10- Transaction-Dead Lock

School of Computer Science
University of Windsor

Dr Shafaq Khan

Agenda

- **Lecture**
  - 2PL
  - Deadlock
  - Concurrency Control: Timestamping

- **Assignment 2 Quiz**

Announcements

- **Final Report**

  Submission deadline: Sec 1 & 4: Jul 30; Sec 2: Jul 31; Sec 3: Aug 1

- **Test 2** – Saturday, Aug 5th

Introductory Questions

What is the meaning of deadlock and how it can be resolved?

Apart from using locks, what are other concurrency control mechanisms?

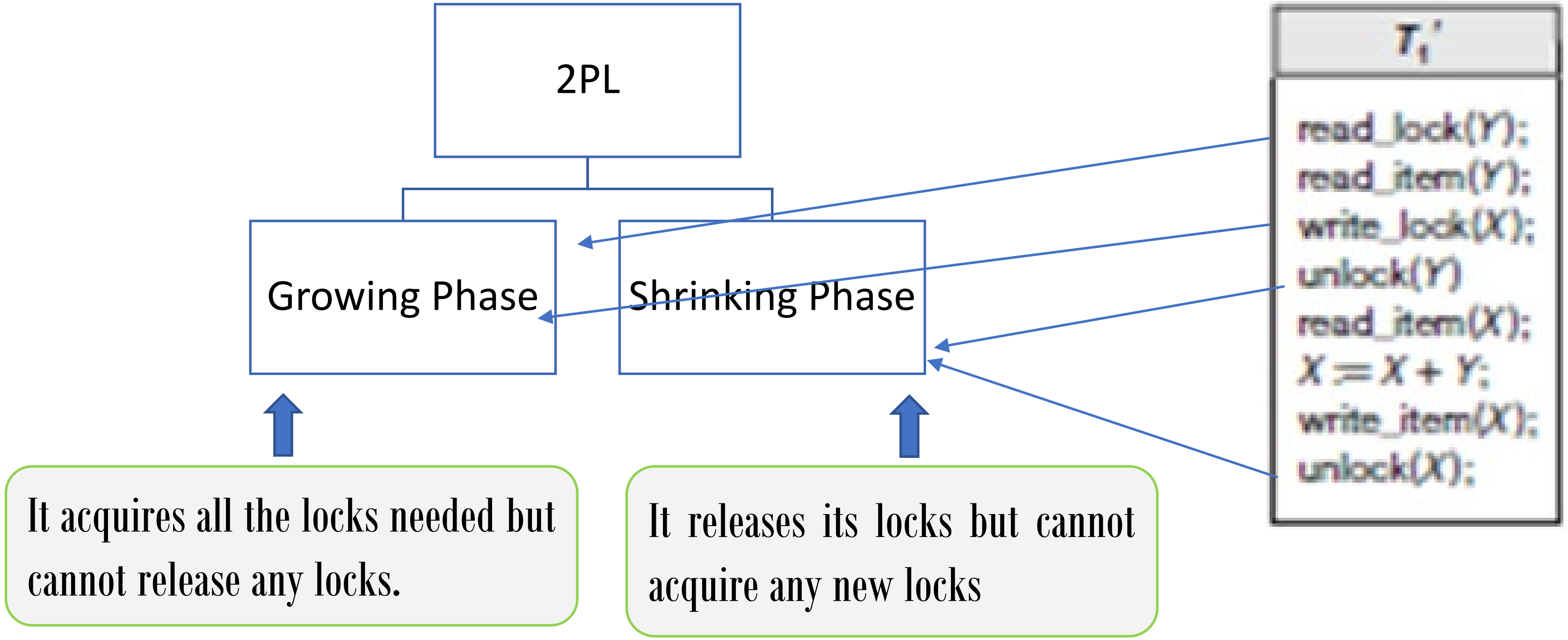
Locks: Problem

Time	T_1	T_2
t_1	BEGIN	
t_2	READ(X)	
t_3	X=X+100	
t_4	WRITE(X)	BEGIN
t_5		READ(X)
t_6		X=X*1.1
t_7		WRITE(X)
t_8		READ(Y)
t_9		Y=Y*1.1
t_{10}		WRITE(Y)
t_{11}	READ(Y)	COMMIT
t_{12}	Y=Y-100	
t_{13}	WRITE(Y)	
t_{14}	COMMIT	

