CS3402 Chapter 11: Concurrency Control

Database Concurrency Control

- Purposes of Concurrency Control
  - To preserve database consistency
    - To ensure all schedules are serializable and recoverable
  - To maximize the system performance (higher concurrency)

- Example:
  - In concurrent execution environment, if T1 conflicts with T2 over a data item A, then the existing concurrency control decides whether T1 or T2 should get the A and whether the other transaction is rolled-back or waits

Locking Techniques for Concurrency Control

- 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.
- Locking is an operation which secures permission to read or write a data item for a transaction
  - Example:
    - Lock (X). Data item X is locked on behalf of the requesting transaction
- Unlocking is an operation which removes these permissions from the data item
  - Example:
    - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are atomic operations

Binary Lock

- 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.

Binary Lock

- 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(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 finishes accessing the item, it issues an UNLOCK(X) operation, which sets lock of 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.

Implementation of Binary Lock

- The following code performs the lock operation:

```
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);
     goto B
   end;
```

- The following code performs the unlock operation:

```
   LOCK (X) ← 0 (*unlock the item*)
   if any transactions are waiting then
     wake up one of the waiting transactions;
```

Multiple-Mode Locks

- Two lock modes:
  - (a) shared (read)   (b) 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 of Multiple-mode Lock

- The following code performs the read lock operation:

```
B: if LOCK (X) = “unlocked” then
     begin LOCK (X) ← “read-locked”;
         no_of_reads (X) ← 1;
     end
     else if LOCK (X) = “read-locked” then
               no_of_reads (X) ← no_of_reads (X) +1
           else begin wait (until LOCK (X) = “unlocked” / “read-locked”and
              the lock manager wakes up the transaction);
              go to B
           end;
```

Implementation of Multiple-mode Lock

- The following code performs the write lock operation:

```
B: if LOCK (X) = “unlocked” then
        LOCK (X) ← “write-locked”;
    else begin
        wait (until LOCK(X)=“unlocked”
            and the lock manager wakes up the transaction);
        go to B
        end;
```

Implementation of Multiple-mode Lock

- The following code performs the unlock operation:

```
if LOCK (X) = “write-locked” then
 begin LOCK (X) ← “unlocked”;
     wakes up one of the transactions, if any
 end
 else if LOCK (X) = “read-locked” then
     begin
          no_of_reads (X) ← no_of_reads (X) -1
          if  no_of_reads (X) = 0 then
          begin
                  LOCK (X) = “unlocked”;
                 wake up one of the transactions, if any
          end
     end;
```

Implementation Issue

- Lock conversion (read lock to write lock)
  - Lock upgrade: existing read lock to write lock

```
if Ti has a read-lock (X) and Tj has no read-lock (X) (i ≠ j) then
    convert read-lock (X) to write-lock (X)
else
    force Ti to wait until Tj unlocks X
```

  - Lock down grade: existing write lock to read lock

```
if Ti has a write-lock (X) (*no transaction can have any lock on X*)
convert write-lock (X) to read-lock (X)
```

