Discussion 11

Transactions & Two-Phase Locking

EECS 484

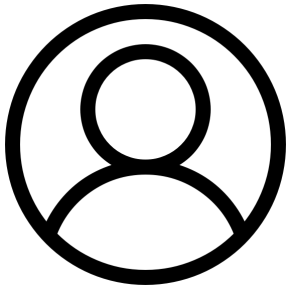
Logistics

- **Project 4** due **Today** at 11:45 PM ET
- **HW 6** due **Dec 6th** at 11:45 PM ET
- Final exam: **Dec 12th**, 7:00-9:00 PM ET
  - Alternate final form due today
  - Practice Final released on Canvas
  - More details for the final are coming soon

Transactions

Transactions

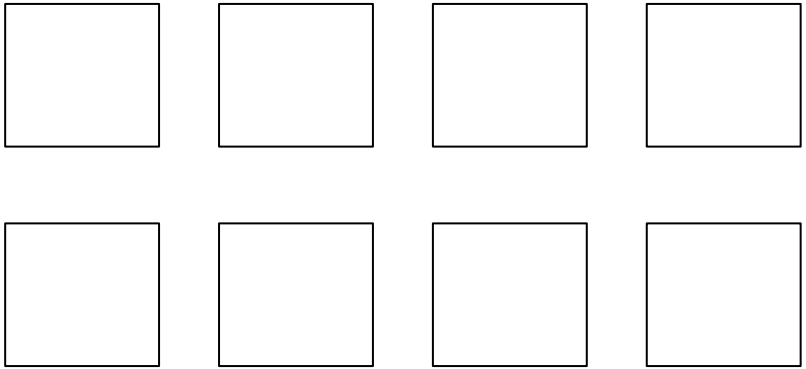
- A transaction is an atomic unit of work
  - Either everything in a transaction happens or it doesn’t
  - Multiple actions in a transaction
- Transactions can interleave actions
  - Can do action 2 of transaction 1 at the same time as action 4 of transaction 2
  - Need to ensure that we avoid any inconsistencies
    - Same result as doing transactions serially (consecutively)

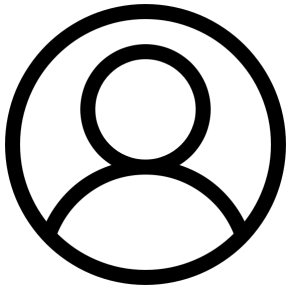


Good User’s transaction:

G1: Give the first 4 students an A+

G2: Give the last 4 students an A+

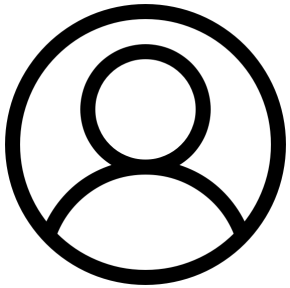




Evil User’s transaction:

E1: Give the first 4 students an F

E2: Give the last 4 students an F

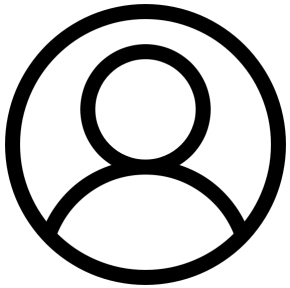


Good User's transaction:

G1: Give the first 4 students an A+

G2: Give the last 4 students an A+

A+	A+	A+	A+

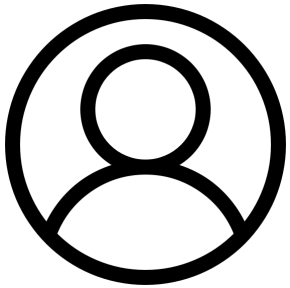


Evil User's transaction:

E1: Give the first 4 students an F

E2: Give the last 4 students an F

Interleaving: G1

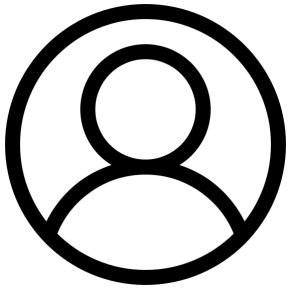


Good User's transaction:

G1: Give the first 4 students an A+

G2: Give the last 4 students an A+

F	F	F	F

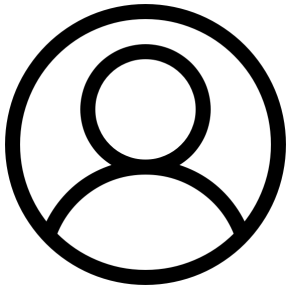


Evil User's transaction:

