Large Scale Systems
CS 410 / 510

Lecture 3:

Transactions

Suyash Gupta

Assistant Professor

Distopia Labs and ORNG

Dept. of Computer Science

(E) suyash@uoregon.edu

(W) gupta-suyash.github.io

UNIVERSITY OF OREGON

Assignment 1 is Out!

- **Assignment 1 is out!**
- Please work with your groups to understand the underlying system.
- Assignment 1 report **deadline** → April 16, 2026 at 11:59pm.

Transactions?

Transactions

- Transactions are ubiquitous!

- Examples: Banking, Online shopping, Trading, Social media, and so on.

How to define a Transaction?

How to define a Transaction?

- Transaction is a collection of operations.

- For example:
  - Moving money from one checkings account to savings account.
  - Buying a product from Amazon.

How to define a Transaction?

- Transaction is a unit of program that reads and/or writes one or more data items.

- A common way to write a transaction in popular DBMS is by placing the body of the transaction between, "**begin transaction**" and "**end transaction**".

How to define a Transaction?

- Transaction is a unit of program that reads and/or writes one or more data items.
- A common way to write a transaction in popular DBMS is by placing the body of the transaction between, "**begin transaction**" and "**end transaction**".
- Also, the reason why transaction is termed as an indivisible unit.
  - It either executes in its entirety or nothing at all.

Definitions and Notations

- **Database:**
  - A collection of data-items or records (**A, B, C, D**, …).

- **Transactions:**
  - A set of read/write operations:
  - **R(A)** → implies Read a data-item/record A.
  - **W(A)** → implies Write a data-item/record A.

ACID Properties for a Transaction

?

ACID Properties for a Transaction

- Each database should provide the following four properties for transactions :

ACID Properties for a Transaction

- Each database should provide the following four properties for transactions :

- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.

ACID Properties for a Transaction

- Each database should provide the following four properties for transactions:
- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

ACID Properties for a Transaction

- Each database should provide the following four properties for transactions:

- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.

- **Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
  - Not referring to database consistency constraints.

- **Isolation**: For every pair of concurrent (executing at the same time) transactions Ti and Tj , either Ti finished execution before Tj started, or Ti started execution after Tj finished.
  - Transactions are unaware of other transactions executing

ACID Properties for a Transaction

- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.

- **Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
  - Not referring to database consistency constraints.

- **Isolation**: For every pair of concurrent (executing at the same time) transactions Ti and Tj , either Ti finished execution before Tj started, or Ti started execution after Tj finished.
  - Transactions are unaware of other transactions executing

- **Durability:** Once a transaction completes successfully, any changes it made to the database should persist, even if there are system failures.

ACID Properties for a Transaction

- **When do ACID properties come into play?**

ACID Properties for a Transaction

- **When do ACID properties come into play?**
  - **Concurrency!**

- In a concurrent system or database, two or more transactions may attempt to fetch the same data.

- **But, why is this an issue?**

ACID Properties for a Transaction

- **When do ACID properties come into play?**
  - **Concurrency!**

- In a concurrent system or database, two or more transactions may attempt to fetch the same data.

- **But, why is this an issue?**
  - Concurrency if not handled well can lead to ACID violations.
  - For instance. → Race conditions!

Isolation

- Users submit transactions.
- Each transaction should execute as if it were running by itself.
- But running one transactions at a time will give poor performance.
- With the prevalence of multi-core architecture, DBMS should take advantage of the multiple cores.
- Concurrency permits interleaving the transaction operations.
  - Interleaving transactions also permits running one transaction when another is waiting for some resource (I/O, user input, or fetching data from disk).
  - Need a mechanism to interleave transactions but make it appear as if they ran one-at-a-time.

Concurrent Transactions

- Before we determine possible inter-leavings, we need to do a bunch of tasks.

- First, we need to know the **possible set of values for A and B** at the end of running these transactions (Say, **initially A = B = 50):**
  - **A + B = 100 * 1.05 = 105**

T1:

read(**A**);
A = **A** − 50;
write(**A**);
read(**B**);
B = **B** + 50;
write(**B**).

T2:

read(**A**);
A = **A** * 1.05;
write(**A**);
read(**B**);
B = **B** * 1.05 ;
write(**B**)

Concurrent Transactions

- Next, we **transform these transactions** to the database perspective.
- Specifically, we need to worry only about read/write operations as only those impact the database.

T1:

read(**A**);
A = **A** − 50;
write(**A**);
read(**B**);
B = **B** + 50;
write(**B**).

T2:

read(**A**);
A = **A** * 0.95;
write(**A**);
read(**B**);
B = **B** * 1.05 ;
write(**B**)

Concurrent Transactions

- We need to worry only about read/write operations as only those impact the database.

- So, we **re-write these transactions** as just a set of read/write operations.

T1:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
End

T2:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
End

Serial Execution

- One legal interleaving is **serial execution**:

- Either T1 → T2, or T2 → T1.
  - Here, the notation T1 → T2 states that first execute transaction T1, and then execute T2.

