Unit 9

Concurrency Control

Content

- 9.1 Introduction
- 9.2 Locking Technique
- 9.3 Optimistic Concurrency Control

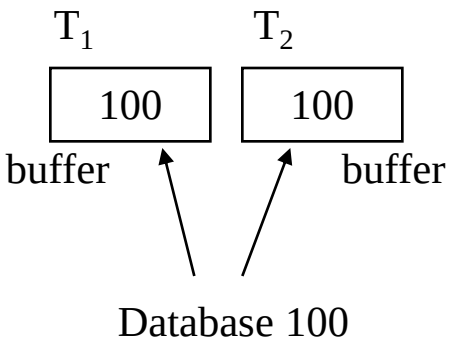
9.1 Introduction

Concurrency Control: Introduction

- **The Problem**
  - In a multiple-user DBMS, how to ensure **concurrent transactions** do not interfere with each other's operation?
- **Why concurrent transaction?**
  - minimize response time
  - maximize throughout
- **Concurrency control techniques**
  - Locking (§9.2)
    - 2PL
    - Tree protocol locking

      ⋮
  - Optimistic method
    - Time stamp ordering (§9.3)

      ⋮



Problem: Lost Update

- **The problem when works without concurrency control**
  1. Lost Update: Fig. 9.1
  2. Uncommitted Dependence: Fig. 9.2

<Example 9.1> Lost Update

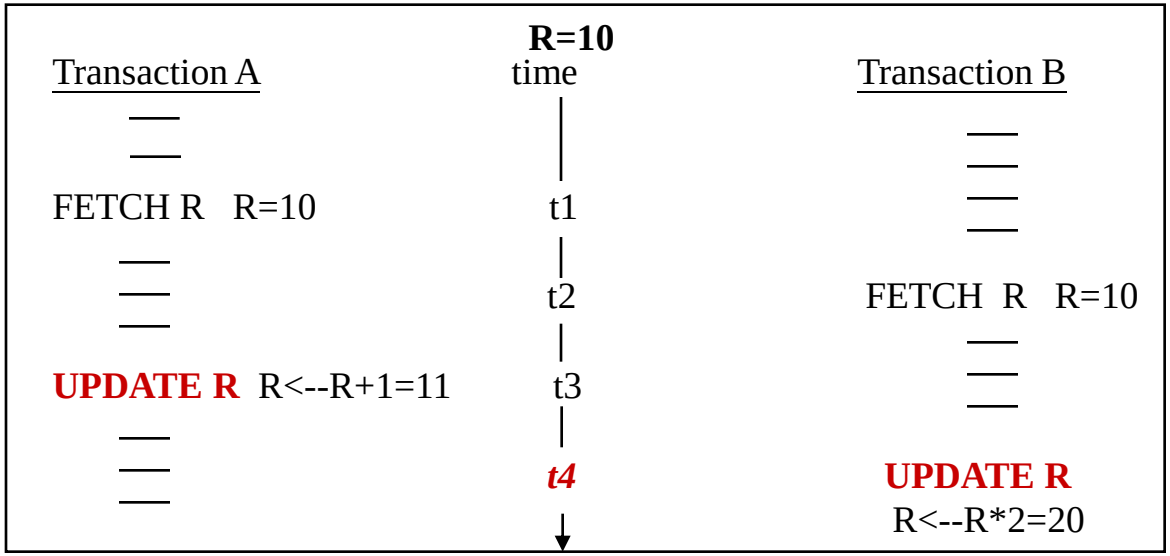


Fig. 9.1: Transaction A loses an update at time t4
if A--> B : R= (10+1) * 2 = 22
if B--> A : R= (10*2) + 1 = 21

