COSC 460 Lecture 16: Transactions 3: Two Phase Locking

Professor Michael Hay
Fall 2018

Transactions

- **Transaction** a sequence of SQL operations treated as a unit.
- Transactions appear to run in isolation
- Transaction either runs to completion or not at all
- ACID Properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability

```
BEGIN TRANSACTION;
insert into archive
(select *
 from apply
 where decision = 'n');
delete from apply
where decision = 'n';
COMMIT;
```

Isolation

- Goal: develop ways to **I**solation. We will worry about A,C, and D later.
- Plan:

  1. See what isolation looks like (serializable schedules, confict-serializable schedules, …)

  2. See how to ensure isolation (locking protocols)

Serializable schedules

- Schedule S’ is **serializable** if it is equivalent to some serial schedule S.

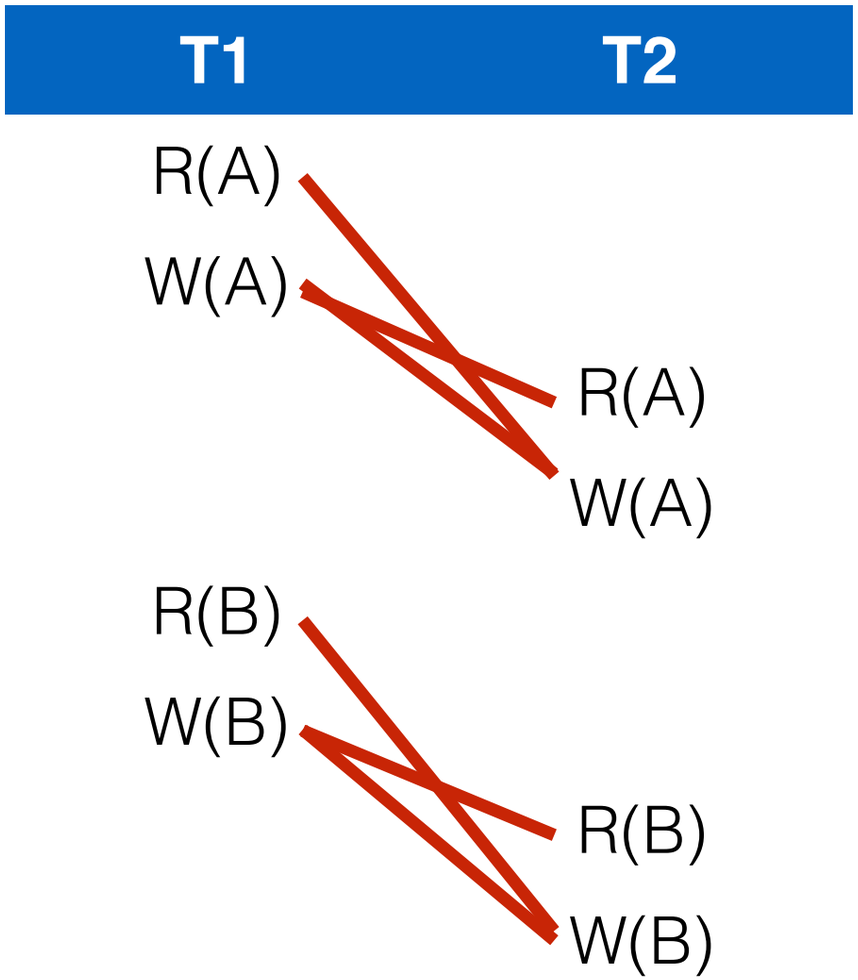
Conflict serializable

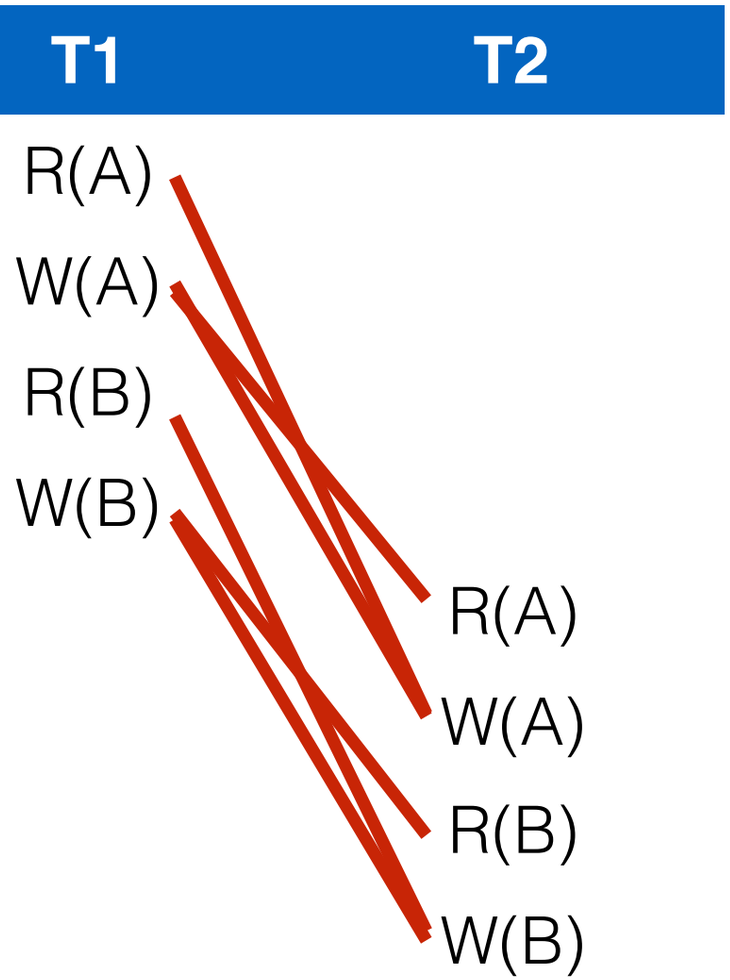
- A schedule S' is **conflict serializable** if it is *conflict equivalent* to SOME serial schedule S.

- Conflict equivalent: every pair of conflicting statements is ordered in the same way

Conflict equivalent

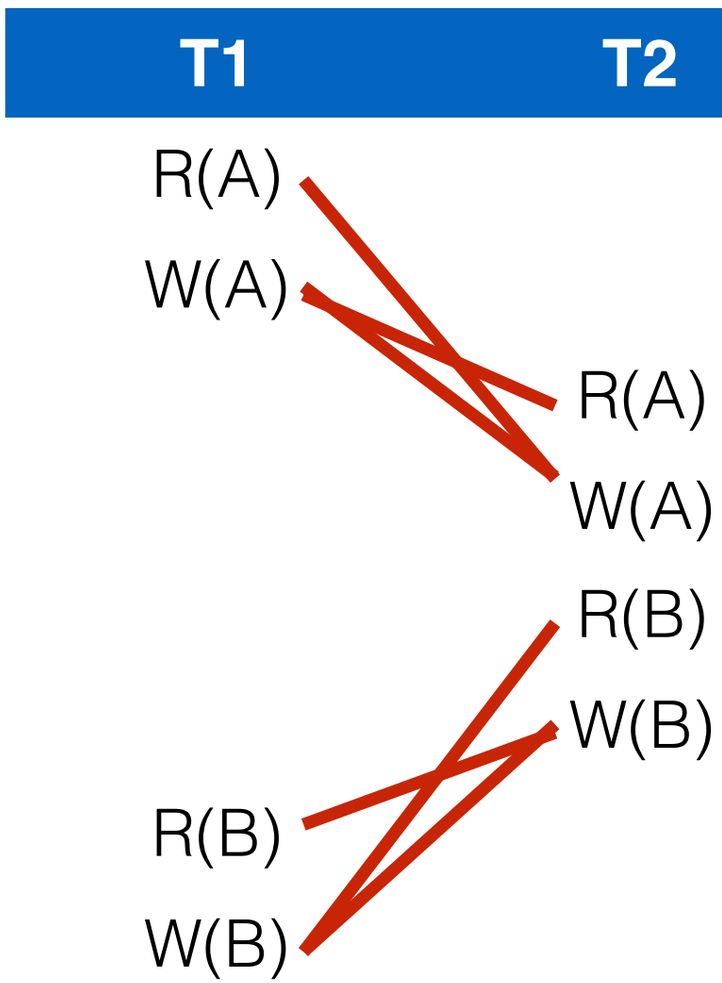
- Schedule S is **conflict equivalent** to S' if every pair of conflicting statements is ordered in the same way





