Database Management Systems

Transactions – 2PL & Deadlocks

Overview

- Serializability
- 2PL
- Deadlocks

Locking

- Locking is a procedure used to control concurrent access to data (to ensure serializability of concurrent transactions)
- In order to use a ‘resource’ (table, row, etc) a transaction must first acquire a *lock* on that resource
- This may deny access to other transactions to prevent incorrect results

Two types of locks

- Two types of lock
  - Shared lock (S-lock or read-lock)
  - Exclusive lock (X-lock or write-lock)
- Read lock allows several transactions simultaneously to read a resource (but no transactions can change it at the same time)
- Write lock allows one transaction exclusive access to write to a resource. No other transaction can read this resource at the same time.
- The lock manager in the DBMS assigns locks and records them in the data dictionary

Locking

- Before reading from a resource a transaction must acquire a read-lock
- Before writing to a resource a transaction must acquire a write-lock
- Locks are released on commit/rollback
- A transaction may not acquire a lock on any resource that is write-locked by another transaction
- A transaction may not acquire a write-lock on a resource that is locked by another transaction
- If the requested lock is not available, transaction waits

Two-Phase Locking

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

- Two phases
  - **Growing phase** where locks are acquired on resources
  - **Shrinking phase** where locks are released

Example

T1	T2
read-lock(X)	read-lock(X)
Read(X)	Read(X)
write-lock(Y)	unlock(X)
unlock(X)	write-lock(Y)
Read(Y)	Read(Y)
Y = Y + X	Y = Y + X
Write(Y)	Write(Y)
unlock(Y)	unlock(Y)

Example

- T1 follows 2PL protocol
  - All of its locks are acquired before it releases any of them
- T2 does not
  - It releases its lock on X and then goes on to later acquire a lock on Y

T1	T2
read-lock(X)	read-lock(X)
Read(X)	Read(X)
write-lock(Y)	unlock(X)
unlock(X)	write-lock(Y)
Read(Y)	Read(Y)
Y = Y + X	Y = Y + X
Write(Y)	Write(Y)
unlock(Y)	unlock(Y)

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1	**T2**	**Result**
read_lock (Y);	read_lock (X);	Initial values: X=20; Y=30
read_item (Y);	read_item (X);	Result of serial execution
unlock (Y);	unlock (X);	T1 followed by T2
write_lock (X);	Write_lock (Y);	X= _____, Y= _____.
read_item (X);	read_item (Y);	Result of serial execution
X:=X+Y;	Y:=X+Y;	T2 followed by T1
write_item (X);	write_item (Y);	X=_____, Y= _____
unlock (X);	unlock (Y);	

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1	**T2**	**Result**
read_lock (Y);	read_lock (X);	Initial values: X=20; Y=30
read_item (Y);	read_item (X);	Result of serial execution
unlock (Y);	unlock (X);	T1 followed by T2
write_lock (X);	Write_lock (Y);	X=50, Y=80.
read_item (X);	read_item (Y);	Result of serial execution
X:=X+Y;	Y:=X+Y;	T2 followed by T1
write_item (X);	write_item (Y);	X=70, Y=50
unlock (X);	unlock (Y);	

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1	T2
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);	

Time

Result

X=_____; Y=_____

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1	T2
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);	

Time

Result

X=50; Y=50
Nonserializable because it.
violated two-phase policy.

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T'1

```
read_lock (Y);
read_item (Y);
write_lock (X);
unlock (Y);
read_item (X);
X:=X+Y;
write_item (X);
unlock (X);
```

T'2

```
read_lock (X);
read_item (X);
Write_lock (Y);
unlock (X);
read_item (Y);
Y:=X+Y;
write_item (Y);
unlock (Y);
```

T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

Serialisability Theorem

- Any schedule of two-phased transactions is conflict serialisable

Lost Update

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

Lost Update

- T1 and T2 read X, both modify it, then both write it out
  - The net effect of T1 and T2 should be no change on X
  - Only T2's change is seen, however, so the final value of X has increased by 5

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

Lost Update

This update
Is lost

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

Only this update succeeds

Lost Update can't happen with 2PL

read-lock(X)

cannot acquire write-lock(X): T2 has read-lock(X)

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

read-lock(X)

cannot acquire write-lock(X): T1 has read-lock(X)