Problems: Uncommitted Dependence

<Example 9.2> Uncommitted Dependence

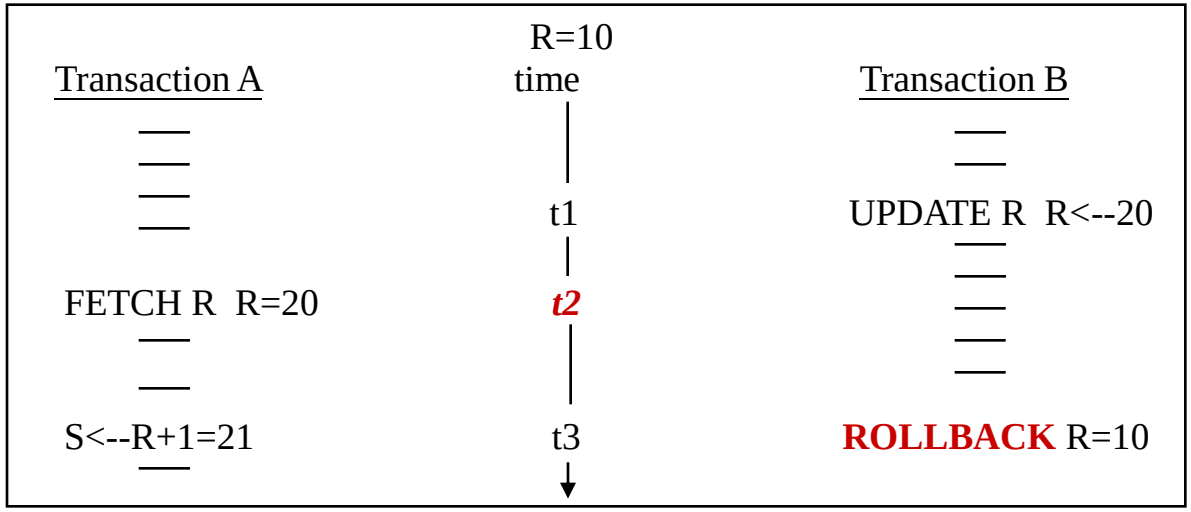


Fig. 9.2: Transaction A becomes dependent on an uncommitted change at time t2.

Introduction: Serializability

- A formal criterion for correctness.
- Def : A given **interleaved execution (schedule)** of some set of transactions is said to be **serializable** iff it produces the same result as some serial execution of those transactions, for any given initial database state.
- Assumptions:
  - all transactions are individually correct.
  - Any serial execution of those transactions is also correct.
  - Transactions are all independent of one another.

Note: if **Tx A** does have to be run before **Tx B**, then user cannot submit **Tx B** until **Tx A** is committed.

Introduction: Serializability (cont.)

- **<Example 9.4>**

  T1 = { r1(d), w1(d) }
  T2 = { r2(a), w2(a) }

  - **Serial execution** : r1(d), w1(d), r2(a), w2(a) T1 → T2
    T1 T2
  - **Interleaved execution** : r1(d), r2(a), w1(d), w2(a)
    (schedule) T1 T2 T1 T2
  - Serializable?

- Serialization: the serial execution that equivalent to the serializable execution, e.g. T1 - T2

Introduction: Serializability (cont.)

<Example 9.5> [Exercise 15.3 p.491]

Given : T1: Add 1 to A

T2: Double A

T3: Display A and set A to 1

Initial value: A = 0

<a> How many possible correct results ? 3! = 6.

T1 - T2 - T3 : A=1

T1 - T3 - T2 : A=2

T2 - T1 - T3 : A=1

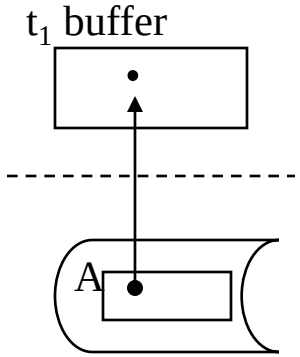
T2 - T3 - T1 : A=2

T3 - T1 - T2 : A=4

T3 - T2 - T1 : A=3

Introduction: Serializability (cont.)

