# Discussion 11

Transactions

EECS 484

# Logistics

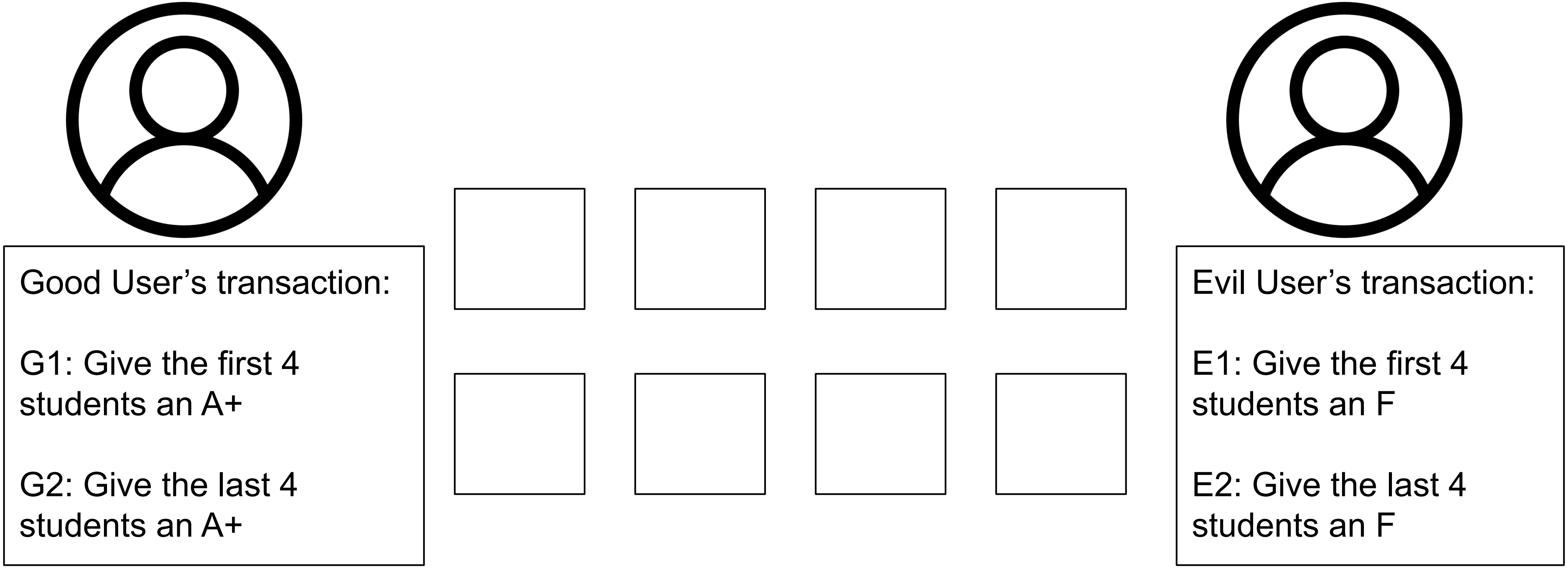
- HW 6 Due Dec. 1st at 11:55 pm
- Project 4 Due Dec. 8th at 11:55 pm
- Final Exam on Dec. 13th, Tuesday 7-9PM
    - Only multiple choice questions
    - Covers second half of course (Indexing and beyond)
    - Practice exams posted on Canvas
    - Final Review Session - Dec. 7th, lecture time
    - Exam time and location for SSD students sent via email
        - If you applied for SSD accommodation and didn’t receive any email about it, please email us at eecs484staff@umich.edu
    - More logistics posted later on Piazza
- Today
    - Transactions

# Transaction Management

# Transactions

⇗ all / none

- 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

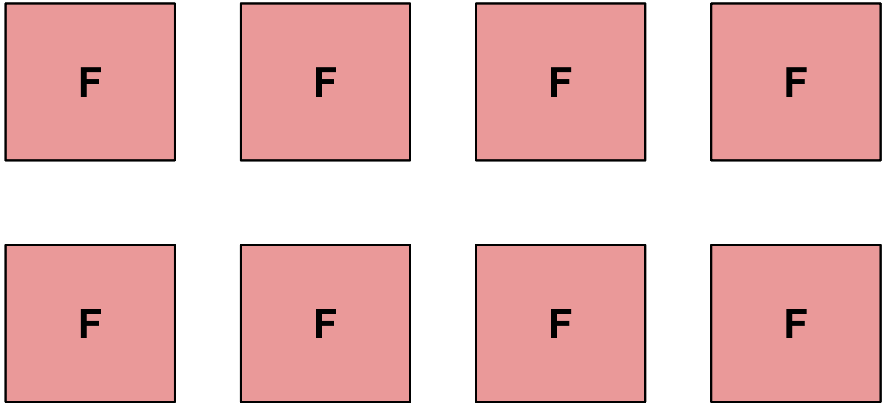
E2: Give the last 4 students an F

Interleaving: G1, E1

Good User’s transaction:

G1: Give the first 4 students an A+

G2: Give the last 4 students an A+



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

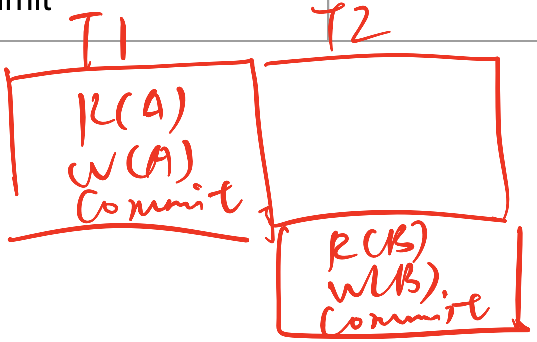
- Properties we need to enforce to ensure the database is valid
- Atomicity
  - All actions in the transaction happen or none of them happen
- Consistency
  - If we start with a consistent database and perform a consistent transaction we have a consistent database at the end
- Isolation
  - Each transaction 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) result same + intermidiate same
- 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 | |

serial scedule



# Conflicts

Don't care about commit/abort/running

- 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 data
  - **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 ≠ anomaly

(don't consider abort for precedence graph) But conflict still occurs even aborted

Conflict serializable <=> no cycle

# 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

Serializablity

Conflict Serializablity

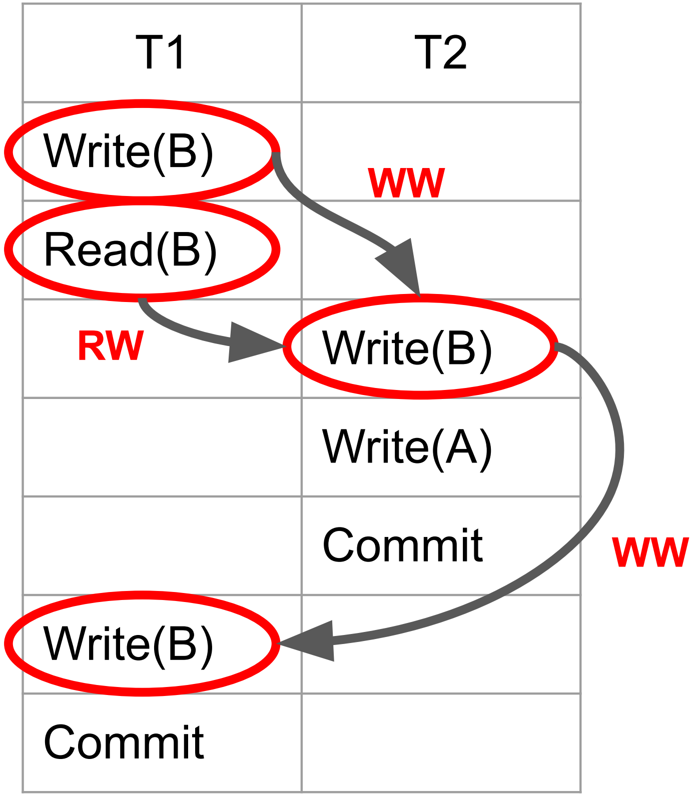
serializability => Conflict Serializability

(graph)

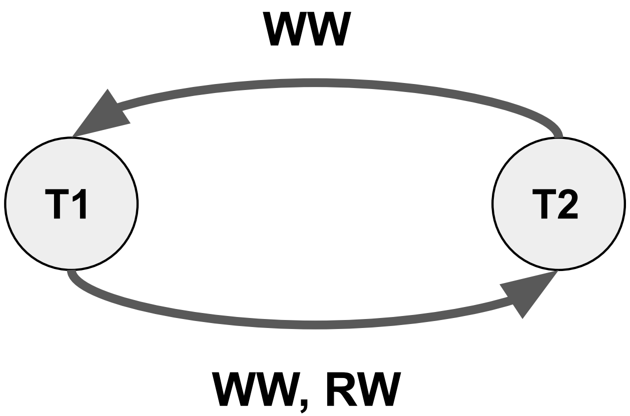
(try every possibility)