Conflict equivalent

- Schedule S is **conflict equivalent** to S’ if every pair of conflicting statements is ordered in the same way



Not conflict equivalent to any serial schedule

Concurrency control

- How ensure serializability?
- Two high-level strategies
  - *Optimistic*: detect “bad” schedules and abort offending transactions
  - *Pessimistic*: prevent “bad” schedules through locking protocol

Lock Management

In ColgateDB:
"Item" = page

- Each "item" has lock.
- Responsibilities of ***transaction:***
  - *Request* lock before R or W
  - *Wait* until request is granted
  - *Release* lock when done
- Responsibilities of ***lock manager:***
  - Grant/deny requests
  - Maintain *status* of locks and lock requests (details later)

Example

Instructions: ~1 minute to think/ answer on your own; then discuss with neighbors; then I will call on one of you

- T1 transfers between accounts
- T2 displays total account balance
- Is this schedule *serializable*?

Locking alone is not enough! Need ***locking protocol***.

T1	T2
L(A)	
	L(A)
A=A+100	|
U(A)	|
	L(B)
	print(A+B)
	U(A), U(B)
L(B)	
B = B - 100	
U(B)	

Two phase locking

- **2PL**: a transaction cannot acquire additional locks once it has released any lock
  - Growing phase (acquiring locks)
  - Shrinking phase (releasing locks)
- **Lockpoint**: time at which transaction T acquired its last lock

Exercise

Instructions: ~1 minute to think/ answer on your own; then discuss with neighbors; then I will call on one of you

- Is this schedule feasible under the *2PL protocol*?
- If not, why not?
- If so, is it a *serializable schedule*?

T1	T2
L(A)	
	L(A)
A=A+100	|
U(A)	|
	temp = A
	U(A)
	L(B)
L(B)	
|	temp += B
|	U(B)
B = B - 100	
U(B)	
	print(temp)

Exercise

Instructions: ~1 minute to think/ answer on your own; then discuss with neighbors; then I will call on one of you

- T1 and T2 are the same as in the previous example except for the order of the operations in **red**.
- Is this schedule feasible under the *2PL protocol*?
- If not, why not?
- If so, is it a *serializable schedule*?

T1	T2
L(A)	
	L(A)
A=A+100	|
L(B)	|
U(A)	|
	temp = A
	L(B)
B = B - 100	**I**
	I
U(B)	**U(A)**
	temp += B
	U(B)
	print(temp)

2PL → Conflict serializability

- Any 2PL schedule is conflict equivalent to the schedule where transactions are ordered by lockpoint

- Proof sketch: proof by contraction. Suppose schedule is 2PL but *not* conflict serializable.