<b> Suppose the internal structures of T1, T2, T3 are :



T1:	T2 :	T3 :
F1: Fetch A into t1	F2 : Fetch A into t2	F3 : Fetch A into t3
t1:= t1+1	t2 := t2*2	display t3
U1: Update A from t1	U2 : Update A from t2	U3 : Update A to 1

How many possible interleaved executions? 90

Fi- Fj- Fk- Up- Uq- Ur: 3*2*1*3*2*1 = 36

Fi- Fj- Up- Fk- Uq- Ur: 3*2*2*1*2*1 = 24

Fi- Fj- Up- Uq- Fk- Ur: 3*2*2*1*1*1 = 12

Fi- Up- Fj- Fk- Uq- Ur: 3*1*2*1*2*1 = 12

Fi- Up- Fj- Uq- Fk- Ur: 3*1*2*1*1*1 = 6

Fi- Up- Uq- Fj- Fk- Ur ?

Introduction: Serializability (cont.)

<c> Is there any interleaved executions **that produces "correct" result but is <u>not serializable</u> ?**

Consider the schedule (△):

△ : F1- F2- F3- U3- U2- U1 = 1 (if A=0) same as T1-T2-T3

But, Consider initial value of A is 10***:***

T1-T2-T3 : 1
T1-T3-T2 : 2
T2-T1-T3 : 1
T2-T3-T1 : 2
T3-T1-T2 : 4
T3-T2-T1 : 3

F1-F2-F3-U3-U2-U1=11 ≠ any serial execution
(10) (10) (10) (1)(20) (11)

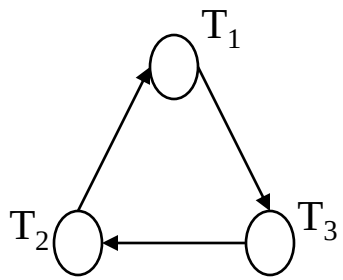
∴ not serializable !

- An **interleaved execution** of some set of transactions is considered to be **correct** iff it is **serializable** !

Introduction: Testing for Serializability

- Two operations are said to be **conflict** if
  <1> they come from different transaction.
  <2> they operate on the same data element.
  <3> at least one of them is write operation.

  <e.g.> T1= { r1(a), w1(b) }
  T2= { r2(b), w2(c) }
  T3= { r3(c), w3(a) }
  - r1(a), w3(a) are conflict
  - r2(b), w1(b) are conflict
  - r3(c), w2(c) are conflict



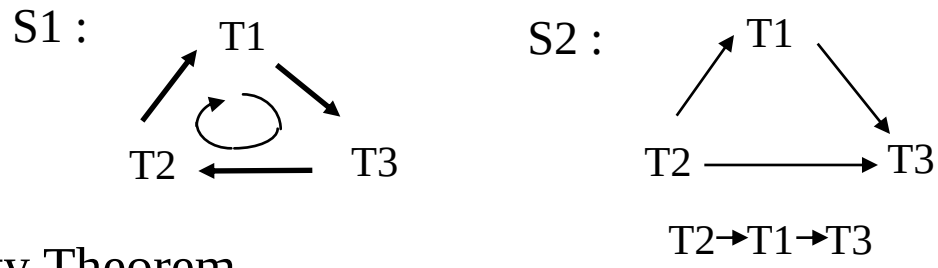
Introduction: Testing for Serializability (cont.)

- Transaction Dependency Graph:
  - Nodes: transactions, e.g. T1, T2, T3
  - Arcs: dependence

    T1 --> T2 if O1 and O2 are conflict, and O1 before O2 in a schedule **S.**

  <e.g.> S1: r1(a), r3(c), r2(b), w1(b), w3(a), w2(c)

  S2: r1(a), r2(b), w1(b), w2(c), r3(c), w3(a)



- The Acyclicity Theorem
  - An interleaved transaction schedule is serialization iff its transaction dependency graph is acyclic.