Implementation Issue

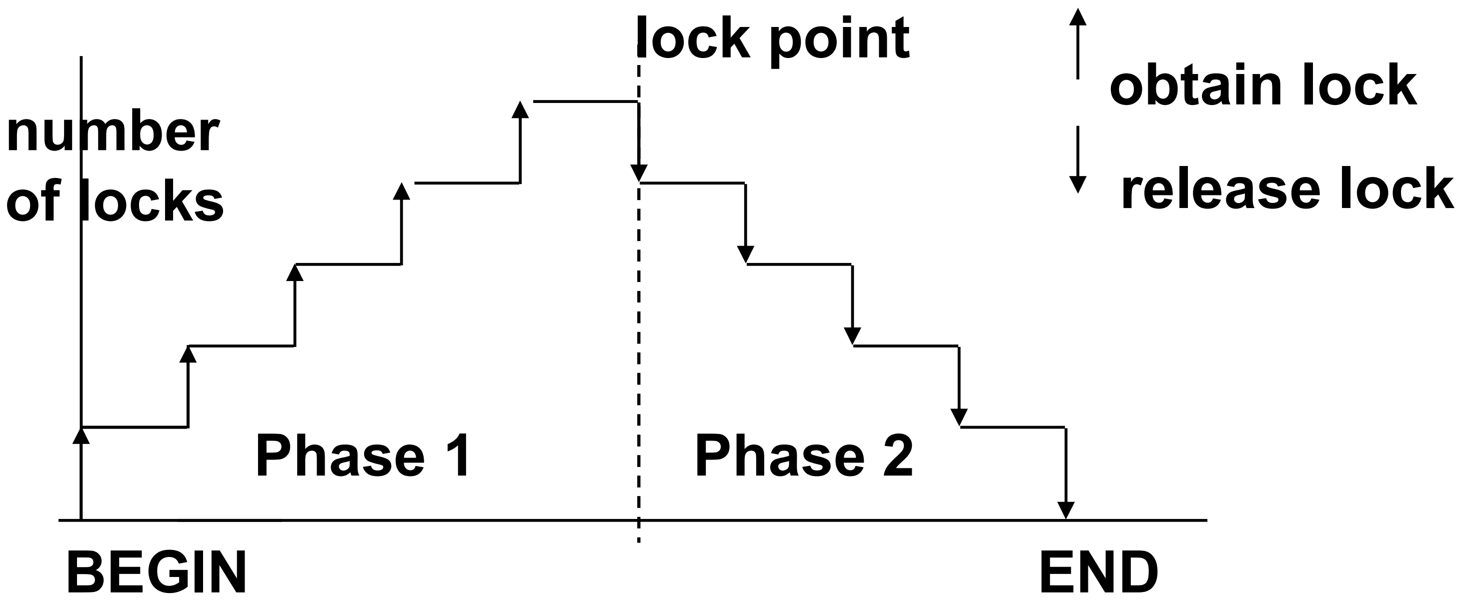
- Lock Manager:
  - Programe managing locks on data items
- Lock table:
  - An array to show the lock entry for each data item
  - Each entry of the array stores the identification of transaction that has set a lock on the data item including the mode
- One entry for one table? One record? One field? Lock granularity (Coarse granularity vs. fine granularity)
- Larger granularity -> higher conflict probability but lower locking overhead

Basic Two Phase Locking (B2PL)

- Definition: 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 readlocked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

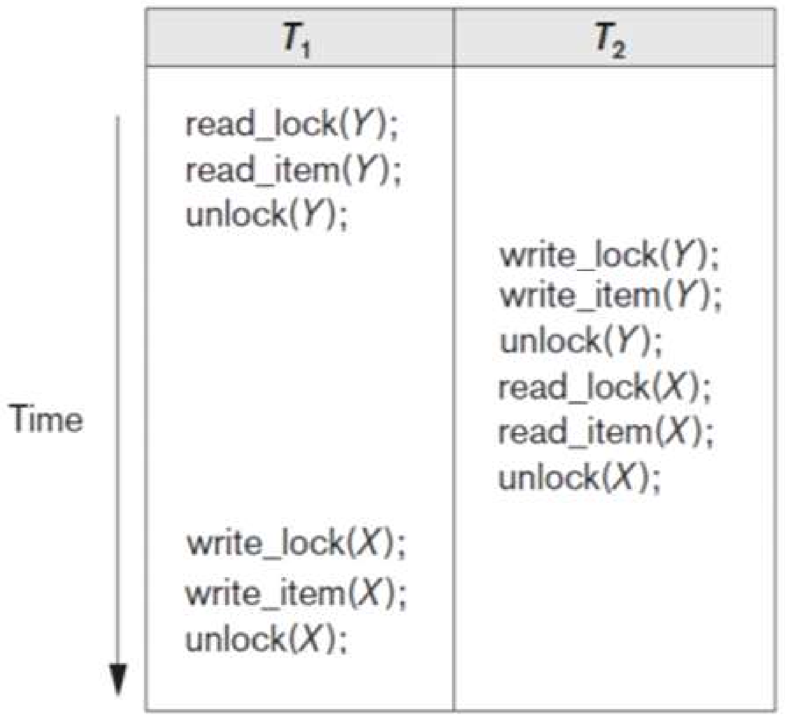
Basic Two Phase Locking (B2PL)

- The two phase rule: growing phase and shrinking phase
- It can guarantee that all pairs of conflicting operations of two transactions are scheduled in the same order, Why?
  - E.g., T1 -> T2 or T2 -> T1 and NO T1<->T2
- B2PL enforce serializability but may produce deadlock.



