Concurrency

Gary KL Tam

Department of Computer Science
Swansea University

This lecture

Overview

- Concurrency Protocols
- Involves two parts:
  - Locking Protocol
  - Deadlock Detection and Handling

Concurrency Control - Review

- We already learned that every schedule executed by the database must be serializable.
- We discuss one important type of schedule: ***conflict seriablisable schedule***
- Important questions
  - How do we test a schedule is not conflict serialisable?
    - Detecting cycle in Precedence Graph (DONE)
  - How do we construct conflict serialisable schedules?
    - focus of this lecture

Motivation

- Assume that the database receives several transactions now, and it needs to find a serializable schedule.

- Two possible ways:
  - **Analyze** the statements of all transactions before execution. Rearrange operations.
    - But the analysis time may be **too long**.

Our mum/ super nan use their experiences to prepare muffins! (plan in advance)

  - **Run** whatever statement that comes next immediately.
    - Fast but needs to apply some "execution rules".
    - ===> **Concurrency protocols**.

DBMS does this!

This lecture

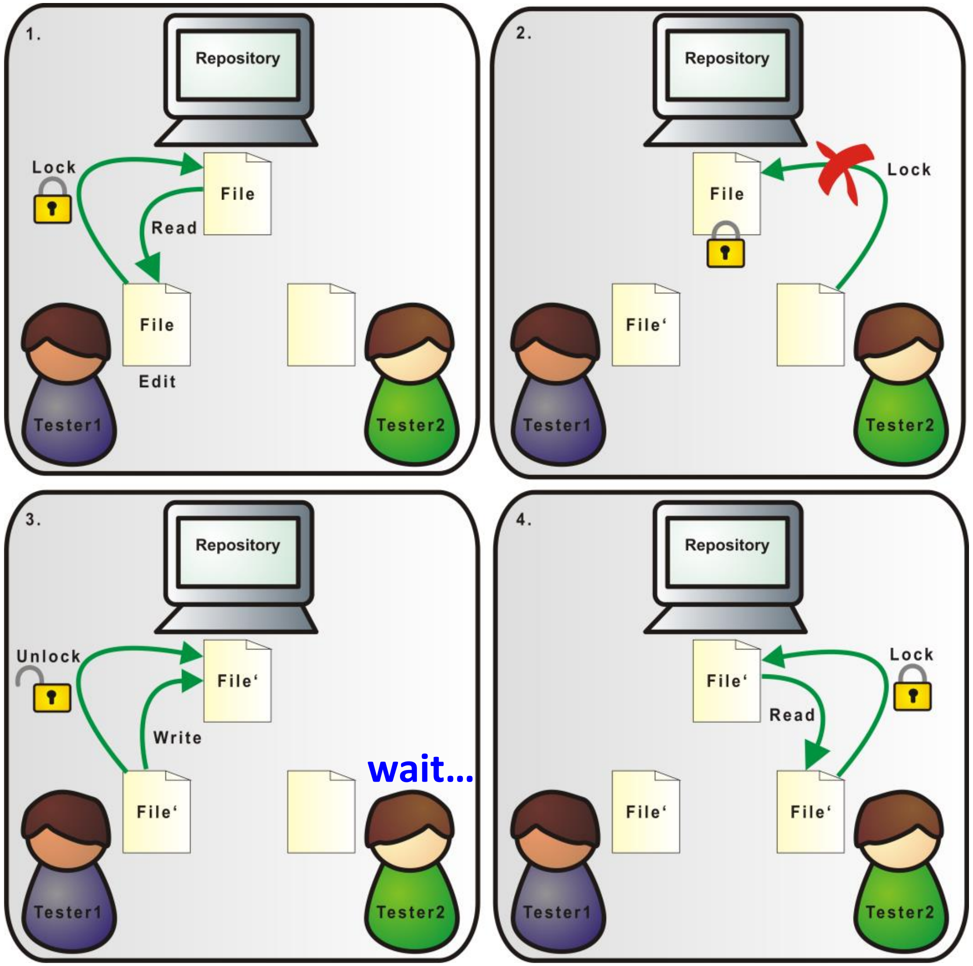
Overview

- Concurrency Protocols
- Involves two parts:
  - Locking Protocol
  - Deadlock Detection and Handling

Lock protocol – an Idea

- Ideas: transactions can proceed only after the necessary **locks are** obtained.
- If a *lock cannot be granted*, the requesting transaction must **wait** until other transactions release locks.

