# Lecture 10: Concurrency Control

## CS3402 Database Systems

## Database Concurrency Control

- Purposes of Concurrency Control
  - To preserve database consistency to ensure all schedules are serializable
  - To maximize the system performance (higher concurrency)
- Example: In concurrent execution environment, if $T_1$ conflicts with $T_2$ over a data item A, then the existing concurrency control decides whether $T_1$ or $T_2$ should get the A and whether the other transaction is rolled-back or waits.

## Locking Techniques for Concurrency Control

- The main techniques used to control concurrent execution of transactions are based on the concept of locking data items.
- A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database.
- Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

## Binary Locks

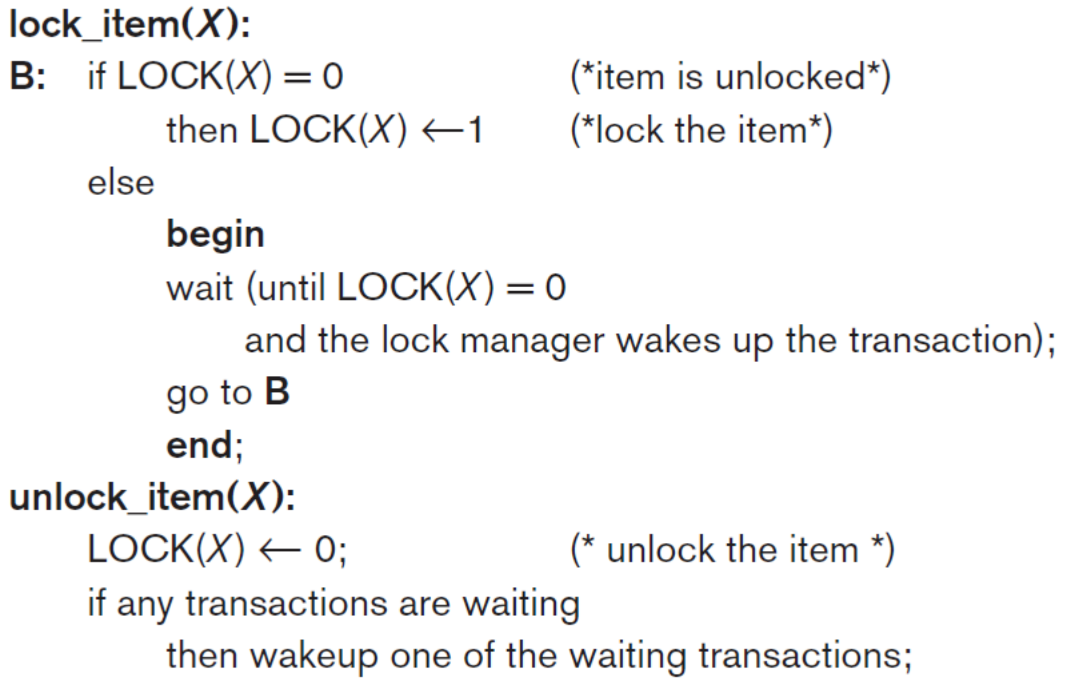
- A **binary lock** can have two **states** or **values**: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X.
- If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item.
- If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1.
- We refer to the current value (or state) of the lock associated with item X as **lock(X)**.

## Locking Operations (1/2)

- Two operations, lock_item and unlock_item, are used with binary locking.
- A transaction requests access to an item X by first issuing a **lock_item(X)** operation.
  - If LOCK(X) = 1, the transaction is forced to wait.
  - If LOCK(X) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item X.
- When the transaction is through using the item, it issues an **unlock_item(**X**)** operation, which sets LOCK(X) back to 0 (**unlocks** the item) so that X may be accessed by other transactions.
- Hence, a binary lock enforces **mutual exclusion** on the data item.