Serial Execution

T1:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
End

T2:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
End

T1:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
End

T2:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
End

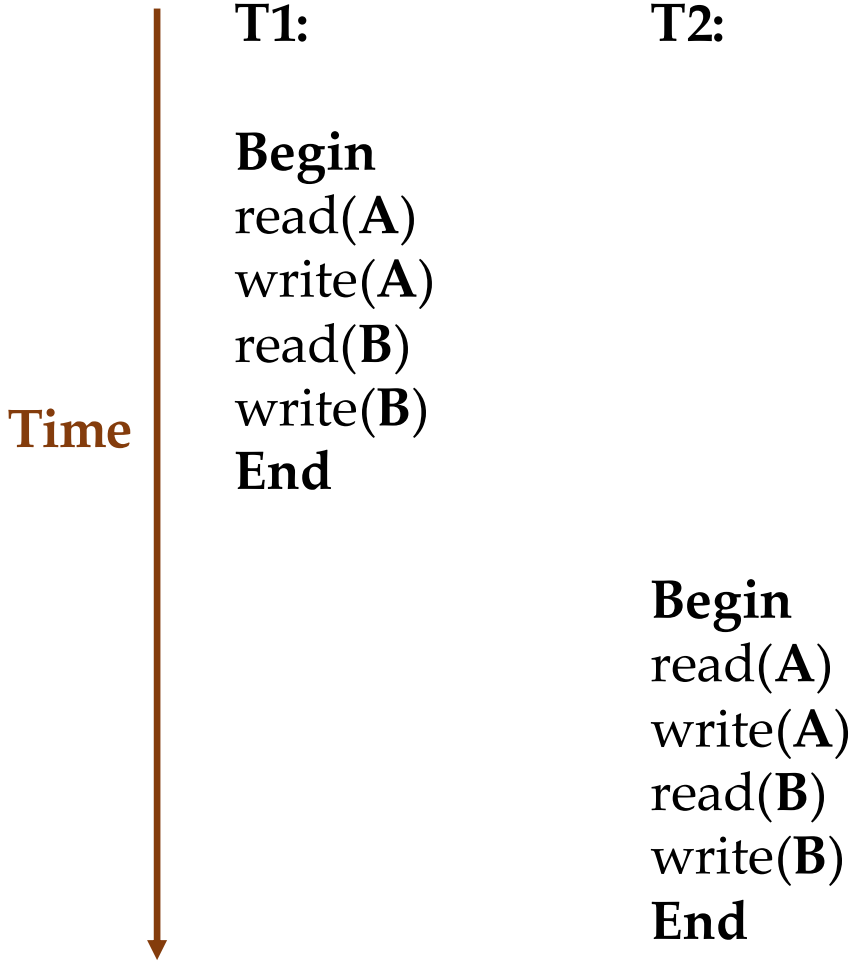
Serial Execution

- **Serial execution** is a legal interleaving:
- It guarantees **isolation**.
- But, serial execution does not take advantage of multi-core architecture.

Schedule

- A **schedule** states the **order of executing** different operations of a transaction.

- The following is a **serial schedule** as it **does not interleave** the operations of different transactions.



Another Interleaving (I)

T1:

Begin
read(**A**)
write(**A**)

read(**B**)
write(**B**)
End

T2:

Begin
read(**A**)
write(**A**)

read(**B**)
write(**B**)
End

Is this a legal interleaving?

Another Interleaving (I)

T1:

Begin
read(**A**)
write(**A**)

read(**B**)
write(**B**)
End

T2:

Begin
read(**A**)
write(**A**)

read(**B**)
write(**B**)
End

Is this a legal interleaving?

Yes!

Another Interleaving (II)

T1:	T2:
Begin	
read(**A**)	
	Begin
	read(**A**)
	write(**A**)
write(**A**)	
read(**B**)	
write(**B**)	
End	
	read(**B**)
	write(**B**)
	End

Is this a legal interleaving?

Another Interleaving (II)

T1:	T2:
Begin	
read(**A**)	
	Begin
	read(**A**)
	write(**A**)
write(**A**)	
read(**B**)	
write(**B**)	
End	
	read(**B**)
	write(**B**)
	End

Is this a legal interleaving?

No!

T1's **write(A)** does not follow its **read(A)**.

Conflicting Transactions

Conflicting Transactions

Two transactions T1 and T2 if they **concurrently access the same variable** and at least one of that access is a **write** operation, then they **conflict**!

Conflicting Transactions

- Interleaving concurrent transactions can lead to the following three anomalies:
  - Read-Write Conflicts (**R-W**)
  - Write-Read Conflicts (**W-R**)
  - Write-Write Conflicts (**W-W**)

Read-Write Conflict

- **Unrepeatable Read?**

Read-Write Conflict

- **Unrepeatable Read**: A transaction gets different values when reading the same object multiple times.

- Say, initially A = 50.

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
	write(**A**)
read(**A**)	**End**
End	

Read-Write Conflict

- **Unrepeatable Read**: A transaction gets different values when reading the same object multiple times.

- Say, initially A = 50.

Output → 50

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
	write(**A**)
read(**A**)	**End**
End	

Read-Write Conflict

- **Unrepeatable Read**: A transaction gets different values when reading the same object multiple times.

- Say, initially A = 50.

Output → 50

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
	write(**A**)
read(**A**)	**End**
End	

Read-Write Conflict

- **Unrepeatable Read**: A transaction gets different values when reading the same object multiple times.

- Say, initially A = 50.

	T1:	**T2:**
	Begin	
A = 100	read(**A**)	
		Begin
		read(**A**)
		write(**A**)
	read(**A**)	**End**
	End	

Read-Write Conflict

- **Unrepeatable Read**: A transaction gets different values when reading the same object multiple times.

- Say, initially A = 50.

Output → 100

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
	write(**A**)
read(**A**)	**End**
End	

Write-Read Conflict

- **Dirty Read?**

Write-Read Conflict

- **Dirty Read**: One transaction reads data written by another transaction that has not committed yet.

- Say, initially A = 50.

T1:	**T2:**
Begin	
read(**A**)	
write(**A**)	**Begin**
	read(**A**)
	write(**A**)
	End
Abort	

Write-Read Conflict

- **Dirty Read**: One transaction reads data written by another transaction that has not committed yet.

- Say, initially A = 50.

Output → 50

T1:	**T2:**
Begin	
read(**A**)	
write(**A**)	**Begin**
	read(**A**)
	write(**A**)
	End
Abort	

Write-Read Conflict

- **Dirty Read**: One transaction reads data written by another transaction that has not committed yet.

- Say, initially A = 50.

	T1:	**T2:**
	Begin	
A = 100	read(**A**)	
	write(**A**)	**Begin**
		read(**A**)
		write(**A**)
		End
	Abort	

Write-Read Conflict

- **Dirty Read**: One transaction reads data written by another transaction that has not committed yet.

- Say, initially A = 50.

Output → 100

T1:	**T2:**
Begin	
read(**A**)	
write(**A**)	**Begin**
	read(**A**)
	write(**A**)
	End
Abort	

Write-Read Conflict

- **Dirty Read**: One transaction reads data written by another transaction that has not committed yet.

- Say, initially A = 50.

Output → 150

T1:	**T2:**
Begin	
read(**A**)	
write(**A**)	**Begin**
	read(**A**)
	write(**A**)
	End
Abort	

Write-Read Conflict

- **Dirty Read**: One transaction reads data written by another transaction that has not committed yet.

- Say, initially A = 50.

Transaction T1 has to be aborted

T1:	**T2:**
Begin	
read(**A**)	
write(**A**)	
	Begin
	read(**A**)
	write(**A**)
	End
Abort	

Write-Write Conflict

- **Lost Update?**

Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.

