Locking to ensure serializability

- *Concurrent access* to database items is controlled by strategies based on *locking*, *timestamping* or *certification*
- A *lock* is an *access privilege to a single database item*
- *Lock Manager:* manages the locks requested by transactions.
- Locks are
  - *obtained by transactions*
  - stored in a *lock table*
  - Lock is an entry of the form *(item, lock-type, transactionID)*
    - *item* is the item that the transaction locks
    - *lock-type* can be shared or exclusive
    - *transactionID* is the transaction identifier

Locking to Ensure Serializability

- Locking can *prevent the lost update problem*
  - T_1 = *Lock(A)* $R_1(A)$ $W_1(A)$ *Unlock(A)* C_1
  - T_2 = *Lock(A)* $R_2(A)$ $W_2(A)$ *Unlock(A)* C_2
- Under the *locking policy*, *only serial execution of the transactions is permitted*

Locking

- When a transaction holds an *exclusive* lock on a database item, *no other transaction can read or write the item*
  - used for writing
- When a transaction holds a *shared* lock, *other transactions can obtain a shared lock on the same item*
  - used for reading
- ***Assumptions***
  - there is a *single type of lock* and
  - *every transaction must obtain a lock* on an item before accessing it.
  - *all items locked by a transaction must be unlocked*, *otherwise no other transaction may gain access to them.*
  - a transaction *must wait* until the lock it requests is released by the transaction that holds it.

Transaction Management

1. *Locking* can prevent the *lost update* problem:

   $T_1 = Lock_1(A)\ R_1(A)\ W_1(A)\ Unlock_1(A)\ C_1$

   $T_2 = Lock_2(A)\ R_2(A)\ W_2(A)\ Unlock_2(A)\ C_2$

2. *Locking enforces a serial execution of the transactions*

3. Locking can also prevent the blind write problem:

   $T_1 = Lock_1(A)\ W_1(A)\ Lock_1(B)\ W_1(B)\ Unlock_1(A)\ Unlock_1(B)\ C_1$

   $T_2 = Lock_2(A)\ W_2(A)\ Lock_2(B)\ W_2(B)\ Unlock_2(A)\ Unlock_2(B)\ C_1$

   ✓Then the following schedule is valid:

$Lock_1(A)\ W_1(A)\ Lock_1(B)\ W_1(B)\ Unlock_1(A)\ Lock_2(A)\ W_2(A)$
$Unlock_1(B)\ Lock_2(B)\ W_2(B)\ Unlock_2(A)\ Unlock_2(B)\ C_1\ C_2$

LiveLock

- Undesirable phenomena if *locks are granted in an arbitrary manner*
- **Example:**
  - while T2 is waiting for T1 to release the lock on A, another transaction T3 that has also requested a lock on A is granted the lock instead of T2. When T3 releases the lock on A the lock is granted to T4 etc.
- *Livelock: The situation where a transaction may wait for ever while other transactions obtain a lock on a database item*
  - Can be avoided by using a first-come-first-served lock granting strategy but, even then a *deadlock* might occur

Deadlock

- Occurs *when a transaction is waiting to lock an item that is currently locked by some other transaction*
- ***Example:*** Consider the transactions:

  $T_1 = Lock_1(A)\ Lock_1(B)\ ....\ Unlock_1(A)\ Unlock_1(B)\ C_1$

  $T_2 = Lock_2(B)\ Lock_2(A)\ ....\ Unlock_2(B)\ Unlock_1(A)\ C_2$

  - Assume T_1 is *granted* a *lock* on *A* and T_2 is *granted* a *lock* on *B*
  - Then T_1 *requests* a *lock on B* but is **forced to wait** because T_2 *has* the *lock on B*.
  - Similarly, T_2 *requests a lock on A* but is **forced to wait** because T_1 has the *lock on A.*

Neither transaction can proceed because each is waiting for the other to release a lock: both processes wait for ever

Different solutions for Deadlocks

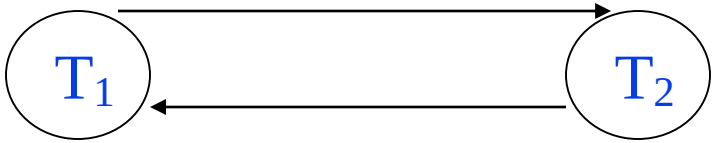
- ***Solution 1:*** Require *each transaction to request all locks at once. Either all locks are granted or none.*
- ***Solution 2:*** Assign *an arbitrary linear order* to the *items* and *require all transactions to request their locks in that order.*
- ***Solution 3:*** *Do nothing to prevent deadlocks: abort* one or more of the deadlocked transactions if a deadlock arises.

