Large Scale Systems
CS 410 / 510

Lecture 4:

Distributed Transaction Processing

Suyash Gupta

Assistant Professor

Distopia Labs and ONRG

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

- Last class we looked at:

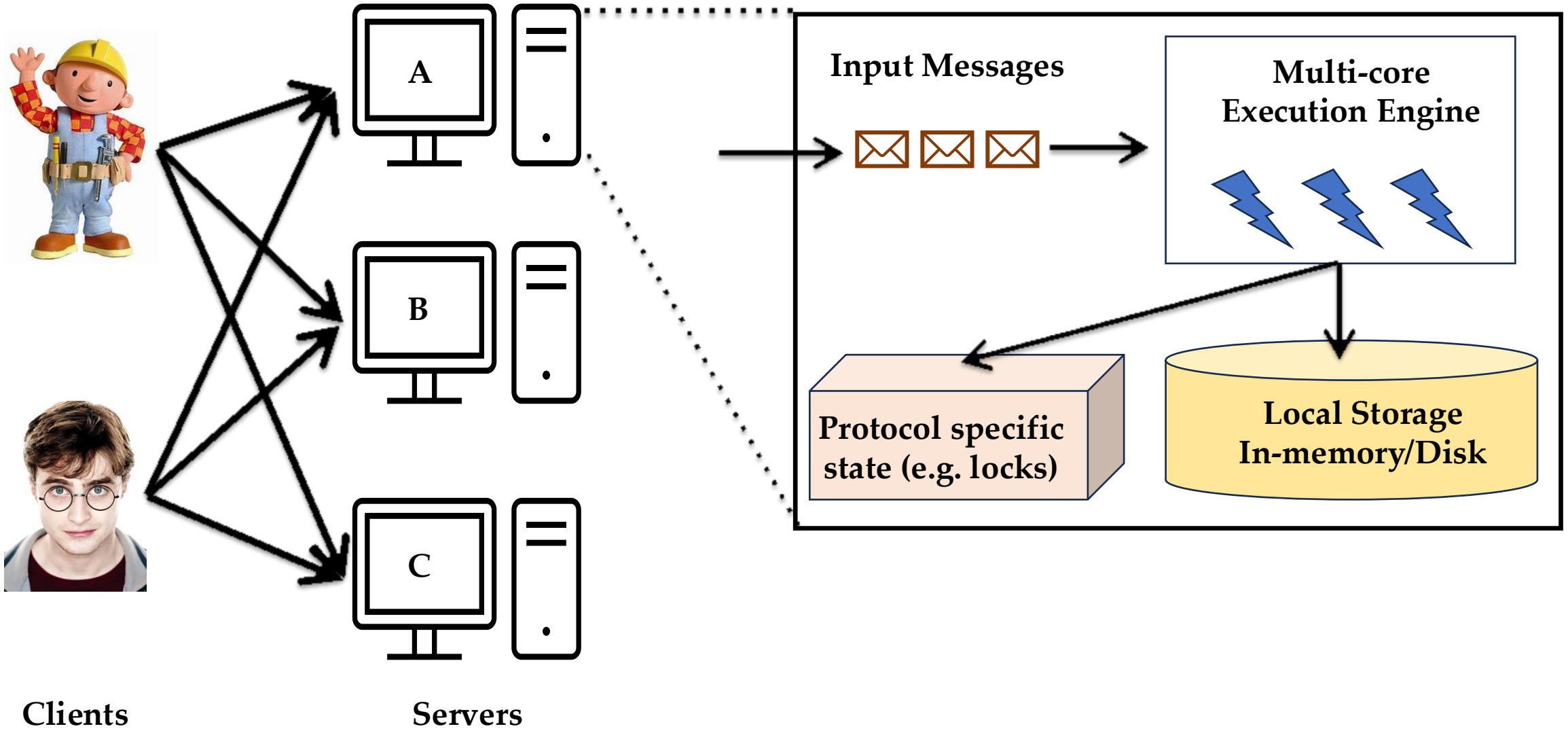
- Transactions
- ACID
- Serializability
- Concurrency Control

What is a Distributed Transaction

What is a Distributed Transaction

- A transaction that accesses data from multiple machines or nodes.
- Recall that we discussed Sharded or Partitioned systems.
  - Each shard/partition has access to a unique set of data items.
- A distributed transaction may require access to one or more shards.

Sharded Distributed System



Sharded Distributed System

- A sharded distributed system/database receives two types of transactions:
  - **Intra-Shard Transaction**: Transaction that accesses only one shard/partition.
  - **Inter-Shard Transaction**: Transaction that accesses multiple shards/partitions.

Sharded Distributed System

- A sharded distributed system/database receives two types of transactions:
  - **Intra-Shard Transaction**: Transaction that accesses only one shard/partition.
  - **Inter-Shard Transaction**: Transaction that accesses multiple shards/partitions.
- A distributed system that receives a large number of intra-shard transactions can yield higher performance than a single monolithic database. **Why?**

Sharded Distributed System

- A sharded distributed system/database receives two types of transactions:
  - **Intra-Shard Transaction**: Transaction that accesses only one shard/partition.
  - **Inter-Shard Transaction**: Transaction that accesses multiple shards/partitions.
- A distributed system that receives a large number of intra-shard transactions can yield higher performance than a single monolithic database. **Why?**
  - Because each partition can work independently; equivalent to multiple monolithic databases running parallel!