☆ Conflict serializability → serializability

# Conflicts and Conflict Serializability Example



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



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

# Not Conflict Serializable BUT Serializable

same intermediate read

| T1 | T2 |
|---|---|
| ① Write(B) | |
| Read(B) | |
| | Write(B) |
| | Write(A) ② → final write to A |
| | Commit |
| ③ Write(B) → final write to B | |
| Commit | |

WW

RW

WW

Not conflict serializable

Same

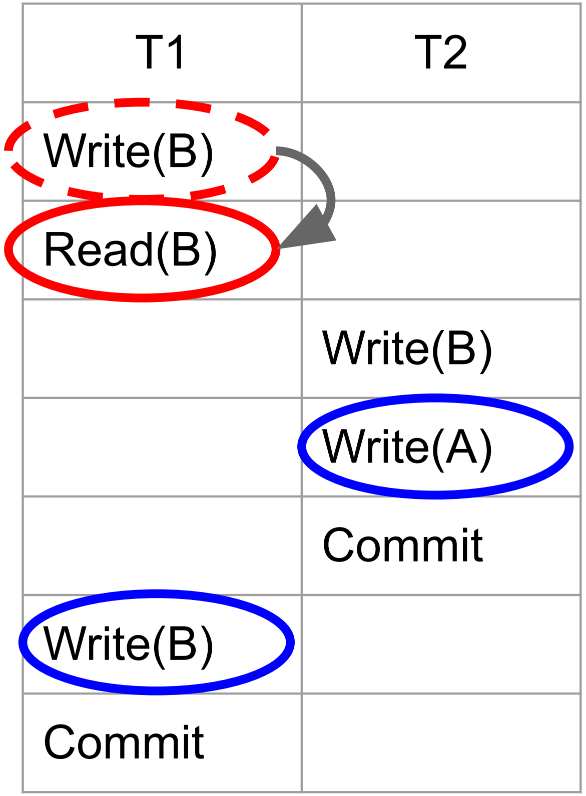
**From the previous slide, we established that the schedule on the left is not conflict serializable**

| T1 | T2 |
|---|---|
| | Write(B) |
| ② | Write(A) final write to A |
| | Commit |
| Write(B) ① | |
| Read(B) | intermediate read result written by T1 itself |
| ③ Write(B) | |
| Commit | |

final write to B

but serializable

# Not Conflict Serializable BUT Serializable

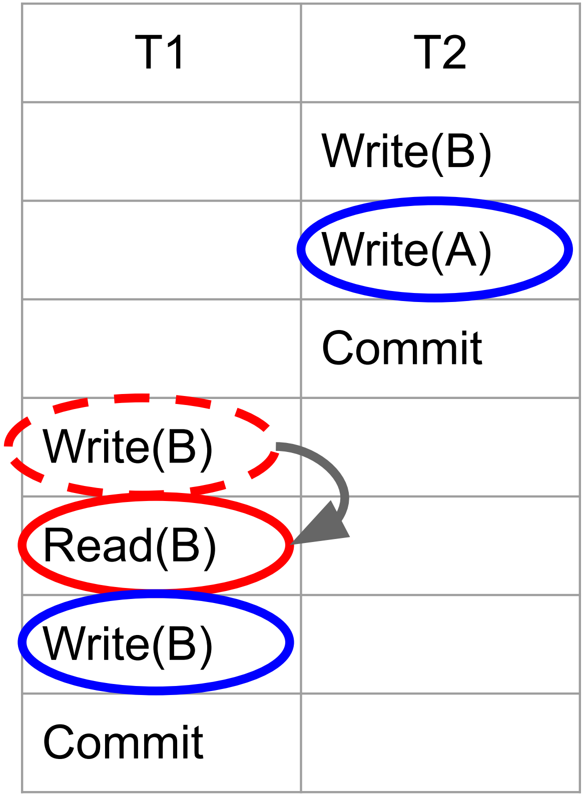


| T1 | T2 |
|---|---|
| Write(B) | |
| Read(B) | |
| | Write(B) |
| | Write(A) |
| | Commit |
| Write(B) | |
| Commit | |