- Say, initially A = 50.

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
write(**A**)	
	write(**A**)
	write(**B**)
	End
write(**B**)	
End	

Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.

- Say, initially A = 50.

Output → 50

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
write(**A**)	
	write(**A**)
	write(**B**)
	End
write(**B**)	
End	

Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.

- Say, initially A = 50.

Output → 50

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
write(**A**)	
	write(**A**)
	write(**B**)
	End
write(**B**)	
End	

Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.

- Say, initially A = 50.

A = 100

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
write(**A**)	
	write(**A**)
	write(**B**)
	End
write(**B**)	
End	

Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.

- Say, initially A = 50.

A = 100

Previous update missed!

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
write(**A**)	
	write(**A**)
	write(**B**)
	End
write(**B**)	
End	

Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.

- Say, initially A = 50.

A = 100

B = 100

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
write(**A**)	
	write(**A**)
	write(**B**)
	End
write(**B**)	
End	

Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.

- Say, initially A = 50.

A = 100

B = 150

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
write(**A**)	
	write(**A**)
	write(**B**)
	End
write(**B**)	
End	

Write-Write Conflict

- **Lost Update** → One transaction overwrites uncommitted data from another uncommitted transaction.

- Say, initially A = 50.

A = 100

B = 150

T1:	**T2:**
Begin	
read(**A**)	
	Begin
	read(**A**)
write(**A**)	
	write(**A**)
	write(**B**)
	End
write(**B**)	
End	

This leads to unexpected results of 100,150 when it should have been 150, 150.

Isolation Support: Concurrency Control

- A **concurrency control protocol** lays down the mechanism for the DBMS to decide a legal/valid schedule of transactions.

- What are the **types** of concurrency control protocols?

Isolation Support: Concurrency Control

- A concurrency control protocol lays down the mechanism for the DBMS to decide legal/valid schedule of transactions.
- What are the types of concurrency control protocols?
- **Pessimistic:** Prevent problems from arising in the first place.

Isolation Support: Concurrency Control

- A concurrency control protocol lays down the mechanism for the DBMS to decide legal/valid schedule of transactions.
- What are the types of concurrency control protocols?
- **Pessimistic:** Prevent problems from arising in the first place.
- **Optimistic:** Assume that conflicts are rare; deal with them after they occur.

Isolation Support: Concurrency Control

- A concurrency control protocol lays down the mechanism for the DBMS to decide legal/valid schedule of transactions.
- What are the types of concurrency control protocols?
- **Pessimistic:** Prevent problems from arising in the first place.
- **Optimistic:** Assume that conflicts are rare; deal with them after they occur.
- **But, how does a concurrency control protocol determine a valid schedule?**

Serializable Schedules

- **Serial Schedule** → A schedule that does not interleave the operations of different transactions.

- **Equivalent Schedule** → For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.

- **Serializable Schedule?**

Serializable Schedules

- **Serial Schedule** → A schedule that does not interleave the operations of different transactions.

- **Equivalent Schedule** → For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.

- **Serializable Schedule** → A schedule that is equivalent to some serial execution of the transactions (serial schedule).

Serializable Schedules

- **Serial Schedule** → A schedule that does not interleave the operations of different transactions.
- **Equivalent Schedule** → For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- **Serializable Schedule** → A schedule that is equivalent to some serial execution of the transactions (serial schedule).
- If each transaction preserves consistency, then the corresponding serializable schedule preserves consistency!

Isolation Levels vs. Consistency Levels

Isolation Levels	Consistency Levels

Isolation Levels vs. Consistency Levels

Isolation Levels	Consistency Levels
• **Correspond** to the **I in ACID**. • **Database isolation** is the ability of a database to allow a transaction to execute as if there are no other concurrently running transactions. • Greater the guaranteed isolation among the transactions, lesser the system performance. • **Isolation levels** trade off isolation guarantees for improved performance.	

Isolation Levels vs. Consistency Levels

Isolation Levels	Consistency Levels
• **Correspond** to the **I in ACID**.	• **Do not correspond to C in ACID**.
• **Database isolation** is the ability of a database to allow a transaction to execute as if there are no other concurrently running transactions.	• Unlike the C in ACID, the **database consistency** refers to the rules that make a **concurrent**, **distributed system** appear as a **single-threaded, centralized system**.
• Greater the guaranteed isolation among the transactions, lesser the system performance.	• **Reads** at a particular point in time **must reflect the most recently completed write** (in real-time) of that data item, no matter which server processed that write.
• **Isolation levels** trade off isolation guarantees for improved performance.	• **Consistency levels** trade off read results for improved performance.

Isolation Levels vs. Consistency Levels

- More simply said:
  - Whenever you talk about transaction isolation, you will be talking about isolation levels.
  - Whenever you talk about individual operations like read/write, you will talk about consistency levels.

Serializable Isolation Level

- Also known as serializability.
- The ability of a DBMS to run transactions in parallel, but in a way that they are running, **serially**, that is, **one after another**.
- Thus, if the DBMS can ensure a serializable schedule for a set of transactions, then we say that the DBMS is offering serializability or serializable isolation level.

Levels of Serializability

- **Conflict Serializability**
  - Most DBMS try to support this.

- **View Serializability**
  - No DBMS can do this!

Dependency Graphs

- **Help to determine the level of serializability among other things.**
- **How can you create a dependency graph?**

