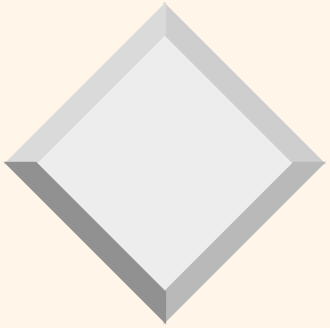


# *Concurrency Control*

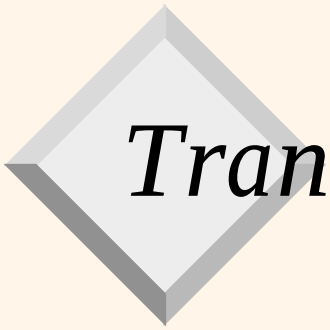
Module 6, Lectures 1 and 2

*The controlling intelligence understands its own nature, and what it does, and whereon it works.*

*-- Marcus Aurelius Antoninus, 121-180 A. D.*

# *Why Have Concurrent Processes?*

- Better transaction throughput, response time
- Done via better utilization of resources:
  - While one processes is doing a disk read, another can be using the CPU or reading another disk.
- **DANGER DANGER!** Concurrency could lead to incorrectness!
  - Must carefully manage concurrent data access.
  - There's (much!) more here than the usual OS tricks!

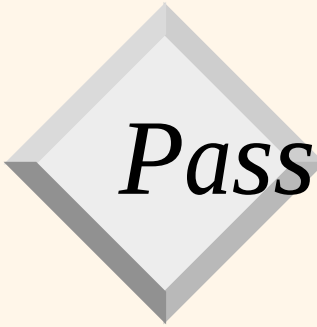


# *Transactions*

- Basic concurrency / recovery concept: a *transaction (Xact).*
  - A sequence of many actions which are considered to be one atomic unit of work.
- DBMS "actions":
  - reads, writes
  - Special actions: commit, abort
  - for now, assume reads and writes are on tuples; we'll revisit this assumption later.

# *The ACID Properties*

- **A** tomicity: All actions in the Xact happen, or none happen.
- **C** onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one Xact is isolated from that of other Xacts.
- **D**urability: If a Xact commits, its effects persist.



# *Passing the ACID Test*

- Concurrency Control
  - Guarantees Consistency and Isolation, given Atomicity.
- Logging and Recovery
  - Guarantees Atomicity and Durability.
- We'll do C. C. today:
  - What problems could arise?
  - What is acceptable behavior?
  - How do we guarantee acceptable behavior?

# Schedules

- Schedule: An interleaving of actions from a set of Xacts, where the actions of any 1 Xact are in the original order.
  - Represents some actual sequence of database actions.
  - Example: $R_1(A)$, $W_1(A)$, $R_2(B)$, $W_2(B)$, $R_1(C)$, $W_1(C)$
  - In a *complete* schedule, each Xact ends in commit or abort.
- Initial State + Schedule $\rightarrow$ Final State