Not conflict serializable

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

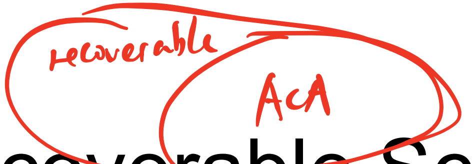
**The same transactions still make the same final Write() to A and B**



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

but serializable

# Recoverable Schedules and ACA



ACA → recoverable

| T1 | T2 |
|---|---|
| | W(A) |
| | Commit |
| R(A) | |
| commit | |

ACA

- Recoverable schedules
  - Any transaction, T1, that reads a change from a different transaction, T2, must commit after T2
  - Not necessarily serializable or vice versa
- Avoid Cascading Aborts (ACA)
  - Transactions only read changes by committed transactions
    - We don't have to abort multiple transactions if one is aborted
  - ACA implies recoverable but not vice versa

T1 read from T2, commit after

not recoverable

| T1 | T2 |
|---|---|
| | W(A) |
| R(A) | |
| Commit | |
| | abort |

already commited
no chance for rollback

| T1 | T2 |
|---|---|
| | W(A) |
| R(A) | |
| | Commit |
| Commit | |

recoverable

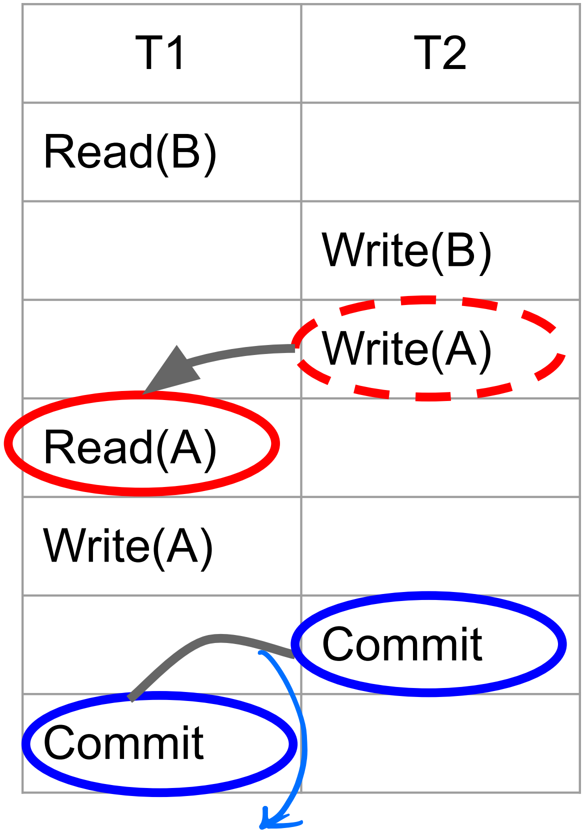
| T1 | T2 |
|---|---|
| | W(A) |
| R(A) | |
| | abort |
| abort | |

recoverable
but not ACA

(cascading abort)

still have chance to abort after T2 abort

# Recoverable and ACA Schedule Examples

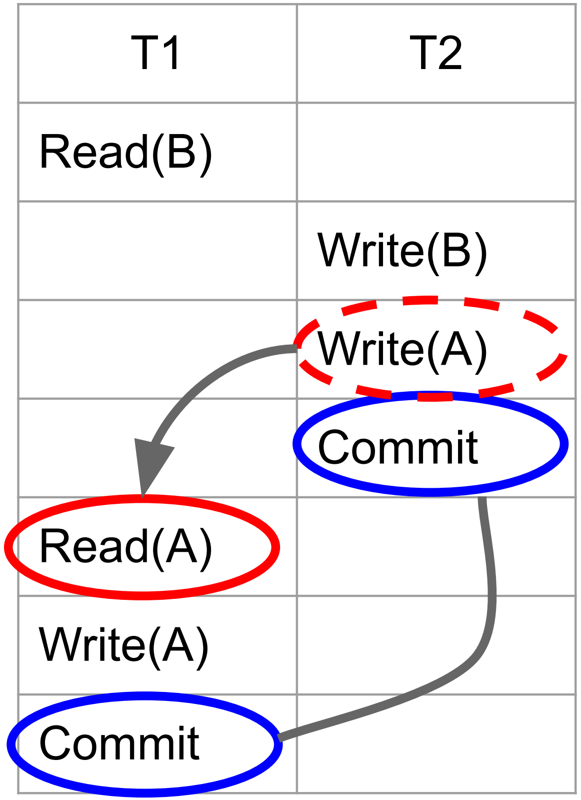


| T1 | T2 |
|---|---|
| Read(B) | |
| | Write(B) |
| | Write(A) |
| Read(A) | |
| Write(A) | |
| | Commit |
| Commit | |

Recoverable but not ACA

**Left:** T1 read uncommitted change made by T2.
- If T2 aborts, T1 has to abort as well. Hence not ACA.
- T1 only commits after T2 commits. Hence it’s recoverable.

