CS 422 - DBMS

# Lesson 10 - Chapter 22

Lesson10_Transactions-II

# Concurrency Control: 2PL and Deadlock

*Life without problems*

## WHOLENESS OF THE LESSON

A lock based protocol ensures serializability by requiring that access to data items be done in a mutually exclusive manner. This is the way that database integrity is maintained.

Science & Technology of Consciousness: When activity is undertaken from the level of pure intelligence, no errors or obstacles arise.

## Concurrency Control

- Concurrency control refers to the various techniques that are used to preserve the integrity of the database when multiple users are updating the same data items at the same time.
- Incorrect concurrency can lead to problems such as dirty reads, phantom reads, and non-repeatable reads.

## Concurrency Control Techniques

- Two basic concurrency control techniques (for achieving Serializability):
  - **Locking**
  - **Timestamping**
- Both are *conservative* (*pessimistic*) approaches:
  - delay transactions in case they conflict with other transactions some time in future.
- Optimistic methods are available which assume conflict is rare and only check for conflicts at commit.
  - They allow transactions to proceed unsynchronized and check for conflicts only at the end, when a transaction commits.

## Locking

- Locking - A procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results.
- Transaction uses locks to deny access to other transactions in order to prevent incorrect updates.
- Most widely used approach to ensure Serializability.

## Locking

- **Two types of Locks: Shared and Exclusive**
- Generally, a transaction must claim a ***shared* *(read)*** lock or ***exclusive* *(write)*** lock on a data item before the corresponding database read or write.
  - If you are going to do nothing but read an item, you acquire a **shared (read) lock.**
  - If your intent is to modify the data item, you acquire an **exclusive (write) lock**.
  - It is not necessary to get both a read and a write lock to read and write a data item, a write lock is sufficient.
- Lock prevents another transaction from modifying the data item or even reading it, in the case of an exclusive (write) lock.

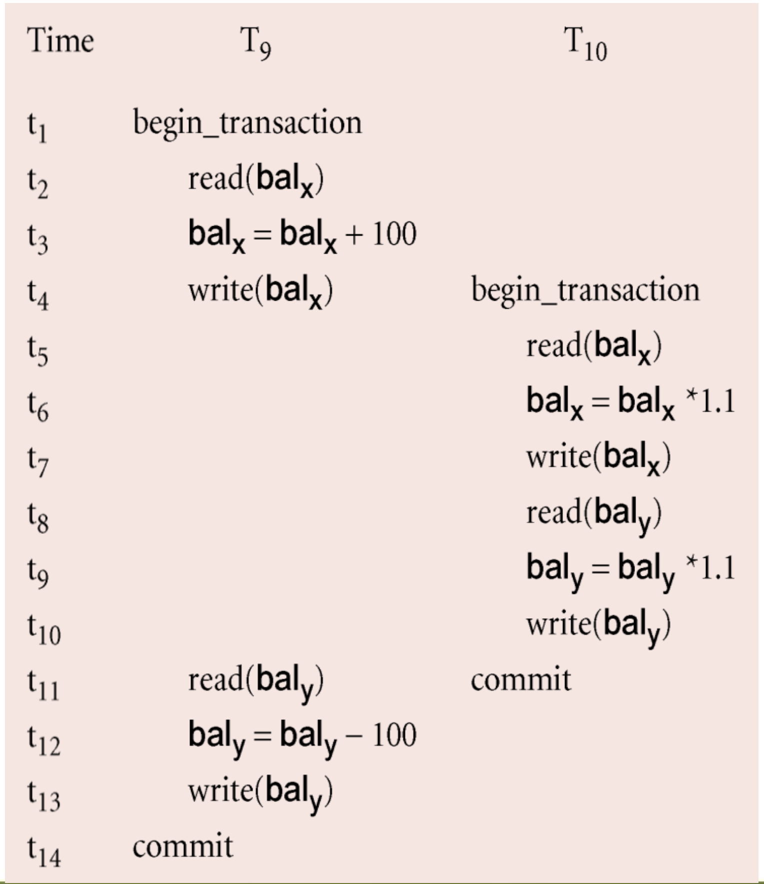
## Locking - Basic Rules

- If a transaction has ***shared*** *lock* on an item, it can ***read*** but not update that item.
- If a transaction has ***exclusive*** *lock* on an item, it can both ***read*** and ***update*** that item.
- Reads cannot conflict, so more than one transaction can hold *shared locks* simultaneously on same item.
- *Exclusive lock* gives transaction exclusive access to that item.
- Some systems allow transaction to upgrade *read lock* to an *exclusive lock*, or downgrade *exclusive lock* to a *shared lock*.

## Example - Incorrect Locking Schedule

- For the given 2 transactions, a valid schedule using the locking rules is:

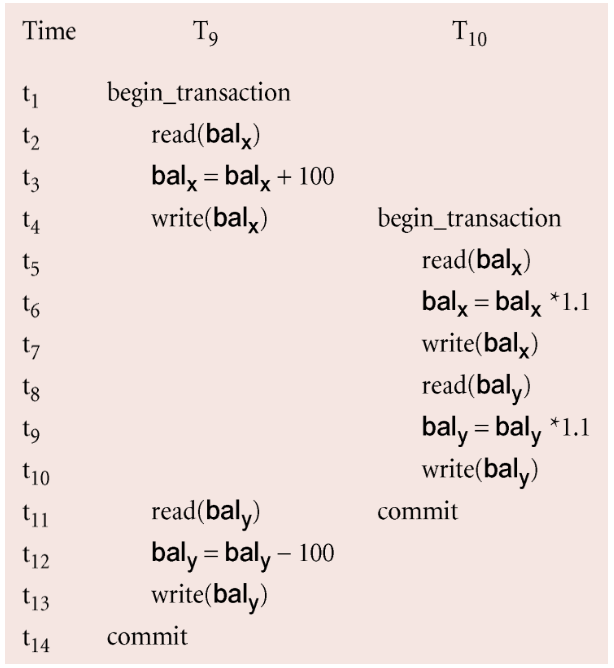
S = {write_lock($T_9$, $bal_x$), read($T_9$, $bal_x$), write($T_9$, $bal_x$), unlock($T_9$, $bal_x$), write_lock($T_{10}$, $bal_x$), read($T_{10}$, $bal_x$), write($T_{10}$, $bal_x$), unlock($T_{10}$, $bal_x$), write_lock($T_{10}$, $bal_y$), read($T_{10}$, $bal_y$), write($T_{10}$, $bal_y$), unlock($T_{10}$, $bal_y$), commit($T_{10}$), write_lock($T_9$, $bal_y$), read($T_9$, $bal_y$), write($T_9$, $bal_y$), unlock($T_9$, $bal_y$), commit($T_9$) }



| Time | $T_9$ | $T_{10}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | read($bal_x$) | |
| $t_3$ | $bal_x = bal_x + 100$ | |
| $t_4$ | write($bal_x$) | begin_transaction |
| $t_5$ | | read($bal_x$) |
| $t_6$ | | $bal_x = bal_x * 1.1$ |
| $t_7$ | | write($bal_x$) |
| $t_8$ | | read($bal_y$) |
| $t_9$ | | $bal_y = bal_y * 1.1$ |
| $t_{10}$ | | write($bal_y$) |
| $t_{11}$ | read($bal_y$) | commit |
| $t_{12}$ | $bal_y = bal_y - 100$ | |
| $t_{13}$ | write($bal_y$) | |
| $t_{14}$ | commit | |

