

Today:

- **break planned ~13-13.10**
- **end planned ~ 13.50**

(I have to give a talk after the lecture)

# Concurrency Control

Databases, Aarhus University

Ira Assent

# Intended learning outcomes

---

- ▶ Be able to
  - ▶ Determine serializable schedules
  - ▶ Apply the two-phase locking protocol

# Welcome to IADB students

---

- ▶ Instructor: Ira Assent
- ▶ “Super” TAs Pernille Matthews and Cheng Huang
- ▶ TAs for exercises classes and at **study café**
- ▶ **Lectures:** Tuesdays 12-14, Thursdays 10-12
  - ▶ **TA sessions** Tue afternoon → Tue morning
- ▶ Weekly **handins** and multiple choice **exam** in June
  - ▶ Handins count for 20% of grade, exam for 80%
  - ▶ Need to sign up for handin groups (contact TA regarding enrollment)
- ▶ **Forum:** make good use of it: your questions here benefit everyone
- ▶ Please check “the to success” on homepage
- ▶ DBS pre-requisite
  - ▶ Access to earlier material from this year if you need to brush up
- ▶ See also details on brightspace page
- ▶ And for the rest of you: Welcome back after Easter!



# Recap: Transactions

---

- ▶ Transaction
  - ▶ **Logical unit** of database processing
  - ▶ includes one or more operations on the database
- ▶ Basic operations are read and write
  - ▶ read\_item(X)
    - ▶ Reads a database item named X
      - ▶ E.g. into a program variable
  - ▶ write\_item(X)
    - ▶ Writes the value of database item X
  - ▶ We do not need any other information to understand the impact of transactions on the database state
    - ▶ I.e., which value a database item has, and which transaction has seen which value of a database item
- ▶ BEGIN **or** START TRANSACTION, COMMIT, ROLLBACK

<https://dev.mysql.com/doc/refman/8.0/en/commit.html>

<https://dev.mysql.com/doc/refman/8.0/en/sql-transactional-statements.html>



# Recap:ACID

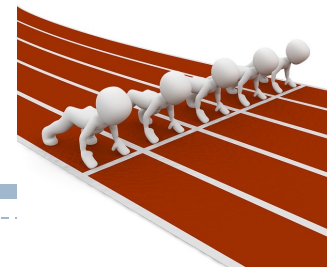
---



- ▶ DBMS supports **ACID** transactions
  - ▶ **Atomic** : Either the whole transaction process is done or none is
    - ▶ Any transaction does all its work on the database or none at all (atomicity)
      - so that there are no partial updates to the data or the like
  - ▶ **Consistent** : Database constraints are preserved
    - ▶ Any transaction that starts on a consistent database leaves it again in a consistent state; temporary inconsistency while it is still working is permitted;
      - so that there are no data integrity issues or the like
  - ▶ **Isolated** : It appears to user as if only one transaction process executes at a time
    - ▶ No transaction interferes with another
      - So that transactions do not produce conflicting results; looks like first one of the transactions has the database to itself until done, then the other
  - ▶ **Durable** : Effects of a process do not get lost if the system crashes
    - ▶ So that the changes made by a transaction actually persist and do not disappear
      - E.g. data is not actually written to the disk or the like

# Concurrency control and schedules

---



- ▶ Goal of concurrency control: support more than one user / query at the same time
  - ▶ Sequential / serial: simply queue → slow
- ▶ Concurrent: allow access to database at the same time in interleaved fashion, but ensure same result
- ▶ **Schedule**: sequence of operations from one or more transactions
  - ▶ Operations considered: read(A), write (A) (start, commit, end, abort)
  - ▶ In concurrent transactions, the operations are interleaved

# ACID: What is concurrency control mostly concerned with?

- A. A, I
- B. A, C
- C. A, D
- D. I, C
- E. I, D
- F. C, D



# Types of schedules

---

## ▶ **Serial** schedule:

- ▶ Schedule S serial if, for every transaction T in S, all operations of T executed consecutively in S
  - ▶ Otherwise, **nonserial** schedule
- ▶ Simplest case: each transaction gets its turn to have the database by itself, while others wait

## ▶ **Serializable** schedule:

- ▶ Schedule S serializable if **equivalent** to some serial schedule of same transactions
- ▶ Typically, operations of different transactions interweaved, but without creating issues
- ▶ These are the kinds of schedules we want to create
  - ▶ Fast, multi-user access and **correct** result