**Right:** T1 only reads committed changes. Hence it avoids cascading aborts (ACA)



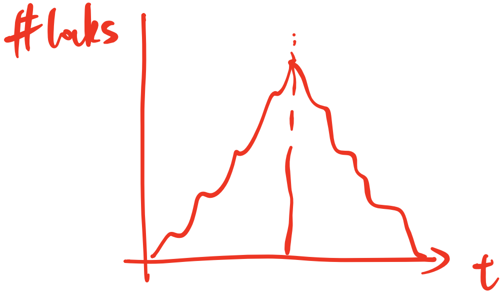
| T1 | T2 |
|---|---|
| Read(B) | |
| | Write(B) |
| | Write(A) |
| | Commit |
| Read(A) | |
| Write(A) | |
| Commit | |

ACA (and thus also recoverable)

# 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
    - 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

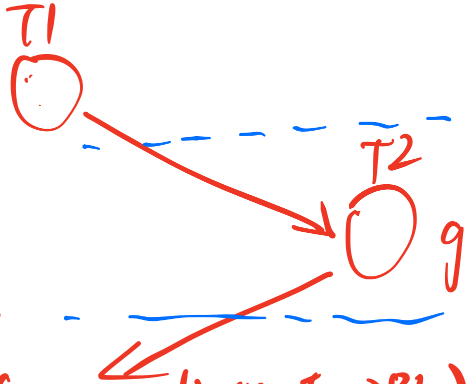
# Two Phase Locking



- 2PL
  - If a transaction releases any lock, it can no longer acquire any new locks
  - Guarantees conflict-serializability (and thus serializability)
  - May sometimes cause deadlocks
- Strict 2PL (S2PL)
  - 2PL + Hold all locks acquired until the end of the transaction
  - Guarantees conflict serializability (and thus serializability)
  - Guarantees ACA (and thus recoverability)

others will action after you COMMIT !!

if cycle:

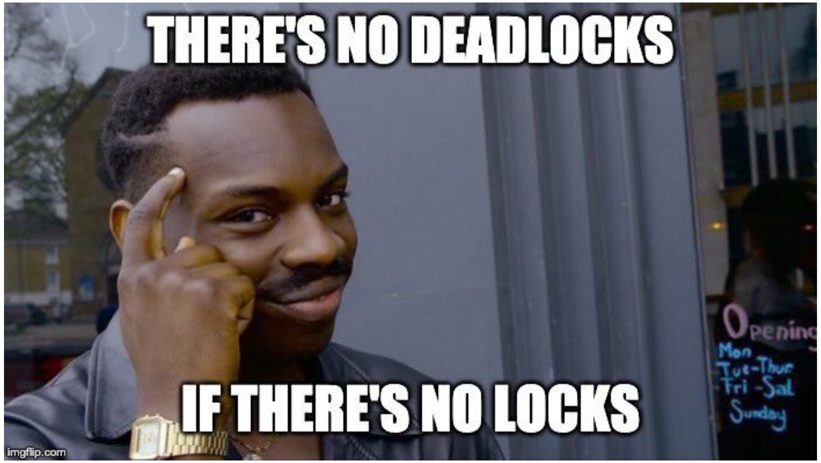


T1 releases lock at some point

grab a lock T1 released

T2 releases lock at some point

T1 grabs a lock (contradicts to 2PL)