# Locking Operations (2/2)

- Notice that the lock_item and unlock_item operations must be implemented as indivisible units or atomic functions (known as **critical sections** in operating systems)
- No interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits



```
lock_item(X):
B:  if LOCK(X) = 0                  (*item is unlocked*)
        then LOCK(X) ←1             (*lock the item*)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                    (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

## Multiple-Mode Locks (1/3)

- The binary locking scheme is too restrictive for database items because at most one transaction can hold a lock on a given item.
- We should allow several transactions to access the same item X if they all access X for reading purposes only. This is because read operations on the same item by different transactions are not conflicting.
- To this end, we have **multiple-mode lock**, is used. In this scheme—called **shared/exclusive** or **read/write** locks—there are three locking operations: read_lock(X), write_lock(X), and unlock(X).
- A lock associated with an item X, LOCK(X), now has three possible states: read-locked, write-locked, or unlocked.
  - A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

## Multiple-Mode Locks (2/3)

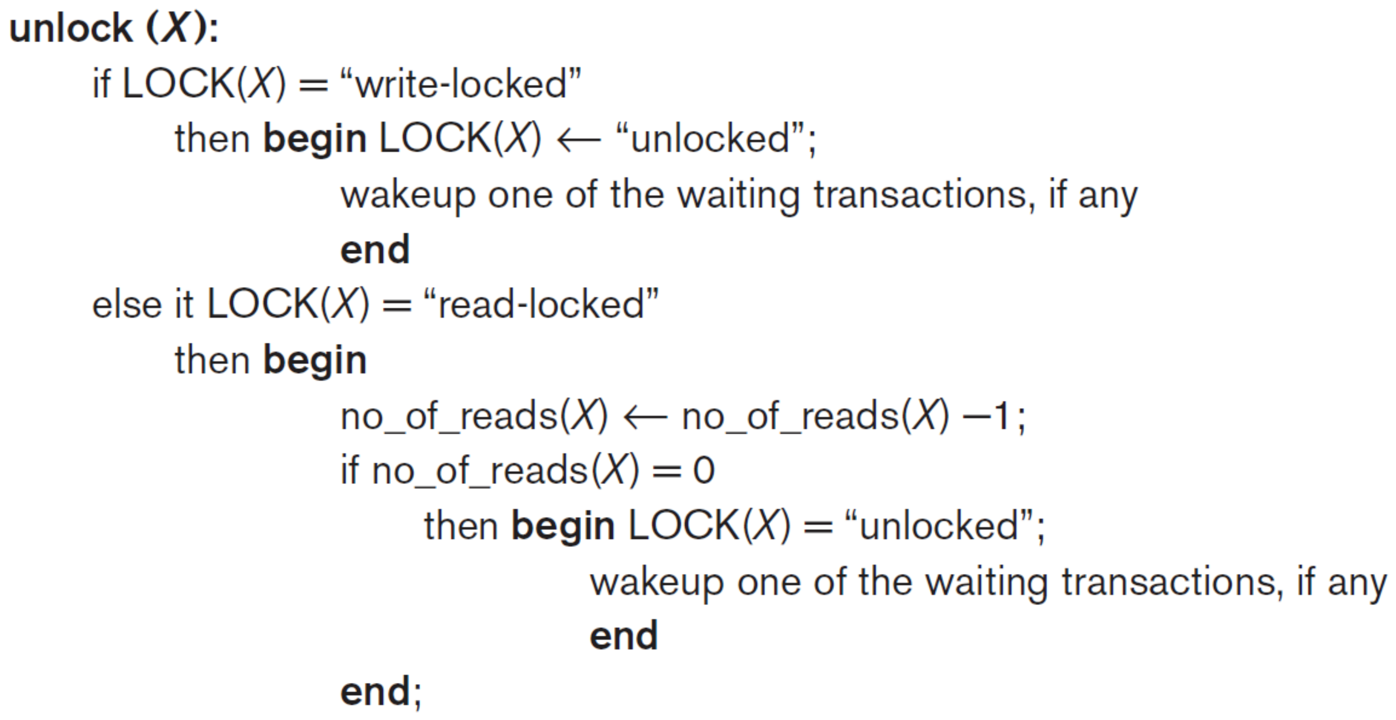
- The three operations read_lock(X), write_lock(X), and unlock(X) are atomic functions. Each of the three locking operations should be considered indivisible; no interleaving should be allowed once one of the operations is started until either the operation terminates by granting the lock or the transaction is placed in a waiting queue for the item.

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) $\leftarrow$ "read-locked";
            no_of_reads(X) $\leftarrow$ 1
            end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) $\leftarrow$ no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;

write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) $\leftarrow$ "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
```

# Multiple-Mode Locks (3/3)

- Unlocking operation for two-mode (read/write, or shared/exclusive) locks.



```
**unlock ($X$):**
    if LOCK($X$) = "write-locked"
        then **begin** LOCK($X$) $\leftarrow$ "unlocked";
                wakeup one of the waiting transactions, if any
                **end**
    else it LOCK($X$) = "read-locked"
        then **begin**
                no_of_reads($X$) $\leftarrow$ no_of_reads($X$) −1;
                if no_of_reads($X$) = 0
                    then **begin** LOCK($X$) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            **end**
                **end**;
```

## Conversion of Locks (1/2)

- It is desirable to allow **lock conversion,** that is, a transaction that already holds a lock on item X is allowed under certain conditions to **convert** the lock from one locked state to another.
- For example, it is possible for a transaction T to issue a read_lock(X) and then later to **upgrade** the lock by issuing a write_lock(X) operation. If T is the only transaction holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded; otherwise, the transaction must wait.

```
if $T_i$ has a read-lock(X) and $T_j$ has no read-lock(X) ($i \neq j$) then
      Upgrade read-lock(X) to write-lock(X) for $T_i$;
else
      Force $T_i$ to wait until $T_j$ unlocks X;
```

## Conversion of Locks (2/2)

- It is also possible for a transaction T to issue a write_lock(X) and then later to **downgrade** the lock by issuing a read_lock(X) operation.

