Transaction Management

(Concurrency Control)

Concurrency Control

Transactions are submitted to system

How can concurrency control help us product correct results?

We now categorized schedules that produces a correct results

Why do we need concurrency control?

Recap:

- Non-serial schedules
- serializability
- Conflict serializability

Locks

A **lock** is a mechanism to control concurrent access to a data item

Data items can be locked in two modes:

1. ***Exclusive*** *(X) mode*.
   - The data item can be both read as well as written.
   - Also known as a **Write Lock**.
2. ***Shared*** *(S) mode*.
   - The data item can only be read.
   - Also known as a **Read Lock**.

To use a data item you must acquire the relevant locks, which each transaction will Transaction can only after request is granted.

Lock-Based Protocols

Lock requests are made to **concurrency-control manager**.

An exclusive lock is requested using **write_lock()** instruction.
A shared lock is requested using **read_lock()** instruction.

A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions

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.

Goal: Locking protocols enforce serializability by restricting the set of possible schedules

Lock-Based Protocols

T_1	T_2	concurrency-control manager
write_lock(B)		
		grant-write_lock(B, T_1)

read(B)
$B := B - 50$
write(B)
unlock(B)		

	read_lock(A)	
		grant-read_lock(A, T_2)

	read(A)
	unlock(A)
	read_lock(B)	

		grant-read_lock(B, T_2)

	read(B)
	unlock(B)
	display($A + B$)	

write_lock(A)		
		grant-write_lock(A, T_1)

read(A)
$A := A + 50$
write(A)
unlock(A)		

Note:

- Grants omitted in rest of slides
- Assume grant happens just before the next instruction following lock request

Lock-Based Protocols

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

A transaction may be granted a lock on an item:

- if the requested lock is compatible with locks already held on the item by other transactions

This means:

- Any number of transactions can hold shared locks on an item,
- But if a transaction holds an exclusive on an item no other transaction may hold any lock on the item.

Lock-Based Protocols

A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.

Locking protocol 1: A simple locking protocol

If *T* has only one operation manipulating an item *X*:

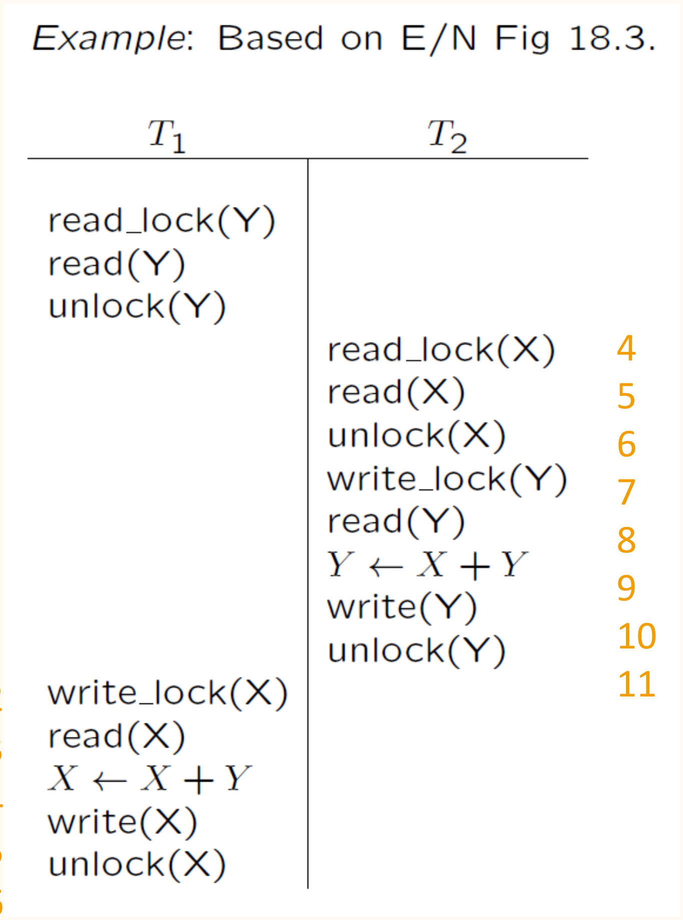
- *if read:* obtain a read lock on *X* before reading
- *if write:* obtain a write lock on *X* before writing
- unlock *X after this operation on X*