Dependency Graphs

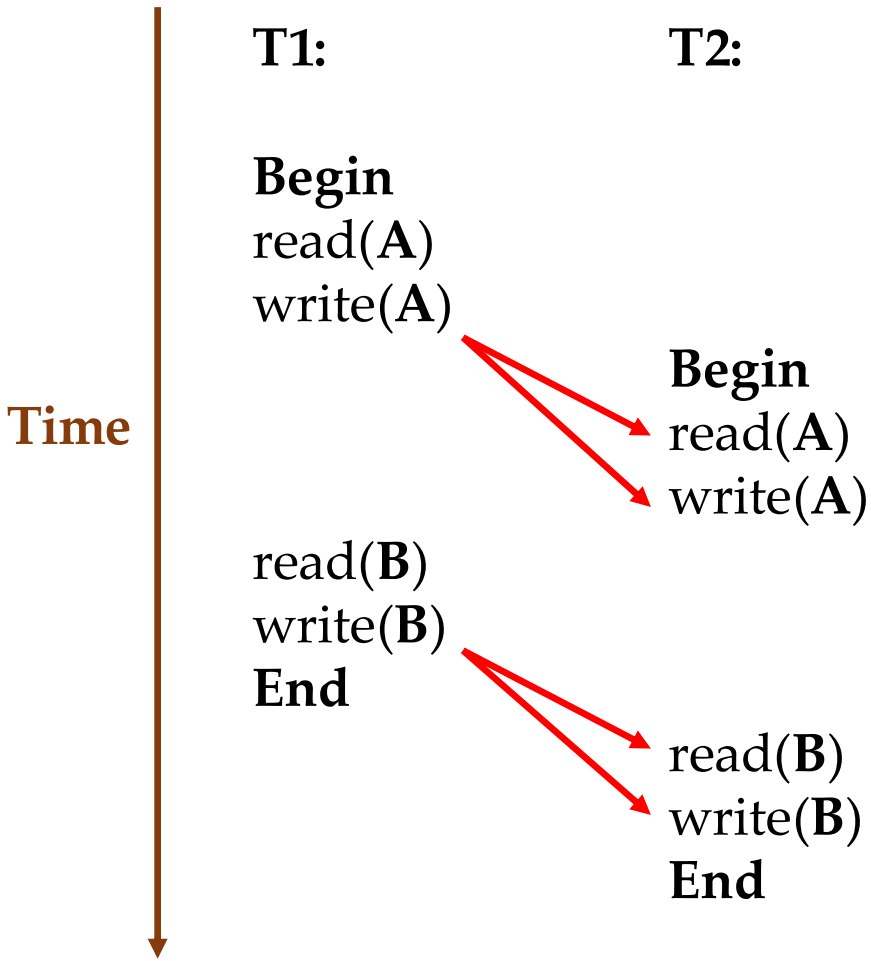
- Help to determine the level of serializability among other things.
- How can you create a dependency graph?
- One node per transaction.
- **Add an edge from transaction Ti to transaction Tj** if you the following are met:
  - An operation **Oi of Ti** conflicts with an operation **Oj of Tj**.
  - **Oi** appears earlier in the schedule than **Oj**.
- Also known as **precedence graph**.

Dependency Graphs Example I

Time

T1:	T2:
Begin	
read(**A**)	
write(**A**)	
	Begin
	read(**A**)
	write(**A**)
read(**B**)	
write(**B**)	
End	
	read(**B**)
	write(**B**)
	End