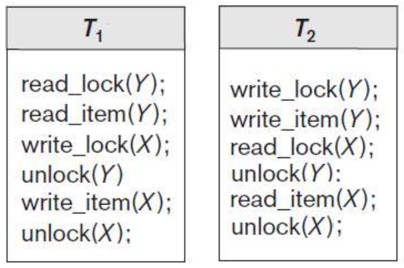
14

Basic Two Phase Locking (B2PL)



Time

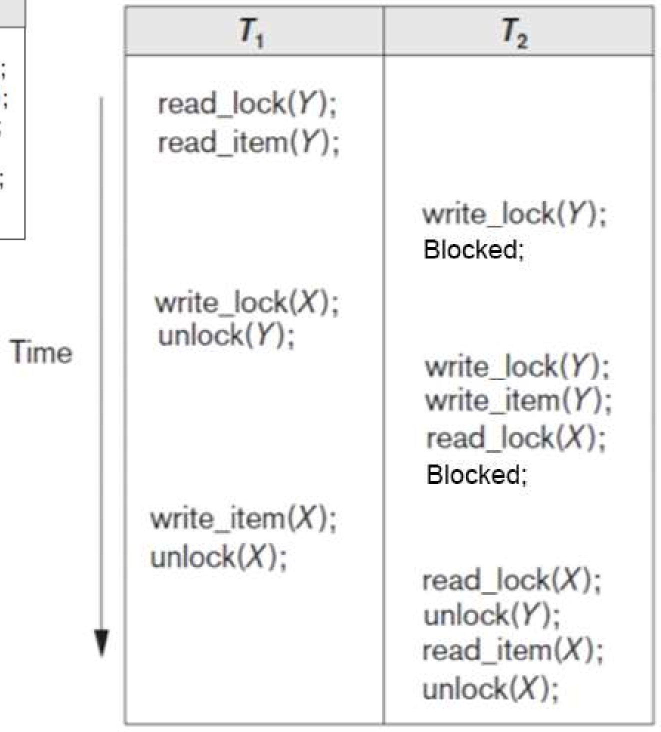
T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	write_lock(Y);
	write_item(Y);
	unlock(Y);
	read_lock(X);
	read_item(X);
	unlock(X);
write_lock(X);	
write_item(X);	
unlock(X);	



T_1
read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
write_item(X);
unlock(X);

T_2
write_lock(Y);
write_item(Y);
read_lock(X);
unlock(Y);
read_item(X);
unlock(X);

B2PL



Time

T_1	T_2
read_lock(Y);	
read_item(Y);	
	write_lock(Y);
	Blocked;
write_lock(X);	
unlock(Y);	
	write_lock(Y);
	write_item(Y);
	read_lock(X);
	Blocked;
write_item(X);	
unlock(X);	
	read_lock(X);
	unlock(Y);
	read_item(X);
	unlock(X);