Even if *T* has several operations manipulating *X, we* obtain **one** lock still:

- if all operations on *X* are reads, obtain read lock
- if at least **one** operation on *X* is a write, obtain write lock
- unlock *X* after the **last** operation on *X*

Lock-Based Protocols

Simple locking protocol in action:

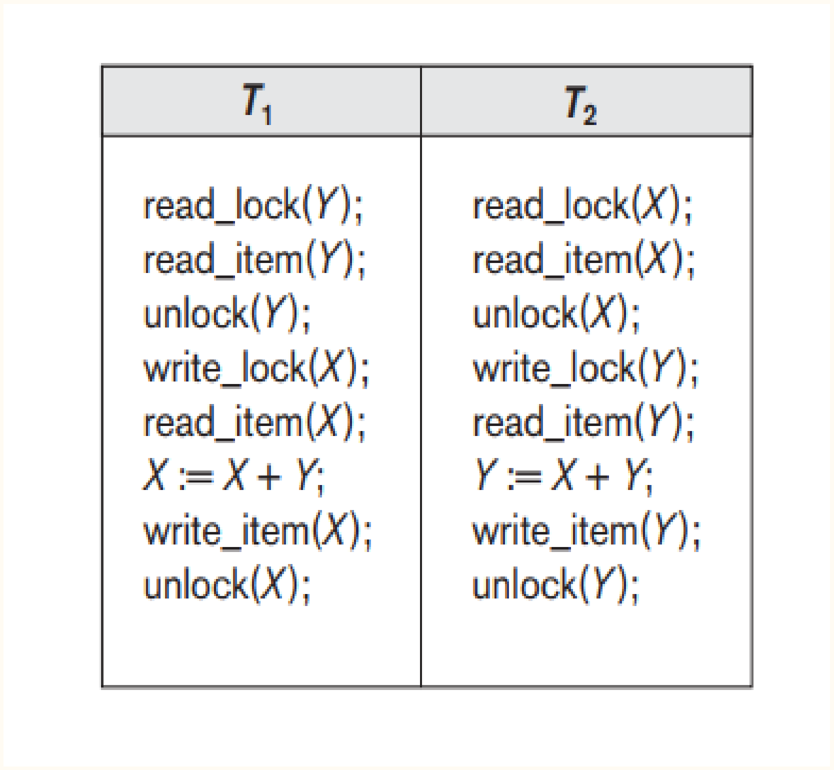


Example: Based on E/N Fig 18.3.

	T_1	T_2	
1	read_lock(Y)		
2	read(Y)		
3	unlock(Y)		
		read_lock(X)	4
		read(X)	5
		unlock(X)	6
		write_lock(Y)	7
		read(Y)	8
		$Y \leftarrow X + Y$	9
		write(Y)	10
		unlock(Y)	11
12	write_lock(X)		
13	read(X)		
14	$X \leftarrow X + Y$		
15	write(X)		
16	unlock(X)		