Increasing concurrency

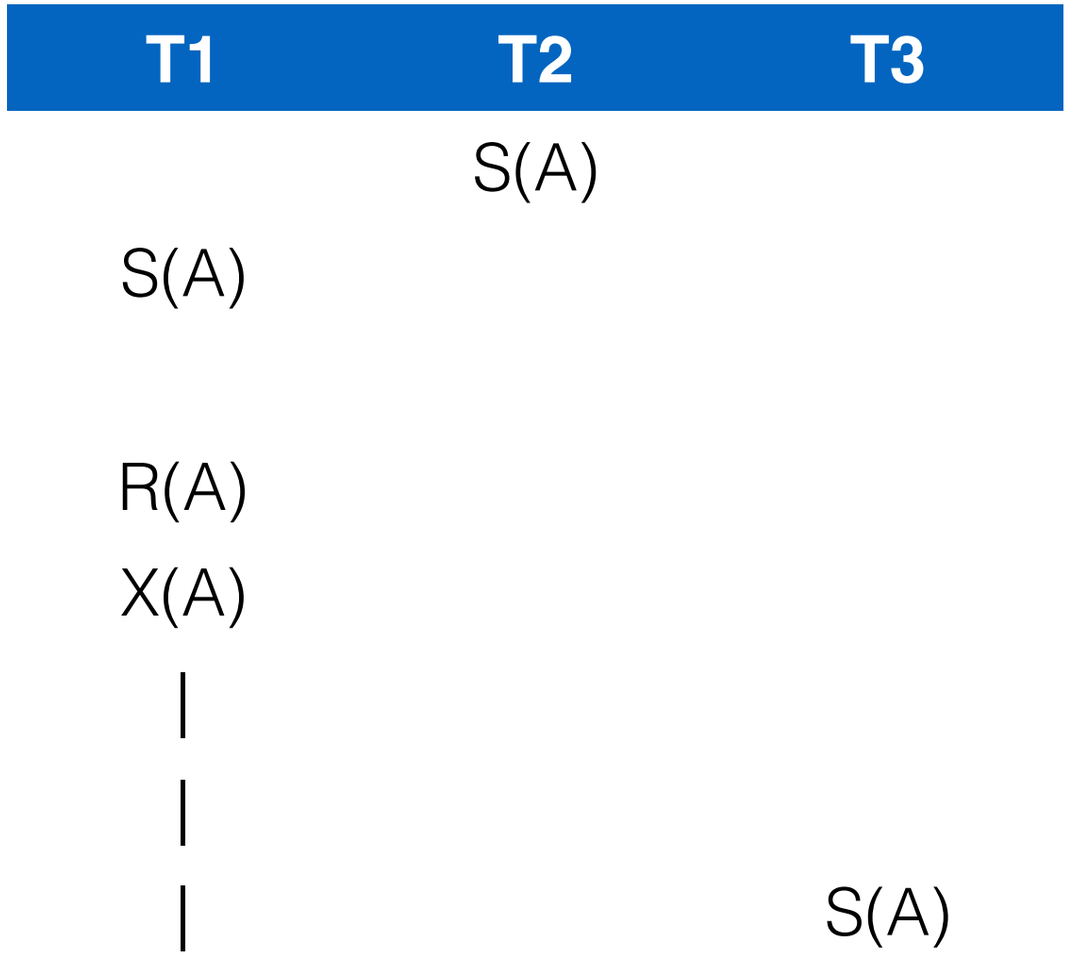
- Observation: reads do not conflict with each other
- Associate “permission” with each lock request:
  - R only → ***shared lock***
  - R&W→ ***exclusive lock***
- Upgrades/downgrades
  - Upgrade: have shared, get exclusive
  - Downgrade: have exclusive, allow shared
  - 2PL: upgrade *only* during growing; downgrade only during shrinking

Lock Requests and Priority

- T3 is requesting a shared lock while T1 is waiting on an upgrade. Should T3 be granted the lock?

Concern with granting lock to T3: T1 might starve. Make T3 wait.