Sharded Distributed System

- A sharded distributed system/database receives two types of transactions:
  - **Intra-Shard Transaction**: Transaction that accesses only one shard/partition.
  - **Inter-Shard Transaction**: Transaction that accesses multiple shards/partitions.
- A distributed system that receives a large number of intra-shard transactions can yield higher performance than a single monolithic database. **Why?**
  - Because each partition can work independently; equivalent to multiple monolithic databases running parallel!
- **Real Challenge** → Inter-shard transactions as it requires coordination among multiple shards.

Common Terminology

- **Local Partition** → The one that receives the transaction.
- **Remote Partition** → The one that has access to other required data.

Concurrency Control protocol: Two-Phase Locking

Two-Phase Locking

- Two-phase locking (2PL) protocol determines whether a transaction can access an object in the database at runtime.

- The 2PL protocol does not need to know all the queries that a transaction will execute ahead of time.

Two-Phase Locking

- **Two phases of 2PL**.

- **Growing Phase**:
  - Each transaction requests the locks that it needs from the Lock manager.
  - The lock manager grants/denies lock requests.

- **Shrinking Phase**:
  - A transaction is allowed to only release/downgrade locks that it previously acquired.
  - It cannot acquire new locks.
  - A transaction attempting to acquire a lock after releasing any lock is a violation!

2PL Example I

T1:

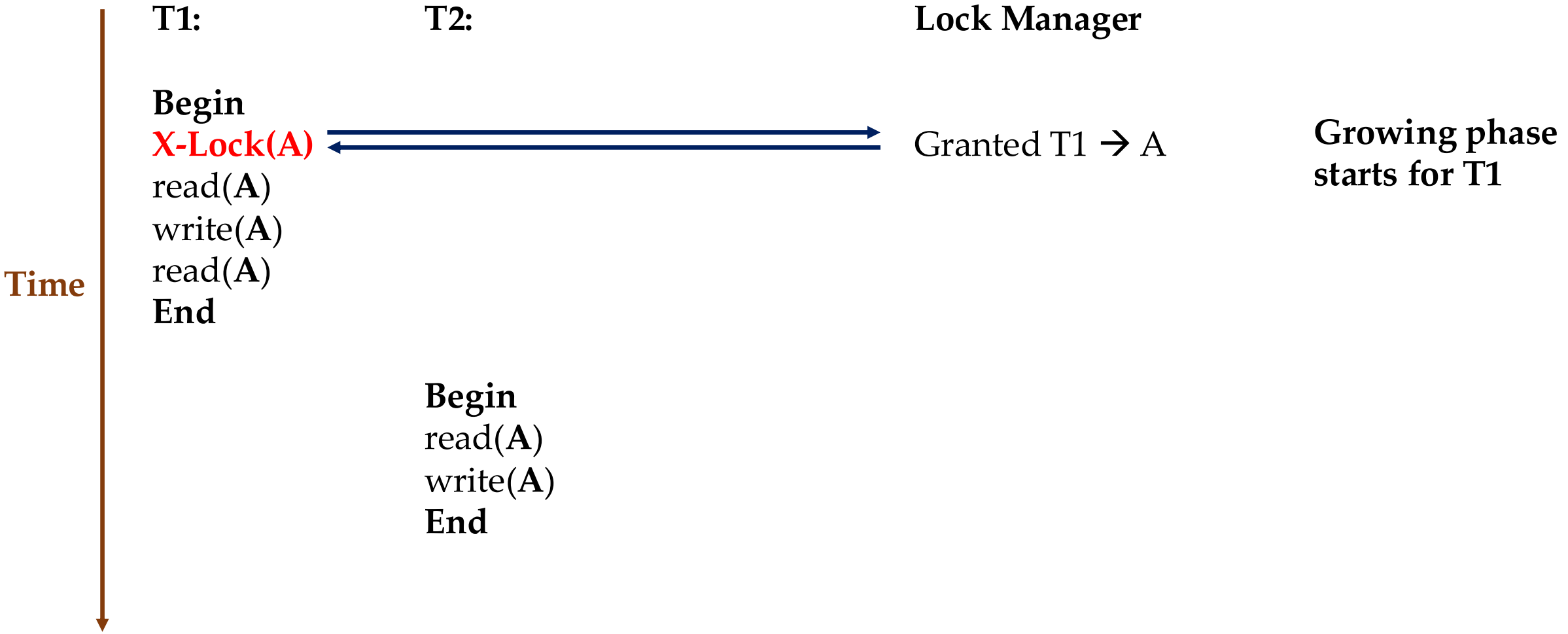
Begin
read(**A**)
write(**A**)
read(**A**)
End

T2:

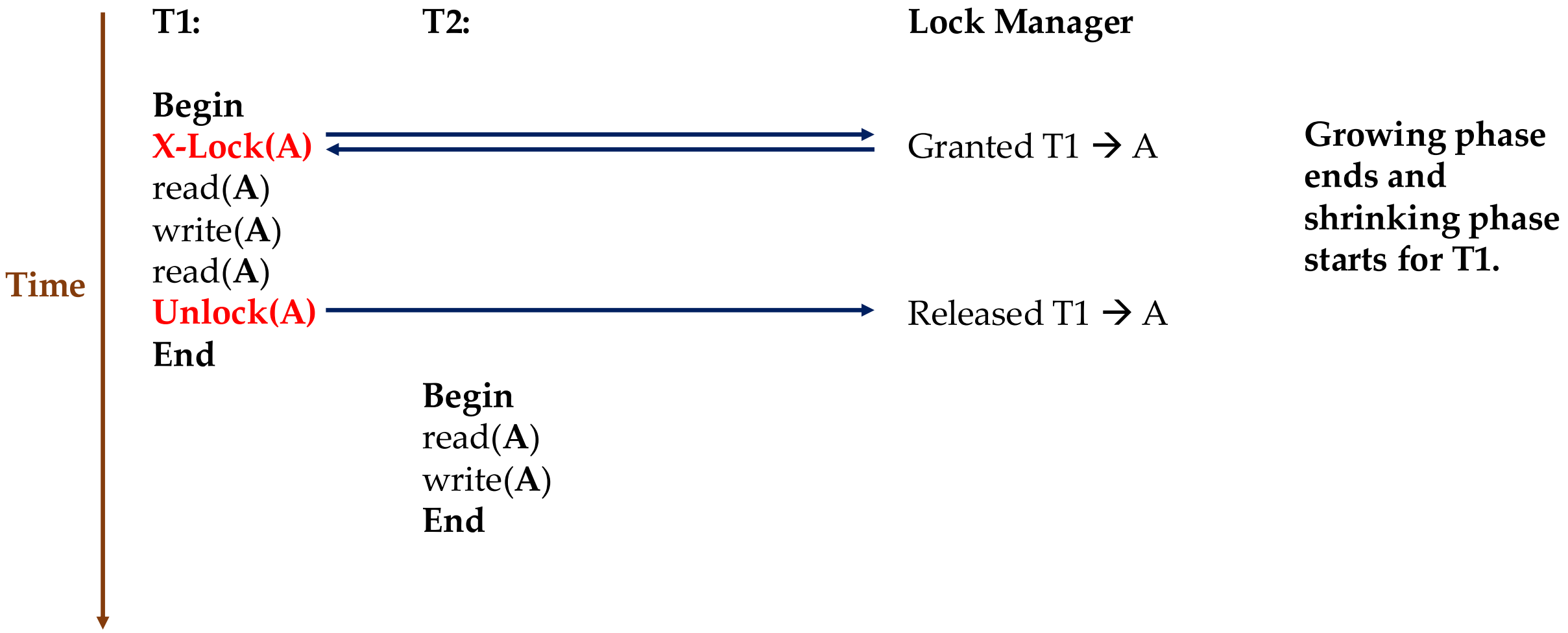
Begin
read(**A**)
write(**A**)
End

Lock Manager

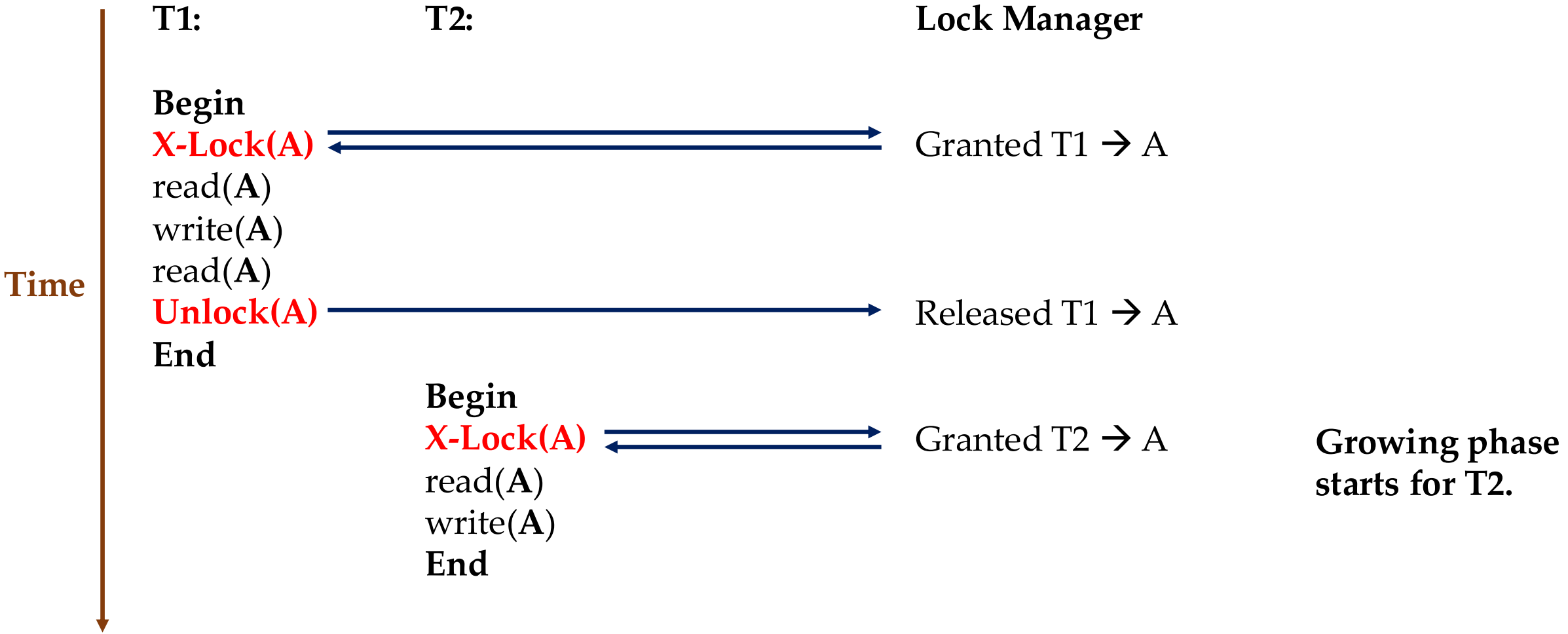
2PL Example I



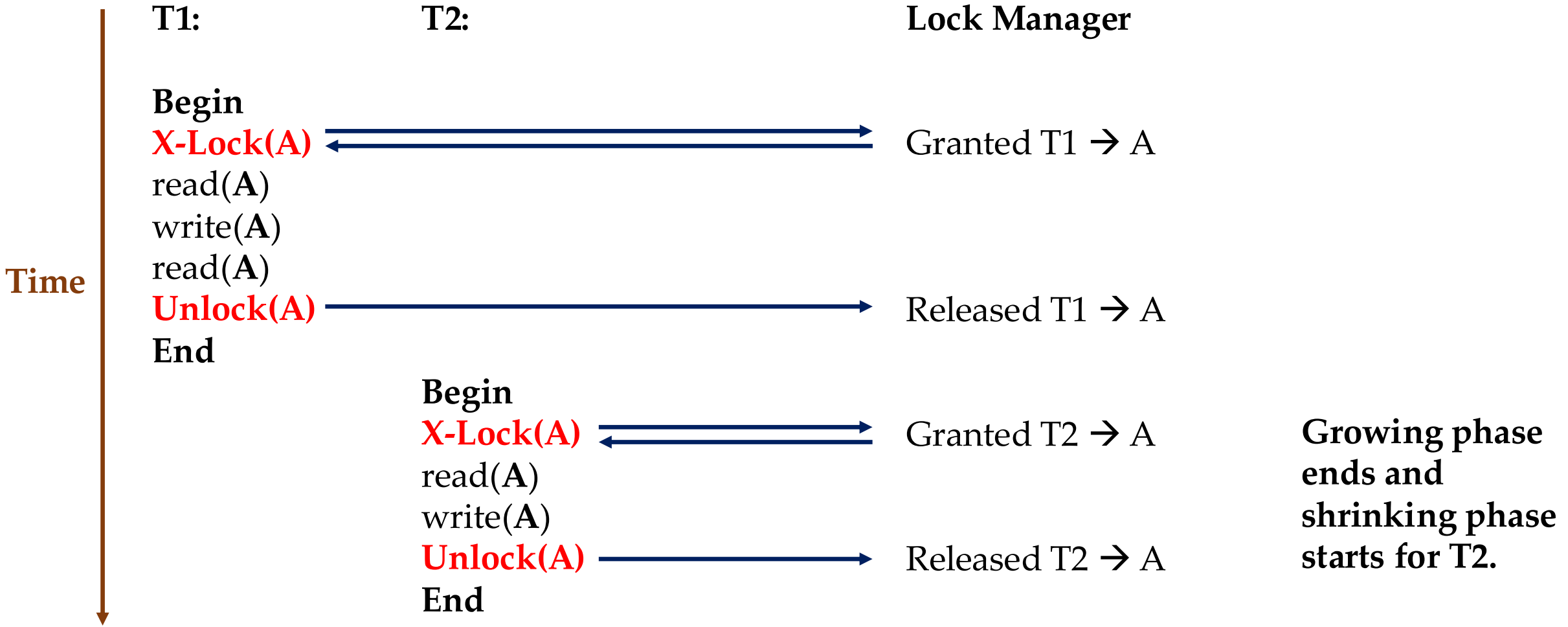
2PL Example I



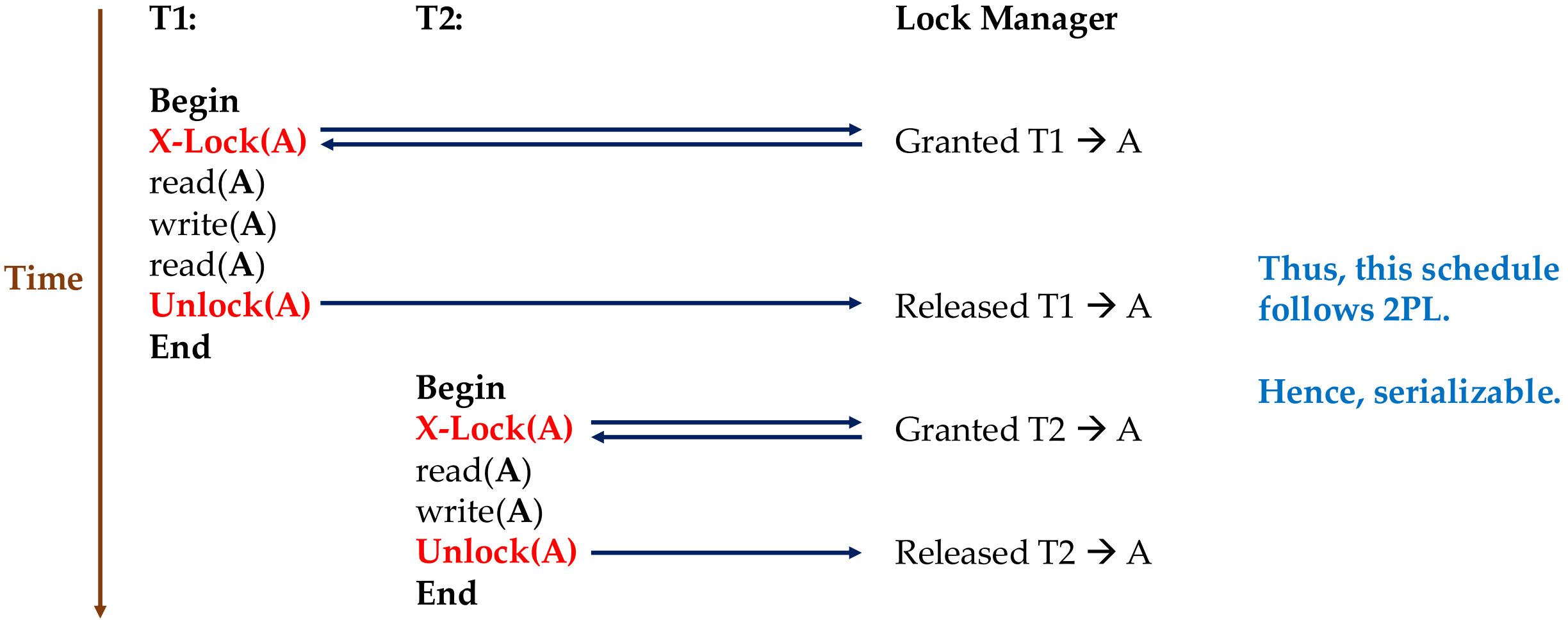
2PL Example I



2PL Example I



2PL Example I



2PL Example II

T1:	T2:	Lock Manager
Begin		
read(**A**)		
write(**A**)		
	Begin	**Let's try to force 2PL on this schedule.**
	read(**A**)	
	write(**A**)	
	End	
read(**A**)		
End		

2PL Example II

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

2PL Example II

T1:	T2:	Lock Manager	
Begin			
X-Lock(A)		Granted T1 → A	**Growing phase starts for T1**
read(**A**)			
write(**A**)			