Deadlock Discovery

- Deadlocks can be discovered using *wait-for graphs*:
  - Given a set of transactions S, a *wait-for graph* is a *directed graph:*
    - vertices correspond to transactions in the set
    - there exists an edge from T_i to T_j if T_i is waiting to lock an item on which T_j is holding a lock.
- *Theorem*: A set of transactions is deadlocked if and only if there exists a cycle in the *wait-for graph*.
- **Example**: The ***wait-for graph*** for the transactions contains a cycle

  $T_1 = Lock_1(A)\ Lock_1(B)\ ....\ Unlock_1(A)\ Unlock_1(B)\ C_1$

  $T_2 = Lock_2(B)\ Lock_2(A)\ ....\ Unlock_2(B)\ Unlock_2(A)\ C_2$



2-Phase Locking (2PL)

- ✓ 2-Phase Locking (2PL): a protocol ensuring *serializability of schedules*
- ✓ *Definition:* A schedule is said to obey the *2-phase locking* protocol if the following rules are obeyed by *each transaction* in the schedule
  1. When a transaction attempts to *read (write)* a data item, a read *lock (write lock)* must be acquired first
  2. If a transaction T_1 holds a lock on data item A for operation op_1 and some other transaction T_2 requests the lock to perform a *conflicting operation* op_2 on the same item, the transaction requesting the lock (T_2) is forced to *wait until no conflicting lock on the item exists*
     - ✓ ***(only read locks are non-conflicting)***
  3. *A transaction cannot request additional locks once it releases any lock!*

2-Phase Locking (2PL): Conflicts

- two locks by the *same transaction never conflict*
- a transaction with a *read lock on a data item* can acquire a *write lock on the item as long as no other transaction has a lock on the data item;*
- a transaction with *a write lock on a data item need not acquire a read lock* on the same item.
- *2PL permits the early release of locks*
- Notation:
  - RL_i: transaction T_i *obtains* a *read* lock
  - WL_i: transaction T_i *obtains* a *write* lock
  - RU_i: transaction T_i *releases* a *read* lock
  - WU_i: transaction T_i releases a *write* lock

2-Phase Locking (2PL): Example

- Does the following schedule obey the 2PL protocol?

$S = R_1(A)\ R_2(B)\ W_2(B)\ R_2(A)\ W_2(A)\ R_1(B)\ C_1\ C_2$

Lock/unlock operations must be *added first*. The schedule becomes:

$S' = \boxed{RL_1(A)}\ R_1(A)\ RU_1(A)\ \boxed{RL_2(B)}\ R_2(B)\ \boxed{WL_2(B)}\ W_2(B)\ WU_2(B)\ \boxed{RL_2(A)}$

$R_2(A)\ \boxed{WL_2(A)}\ W_2(A)\ \boxed{RL_1(B)}\ R_1(B)\ C_1\ C_2$

- *Rule 1 :* no item is accessed without a lock being granted to the requested transaction
  - *obeyed*

2-Phase Locking (2PL): Example

- Does the following schedule obey the 2PL protocol?

$S = R_1(A)\ R_2(B)\ W_2(B)\ R_2(A)\ W_2(A)\ R_1(B)\ C_1\ C_2$

Lock/unlock operations must be *added first*. The schedule becomes:

$S = RL_1(A)\ R_1(A)\ RU_1(A)\ RL_2(B)\ R_2(B)\ WL_2(B)\ W_2(B)\ WU_2(B)\ RL_2(A)$
$R_2(A)\ WL_2(A)\ W_2(A)\ RL_1(B)\ R_1(B)\ C_1\ C_2$

- *Rule 2* : no two conflicting operations have a lock on the same item at the same time
  - *obeyed*

2-Phase Locking (2PL): Example

- Does the following schedule obey the 2PL protocol?