# Example

- ▶ Database with two items, X and Y, program variables x, y

- ▶ Only criterion for correctness:  $X=Y$

- ▶ Two concurrent transactions for programs

$T_1$ :       $X := X + 1$

$Y := Y + 1$

$T_2$ :       $X := 2 * X$

$Y := 2 * Y$

- ▶ Initially,  $X=10$  and  $Y=10$

- ▶ Serial schedule to the right:

- ▶ execute first  $T_2$  entirely, then  $T_1$  entirely

- ▶ Resulting database state:

- ▶  $X = 21, Y = 21, X = Y$

- ▶ Alternatively, first  $T_1$ , then  $T_2$ :  $X=Y=22$

$T_1$	$T_2$
	read_item(X); X:=2*X; write_item(X); read_item(Y); Y:=2*Y; write_item(Y);
read_item(X); X:=X+1; write_item(X); read_item(Y); Y:=Y+1; write_item(Y);	

# Motivation for serializability

$T_1$	$T_2$
read_item(X); $X := X + 1$ ; write_item(X);	
	read_item(X); $X := 2 * X$ ; write_item(X);
	read_item(Y); $Y := 2 * Y$ ; write_item(Y);
read_item(Y); $Y := Y + 1$ ; write_item(Y);	

- ▶ Alternative concurrent schedule “incorrect”:  $X=22, Y=21, X \neq Y$
- ▶ Concurrent execution of transactions **correct** if equivalent to some serial execution of those transactions
- ▶ For transactions  $T_1$  and  $T_2$  there are only two serial schedules:  $T_1, T_2$  or  $T_2, T_1$  and the result of this schedule is different from both of these
  - ▶  $X=Y=21$  or  $X=Y=22$
  - ▶ Why? Because both  $T_1$  and  $T_2$  write  $X$

# Defining equivalence of schedules

---

- ▶ When are two schedules **equivalent**?
  - ▶ In particular, we would like to understand if an interweaved schedule is equivalent to a serial schedule
- ▶ Two schedules are **result equivalent** if they produce the same final state of the database
  - ▶ This can happen by “accident” – state of the database is the same, but the operations differ → cannot be used to find the right schedules
- ▶ Instead, look at the operations in the transactions
  - ▶ Do they interfere with one another?
  - ▶ Notion of **conflict**



# Determining Serializability

---

- ▶ TAs  $T_i$  and  $T_j$  **conflict** iff there exists some item  $X$ , accessed by both  $T_i$  and  $T_j$ , and at least one of them wrote  $X$ 
  - ▶ Intuitively, conflict between TAs forces an execution order between them
  - ▶ If there are no writes, the database state does not change, so no issue if both of them read it
- ▶ Let  $I$  and  $J$  be consecutive instructions by two different transactions within schedule  $S$ 
  - ▶ If  $I$  and  $J$  do not conflict, may swap order to produce new schedule  $S'$
  - ▶  $S$  and  $S'$  **conflict equivalent**
- ▶ Schedule **conflict serializable** if conflict equivalent to a serial schedule



# Serializability

This schedule with an initial database satisfying  $X = Y$  is

- A. Equivalent to a serial execution of  $T_3, T_4$
- B. Equivalent to a serial execution of  $T_4, T_3$
- C. Equivalent to a serial execution of  $T_3, T_4$  and  $T_4, T_3$
- D. Not equivalent to a serial execution of  $T_3, T_4$  or  $T_4, T_3$

$T_3$	$T_4$
read_item(X);	
$X := X + 1$ ;	
	read_item(X);
write_item(X);	
	$X := 2 * X$ ;
	write_item(X);
	read_item(Y);
	$Y := 2 * Y$ ;
read_item(Y);	
$Y := Y + 1$ ;	
write_item(Y);	
	write_item(Y);



# Possible Transaction Conflicts

## ▶ Write/Read conflict:

- ▶  $T_2$  must be executed after  $T_1$ , as  $T_2$  reads value provided by  $T_1$

$T_1$	$T_2$
write_item(X);	read_item(X);

## ▶ Read/Write conflict:

- ▶  $T_2$  must be executed after  $T_1$ , as  $T_2$  writes a new value after  $T_1$  reads the old value

$T_1$	$T_2$
read_item(X);	write_item(X);

## ▶ Write/Write conflict:

- ▶  $T_2$  must be executed after  $T_1$ , as  $T_2$  overwrites value created by  $T_1$

$T_1$	$T_2$
write_item(X);	write_item(X);

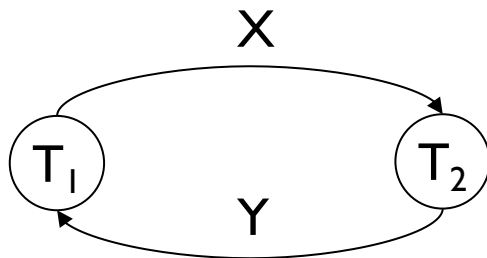
## ▶ No conflict:

- ▶ no implied execution order of  $T_1$  and  $T_2$  as both read same value of X

$T_1$	$T_2$
read_item(X);	read_item(X);

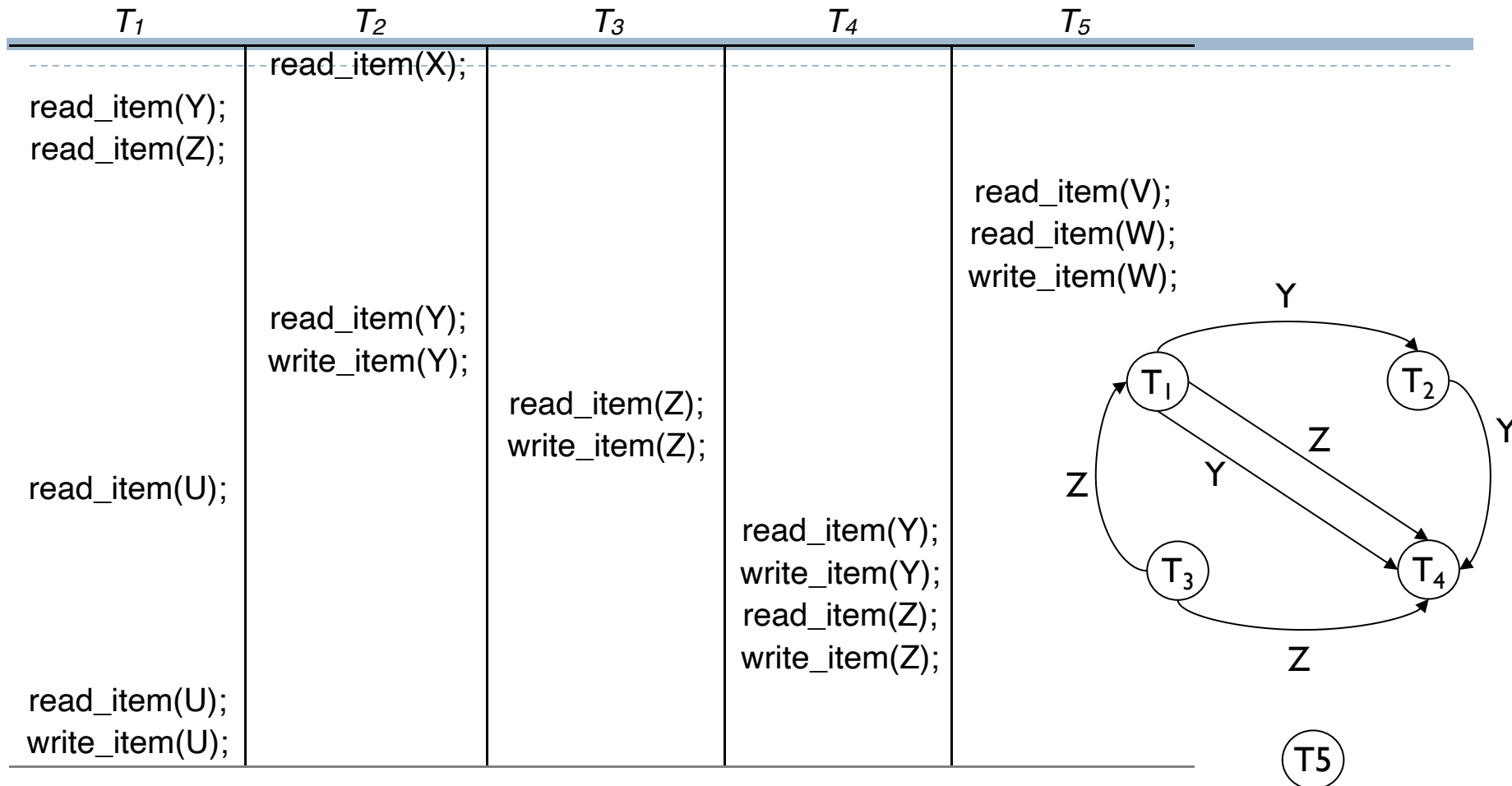
# Precedence Graph (serialization graph)

- ▶ Directed graph depicting conflicts in schedule
- ▶ Each transaction is a node
- ▶ edge labeled  $X$  from  $T_i$  to  $T_j$  if  $T_i$  conflicts on  $X$  (and is before)  $T_j$