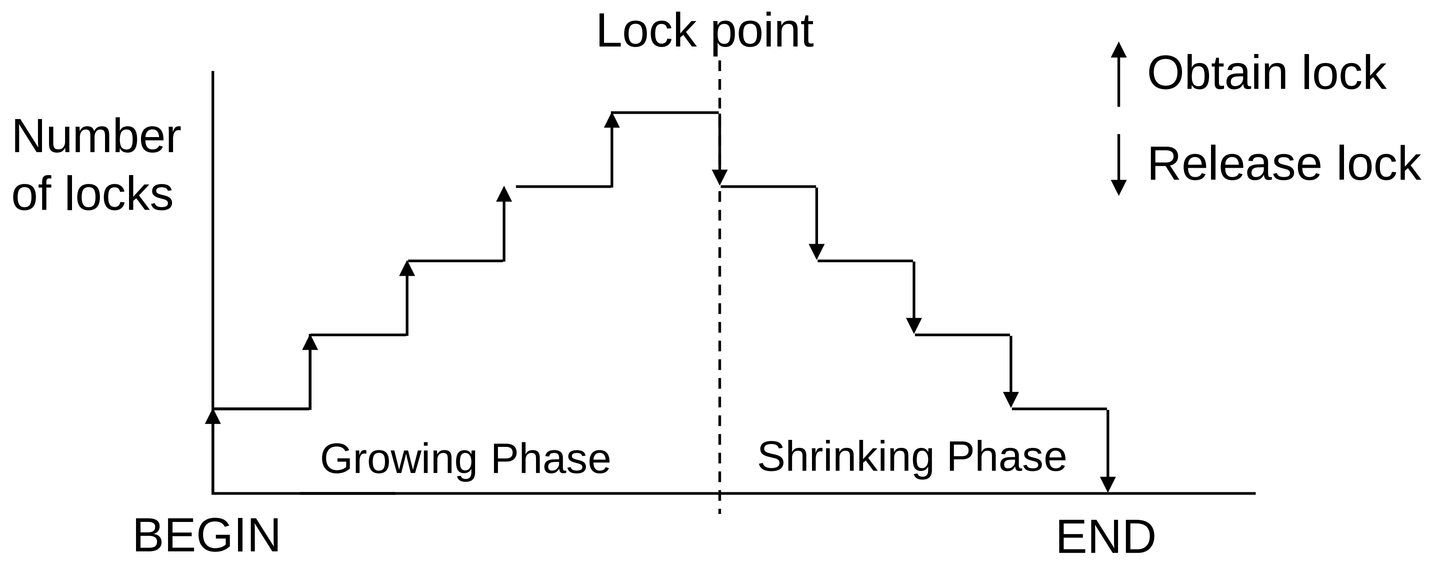
```
if $T_i$ has a write-lock(X) then
    //no transaction can have any lock on X
    Downgrade write-lock(X) to read-lock(X);
```

## Basic Two-Phase Locking (2PL) Protocol (1/3)

- If all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.
- Such a transaction can be divided into two phases: an **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released; and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired.
- If lock conversion is allowed, then upgrading of locks (from read-locked to write-lock ed) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