	Begin		
	X-Lock(A)	Request denied	
read(**A**)			
End			
	read(**A**)		
	write(**A**)		
	End		

2PL Example II

T1:	T2:	Lock Manager	
Begin			
X-Lock(A)		Granted T1 → A	**Shrinking phase starts for T1**
read(**A**)			
write(**A**)			
	Begin		
	X-Lock(A)	Request denied	
read(**A**)			
Unlock(A)		Released T1 → A	
End			
	read(**A**)		
	write(**A**)		
	End		

2PL Example II

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

Growing phase starts for T2

2PL Example II

T1:	T2:	Lock Manager	
Begin			
X-Lock(A)		Granted T1 → A	
read(**A**)			
write(**A**)			
	Begin		
	X-Lock(A)	Request denied T2	
read(**A**)			
Unlock(A)		Released T1 → A	**Shrinking phase starts for T2**
End			
	X-Lock(A)	Granted T2 → A	
	read(**A**)		
	write(**A**)		
	Unlock(A)	Released T2 → A	
	End		

Two-Phase Locking

- **2PL can guarantee conflict serializability**.

- But, what are the major challenges with 2PL?

Two-Phase Locking

- **2PL can guarantee conflict serializability**.
- But, what are the major challenges with 2PL?
- **Cascade aborts** → Aborting one transaction causes aborting all dependent transactions.
- **Deadlocks** → Two transactions waiting on resources held by each other.

Strong Strict Two-Phase Locking

Prevents Cascade Aborts

Strong Strict Two-Phase Locking

- A transaction is only allowed to release locks after it has ended (i.e., committed or aborted).