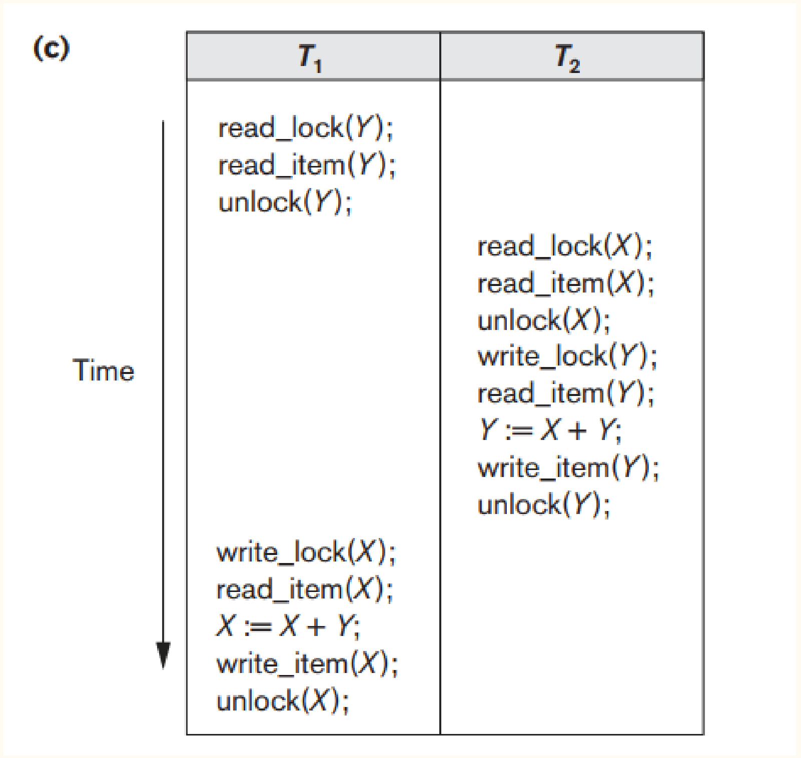
Lock-Based Protocols

Example:



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

Locking Protocol



(c)

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

Lock-Based Protocols (3)

Example of transactions performing locking:

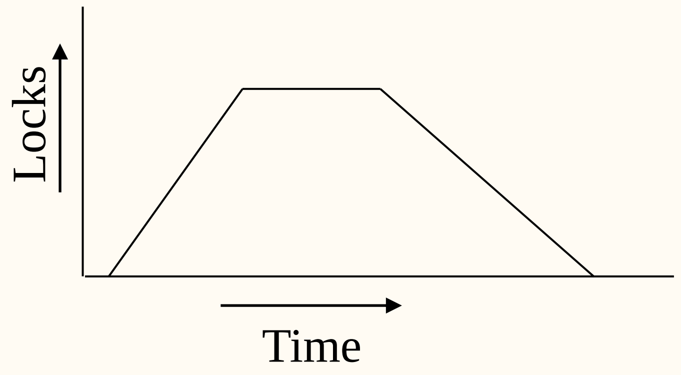
```
T1: write_lock(B);            T2: read_lock(A);
    read(B);                      read(A);
    B: = B – 50;                  unlock(A);
    write(B);                     read_lock(B);
    unlock(B);                    read(B);
    write_lock(A);                unlock(B);
    read(A);                      display(A+B);
    A: = A + 50;
    write(A);
    unlock(A);
```

Locking as above is *not sufficient* to guarantee serializability

Two Phase Locking (2PL)

Locking protocol 2:

1. Phase 1: Growing Phase
   - Transaction obtains locks
   - Transaction does not release any locks
2. Phase 2: Shrinking Phase
   - Transaction releases locks
   - Transaction does not obtain new locks



This protocol ensures serializability: and produces conflict-serializable schedules. 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).

Two Phase Locking (2PL)