E1: Give the first 4 students an F

E2: Give the last 4 students an F

Interleaving: G1, E1

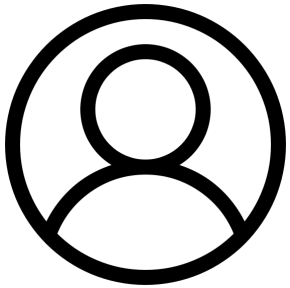


Good User's transaction:

G1: Give the first 4 students an A+

G2: Give the last 4 students an A+

F	F	F	F
F	F	F	F

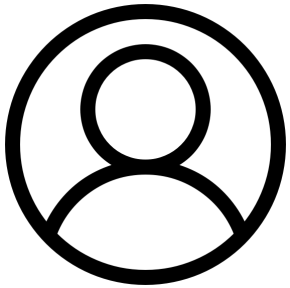


Evil User's transaction:

E1: Give the first 4 students an F

E2: Give the last 4 students an F

Interleaving: G1, E1, E2

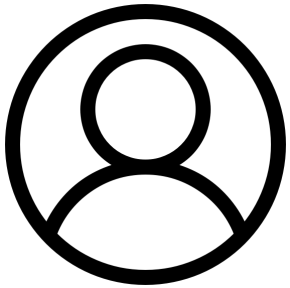


Good User's transaction:

G1: Give the first 4 students an A+

G2: Give the last 4 students an A+

F	F	F	F
A+	A+	A+	A+



Evil User's transaction:

E1: Give the first 4 students an F

E2: Give the last 4 students an F

Interleaving: G1, E1, E2, G2

ACID

- Properties we need to enforce to ensure the database is valid
- Atomicity
  - All actions in the transaction happen or none of them happen
- Consistency
  - If we start with a consistent database and perform a consistent transaction we have a consistent database at the end
- Isolation
  - Each transaction appears to occur serially
- Durability
  - If a transaction occurs, its effects persist

Scheduling

- Schedules are a list of ordered actions across transactions
  - Can interleave actions from various transactions
  - Serial schedule = no interleaving of transactions
  - Serializable schedule = result matches what some serial schedule would have produced (all reads and final states)
- Different serial schedules for the same transactions can have different results
  - All are assumed to be okay