9.2 Locking Technique

Locking Technique: Concept

- The effect of a lock is to lock other transaction out of the object.
- Two kinds of locks
  - Exclusive lock (X locks): for UPDATE
  - Shared lock (S locks): for RETRIEVE
- Compatibility matrix

A \ B	X	S	—
X	N	N	Y
S	N	Y	Y
—	Y	Y	Y

— : no lock
N : request not compatible
Y : request compatible

Locking Technique: Concept (cont.)

A \ B	X	S	—
X	N	N	Y
S	N	Y	Y
—	Y	Y	Y

— : no lock
N : request not compatible
Y : request compatible

<e.g.> if transaction A holds a S lock on R, then

<1> a request from B for X lock on R

=> B goes into wait state.

<2> a request from B for S lock on R

=> B also hold the S lock on R

Ref: X locks and S locks are normally held until the next synchpoint. (Ref. p.8-12)

How locking solves the problems

- The lost Update Problem

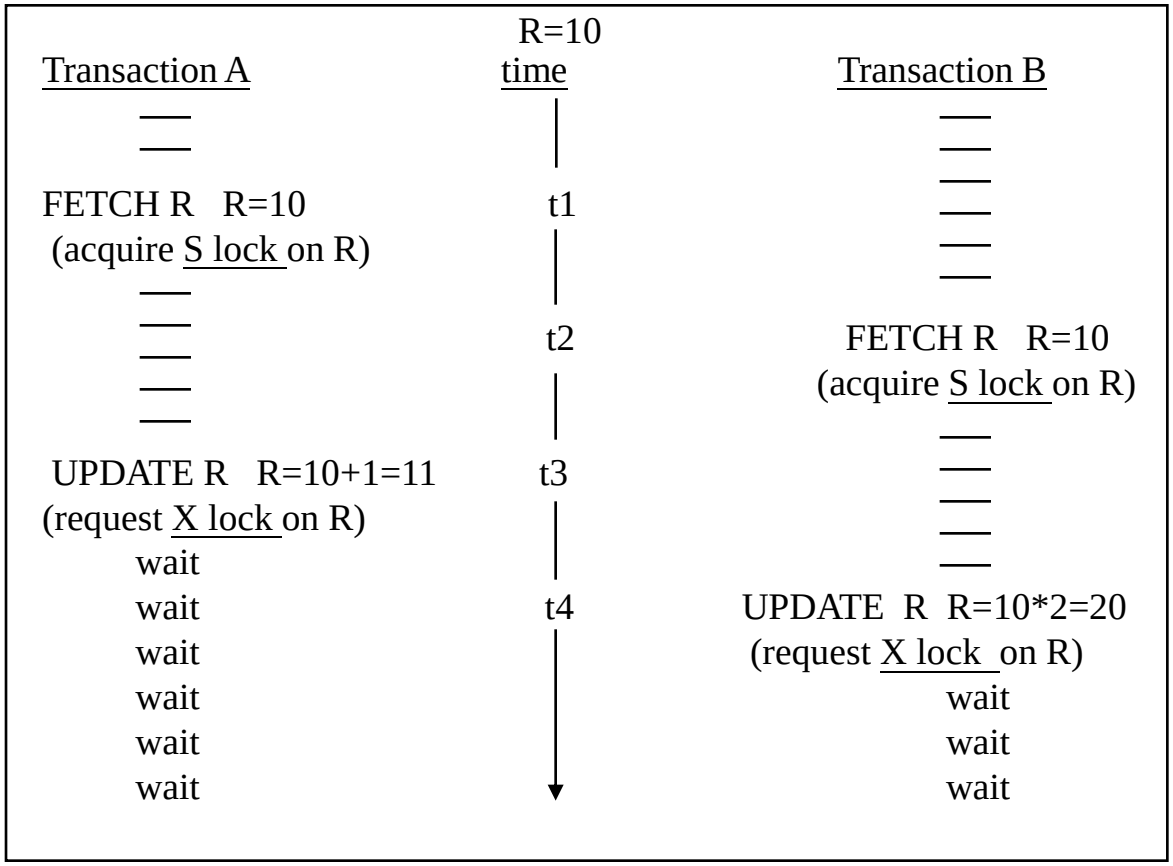


Fig. 9.4: No update is lost, but deadlock occurs at time t4.