# Example - Incorrect Locking Schedule contd..

- If at start, $bal_x$ = 100, $bal_y$ = 400, result should be:
  - $bal_x$ = 220, $bal_y$ = 330, if $T_9$ executes before $T_{10}$, or
  - $bal_x$ = 210, $bal_y$ = 340, if $T_{10}$ executes before $T_9$.
- However, result gives $bal_x$ = 220 & $bal_y$ = 340.
- **S is not a serializable schedule. (Locks did not help!)**
- Problem is that transactions release locks too soon, resulting in loss of total isolation and atomicity.
- To guarantee Serializability, need an additional protocol concerning the positioning of lock and unlock operations in every transaction.



| Time | $T_9$ | $T_{10}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | read($bal_x$) | |
| $t_3$ | $bal_x = bal_x + 100$ | |
| $t_4$ | write($bal_x$) | begin_transaction |
| $t_5$ | | read($bal_x$) |
| $t_6$ | | $bal_x = bal_x * 1.1$ |
| $t_7$ | | write($bal_x$) |
| $t_8$ | | read($bal_y$) |
| $t_9$ | | $bal_y = bal_y * 1.1$ |
| $t_{10}$ | | write($bal_y$) |
| $t_{11}$ | read($bal_y$) | commit |
| $t_{12}$ | $bal_y = bal_y - 100$ | |
| $t_{13}$ | write($bal_y$) | |
| $t_{14}$ | commit | |

## Two-Phase Locking (2PL)

**A transaction follows the two-phase locking protocol if all locking operations precede the first unlock operation in the transaction.**

- A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.
- Once the transaction releases a lock, it can never acquire any new locks.
- 2PL ensures conflict serializable schedules.

## 2PL Rules

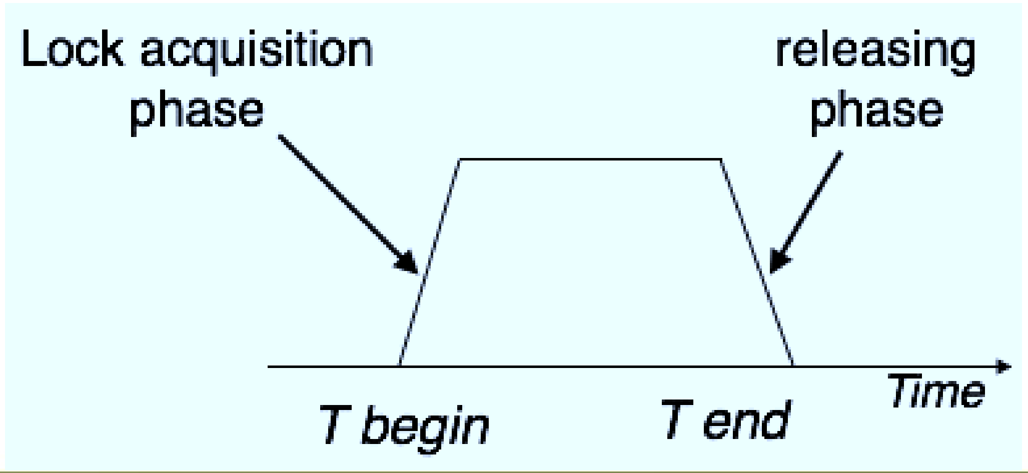
- **In 2PL, there are two types of locks:**

  Read_lock (or shared lock) and
  Write_lock (or exclusive lock)

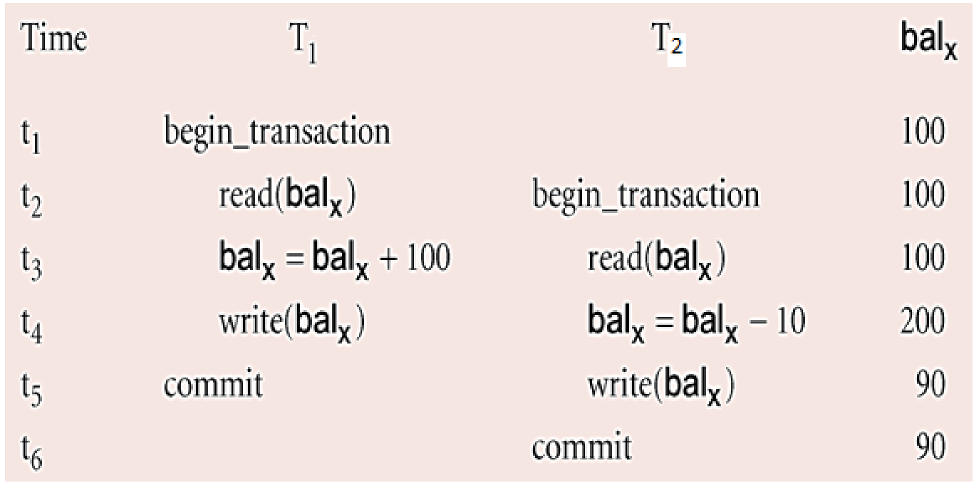
- A transaction must have the  Read_lock to Read
- A transaction must have the Write_lock to Write. If a transaction has a Write_lock, it can read as well.
- A transaction must WAIT, if the necessary lock is not available.