Time	T_1	T_2
t_1	X-LOCK(X)	
t_2	Read(X)	
t_3	Write(X)	
t_4	UNLOCK(X)	
t_5		X-LOCK(X)
t_6		Read(X)
t_7		Write(X)
t_8		UNLOCK(X)
t_9		X-LOCK(Y)
t_{10}		Read(Y)
t_{11}		Write(Y)
t_{12}		UNLOCK(Y)
t_{13}	X-LOCK(Y)	Commit
t_{14}	Read(Y)	
t_{15}	Write(Y)	
t_{16}	UNLOCK(Y)	
t_{17}	Commit	

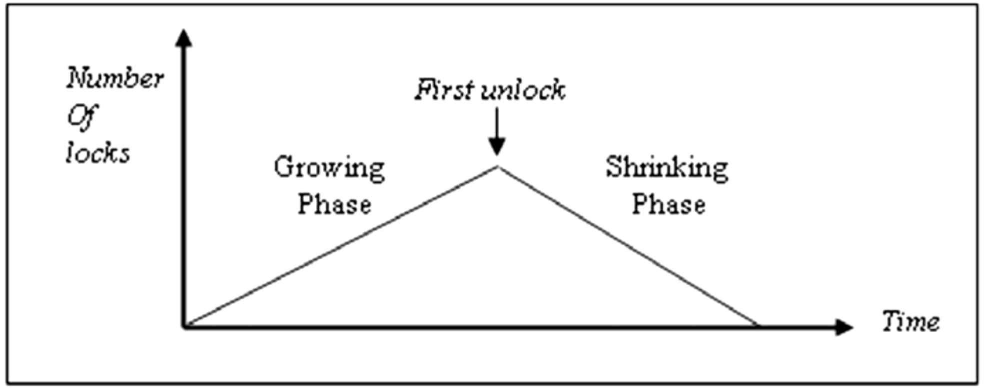
Two-Phase Locking (2PL) Protocol

2PL: A transaction follows the two-phase locking protocol if **all locking operations precede the first unlock** operation in the transaction.

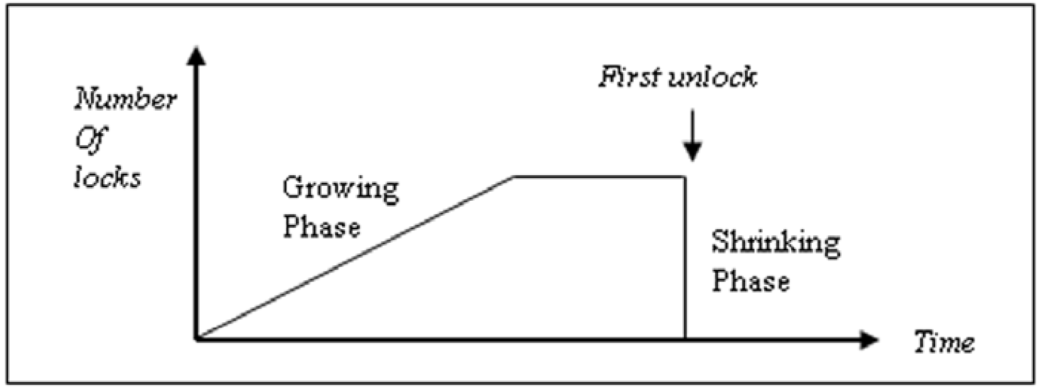


Two-phase locking (2PL)

- ✓ There is no requirement that all locks be obtained simultaneously.
- ✓ Normally, the transaction acquires some locks, does some processing, and goes on to acquire additional locks as needed.
- ✓ However, it never releases any lock until it has reached a stage where no new locks are needed.
- ✓ The rules are:
  - ✓ A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.
  - ✓ Once the transaction releases a lock, it can never acquire any new locks.