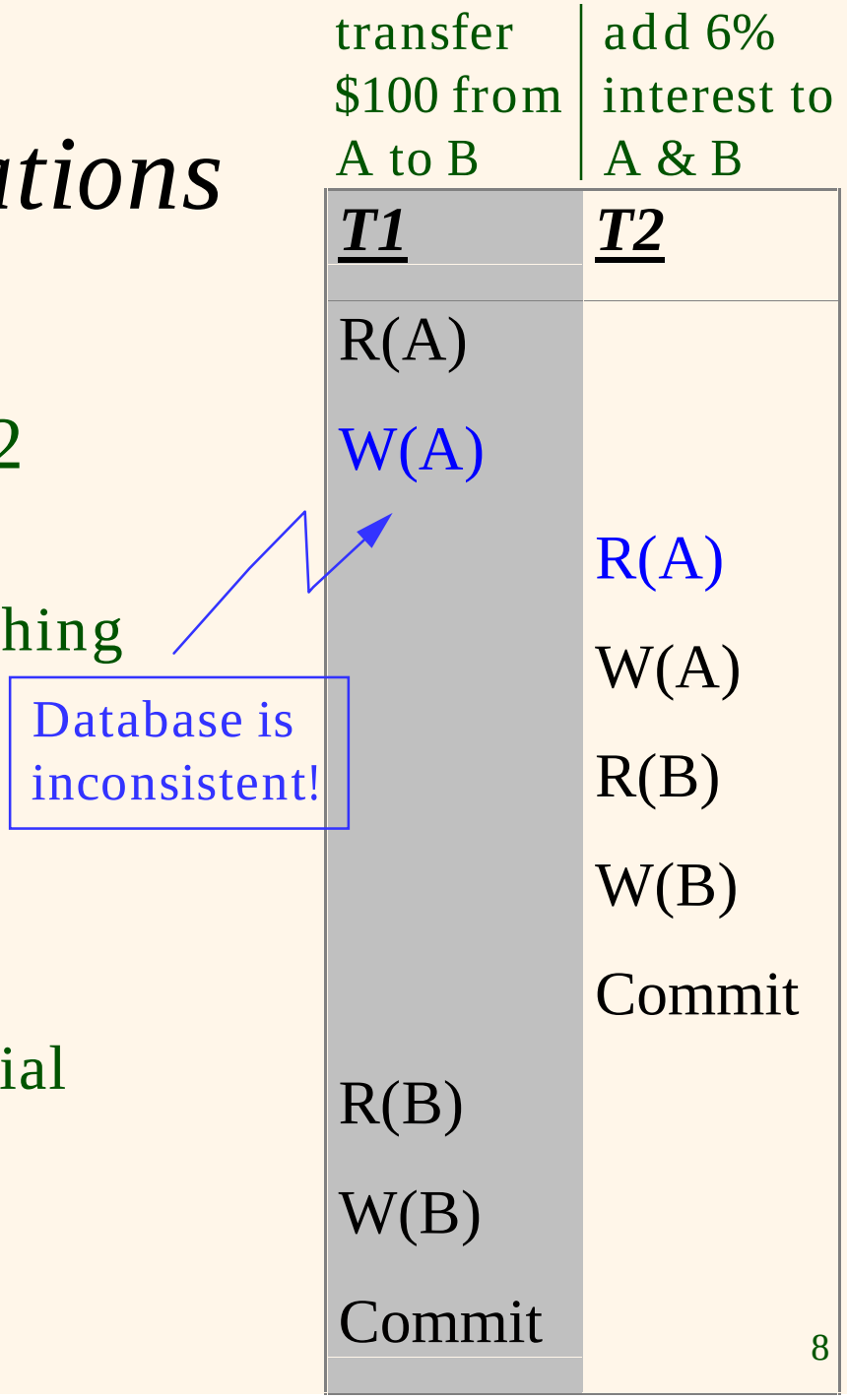
| ***T1*** | ***T2*** |
|---|---|
| R(A) | |
| W(A) | |
| | R(B) |
| | W(B) |
| R(C) | |
| W(C) | |

# *Acceptable Schedules*

- One sensible "isolated, consistent" schedule:
  - Run Xacts one at a time, in a series.
  - This is called a serial schedule.
  - NOTE: Different serial schedules can have different final states; all are "OK" -- DBMS makes no guarantees about the order in which concurrently submitted Xacts are executed.
- Serializable schedules:
  - Final state is what *some* serial schedule would have produced.
  - Aborted Xacts are not part of schedule; ignore them for now (they are made to `disappear' by using logging).

# *Serializability Violations*

- Two actions conflict when 2 xacts access the same item:
  - W-R conflict: T2 reads something T1 wrote.
  - R-W and W-W conflicts: Similar.
- WR conflict (dirty read):
  - Result is not equal to any serial execution!



# *More Conflicts*

- RW Conflicts (Unrepeatable Read)
  - T2 overwrites what T1 read.
  - If T1 reads it again, it will see something new!
    - Example when this would happen?
    - The increment / decrement example.
  - Again, not equivalent to a serial execution.
- WW Conflicts (Overwriting Uncommited Data)
  - T2 overwrites what T1 wrote.
    - Example: 2 Xacts to update items to be kept equal.
  - Usually occurs in conjunction w / other anomalies.
    - Unless you have “blind writes”.