## Phases of 2PL

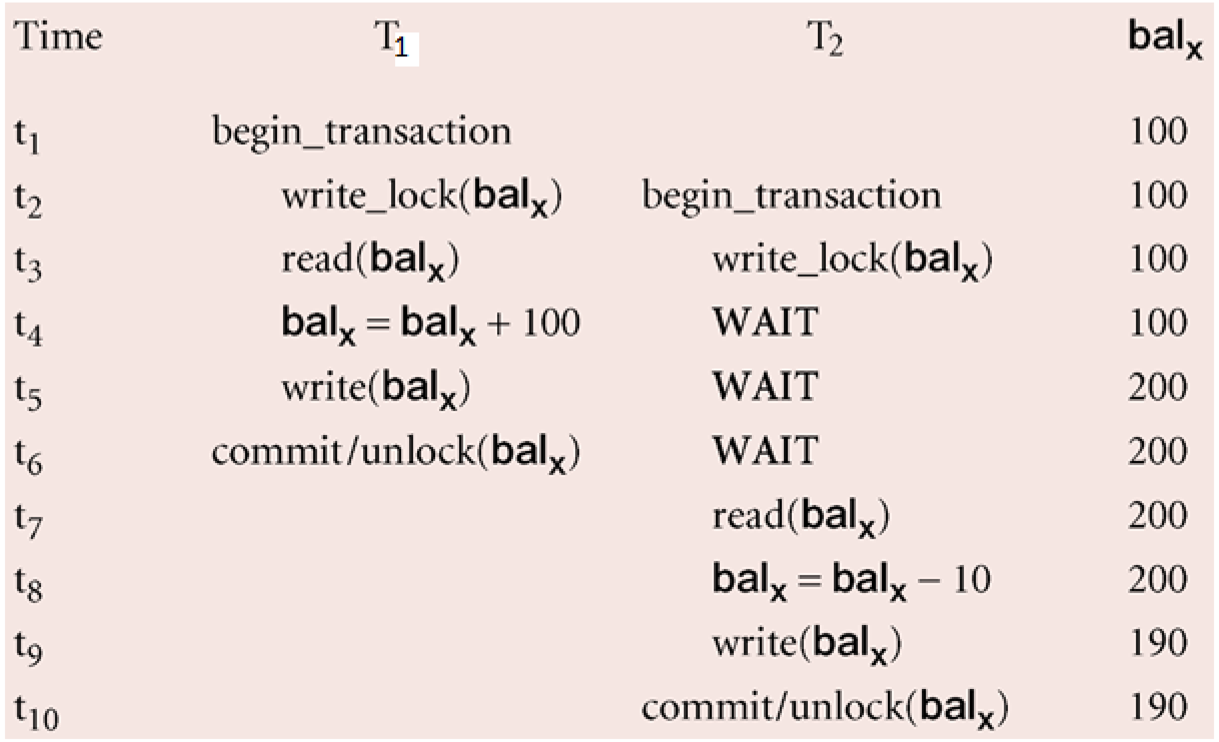
- Every transaction can be divided into two phases:
  - **Growing phase** - acquires all locks but cannot release any locks.
    In this first part, when the transaction starts executing, it seeks permission for the locks it requires.
  - **Shrinking phase** - releases locks but cannot acquire any new locks.
    This phase starts as soon as the transaction releases its first lock.



## Preventing Lost Update Problem using 2PL

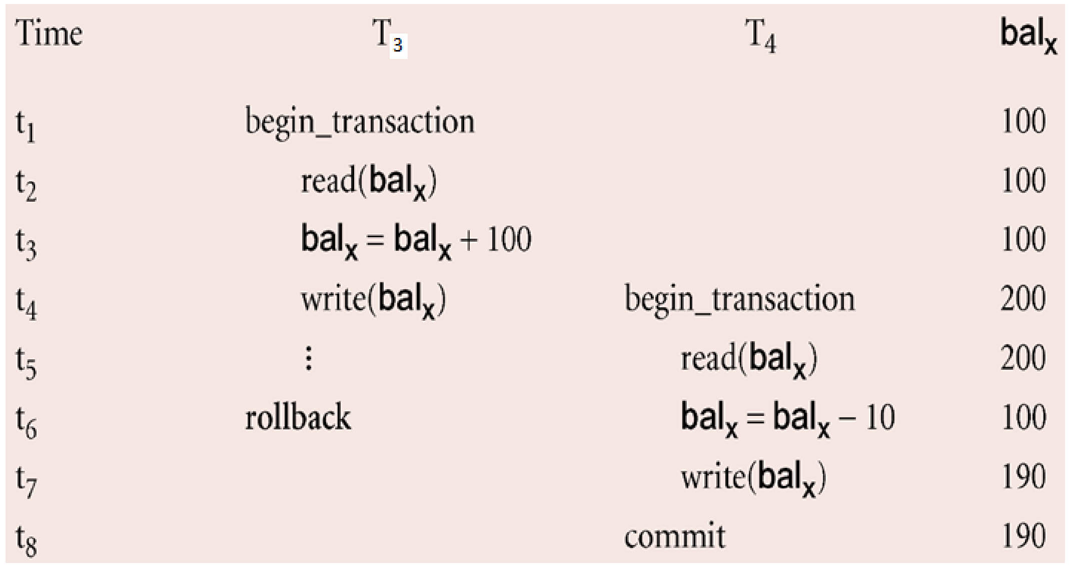


| Time | $T_1$ | $T_2$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | begin_transaction | | 100 |
| $t_2$ | read($bal_x$) | begin_transaction | 100 |
| $t_3$ | $bal_x = bal_x + 100$ | read($bal_x$) | 100 |
| $t_4$ | write($bal_x$) | $bal_x = bal_x - 10$ | 200 |
| $t_5$ | commit | write($bal_x$) | 90 |
| $t_6$ | | commit | 90 |