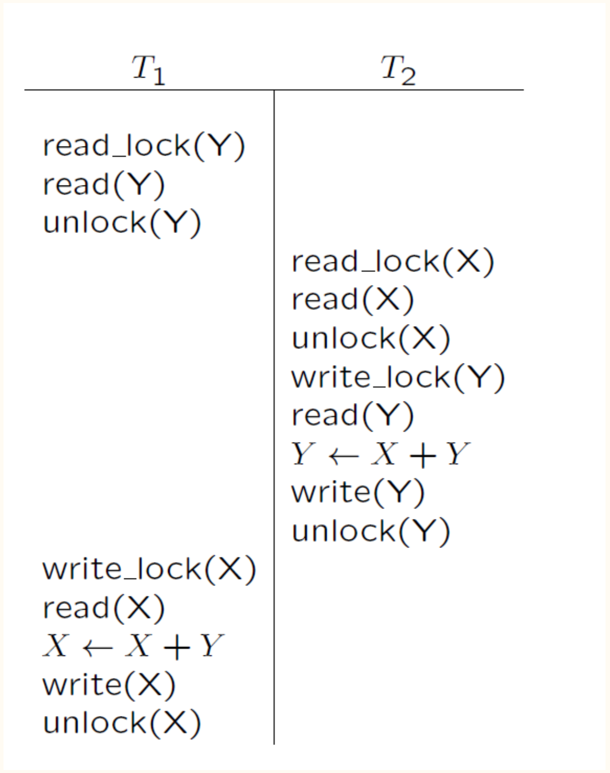
Example of transactions performing locking:

```
T3: write_lock(B);
    read(B);
    B: = B – 50;
    write(B);
    write_lock(A);
    read(A);
    A: = A + 50;
    write(B);
    unlock(B);
    unlock(A);
```

```
T4: read_lock(A);
    read(A);
    read_lock(B);
    read(B);
    display(A+B);
    unlock(A);
    unlock(B);
```

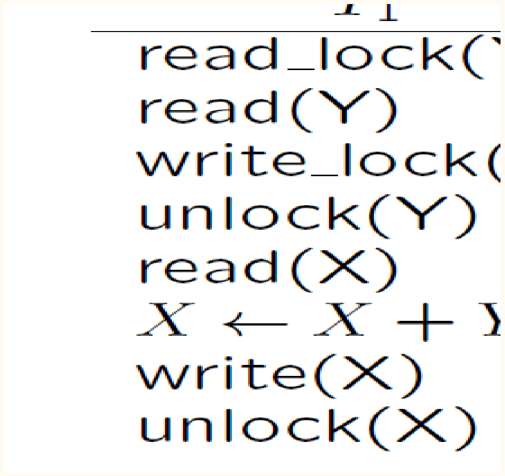
Locking as above is *sufficient* to guarantee serializability, if unlocking is delayed to the end of the transaction.

Two Phase Locking (2PL)



T_1	T_2
read_lock(Y)	
read(Y)	
unlock(Y)	
	read_lock(X)
	read(X)
	unlock(X)
	write_lock(Y)
	read(Y)
	$Y \leftarrow X + Y$
	write(Y)
	unlock(Y)
write_lock(X)	
read(X)	
$X \leftarrow X + Y$	
write(X)	
unlock(X)	

T1 that's with no the 2PL scheme



```
read_lock(
read(Y)
write_lock(
unlock(Y)
read(X)
X ← X +
write(X)
unlock(X)
```

A transaction that locks hat 2PL scheme

Summary

2PL can guarantee serializability, thus allowing real-time control of concurrent executions (scheduling).

Deadlocks

Deadlock occurs when *each* transaction *T* in a set of *two or more transactions* is waiting for some item that is locked by some other transaction *T* in the set.

In most locking protocols, a deadlock can exist.

Deadlocks

Consider the partial schedule to the right.

The instructions from T3 and T4 arrive at the system…

- executing **read_lock(B):**
  - T4 waits for T3 to release its lock on B
- executing **write_lock(A):**
  - T3 waits for T4 to release its lock on A

Such a situation is called a **deadlock**.

Issue: neither T3 nor T4 can make progress

	T_3	T_4
1	write_lock(B)	
2	read(B)	
3	$B := B - 50$	
4	write(B)	
5		read_lock(A)
6		read(A)
7		read_lock(B)
8	write_lock(A)	

Deadlock Prevention Scheme

Locking protocols are used in most commercial DBMSs.

We need to address this issue of deadlocks

One way to prevent deadlock is to use a **deadlock prevention protocol**.

