CMPT 506

Database Concurrency Control

Read Chapters 20 and 21

Dr. Abdelkarim Erradi

Department of Computer Science and Engineering

QU

Outline

- DB Transaction as a mechanism to achieve Atomicity
- Concurrency control
- Lock-based Concurrency control
- Transaction Isolation Levels
- Time-based Concurrency control
- Optimistic Concurrency Control

DB Transaction as a mechanism to achieve Atomicity

< Back

Transactions ACID properties

- **A**tomic: Everything in a transaction succeeds or the entire transaction is rolled back (All or Nothing)
- **C**onsistent: data affected meet all validation rules such as constraints
- **I**solated: Transactions cannot interfere with each other => The updates of a transaction must not be made visible to other transactions until it is committed
- **D**urable: Results from completed transactions survive failures

Atomicity

- Consider a bank transaction T:

T:	A=A+100,	B=B-100

$\longrightarrow$ time

- T is transferring $100 from B's account to A's account.
- What if there is an error right after the first statement of T has been executed, i.e. the second statement is not executed?

=> You will get **partial update** = inconsistent state

- The DBS has to ensure that every transaction is treated as an *atomic* unit, i.e. either all succeed otherwise rollback

Concept of Transaction

- **Atomicity is achieved using Transaction**
- Transaction = Logical unit of work on the database
    - Transfer money from one bank account to another -
    - Checkout: place order and process payment
- A transaction consists of a sequence of read / write operations that must be performed as a **single** logical unit
- Transaction boundaries are defined by the database user / application programmer

A transaction is a sequence of operations that must be executed as a whole

Ali

Bank Acct

$1000

Transfer $500

Sara

Bank Acct

$0

(1) Debit Ali savings acct

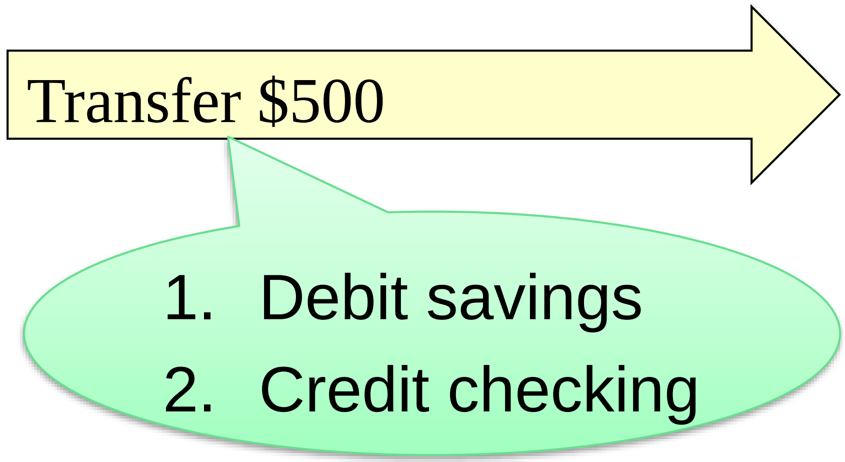
(2) Credit Sara checking Acct

Either both (1) and (2) happen or neither!

Every DB action takes place inside a transaction

We abstract away most of the application code when thinking about transactions

User's point of view



Programmer's point of view

Read Balance1
Write Balance1
Read Balance2
Write Balance2

DB's point of view

Transaction = a sequence of DB reads (R) and writes (W)

T: R(A), W(A), R(B), W(B)

time

SQL Transaction

- By default, each SQL statement (any query or modification of the database or its schema) is treated as a separate transaction
- Transactions can also be defined explicitly

  START TRANSACTION;

  <sequence of SQL statements>

  COMMIT;

  or ROLLBACK;

- COMMIT makes all modifications of the transaction permanent, ROLLBACK undoes all DB modifications made

Concurrency control

Concurrency Control

- Multiple **concurrent** transactions T_1, T_2, … may read/write Data Items A_1, A_2, … concurrently
- Concurrency Control is the process of managing concurrent operations performed on shared data so that data manipulation does not generate inconsistent databases or produce wrong results
- Objective
  - Maximise throughput (i.e., work performed)
  - Minimize response time
- Constraint
  - Avoid **interference** between transactions

Three Concurrency Anomalies

- **Lost update**
  - Two transactions T_1 and T_2 both modify the same data
  - T_1 and T_2 both commit
  - Final state shows effects of only T_1 but not of T_2

- **Dirty read**
  - T_1 reads data written by T_2 while T_2 has not committed
  - If T_2 aborts then T_1 will have dirty data

- **Unrepeatable read**
  - Getting inconsistent results when a read operation is reexecuted within a Transaction T

Illustrative Example

- **Example (to illustrate consistency issues that can be introduced by concurrent updates)**
- Ali at ATM1 withdraws $100
- Sara at ATM2 withdraws $50
- Initial balance = $400, final balance = ?
  - Should be $250 no matter who goes first

```
Read balance from DB;
If balance > withdrawalAmount {
    balance = balance - withdrawalAmount;
    Write balance to DB;
}
```

No concurrent transactions scenario

Ali withdraws $100:

```
read balance; => $400
if balance > amount then
    balance = balance - amount; => $300
    write balance; => $300
```

Sara withdraws $50:

```
read balance; => $300
if balance > amount then
    balance = balance - amount; => $250
    write balance; => $250
```

Lost update problem

Ali withdraws $100:

Sara withdraws $50:

```
read balance; => $400
```

```
read balance; => $400
If balance > amount then
 balance = balance - amount; => $350
 write balance; => $350
```

```
if balance > amount then
  balance = balance - amount; => $300
  write balance; => $300
```