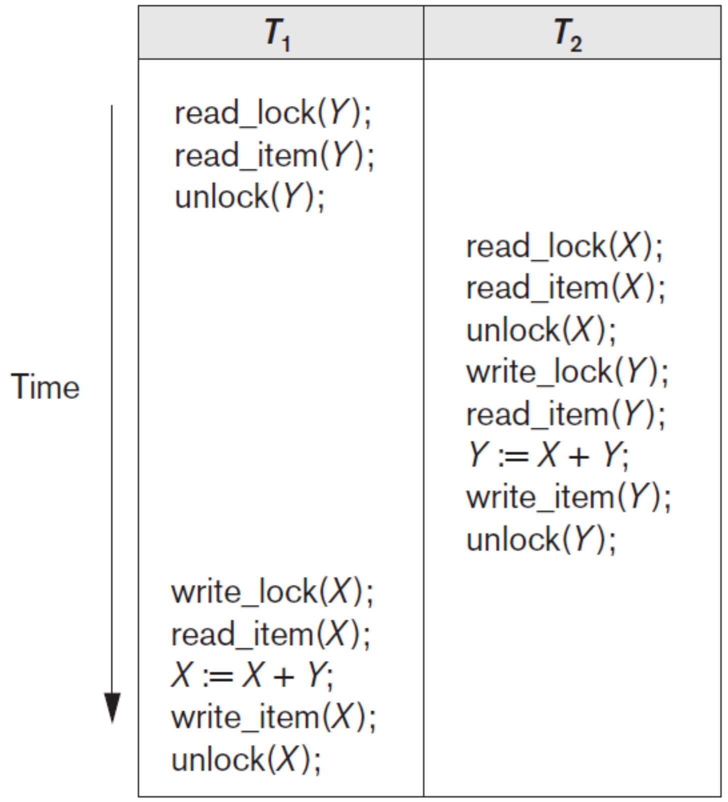
## Basic Two-Phase Locking (2PL) Protocol (2/3)

- It can be proved that, if every transaction in a schedule follows the basic 2PL protocol, the schedule is guaranteed to be serializable, obviating the need to test for serializability of schedules. The locking protocol, by enforcing two-phase locking rules, also enforces serializability.
- However, the basic 2PL protocol can produce deadlock.



## Basic Two-Phase Locking (2PL) Protocol (3/3)

- Transactions $T_1$ and $T_2$ do not follow the basic 2PL protocol because the write_lock(X) operation follows the unlock(Y) operation in $T_1$, and similarly the write_lock(Y) operation follows the unlock(X) operation in $T_2$.