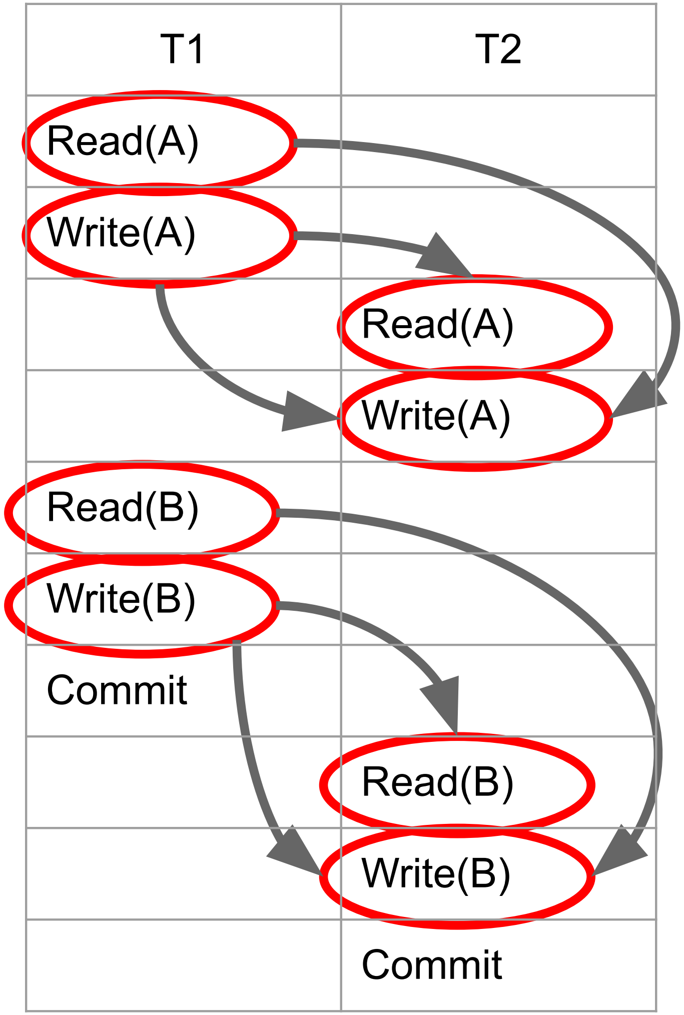
# Example Problems

- List the conflicts between T1 and T2
- Is the schedule conflict serializable?
- Is the schedule serializable?
- Is the schedule recoverable?
- Is the schedule ACA?
- Is the schedule 2PL?
- Is the schedule S2PL?

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

# Example Problems

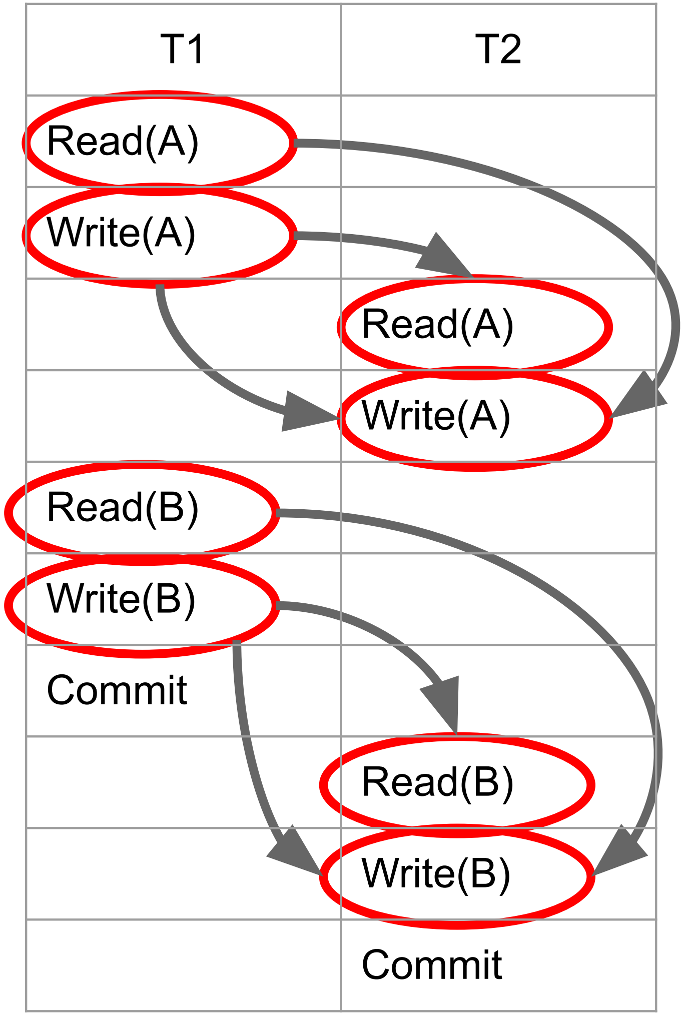
- **List the conflicts between T1 and T2**
  RW conflict from T1 to T2 on A
  WR conflict from T1 to T2 on A
  WW conflict from T1 to T2 on A
  RW conflict from T1 to T2 on B
  WR conflict from T1 to T2 on B
  WW conflict from T1 to T2 on B