$T_1$	$T_2$
read_item(X); $X := X + 1$ ; write_item(X);	read_item(X); $X := 2 * X$ ; write_item(X); read_item(Y); $Y := 2 * Y$ ; write_item(Y);
read_item(Y); $Y := Y + 1$ ; write_item(Y);	

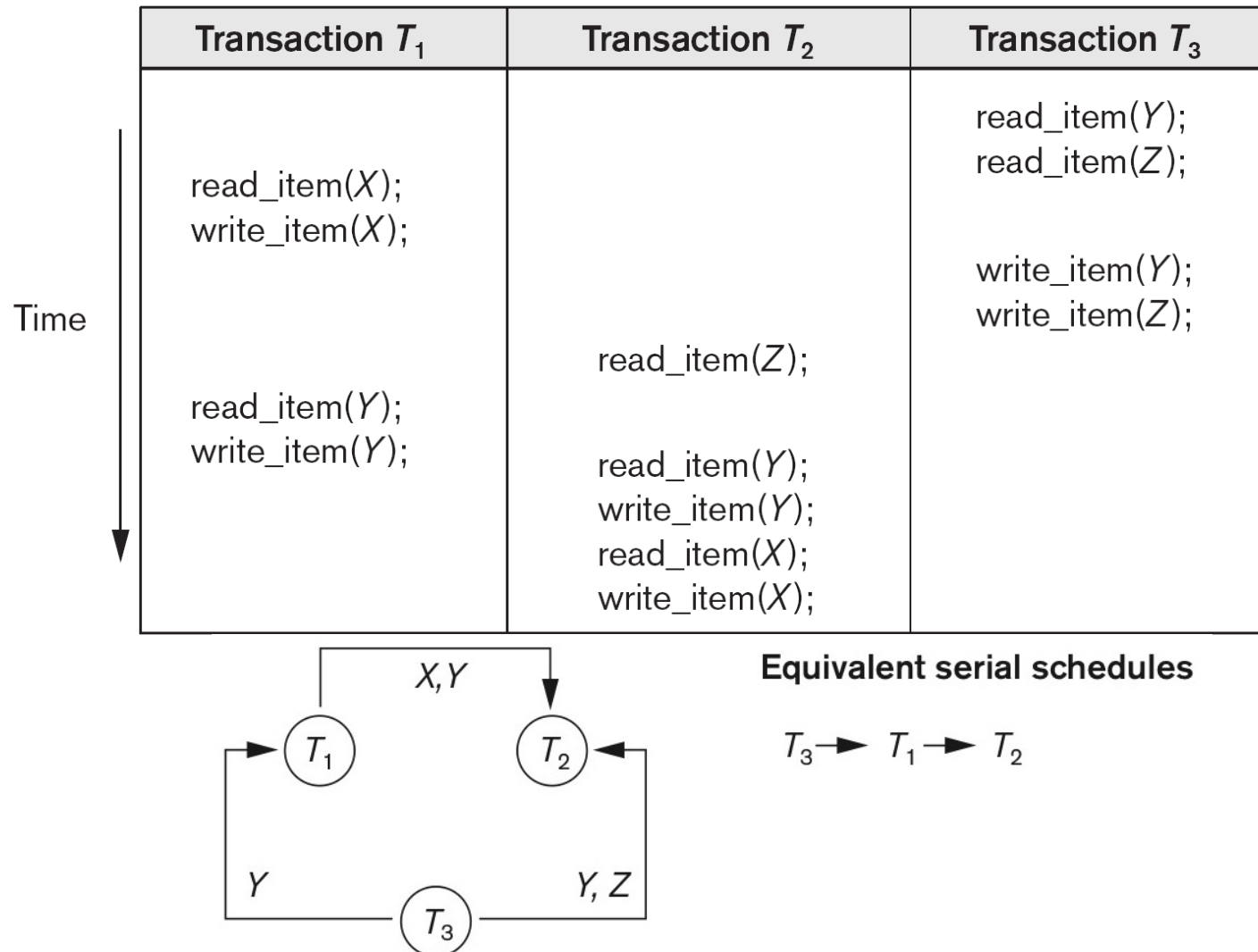
# Is this precedence graph correct?



