Department of Computer Science
University of Cyprus

## EPL646 – Advanced Topics in Databases

# Lecture 9

## Concurrency Control

## (with Locking, with Timestamps)

**Chapter 18.1: Elmasri & Navathe, 5ED**

**Chapter 17.1-17.4: Ramakrishnan & Gehrke, 3ED**

**Demetris Zeinalipour**

http://www.cs.ucy.ac.cy/~dzeina/courses/epl646

# Intro to DBMS Concurrency Control (Εισαγωγή σε Έλεγχο Ταυτοχρονίας ΣΔΒΔ)

- In the previous lecture we **characterized** transaction schedules based on **serializability** and **recoverability**
  - **Serializability =>** Guarantee the **correctness** of a schedule.
  - **Recoverability** => Guarantee that **commits** are **durable**.
- We shall now study how the DBMS **enforces (επιβάλει)** these types of **schedules. In particular, we shall see:**

- **Concurrency Control with Locking**
  - **Locking (Κλείδωμα):** Use locks to prevent multiple transactions from accessing the items concurrently.
- **Concurrency Control without Locking**
  - **Timestamp Ordering (Διάταξη Χρονόσημων) :** Ensure serializability using the ordering of timestamps generated by the DBMS.
  - **Multiversion CC:** Use multiple version of items to enforce serializability.
  - **Optimistic CC:** No checking done during execution of a Transaction but post-execution validation (επικύρωση) enforces serializability.

# Concurrency Control with Locking (Έλεγχος Ταυτοχρονίας με Κλείδωμα)

- **What is a Lock (Κλειδαριά)**
  - A **variable** associated with a **data item** that describes the **status** of the item with respect to **possible operations** that can be applied to it (e.g., read lock, write lock).
  - Locks are used as **means of synchronizing** the **access** by concurrent transactions to the **database items**.
  - There is one lock per database item.
- The portion of the database a **data item** represents is referred to as **Locking Granularity (Κλιμάκωση Κλειδώματος)**
  - Single attribute (field), Disk Block, Whole File or even whole DB!
- **Problems arising from Locks? (studied next)**
  - **Deadlock (Αδιέξοδο):** Two more competing transactions are waiting for each other to complete to obtain a missing lock.
  - **Starvation (Λιμοκτονία):** A transaction is continually denied access to a given data item.

# Concurrency Control with Locking (Έλεγχος Ταυτοχρονίας με Κλείδωμα)

- **Purpose of CC with Locking**
  - To enforce (επιβάλει) **Isolation (Απομόνωση)** through mutual exclusion among **conflicting transactions**.
    - ***Mutual Exclusion (Αμοιβαία Απόκλιση)****: avoid the simultaneous use of a common resource by pieces of code called critical sections that utilize locks.*
  - To preserve (διατηρήσει) database **Consistency (Συνέπεια)** through consistency preserving execution of transactions.
  - To resolve **RW, WR and WW** conflicts.
- **Example Scenario:**
  - In concurrent execution environment if **T1** conflicts with **T2** over a data item **A**,
  - Then the **existing** CC decides which of **T1** or **T2** should get access to A and which of T1 or T2 should be rolled-back or wait.

# Locks and Types of Locks
## (Κλειδαριές και Τύποι Κλειδαριών)

- **Locking (Κλείδωμα)** is an operation that secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.

***Example:***

- ***Lock (X).*** *Data item X is locked on behalf of the requesting transaction.*

- **Unlocking (Ξεκλείδωμα)** is an operation that removes these permissions from the data item.

**Example**:

- **Unlock (X):** Data item X is made available to all other transactions.

- **Lock** and **Unlock** are atomic operations (all-or-nothing).
  - e.g., if we lock a table, either the complete table is locked or no tuple in the table is locked.
- **Locking Protocols** are the defacto mechanism to enforce serializability in DBMSs. They usually operate in 2 **phases**:
  - Phase 1: Locking, Phase 2: Unlocking. 2PL (Two Phase Locking), that we will see next, is one such protocol.

# Types of Locks
# (Τύποι Κλειδαριών)

- There are two types of Locks: **Binary Locks** and **Shared/Exclusive** (or **Read/Write Locks)**.
- **Binary Locks (Δυαδικές Κλειδαριές)**
  - A binary lock has two **states**: **Locked** / **Unlocked**
  - If a DB item **X is locked**, then **X cannot** be accessed by any other **DB operation.**
  - Simple Model but two **restrictive (περιοριστικό)** as only a **single transaction** can hold a **lock** on given item.
  - Consequently, we will not use these locks…
- **Shared/Exclusive** (or **Read/Write Locks)**.
  - In Greek: **Μεριζόμενο/Αποκλειστικό** Κλείδωμα
  - Three **states**: **Read Locked** / **Write Locked** / **Unlocked**
  - Several Xacts can access the same item X for reading (Shared Lock), however writing requires a Write (Exclusive) Lock.