- Stricter than standard 2PL.
  - Smaller subset of schedules than standard 2PL allowed.

- Advantages:
  - No cascade aborts.
  - Aborted transactions can simply be undone!

Example

- Assume, the following two transactions, and initially **A = B = 1000**.

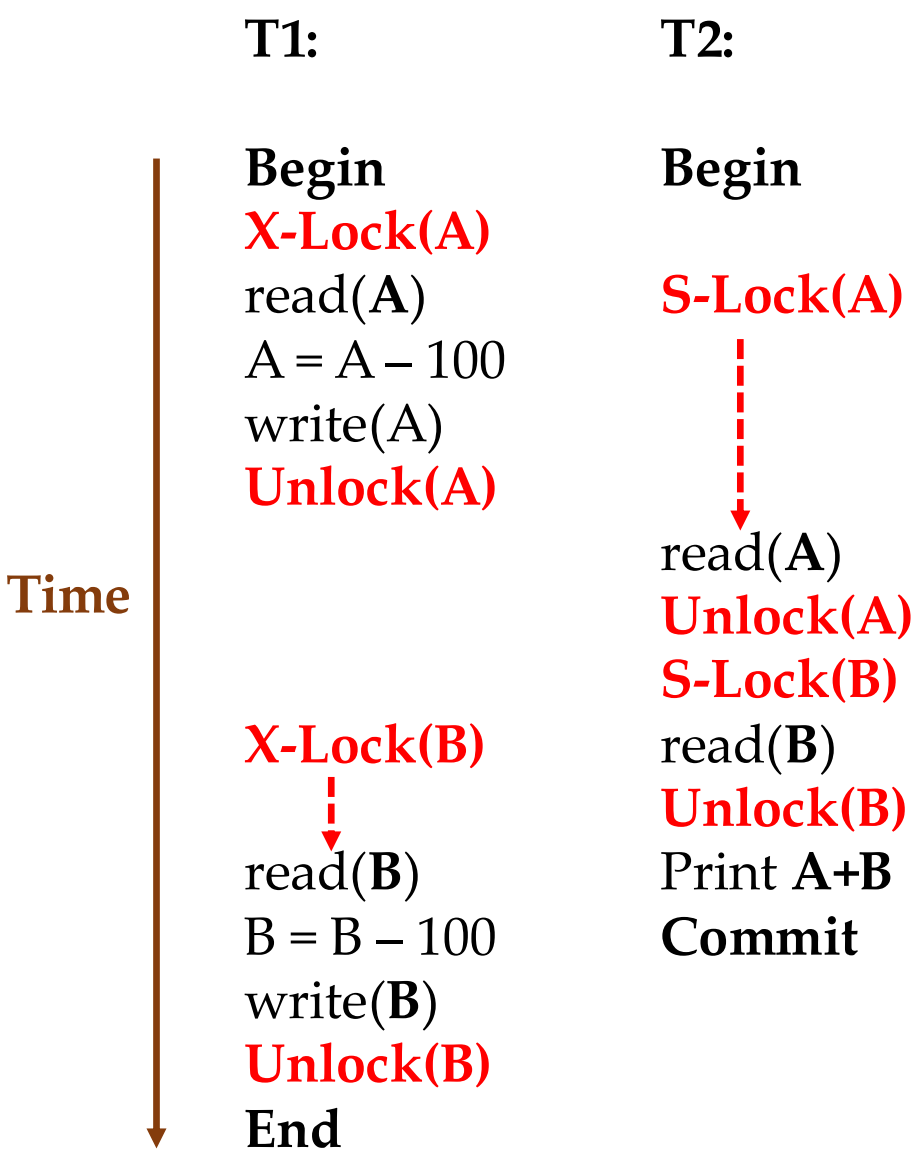
T1:

Begin
A = **A** − 100;
B = **B** + 100;
Commit

T2:

Begin
Print **A + B**
Commit

Non 2PL



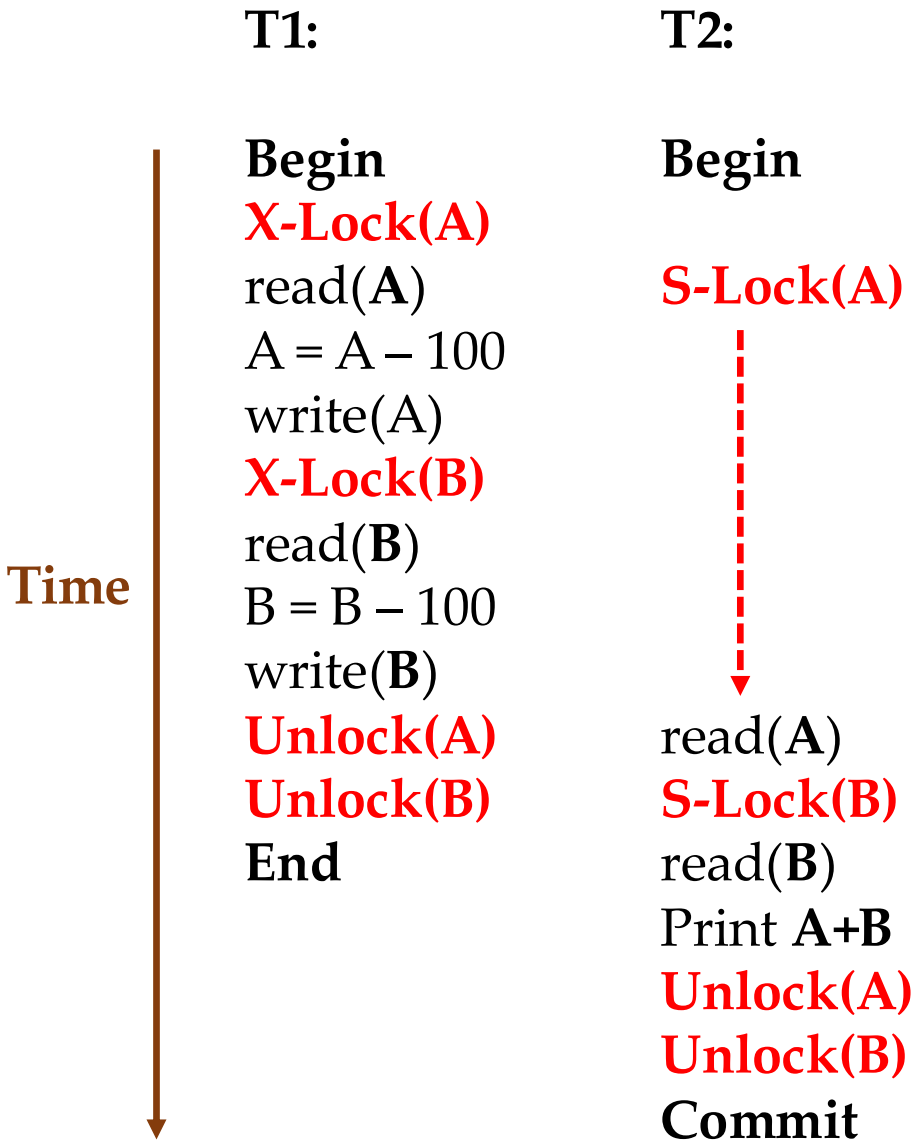
This can be made into a serializable schedule (not yet).

2PL

T1:	T2:
Begin	**Begin**
X-Lock(A)	
read(**A**)	**S-Lock(A)**
A = A – 100	
write(A)	read(**A**)
X-Lock(B)	**S-Lock(B)**
Unlock(A)	
read(**B**)	read(**B**)
B = B – 100	**Unlock(A)**
write(**B**)	**Unlock(B)**
Unlock(B)	Print **A+B**
End	**Commit**

This is a 2PL schedule!

Strong Strict 2PL



This is a Strong Strict 2PL schedule and it will not suffer cascade aborts!

Self Reading Tasks

- 2PL faces deadlocks!
- How can you detect deadlocks in 2PL?
- Learn about deadlock prevention and deadlock avoidance schemes.

2PL in Distributed Systems

- What changes do we need to enable 2PL in a distributed system?

2PL in Distributed Systems

- **Coordinator**
  - For each distributed transaction, one of the partitions will act as the coordinator.
  - Often, the shard which received the client transaction.

2PL in Distributed Systems

- **Coordinator**
  - For each distributed transaction, one of the partitions will act as the coordinator.
  - Often, the shard which received the client transaction.

- **Transaction Manager**
  - The transaction manager at a partition analyzes the transaction and finds out the read/write sets for the transaction.
  - Determines the locks needed by the transaction.
  - Requests the locks from Lock Manager

2PL in Distributed Systems

- **Coordinator**
  - For each distributed transaction, one of the partitions will act as the coordinator.
  - Often, the shard which received the client transaction.

- **Transaction Manager**
  - The transaction manager at a partition analyzes the transaction and finds out the read/write sets for the transaction.
  - Determines the locks needed by the transaction.
  - Requests the locks from Lock Manager

- **Lock Manager**
  - Assigns or denies access to the locks.

Centralized 2PL

- The transaction manager at the coordinator partition determines the read/write sets.
- The transaction manager at the coordinator partition requests locks from lock manager.

Centralized 2PL

- The transaction manager at the coordinator partition determines the read/write sets.
- The transaction manager at the coordinator partition requests locks from lock manager.
- There is only one **central lock manager**.

Centralized 2PL

- The transaction manager at the coordinator partition determines the read/write sets.
- The transaction manager at the coordinator partition requests locks from lock manager.
- There is only one **central lock manager**.
- The lock manager grants the necessary locks to the transaction manager at the coordinator partition.

Centralized 2PL

- The transaction manager at the coordinator partition determines the read/write sets.
- The transaction manager at the coordinator partition requests locks from lock manager.
- There is only one **central lock manager**.
- The lock manager grants the necessary locks to the transaction manager at the coordinator partition.
- The transaction manager asks participants to execute required operations.

Distributed 2PL

Distributed 2PL

- Each participant has its own lock manager.

- Each participant can request locks from its own lock manager.