# *Now, Aborted Transactions*

- Serializable schedule: Equivalent to a serial schedule of *committed* Xacts.
  - as if aborted Xacts *never happened.*
- Two Issues:
  - How does one undo the effects of an xact?
    - We'll cover this in logging/recovery
  - What if another Xact sees these effects??
    - Must undo that Xact as well!

# *Cascading Aborts*

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

- Abort of T1 requires abort of T2!
  - Cascading Abort
- What about WW conflicts & aborts?
  - T2 overwrites a value that T1 writes.
  - T1 aborts: its "remembered" values are restored.
  - Lose T2's write! We will see how to solve this, too.
- An ACA (avoids cascading abort) schedule is one in which cascading abort cannot arise.
  - A Xact only reads/writes data from committed Xacts.

# *Recoverable Schedules*

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

- Abort of T1 requires abort of T2!
  - But T2 has already committed!
- A recoverable schedule is one in which this cannot happen.
  - i.e. a Xact commits only after all the Xacts it "depends on" (i.e. it reads from or overwrites) commit.
  - Recoverable implies ACA (but not vice-versa!).
- Real systems typically ensure that only recoverable schedules arise (through locking).

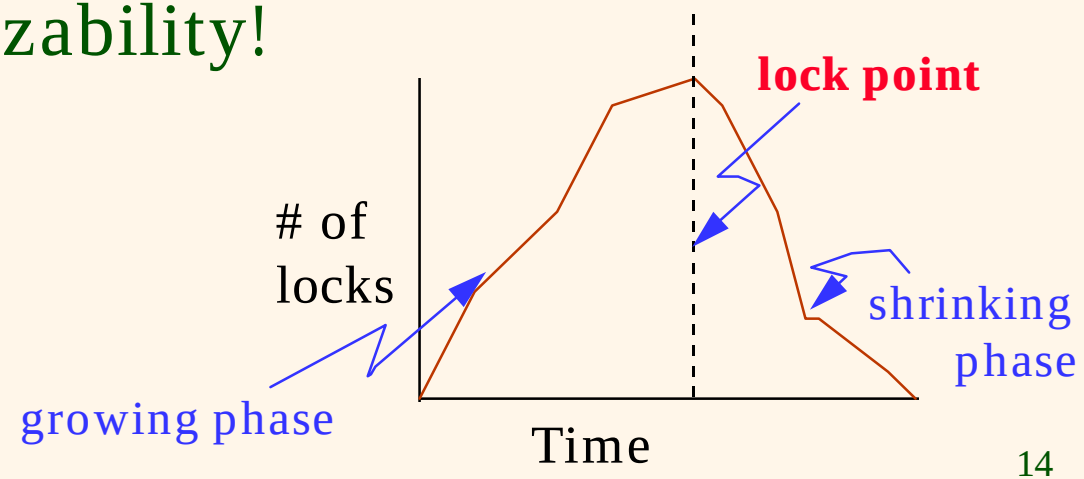
# *Locking: A Technique for C. C.*

- Concurrency control usually done via locking.
- Lock info maintained by a "lock manager":
  - Stores (XID, RID, Mode) triples.
    - This is a simplistic view; suffices for now.
  - Mode ∈ {S,X}
  - Lock compatibility table:
- If a Xact can't get a lock, it is suspended on a wait queue.