(a)



(b)

Two-phase locking (2PL)

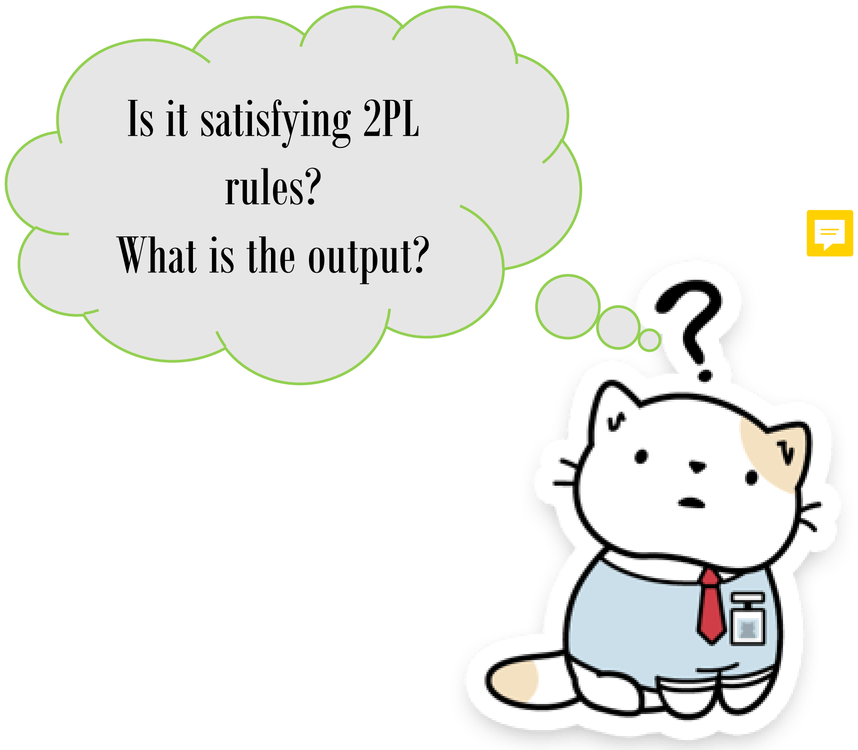
- ✓If upgrading of locks is allowed, upgrading can take place only during the growing phase and may require that the transaction wait until another transaction releases a shared lock on the item.
  - ✓If a transaction has a read lock on a database item, it can upgrade it to a write lock, provided **it is the only transaction** currently having a read lock on the database item.

- ✓Downgrading can take place only during the shrinking phase.
  - ✓Is a simple procedure, since write locks are exclusive locks

Two-phase locking (2PL)

Initial Database state: A=1000, B= 1000

Time	T_1	T_2
t_1	X-LOCK(A)	
t_2	Read(A)	
t_3		S-LOCK(A)
t_4	A=A-100	
t_5	Write(A)	
t_6	UNLOCK(A)	
t_7		Read(A)
t_8		UNLOCK(A)
t_9		S-LOCK(B)
t_{10}	X-LOCK (B)	
t_{11}		Read(B)
t_{12}		UNLOCK (B)
t_{13}	Read(B)	PRINT A+B
t_{14}	B=B+100	Commit
t_{15}	Write(B)	
t_{16}	UNLOCK (B)	
t_{17}	Commit	



Two-phase locking (2PL)

Time	T_1	T_2
t_1	X-LOCK(A)	
t_2	Read(A)	
t_3		S-LOCK(A)
t_4	A=A-100	
t_5	Write(A)	
t_6	X-LOCK (B)	
t_7	UNLOCK(A)	Read(A)
t_8		S-LOCK(B)
t_9	Read(B)	
t_{10}	B=B+100	
t_{11}	Write(B)	Read(B)
t_{12}	UNLOCK (B)	UNLOCK (B)
t_{13}	Commit	UNLOCK(A)
t_{14}		PRINT A+B
t_{15}		Commit

Initial Database state: A=1000, B= 1000

Is it satisfying 2PL rules?
What is the output?