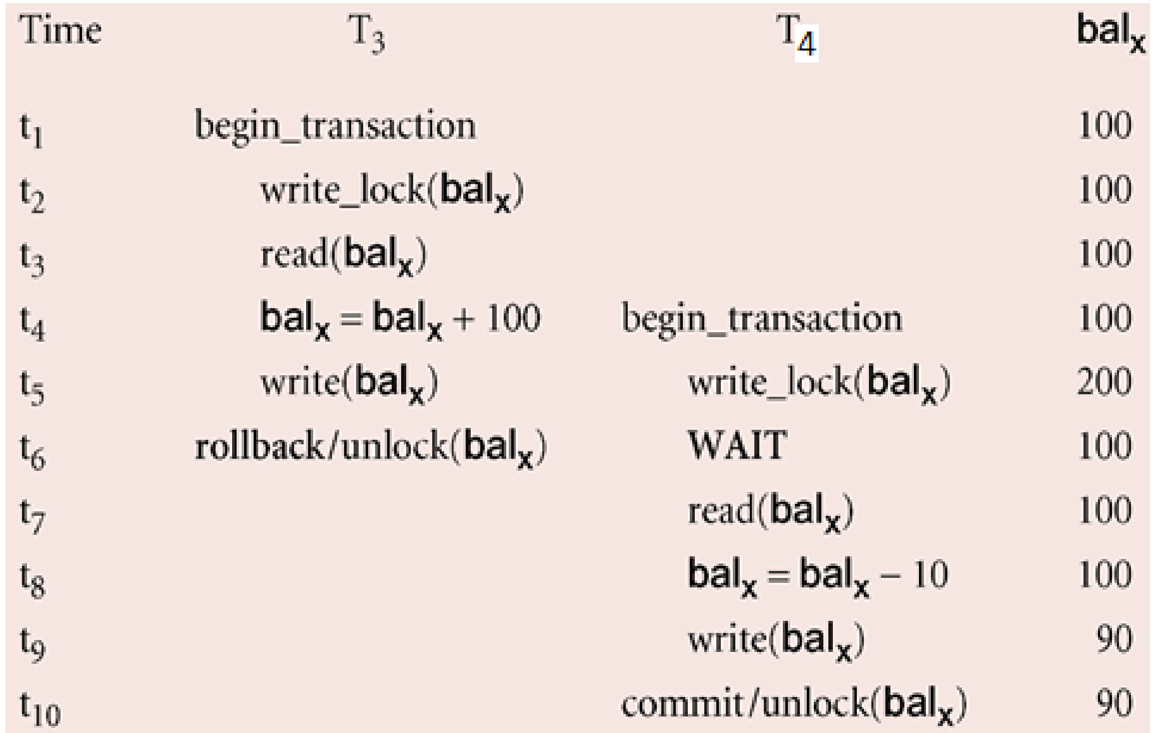


| Time | $T_1$ | $T_2$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | begin_transaction | | 100 |
| $t_2$ | write_lock($bal_x$) | begin_transaction | 100 |
| $t_3$ | read($bal_x$) | write_lock($bal_x$) | 100 |
| $t_4$ | $bal_x = bal_x + 100$ | WAIT | 100 |
| $t_5$ | write($bal_x$) | WAIT | 200 |
| $t_6$ | commit/unlock($bal_x$) | WAIT | 200 |
| $t_7$ | | read($bal_x$) | 200 |
| $t_8$ | | $bal_x = bal_x - 10$ | 200 |
| $t_9$ | | write($bal_x$) | 190 |
| $t_{10}$ | | commit/unlock($bal_x$) | 190 |

## Preventing Uncommitted Dependency Problem using 2PL

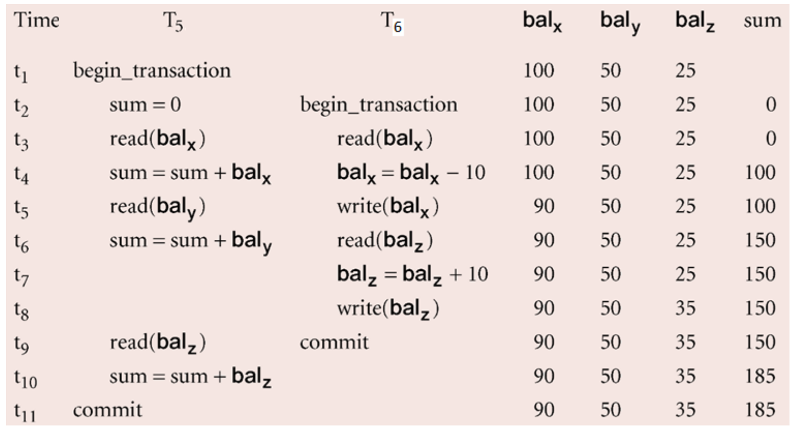


| Time | $T_3$ | $T_4$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | begin_transaction | | 100 |
| $t_2$ | read($bal_x$) | | 100 |
| $t_3$ | $bal_x = bal_x + 100$ | | 100 |
| $t_4$ | write($bal_x$) | begin_transaction | 200 |
| $t_5$ | ⋮ | read($bal_x$) | 200 |
| $t_6$ | rollback | $bal_x = bal_x - 10$ | 100 |
| $t_7$ | | write($bal_x$) | 190 |
| $t_8$ | | commit | 190 |