Lost update problem => DB is in inconsistent state

Lost update problem

Ali withdraws $100:

```
read balance; => $400
```

Sara withdraws $50:

```
read balance; => $400
```

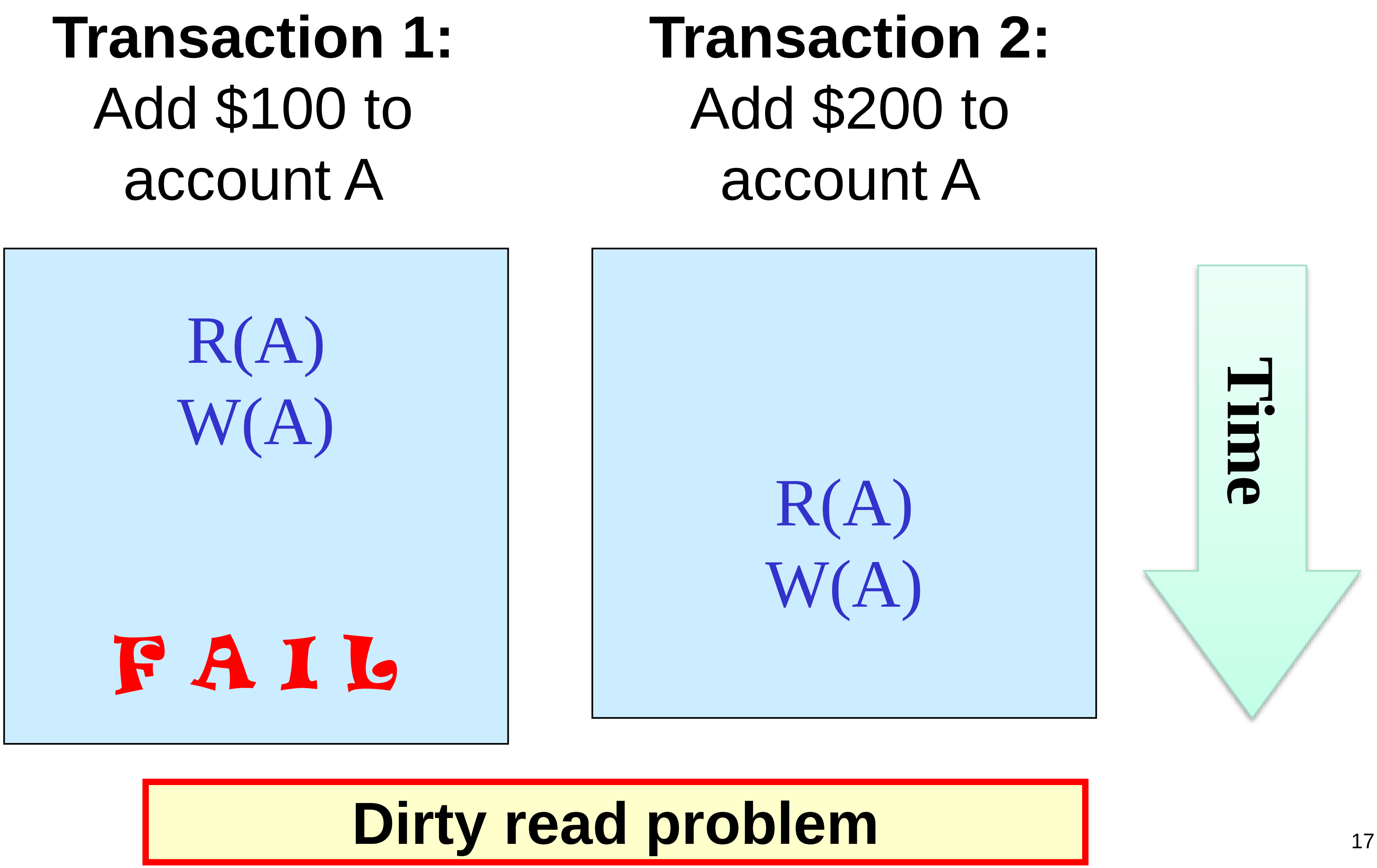
```
if balance > amount then
 balance = balance - amount; => $300
 write balance; => $300
```

```
if balance > amount then
 balance = balance - amount; => $350
 write balance; => $350
```

Lost update problem => DB is in inconsistent state

Dirty read problem

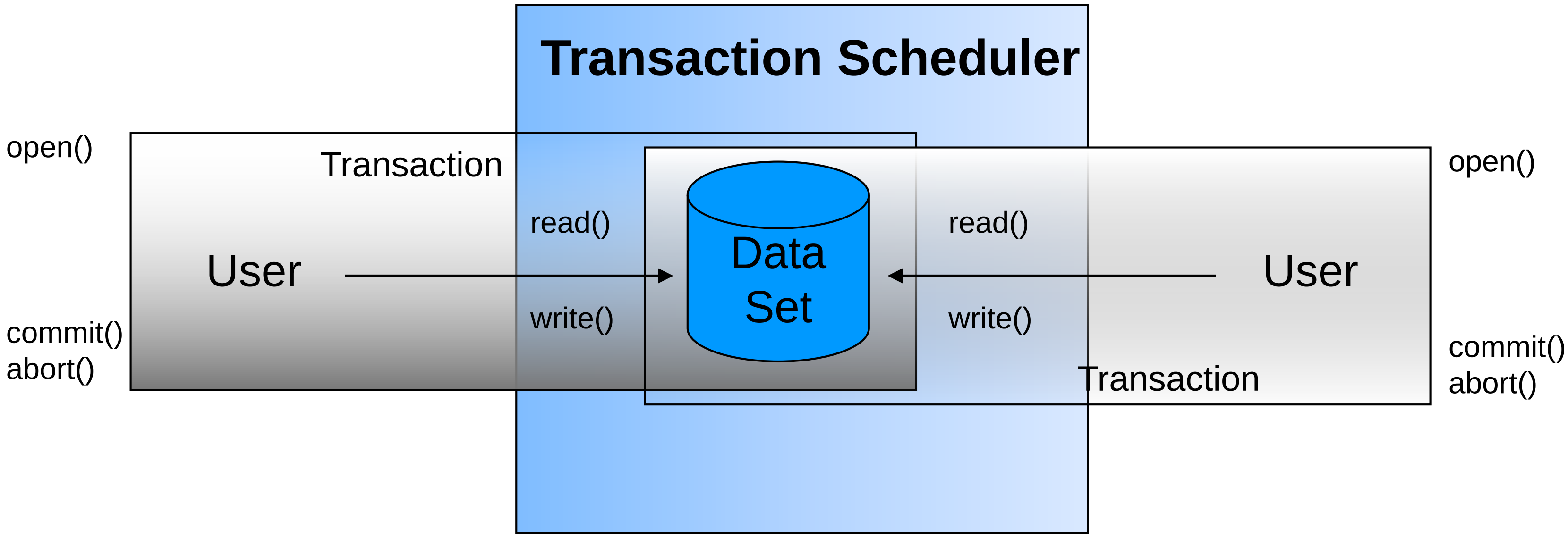
What will be the final account balance?