T1	T2
Read(A)	
	Read(B)
	Write(B)
Write(A)	
	Commit
Commit	

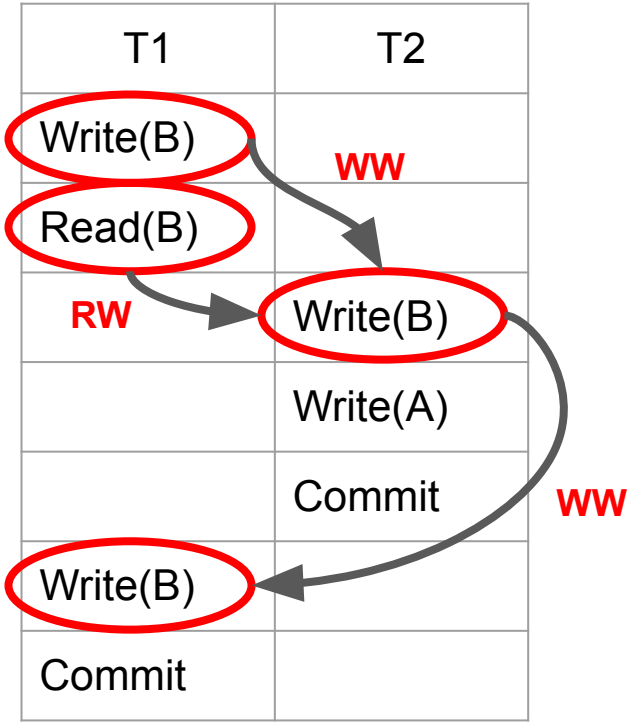
Conflicts

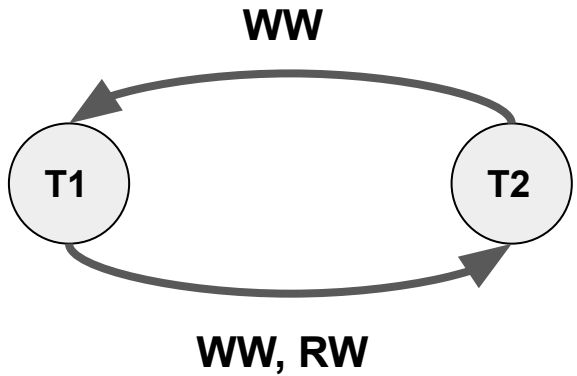
- Conflicts happen when we try to access a certain resource multiple times in non-compatible ways
  - All happen with writes - updating data
  - **Write-Read (WR)** - could indicate a dirty read
  - **Read-Write (RW)** - could indicate an unrepeatable read
  - **Write-Write (WW)** - could indicate overwriting of uncommitted data
- Conflicts **do not** necessarily mean anomalies
  - **WR, RW, WW conflicts only indicate the order (precedence) that these operations need to be executed**
  - Only if we have an inconsistent database after the schedule do we have an anomaly

Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic
- Precedence graph is a series of connected nodes for committed transactions
  - A node for each committed transaction
  - Arc from T_x to T_y if some action in T_x precedes and conflicts with some action in T_y
- All conflict serializable schedules are serializable
  - Serializable: all reads and the final state is what some complete serial schedule of committed transactions would have produced.
  - Not the other way around!
  - Testing conflict serializability is much easier than testing serializability

Conflicts and Conflict Serializability Example

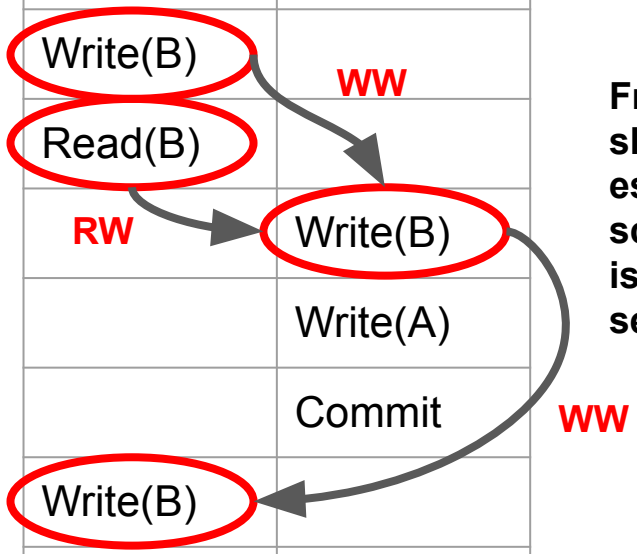




A cycle in the precedence graph indicates this schedule is not conflict serializable

Not Conflict Serializable BUT Serializable

T1	T2
Write(B)	
Read(B)	
	Write(B)
	Write(A)
	Commit
Write(B)	
Commit	



From the previous slide, we established that the schedule on the left is not conflict serializable

T1	T2
	Write(B)
	Write(A)
	Commit
Write(B)	
Read(B)	
Write(B)	
Commit	

Not conflict serializable

but serializable

Not Conflict Serializable BUT Serializable

T1	T2
Write(B)	
Read(B)	
	Write(B)
	Write(A)
	Commit
Write(B)	
Commit	

Not conflict serializable

All Reads() still read the same value (i.e. the B value written by T1)

The same transactions still make the same final Write() to A and B

T1	T2
	Write(B)
	Write(A)
	Commit
Write(B)	
Read(B)	
Write(B)	
Commit	

but serializable

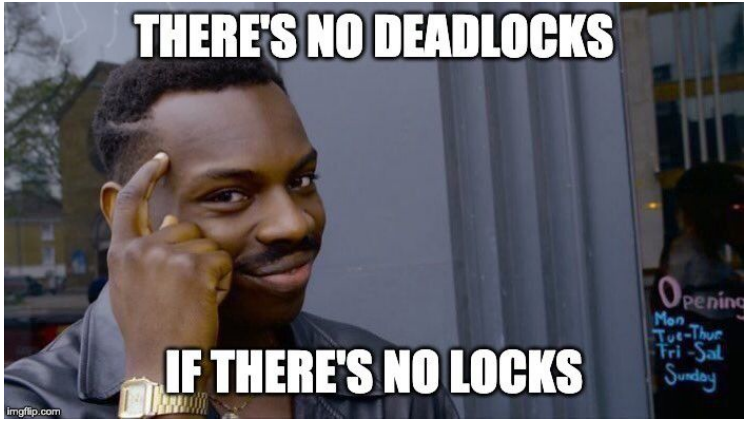
Two Phase Locking

Two Phase Locking (2PL)