Uncommitted Update

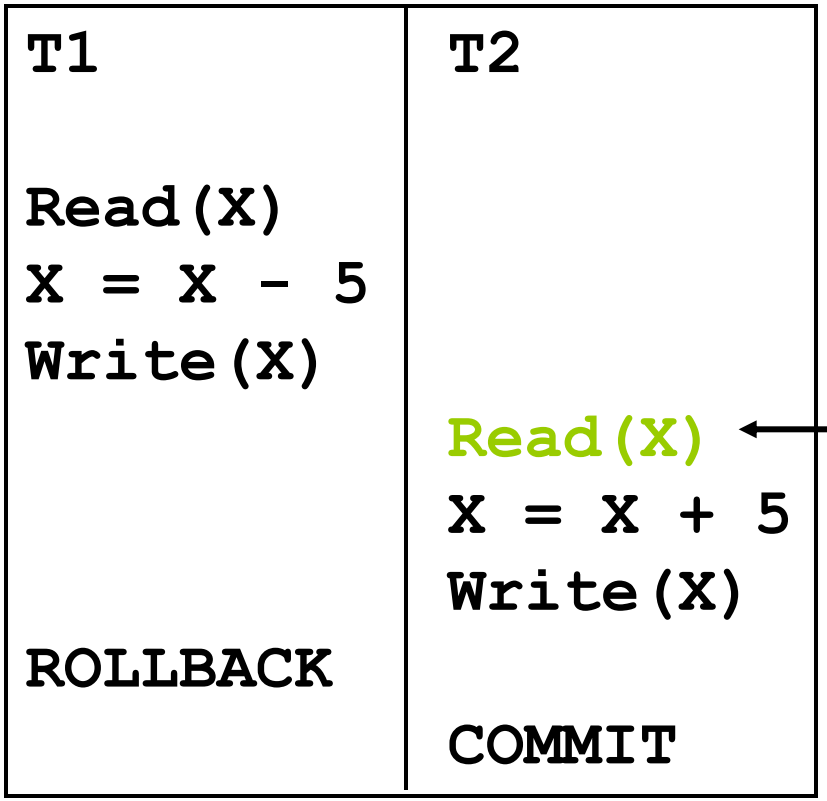
T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	X = X + 5
	Write(X)
ROLLBACK	
	COMMIT

Uncommitted Update

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	X = X + 5
	Write(X)
ROLLBACK	
	COMMIT

- T2 sees the change to X made by T1, but T1 is rolled back
  - The change made by T1 is undone on rollback
  - It should be as if that change never happened

Uncommitted Update ("dirty read")



Uncommitted Update cannot happen with 2PL

	T1	T2
read-lock(X)	Read(X)	
	X = X - 5	
write-lock(X)	Write(X)	
		Read(X)
		X = X + 5
		Write(X)
Locks released	ROLLBACK	
		COMMIT

Waits till T1 releases write-lock(X)

Inconsistent analysis

- T1 doesn't change the sum of X and Y, but T2 sees a change
  - T1 consists of two parts – take 5 from X and then add 5 to Y
  - T2 sees the effect of the first, but not the second

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	Read(Y)
	Sum = X+Y
Read(Y)	
Y = Y + 5	
Write(Y)	

Inconsistent analysis

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	Read(Y)
	Sum = X+Y
Read(Y)	
Y = Y + 5	
Write(Y)	

Summing up data while it is being updated

Inconsistent analysis cannot happen with 2PL

	T1	T2
read-lock(X)	Read(X)	
	X = X - 5	
write-lock(X)	Write(X)	
		Read(X)
		Read(Y)
		Sum = X+Y
read-lock(Y)	Read(Y)	
	Y = Y + 5	
write-lock(Y)	Write(Y)	

Waits till T1 releases write-locks on X and Y

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

- Two-phase policy generates two locking algorithms
  - (a) **Basic**
  - (b) **Conservative**
- **Conservative**:
  - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Basic**:
  - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Strict**:
  - A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Deadlocks

- A deadlock is an impasse that may result when two or more transactions are waiting for locks to be released which are held by each other.
  - For example: T1 has a lock on X and is waiting for a lock on Y, and T2 has a lock on Y and is waiting for a lock on X.
- Given a schedule, we can detect deadlocks which will happen in this schedule using a *wait-for graph* (WFG).