Deadlock Prevention Scheme

Timeouts

If a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it.

Pro: Small overhead and is simplicity.

Con: There may not be a deadlock

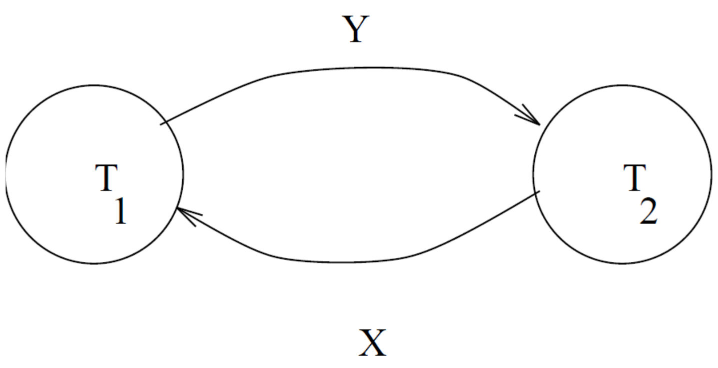
Testing for Deadlocks

Create a *wait-for graph* for currently active transactions:

- create a vertex for each transaction; and
- an arc from T_i to T_j if T_i **is waiting** for an item locked by T_j.

If the graph has a cycle, then a *deadlock* has occurred.

Example:



Testing for Deadlocks

Case: wait-for graph with cycle(s)

	T_3	T_4
1	write_lock(B)	
2	read(B)	
3	$B := B - 50$	
4	write(B)	
5		read_lock(A)
6		read(A)
7		read_lock(B)
8	write_lock(A)	

Summary

- The use of locks, combined with the 2PL protocol, guarantees serializability of schedules.
- The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire.
- If a transaction needs an item that is already locked, it may be forced to wait until the item is released. Some transactions may be aborted and restarted because of the deadlock problem.

Deadlock Prevention Scheme

Lock-based concurrency control cannot prevent deadlocks.

We need an active solution, a solution not based on locks maybe?

A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equivalent serial schedule.

Concurrency Control with Timestamps

- Not based on locks!
- *Assign each transaction* **T_i** *a timestamp T* **TS(T_i)**
- unique identifier to identify a transaction

Implementing Timestamps

Timestamps can be generated in several ways.

- Possibility 1:
  - A simple **counter** (e.g., int counter)
  - (Increment value each time its value is assigned to a transaction.)
- Possibility 2:
  - Use the current **date/time value** of the system clock.

Timestamp Ordering

Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.

- For two transactions T_i and T_j that may be involved in a deadlock.

  Assume T_i wants a lock that T_j holds, two policies are possible:

  - Policy 1 (**Wait-Die Protocol**):
    - If T_i is older, T_i waits for T_j
    - If T_i is younger, T_i aborts*

  - Policy 2 (**Wound-wait Protocol**):
    - If T_i is younger, T_i waits for T_j
    - If T_i is older, T_j aborts*

T_i older than T_j if $\mathbf{TS(T_i) < TS(T_j)}$
* If a transaction re-starts, it retains its original timestamp

Cyclic Restart

Notice: both the schemes end up aborting the younger of the two transactions, Why?

Recall: If a transaction re-starts, it retains its original timestamp

It could lead to **cyclic restart**

- A kind of “live lock” can occur - transactions may be constantly aborted and restarted.

Deadlock Prevention Scheme

- Concurrency control via timestamp ordering:
  - Ensures that the result final schedule recorded is equivalent to executing the transactions serially in timestamp order
  - Therefore, ensuring serializability

- Compare this with concurrency control via 2PL:
  - In 2PL, a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols.
  - In timestamp ordering, however, the schedule is equivalent to the particular serial order corresponding to the order of the transaction timestamps.

Learning Outcomes

- Concurrency Control Techniques:
  - Lock-based
    - Ensures serializability, but could result in deadlocks
  - Timestamp-based
    - Prevents deadlocks