- Locking allows system to ensure one transaction occurs before another
  - Use locks, where a transaction must get a lock before proceeding
    - Only acquire the lock you need at the time of the request
    - If not available, wait
  - Two types of locks
    - Shared (read) locks: multiple transactions can hold same lock at the same time
      - Can only read resource
    - Exclusive (write) locks: one transaction can hold lock at a time
      - Can read or write resource
      - Prevents other transactions from reading resource while we are writing

Two Phase Locking

- 2PL
  - If a transaction releases any lock, it can no longer acquire any new locks
  - Guarantees conflict-serializability (and thus serializability)
  - … but may sometimes allow/cause cascading aborts & deadlocks
- Strong Strict 2PL (SS2PL)
  - 2PL + Hold all locks acquired until the end of the transaction
  - Guarantees conflict serializability (and thus serializability)



2PL Shortcomings: Cascading Aborts

- Aborting transactions may mean other transactions have to be aborted

	T1	T2
X-Lock(A)	Write(A)	
X-Lock(B)	Write(B)	
		Read(A)
	Read(B)	
	Write(B)	
	Abort	
		Abort

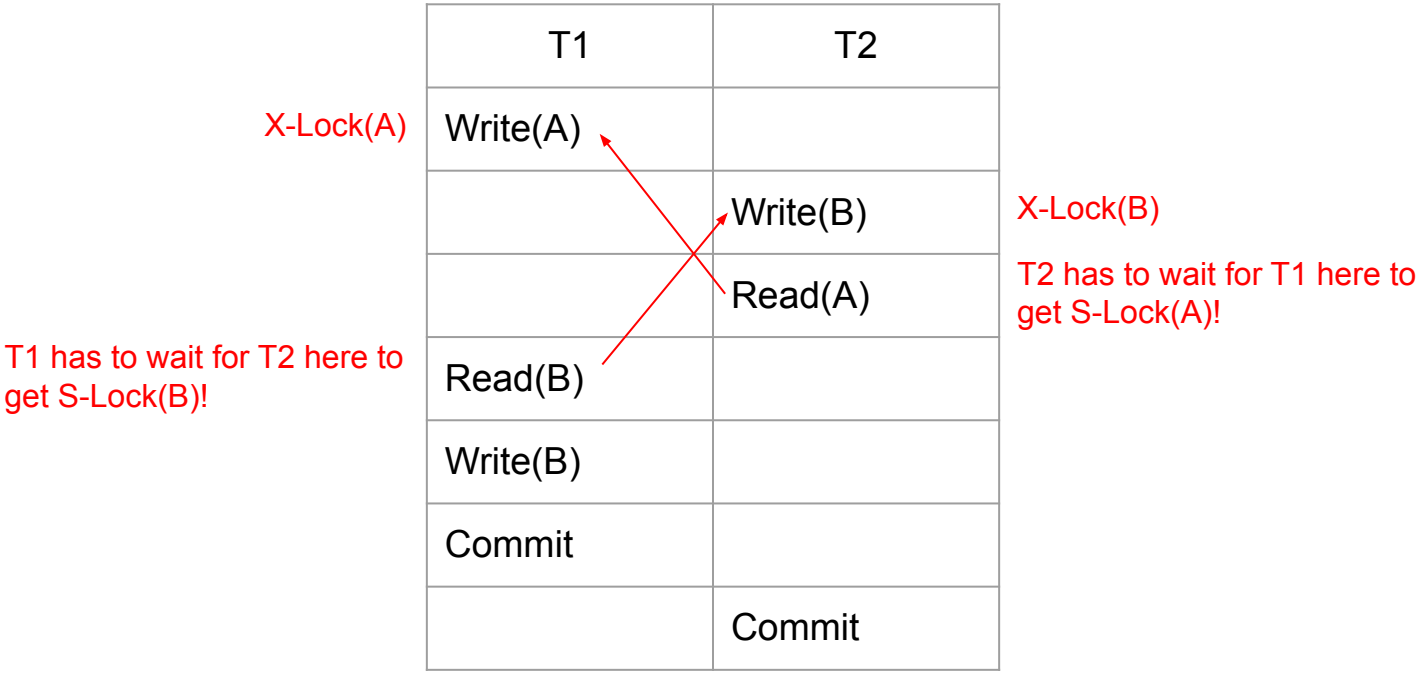
T1 released X-Lock(A) &T2 acquires S-Lock(A)

