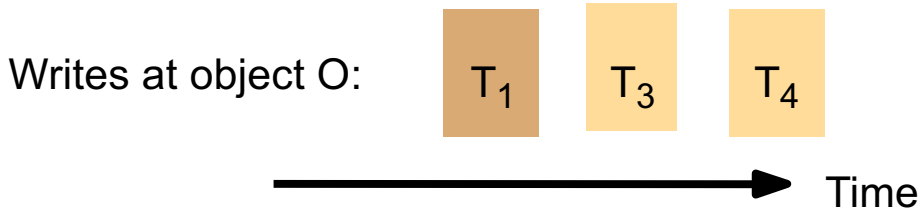
Timestamped ordering: read rule

```
Transaction Tc requests a read operation on object D
    if (Tc > write timestamp on committed version of D) {
        Ds = version of D with the maximum write timestamp that is ≤ Tc
        //search across the committed timestamp and the TW list for object D.
        if (Ds is committed)
            read Ds and add Tc to RTS list (if not already added)
        else
            if Ds was written by Tc, simply read Ds
            else
                wait until the transaction that wrote Ds is committed or aborted, and
                reapply the read rule.
                // if the transaction is committed, Tc will read its value after the wait.
                // if the transaction is aborted, Tc will read the value from an older
                transaction.
    } else
        abort transaction Tc
        //too late; a transaction with later timestamp has already written the object.
```

Timestamped ordering: read rule

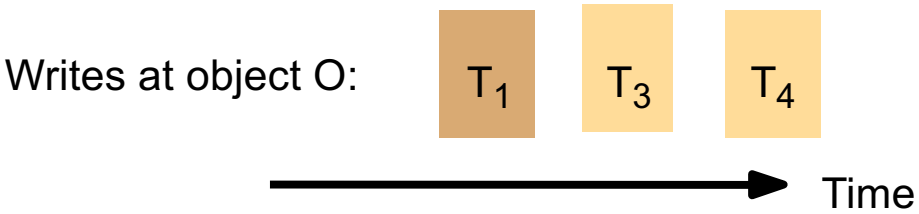


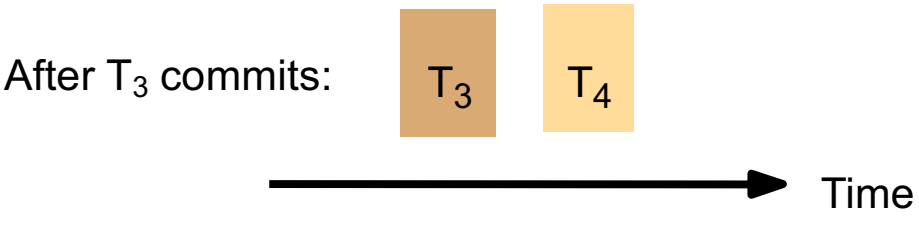
Timestamped ordering: committing



- Suppose T_4 is ready to commit.
- Must wait until T_3 commits or aborts.

- When a transaction is committed, the committed value of the object and associated timestamp are updated, and the corresponding write is removed from TW list.





Lost Update Example with Timestamped Ordering

Transaction T1	Transaction T2	ABC123: state committed value = 10 committed timestamp = 0 RTS: TW:
x = getSeats(ABC123);		
	x = getSeats(ABC123);	
if(x > 1)	if(x > 1)	
x = x – 1;		
write(x, ABC123);		
	x = x – 1;	
	write(x, ABC123);	
commit		
	commit	

Next Example with Timestamped Ordering

Transaction T1	Transaction T2	
		ABC123: state committed value = 10 committed timestamp = 0 RTS: TW:

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

	x = getSeats(ABC123); y = getSeats(ABC789);	
write(y+5, ABC789);		
		ABC789: state committed value = 5 committed timestamp = 0 RTS: TW:
	print("Total:" x+y);	
commit		
	commit	

Concurrency Control: Summary

- *How to prevent transactions from affecting one another?*
- Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).
- Target *serial equivalence.*
- Two approaches:
  - Pessimistic concurrency control: locking based.
    - read-write locks with two-phase locking and deadlock detection.
  - Optimistic concurrency control: abort if too late.
    - timestamped ordering.

Next Class

- Distributed Transactions.

Next Class

- Pessimistic Concurrency Control
  - Two-phase Locking
  - Deadlocks
- Optimistic Concurrency Control