# Implementing Locks in a DBMS (Υλοποίηση Κλειδαριών στη ΣΔΒΔ)

## How is Locking Implemented Inside a DBMS?

- **Lock Manager (Διαχειριστής Κλειδωμάτων):**
  - Managing locks on data items using a Lock Table.
- **Lock table (Πίνακας Κλειδώσεων):**
  - Lock manager uses it to store:
    - **Transaction ID:** Identity of transaction locking a data item.
    - **The data item ID:** E.g., DBName, TableID, PageID or RID
    - **Lock mode:** Read-Locked, Write-Locked, Unlocked
    - **Pointer to Lock Queue:** Such that next in queue can be notified

| Transaction ID | Data item id | lock mode | Pointer to Lockqueue |
|---|---|---|---|
| T1 | X1 | Read | 0x00AF04BF |

# Conversion of Locks
## (Μετατροπή Κλειδωμάτων)

- **Conversion of Locks (Μετατροπή Κλειδώματος):** When a transaction that already holds a lock on a data item X is allowed to change its lock status as follows:
  - **Lock Upgrade (Αναβάθμιση Κλειδώματος):**

**if** $T_i$ is the ONLY Xact that has a Read-lock(X) **then**
**Convert Read-lock(X)** to **Write-lock(X)**
**else** // Other Xact needs to release X first
**Force $T_i$ to wait**
(Wait until all the Xact that holds the lock unlocks X)

  - **Lock Downgrade (Υποβάθμιση Κλειδώματος):**

if **$T_i$** has a **Write-lock(X)**
**// Note that** no other Xact can have any lock on X
Convert **Write-lock(X)** to **Read-lock(X)**

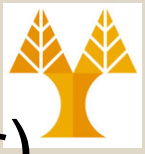
# CC with Locking: Techniques

## (Έλεγχος Ταυτοχρονίας με Κλείδωμα: Τεχνικές)

- **Do locks guarantee serializability (correctness) of schedules on their own? NO**
- **Let us consider the following example:** Schedules T1;T2 and T2;T1 generate the expected correct results.
- Now consider the following correct execution T1;T2 or T2;T1. We shall see in the next slide that by using a wrong sequence of locks might generate a wrong result.

| | **T1** | **T1;T2** | **T2** | **Result** |
|---|---|---|---|---|
| 1)Y=30 | read_lock (Y); | 4) X=50 | read_lock (X); | **Initial values: X=20; Y=30** |
| | read_item (Y); | | read_item (X); | **Result of serial execution A (correct)** |
| | unlock (Y); | | unlock (X); | T1 followed by T2 **(T1; T2)** |
| 2) X=20 | write_lock (X); | 5) Y=30 | write_lock (Y); | **X=50, Y=80.** |
| 3) X=50 | read_item (X); | 6) Y=80 | read_item (Y); | |
| | X:=X+Y; | | Y:=X+Y; | **Result of serial execution B (correct)** |
| | write_item (X); | | write_item (Y); | T2 followed by T1 **(T2; T1)** |
| | unlock (X); | | | **X=70, Y=50** |

# CC with Locking: Techniques
## (Έλεγχος Ταυτοχρονίας με Κλείδωμα: Τεχνικές)



- **Why Locking is not adequate in its own to guarantee serializability?**

| | T1 | T2 |
|---|---|---|
| | **read_lock (Y);** | |
| Y=30 | read_item (Y); | |
| | **unlock (Y);** | |
| | | **read_lock (X);** |
| | | read_item (X); X=20 |
| | | **unlock (X);** |
| | | **write_lock (Y);** |
| | | read_item (Y); Y=30 |
| | | Y:=X+Y; Y=20+30=50 |
| | | write_item (Y); |
| | | **unlock (Y);** |
| y=30!!! | **write_lock (X);** | |
| X=20 | read_item (X); | |
| X=20+30=50 | X:=X+Y; | |
| | write_item (X); | |
| | **unlock (X);** | |

Time

**Initial values: X=20; Y=30**

Serial Result T1;T2: **X=50, Y=80**

Serial Result T2;T1: **X=70, Y=50**

**Result of Schedule on Left**

**X=50; Y=50**

Since result not equal to any serial schedule the left schedule is **Nonserializable**.

The above example demonstrates that locks **can not** be obtained in an **arbitrary manner**!

See next slide for solutions….