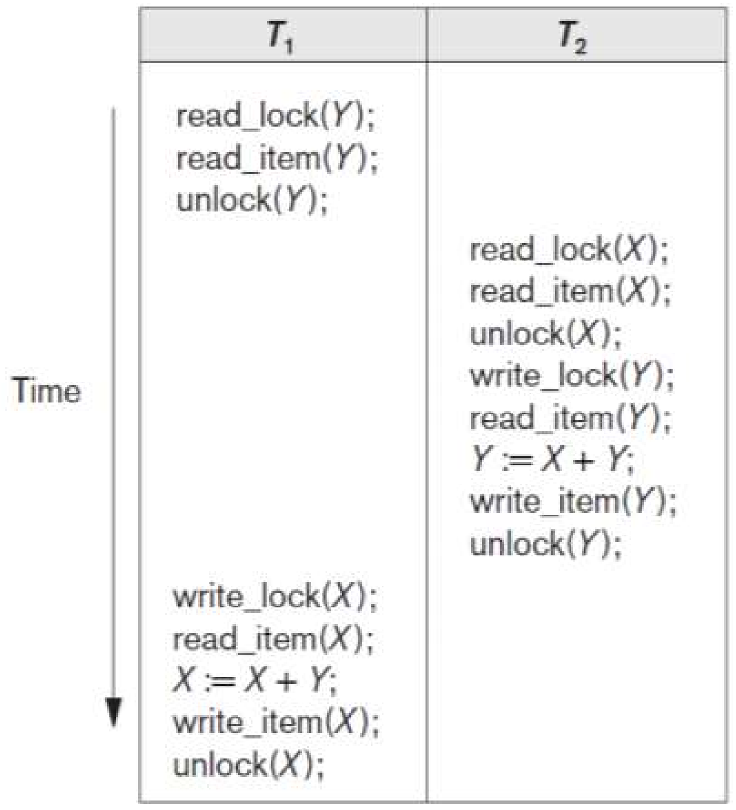
Not follow the B2PL protocol

Cannot guarantee Serializable

Follow the B2PL protocol

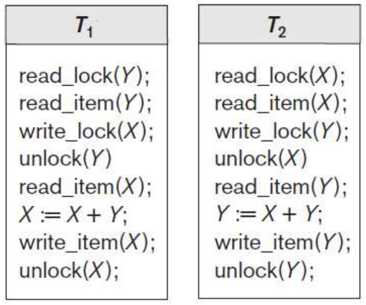
Guarantee Serializable

Basic Two Phase Locking (B2PL)



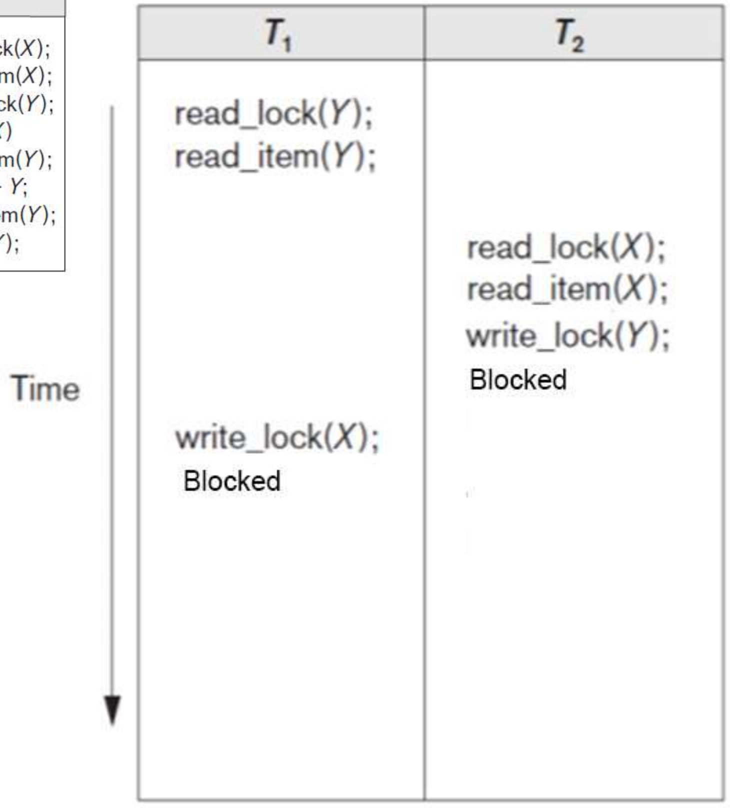
Time

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);	



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

B2PL



Time

T_1	T_2
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
	write_lock(Y);
	Blocked
write_lock(X);	
Blocked	

Not follow the B2PL protocol

Cannot guarantee Serializable

Follow the B2PL protocol

Guarantee Serializable

But produce a deadlock

Conservative Two Phase Locking (C2PL)

- A variation known as conservative 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.

- It set lock of a transaction in one step in the begining of transaction and lock release in another step at the end of the transaction.