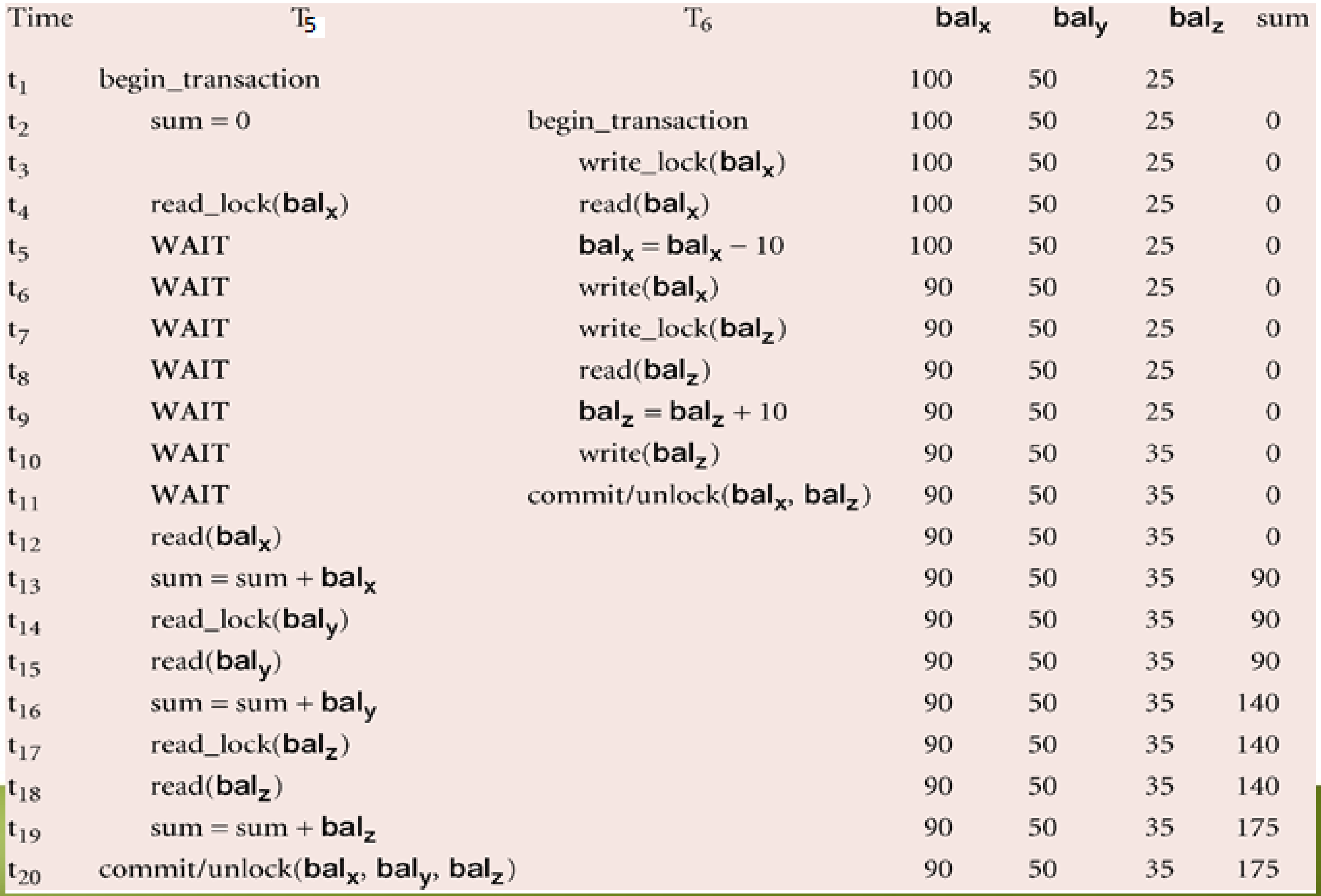


| Time | $T_3$ | $T_4$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | begin_transaction | | 100 |
| $t_2$ | write_lock($bal_x$) | | 100 |
| $t_3$ | read($bal_x$) | | 100 |
| $t_4$ | $bal_x = bal_x + 100$ | begin_transaction | 100 |
| $t_5$ | write($bal_x$) | write_lock($bal_x$) | 200 |
| $t_6$ | rollback/unlock($bal_x$) | WAIT | 100 |
| $t_7$ | | read($bal_x$) | 100 |
| $t_8$ | | $bal_x = bal_x - 10$ | 100 |
| $t_9$ | | write($bal_x$) | 90 |
| $t_{10}$ | | commit/unlock($bal_x$) | 90 |

# Preventing Inconsistent Analysis Problem using 2PL



| Time | $T_5$ | $T_6$ | $bal_x$ | $bal_y$ | $bal_z$ | sum |
|---|---|---|---|---|---|---|
| $t_1$ | begin_transaction | | 100 | 50 | 25 | |
| $t_2$ | sum = 0 | begin_transaction | 100 | 50 | 25 | 0 |
| $t_3$ | read($bal_x$) | read($bal_x$) | 100 | 50 | 25 | 0 |
| $t_4$ | sum = sum + $bal_x$ | $bal_x = bal_x - 10$ | 100 | 50 | 25 | 100 |
| $t_5$ | read($bal_y$) | write($bal_x$) | 90 | 50 | 25 | 100 |
| $t_6$ | sum = sum + $bal_y$ | read($bal_z$) | 90 | 50 | 25 | 150 |
| $t_7$ | | $bal_z = bal_z + 10$ | 90 | 50 | 25 | 150 |
| $t_8$ | | write($bal_z$) | 90 | 50 | 35 | 150 |
| $t_9$ | read($bal_z$) | commit | 90 | 50 | 35 | 150 |
| $t_{10}$ | sum = sum + $bal_z$ | | 90 | 50 | 35 | 185 |
| $t_{11}$ | commit | | 90 | 50 | 35 | 185 |



| Time | $T_5$ | $T_6$ | $bal_x$ | $bal_y$ | $bal_z$ | sum |
|---|---|---|---|---|---|---|
| $t_1$ | begin_transaction | | 100 | 50 | 25 | |
| $t_2$ | sum = 0 | begin_transaction | 100 | 50 | 25 | 0 |
| $t_3$ | | write_lock($bal_x$) | 100 | 50 | 25 | 0 |
| $t_4$ | read_lock($bal_x$) | read($bal_x$) | 100 | 50 | 25 | 0 |
| $t_5$ | WAIT | $bal_x = bal_x - 10$ | 100 | 50 | 25 | 0 |
| $t_6$ | WAIT | write($bal_x$) | 90 | 50 | 25 | 0 |
| $t_7$ | WAIT | write_lock($bal_z$) | 90 | 50 | 25 | 0 |
| $t_8$ | WAIT | read($bal_z$) | 90 | 50 | 25 | 0 |
| $t_9$ | WAIT | $bal_z = bal_z + 10$ | 90 | 50 | 25 | 0 |
| $t_{10}$ | WAIT | write($bal_z$) | 90 | 50 | 35 | 0 |
| $t_{11}$ | WAIT | commit/unlock($bal_x$, $bal_z$) | 90 | 50 | 35 | 0 |
| $t_{12}$ | read($bal_x$) | | 90 | 50 | 35 | 0 |
| $t_{13}$ | sum = sum + $bal_x$ | | 90 | 50 | 35 | 90 |
| $t_{14}$ | read_lock($bal_y$) | | 90 | 50 | 35 | 90 |
| $t_{15}$ | read($bal_y$) | | 90 | 50 | 35 | 90 |
| $t_{16}$ | sum = sum + $bal_y$ | | 90 | 50 | 35 | 140 |
| $t_{17}$ | read_lock($bal_z$) | | 90 | 50 | 35 | 140 |
| $t_{18}$ | read($bal_z$) | | 90 | 50 | 35 | 140 |
| $t_{19}$ | sum = sum + $bal_z$ | | 90 | 50 | 35 | 175 |
| $t_{20}$ | commit/unlock($bal_x$, $bal_y$, $bal_z$) | | 90 | 50 | 35 | 175 |

At time $t_7$, transaction $T_6$ was done with $bal_x$ but it did not release the lock on $bal_x$ as it still needs to acquire another lock on $bal_z$.
Under 2PL rules, once you do an unlock, you cannot acquire any new locks.