# Two-Phase Locking (2PL)
# (Πρωτόκολλο Κλειδώματος Δυο Φάσεων)

- **Two-Phase Locking Protocol (2PL) (Πρωτόκολλο Κλειδώματος Δυο Φάσεων):** A Concurrency Control locking protocol that guarantees **serializability**.
- The protocol operates in the following phases:
  1. **Locking (Growing) Phase**: **locks are acquired** and **NO locks** are **released**.
  2. **Unlocking (Shrinking) Phase**: **locks are released** and **NO locks** are **acquired**.
- **Requirements:**
  - For a transaction these two phases must be **mutually exclusively – αμοιβαία αποκλειόμενα (i.e., no locking during Unlocking and vice-versa)**
  - If **lock conversion** is allowed, then upgrading of locks must happen during the **Growing Phase** and **downgrading** of locks during the **Shrinking Phase**.
- ***Theorem:*** *If every transaction in a schedule follows the two-phase locking protocol, the schedule is **guaranteed** to be **serializable**.*

# Two-Phase Locking Variants
## (Παραλλαγές Πρωτοκόλλου Κλειδώματος Δυο Φάσεων)

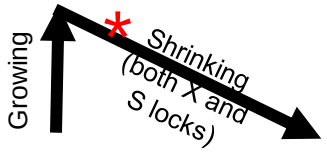
We will study four (4) variants of two-phase locking:

### 1. Conservative or Static 2PL (Συντηρητικό 2PL):

- **Action:** Locks ALL desired data items before transaction begins execution.
  - **Deadlock-free**, obtain all locks or none.
  - **Guarantees Serializability** (recall the 2PL Theorem)
  - **Not-Practical,** as read and write sets need to be pre-declared)
  - **Not Strict Schedule for Recoverability,** as some transaction can read or write an **item that is written by T, before T has committed.**

T | T'
X(A)
W(A)
Rel(A) *
X(A)
**W(A)**
**Abort**
**Commit**
**(Recoverable but not Strict)**

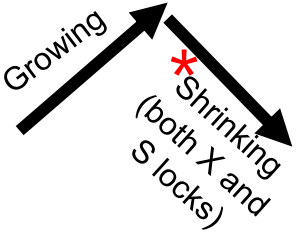
**Conservative 2PL**




### 2. Basic (Βασικό 2PL):

- **Action:** Locks data items incrementally. Unlocks data items incrementally w/out obtaining new locks.
  - **NOT Deadlock-free (see example in 1 slide)**
  - **Guarantees Serializability** (recall the 2PL Theorem)
  - **Practical (but not as practical as Strict 2PL)**
  - **Not Strict Schedule for Recoverability,** as some transaction can **read** or **write** an item that is **written by T, before T has committed**.

**Basic 2PL**

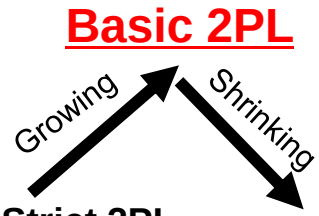


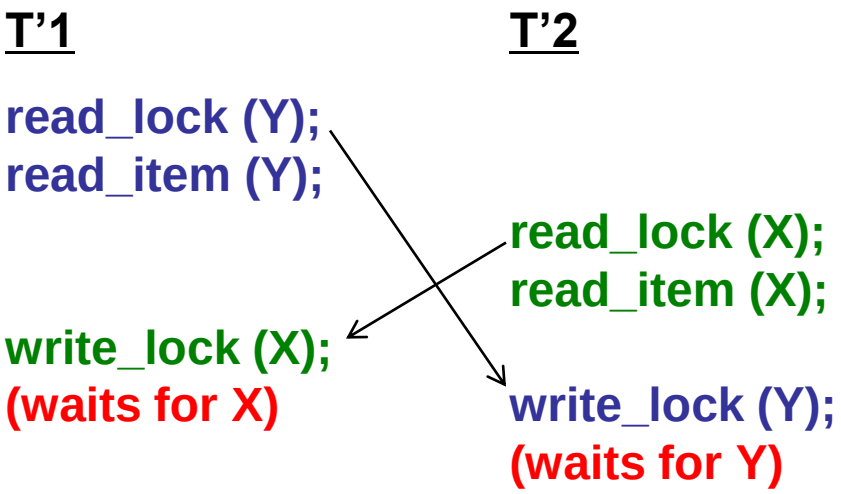
# Deadlocks in Locking
## (Αδιέξοδα στο Κλειδώματα)

- Although **Basic** 2PL guarantees serializability (correctness) by incrementally acquiring the locks and by releasing them at the end,

  **Basic 2PL suffers from Deadlocks!**

  – **The fact that it does not offer recoverability will be addressed next with Strict 2PL**