Distributed 2PL

- Each participant has its own lock manager.
- Each participant can request locks from its own lock manager.
- The transaction manager at the coordinator partition determines the read/write sets.
- The transaction manager at the coordinator partition informs other partitions what operations to execute.

Challenges for Distributed CC

Challenges for Distributed CC

1. **Communication**
   - A lot of communication takes place between the coordinator, participants, and lock managers.

   - Existing distributed systems attempt to optimize this communication.

Challenges for Distributed CC

1. **Communication**
   - A lot of communication takes place between the coordinator, participants, and lock managers.
   - Existing distributed systems attempt to optimize this communication.
2. **Transaction Fate**
   - Post execution, the fate of the transaction has to be determined.
   - Transaction may be committed or aborted.

Communication Reduction

- How can distributed systems reduce communication among participants?

Communication Reduction

- How can distributed systems reduce communication among participants?

- **Batching**
  - Batching implies packaging multiple transactions or requests together.

  - Instead of sending one transaction at a time, participants can send one batch of transactions at a time.

Communication Reduction

- How can distributed systems reduce communication among participants?

- **Batching**
  - Batching implies packaging multiple transactions or requests together.

  - Instead of sending one transaction at a time, participants can send one batch of transactions at a time.

- **Who can use Batching?**
  - Anyone can employ batching.

  - For instance, if a client has multiple transactions to send, it can batch these transactions.

  - Similarly, client proxy can batch transactions.

Communication Reduction

- **Challenges of Batching?**

Communication Reduction

- **Challenges of Batching?**

- **Optimal size of batch**
  - If the batch size is too large, then a long wait time before the batch is ready to send → **increase in latency**.

  - If the batch size is too small, then **minimal reduction** in communication.

Communication Reduction

- **Challenges of Batching?**

- **Optimal size of batch**
  - If the batch size is too large, then a long wait time before the batch is ready to send → **increase in latency**.

  - If the batch size is too small, then **minimal reduction** in communication.

- **Lack of sufficient transactions**
  - If there are not enough transactions, batch may not reach an optimal size.

Communication Reduction

- **Challenges of Batching?**

- **Optimal size of batch**
  - If the batch size is too large, then a long wait time before the batch is ready to send → **increase in latency**.

  - If the batch size is too small, then **minimal reduction** in communication.

- **Lack of sufficient transactions**
  - If there are not enough transactions, batch may not reach an optimal size.

- **Fixed or Variable Size of Batch**
  - Fixed size batches are easy to process as size is known to all but face the optimality challenge.
  - Variable size batches are tricky and require extra handling during failures.

Transaction Fate

- Once a distributed transaction is executed by all the participants, we need to decide whether to commit or abort the transaction.

- When will a transaction commit or abort?

Transaction Fate

- Once a distributed transaction is executed by all the participants, we need to decide whether to commit or abort the transaction.

- When will a transaction commit or abort?

- A transaction **commits** when all the participants agree.

Transaction Fate

- Once a distributed transaction is executed by all the participants, we need to decide whether to commit or abort the transaction.
- When will a transaction commit or abort?
- A transaction **commits** when all the participants agree.
- A transaction **aborts** when one of the participant disagrees.
- When can a participant ask for aborting a transaction?

Transaction Fate

- Once a distributed transaction is executed by all the participants, we need to decide whether to commit or abort the transaction.
- When will a transaction commit or abort?
- A transaction **commits** when all the participants agree.
- A transaction **aborts** when one of the participant disagrees.
- When can a participant ask for aborting a transaction?
- **Reasons:** violation of integrity constraints, conflict with intra-transactions, etc.

Transaction Fate

- What are the necessary tasks post taking a decision to abort the transaction?

Transaction Fate

- What are the necessary tasks post taking a decision to abort the transaction?

- **Undo**
  - Each participant that executed the transaction would need to undo the effects of the aborted transaction.
  - This would require **rollbacking** the state!
  - What if the write set of this transaction were read by some other transaction? **Cascade Rollbacks!**
  - In fact, several systems avoid cascade rollbacks by **disallowing** other transactions to read the write set of an uncommitted transaction.
  - However, some systems **allow** reading writes of an uncommitted transaction.

Transaction Fate

- **Redo**
  - Post aborting a transaction, you have two choices:

  - **Inform client** that its transaction got aborted.
    - The client may get upset and will abandon the application!

Transaction Fate

- **Redo**
  - Post aborting a transaction, you have two choices:
  - **Inform client** that its transaction got aborted.
    - The client may get upset and will abandon the application!
  - **Re-run the aborted transaction**.
    - The more optimal choice → place the aborted transaction back in the queue.
    - However, it may again get aborted?

Transaction Fate

- **Redo**
  - Post aborting a transaction, you have two choices:
  - **Inform client** that its transaction got aborted.
    - The client may get upset and will abandon the application!
  - **Re-run the aborted transaction**.
    - The more optimal choice → place the aborted transaction back in the queue.
    - However, it may again get aborted?
    - Place the transaction back in the queue after an **exponential backoff** time; increases probability of not aborting.