- A. No,  $T_5$  is not connected
- B. No, the label from  $T_2$  to  $T_4$  should be Z
- C. No,  $T_2$  should precede  $T_1$
- D. No,  $T_1$  should precede  $T_3$
- E. No, arrows point the wrong way
- F. Yes



# Textbook Example of Serializability Testing



# Another Example of Serializability Testing

	Transaction $T_1$	Transaction $T_2$	Transaction $T_3$
Time ↓	read_item(X); write_item(X);	read_item(Z); read_item(Y); write_item(Y);	read_item(Y); read_item(Z);
	read_item(Y); write_item(Y);	read_item(X);  write_item(X);	write_item(Y); write_item(Z);

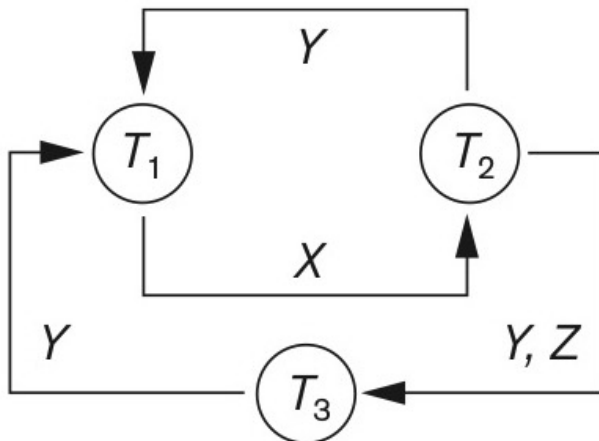
## Equivalent serial schedules

None

## Reason

*Cycle*  $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$

*Cycle*  $X(T_1 \rightarrow T_2), YZ(T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$



# Testing Serializability

---

- ▶ Test: schedule is (conflict) serializable if its precedence graph is acyclic
- ▶ Topological sorting gives a serialization order
- ▶ Need to guarantee serializability
  - ▶ An inefficient strategy
    - ▶ Generate a schedule, build the precedence graph, and test for cycle
    - ▶ If cycle found, need to generate another schedule, ...




# Determining serializability

---

- ▶ Serializability hard to check
  - ▶ Interleaving of operations in operating system through some scheduler
  - ▶ Difficult to determine beforehand how operations in schedule will be interleaved

Practical approach:

- ▶ Devise **protocols** (methods) to ensure serializability
- ▶ Not possible to determine when schedule begins and ends
  - ▶ “the database never sleeps” – endless stream of transactions 
  - ▶ Reduce the problem of checking the whole schedule to checking only a **committed projection** of the schedule
    - ▶ i.e. operations from only the committed transactions
- ▶ Current approach used in most DBMSs:
  - ▶ Use of locks with two phase locking

# Locks for concurrency control



- ▶ Locking is an operation which secures
  - ▶ (a) permission to read and/or
  - ▶ (b) permission to write a data item for a transaction
  - ▶ **Lock (X)**: data item X is locked in behalf of the requesting transaction, so it obtains access permission
- ▶ Unlocking is an operation which removes these permissions from the data item
  - ▶ **Unlock (X)**: data item X is made available to all other transactions
- ▶ Lock and Unlock are atomic operations
- ▶ Transaction must be **well-formed**:
  - ▶ It must lock the data item before it reads or writes to it
  - ▶ It must not lock an already locked data item and it must not try to unlock a free data item
- ▶ Lock Manager maintains lock table with information on (at least) data items with locks currently granted, queue for waiting transactions

# Lock modes

- ▶ Two **locks modes**:
- ▶ Shared mode: **shared lock (X)** or **read lock (X)**
  - ▶ More than one transaction can apply shared lock on X for reading its value, but no write lock can be applied on X by any other transaction
- ▶ Exclusive mode: **exclusive lock (X)** or **write lock (X)**
  - ▶ Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X
- ▶ Conflict matrix
  - ▶ Same as for conflict serializability

	Read	Write
Read	Y	N
Write	N	N



# The Two-Phase Protocol

---

- ▶ Phase 1 (*growing phase*):
  - ▶ Transaction may request locks
  - ▶ Transaction may not release locks
- ▶ Phase 2 (*shrinking phase*):
  - ▶ Transaction may not request locks
  - ▶ Transaction may release locks
- ▶ When the first lock is released, the transaction moves from phase 1 to phase 2



# Two-Phase With Lock Conversion

---

- ▶ First phase
  - ▶ Can acquire a shared lock on item X
  - ▶ Can acquire an exclusive lock on item X
  - ▶ Can **convert (upgrade)** a shared lock on X to an exclusive lock on X
- ▶ Second phase
  - ▶ Can release a shared lock
  - ▶ Can release an exclusive lock
  - ▶ Can **convert (downgrade)** an exclusive lock to a shared lock
- ▶ This protocol assures serializability