**Deadlock (Αδιέξοδο):**
Cycle of transactions waiting for locks to be released by each other.

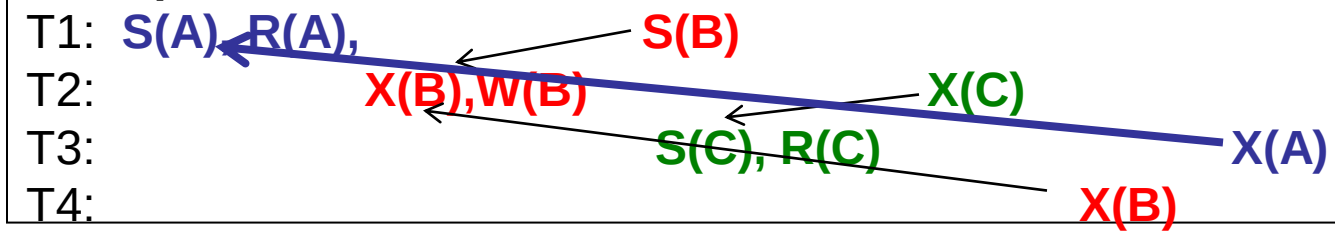
Let's see how to overcome deadlocks…

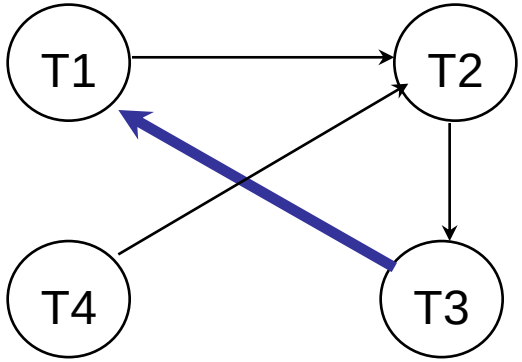
# Dealing with Deadlocks in Locking (Διαχείριση Αδιέξοδων στο Κλείδωμα)

## 1. Deadlock Detection (Ανίχνευση Αδιεξόδου)

- In this approach, **deadlocks are allowed to happen**
- The scheduler maintains a **wait-for-graph** (add edges when waiting, drop edges when obtaining the expected lock)
- If a **cycle exists**, then one transaction involved in the cycle is **selected (victim)** and **rolled-back (need to carefully select victim)**

**Example of Deadlock Detection:**




**Cycle Found! ➔Deadlock (DFS search in O(V+E))**

# Dealing with Deadlocks in Locking
## (Διαχείριση Αδιέξοδων στο Κλείδωμα)

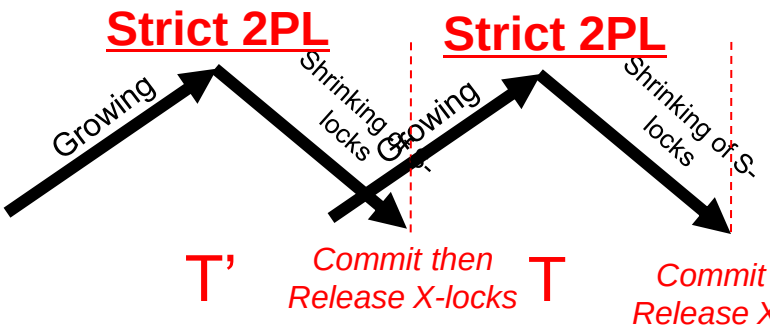
- **Starvation (Λιμοκτονία)**
  - Occurs when a particular transaction **consistently waits** or **restarted** and **never gets a chance to proceed further.**
    - e.g., in previous example it is possible that the **same transaction** may **consistently** be selected as **victim** and **rolled-back.**
  - This limitation is inherent in **all priority based scheduling mechanisms**
    - i.e., if we favor transactions with given characteristics, other transactions might always starve.

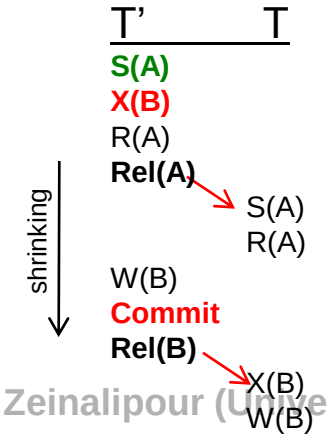
# Two-Phase Locking Variants
## (Παραλλαγές Πρωτοκόλλου Κλειδώματος Δυο Φάσεων)

### 3. Strict (Αυστηρό 2PL):

- Stricter than the **Basic** algorithm (i.e., **exclusive locks X-locks** are only released after commit or abort, no xact can utilize them in its growing phase). **Shared-locks (S-locks)** are released earlier.
- **Characteristics:**
  - **Not Deadlock-free**, **Guarantees Serializability (2PL Theorem),**
  - **Practical,** in fact this is the most commonly used two-phase locking algorithm, **Leads to Strict Schedule for Recoverability**, as no other transaction can **read** or **write** an item that is **written by T, unless T has committed**.

- Note that a **Strict Schedule != Serial Schedule** as a **Strict Schedule** allows interleaving of non-conflicting actions.