Dependency Graphs Example I



T2 depends on T1

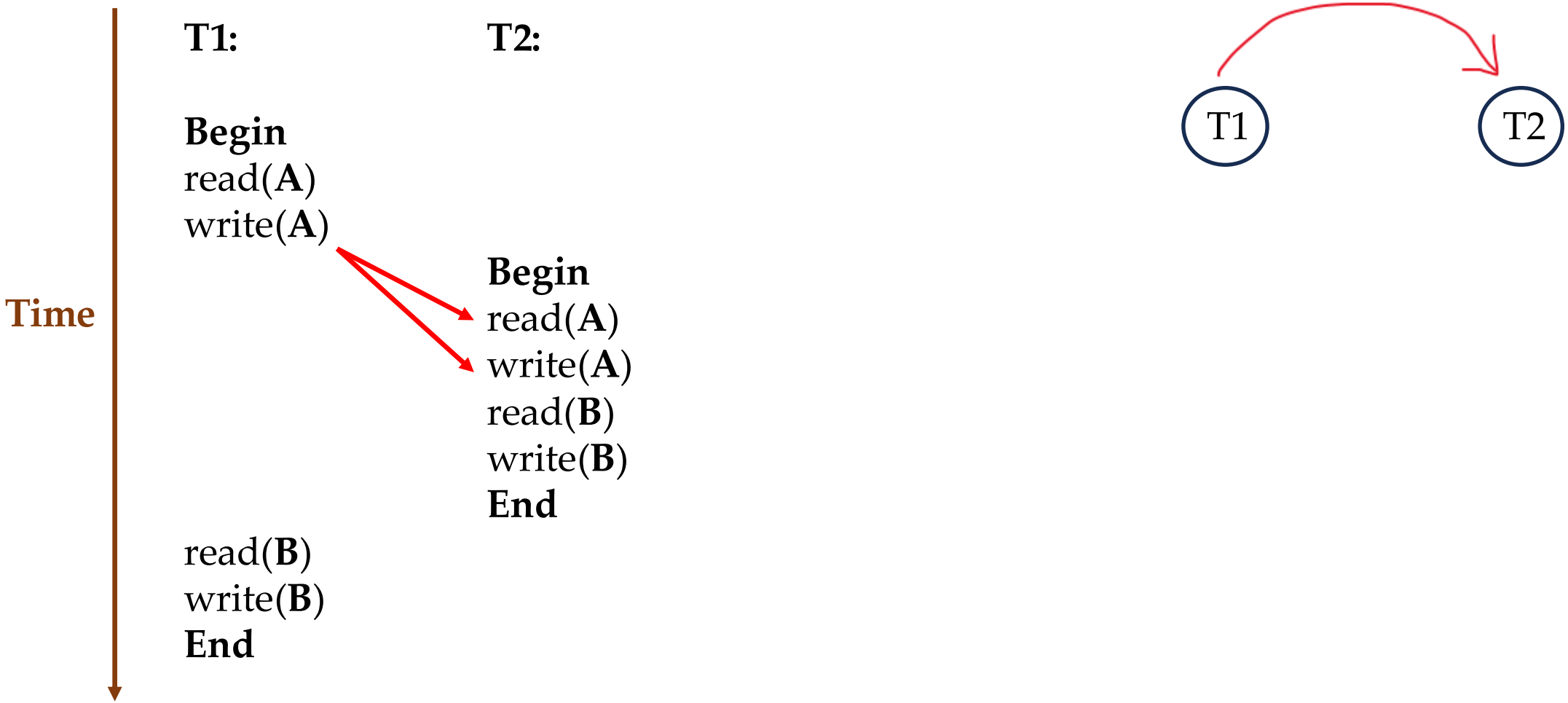
W-R and W-W Conflicts

Dependency Graphs Example II

Time

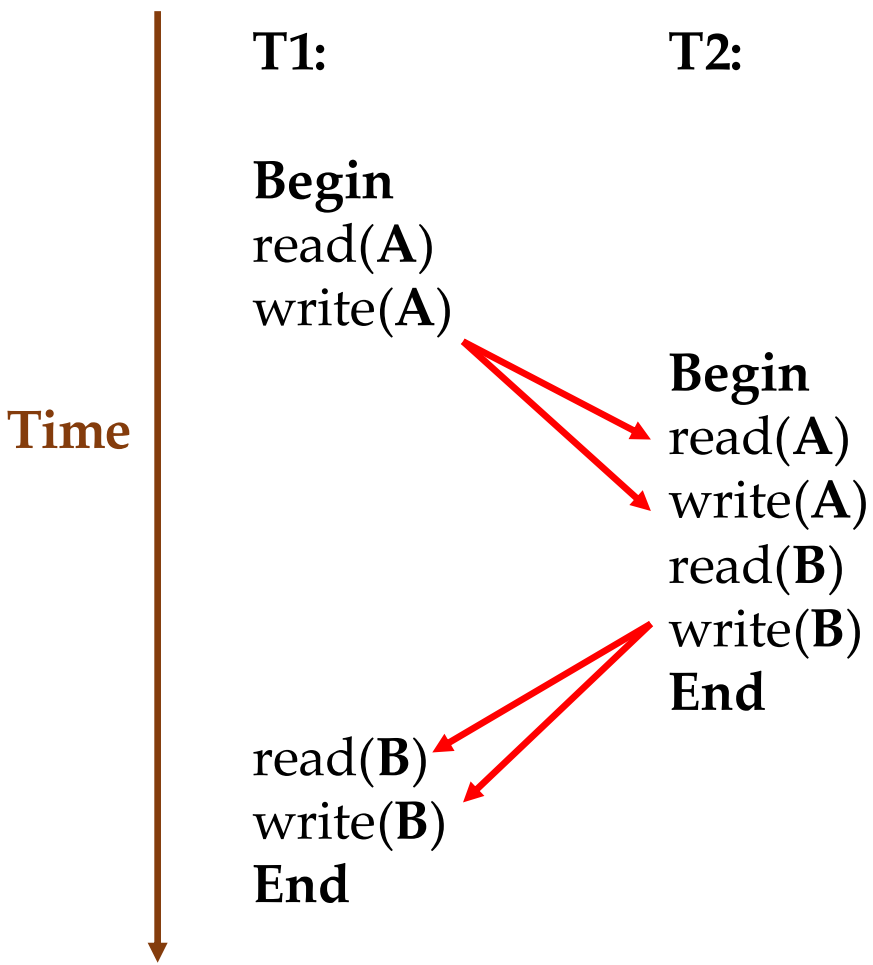
T1:	T2:
Begin	
read(**A**)	
write(**A**)	
	Begin
	read(**A**)
	write(**A**)
	read(**B**)
	write(**B**)
	End
read(**B**)	
write(**B**)	
End	

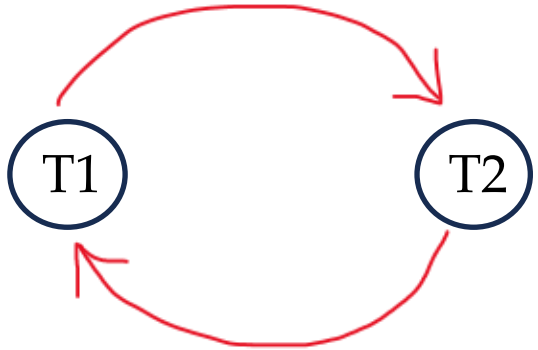
Dependency Graphs Example II



W-R and W-W Conflicts

Dependency Graphs Example II





Conflict Cycle!

W-R and W-W Conflicts

How to Guarantee Serializability?

- If we know the full schedule (all the transactions that are part of the schedule) ahead of time, we can try to create a serializable schedule.
- Unfortunately, this is not a practical expectation.
- How to guarantee serializability?

Locks

- Use **Locks** to restrict access to database records.
- **Lock Manager** → Stores and grants access to Locks.

Locks

- Use **Locks** to restrict access to database records.
- **Lock Manager** → Stores and grants access to Locks.

Type

- **Shared Lock** → A shared lock on a data-item **D permits** concurrent access to the data-item **D** by multiple transactions. **Good for Reads**!
- **Exclusive Lock** → An exclusive lock on a data-item **D disallows** concurrent access to the data-item **D** by multiple transactions (only one transaction at a time). **Good for Writes**!

Granularity

- Granularity defines the **level** at which a transaction acquires a lock. For example: a lock can be acquired for a full transaction or before access to a specific data-item.

Lock Compatibility Matrix

If a transaction Ti holds a S-Lock/X-Lock can another transaction acquire a S-Lock/X-Lock.

	Shared Lock (S-Lock)	Exclusive Lock (X-Lock)
Shared Lock (S-Lock)	✓	✗
Exclusive Lock (X-Lock)	✗	✗

Transaction Lock Phases

- First, each transaction **determines the type of lock** (S-Lock or X-Lock) it wants.

Transaction Lock Phases

- First, each transaction **determines the type of lock** (S-Lock or X-Lock) it wants.

- Next, it **requests the specific type lock for a data-item** from Lock Manager.

Transaction Lock Phases

- First, each transaction **determines the type of lock** (S-Lock or X-Lock) it wants.
- Next, it **requests the specific type lock for a data-item** from Lock Manager.
- **Two Possible Cases**:
  - **Transaction gets the requested lock for the data-item**
  - **Request Denied**