# Practical Two-Phase Protocol (2PL)



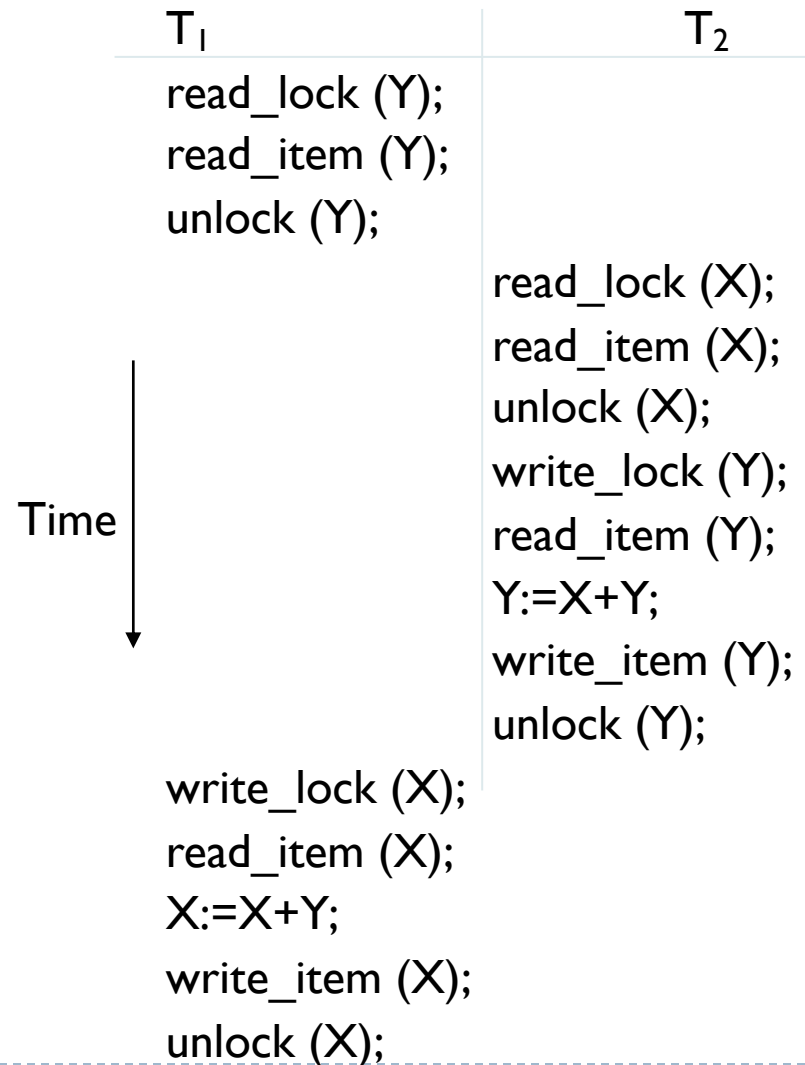
- ▶ Transaction issues standard read/write instructions
- ▶ System manages protocol, including lock operations

**read X:**           if T has a lock on X  
                          perform read  
          else       wait until no other transaction has write\_lock(X)  
                          grant read\_lock(X) to T  
                          perform read

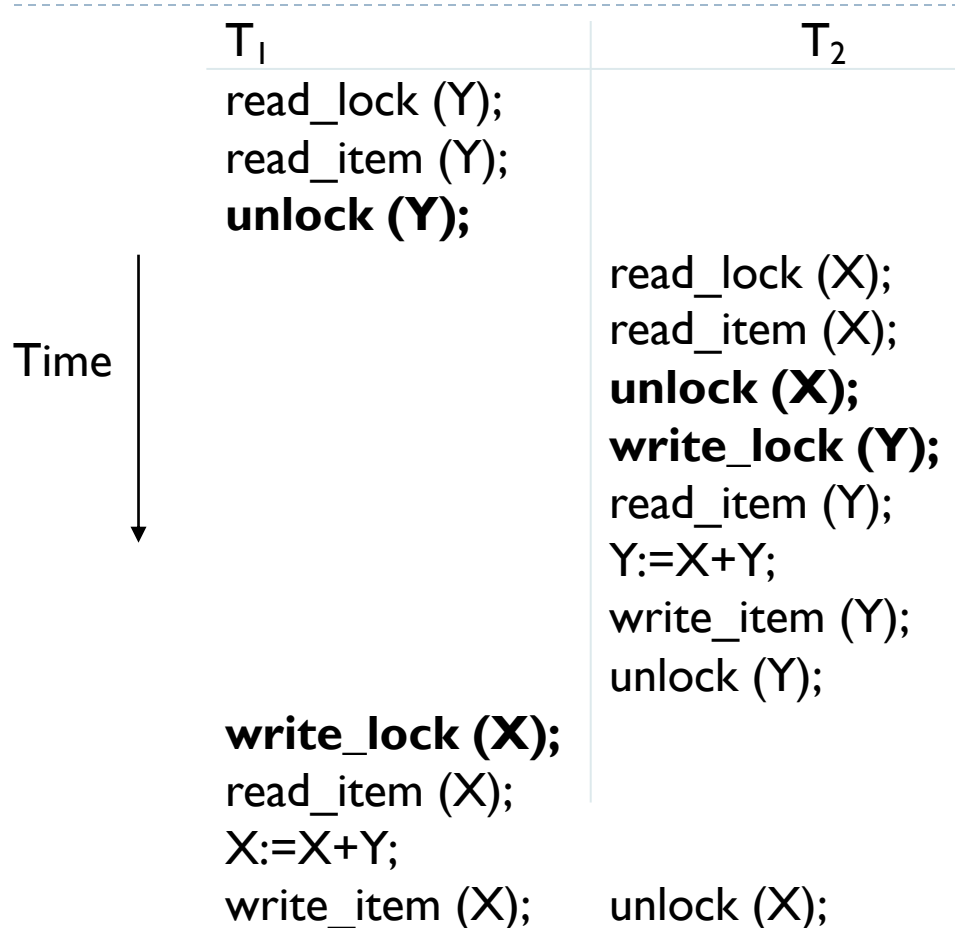
**write X:**       if T has a write\_lock(X)  
                          perform write  
          else       wait until no other transaction has a lock on X  
                          if T has read\_lock(X)  
                                  convert it to write\_lock(X)  
                          else grant it write\_lock(X)  
                          perform write