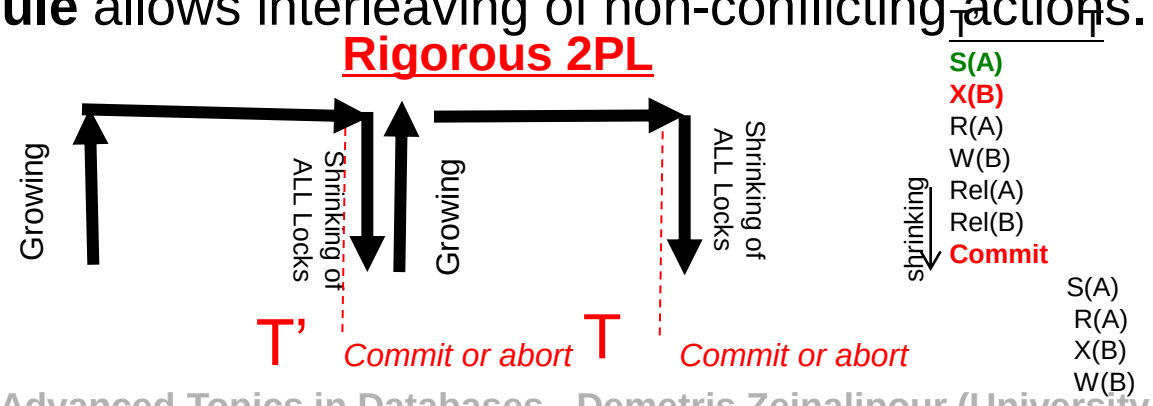




**Example:**
Releasing Shared Lock earlier gives space for higher concurrency

# Two-Phase Locking Variants

(Παραλλαγές Πρωτοκόλλου Κλειδώματος Δυο Φάσεων)

4. **Rigorous (Ακριβής 2PL):**
   - Even more restrictive than **Strict 2PL** (i.e., **exclusive and shared locks** are only released after commit or abort), i.e., **Serialization Order == Commit Order**
     - **Deadlock-free (no Xact holds S-lock on item we might need)**,
     - **Guarantees Serializability,**
     - **Not Practical, but used in many DBMS!**
     - **Leads to Strict Schedule for Recoverability**, as no other transaction can **read** or **write** an item that is **written by T unless T has committed**.
   - Note that a **Rigorous Schedule != Serial Schedule** as a **Rigorous Schedule** allows interleaving of non-conflicting actions.

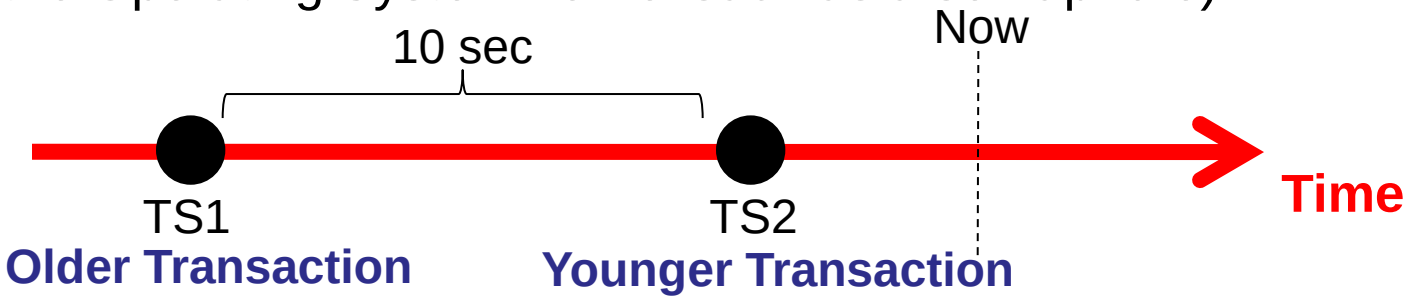


# Concurrency Control in DBMSs
## (Έλεγχος Ταυτοχρονίας σε ΣΔΒΔ)

- We will now see another class of protocols based on **Timestamps**.
- **Concurrency Control with Timestamps (without Locking)**
  - **Timestamp Ordering (Έλεγχος Ταυτοχρονισμού με Διάταξη Χρονόσημων):** Ensure serializability using the ordering of timestamps generated by the DBMS.
  - **Multiversion CC (Έλεγχος Ταυτοχρονισμού Με Πολλαπλές Εκδόσεις):** Use multiple version of items to enforce serializability.
  - **Optimistic CC (Αισιόδοξος (Οπτιμιστικός) Έλεγχος Ταυτοχρονισμού):** No checking done during execution of a Transaction but post-execution validation (επικύρωση) enforces serializability.