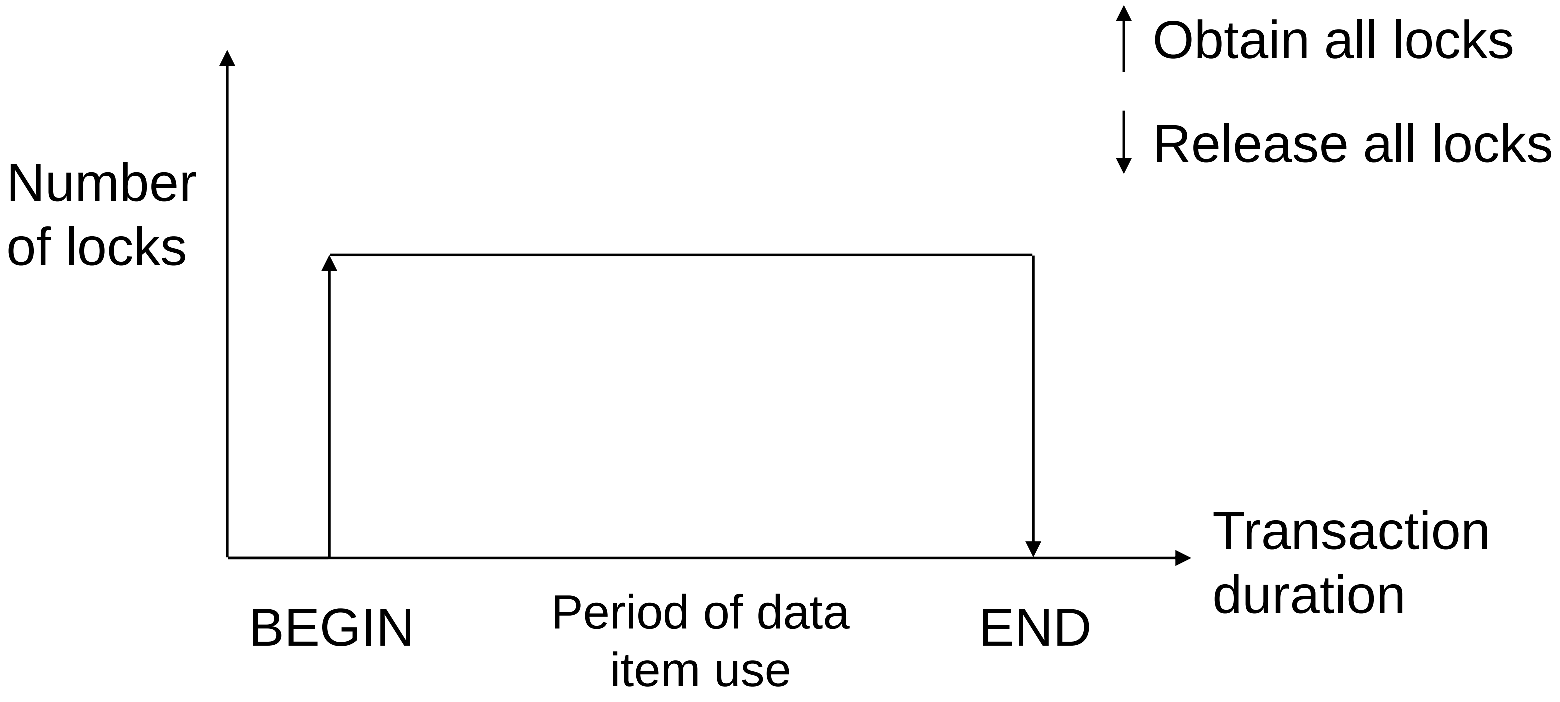


| $T_1$ | $T_2$ |
|---|---|
| read_lock($Y$);<br>read_item($Y$);<br>unlock($Y$); | |
| | read_lock($X$);<br>read_item($X$);<br>unlock($X$);<br>write_lock($Y$);<br>read_item($Y$);<br>$Y := X + Y$;<br>write_item($Y$);<br>unlock($Y$); |
| write_lock($X$);<br>read_item($X$);<br>$X := X + Y$;<br>write_item($X$);<br>unlock($X$); | |

# Conservative 2PL (1/3)

- A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses before the transaction begins execution, by **predeclaring** its read-set and write-set.
- The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes.
- If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking.
- Conservative 2PL is a deadlock-free protocol.
- However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in some situations.

## Conservative 2PL (2/3)



# Conservative 2PL (3/3)

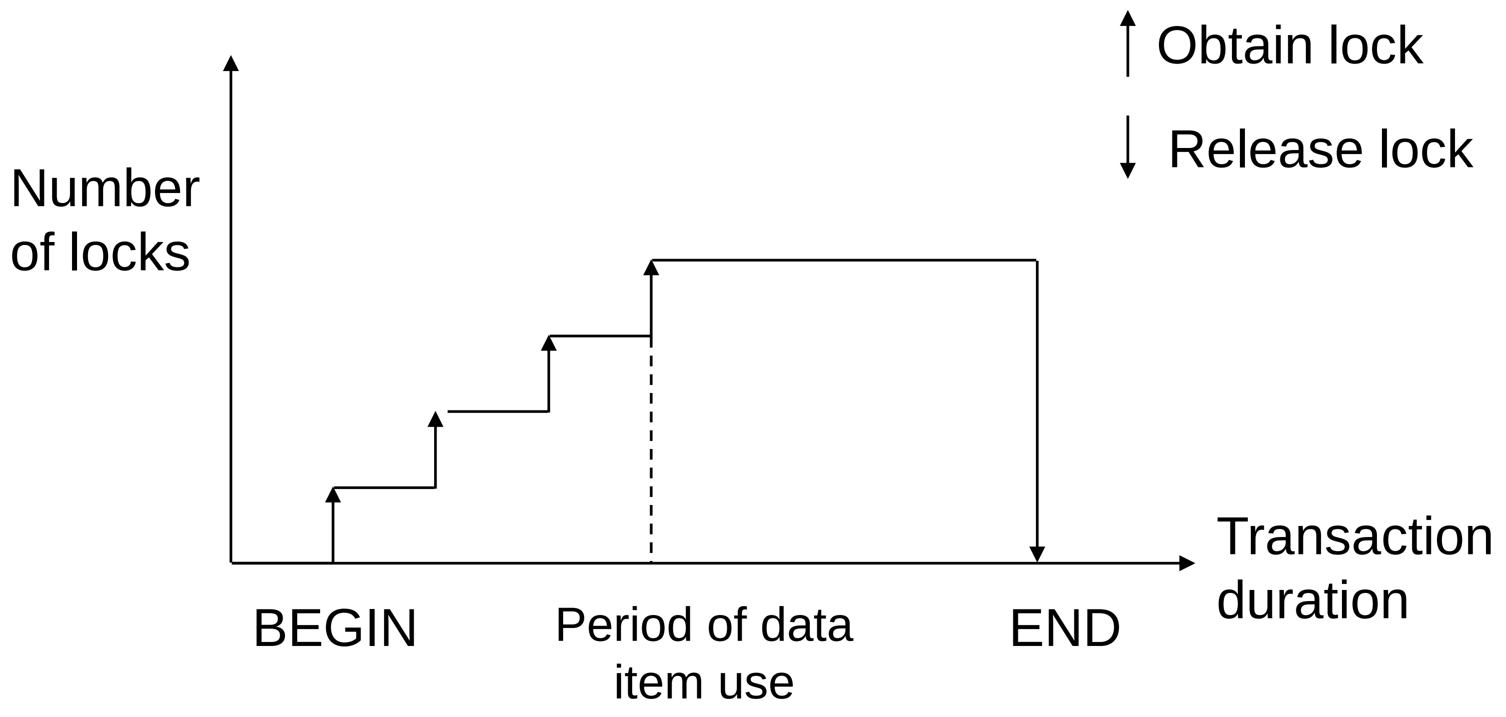
| $T_1$ | $T_2$ |
|---|---|
| | write(a) |
| write(a) | |
| | write(b) |
| | commit |
| write(b) | |
| commit | |