Isolation

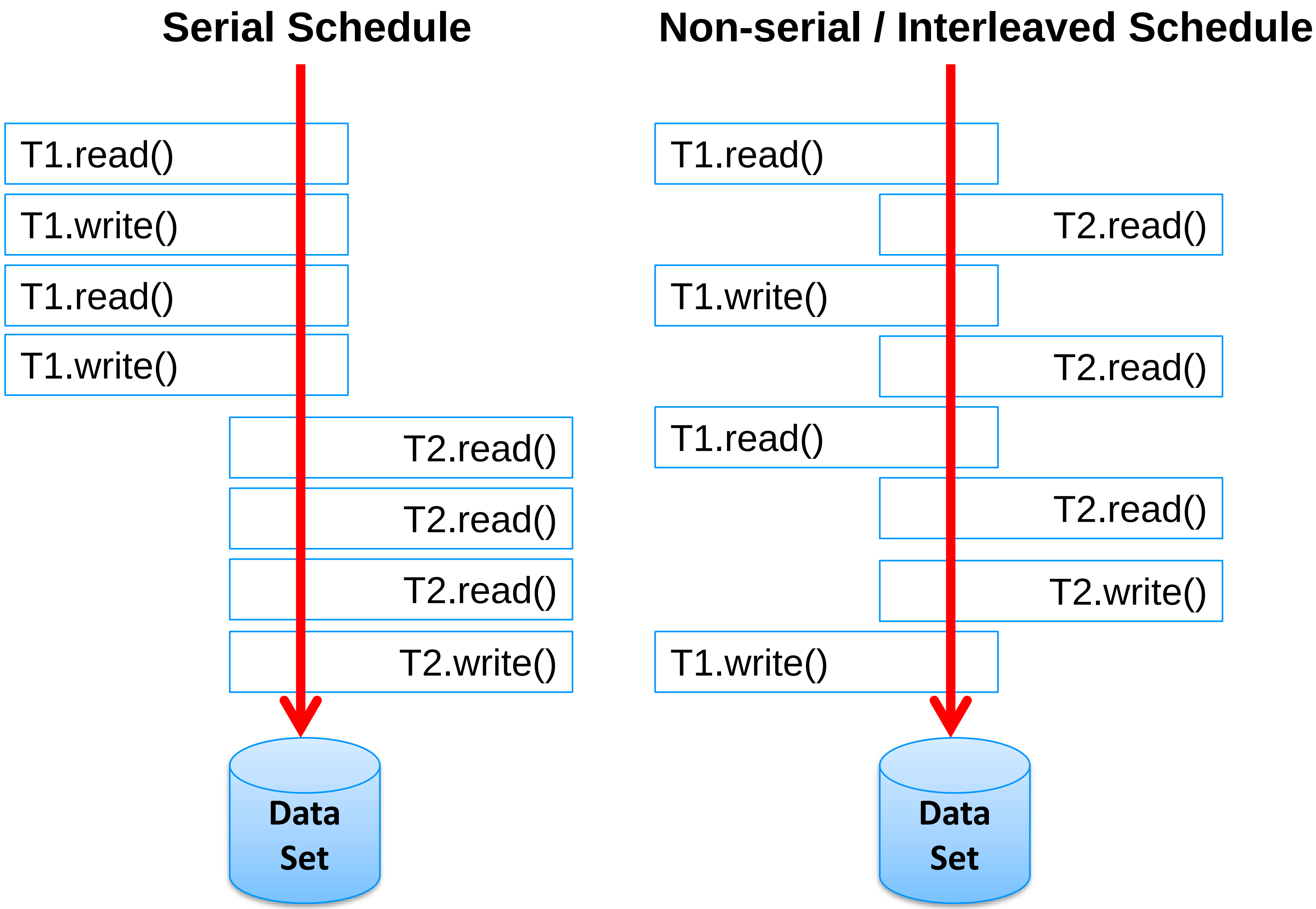
- To preserve consistency the DBMS must manage concurrency to guarantee the **Isolation** property.
  - Independence from all other transactions (serializability) => **same results as if the statements would have been executed in a single user scenario**
  - A transaction should not be affected by other concurrently running transactions

Scheduling Concurrent Transactions



- The transaction scheduler has to organise or "schedule" database read and write actions of all concurrent transactions
- A "**schedule**" is a sequence of operations from different transactions that may be executed in an interleaved fashion
  - Such a schedule may compromise the integrity / consistency of a database

Serial vs. Non-serial Schedule



Serial schedule in which T_1 precedes T_2

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

Constraint of A = B is satisfied after this schedule

Serial schedule in which T_2 precedes T_1

T_1	T_2	A	B
		25	25
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	50	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(A,t)			
t := t+100			
WRITE(A,t)		150	
READ(B,t)			
t := t+100			
WRITE(B,t)			150

Constraint of A = B is satisfied after this schedule

Interleaved Transaction Schedules

- We want that the database system schedules transactions in an interleaved fashion:
  - Improve the responsiveness and increase throughput

- Interleaved schedules also create problems
  - Transactions may overwrite each others' updates
  - Transactions may base their calculations on retrieved data that is already out-of-date or on "dirty reads"

Serializable Transaction Schedules

- In order to avoid the concurrency problems described, one obvious solution would be to **schedule only one transaction at a time** for execution
- Such a completely “serialised” schedule will ensure that the **transactions are completely isolated** and cannot interfere with each other
  - But this strategy will reduce **concurrency** and **throughput**

Serializable Schedule

- DB will try to find a non-serial schedule that is ***equivalent*** to a serial schedule:
  - Schedule is **serializable** if its effect on the database state is the same as that of some serial schedule
- A **serializable schedule** is used by the Transaction Manager to schedule operations of different transactions in a way that interference and problems such as "lost updates" are avoided

A serializable, but not serial, schedule

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

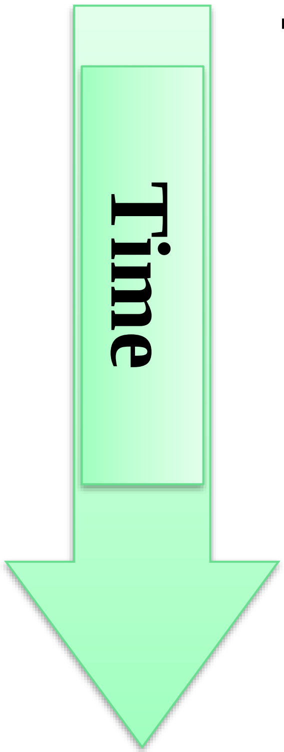
Constraint of A = B is satisfied after this schedule

Non-serializable schedule

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(B,t)			
t := t+100			
WRITE(B,t)			150

Constraint of A = B is violated after this schedule => Non-serializable schedule

Interleaved execution



T1	T2
Read(X)	
	Read(X)
X=X-10	
Write(X)	
Read(Y)	
Y=Y+10	
Write(Y)	
	X = X * 1.1
	Write(X)
	Read(Y)
	Y=Y*1.1
	Write(Y)

‘correct’?