T1 aborts, all modifications made by T1 are rolled back

Since T2 read A that is modified by aborted T1, T2 also needs to abort

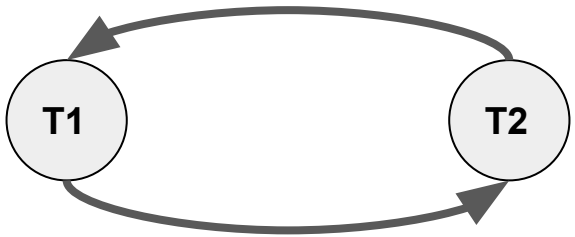
2PL Shortcomings: Deadlocks

- 2 transactions wait on each other to acquire locks



Deadlock Detection

- Similar to determining conflict serializability!
- Draw a “waits-for” graph
  - Arc from T_x to T_y if T_x has to wait for some action in T_y to complete to acquire a lock
- Cycle = deadlock!
  - Can abort a transaction in the cycle to fix



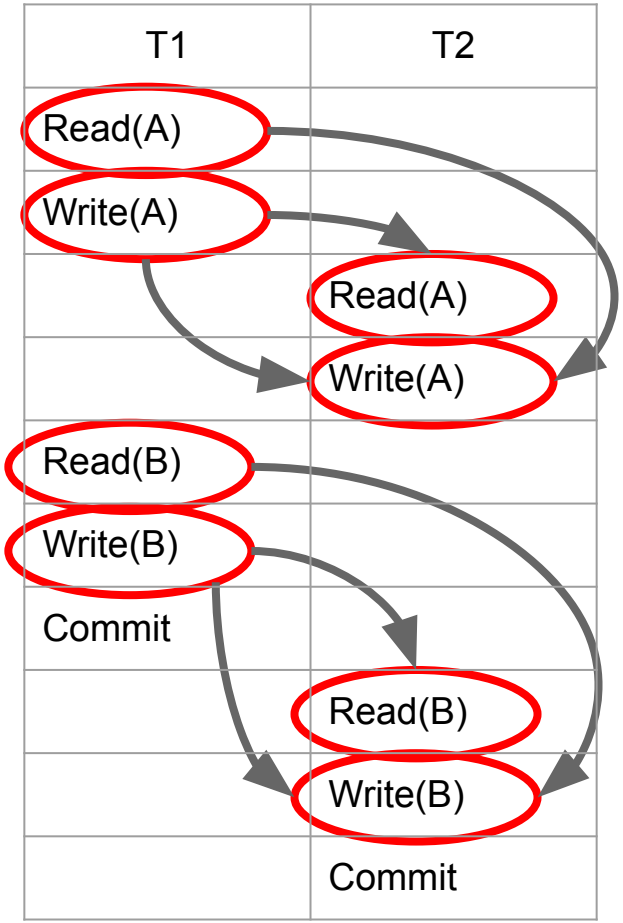
Example Problems

- List the conflicts between T1 and T2
- Is the schedule conflict serializable?
- Is the schedule serializable?
- Is the schedule 2PL?
- Is the schedule SS2PL?

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
Commit	
	Read(B)
	Write(B)
	Commit

