Concurrency Control: 2PL

Dr. L.M. Jenila Livingston
VIT Chennai

Lock Management

- A lock is a mechanism to control concurrent access to a data item
- Lock requests are made to concurrency-control manager or lock manager. Transaction can proceed only after request is granted.
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests

3 Types of Locks

- We allow transactions to **lock** objects.

Shared lock (S): Data item can **only be read**. S-lock is requested using **lock-S** instruction.

Exclusive lock (X): Data item can be **both read** as well as **write**. X-lock is requested using **lock-X** instruction.

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

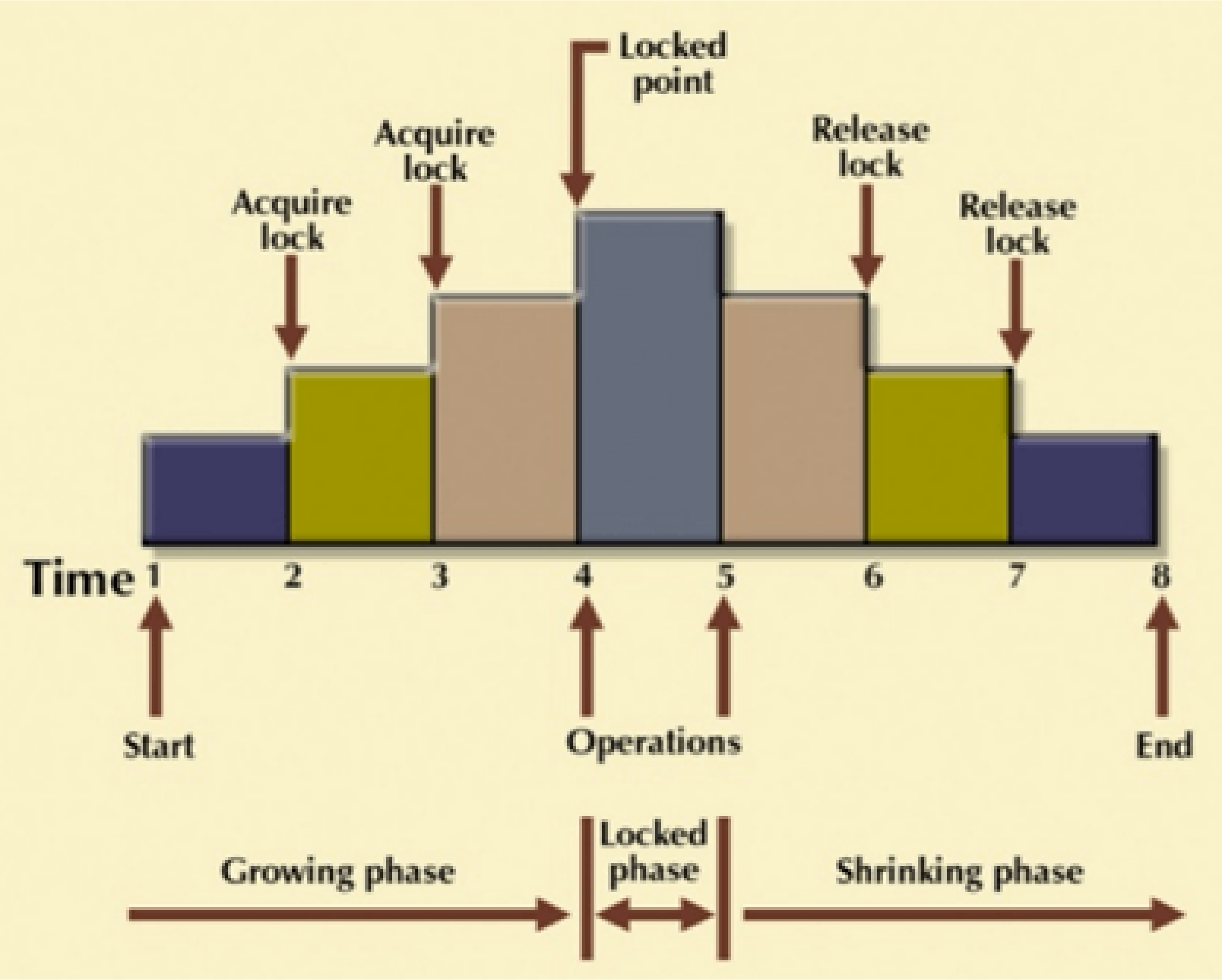
	S	X
S	true	false
X	false	false

- **Any number of transactions can hold shared locks on an item**,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

The Two-Phase Locking (2PL) Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

2 PL



Locked point
Acquire lock
Acquire lock
Release lock
Release lock
Time
1
2
3
4
5
6
7
8
Start
Operations
End
Growing phase
Locked phase
Shrinking phase

The Two-Phase Locking Protocol (Cont.)

- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules
- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.
- .

Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

2 PL: Automatic Acquisition of Locks - read

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(*D*) is processed as:

if T_i has a lock on *D* // **lock-X** OR **lock-S**

then

read(*D*)

else begin

if necessary wait until no other transaction has a **lock-X** on *D*

grant T_i a **lock-S** on *D*;

read(*D*)

end

2PL: Automatic Acquisition of Locks - write

- **write**(*D*) is processed as:

if T_i has a **lock-X** on *D*
  then
    write(*D*)
  else begin
      if necessary wait until no other trans. has any lock on *D*,
      if T_i has a **lock-S** on *D*
        then
          upgrade lock on *D* to **lock-X**
        else
          grant T_i a **lock-X** on *D*
        write(*D*)
    end;

- **All locks are released after commit or abort**

Lock-Based Protocols (Cont.)

- **Example** of a transaction performing locking:

T_2: **begin**

lock-S(*A*);

read (*A*);

unlock(*A*);

lock-S(*B*);

read (*B*);

unlock(*B*);

display(*A+B*)

commit;

T2:

read(A)
read(B)
display(A+B)

Lock-Based Protocols – 2PL

T1:	T2:
read(X)	
read(Y)	
	read(Y)
write (Y)	

T1:	T2:
begin	
lock-S(X)	
read(X)	
lock-S(Y)	
read(Y)	
	begin
	lock-S(Y)
	read(Y)
	unlock(Y)
	commit
upgrade(Y)	
write (Y)	
unlock(X)	
unlock(Y)	
commit	

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

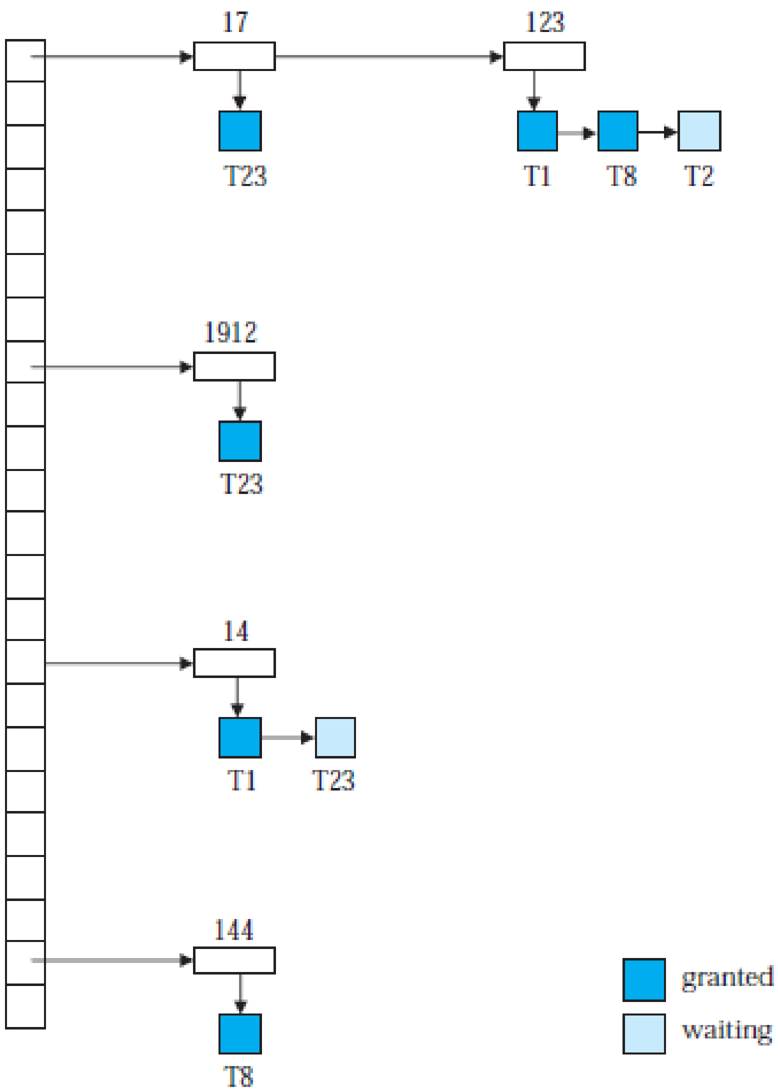
Pitfalls of Lock-Based Protocols (Cont.)

- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



- Lock table records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Timestamp-Based Protocols

1. Use the value of the system clock as the timestamp; that is, a transaction’s timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction’s timestamp is equal to the value of the counter when the transaction enters the system.

Timestamp-based Protocols

- Suppose there are there transactions T1, T2, and T3.
- T1 has entered the system at time 0010
- T2 has entered the system at 0020
- T3 has entered the system at 0030
- Priority will be given to transaction T1, then transaction T2 and lastly Transaction T3.

Thank You!