Schedule: The order of execution of operations of two or more transactions

How to define correctness?

Let’s start from something definitely correct:

Serial executions

T1	T2
Read(X)	
X=X-10	
Write(X)	
Read(Y)	
Y=Y+10	
Write(Y)	
	Read(X)
	X = X * 1.1
	Write(X)
	Read(Y)
	Y=Y*1.1
	Write(Y)

‘correct’

by definition

A schedule is ‘correct’ if it is **serializable**
i.e., equivalent to a serial schedule

How to draw a Precedence Graph?

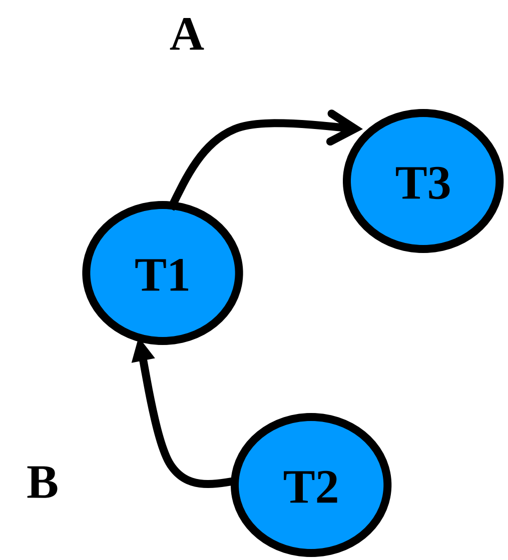
- Precedence Graph = {**Nodes**: transactions,
  Arcs: r/w or w/w conflicts}

- The precedence graph for a schedule S contains:
  - A node for each committed transaction in S
  - An **arc** from T_i to T_j if an action of T_i **precedes and conflicts with** one of T_j's actions.

- The **schedule S is serializable if and only if the precedence graph has no cycles**.

Example 1

T1	T2	T3
Read(A)		
…		
Write(A)		
		Read(A)
		…
		Write(A)
	Read(B)	
	…	
	Write(B)	
Read(B)		
…		
Write(B)		



serial execution?

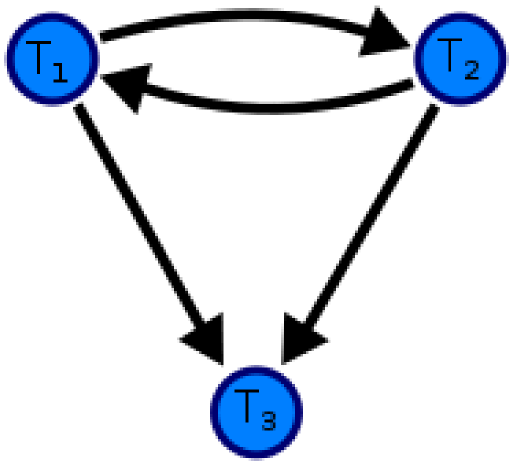
Serializable Schedule: T2, T1, T3

(Notice that T3 should go after T2, although it starts before it!) => always ‘read before write’

Precedence Graph Example 2

$$D = \begin{array}{c} \begin{array}{ccc} T1 & T2 & T3 \end{array} \\ \begin{bmatrix} R(A) & & \\ & W(A) & \\ W(A) & & \\ & & W(A) \end{bmatrix} \end{array}$$

$D = R_1(A)\ W_2(A)\ W_1(A)\ W_3(A)$



As T_1 and T_2 constitute a cycle the above schedule is not serializable.

Example 3 - ‘Lost-update’ problem

T1
Read(N)

N=N-1

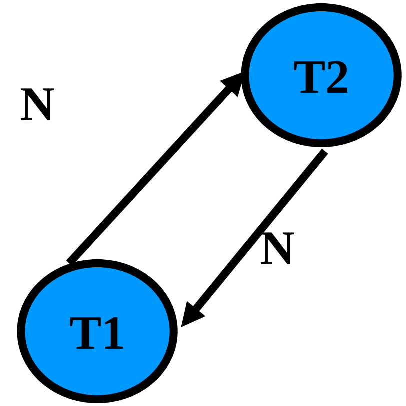
Write(N)

T2

Read(N)

N= N-1

Write(N)



Cycle -> not serializable

Not equivalent to any serial execution (why not?) -> incorrect!

We can draw a precedence graph to prove this

Concurrency Control Protocols

- The basis of concurrency control is protocols to maintain serialization in DBMSs
- E.g. protocols
  - **Two-Phase Locking (2PL)**
  - **Timestamps Ordering**
  - **Optimistic Concurrency Control (OCC)**
  - **Etc.**

Lock-based Concurrency control

Enforcing Serializability by Locks

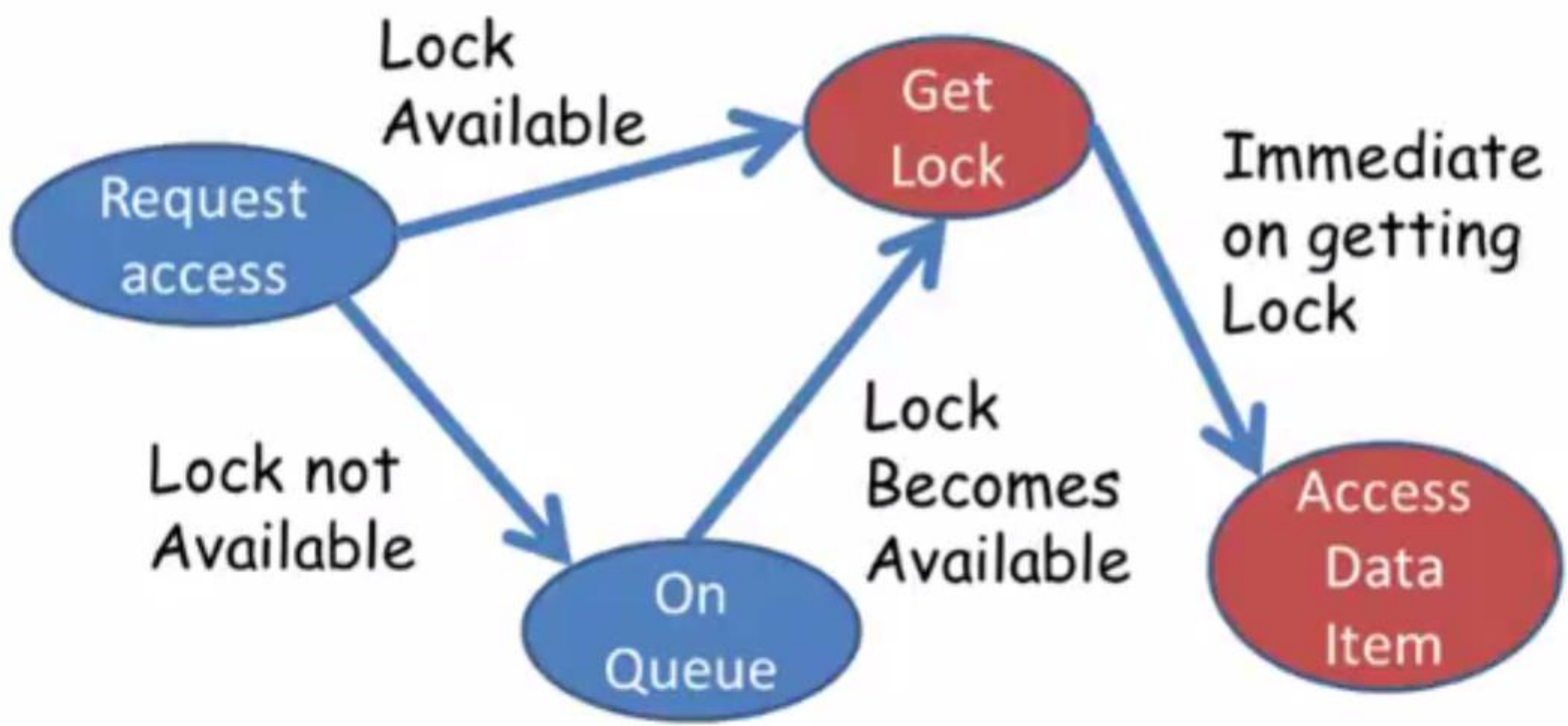
How to achieve correct concurrency control?