Guaranteeing Serializability by Two-Phase Locking

"If every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable"

i.e the transactions in the schedule can be executed concurrently

University of Windsor

Preventing the lost update problem using 2PL

Problem:

Time	T_1	T_2	X
t_1		BEGIN	100
t_2	BEGIN	READ(X)	100
t_3	READ(X)	X= X+100	100
t_4	X= X-10	WRITE(X)	200
t_5	WRITE(X)	COMMIT	90
t_6	COMMIT		90

Solution:

Time	T_1	T_2	X
t_1		BEGIN	100
t_2	BEGIN	**X_LOCK(X)**	100
t_3	**X_LOCK(X)**	READ(X)	100
t_4		X= X+100	100
t_5		WRITE(X)	200
t_6		COMMIT/**UNLOCK(X)**	200
t_7	READ(X)		200
t_8	X= X-10		200
t_9	WRITE(X)		190
t_{10}	COMMIT/**UNLOCK(X)**		190

Preventing the uncommitted dependency problem using 2PL

Problem:

Time	T_1	T_2	X
t_1		BEGIN	100
t_2		READ(X)	100
t_3		X= X+100	100
t_4	BEGIN	WRITE(X)	200
t_5	READ(X)	…	200
t_6	X= X-10	ROLLBACK	100
t_7	WRITE(X)		190
t_8	COMMIT		190

Solution:

Time	T_1	T_2	X
t_1		BEGIN	100
t_2		**X_LOCK(X)**	100
t_3		READ(X)	100
t_4	BEGIN	X= X+100	200
t_5	**X_LOCK(X)**	WRITE(X)	200
t_6		ROLLBACK/**UNLOCK(X)**	100
t_7	READ(X)		100
t_8	X= X-10		100
	WRITE(X)		90
	COMMIT/**UNLOCK(X)**		90

Preventing the inconsistent analysis problem using 2PL

Problem:

Time	T_1	T_2	X	Y	Z	SUM
t_1		BEGIN	100	50	25	
t_2	BEGIN	SUM=0	100	50	25	0
t_3	READ(X)	READ(X)	100	50	25	0
t_4	X= X-10	SUM=SUM+X	100	50	25	100
t_5	WRITE(X)	READ(Y)	90	50	25	100
t_6	READ(Z)	SUM=SUM+Y	90	50	25	150
t_7	Z=Z+10		90	50	25	150
t_8	WRITE(Z)		90	50	35	150
t_9	COMMIT	READ(Z)	90	50	35	150
t_{10}		SUM=SUM+Z	90	50	35	185
t_{11}		COMMIT	90	50	35	185

Solution:

Time	T_1	T_2	X	Y	Z	SUM
t_1		BEGIN	100	50	25	
t_2	BEGIN	SUM=0	100	50	25	0
t_3	X_LOCK(X)		100	50	25	0
t_4	READ(X)	S_LOCK(X)	100	50	25	0
t_5	X= X-10		90	50	25	0
t_6	WRITE(X)		90	50	25	0
t_7	X_LOCK(Z)		90	50	25	0
t_8	READ(Z)		90	50	25	0
t_9	Z=Z+10		90	50	25	0
t_{10}	WRITE(Z)		90	50	35	0
t_{11}	COMMIT/UNLOCK(X,Z)		90	50	35	0
t_{12}		READ(X)	90	50	35	0
t_{13}		SUM=SUM+X	90	50	35	90
t_{14}		S_LOCK(Y)	90	50	35	90
t_{15}		READ(Y)	90	50	35	90
t_{16}		SUM=SUM+Y	90	50	35	140
t_{17}		S_LOCK(Z)	90	50	35	140
t_{18}		READ(Z)	90	50	35	140
t_{19}		SUM=SUM+Z	90	50	35	175
t_{20}		COMMIT/ UNLOCK(X,Y,Z)	90	50	35	175

Cascading Rollback

- ✓ Transaction T_1 obtains an exclusive lock on X and then updates it using Y, which has been obtained with a shared lock, and writes the value of X back to the database before releasing the lock on X.
- ✓ Transaction T_2 then obtains an exclusive lock on X, reads the value of X from the database, updates it, and writes the new value back to the database before releasing the lock.
- ✓ T_3 share locks X and reads it from the database.
- ✓ By now, T_1 has failed and has been rolled back. However, because T_2 is dependent on T_1 (it has read an item that has been updated by T_1), T_2 must also be rolled back.
- ✓ Similarly, T_3 is dependent on T_2, so it too must be rolled back. This situation, in which a single transaction leads to a series of rollbacks, is called **Cascading rollback**.