## Example: 2PL → S: $T_1(R,Y)$, $T_2(R,X)$, $T_3(W,X)$, $T_2(W,X)$, $T_1(W,Y)$, $c(T_1)$, $c(T_2)$, $c(T_3)$

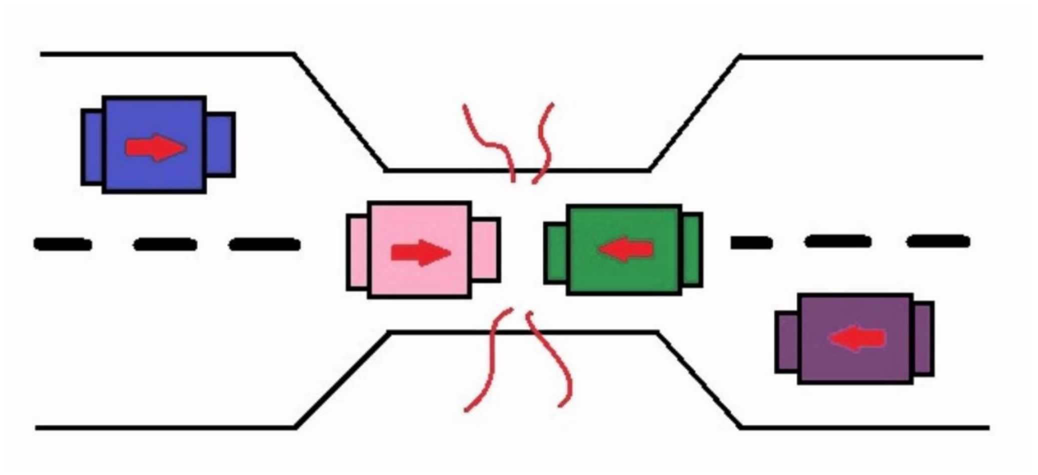
| | | |
|---|---|---|
| $t_1$ | START ($T_1$) | |
| $t_2$ | WL ($T_1$, Y) | // $T_1$ asks for Write lock on Y and gets it |
| $t_3$ | $T_1$ (R, Y) | |
| $t_4$ | START ($T_2$) | |
| $t_5$ | WL ($T_2$, X) | // $T_2$ asks for Write lock on X and get's it |
| $t_6$ | $T_2$ (R, X) | |
| $t_7$ | START ($T_3$) | |
| $t_8$ | WL ($T_3$, X) | // $T_3$ asks for Write lock on X but needs to wait |
| $t_9$ | $T_2$ (W, X) | // $T_2$ continues |
| $t_{10}$ | $T_1$ (W, Y) | |
| $t_{11}$ | c($T_1$) | // $T_1$ commits and unlocks Y |
| $t_{12}$ | c($T_2$) | // $T_2$ commits and unlocks X |
| $t_{13}$ | $T_3$ (W, X) | // $T_3$ now acquires WL on X and continues |
| $t_{14}$ | c($T_3$) | // $T_3$ commits and unlocks X |

## Main Point

The Two-Phase Locking Protocol provides a systematic approach for managing concurrent transactions in a database to prevent conflicts and ensure serializability. Science & Technology of Consciousness: The TM technique offers a systematic way for individuals to enhance mental well-being and experience a deeper level of consciousness which prevents conflicts in life.

## Deadlock

An impasse (a situation in which no progress is possible) that may result when two (or more) transactions are each waiting for locks held by the other to be released.



## Deadlock

| Time | $T_{17}$ | $T_{18}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | write_lock($bal_x$) | begin_transaction |
| $t_3$ | read($bal_x$) | write_lock($bal_y$) |
| $t_4$ | $bal_x = bal_x - 10$ | read($bal_y$) |
| $t_5$ | write($bal_x$) | $bal_y = bal_y + 100$ |
| $t_6$ | write_lock($bal_y$) | write($bal_y$) |
| $t_7$ | WAIT | write_lock($bal_x$) |
| $t_8$ | WAIT | WAIT |
| $t_9$ | WAIT | WAIT |
| $t_{10}$ | $\vdots$ | WAIT |
| $t_{11}$ | $\vdots$ | $\vdots$ |

## Deadlock

- Only one way to break deadlock: abort one or more of the transactions.
- Deadlock should be transparent to user, so DBMS should automatically restart the aborted transaction(s).
  - However, in practice DBMS cannot restart aborted transaction since it is unaware of transaction logic even if it was aware of the transaction history (unless there is no user input in the transaction, or the input is not a function of the database state).

## Techniques for Handling Deadlock

- Three general techniques for handling deadlock:
  - Timeouts
  - Deadlock prevention
  - Deadlock detection and recovery

## Techniques for Handling Deadlock: Timeouts

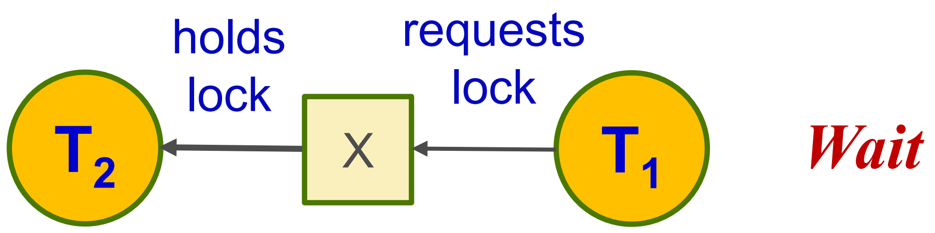
- Transaction that requests lock will only wait for a system-defined period of time.
- If lock has not been granted within this period, lock request times out.
- In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.
- This is a very simple and practical solution to deadlock prevention that is used by several commercial DBMSs.

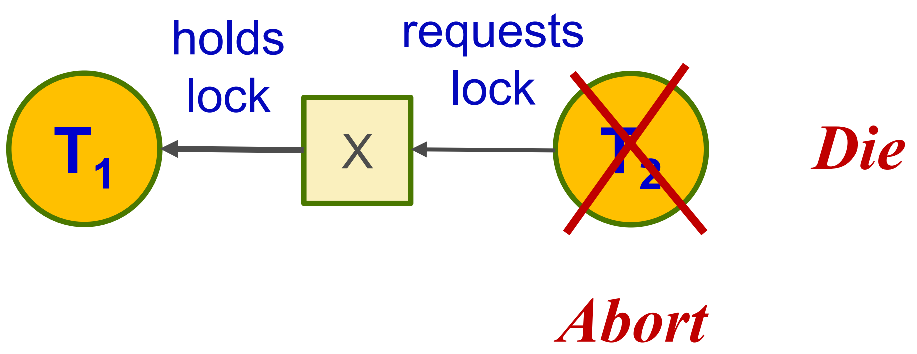
## Techniques for Handling Deadlock: Deadlock Prevention

- DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.
- Transactions are ordered using transaction timestamps.
  - Each transaction is assigned a ***unique increasing*** timestamp: $T_1$, $T_2$, $T_3$, ...
  - ***Earlier*** **transactions** receive a ***smaller*** **timestamp.**
- Suppose that a transaction requests a lock on a DB element that has been locked by another transaction then there are two algorithms for deadlock prevention:
  - **Wait-Die**
  - **Wound-Wait**

## Deadlock Prevention : Wait-Die

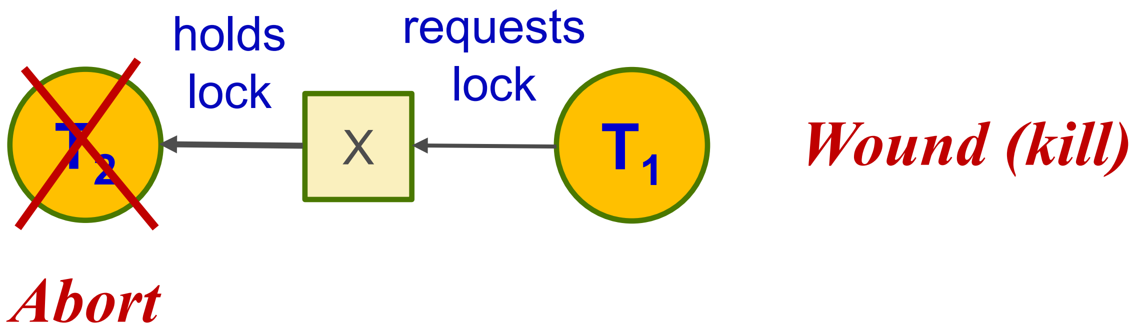
- In this algorithm, only an older transaction can wait for younger one.
- If a situation arises that the younger one needs to wait for the older one, then that younger transaction is aborted (*dies*) and restarted with the *same timestamp* so that eventually it'll become the oldest active transaction and will not die.

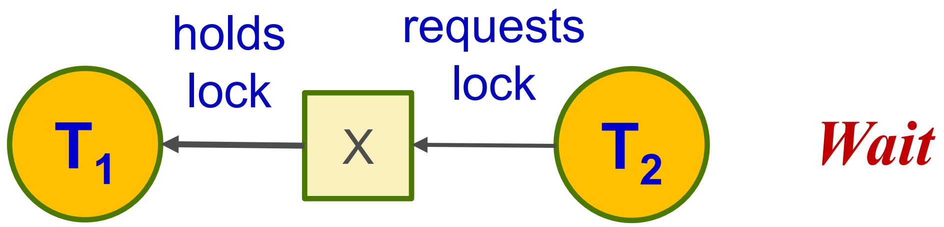




# Deadlock Prevention : Wound-Wait

- In this algorithm, if older transaction requests lock held by younger one, then the younger one is aborted (*wounded*) and later restarted with *same timestamp*.
- In this algorithm, only younger transaction can wait for an older one.





## Example execution that obeys the **wait-die** rule:

T1(R,A), T1(W,B)

T2(W,A) ,T2(R,C)

T3(R,B), T3(W,C)

T4(W,A) ,T4(R,D)

| T1 | T2 | T3 | T4 |
|---|---|---|---|
| L(A), R(A) | | | |
| | L(A) *die!!* | | |
| | | L(B), R(B) | |
| | | | L(A) *die!!* |
| | | L(C), W(C) U(B), U(C) | |
| L(B), W(B) U(A), U(B) | | | *Restart* |
| | | | L(A), W(A) |
| | *Restart* | | L(D), R(D) |
| | L(A) *wait!!* | | |
| | | | U(A), U(D) |
| | L(A),W(A),L(C),R(C) U(A), U(C) | | |

L → Lock
(could be read or write)

U → Unlock

R → Read

W → Write

## Example execution that obeys the **wound-wait** rule:

T1(R,A), T1(W,B)

T2(W,A) ,T2(R,C)

T3(R,B), T3(W,C)

T4(W,A) ,T4(R,D)

| T1 | T2 | T3 | T4 |
|---|---|---|---|
| L(A), R (A) | | | |
| | L(A) *wait!!* | | |
| | | L(B), R(B) | |
| | *Wound (kill)!!* | | L(A) *wait!!* |
| L(B) W(B) | | | |
| U(A), U(B) | | | |
| | L(A), W(A) L(C), R(C) U(A), U(C) | | |
| | | | L(A), W(A) L(D), R(D) U(D), U(A) |
| | | *Restart* | |
| | | L(B), R(B) L(C), W(C) U(B), U(C) | |

L → Lock
(could be read or write)

U → Unlock

R → Read

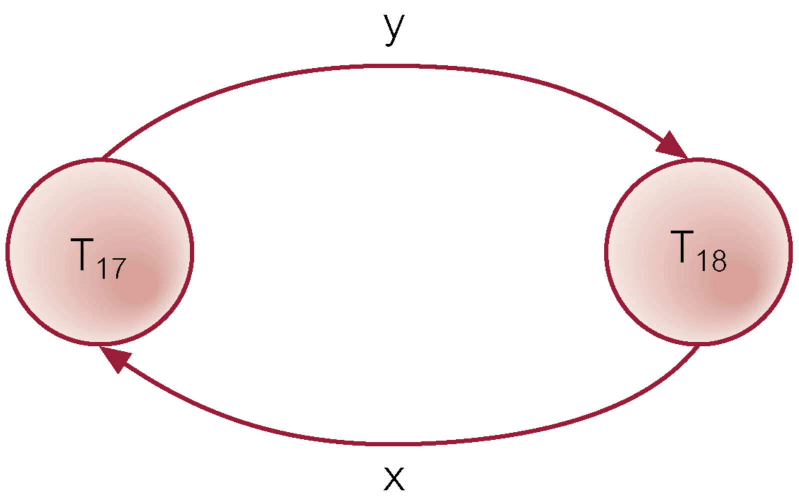
W → Write

## Techniques for Handling Deadlock: **Deadlock Detection and Recovery**

- DBMS allows deadlock to occur but recognizes it and breaks it.
- Usually handled by construction of **wait-for graph (WFG)** showing transaction dependencies:
  - Create a node for each transaction.
  - Create a directed edge $T_i$ -> $T_j$, if $T_i$ is waiting to lock an item that is currently locked by $T_j$.
- Deadlock exists if and only if WFG contains cycle.
- WFG is created at regular intervals by deadlock detection algorithms.