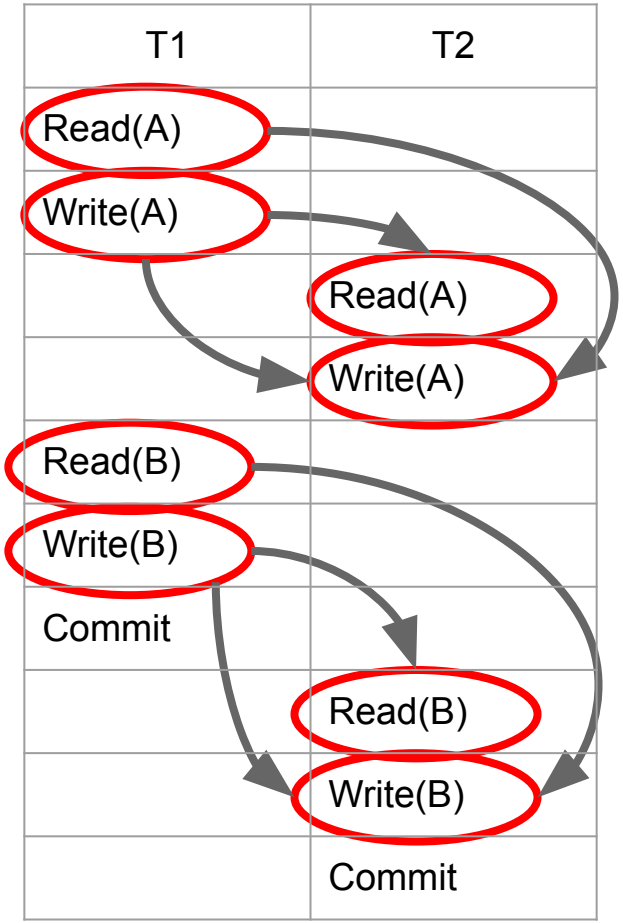
Example Problems

- **List the conflicts between T1 and T2**
  RW conflict from T1 to T2 on A
  WR conflict from T1 to T2 on A
  WW conflict from T1 to T2 on A
  RW conflict from T1 to T2 on B
  WR conflict from T1 to T2 on B
  WW conflict from T1 to T2 on B



Example Problems

- **Is the schedule conflict serializable?**
- Yes
- Draw the precedence graph
  - Or examine the dependencies
  - No dependencies from T2 to T1
    - For sure acyclic then



Example Problems

- **Is the schedule serializable?**
- Yes. We get for free since conflict serializable :)

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
Commit	
	Read(B)
	Write(B)
	Commit

Example Problems

- **Is the schedule 2PL?**
  - Assume we acquire locks at the necessary step
- T1 needs shared lock on A in step 1 - okay
- T1 acquires exclusive lock on A in step 2 - okay
- T2 needs shared lock on A - :(
  - T1 holds exclusive lock on A
  - If T1 lets go, then T1 cannot acquire another lock
  - T1 needs shared and exclusive lock on B later
- Not 2PL
  - If we acquire all necessary locks at the beginning and when available, then this can be 2PL

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
Commit	
	Read(B)
	Write(B)
	Commit

Example Problems

- **Is the schedule SS2PL?**
  - No since it's not 2PL

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
Commit	
	Read(B)
	Write(B)
	Commit

Extra Problems

- List the conflicts between T1 and T2
- Is the schedule conflict serializable?
- Is the schedule serializable?
- Is the schedule 2PL?
- Is the schedule SS2PL?

T1	T2
Read(A)	
	Read(B)
	Write(B)
	Read(C)
	Write(C)
	Commit
Read(B)	
Write(B)	
Write(A)	
Commit	

Extra Problems

- List the conflicts between T1 and T2
  - WW, RW, WR From T2->T1 on B
- Is the schedule conflict serializable?
  - Yes, no dependencies from T1->T2
- Is the schedule serializable?
  - Yes, since conflict serializable

T1	T2
Read(A)	
	Read(B)
	Write(B)
	Read(C)
	Write(C)
	Commit
Read(B)	
Write(B)	
Write(A)	
Commit	

Extra Problems Cont.

- Is the schedule 2PL? - yes
  - T1 acquires shared lock on A in step 1
  - T2 acquires shared lock on B in step 2
  - T2 acquires exclusive lock on B in step 3
  - T2 acquires shared lock on C in step 4
  - T2 acquires exclusive lock on C in step 5
  - T2 commits and lets go of all locks in step 6
  - T1 acquires shared lock on B in step 7
  - T1 acquires exclusive lock on B in step 8
  - T1 acquires exclusive lock on A in step 9
  - T1 commits and lets go of all locks in step 10
- Is the schedule SS2PL?
  - Yes, T1 and T2 only release locks on commit

T1	T2
Read(A)	
	Read(B)
	Write(B)
	Read(C)
	Write(C)
	Commit
Read(B)	
Write(B)	
Write(A)	
Commit	

Get started on HW6!

We’re here if you need any help!!

- Office Hours: Schedule is here, both virtual and in person offered
- Piazza
- Next week’s discussion!!!