How locking solves the problems (cont.)

- The Uncommitted Dependence Problem

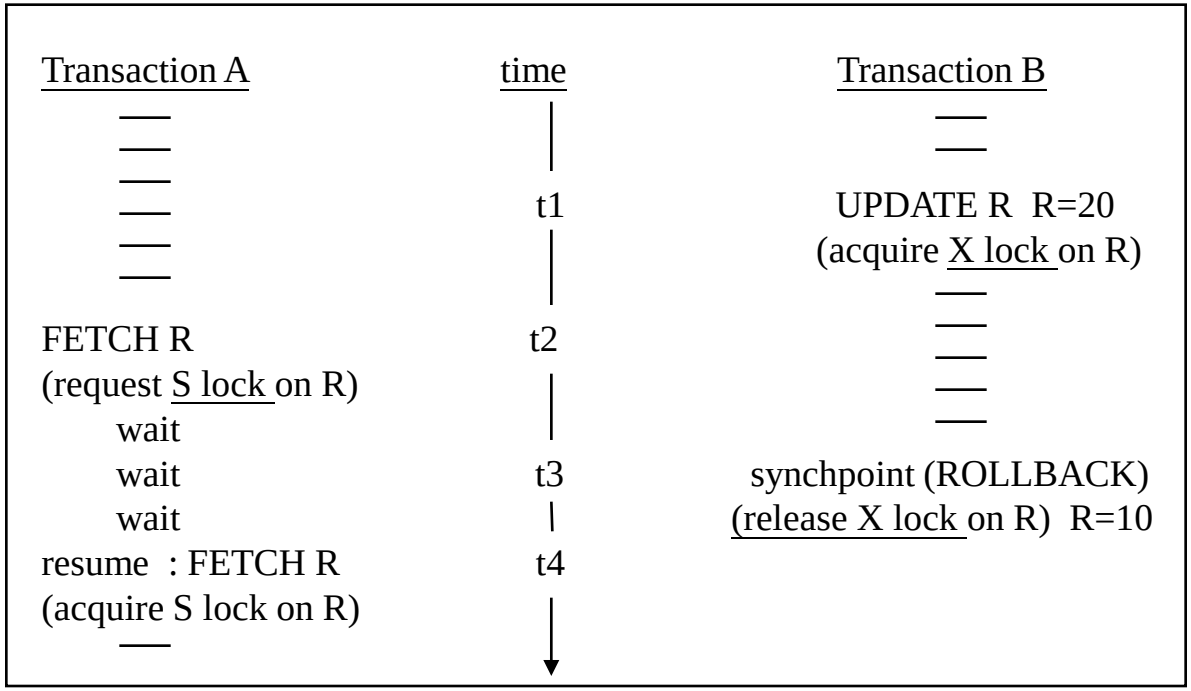


Fig 9.5: Transaction A is prevented from seeing an uncommitted change at time t2.

Deadlock Detection: Wait-for-Graph

- Deadlock Detection: **Wait-for-Graph**
  - **node**: transactions
  - **arc**: an edge from node T_i to T_j means Ti request a lock on an object that is hold by Tj.
  - the system draw an edge from Ti to Tj when the request is issued and erase that edge when Tj release the lock.
  - if there are edges from T1 to T2, T2 to T3, ..., Tn-1 to Tn, and Tn to T1 ==> T1, T2, ..., Tn are deadlocked.