| $T_1$ | $T_2$ |
|---|---|
| | write_lock(a,b) |
| | write(a) |
| write_lock(a,b) $\rightarrow$ blocked | |
| | write(b) |
| | commit |
| | unlock(a,b) |
| write_lock(a,b) | |
| write(a) | |
| write(b) | |
| commit | |
| unlock(a,b) | |

## Strict 2PL (1/5)

- In practice, the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules.
- In this variation, a transaction T does not release any of its exclusive (write) locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability.
- Strict 2PL is NOT deadlock-free.

## Strict 2PL (2/5)



## Strict 2PL (3/5)

| $T_1$ | $T_2$ |
|---|---|
| | write(a) |
| write(a) | |
| | write(b) |
| | commit |
| write(b) | |
| commit | |

| $T_1$ | $T_2$ |
|---|---|
| | write_lock(a) |
| | write(a) |
| write_lock(a) $\rightarrow$ blocked | |
| | write_lock(b) |
| | write(b) |
| | commit |
| | unlock(a,b) |
| write_lock(a) | |
| write(a) | |
| write_lock(b) | |
| write(b) | |
| commit | |
| unlock(a,b) | |

## Strict 2PL (4/5)

- Basic 2PL only defines the earliest time when the schedule may release a lock for a transaction.
- In Strict 2PL, it requires the scheduler to release all of a transaction’s locks altogether.
- Compared with Basic 2PL, the lock holding time may be longer and the concurrency may be lower.
- Compared with Conservative 2PL, the lock holding time may be shorter and the concurrency may be higher.
- Strict 2PL may have the problem of deadlock but all schedules are recoverable.

## Strict 2PL (5/5)

- Strict 2PL is better than Conservative 2PL when the transaction workload is not heavy (not too many concurrent transactions) since the lock holding time is shorter in Strict 2PL
- When the transaction is heavy, Conservative 2PL is better than Strict 2PL since deadlock may occur in Strict 2PL.

## Implementation Issue: Essential Components (1/2)

- Two lock modes:
  - Shared (read)
  - Exclusive (write)
- Shared mode: read lock (X)
  - More than one transaction can apply shared lock on X for reading its value but no write lock can be applied on X by any other transaction.
- Exclusive mode: write lock (X)
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
- Conflict matrix (permission of doing at the same time)

| | Read | Write |
|---|---|---|
| Read | Y | N |
| Write | N | N |

## Implementation Issue: Essential Components (2/2)

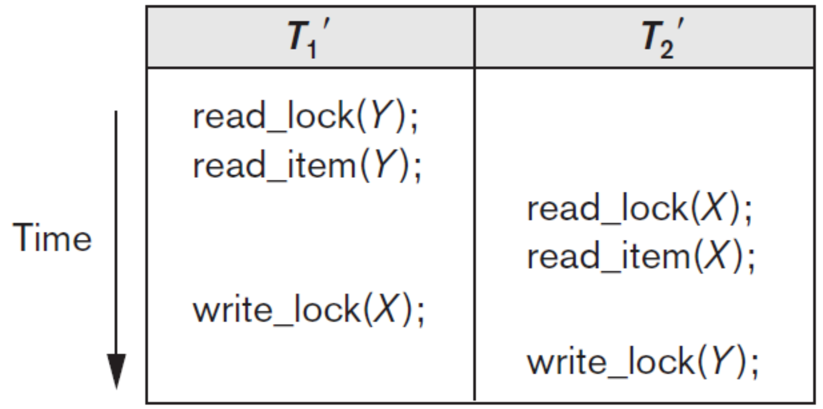
- Lock Manager: Managing locks on data items
- Lock table:
  - An array to show the lock entry for each data item
  - Each entry of the array stores the identify of transaction that has set a lock on the data item including the mode
- One entry for one database? One record? One field? Lock granularity (Coarse granularity vs. fine granularity)
- Larger granularity $\rightarrow$ higher conflict probability but lower locking overhead
- How to detect lock conflict for insertion operations from a transaction?
  - A transaction reads all data items and another one inserts a new item

## Deadlock (1/2)

- **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.
- Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.
- But because the other transaction is also waiting, it will never release the lock.