- **Locks** = most popular solution for concurrency control
- Lock manager: grants/denies lock requests
- Locking mechanisms prevent conflicts
  - Readers block writers
  - Writers block readers

How lock works

- Transaction needs lock before read or write
- Transaction must ask for lock
- If it does not get the lock
    - maybe another transaction already holds the lock
  - it is suspended
  - put on queue waiting for lock
- When transaction releases lock
  - Some suspended transaction awakened and given lock.

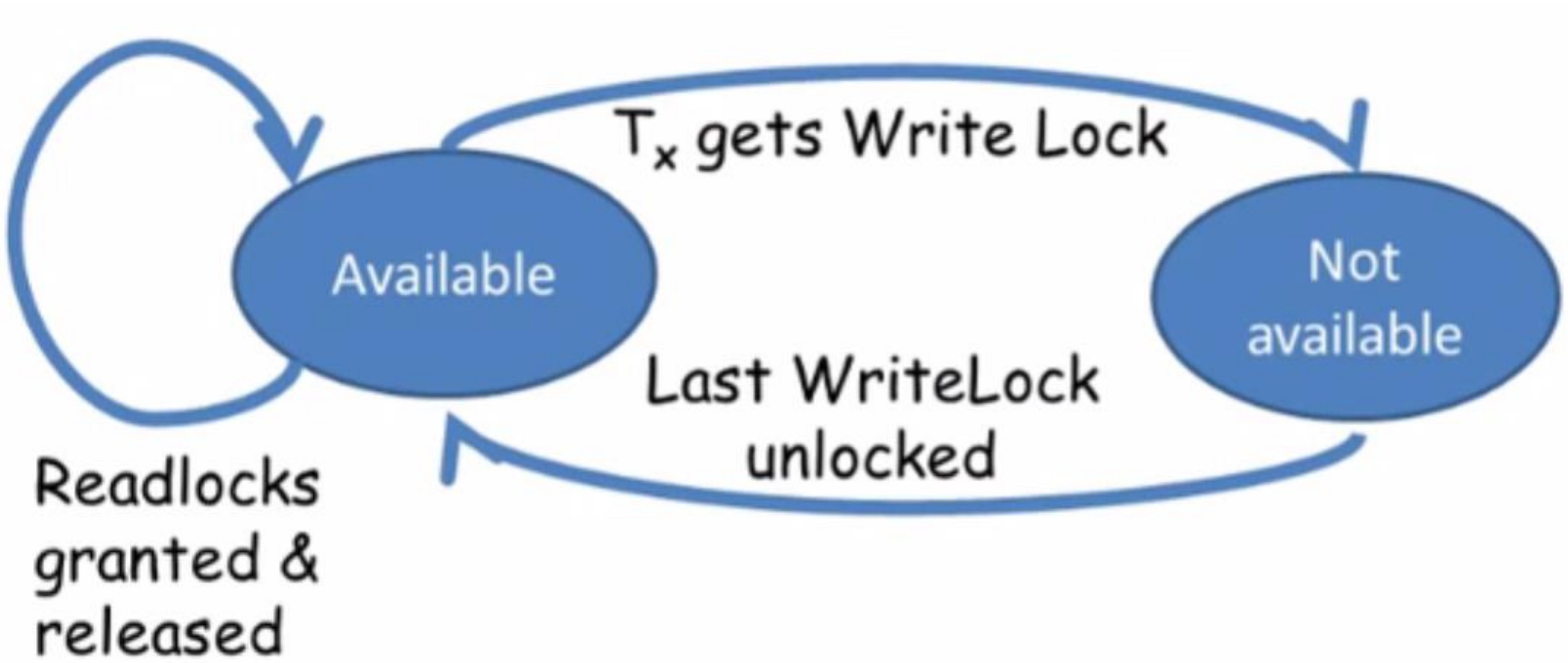
Lock State Diagram



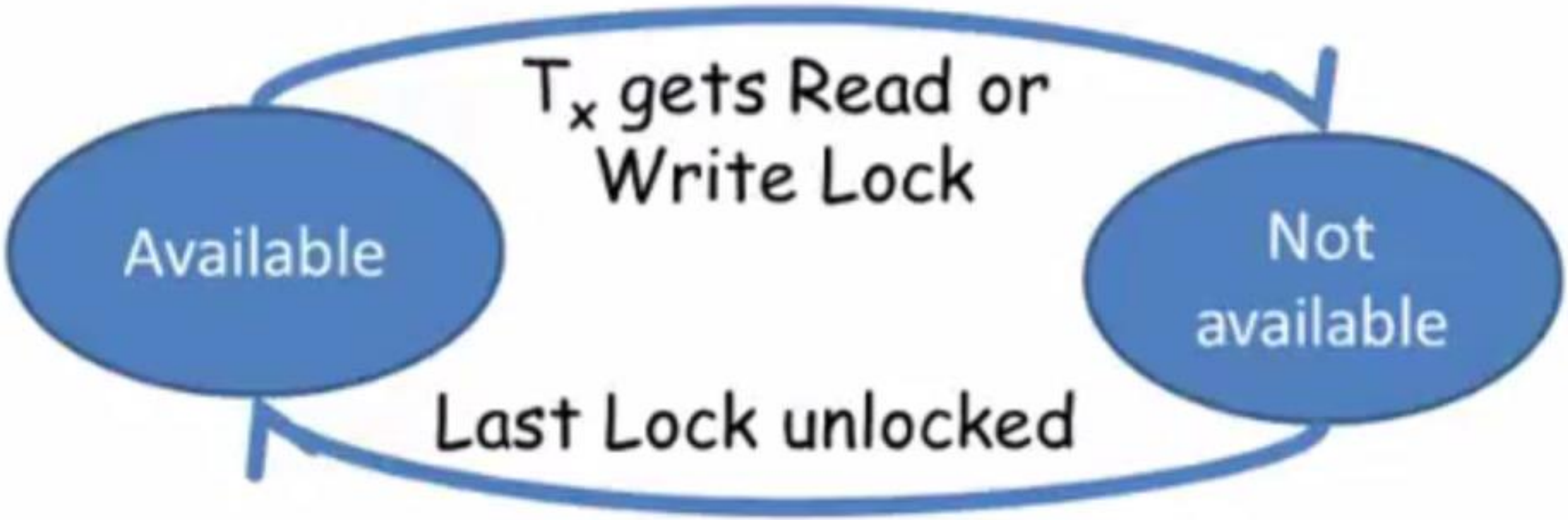
Multi-transaction Rules

- Multiple Read Locks OK