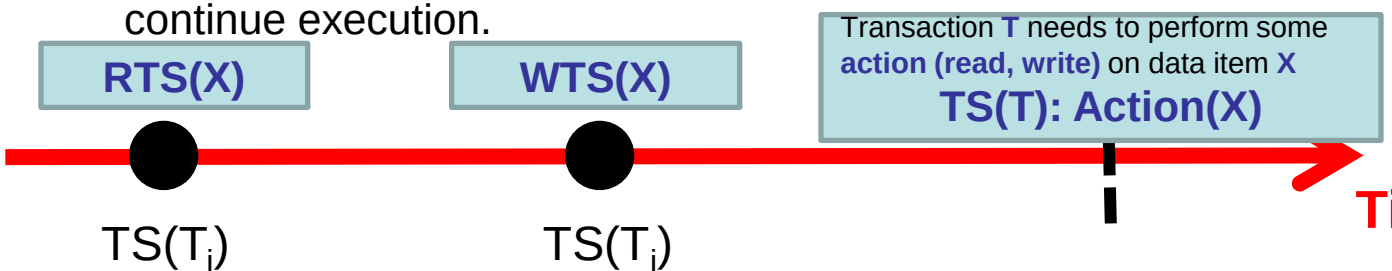
# Timestamp based CC: Definitions
## (Έλεγχος Ταυτοχρονίας με Χρονόσημα: Ορισμοί)

- **Timestamp (Χρονόσημο)**
  - A **monotonically** increasing **variable (integer)** indicating the **age** of an **operation** or a **transaction**.
- A **larger timestamp** indicates a more recent transaction
  - Timestamps are assigned in our context during Xact creation.
  - **Using date timestamps** (e.g., a long integer that represents the number of seconds that have elapsed from 1/1/1970)
    - **TS1: 1237917600 (2009-03-24 18:00:00)**
    - **TS2: 1237917610 (2009-03-24 18:00:10)**
  - **Using a counter timestamp** (e.g., using a counter stored inside the Operating System kernel such as a semaphore)



# Timestamp based CC: Definitions
## (Έλεγχος Ταυτοχρονίας με Χρονόσημα: Ορισμοί)

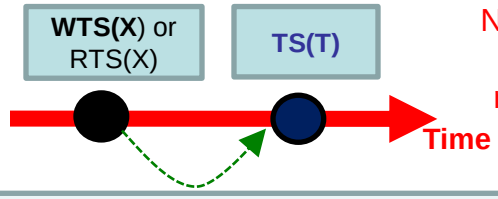
- Assume a collection of **data items** that are accessed, with read and write operations, by transactions.
- **For each data item X the DBMS maintains the following values:**
  - **RTS(X):** The Timestamp on which object **X** was **last read** (by some transaction $T_i$, i.e., RTS(X):=TS($T_i$))
  - **WTS(X):** The Timestamp on which object **X** was **last written** (by some transaction $T_j$, i.e., WTS(X):=TS($T_j$))
- **For the following algorithms we use the following assumptions:**
  - A data item **X** in the database has a **RTS(X)** and **WTS(X)** (recorded when the object was last accessed for the given action)
  - A transaction **T** attempts to perform some action (read or write) on data item **X** on timestamp **TS(T)**
    - **Problem:** We need to decide whether T has to be aborted or whether T can continue execution.



# Basic Timestamp Ordering Algorithm (Βασικός Αλγόριθμος Διάταξης Χρονοσήμων)

- We shall now present the first algorithm, coined **Basic Timestamp Ordering (TO),** that utilizes Timestamps to guarantee **serializability** of concurrent transactions.

- **Timestamp Ordering (TO) Rule**
  - if $\mathbf{p_a(x)}$ and $\mathbf{q_b(x)}$ are conflicting operations, of xacts $\mathbf{T_a}$ and $\mathbf{T_b}$ for item **x**, then $\mathbf{p_a(x)}$ is processed before $\mathbf{q_b(x)}$ iff (🡘🡚) $\mathbf{ts(T_a) < ts(T_b)}$
  - **Main Idea:** Conflicts are only allowed from **older transactions** (with smaller ts) to a **younger transaction T** (with larger ts)
  - **Main Idea Example:**


Note that the conflict moves only to the right not to the left!

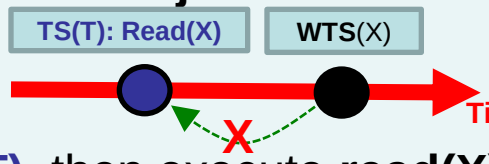
- **Theorem: If the TO rule is enforced in a schedule then the schedule is (conflict) serializable.**
  - **Why?** Because **cycles** are **not possible** in the **Conflict Precedence** Graph (Γράφος Προτεραιότητας Συγκρούσεων)!