# Deadlock (2/2)

- Here is a simple example, where the two transactions $T_1'$ and $T_2'$ are deadlocked in a partial schedule
- $T_1'$ is in the waiting queue for X, which is locked by $T_2'$, whereas $T_2'$ is in the waiting queue for Y, which is locked by $T_1'$.
- Meanwhile, neither $T_1'$ nor $T_2'$ nor any other transaction can access items X and Y.



| $T_1'$ | $T_2'$ |
|---|---|
| read_lock($Y$); | |
| read_item($Y$); | |
| | read_lock($X$); |
| | read_item($X$); |
| write_lock($X$); | |
| | write_lock($Y$); |

## Deadlock Prevention Protocols (1/5)

- One way to prevent deadlock is to use a **deadlock prevention protocol**.
- One deadlock prevention protocol, which is used in conservative 2PL, requires that every transaction lock all the items it needs in advance (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously, this solution further limits concurrency.

## Deadlock Prevention Protocols (2/5)

- A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation: Should it be blocked and made to wait or should it be aborted, or should the transaction preempt and abort another transaction?
- Some of these techniques use the concept of **transaction timestamp** TS(T′), which is a unique identifier assigned to each transaction. The timestamps are typically based on the order in which transactions are started; hence, if transaction $T_1$ starts before transaction $T_2$, then $TS(T_1) < TS(T_2)$.
- Notice that the older transaction (which starts first) has the smaller timestamp value.
- Two schemes that prevent deadlock are called **wait-die** and **wound-wait**.

# Deadlock Prevention Protocols (3/5)

- Suppose that transaction $T_i$ tries to lock an item X but is not able to because X is locked by some other transaction $T_j$ with a conflicting lock. The rules followed by these schemes are:
- **Wait-die.** If $TS(T_i) < TS(T_j)$, then ($T_i$ older than $T_j$) $T_i$ is allowed to wait; otherwise ($T_i$ younger than $T_j$) abort $T_i$ ($T_i$ dies) and restart it later with the same timestamp.
  - An older transaction is allowed to wait for a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted.
- **Wound-wait.** If $TS(T_i) < TS(T_j)$, then ($T_i$ older than $T_j$) abort $T_j$ ($T_i$ wounds $T_j$) and restart it later with the same timestamp; otherwise ($T_i$ younger than $T_j$) $T_i$ is allowed to wait.
  - This scheme does the opposite: A younger transaction is allowed to wait for an older one, whereas an older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it.

# Deadlock Prevention Protocols (4/5)

- Both **wait-die** and **wound-wait** schemes end up aborting the younger of the two transactions (the transaction that started later) that may be involved in a deadlock, assuming that this will waste less processing.
- It can be shown that these two techniques are deadlock-free, since in wait-die, transactions only wait for younger transactions so no cycle is created.
- Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created.
- However, both techniques may cause some transactions to be aborted and restarted needlessly, even though those transactions may never actually cause a deadlock.

## Deadlock Prevention Protocols (5/5)

- Both **wait-die** and **wound-wait** schemes achieve deadlock avoidance by using timestamp
- Consider this deadlock cycle: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$
  - It is impossible since if $T_1 \ldots \rightarrow T_n$, then $T_n$ is not allowed to wait for $T_1$
- Wait-die: Older transaction is allowed to wait
  - If $T_1 \ldots \rightarrow T_n$, $T_1$ is order than $T_n$; hence, $T_n$ is not allowed to wait for $T_1$ ($T_n$ aborts).
- Wound-wait: Older transaction is allowed to get the lock
  - If $T_1 \ldots \rightarrow T_n$, $T_1$ is younger than $T_n$; hence, $T_n$ does not need to wait for $T_1$ ($T_1$ aborts).
- In distributed database environments, each transaction is assigned a unique time-stamp, e.g., its creation time, plus a site ID, because different sites may have transactions created at the same time.

## Deadlock Example

- TS of $T_1$ < TS of $T_2$

| $T_1$ | $T_2$ |
|---|---|
| read_lock(a) | |
| read(a) | |
| write_lock(b) | |
| write(b) | |
| | read_lock(c) |
| | read(c) |
| | write_lock(a) $\rightarrow$ blocked |
| write_lock(c) $\rightarrow$ blocked (deadlock formed) | |

# Deadlock Example (wait-die)

- TS of $T_1$ < TS of $T_2$

| $T_1$ | $T_2$ |
|---|---|
| read_lock(a); read(a) | |
| write_lock(b); write(b) | |
| | read_lock(c); read(c) |
| | write_lock(a) (restarts because it is younger than $T_1$ and $T_2$ releases its read lock on c before it restarts) |
| write_lock(c); write(c) | |
| release_lock($T_1$) | |
| | read_lock(c); read(c) |
| | write_lock(a); write(a) |
| | release_lock($T_2$) |