  $S = R_1(A)\ R_2(B)\ W_2(B)\ R_2(A)\ W_2(A)\ R_1(B)\ C_1\ C_2$

- *Lock/unlock operations* must be *added first*. The schedule becomes:

$S = RL_1(A)\ R_1(A)\ RU_1(A)\ RL_2(B)\ R_2(B)\ WL_2(B)\ W_2(B)\ WU_2(B)\ RL_2(A)\ R_2(A)$

$WL_2(A)\ W_2(A)\ RL_1(B)\ R_1(B)\ C_1\ C_2$

- *Rule 3: A transaction cannot request additional locks once it releases any lock!*
  - *Violated!*

2-Phase Locking (2PL): Example

- Applying the 2PL discipline to the schedule

$S = R_1(A)\ R_2(B)\ W_2(B)\ R_2(A)\ W_2(A)\ R_1(B)\ C_1\ C_2$

yields the following interleaved execution (all locks released at commit)

T_1	$RL_1(A)$	$R_1(A)$							
T_2			$RL_2(B)$	$R_2(B)$	$WL_2(B)$	$W_2(B)$	$RL_2(A)$	$R_2(A)$	$WL_2(A)$

T_1		$RL_1(B)$	*wait*	*abort*			*restart*	C_1
T_2	*wait*				$W_2(A)$	C_2		

The deadlock had to be resolved by aborting and restarting one of the transactions.

Under 2PL S is *equivalent* to the *serial schedule* $T_2\ T_1$

2-Phase Locking (2PL): Example

- ***Theorem:*** A schedule that follows 2PL *is always serializable*.
- ***Example:***
  - The schedule $S' = R_1(A)\ R_2(A)\ W_1(A)\ W_2(A)\ C_1\ C_2$ is forced to execute as follows by a transaction scheduler that uses 2PL:

T1	$RL_1(A)$	$R_1(A)$			$WL_1(A)$	wait			abort
T2			$RL_2(A)$	$R_2(A)$			$WL_2(A)$	wait	

T1			restart	C_1
T2	$W_2(A)$	C2		

If no locking was imposed S' would be non-serializable

Transaction Management

- Example: Consider the following transactions
  - T_1: $W_1(U)$ $R_1(Y)$ $W_1(U)$ C_1
  - T_2: $R_2(X)$ $W_2(U)$ $W_2(Y)$ $W_2(W)$ C_2
  - T_3: $W_3(W)$ $R_3(X)$ $W_3(U)$ $W_3(Z)$ C_3
- Is it possible to add lock/unlock steps to these transactions so that every legal schedule is serializable?
- Answer: *yes by adding add lock/unlock steps using 2PL*

Transaction Management

1. T_1: $W_1(U)$ $R_1(Y)$ $W_1(U)$ C_1
2. T_2: $R_2(X)$ $W_2(U)$ $W_2(Y)$ $W_2(W)$ C_2
3. T_3: $W_3(W)$ $R_3(X)$ $W_3(U)$ $W_3(Z)$ C_3

T_1							
T_2					$RL_2(X)$	$WL_2(U)$	wait
T_3	$WL_3(W)$	$RL_3(X)$	$WL_3(U)$	$WL_3(Z)$			$W_3(W)$

T_1							$WL_1(U)$
T_2	wait	wait	wait	wait	wait	$WL_2(Y)$	
T_3	$WU_3(W)$	$R_3(X)$	$RU_3(X)$	$W_3(U)$	$WU_3(U)$		

Transaction Management

1. T_1: $W_1(U)$ $R_1(Y)$ $W_1(U)$ C_1
2. T_2: $R_2(X)$ $W_2(U)$ $W_2(Y)$ $W_2(W)$ C_2
3. T_3: $W_3(W)$ $R_3(X)$ $W_3(U)$ $W_3(Z)$ C_3

T_1	wait	wait	wait	wait	wait	$RL_1(Y)$	wait
T_2	$WL_2(W)$	$R_2(X)$		$W_2(U)$	$WU_2(U)$		$W_2(Y)$
T_3			$W_3(Z)$				

T_1	$WL_1(U)$	wait	$W_1(U)$					$R_1(Y)$	$W_1(U)$
T_2					$W_2(W)$	$WU_2(W)$	C_2		
T_3		$WU_3(Z)$		C_3					

T_1	$WU_1(U)$	$RU_1(Y)$	$WU_1(Z)$	C_1
T_2				
T_3				