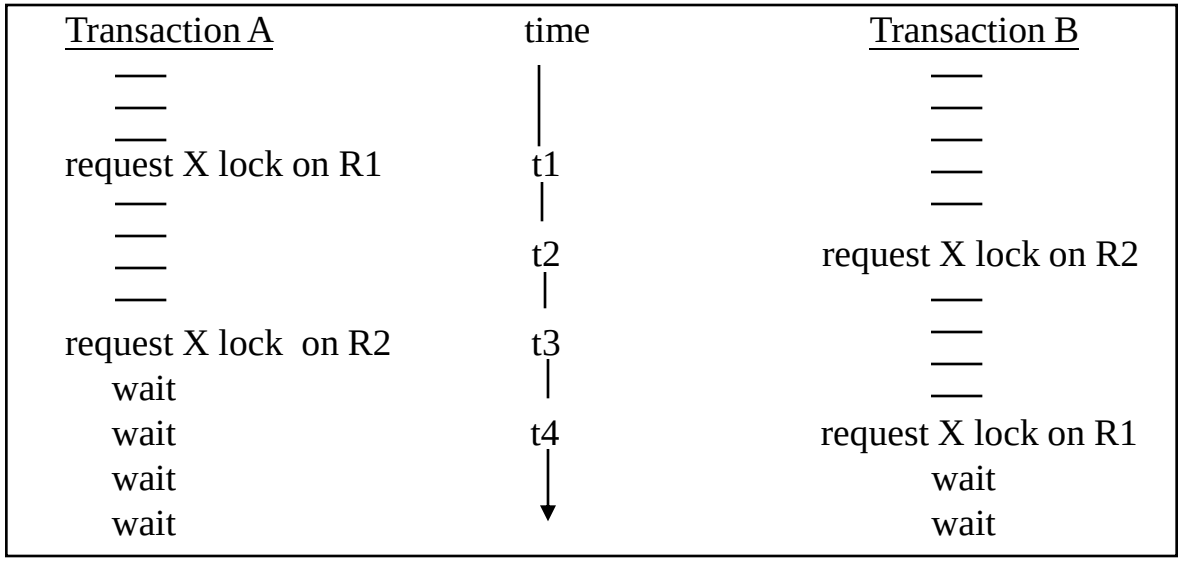


Fig. 9.7:
An example of deadlock.

Locking Protocol

- **Locking Protocol**: to ensure the Serializability
  1. Two-phase locking (2PL)
  2. Non-two-phase locking (skip)
     - tree protocol locking
     - directed acyclic graph protocol

Testing for Serializability

- **Testing for Serializability in Locking Protocol**
  - **Precedence Graph:**
    - **Node**: transactions
    - **Arc**: an **arc** from Ti to Tj ($Ti \longrightarrow Tj$)
      if $Oi \in Ti$, $Oj \in Tj$ and
      <1> Oi and Oj operates on the same data.
      <2> Oi is UNLOCK, Oj is LOCK.
      <3> Oi precedes Oj.
    - e.g. T1, T2

$T_1 \rightarrow T_2$	$T_2 \rightarrow T_1$
T_1 UNLOCK(d)	T_2 UNLOCK(d)
T_2 LOCK(d)	T_1 LOCK(d)

Theorem for Testing Serializability

- **Thm 9.1**: If the precedence graph of a schedule contains **no cycle** then schedule is **serializable**.

<Example 9.6>

Consider the following three transactions:

T_1: Lock A	T_2: Lock B	T_3:
Lock B	Lock C	
UNLock A	UNLock B	
UNLock B	Lock A	
	UNLock C	
	UNLock A	

- A schedule for T1, T2, and T3:

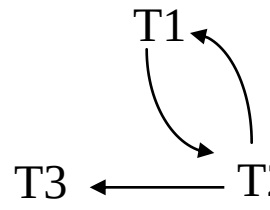
(1) T1 : LOCK A
(2) T2 : LOCK B
(3) T2 : LOCK C
(4) T2 : UNLOCK B
(5) T1 : LOCK B
(6) T1 : UNLOCK A
(7) T2 : LOCK A
(8) T2 : UNLOCK C
(9) T2 : UNLOCK A
(10) T3 : LOCK A
(11) T3 : LOCK C
(12) T1 : UNLOCK B
(13) T3 : UNLOCK C
(14) T3 : UNLOCK A

From (4), (5) we have T2 → T1

From (6), (7) we have T1 → T2

From (8), (11) we have T2 → T3

- The precedence graph of the schedule is:



not serializable !

