Issues with locks

Topic 6, Lesson 4
Deadlock and its algorithms

Deadlock

An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock($\mathbf{bal_x}$)	begin_transaction
t_3	read($\mathbf{bal_x}$)	write_lock($\mathbf{bal_y}$)
t_4	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	read($\mathbf{bal_y}$)
t_5	write($\mathbf{bal_x}$)	$\mathbf{bal_y} = \mathbf{bal_y} + 100$
t_6	write_lock($\mathbf{bal_y}$)	write($\mathbf{bal_y}$)
t_7	WAIT	write_lock($\mathbf{bal_x}$)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Dealing with deadlocks

Only one way to break a deadlock: abort one or more of the transactions.

Deadlock should be transparent to a user, so the DBMS should restart the aborted transaction(s).

However, in practice the DBMS cannot restart the aborted transaction since it is unaware of transaction logic. Even if it was aware of the transaction history (unless there is no user input in the transaction, or the input is not a function of the database state).

Algorithmic solutions for deadlocks

Three general techniques for handling deadlock:

- Timeouts.
- Deadlock prevention.
- Deadlock detection and recovery.

Timeouts

Transaction that requests lock will only wait for a system-defined time period.

If lock has not been granted within this period, lock request times out.

In this case, DBMS **assumes transaction is deadlocked**, even though it may not be, and it aborts and automatically restarts the transaction.

This approach penalizes long-running transactions and aborts transactions that may not be deadlocked

Deadlock prevention

DBMS looks ahead to see if a transaction would cause a deadlock, and if so, **does not allow deadlock to occur**.

Algorithm: order the transactions by timestamps. The value of the timestamp determines who can wait on whom.

- **Wait-Die** - only an older transaction can wait for younger one, otherwise transaction is aborted (**dies**) and restarted with same timestamp.
- **Wound-Wait** - only a younger transaction can wait for an older transaction. If older transaction requests lock held by younger one, younger one is aborted (**wounded**).

Deadlock prevention alternative

Conservative two-phase locking (C2PL)

A transaction must request all locks at the beginning of the transaction. If it does not receive all requested locks, it must wait until all locks can be granted.

A transaction that holds locks never waits on another transaction – they wait when acquiring all the locks they need.

Works best in systems with heavy lock contention

Deadlock Detection and Recovery

DBMS allows deadlock to occur but recognizes it and breaks it.

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

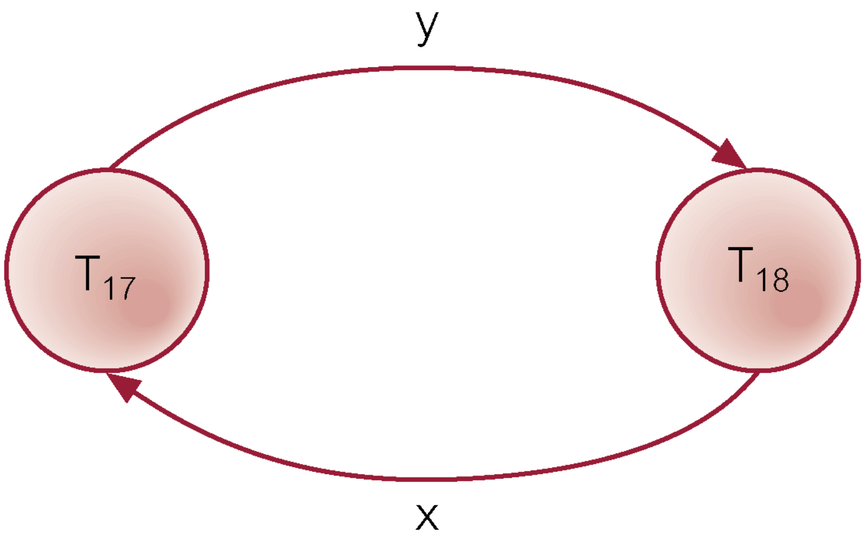
Created a directed graph where:

- Each transaction is represented as a node
- Create an edge $T_i \rightarrow T_j$ if T_i is waiting to lock an item that is locked by T_j
- Deadlock exists if and only if W F G contains cycle.
- WFG is created at regular intervals.

Example: wait-for-graph

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock($\mathbf{bal_x}$)	begin_transaction
t_3	read($\mathbf{bal_x}$)	write_lock($\mathbf{bal_y}$)
t_4	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	read($\mathbf{bal_y}$)
t_5	write($\mathbf{bal_x}$)	$\mathbf{bal_y} = \mathbf{bal_y} + 100$
t_6	write_lock($\mathbf{bal_y}$)	write($\mathbf{bal_y}$)
t_7	WAIT	write_lock($\mathbf{bal_x}$)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Deadlock?

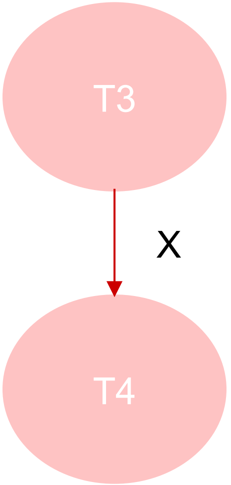


Yes, since there is a cycle

Wait-for graph Example 2

Time	T_3	T_4
t_1		begin_transaction
t_2		write_lock(bal_x)
t_3		read(bal_x)
t_4	begin_transaction	$bal_x = bal_x + 100$
t_5	write_lock(bal_x)	write(bal_x)
t_6	WAIT	rollback/unlock(bal_x)
t_7	read(bal_x)	
t_8	$bal_x = bal_x - 10$	
t_9	write(bal_x)	
t_{10}	commit/unlock(bal_x)	

Deadlock?



No, there is no cycle

Recovery from deadlock detection

Several issues:

- choice of deadlock victim: need an algorithmic method for choosing which transaction to restart
- how far to roll a transaction back: what work need not be repeated by the application
- avoiding starvation: starvation occurs when the same transaction repeatedly is restarted

Summary

- Locks mark objects as being used by a transaction however locks have complications
- Locks lead to deadlocks
  - DBMS must choose a method for dealing with deadlocks
  - We can **prevent** deadlocks by placing an ordering on the transactions and allowing the order to determine which transactions can wait (Wound-Wait) (Wait-Die)
  - We can **detect** a deadlock via a wait-for graph
  - Using timeouts to address deadlocks is not a viable solution