Time	T_1	T_2	T_3
t_1			
t_2	BEGIN		
t_3	X_LOCK(X)		
t_4	READ(X)		
t_5	S_LOCK(Y)		
t_6	READ (Y)		
t_7	X=Y+X		
t_8	WRITE(X)		
t_9	UNLOCK(X)	BEGIN	
t_{10}		X_LOCK(X)	
t_{11}		READ(X)	
t_{12}		X=X+100	
t_{13}		WRITE(X)	
t_{14}		UNLOCK(X)	
t_{15}	ROLLBACK		
t_{16}			BEGIN
t_{17}			X_LOCK(X)
t_{18}		ROLLBACK	
t_{19}			ROLLBACK

Cascading rollback

Design protocols that prevent cascading rollbacks. (Cascadeless Schedules)

Solution:

- ✓ **Rigorous 2PL**: hold the release of all locks until the end of the transaction.
- ✓ **Strict 2PL**: holds only exclusive locks until the end of the transaction

TYPES OF TWO-PHASE LOCKING PROTOCOLS

Basic, Conservative, Strict and Rigorous Two-Phase Locking

- **Basic 2PL**
  - The technique discussed so far
- **Conservative 2PL or Static 2PL**
  - Requires **all the data items** required by the transaction to be **locked** before the beginning of the transaction
  - Pre-declares its read-set and write-set
  - If any of the pre-declared items cannot be locked, the transaction does not lock any item and instead **waits** until all items are available
  - **Deadlock-free protocol**

Strict 2PL and Rigorous 2PL

- **Strict 2PL**
  - Write locks are released only after the transaction commits or aborts
  - No other transaction can read or write into an item written by another transaction (until it is committed)
- **Rigorous 2PL**
  - **Both read and write locks** are released only after the transaction commits or aborts

Limitations of 2PL

write_lock(x) x=x+20 write_item(x) write_lock(y) y=y+20 write_item(y) write_lock(z) z=z+20 write_item(z) unlock(x) unlock(y) unlock(z)

Loss of concurrency

Ex: The lock on X cannot be unlocked immediately (due to the protocol) after write_item(x), and therefore other transactions will not be able to access it

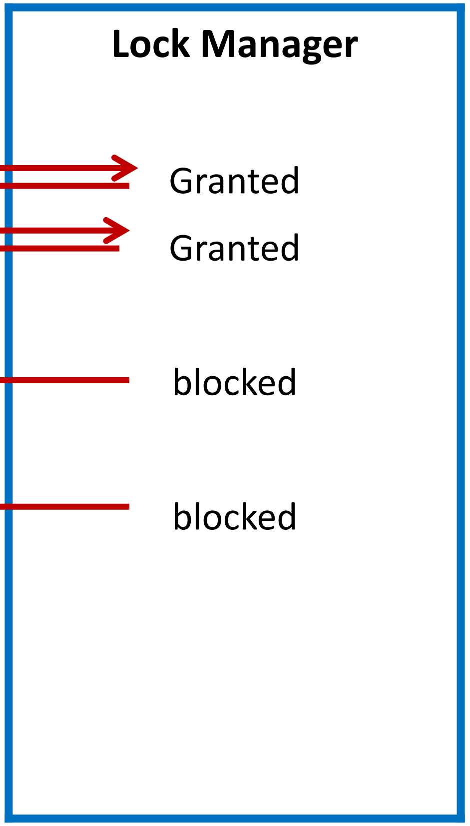
Deadlocks

- Deadlock occurs when ***each*** transaction T in a ***set of two or more transactions*** is waiting for some items that is locked by some other transaction *T*
  - Each transaction is in a waiting queue waiting for another transaction to release the lock on a data item
  - However, since the other transaction is also waiting, it will never release the lock

Deadlock

Problem with two-phase locking, which applies to all locking-based schemes as transactions can wait for locks on data items.