# Basic Timestamp Ordering Algorithm
## (Βασικός Αλγόριθμος Διάταξης Χρονοσήμων)

- **Basic Timestamp Ordering (TO) Algorithm**

**Case 1 (Read X by T): Transaction T issues a read(X) operation**

A. If **TS(T) < WTS(X),** then read(X) is rejected (as the TO rule is violated).
T has to **abort** and be **rejected**.

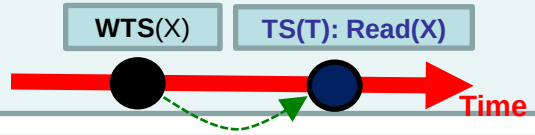



**Reject T (TO rule violated)**

Happens if T started earlier than T' (that wrote X) … see example on slide 16.9

B. If **WTS(X) ≤ TS(T)**, then execute **read(X)** of T and update **RTS(X)**.

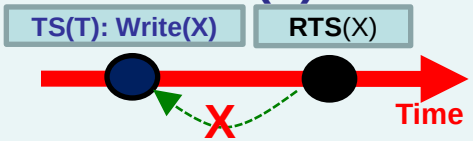
*R/R not conflicting action so RTS(X) ? TS(T) not investigated*

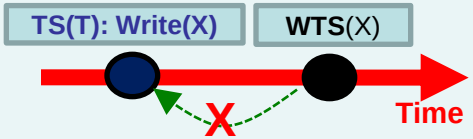



**Accept T (TO ok)**

**Case 2 (Write X by T): Transaction T issues a write(X) operation**

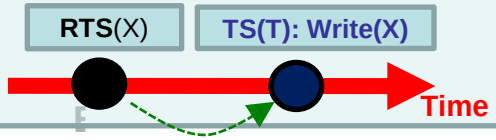
A. If **TS(T) < RTS(X)** or if **TS(T) < WTS(X)**, then write is rejected.

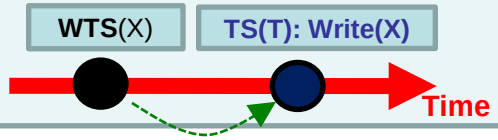



**Reject T (TO rule violated)**

B. If **RTS(X) ≤ TS(T)** or **WTS(X) ≤ TS(T)**, then execute **write(X)** of T and update **WTS(X)**




**Accept T (TO ok)**

# Basic TO Algorithm Example
## (Παράδειγμα Βασικός Αλγόριθμος Διάταξης Χρονοσήμων)

- Consider the following scenario:
  - Two transactions **T1** and **T2**
  - Initially **RTS=0** and **WTS=0** for data items **X, Y**
  - Timestamps are as follows: **TS(T1)=10** and **TS(T2)=20**

| T1(10) | T2(20) |
|---|---|
| **1. A1 = Read(X)** | **1. A1 = Read(X)** |
| **2. A1 = A1 – k** | **2. A1 = A1 * 1.01** |
| **3. Write(X, A1)** | **3. Write(X, A1)** |
| **4. A2 = Read(Y)** | **4. A2 = Read(Y)** |
| **5. A2 = A2 + k** | **5. A2 = A2 * 1.01** |
| **6. Write(Y, A2)** | **6. Write(Y, A2)** |

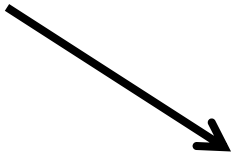
# Basic TO Algorithm Example
## (Παράδειγμα Βασικός Αλγόριθμος Διάταξης Χρονοσήμων)

- **Is the schedule serializable?**
  - Utilize the **Basic TO Algorithm** to justify your answer (otherwise the precedence graph could have be used to answer this question)

T1(10) T2(20)

**RTS(X) : 10**
**WTS(X) : 10**
RTS(Y) : 0
WTS(Y) : 0

1. **A1 = Read(X)**
2. A1 = A1 – k
3. **Write(X, A1)**

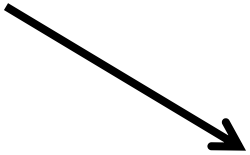


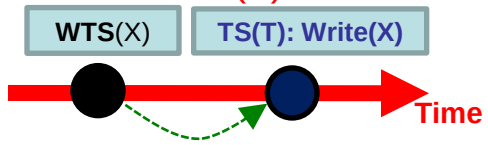

1. **A1 = Read(X)**
2. A1 = A1* 1.01
3. **Write(X, A1)**

**RTS(X) : 20**
**WTS(X) : 20**
RTS(Y) : 0
WTS(Y) : 0

RTS(X) : 20
WTS(X) : 20
**RTS(Y) : 10**
**WTS(Y) : 10**

4. **A2 = Read(Y)**
5. A2 = A2 + k
6. **Write(Y, A2)**






4. **A2 = Read(Y)**
5. A2 = A2 * 1.01
6. **Write(Y, A2)**

RTS(X) : 20
WTS(X) : 20
**RTS(Y) : 20**
**WTS(Y) : 20**

**Yes! The schedule is serializable!**
This can be confirmed by the precedence graph which is acyclic

# Basic TO Algorithm Example
## (Παράδειγμα Βασικός Αλγόριθμος Διάταξης Χρονοσήμων)

- **Is the schedule serializable?**
  - Utilize the **Basic TO Algorithm** to justify your answer