Database Concurrency Control

Dealing with Deadlock and Starvation

- **Deadlock**

T'1	**T'2**
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (Y);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

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

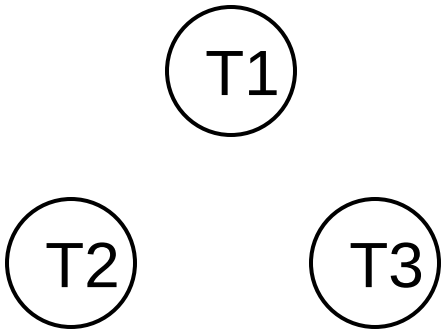
- Deadlock (T'1 and T'2)

Precedence/Wait-For Graphs

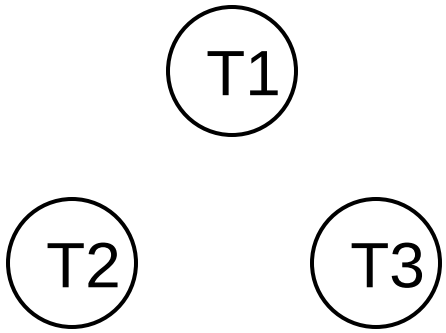
- Precedence graph
  - Each transaction is a vertex
  - Arcs from T1 to T2 if
    - T1 reads X before T2 writes X
    - T1 writes X before T2 reads X
    - T1 writes X before T2 writes X
- Wait-for Graph
  - Each transaction is a vertex
  - Arcs from T2 to T1 if
    - T1 read-locks X then T2 tries to write-lock it
    - T1 write-locks X then T2 tries to read-lock it
    - T1 write-locks X then T2 tries to write-lock it

Example

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



Wait for graph



Precedence graph

Example

T1 Read(X)

T2 Read(Y)

T1 Write(X)

T2 Read(X)

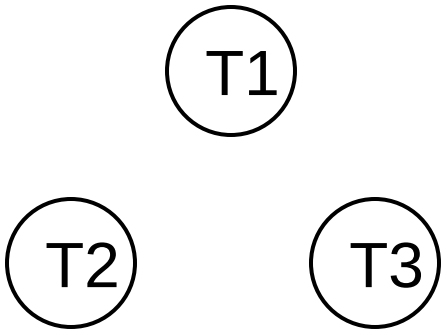
T3 Read(Z)

T3 Write(Z)

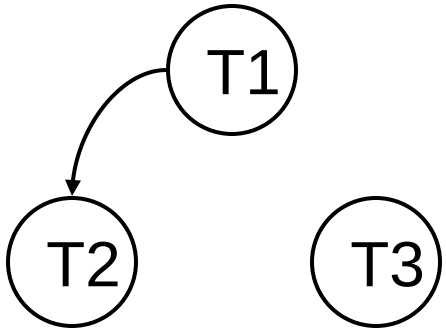
T1 Read(Y)

T3 Read(X)

T1 Write(Y)



Wait for graph



Precedence graph

Example

T1 Read(X)

T2 Read(Y)

T1 Write(X)

T2 Read(X)

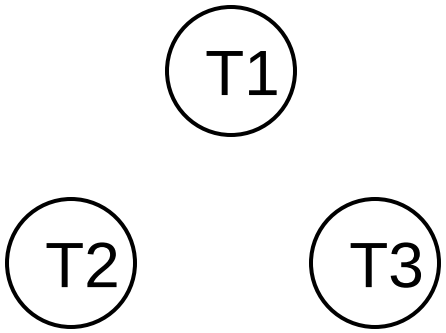
T3 Read(Z)

T3 Write(Z)

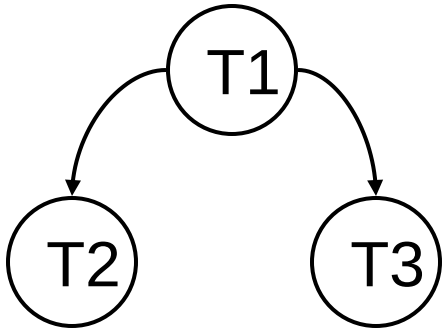
T1 Read(Y)

T3 Read(X)

T1 Write(Y)