Example 1

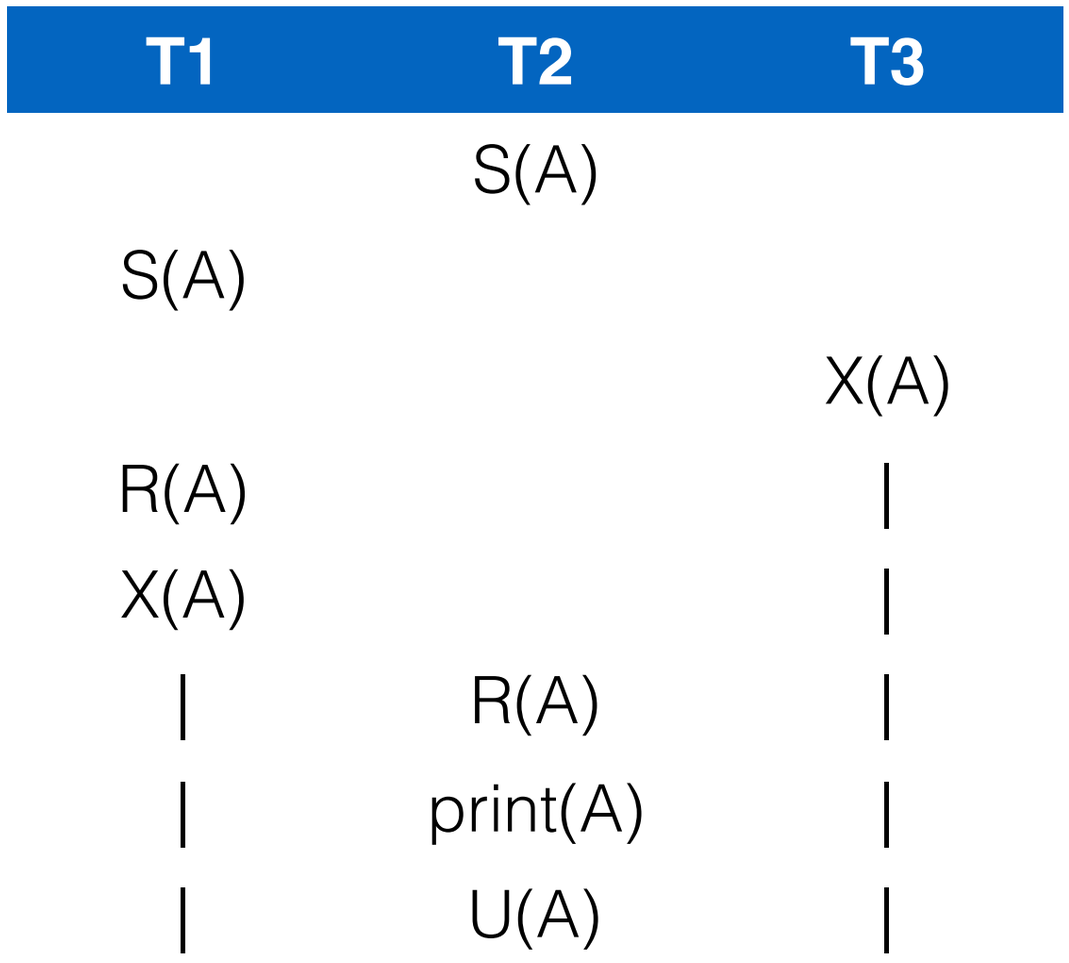


Lock Requests and Priority

- T1 is requesting an upgrade lock while T3 is waiting on an exclusive. Should T1 be granted the lock?

Concern with making T1 wait: deadlock.

Example 2



Lock management

- Lock Table: maps item to LockTableEntry
- LockTableEntry
  - Current lock type: shared/ exclusive/none
  - Current lock holders
  - Requests: list of (transaction, permissions) pairs

ColgateDB: transactions manage their own requests via *shared* lock table. We do not have separate thread “managing” lock table.