## Deadlock Example (wound-wait)

- TS of $T_1$ < TS of $T_2$

| $T_1$ | $T_2$ |
|---|---|
| read_lock(a); read(a) | |
| write_lock(b); write(b) | |
| | read_lock(c); read(c) |
| | write_lock(a) $\rightarrow$ blocked |
| write_lock(c); write(c) ($T_2$ is restarted by $T_1$ because $T_2$ is younger than $T_1$. The write lock on c is granted to $T_1$ after $T_2$ has released its read lock on c) | |
| release_lock($T_1$) | |
| | read_lock(c); read(c) |
| | write_lock(a); write(a) |
| | release_lock($T_2$) |

## Deadlock Detection (1/3)

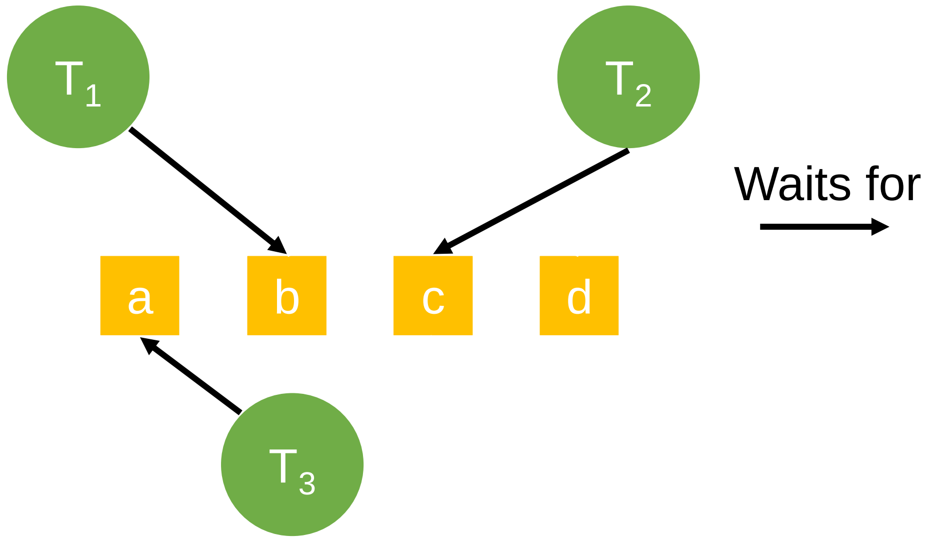
- An alternative approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists.
- A simple way to detect a state of deadlock is for the system to construct and maintain a **wait-for graph**.
- One node is created in the wait-for graph for each transaction that is currently executing.
- Whenever a transaction $T_i$ is waiting to lock an item X that is currently locked by a transaction $T_j$, a directed edge ($T_i \rightarrow T_j$) is created in the wait-for graph. When $T_j$ releases the lock(s) on the items that $T_i$ was waiting for, the directed edge is dropped from the wait-for graph.
- We have a state of deadlock if and only if the wait-for graph has a cycle.

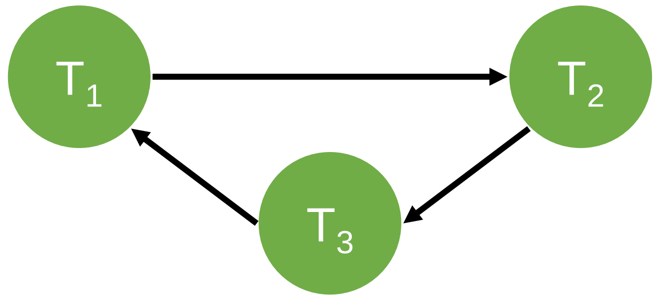
## Deadlock Detection (2/3)

- If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transactions to abort is known as **victim selection**.
- The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).

# Deadlock Detection (3/3)

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| write_lock(d) | | |
| write(d) | | |
| | write_lock(b) | |
| | write(b) | |
| write_lock(a) | | |
| write(a) | | |
| | | write_lock(c) |
| | | write(c) |
| write_lock(b) → blocked | | |
| | write_lock(c) → blocked | |
| | | write_lock(a) → blocked |





wait-for-graph

## Starvation (1/2)

- Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
- This may occur if the waiting scheme for locked items is unfair in that it gives priority to some transactions over others.
- One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock.
- Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.

## Starvation (2/2)

- Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution.
- The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem.
- The wait-die and wound-wait schemes discussed previously avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.