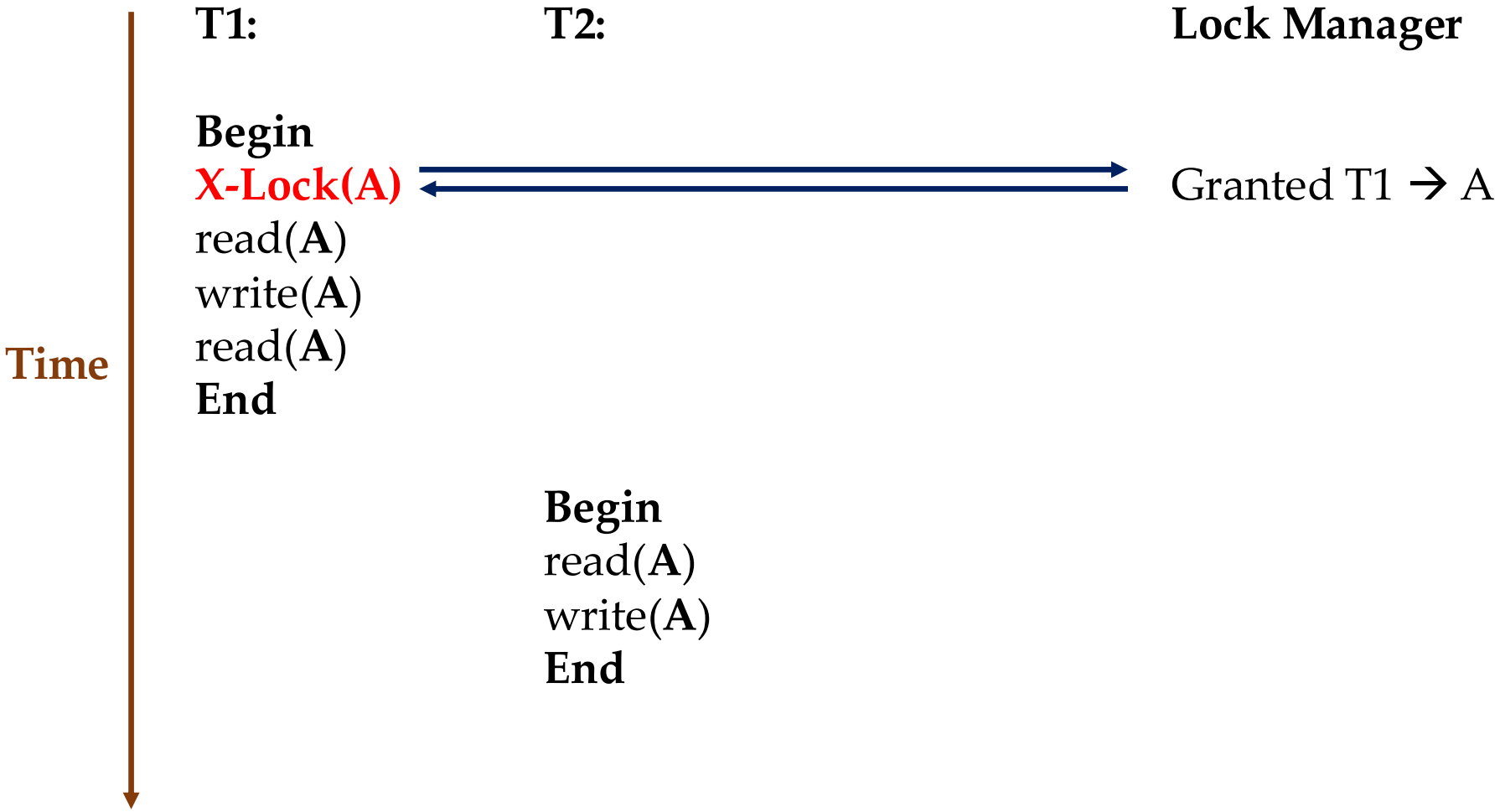
Transaction Lock Phases

- First, each transaction **determines the type of lock** (S-Lock or X-Lock) it wants.
- Next, it **requests the specific type lock for a data-item** from Lock Manager.
- **Two Possible Cases**:
  - **Transaction gets the requested lock for the data-item**
    - **Locks** the data-item.
    - Completes the desired task.
    - **Unlocks** the data-item and **releases** the lock back to Lock Manager.
  - **Request Denied**