Time	T_1	T_2
t_1	X-LOCK(A)	
t_2		S-LOCK(B)
t_3		Read(B)
t_4		S-LOCK(A)
t_5	Read(A)	
t_6	X-LOCK(B)	
t_7		
t_8		
t_9		
t_{10}		
t_{11}		
t_{12}		



Deadlock

Three general techniques for handling deadlock:

1. Timeouts
2. Deadlock Prevention
3. Deadlock Detection and Recovery.

1. Timeout

- ✓A simple approach to deadlock prevention is based on *lock timeouts*.
- ✓Transaction that requests lock will only wait for a system-defined period of time.
- ✓If lock has not been granted within this period, lock request times out.
- ✓In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.
- ✓This is a very simple and practical solution to deadlock prevention that is used by several commercial DBMSs.

2. Deadlock Prevention

What do we do with a transaction involved in a possible deadlock situation?

- **Transaction timestamp** TS(*T*), which is a unique identifier assigned to each transaction.
- The timestamps are based on the order in which transactions are started
  - If transaction *T*1 starts before transaction *T*2, then TS(*T*1) < TS(*T*2). In this case T1 is the older transaction and T2 is the younger transaction.
- ***Older* transaction has the *smaller* timestamp value.**

✓Assign priorities based on timestamps:

Older Timestamp = Higher Priority (e.g., $T_1 > T_2$)

✓ **Wait-Die ("Old Waits for Young")**

✓ **Wound-Wait ("Young Waits for Old")**

Deadlock Prevention – (Wait-Die)

- Suppose 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 ("Old Waits for Young")**

If $TS(T_1) < TS(T_2)$, then (T_1 *is* older than T_2) then T_1 is requesting an item held by T_2 then

T1 is allowed to wait;

Older transaction waits for the young transaction

- If T_2 (younger transaction) requests an item locked from T_1 (older transaction) then

T_2 should abort and restart it later with the same timestamp.

Younger transaction should abort and restart later with the same time stamp

Time	T_1 (Older)	T_2 (Younger)
t_1	X-LOCK(A)	
t_2	Read(A)	X-LOCK(B)
t_3	X-LOCK(B)-Wait (Older can wait for younger transaction)	Read(B)
t_4		X-LOCK(A) Denied (younger cannot wait for older) ABORT

Deadlock Prevention- (Wound-wait)

- **Wound-Wait ("Young Waits for Old")**
- If $TS(T_1) < TS(T_2)$, then (T_1 older than T_2)
- If T1 requests an item locked by T_2, abort T_2 (T_1 *wounds* T_2)

Older transaction (T_1) aborts the younger transaction (T_2) such that the younger transaction restarts later *with the same timestamp;*

- If T_2 requests an item locked by T_1 then it is allowed to wait.

Young transaction can wait for an older transaction

- Both strategies prevent cycles and deadlocks
- However, both the strategies result in **younger transactions** being aborted (even if they do not lead to deadlocks)

Time	T_1 (Older)	T_2 (Younger)
t_1	X-LOCK(A)	
t_2	Read(A)	X-LOCK(B)
t_3	X-LOCK(B)	ABORT-Older transaction T_1 wounds T_2
t_4	X-LOCK(B) Granted	

3. Deadlock Detection and Recovery

✓ More practical approach to dealing with deadlock is deadlock detection, where the system checks if a state of deadlock actually exists.

✓ Usually handled by construction of **wait-for graph (WFG)** showing transaction dependencies:

- Create a node for each transaction.
- Create edge $T_i \rightarrow T_j$, if T_i waiting to lock item locked by T_j.