  - Called Shared Lock
  - Several transactions can read same info
- Write Lock must be only lock on item
  - 2 Write Locks lead to race condition: lost update
  - Sharing with Read lock leads to dirty read or non-repeatable read.

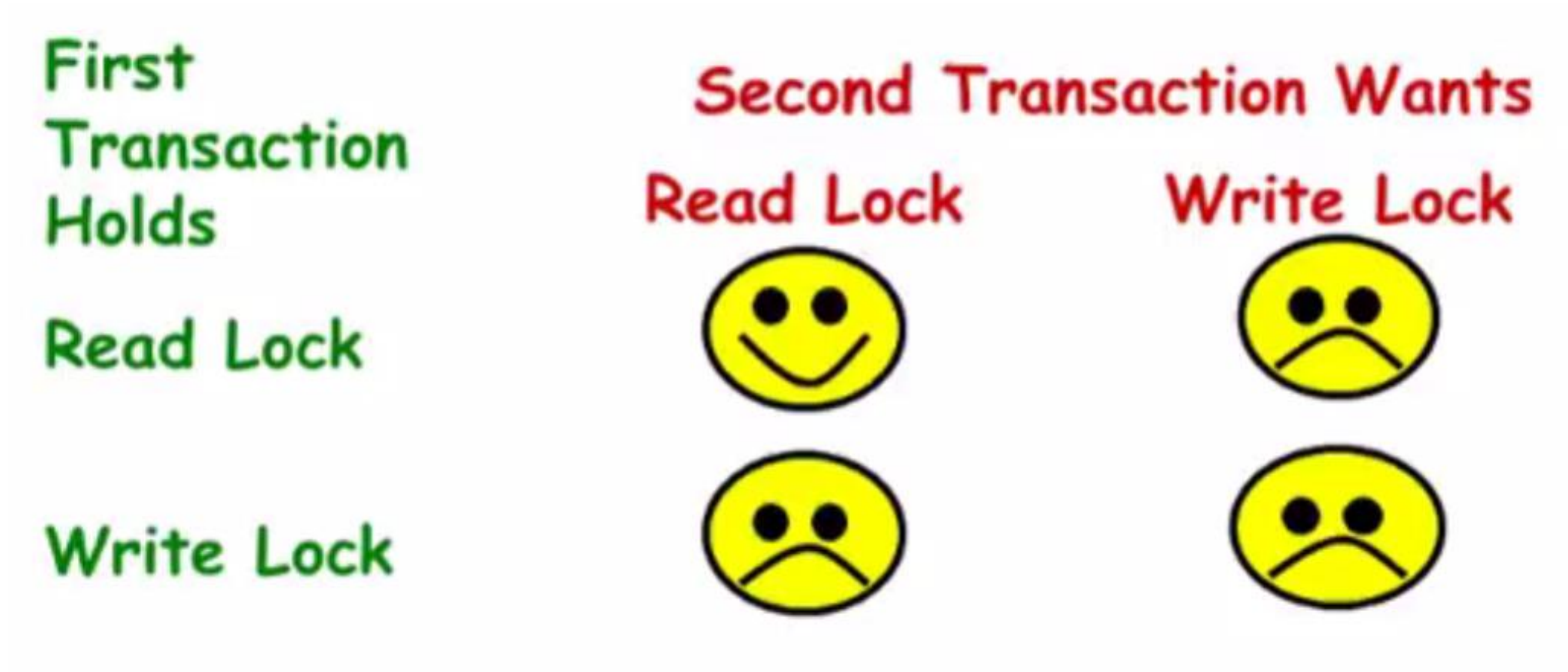
Read Lock State Diagram



Write Lock State Diagram



ReadLock WriteLock Compatibility Matrix



Locking protocol

- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks
- Most popular protocol:

  Strict 2 Phase Locking (2PL) = a transaction must hold all its exclusive locks till it commits/aborts
- **THEOREM**: if all transactions obey 2PL -> all schedules are serializable
  - But reduces concurrency