Two-Phase Locking (2PL)

- A protocol that guarantees **serializability**.
- A transaction obeying the **two-phase locking protocol (2PL)** if

  <a> before operating on any object, the transaction first acquires a lock on that object (the locking phase)

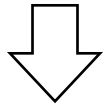
  <b> after releasing a lock, the transaction never acquires any more lock (the unlocking phase)

  i.e. in any transaction, all locks must precede all unlock.

Two-Phase Locking (2PL) (cont.)

<e.g.>

T1:	T2:	T3:
LOCK A	LOCK A	LOCK B
LOCK B	LOCK C	LOCK C
UNLOCK A	UNLOCK C	UNLOCK B
UNLOCK B	UNLOCK A	LOCK A
		UNLOCK C
		UNLOCK A



T1 obey 2PL
T2 obey 2PL
T3 not obey 2PL

Note: In practice, a lock releasing phase is often compressed into the single operation of COMMIT (or ROLLBACK) at end-of-transaction.

Two-Phase Locking (2PL) (cont.)

- **Thm 9.2:** If all transactions obey the "2PL" protocol, then all possible interleaved schedules are serializable.

  [proof]: [by contradiction] (P. 9-22)

  Suppose not. Then by Thm 9.1, the **precedence Graph G** for **S** has a cycle, say

  $T_{i1} \Rightarrow T_{i2} \Rightarrow \ldots \Rightarrow T_{ip} \Rightarrow T_{i1}$.

  eg. T1 → T2 → T3 → T1

  Then some LOCK by T_{i2} follows an UNLOCK by T_{i1};

  some LOCK by T_{i3} follows an UNLOCK by T_{i2};

  ⋮

  some LOCK by T_{i1} follows an UNLOCK by T_{ip};

  => A LOCK by T_{i1} follows an UNLOCK by T_{i1}

  => T_{i1} disobey of 2L protocol !! **Q.E.D. #**

UNLOCK LOCK

Ti1 → Ti2

Ti2 → Ti3

…

…

Tip → Ti1

=>

One lock of Ti1 is after unlock!!

<e.g.1> T1, T2 any interleaved schedules are serializable.

<e.g.2> T1, T2 , T3 perhaps

Two-Phase Locking (2PL) (cont.)

<Example 9.7> [Exercise 15.3 (d)] (p.9-6, or p.491)

<d> Is there any interleaved execution of T1, T2, T3 that is serializable but could not be produced if all three transactions obeyed the 2PL?

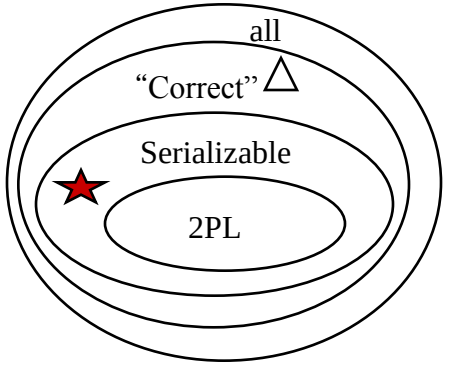
T1 :	**T2 :**	**T3** :
F1 : Fetch A into t1	F2 : Fetch A into t2	F3 : Fetch A into t3
t1 := t1+1	t2 := t2*2	display t3
U1 : Update A from t1	U2 : Update A from t2	U3 : Update A from t1

Two-Phase Locking (2PL) (cont.)

★ **Yes ! F1 - F3 - U1 - U3 - F2 - U2 = T1-T3-T2 is serializable**

SLOCK SLOCK wait until T3 terminates

∵ Operation U1 will not be able to proceed until that SLOCK by F3 has been released, and that will not happen until T3 terminates.



In fact, transaction T3 and T1 will deadlock when U3 is reached.