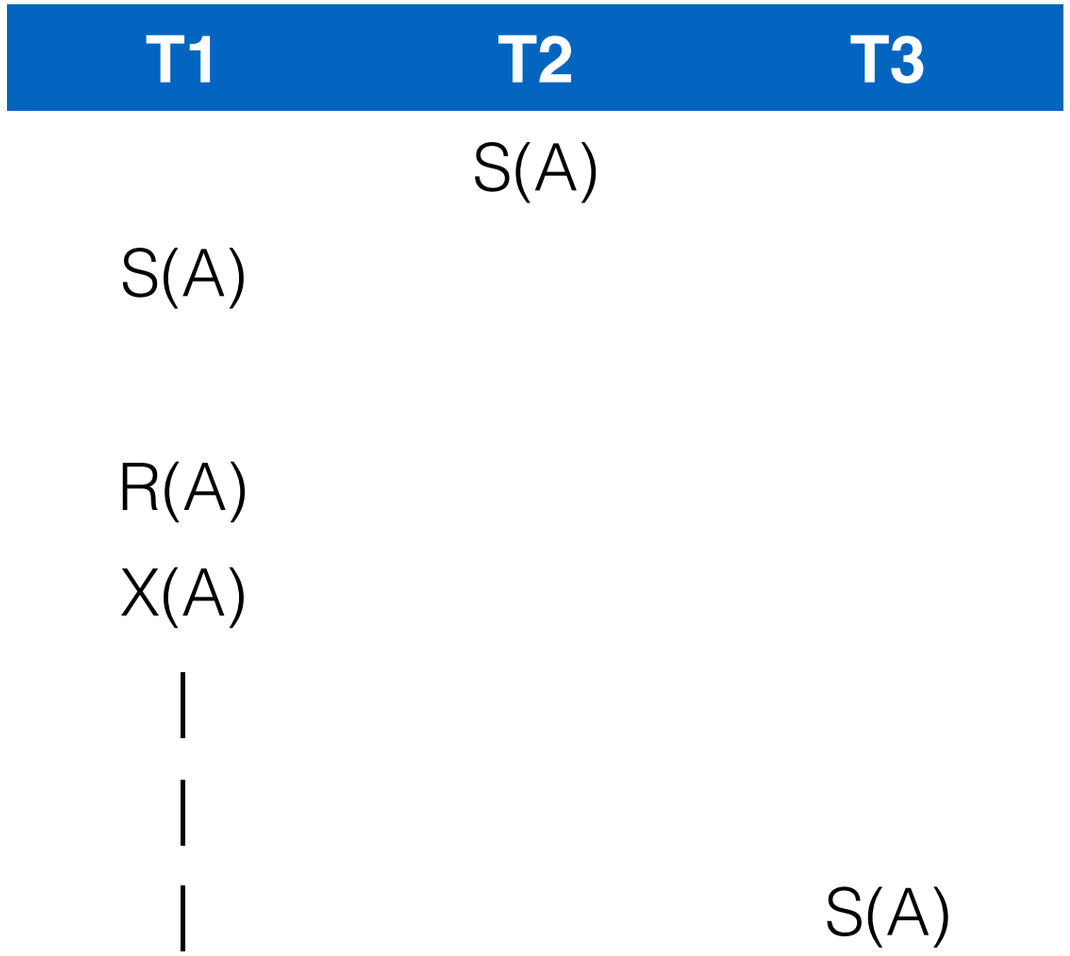
- Handling lock request:
  - If request is upgrade, put at front of queue; else, put at end
  - Only transaction at front of queue can be granted lock!
  - Whether to grant lock depends on current lock type/holders, and permissions being requested
  - If granted: update entry, check request queue
- Handling lock release: update entry, check request queue

Lock Requests and Priority

- T3 is requesting a shared lock while T1 is waiting on an upgrade. Should T3 be granted the lock?

No. T1 is at front of queue. T3 must wait.