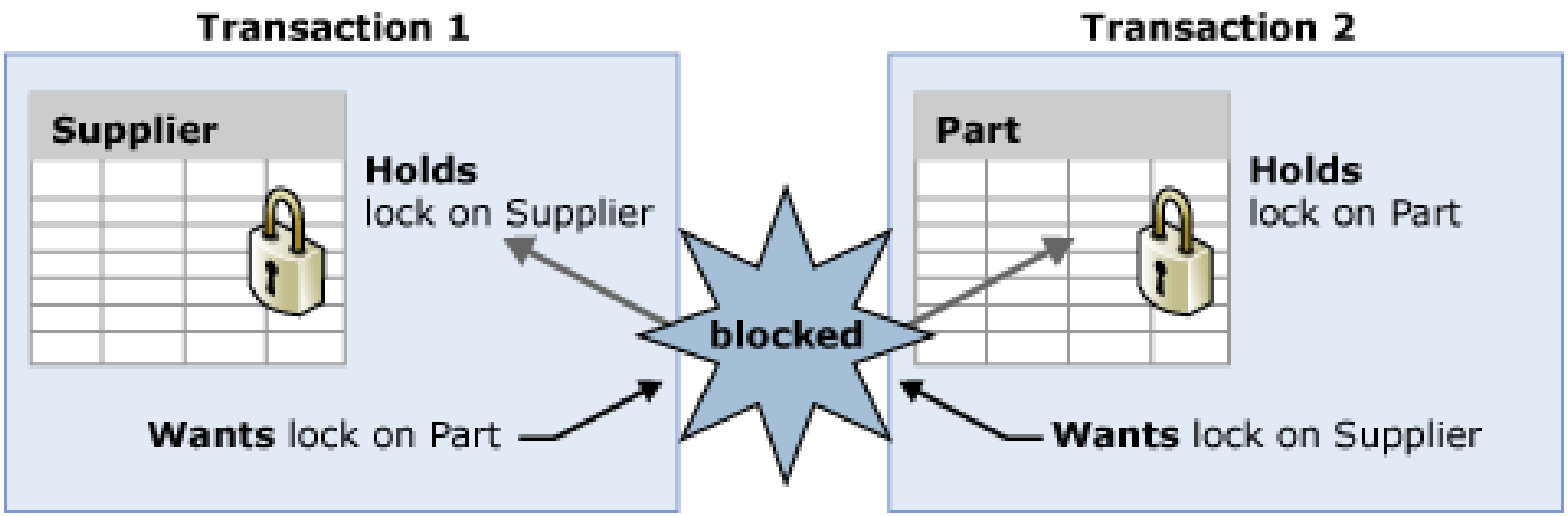
2PL

- Phase 1: **Growing Phase**
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: **Shrinking Phase**
  - transactions issue no lock/upgrade request, after the first unlock/downgrade
  - transaction may release locks
  - transaction may not obtain locks
- 2PL assures serializability

Deadlocks

- The use of locks may lead to deadlocks
- **Definition**
  - A deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock
- The scheduler is responsible for detecting and breaking the deadlock
  - Deadlocks may be broken by simply aborting one of the transactions involved
- How do you choose which transaction to abort?
  - Abort the oldest
  - Abort depending on the complexity of the transactions

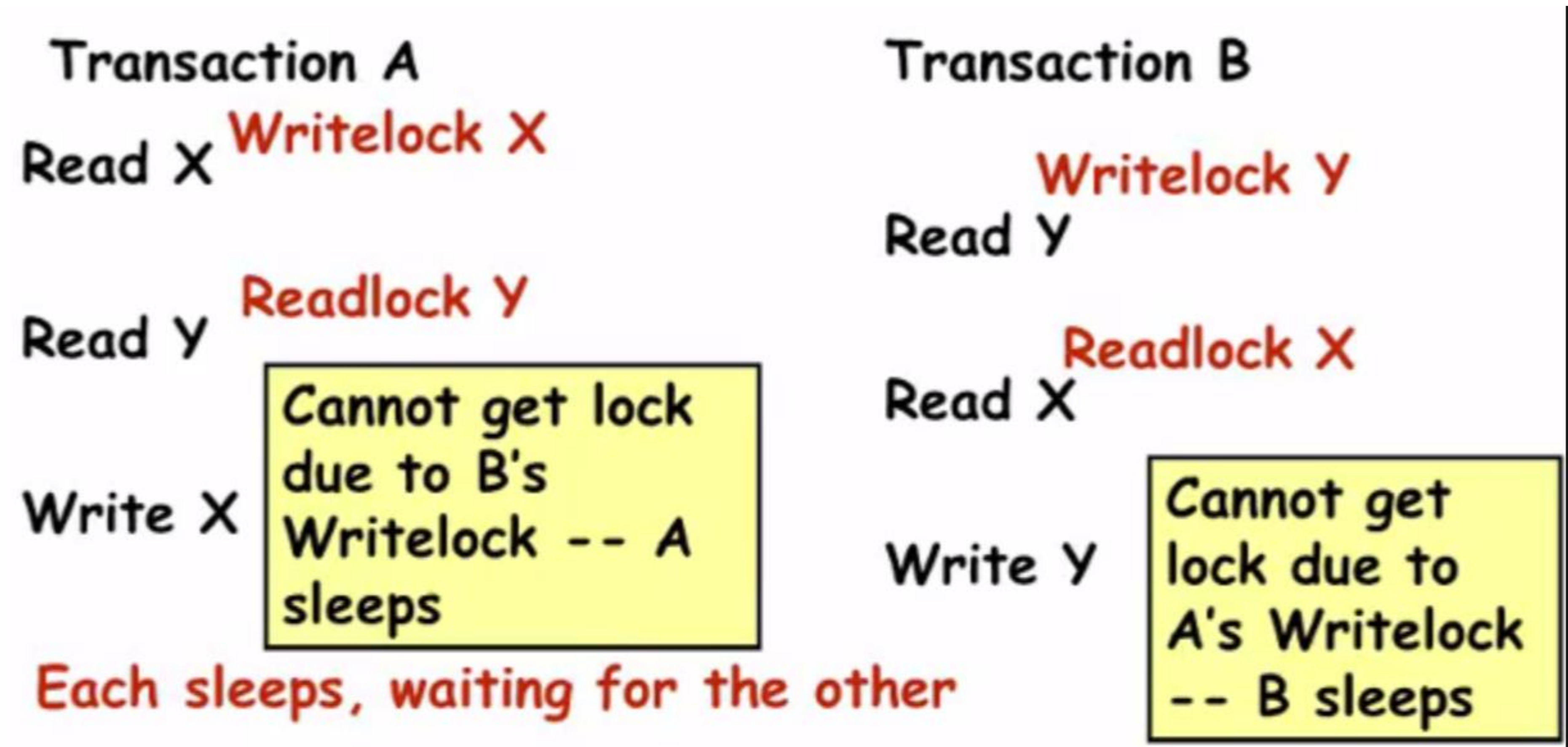
Deadlocks



- Auto-detection and “resolution”

```
Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 232) was deadlocked on lock resources
with another process and has been chosen
as the deadlock victim. Rerun the transaction
```

Deadlock Example



Handling Deadlocks

- Deadlock Detection
  - Use the Precedence graph to identify cycles and select transactions to be aborted
    - Select the transaction involved in most of the cycles
    - Select the oldest transactions
    - Select the one that did the least amount of work
- Deadlock Prevention
  - Lock all objects at the very beginning of a transaction in one atomic action
  - Problem
    - Reduced concurrency: unnecessary access restriction to shared resources