△ Ref. p.9-11

<Note> 2PL ⊆ Serializable ⊂ "Correct" ⊆ All interleaved schedules

9.3 Optimistic Concurrency Control

Optimistic Concurrency Control

- Motivation
  - Provided that the possibility of conflict is small, it is inefficient to lock each data item before using it.

    => Allow reading and writing as we wish,

    if the serializability is violated => abort !

    Optimistic Concurrency Control
  - Deadlock free !
- How to decide whether the serializability is violated ?
  - Timestamp Ordering
  - ...

Optimistic Concurrency Control (cont.)

- **Timestamp Ordering**
  - Timestamp:
    - a number of generate by system.
    - ticks of the computer's internal clock.
    - no two transactions can have the same timestamp.
    - 24 bits is large enough to hold a timestamp.
      (repeat only every half year)
  - How timestamps are used ?

Timestamp Ordering

- **How timestamps are used ?**

  <i> Each transaction, **T**, is assigned a timestamp, **t,** say **t(T)**

  <ii> Each data item, **d**, is assigned two timestamps:

  (1) Read time, **tr(d)**, the highest transaction timestamp that have read the item.

  (2) Write time, **tw(d)**, the highest transaction timestamp that have written the item.

  <iii> When **reading** an item, **d**, by **T**:

  if **t(T)** > **tw(d)**

  then (1) execute read operation

  (2) **tr(d)** = max {**tr(d)**, **t(T)**}

  else abort.

  <iv> When **writing** an item, **d**, by **T**:

  if **t(T)** ≥ **tr(d)** and **t(T)** ≥ **tw(d)**

  then (1) execute the write operation.

  (2) **tw(d)** = max {**tw(d)**, **t(T)**}

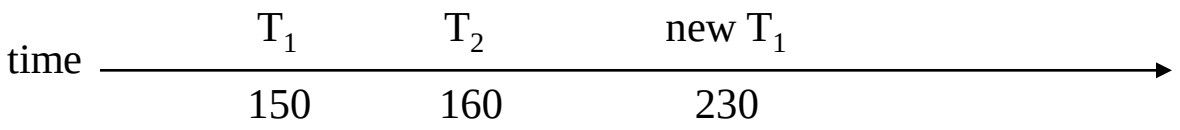
  else abort

Timestamp (cont.)

<Example 9.8>

	T1, t(T1)=150	T2, t(T2)=160	Data A tr(A)	Data A tw(A)	why ?
initial			0	0	
(1)	READ A		150	0	t(T1)=150 > tw(A)=0
(2)		READ A	160	0	t(T2)=160 > tw(A)=0
(3)	A:=A+1				
(4)		A:=A*2			t(T2)=160>=tr(A)=160
(5)		WRITE A	160	160	t(T2)=160>=tw(A)=0, OK!
(6)	WRITE A				t(T1)=150 < tr(A), Abort!

Locking?



Timestamp (cont.)

<Example 9.9>

	T1	T2	T3	Data A	Data B	Data C
	t(T1)=200	t(T2)=150	t(T3)=175	tr(A)=0	tr(B)=0	tr(C)=0
				tw(A)=0	tw(B)=0	tw(C)=0
(1)	READ B				tr(B)=200	
(2)		READ A		tr(A)=150		
(3)	WRITE B				tw(B)=200	
(4)		WRITE C				tw(C)=150
(5)			READ C			tr(C)=175
(6)	WRITE A			tw(A)=200		
(7)			WRITE A	175<200		
				T3 abort !		

Timestamp (cont.)

<Example 9.10> Indefinite Repetition

	T1	T2	T1	T2	A	B
	t(T1)=100	t(T2)=110	t(T1)=120	t(T1)=120	tr(A)=0	tr(B)=0
					tw(A)=0	tw(B)=0
(1)	WRITE B					tw(B)=100
(2)		WRITE A			tw(A)=110	
(3)	READ A				100<110 ==> T1 abort !	
(4)			WRITE B			tw(B)=120
(5)		READ B				110<120, T2 abort !
(6)				WRITE A	tw(A)=130	
(7)			READ A		120<130 ==> T1 abort again !	