Example 1
Revisited

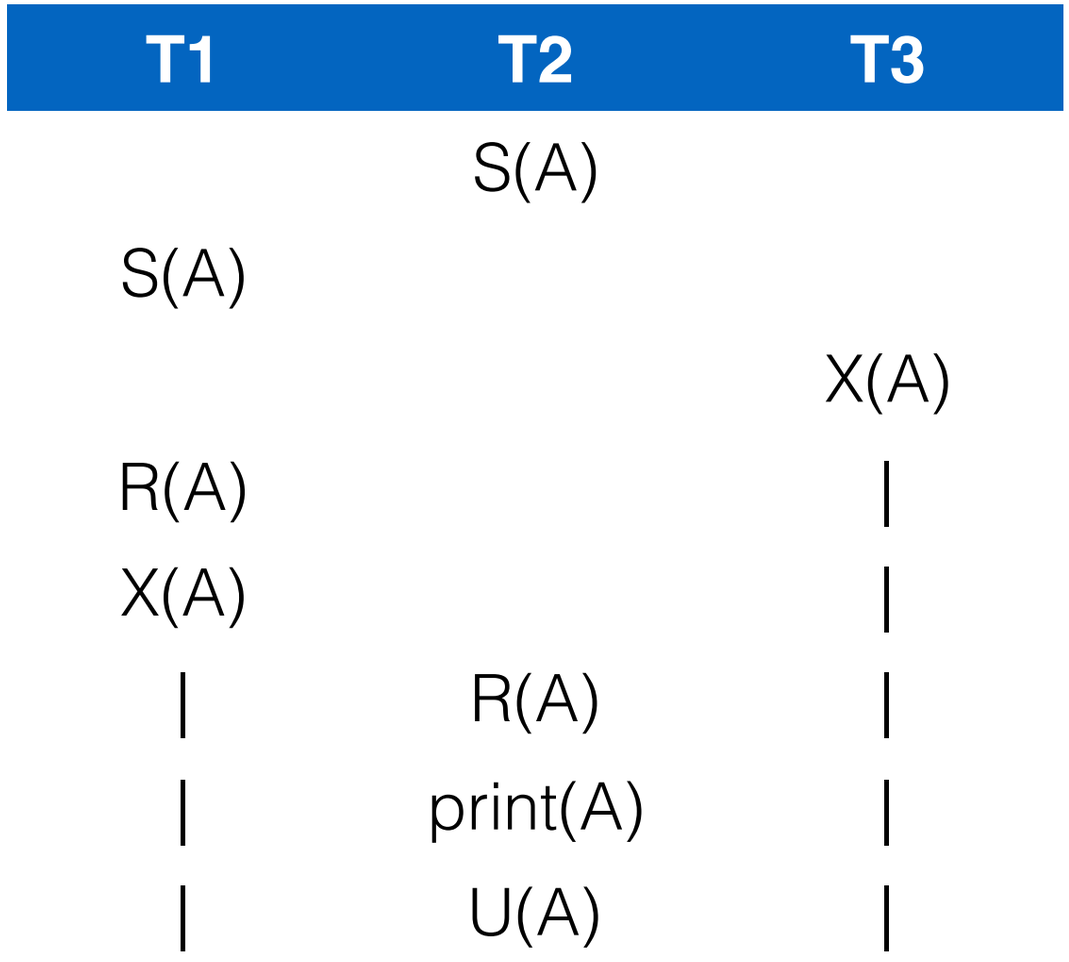


Lock Requests and Priority

- T1 is requesting an upgrade lock while T3 is waiting on an exclusive. Should T1 be granted the lock?

Yes! T1 upgrade request jumps to front of queue. T1 gets lock when T2 releases. T3 waits for T1.

Example 2
Revisited



Deadlock

- T1 transfers money from B to A.
- T2 transfers money from A to B.

T1	T2
L(A)	
	L(B)
A=A+100	
	B=B+50
	L(A)
L(B)	A=A-50
B=B-100	U(A),U(B)
U(A), U(B)	

Deadlock! (Grayed out events never happen)