Handling Deadlocks using Timeout

- Rather than detecting deadlock (an overhead), you could just use timeouts, but how long should the timeout be?
- Timeouts
  - Each lock is given a period of time where it is invulnerable
  - After this timeout, it becomes vulnerable
  - If transaction X holds a lock that becomes vulnerable and transaction Y is waiting for X, then X is aborted
  - Problem
    - Hard to decide on an appropriate length of timeout
    - Transactions may be aborted, when the lock becomes vulnerable and another transaction waits, but there is no deadlock

Transaction Isolation Levels

Transaction Isolation Levels

- Determine how much one user impacts another
- 4 standard levels
  - Read Uncommitted
  - Read Committed (default in many DMBS)
  - Repeatable Read
  - Serializable
- The *isolation level* of a transaction defines what data *that* transaction may see

Read Committed (the default)

- Cannot read uncommitted data
- Share lock before reading data and released after processing is complete
- Dirty reads do not happen but non-repeatable reads can happen

Read Uncommitted

- Does not take or honour locks
- Can read uncommitted data
  - *Dirty data* is data that has been modified by a transaction that has not yet committed.
  - If that transaction is rolled back after another transaction has read its dirty data, inconsistency is introduced
- Sacrificing consistency in favour of high concurrency
- Useful for reporting applications
- Be very careful!

Repeatable Read

- All **data** records read by a SELECT statement cannot be changed
- Holds shared locks on data read until transaction commit/rollback
- Rows that have been read can be read again with confidence they won't have changed
  - A query running more than once within the same transaction returns same values
- If the SELECT statement contains any ranged WHERE clauses, phantom reads can occur
- Reduce concurrency and degrade performance

Unrepeatable Reads Example

- A transaction reads the same data item twice, with different results!
  - Start a transaction (T1) under repeatable read isolation level. Query the *ACCOUNTS* table with predicate *(account_balance > 1000)*. Let us say it returns 10 rows
  - Another transaction (T2) cannot update ACCOUNTS with account_balance > *1000*
  - The reason is that transaction T1 only locked the 10 qualifying rows but did not lock the predicate range. With the result, the transaction T2 could insert a new row in the same predicate range.

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('Shrek Toy','blue')
SELECT * FROM Product WHERE color='blue'	

Phantom Problem

- A “phantom” is a tuple that is invisible during part of a transaction execution but appears when the query is re-executed

- In our example:
  – T1: reads list of products
  – T2: inserts a new product
  – T1: re-reads: a new product appears !

Dealing With Phantoms

- Lock the entire table, or
- Lock the index entries of qualifying rows
  - if index is available

Dealing with phantoms is expensive !

Serializable

- This isolation level specifies that all transactions occur in a completely isolated fashion
  - *i.e.*, as if all transactions in the system had executed serially, one after the other
- Locks index ranges as well as rows or table locks
- Phantom rows will not appear if the same query is issued twice within a transaction
- Greatly reduces concurrency

Concurrency and Consistency

Isolation Table	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

- Concurrency and consistency are mutually opposing goals

Time-based Concurrency control

< Back

Timestamp Ordering

- The timestamp of a transaction T is the time at which that transaction was initiated in the DBMS: TS(T)
- We can use clock time or an incremental identifier (counter) for TS(T)
- Two timestamps are also associated with each data item x.

  1. read_TS(x) is the TS(T) of the last transaction T to read from x.

  2. write_TS(x) is the TS(T) of the last transaction T to write to x.

Protocol Rules

- Two simple rules to follow:
  1. Before T issues a write(x), check to see if
     - TS(T) < read_TS(x) or if
     - TS(T) < write_TS(x)
     - If so, then <u>abort transaction </u>T.
     - If not, then perform write(x) and set write_TS(x) = TS(T)

  2. Before T issues a read(x), check to see if
     - TS(T) < write_TS(x) Then <u>**abort**</u> transaction T.
     - if TS(T) >= write_TS(x) then <u>**execute**</u> read(x) and set read_TS(x) = TS(T) only if TS(T) is greater than the current read_TS(x)

Advantages and Limitation

- When a transaction is aborted, it is the restarted and issued a new TS(T).
- Note that with timestamp ordering, deadlock can not occur.
- However, starvation is possible i.e., a transaction keeps getting aborted over and over.

Cascading rollbacks

- Timestamp ordering can also produce cascading rollbacks:
  - Assume transaction T begins executing and performs some read and write operations on data items *a, b and c*
  - However, T then reaches a data item it can not read or write and T must then be aborted.
  - Any effects of transaction T must then be rolled back.
  - Before T aborts, however, other transactions (T1, T2 and T3) have read and written data items *a, b and c so these other transactions must also be rolled* back.
  - There may be other transactions (T4 and T5) that worked with data items read or written by T1, T2 and T3, etc.

Optimistic Concurrency Control

Back

Optimistic Concurrency Control

- Two Phase Locking (2PL) and Timestamp Ordering (TO) are *pessimistic concurrency* control protocols - they assume transactions will conflict and take steps to avoid it. i.e., they address the concurrency issues *before while the transaction is executing and* before the transaction commits.
- 2PL and TO are also *syntactic concurrency control protocols as they deal only with* the syntax (set of read and write operations) of the transactions.

Optimistic Concurrency Control (2)

- In an *optimistic concurrency control protocol, we assume that most of the time,* transactions will not conflict thus all of the locking and timestamp checking are not necessary.
- No checking for serialization is done while the transaction is executing
- During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction
- During a validation phase the transactions updates are check to see if they violate serializability

Basic idea of OCC

- The idea behind OCC is to do all the checks at once
- If there is little interference between transactions, most will be validated successfully.
- Extra requirements for OCC:
  - Local Copy
  - Transaction Timestamps
  - Must keep track of write_set & read_set

Optimistic Concurrency Control in three stages

1. **Read Stage:** Transactions can read any data item. Writes are done to a local copy of the data item e.g., recorded in a log.
2. **Validation stage:** Transactions containing Write operations that are about to commit are *validated to see if the schedule meets the serializability* requirements.
3. **Write stage:** If the transaction will not conflict with other transactions, then it will be committed (writes to local copy applied to the database). Otherwise, the transaction will be rolled back.

RECAP

- Concurrency control motivation
  - If we insist only one transaction can execute at a time, in serial order, then performance will be quite poor.
- **Concurrency Control is a method for controlling or scheduling the operations of transactions in such a way that concurrent transactions can be executed safely** (i.e., without causing the database to reach an inconsistent state)
- If we do concurrency control properly, then we can maximize <u>*transaction throughput*</u> while avoiding any chance of corrupting the database