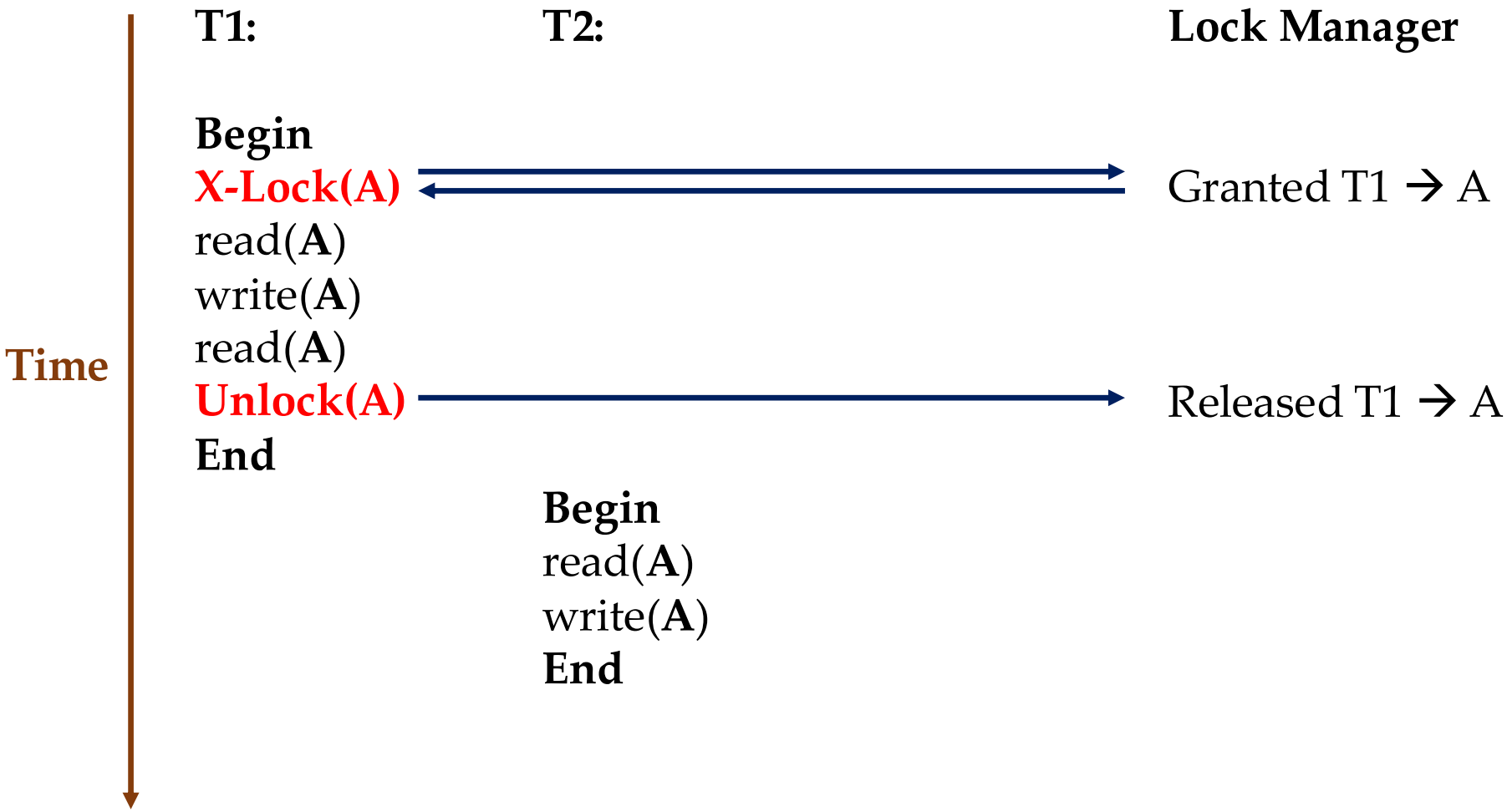
Locking Example I

T1:	T2:	Lock Manager
Begin		
read(**A**)		
write(**A**)		
read(**A**)		
End		
	Begin	
	read(**A**)	
	write(**A**)	
	End	