| 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 conflict serializable?**
- Yes all T1 → T2 no cycle
- Draw the precedence graph
    - Or examine the dependencies
    - No dependencies from T2 to T1
        - For sure acyclic then



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

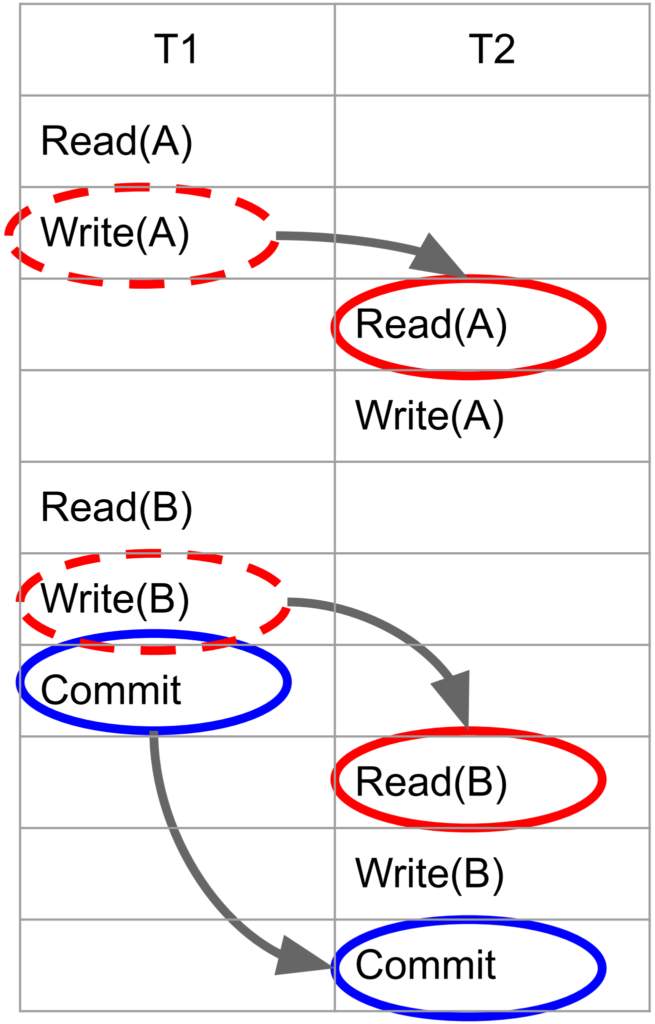
# Example Problems

- **Is the schedule 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 recoverable?**
- T2 reads output of T1 (A and B writes)
- T1 commits before T2
- Recoverable



| 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 ACA?**
- T2 reads output of T1 that is not committed
- Not ACA: if T1 aborts, T2 would have to abort as well

| 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 |

R(A) lock

↓ upgrade

W(A) lock

T1 releases locks

t=3

So T2 can grab R(A) lock

grab R(B) lock ⇒ Not allowed by 2PL

No need for exam

at t=0 T1 grabs exclusive locks on both A & B

# Example Problems

- **Is the schedule S2PL?**
  - 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 recoverable?
- Is the schedule ACA?
- Is the schedule 2PL?
- Is the schedule S2PL?
- Solution on final slide!

| 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
- Is the schedule recoverable?
  - Yes, since T1 reads outputs of T2 and T2 commits first
- Is the schedule ACA?
  - Yes, since T1 reads B only after T2 commits

| 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 S2PL? ✓
  - Yes, T1 and T2 only release exclusive locks on commit

| T1 | T2 |
|---|---|
| share lock A<br>Read(A) | |
| | share lock B<br>Read(B) |
| | exclu<br>Write(B) |
| | share C<br>Read(C) |
| | exclu<br>Write(C) |
| | Commit<br>releases |
| share lock B<br>Read(B) | |
| exclu<br>Write(B) | |
| exclu<br>Write(A) | |
| Commit releases | |

# Get started on P4!