| | -- | S | X |
|---|---|---|---|
| -- | √ | √ | √ |
| S | √ | √ | |
| X | √ | | |

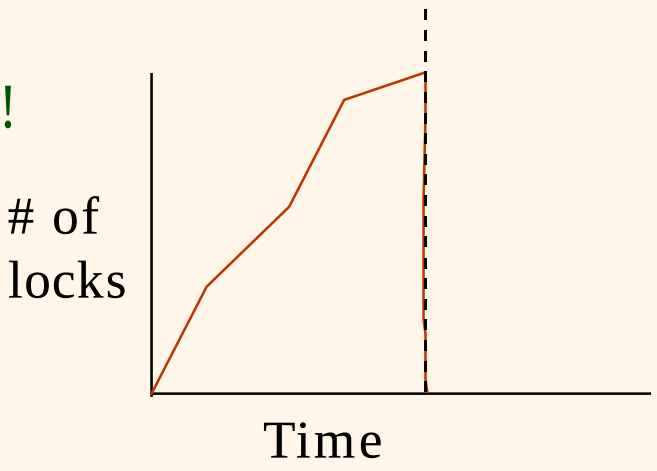
# *Two-Phase Locking (2PL)*

- 2PL:
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - If T releases any lock, it can acquire no new locks!
- Locks are automatically obtained by DBMS.
- *Guarantees* serializability!
  - Why?



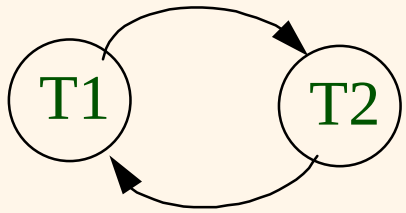
# *Strict 2PL*

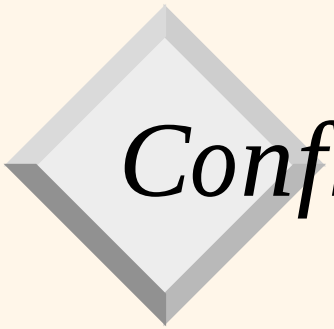
- Strict 2PL:
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - Hold all locks until end of transaction.
- Guarantees serializability, and recoverable schedule, too!
  - also avoids WW problems!



# *Precedence Graph*

- A Precedence (or Serializability) graph:
  - Node for each commited Xact.
  - Arc from Ti to Tj if an action of Ti precedes and conflicts with an action of Tj.
- T1 transfers $100 from A to B, T2 adds 6%
  - $R_1(A)$, $W_1(A)$, $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$, $R_1(B)$, $W_1(B)$





# *Conflict Serializability*

- 2 schedules are conflict equivalent if:
  - they have the same sets of actions, and
  - each pair of conflicting actions is ordered in the same way.
- A schedule is conflict serializable if it is conflict equivalent to a serial schedule.
  - Note: Some serializable schedules are not conflict serializable!