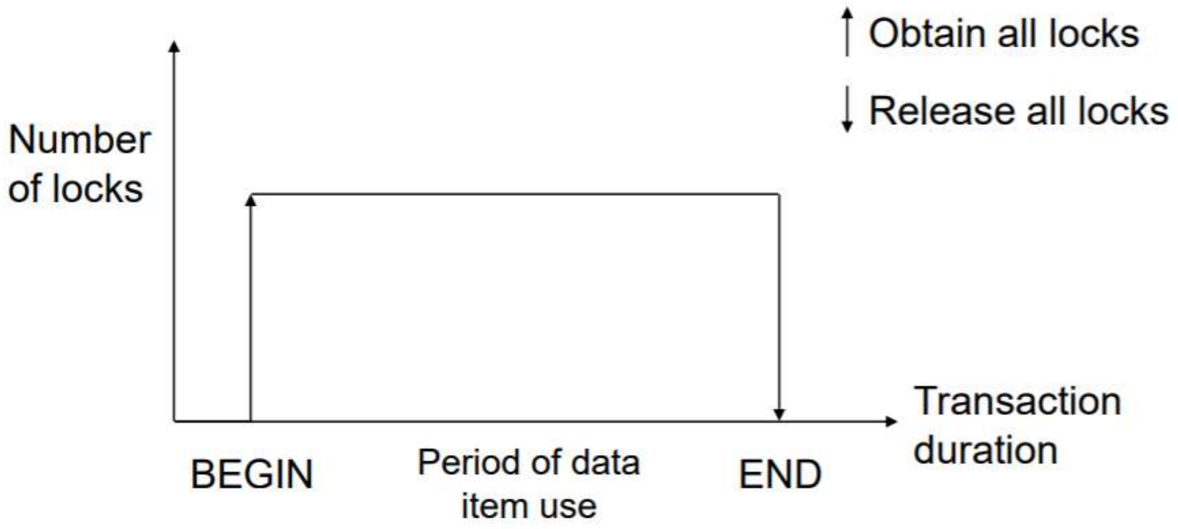
Conservative Two Phase Locking (C2PL)

T_1	T_2
	Write(a)
Write(a)	
	Write (b)
	Commit
Write(b)	
Commit	

$\xrightarrow{\text{C2PL}}$

T_1	T_2
	WriteLock(a,b)
	Write(a)
WriteLock(a,b) => blocked	
	Write (b)
	Commit
	ReleaseLock(a,b)
WriteLock(a,b)	
Write(a)	
Write(b)	
Commit	
ReleaseLock(a,b)	

Conservative Two Phase Locking (C2PL)



- Conservative 2PL is a deadlock-free protocol. In Conservative 2PL, if a transaction T_i is waiting for a lock held by T_j, T_i is holding no locks (no hold and wait situation=> no deadlock).

- 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.

Strict Two Phase Locking (S2PL)

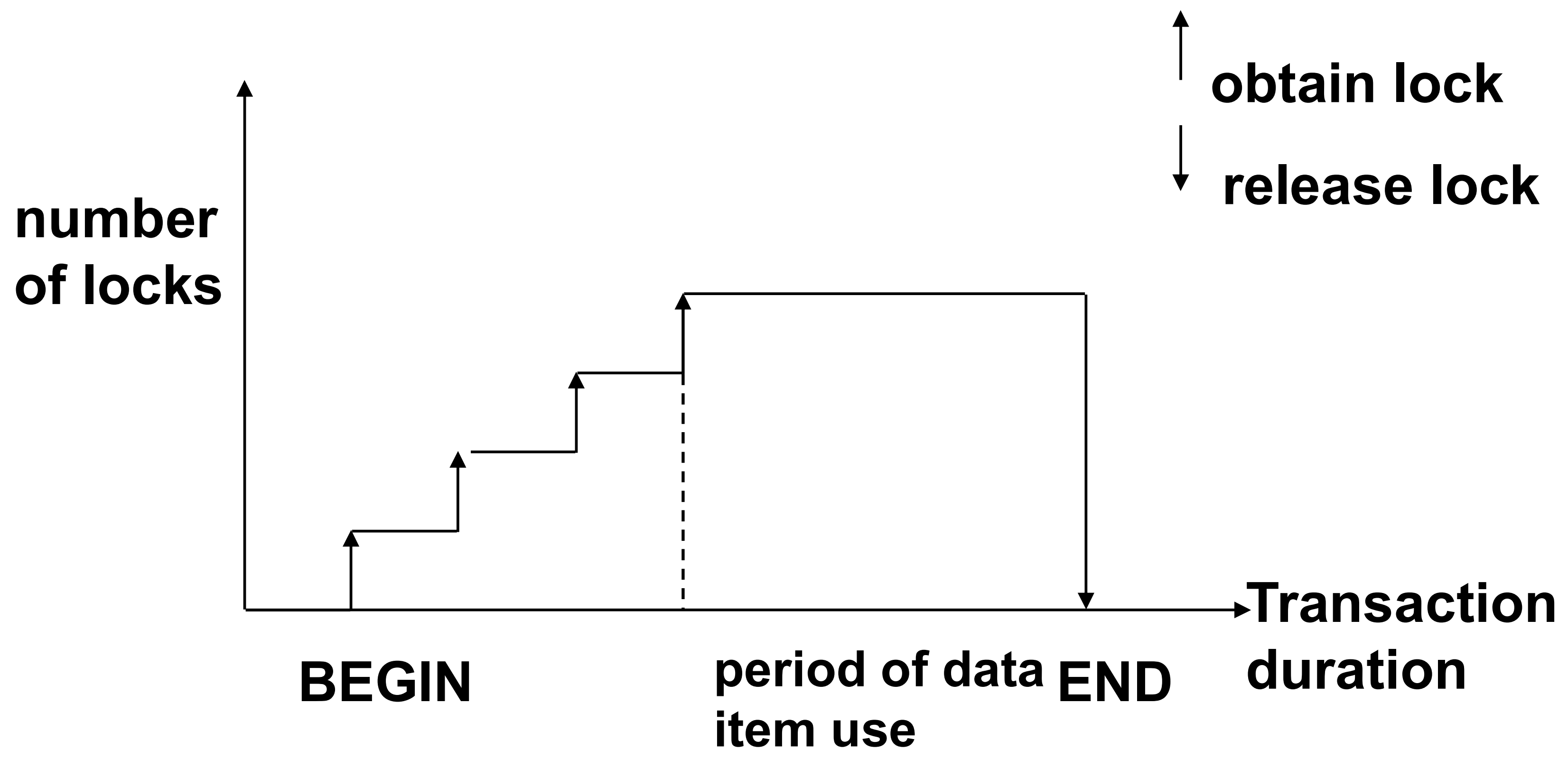
- In practice, the most popular variation of 2PL is strict 2PL, which guarantees strict schedule.