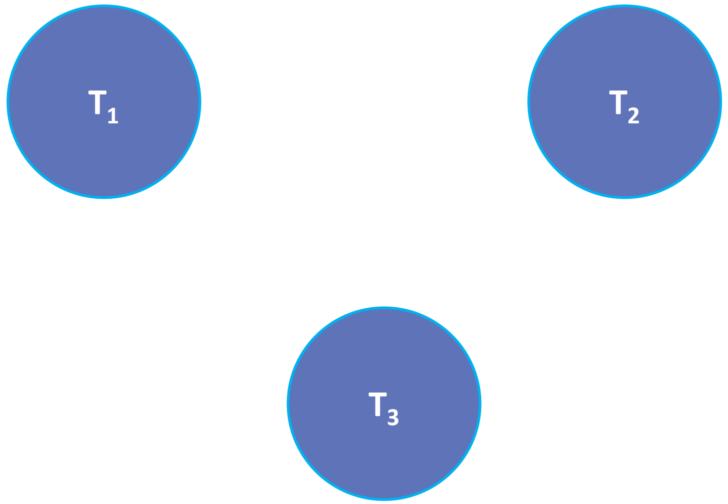
✓ Deadlock exists if and only if WFG contains **cycle**.

✓ WFG is created at regular intervals.

- The system will periodically check for cycles in waits-for graph and then make a decision on how to break it.

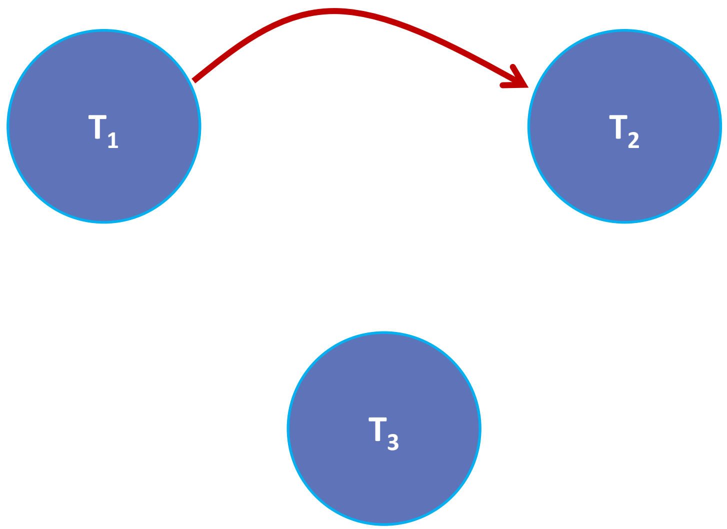
3. Deadlock Detection and Recovery

Time	T_1	T_2	T_3
t_1	**BEGIN**	**BEGIN**	**BEGIN**
t_2	S-LOCK(A)		
t_3		X-LOCK(B)	
t_4			S-LOCK(C)
t_5	S-LOCK(B)		
t_6		X-LOCK(C)	
t_7			X-LOCK(A)
t_8			
t_9			



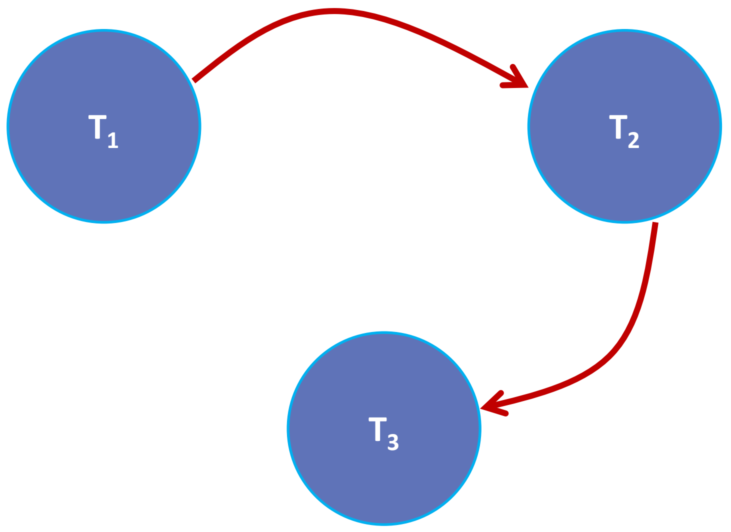
3. Deadlock Detection and Recovery

Time	T_1	T_2	T_3
t_1	BEGIN	BEGIN	BEGIN
t_2	S-LOCK(A)		
t_3		X-LOCK(B)	
t_4			S-LOCK(C)
t_5	S-LOCK(B)		
t_6		X-LOCK(C)	
t_7			X-LOCK(A)
t_8			
t_9			