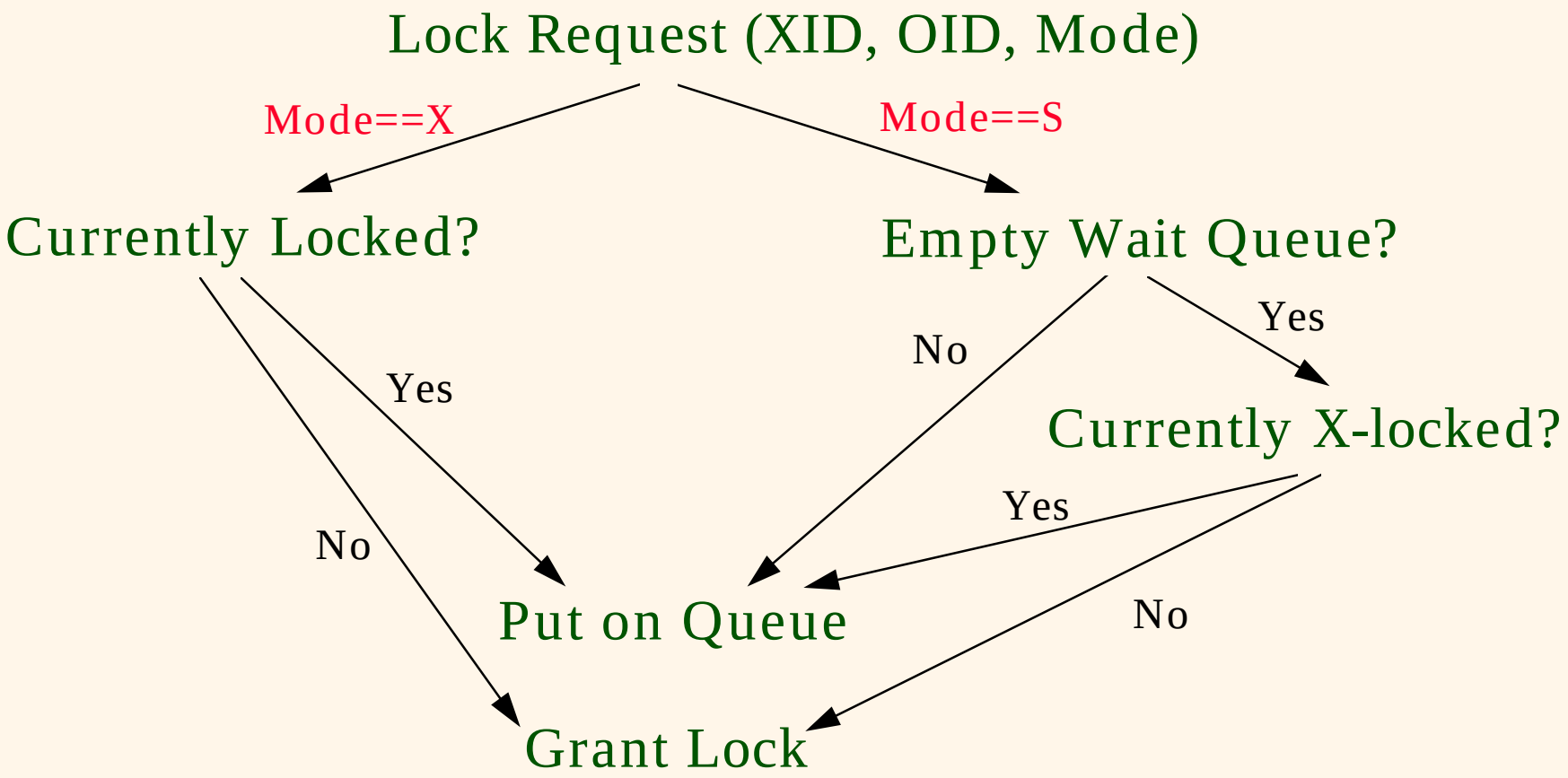
# *Conflict Serializability & Graphs*

- Theorem: A schedule is conflict serializable iff its precedence graph is acyclic.
- Theorem: 2PL ensures that the precedence graph will be acyclic!
- Strict 2PL improves on this by avoiding cascading aborts, problems with undoing WW conflicts; i.e., ensuring recoverable schedules.

# *Lock Manager Implementation*

- Question 1: What are we locking?
  - Tuples, pages, or tables?
  - Finer granularity increases concurrency, but also increases locking overhead.
- Question 2: How do you “lock” something??
- Lock Table: A hash table of Lock Entries.
  - *Lock Entry:*
    - OID
    - Mode
    - List: Xacts holding lock
    - List: Wait Queue

# *Handling a Lock Request*



# *More Lock Manager Logic*

- On lock release (OID, XID):
  - Update list of Xacts holding lock.
  - Examine head of wait queue.
  - If Xact there can run, add it to list of Xacts holding lock (change mode as needed).
  - Repeat until head of wait queue cannot be run.
- Note: Lock request handled atomically!
  - via latches (i.e. semaphores / mutex; OS stuff).

# *Lock Upgrades*

- Think about this scenario:
  - T1 locks A in S mode, T2 requests X lock on A, T3 requests S lock on A. *What should we do?*
- In contrast:
  - T1 locks A in S mode, T2 requests X lock on A, T1 requests X lock on A. *What should we do?*
- Allow such upgrades to supersede lock requests.
  - Consider this scenario:
    - S1(A), X2(A), X1(A): **DEADLOCK!**
- BTW: Deadlock can occur even w/o upgrades:
  - X1(A), X2(B), S1(B), S2(A)

# *Deadlock Prevention*

X1(A), X2(B), S1(B), S2(A)

- Assign a timestamp to each Xact as it enters the system. "Older" Xacts have priority.
- Assume Ti requests a lock, but Tj holds a conflicting lock.
  - Wait-Die: If Ti has higher priority, it waits; else Ti aborts.
  - Wound-Wait: If Ti has higher priority, abort Tj; else Ti waits.
  - Note: After abort, restart with original timestamp!
  - Both guarantee deadlock-free behavior! Pros and cons of each?