# 2PL?

- A. Yes, follows 2PL
- B. No, phases not separate
- C. Yes, if lock conversion is applied
- D. No, there is a conflict



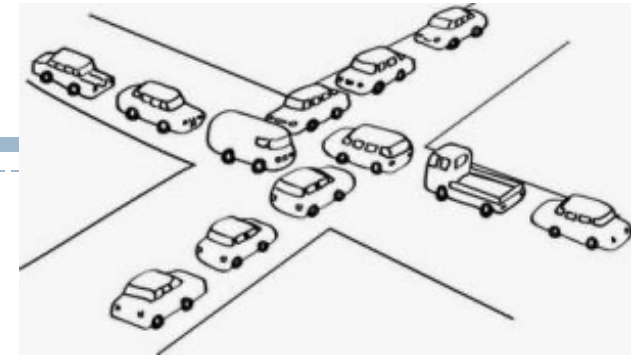
# Textbook locking example



This is **NOT** 2PL, as e.g.  $T_1$  unlocks  $Y$  and then locks  $X$   
→ would not be allowed to run



# Textbook example: 2PL

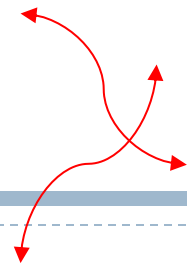


$T_1$	$T_2$
<b>read_lock (Y);</b>	<b>read_lock (X);</b>
read_item (Y);	read_item (X);
	<b>write_lock (Y);</b>
<b>write_lock (X);</b>	
<b>unlock (Y);</b>	
read_item (X);	<b>unlock (X);</b>
$X := X + Y;$	read_item (Y);
write_item (X);	$Y := X + Y;$
<b>unlock (X);</b>	write_item (Y);
	<b>unlock (Y);</b>

## ► Now 2PL policy followed, but **deadlock**

- $T_1$  cannot get **write\_lock(X)** as  $T_2$  has **read\_lock(X)** and  $T_2$  cannot get **write\_lock(Y)** as  $T_1$  has **read\_lock(Y)**

# Deadlocks



## ▶ **Deadlock**

- ▶ A cycle of transactions waiting for one another's unlock (cycle wait)

## ▶ **Deadlock prevention**

- ▶ A transaction locks all data items it refers to before it begins execution
  - ▶ prevents deadlock since a transaction never waits for a data item
  - ▶ Impractical: usually necessary data items not fully known

## ▶ **Deadlock detection and resolution**

- ▶ Scheduler maintains wait-for-graph for detecting cycles
  - ▶ if transaction is blocked, add to graph
  - ▶ chain like  $T_i$  waits for  $T_j$  waits for  $T_k$  waits for  $T_i$  creates cycle

## ▶ **Deadlock avoidance**

- ▶ Avoid deadlock by not letting the cycle complete
  - ▶ When blocking transaction likely to create a cycle, abort the transaction

## ▶ **Starvation** means transaction consistently waits or restarts and never completes

- ▶ In deadlock resolution, same transaction may repeatedly be victim and aborted