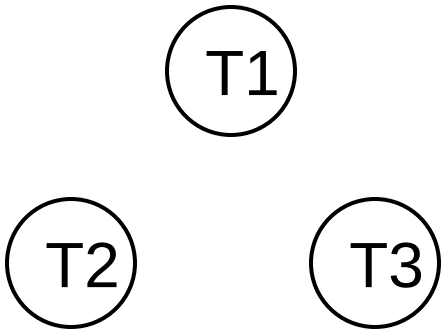
Wait for graph



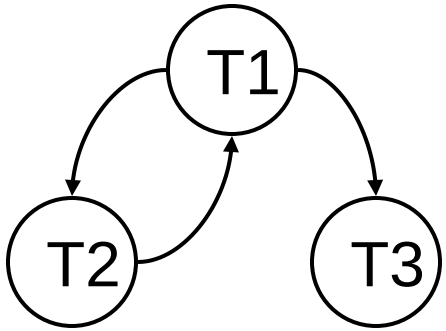
Precedence graph

Example

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



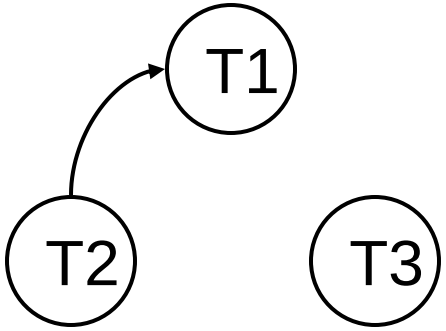
Wait for graph



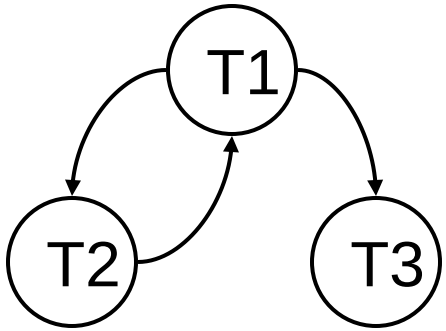
Precedence graph

Example

T1 Read(X) read-locks(X)
T2 Read(Y) read-locks(Y)
T1 Write(X) **write-lock(X)**
T2 Read(X) **tries read-lock(X)**
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



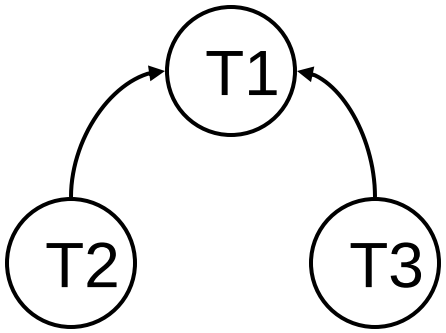
Wait for graph



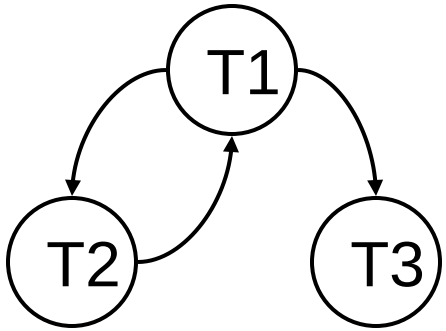
Precedence graph

Example

T1 Read(X) read-locks(X)
T2 Read(Y) read-locks(Y)
T1 Write(X) **write-lock(X)**
T2 Read(X) tries read-lock(X)
T3 Read(Z) read-lock(Z)
T3 Write(Z) write-lock(Z)
T1 Read(Y) read-lock(Y)
T3 Read(X) **tries read-lock(X)**
T1 Write(Y)



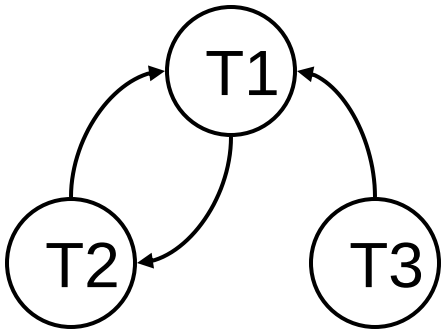
Wait for graph



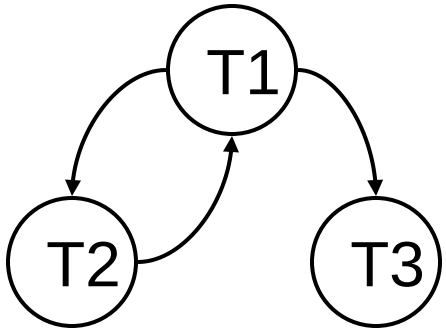
Precedence graph

Example

T1 Read(X) read-locks(X)
T2 Read(Y) **read-locks(Y)**
T1 Write(X) write-lock(X)
T2 Read(X) tries read-lock(X)
T3 Read(Z) read-lock(Z)
T3 Write(Z) write-lock(Z)
T1 Read(Y) read-lock(Y)
T3 Read(X) tries read-lock(X)
T1 Write(Y) **tries write-lock(Y)**



Wait for graph



Precedence graph

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order.