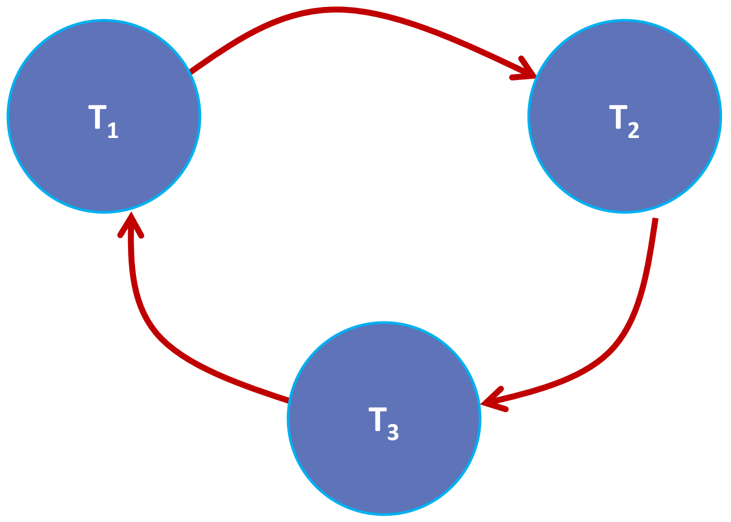
3. Deadlock Detection and Recovery

Time	T_1	T_2	T_3
t_1	BEGIN	BEGIN	BEGIN
t_2	S-LOCK(A)		
t_3		X-LOCK(B)	
t_4			S-LOCK(C)
t_5	S-LOCK(B)		
t_6		X-LOCK(C)	
t_7			X-LOCK(A)
t_8			
t_9			



3. Deadlock Detection and Recovery

Time	T_1	T_2	T_3
t_1	**BEGIN**	**BEGIN**	**BEGIN**
t_2	S-LOCK(A)		
t_3		X-LOCK(B)	
t_4			S-LOCK(C)
t_5	S-LOCK(B)		
t_6		X-LOCK(C)	
t_7			X-LOCK(A)
t_8			
t_9			



Clearly, the graph has a cycle in it ($T_1 \rightarrow T_2 \rightarrow T_3$), so we can conclude that the system is in deadlock.

Recovery from Deadlock Detection

- ✓Once deadlock has been detected the DBMS needs to abort one or more of the transactions.
- ✓There are several issues that need to be considered:
  - o Choice of deadlock victim;
  - o How far to roll a transaction back;
  - o Avoiding starvation

University of Windsor

Choice of deadlock victim

- ✓ In some circumstances, the choice of transactions to abort may be obvious.
- ✓ However, in other situations, the choice may not be so clear.
- ✓ In such cases, we would want to abort the transactions that incur the minimum costs.
- ✓ This may take into consideration:
  - how long the transaction has been running
  - how many data items have been updated by the transaction
  - how many data items the transaction is still to update

University of Windsor

How far to roll a transaction back

- ✓Having decided to abort a particular transaction, we have to decide how far to roll the transaction back.
- ✓Clearly, undoing all the changes made by a transaction is the simplest solution, although not necessarily the most efficient.
- ✓It may be possible to resolve the deadlock by rolling back only part of the transaction.

Avoiding starvation

- ✓Starvation occurs when the same transaction is always chosen as the victim, and the transaction can never complete.
- ✓Solution:
  - ✓The DBMS can avoid starvation by storing a count of the number of times a transaction has been selected as the victim and using a different selection criterion once this count reaches some upper limit.

Summary

We discussed the solution for concurrency control problems: 2PL.
2PL leads to Deadlock: We defined Deadlock and discussed the solutions for deadlock.
We finally discussed another solution for concurrency control problems: Timestamping.

Any Questions