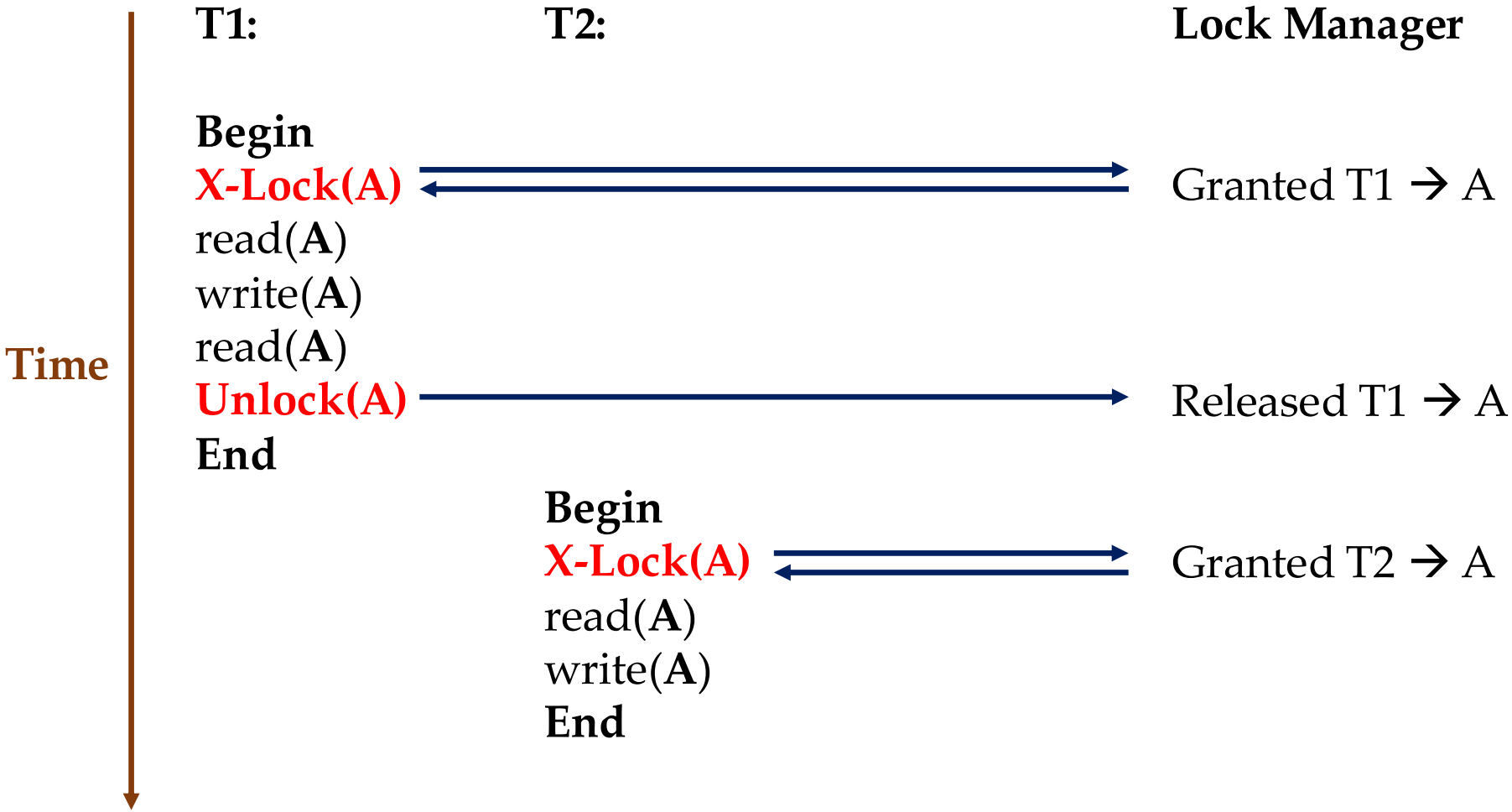
Locking Example I



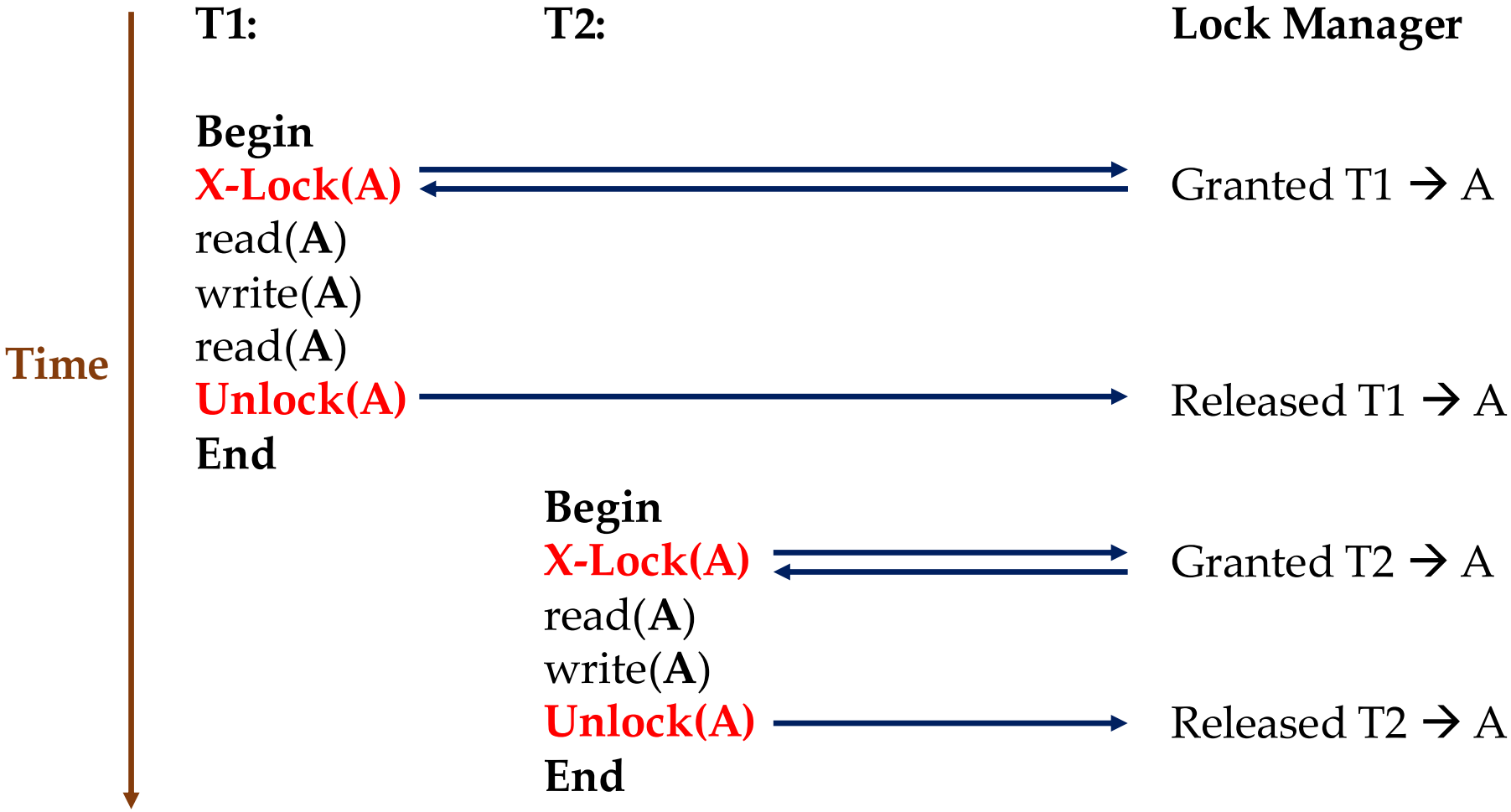
Locking Example I



Locking Example I



Locking Example I



Locking Example II

T1:	T2:	Lock Manager
Begin		
read(**A**)		
write(**A**)		
	Begin	
	read(**A**)	
	write(**A**)	
	End	
read(**A**)		
End		

Locking Example II

Time

T1:	T2:	Lock Manager
Begin		
X-Lock(A)		Granted T1 → A
read(**A**)		
write(**A**)		
	Begin	
	read(**A**)	
	write(**A**)	
	End	
read(**A**)		
End		

Locking Example II

T1:	T2:	Lock Manager
Begin		
X-Lock(A)		Granted T1 → A
read(**A**)		
write(**A**)		
Unlock(A)		Released T1 → A
	Begin	
	read(**A**)	
	write(**A**)	
	End	
read(**A**)		
End		

Locking Example II

Time

T1:	T2:	Lock Manager
Begin		
X-Lock(A)		Granted T1 → A
read(**A**)		
write(**A**)		
Unlock(A)		Released T1 → A
	Begin	
	X-Lock(A)	Granted T2 → A
	read(**A**)	
	write(**A**)	
	End	
read(**A**)		
End		

Locking Example II

Time

T1:	T2:	Lock Manager
Begin		
X-Lock(A)		Granted T1 → A
read(**A**)		
write(**A**)		
Unlock(A)		Released T1 → A
	Begin	
	X-Lock(A)	Granted T2 → A
	read(**A**)	
	write(**A**)	
	Unlock(A)	Released T2 → A
	End	
read(**A**)		
End		

Locking Example II

Time

T1:	T2:	Lock Manager
Begin		
X-Lock(A)		Granted T1 → A
read(**A**)		
write(**A**)		
Unlock(A)		Released T1 → A
	Begin	
	X-Lock(A)	Granted T2 → A
	read(**A**)	
	write(**A**)	
	Unlock(A)	Released T2 → A
	End	
S-Lock(A)		Granted T1 → A
read(**A**)		
End		

Locking Example II

Time

T1:	T2:	Lock Manager
Begin		
X-Lock(A)		Granted T1 → A
read(**A**)		
write(**A**)		
Unlock(A)		Released T1 → A
	Begin	
	X-Lock(A)	Granted T2 → A
	read(**A**)	
	write(**A**)	
	Unlock(A)	Released T2 → A
	End	
S-Lock(A)		Granted T1 → A
read(**A**)		
Unlock(A)		Released T1 → A
End		

Locking Example II

Time

T1:	T2:	Lock Manager
Begin		
X-Lock(A)		Granted T1 → A
read(**A**)		
write(**A**)		
Unlock(A)		Released T1 → A
	Begin	
	X-Lock(A)	Granted T2 → A
	read(**A**)	
	write(**A**)	
	Unlock(A)	Released T2 → A
	End	
S-Lock(A)		Granted T1 → A
read(**A**)		
Unlock(A)		Released T1 → A
End		

Is this serializable?

Did locking help?

Locking Example II

Time

T1:	T2:	Lock Manager
Begin		
X-Lock(A)		Granted T1 → A
read(**A**)		
write(**A**)		
Unlock(A)		Released T1 → A
	Begin	
	X-Lock(A)	Granted T2 → A
	read(**A**)	
	write(**A**)	
	Unlock(A)	Released T2 → A
	End	
S-Lock(A)		Granted T1 → A
read(**A**)		
Unlock(A)		Released T1 → A
End		

Is this serializable?

Did locking help?

No!