Database Concurrency Control

- **Deadlock prevention**
  - A transaction locks all data items it refers to before it begins execution.
  - This way of locking prevents deadlock since a transaction never waits for a data item.
  - The conservative two-phase locking uses this approach.

Database Concurrency Control

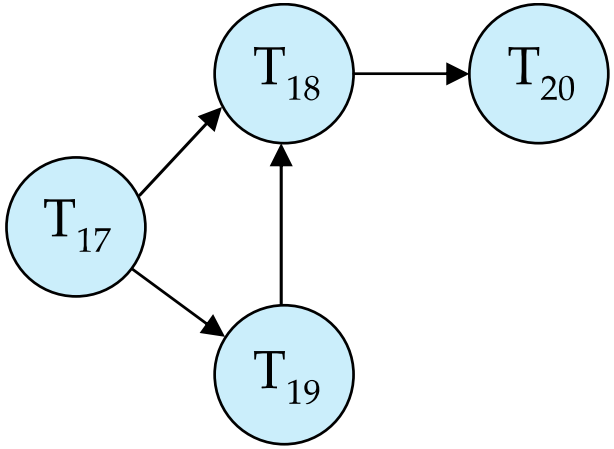
Dealing with Deadlock and Starvation

- **Deadlock detection and resolution**
  - In this approach, deadlocks are allowed to happen. 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.
  - A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle.

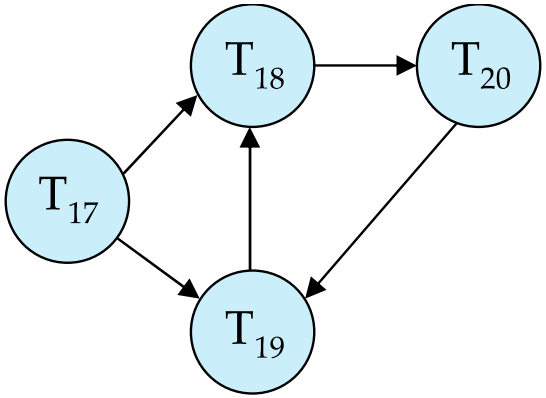
Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Database Concurrency Control

Dealing with Deadlock and Starvation

- **Deadlock avoidance**
  - There are many variations of two-phase locking algorithm.
  - Some avoid deadlock by not letting the cycle to complete.
  - That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
  - Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

Database Concurrency Control

Dealing with Deadlock and Starvation

- **Starvation**
  - Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
  - In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
  - This limitation is inherent in all priority based scheduling mechanisms.
  - In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Deadlock Prevention

- Deadlocks can arise with 2PL
  - Deadlock is less of a problem than an inconsistent DB
  - We can detect and recover from deadlock
  - It would be nice to avoid it altogether
- Conservative 2PL
  - All locks must be acquired before the transaction starts
  - Hard to predict what locks are needed
  - Low ‘lock utilisation’ - transactions can hold on to locks for a long time, but not use them much

Deadlock Prevention

- We impose an ordering on the resources
  - Transactions must acquire locks in this order
  - Transactions can be ordered on the last resource they locked
- This prevents deadlock
  - If T1 is waiting for a resource from T2 then that resource must come after all of T1's current locks
  - All the arcs in the wait-for graph point 'forwards' - no cycles

More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **wait-die** scheme — non-preemptive
  - older transaction may wait for younger one to release data item. (older means smaller timestamp) Younger transactions never Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- **Timeout-Based Schemes:**
  - a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
  - Thus, deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Summary

- Schedules
- Types of Serializability
- 2PL
- Deadlock

Test your understanding

1. Define the following terms: precedence graph, wait-for-graph
2. What are the properties of transaction?
3. What do you mean by serializability.
4. Discuss the types of serializability.

References

1. Ramez Elmasri and Shamkant B. Navathe, “Fundamentals of Database Systems”, Fifth Edition, Pearson Education, 2008.
2. Abraham Silberschatz, Henry F. Korth and S. Sudharshan, “Database System Concepts”,Sixth Edition, Tata McGraw Hill, 2011.