- In this variation, a transaction T does not release any of its 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.

- It may have the problem of deadlock but all schedule is recoverable

- Compared with B2PL, the lock holding time may be longer and the concurrency may be lower
- Compared with C2PL, the lock holding time may be shorter and the concurrency may be higher

Strict Two Phase Locking (S2PL)



Strict Two Phase Locking (S2PL)

T_1	T_2
	Write(a)
Write(a)	
	Write (b)
	Commit
Write(b)	
Commit	

S2PL

T_1	T_2
	WriteLock(a)
	Write(a)
WriteLock(a) => blocked	
	WriteLock(b)
	Write (b)
	Commit
	ReleaseLock(a,b)
WriteLock(a)	
Write(a)	
WriteLock(b)	
Write(b)	
Commit	
ReleaseLock(a,b)	

Performance Issues

- 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.
- In Conservative 2PL, it requires the scheduler to both obtain and release all of a transaction's locks altogether.
- S2PL is better than C2PL when the transaction workload is not heavy since the lock holding time is shorter in S2PL.
- When the transaction is heavy, C2PL is better than S2PL since deadlock may occur in S2PL.

Comparisons

Protocol	Guarantee Serializability	Guarantee Recoverability	Prevent Deadlock	Concurrency
B2PL	YES	No	No	Higher
S2PL	YES	YES	No	Middle
C2PL	YES	YES	YES	Lower

Resolving Deadlock Problem

Deadlock

- 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

T1	**T2**
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (X);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

T1 and T2 do follow two-phase policy but they are deadlock

- Deadlock (T1 and T2)

Deadlock Prevention

- Deadlock condition
  - Hold and wait
  - Cyclic wait

- Deadlock Prevention
  - A transaction locks all data items it refers to before it begins execution
  - The conservative two-phase locking uses this approach
  - This way of locking prevents deadlock since a transaction never hold and wait
  - But the concurrency is low

Deadlock Prevention using TS

- Deadlock prevention: prevent potential deadlock to become deadlock

---Should a transaction be blocked and made to wait or should it be aborted?

- Some of these techniques use the concept of transaction timestamp TS(T), which is a unique identifier assigned to each transaction, e.g., its creation time.
- Notice that the older transaction (which starts first) has the smaller timestamp value. If transaction T1 starts before transaction T2 , then TS(T1) < TS(T2).
- Two schemes using TS to prevent deadlock are called wait-die and wound-wait.