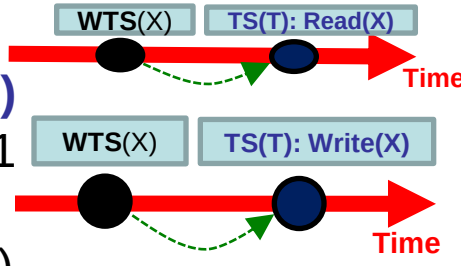
**T1(10)**

RTS(X) : 10
WTS(X) : 10
RTS(Y) : 0
WTS(Y) : 0

1. A1 = Read(X)
2. A1 = A1 – k
3. **Write(X, A1)**

**T2(20)**

1. A1 = **Read(X)**
2. A1 = A1* 1.01
3. **Write(X, A1)**
4. A2 = Read(Y)
5. A2 = A2 * 1.01
6. **Write(Y, A2)**



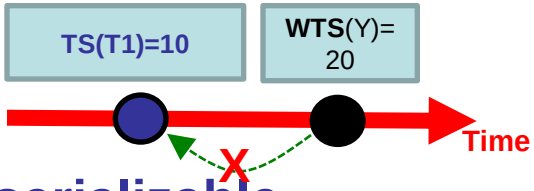

RTS(X) : 20
WTS(X) : 20
RTS(Y) : 20
WTS(Y) : 20

**The Basic TO algorithm only ensures serializability.**

**Recoverability is discussed in the context of ths Strict TO.**

4. A2 = **Read(Y)**
5. A2 = A2 + k
6. Write(Y, A2)

X




**Reject T1 (TO rule violated)**
**(Restart with new TS)**

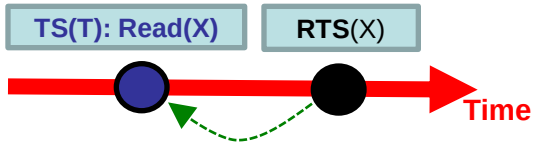
**NO! The schedule is NOT serializable**

- this is confirmed with the precedence graph which is **cyclic**

# Advantages/Disadvantages of Basic TO (Πλεονεκ./Μειονεκ. του Βασικού Αλγ. Διατ. Χρον.)

- **Basic TO Remark**
  - **Note that there is no notion of RR-conflict**

  If **TS(T) < RTS(X),** then execute **read(X)** of T and update **RTS(X)**.




**Accept T**
**(TO rule NOT violated)**

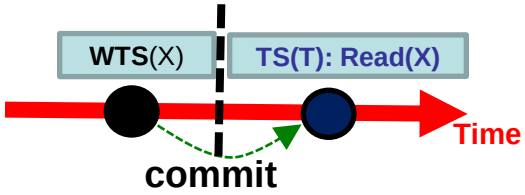
- **Advantages of Basic TO Algorithm**
  - Schedules are serializable (like 2PL protocols)
  - No waiting for transaction, thus, no deadlocks!
- **Disadvantages**
  - Schedule may not be **recoverable** (read uncomit. data)
    - **Solution:** Utilize **Strict TO** Algorithm (see next)
  - **Starvation** is possible (if the same transaction is continually aborted and restarted)
    - **Solution:** Assign new timestamp for aborted transaction

# Strict Timestamp Ordering
## (Αυστηρός Αλγόριθμος Διάταξης Χρονοσήμων)

- The **Basic T.O.** algorithm guarantees **serializability** but not **recoverability (επαναφερσιμότητα)**
- The **Strict T.O.** algorithms introduces **recoverability.**
  - **(Revision) Strict Schedule:** A transaction can neither **read** or **write** an **uncommitted data** item **X.**

- **Strict T.O. Main Idea:** Extend the **Accept cases** of the **Basic T.O. algorithm** by adding the requirement that a commit occurs before T proceeds with its operation. i.e.,

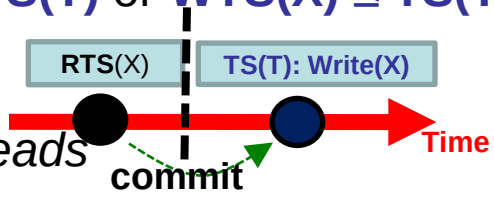
**For read()**
**WTS(X) ≤ TS(T)**

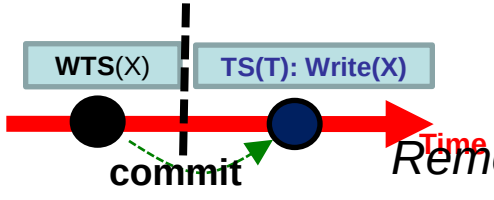



**Accept T (TO ok)**
*Remove Dirty Reads*

**For write()**
**RTS(X) ≤ TS(T)** or **WTS(X) ≤ TS(T)**

*Remove Unrepeatable reads*




**Accept T (TO ok)**
*Remove Lost updates*