# Practical deadlock prevention



- ▶ Prevent deadlocks when a requested lock cannot be obtained
  - ▶ Based on the timestamp (“age”) of transaction: when did it enter?
  - ▶ **Wait-die**: older transaction may wait for younger transaction, younger transaction waiting for older transaction is aborted
    - ▶ Has not yet been running as long, expect fewer resources are wasted
  - ▶ **Wound-wait**: younger transaction may wait for older one, an older transaction will preempt the younger one, i.e., younger one aborted as well
  - ▶ Both techniques handle deadlocks and restart the younger transaction
    - ▶ Still, may abort transactions that are not actually deadlocked
  - ▶ **No waiting**: as soon as lock cannot be obtained, abort transaction, restart with some delay
    - ▶ No deadlocks, but may cause many unnecessary aborts and restarts
  - ▶ **Cautious waiting**: if lock cannot be obtained, abort only if the transaction that has the conflicting lock is waiting, else wait
    - ▶ Also, no deadlocks

# Versions of 2PL

---

## ▶ **Basic**

- ▶ Transaction locks data items incrementally
  - ▶ may cause deadlock which is dealt with

## ▶ **Strict**

- ▶ Exclusive locks unlocked after terminating (commit / abort and rollback)
  - ▶ most commonly used two-phase locking algorithm

## ▶ **Rigorous**

- ▶ All unlocking (shared and exclusive) after terminating (commit / abort and rollback)

## ▶ **Conservative**

- ▶ Prevent deadlock by locking all desired data items before start of transaction
  - ▶ Often not realistic / efficient: as part of execution, determine which data items needed; if not requested from beginning, need to abort and start over



# A working example: 2PL



2PL: growth phase (acquire locks),  
then shrink phase (release locks)

T <sub>1</sub>	T <sub>2</sub>
read_lock (X); read_item (X);	
	read_lock (X); read_item (X);
read_lock(Y); write_lock (Y); unlock (X);	
	write_lock(X); write_item(X);
write_item (Y); unlock (Y);	
	read_lock(Y);
commit;	read_item (Y); write_lock (Y); unlock (X); write_item(Y); unlock (Y); commit;

**2PL basic:** unlock immediately  
when entering 2<sup>nd</sup> phase

T <sub>1</sub>	T <sub>2</sub>
read_lock (X); read_item (X);	
	read_lock (X); read_item (X);
read_lock(Y); write_lock (Y); unlock (X);	
	write_lock(X); write_item(X);
write_item (Y); commit; unlock (Y);	
	read_lock(Y); read_item (Y); write_lock (Y); write_item(Y); commit; unlock (X); unlock (Y);

**2PL strict:** unlock exclusive locks  
when terminating

T <sub>1</sub>	T <sub>2</sub>
read_lock (X); read_item (X);	
	read_lock (X); read_item (X);
read_lock(Y); write_lock (Y); write_item (Y); commit; unlock (X);	
	write_lock(X); write_item(X);
unlock (Y);	
	read_lock(Y); read_item (Y); write_lock (Y); write_item(Y); commit; unlock (X); unlock (Y);

**2PL rigorous:** unlock when  
terminating



# Aggressive versus Conservative Protocols

## ▶ **Aggressive Protocols**

- ▶ Proceed as quickly as possible even if situation may lead to aborts later
- ▶ Example: our practical 2-phase lock conversion protocol



## ▶ **Conservative protocols**

- ▶ Do not do work that may later have to be undone
- ▶ Example: each transaction requests all locks it will ever need at the beginning, and they are held until the end of the transactions
  - ▶ No deadlock!
  - ▶ Livelock (also called starvation) avoided by queuing lock requests
  - ▶ Disadvantage - more waiting, loss of concurrency



# Intended learning outcomes

---

- ▶ Be able to
  - ▶ Determine serializable schedules
  - ▶ Apply the two-phase locking protocol

Acknowledgements: includes slide material by Jeff Ullman, Torben Bach Pedersen, and teachcoop

# What was this all about?

Guidelines for your own review of today's session

---

- ▶ Concurrency control allows transactions to...
- ▶ A schedule is...
  - ▶ It is serial if...
- ▶ A conflict between transactions is...
  - ▶ A precedence graph shows conflicts by...
  - ▶ A (conflict-)serializable schedule is...
- ▶ The two-phase locking protocol consists of...
  - ▶ It guarantees that... because...
  - ▶ The difference between a shared and exclusive lock is...
- ▶ Deadlocks are...
  - ▶ They can happen if...
- ▶ Starvation means...
  - ▶ It occurs when...