# *An Alternative to Prevention*

- In theory, deadlock can involve many transactions:
  - T1 waits-for T2 waits-for T3 ...waits-for T1
- In practice, most “deadlock cycles” involve only 2 transactions.
- Don’t need to prevent deadlock!
  - What’s the problem with prevention?
- Allow it to happen, then notice it and fix it.
  - *Deadlock detection*.

# *Deadlock Detection*

- Lock Mgr maintains a "Waits-for" graph:
  - Node for each Xact.
  - Arc from Ti to Tj if Tj holds a lock and Ti is waiting for it.
- Periodically check graph for cycles.
- "Shoot" some Xact to break the cycle.
- Simpler hack: *time-outs*.
  - T1 made no progress for a while? Shoot it.

To lock such rascal counters from his friends,
Be ready, gods, with all your thunderbolts:
Dash him to pieces!

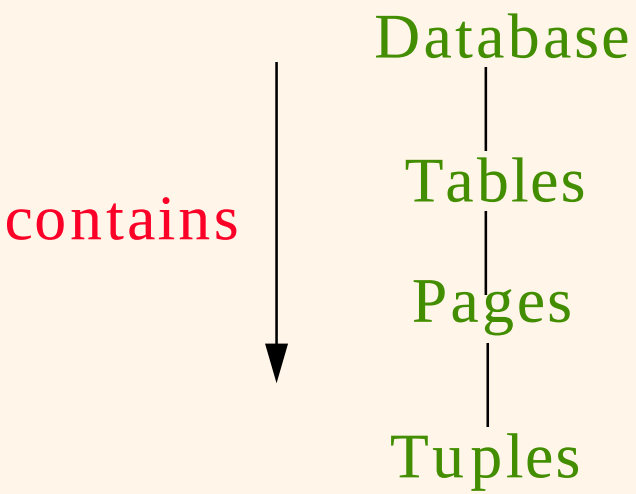
-- Shakespeare, *Julius Caesar*

# *Prevention vs. Detection*

- Prevention might abort too many Xacts.
- Detection might allow deadlocks to tie up resources for a while.
  - Can detect more often, but it's time-consuming.
- The usual answer:
  - Detection is the winner.
  - Deadlocks are pretty rare.
  - If you get a lot of deadlocks, reconsider your schema / workload!

# *Multiple-Granularity Locks*

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to decide!
- Data "containers" are nested:



# *Solution: New Lock Modes, Protocol*

- Allow Xacts to lock at each level, but with a special protocol using new "intention" locks:
- Before locking an item, Xact must set "intention locks" on all its ancestors.
- For unlock, go from specific to general (i.e., bottom-up).
- SIX mode: Like S & IX at the same time.

| | -- | IS | IX | S | X |
|---|---|---|---|---|---|
| -- | √ | √ | √ | √ | √ |
| IS | √ | √ | √ | √ | |
| IX | √ | √ | √ | | |
| S | √ | √ | | √ | |
| X | √ | | | | |

# *Examples*

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.
- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which.

| | -- | IS | IX | S | X |
|---|---|---|---|---|---|
| -- | √ | √ | √ | √ | √ |
| IS | √ | √ | √ | √ | |
| IX | √ | √ | √ | | |
| S | √ | √ | | √ | |
| X | √ | | | | |

# *Summary of C.C.*

- Concurrency control key to a DBMS.
  - More than just mutexes!
- Transactions and the ACID properties:
  - C & I are handled by concurrency control.
  - A & D coming soon with logging & recovery.
- Conflicts arise when two Xacts access the same object, and one of the Xacts is modifying it.
- Serial execution is our model of correctness.

# *Summary, cont.*

- Serializability allows us to “simulate” serial execution with better performance.
- 2PL: A simple mechanism to get serializability.
  - Strict 2PL also gives us recoverability.
- Lock manager module automates 2PL so that only the access methods worry about it.
  - Lock table is a big main-mem hash table
- Deadlocks are possible, and typically a deadlock detector is used to solve the problem.