Deadlock Prevention using TS

- Suppose that transaction Ti tries to lock an item X but is not able to because X is locked by some other transaction Tj with a conflicting lock. The rules followed by these schemes are:

- Wait-die Rule : If TS(Ti) < TS(Tj), then (Ti older than Tj) Ti is allowed to wait; otherwise (Ti younger than Tj) abort Ti (Ti 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 Rule : If TS(Ti) < TS(Tj), then (Ti older than Tj) abort Tj (Ti wounds Tj) and restart it later with the same timestamp; otherwise (Ti younger than Tj) Ti is allowed to wait
  - 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 Avoidance using TS

- If TS(Ti) < TS(Tj), Ti waits, else Ti dies (Wait-die)
- If TS(Ti) < TS(Tj), Ti kill Tj, else Ti waits (Wound-wait)
- Note a smaller TS means the transaction is older
- Note both methods restart the younger transaction

- Both methods prevent cyclic wait:
  - Consider this deadlock cycle: T1->T2->T3->…->Tn->T1
  - It is impossible since if T1 …->Tn, then Tn is not allowed to wait for T1
  - Wait-die: only older transaction is allowed to wait
  - Wound-wait: only younger transaction is allowed to wait

Deadlock Example

Transaction U: TS of U < TS of T	Transaction T:
ReadLock (A); Read (A)	
WriteLock(B); Write (B)	
	ReadLock(C): Read (C)
	WriteLock(A); (blocked)
WriteLock(C) (blocked)	
deadlock formed	

Deadlock Example (wait-die)

Transaction U: TS of U < TS of T	Transaction T:
ReadLock (A); Read (A)	
WriteLock(B); Write (B)	
	ReadLock(C); Read (C)
	WriteLock (A) (restarts)
	*** T is restarted since it is younger than U
	*** T releases its read lock on C before restart
WriteLock(C); Write (C)	
Commit; ReleaseLock(A,B,C)	
	ReadLock(C); Read(C)
	….

Deadlock Example (wound-wait)

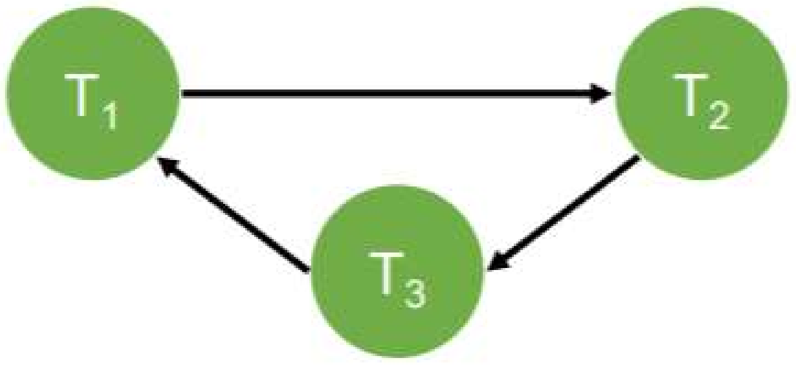
Transaction U: **TS of U < TS of T**	**Transaction T:**
ReadLock (A); Read (A)	
WriteLock(B); Write (B)	
	ReadLock(C); Read (C)
	WriteLock (A) (blocked)
	***** since T is younger than U**
WriteLock(C); Write (C)	
***** T is restarted by U since T is younger than U**	
***** The write lock on C is granted to U after T has released its read lock on C**	
Commit; ReleaseLock(A,B,C)	
	ReadLock(C); Read(C)
	….

Deadlock detection and resolution

- Deadlock Detection
  - In this approach, deadlocks are allowed to happen e.g., in Strict 2PL. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
  - One node is created in the wait-for graph for each transaction that is currently executing.
  - Whenever a transaction Ti is waiting to lock an item X that is currently locked by a transaction Tj , a directed edge ($Ti \rightarrow Tj$) is created in the waitfor graph. When Tj releases the lock(s) on the items that Ti 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 and resolution

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


$$\begin{tikzcd} T_1 \arrow[rr] & & T_2 \arrow[dl] \\ & T_3 \arrow[ul] & \end{tikzcd}$$

wait-for-graph

Starvation

- Starvation
  - In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled back.
  - Starvation occurs when a particular transaction consistently waits or restarts and never gets a chance to proceed further.
  - 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.

References

- 6e
  - *Chapter 21, pages 755-766*