# Example - Wait-For-Graph (WFG)

| Time | $T_{17}$ | $T_{18}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | write_lock($\mathbf{bal_x}$) | begin_transaction |
| $t_3$ | read($\mathbf{bal_x}$) | write_lock($\mathbf{bal_y}$) |
| $t_4$ | $\mathbf{bal_x} = \mathbf{bal_x} - 10$ | read($\mathbf{bal_y}$) |
| $t_5$ | write($\mathbf{bal_x}$) | $\mathbf{bal_y} = \mathbf{bal_y} + 100$ |
| $t_6$ | write_lock($\mathbf{bal_y}$) | write($\mathbf{bal_y}$) |
| $t_7$ | WAIT | write_lock($\mathbf{bal_x}$) |
| $t_8$ | WAIT | WAIT |
| $t_9$ | WAIT | WAIT |
| $t_{10}$ | ⋮ | WAIT |
| $t_{11}$ | ⋮ | ⋮ |


$$\begin{tikzcd} T_{17} \arrow[r, bend left, "y"] & T_{18} \arrow[l, bend left, "x"] \end{tikzcd}$$

## Recovery from Deadlock Detection

- Once a deadlock is detected, DBMS needs to abort one or more of the transactions.
- **Several issues:**
  - Choice of deadlock victim
    - Which transaction should be aborted?
  - How far to roll a transaction back
    - You may want to rollback up to a point so that enough resources are released to break the deadlock.
  - Avoiding starvation
    - Avoid a situation where the same transaction is getting killed every time.

# In-Class Exercise

- The following table summarizes the status of six transactions, showing what relations they have each locked and what relations they need in order to proceed:

| Transaction | Relations Held | Relations Needed |
|---|---|---|
| T1 | R3 | R2 |
| T2 | R5 | R2 |
| T3 | | R1, R3 |
| T4 | R1 | R3, R5 |
| T5 | | R5 |
| T6 | R2 | R1, R4 |

- Draw a wait-for graph to represent this situation.
- Are the transactions in deadlock?

## Main Point

Deadlocks in DBMS happen when transactions are blocked indefinitely waiting for each other to release locks. Science & Technology of Consciousness: Similar to the challenges posed by Deadlocks in a DBMS, in various aspects of life, individuals might encounter situations akin to a deadlock, where progress is halted because of interdependencies and a lack of resolution. A TM practitioner gets all the Nature support to break the deadlocks in life.

## 2PL Example Problem

Consider the following sequence of actions, listed in the order the actions are presented to the DBMS.

Assume that the concurrency control mechanism is 2PL with "Wait-Die" deadlock prevention strategy.

Describe how the concurrency control mechanism handles the following sequence of actions.

T1: R(X), T2: W(X), T2: W(Y), T3: W(Y), T1: W(Y), T1: Commit, T2: Commit, T3: Commit

## Solution

T1: R(X), T2: W(X), T2: W(Y), T3: W(Y), T1: W(Y), T1: Commit, T2: Commit, T3: Commit

**t1** - T1 wants shared lock on X and gets it
**T1:R(X)**

**t2** - T2 wants exclusive lock on X, but since T2 is younger than T1, T2 aborts (will be restarted later : wait-die DL prevention policy)

**t3** - T3 asks for exclusive lock on Y and gets it
**T3:W(Y)**

**t4** - T1 wants exclusive lock on Y, but got denied; T1 waits for T3

**t5** - T3 commits & releases write-lock on Y

**t6** - T1 gets exclusive lock on Y
**T1:W(Y)**
**t7 -** T1 commits & releases locks on X and Y

**t8** - T2 restarts
T2 gets exclusive locks on X and then Y, performs operations **T2:R(X)**, **T2:W(Y)**; commits & releases locks.

# Another Example

Under 2PL and wait-die, acquire locks as late as possible and release locks as early as possible. Waiting transactions continued and brought up to date as early as possible.

**T1: W(X), T2: R(X), T2: W(Y), T3: W(Y), T1: W(Y), T3:R(Y), T1:R(Y), T1: Commit, T2: Commit, T3: Commit**

t1 - T1 gets exclusive lock on X → **T1:W(X)**

t2 - T2 wants read lock on X, but since T2 is younger than T1, T2 aborts

t3 - T3 gets exclusive lock on Y → **T3:W(Y)**

t4 - T1 wants exclusive lock on Y, T1 waits for T3

t5 - T3 continues with read Y → **T3:R(Y)**

t6 - T3 commits & releases lock Y

t7 - T1 gets exclusive lock on Y

t8 - T1 releases lock on X

t9 - T2 restarts and gets shared lock on X → **T2:R(X)**

t10 - **T1:W(Y)**

t11 - **T1:R(Y)**

t12 - T1 commits & releases lock Y

t13 - T2 get exclusive lock on Y → **T2:W(Y)**

t14 – T2 commits & releases locks

Release locks as early as possible, so after getting lock on Y, T1 does not proceed with W(Y), instead first releases lock on X.

Waiting transactions continued and brought up to date as early as possible.

## Is 2PL reducing concurrency in conflict serializable schedule?

- We know that serializable schedules of multiple concurrent transactions are guaranteed consistent.
- We also know that it is impractical to retrospectively test a schedule of transactions to see if it was serializable.
- The solution lies in finding a protocol that transactions should follow that will guarantee that they only participate in serializable schedules.
- So, we have a technique for guaranteeing serializability: all transactions must adhere to the 2PL protocol in regard to the order in which they acquire and release locks.

## Is 2PL reducing concurrency in conflict serializable schedule? Contd..

- Note that permitting only serializable schedules is a restriction: there may well be many non-serializable schedules of a transaction-group that do not cause errors; however, we are willing to live with that restriction for assured correctness.
- Also note that using 2PL to achieve serializability is an even further restriction: many serializable schedules of a transaction-group are not achievable using 2PL; again, we are willing to live with that restriction for assured correctness.
- Note that the DBMS can check the lock & unlock requests in a transaction and determine if that transaction obeys 2PL; if not, it can be denied execution.

## UNITY CHART

## CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE:

### Concurrency Control Protocols for Transaction Management

1. The 2PL protocol consists of two distinct phases: the Growing Phase, during which transactions can acquire locks but not release any, and the Shrinking Phase, where transactions can release locks but not acquire new ones.
2. This disciplined approach helps prevent issues such as data inconsistency and ensures that transactions are executed in a well-defined order, contributing to the overall integrity and reliability of the database system.

---

3. Transcendental consciousness is the field of frictionless flow of information.
4. Impulses within the Transcendental Field: Nature accomplishes what it needs by having its impulses in the transcendental field be as efficient as possible.
5. Wholeness moving within itself: In unity consciousness activity is maximally efficient because it originates at the level of the Self, which is the field of all the laws of Nature.