Some team work scenarios

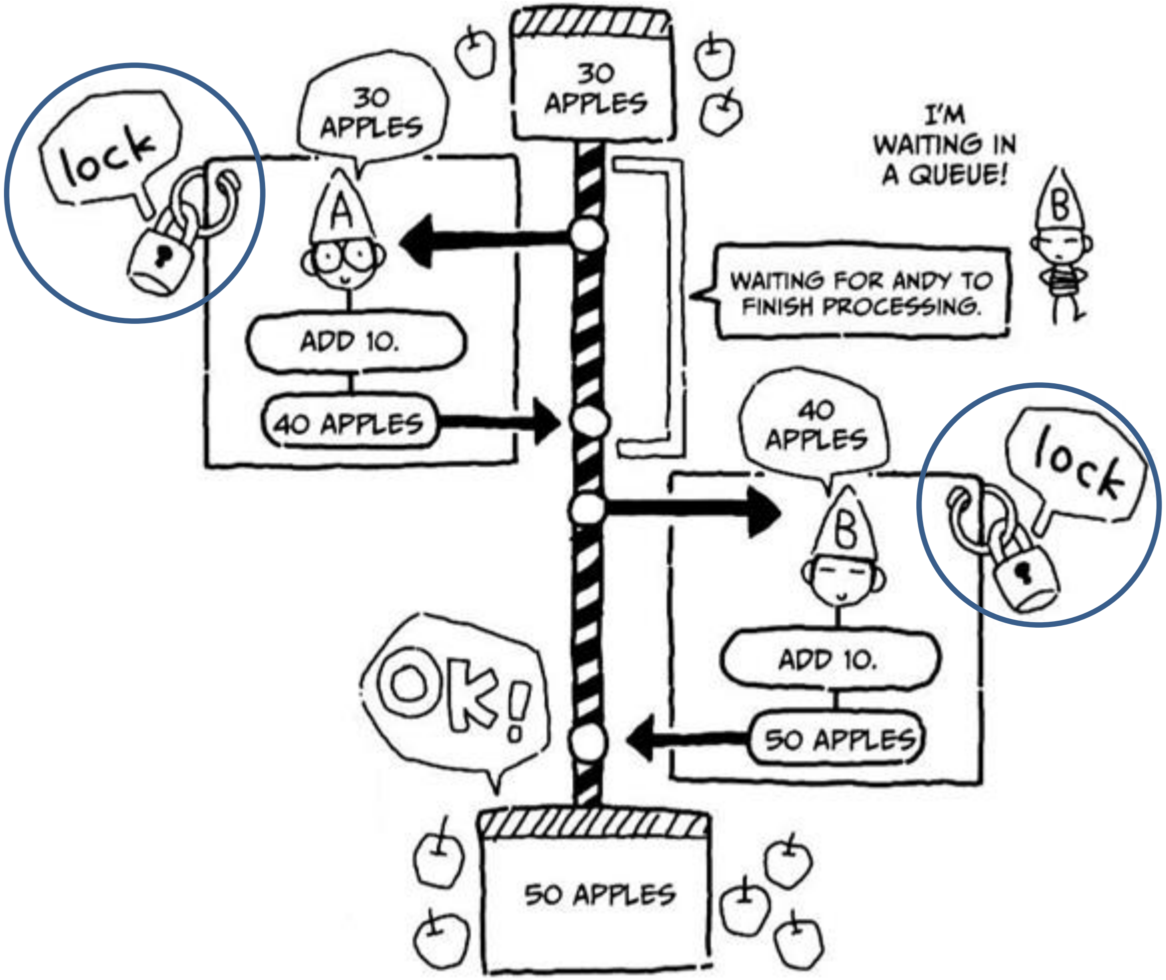


SVN: simple lock-modify-unlock
SVN is more advanced than this.

Apache Subversion (SVN)

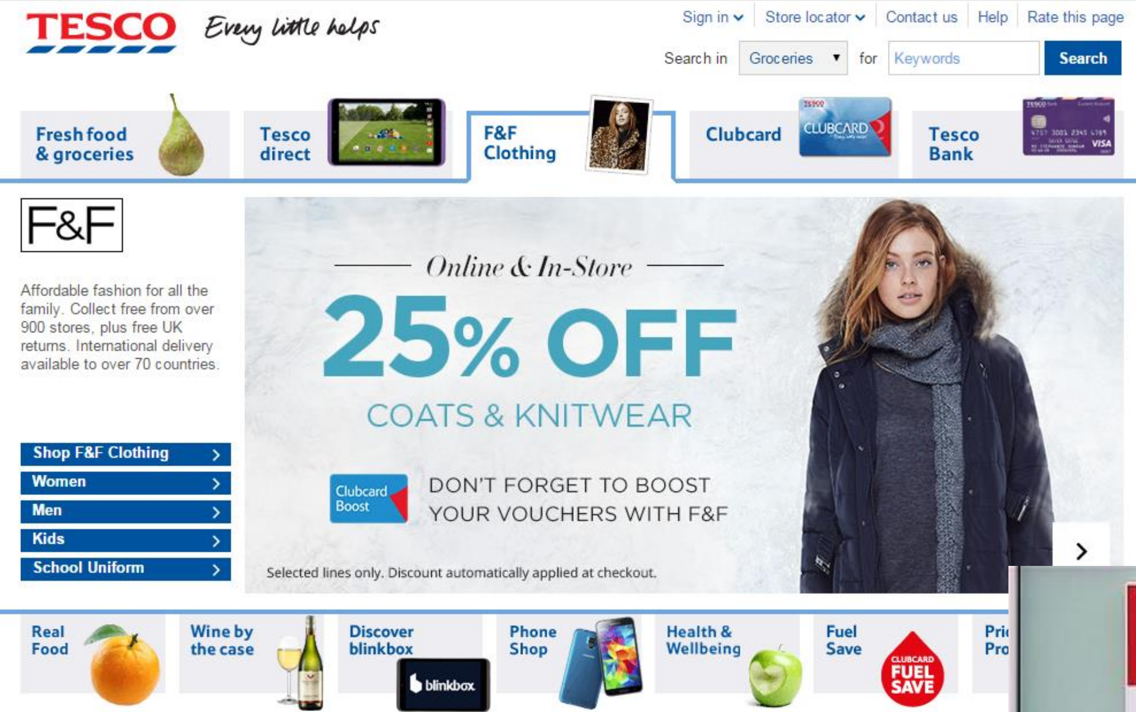
Lock protocols - inefficiency

- Locking the whole process leads to a serial schedule

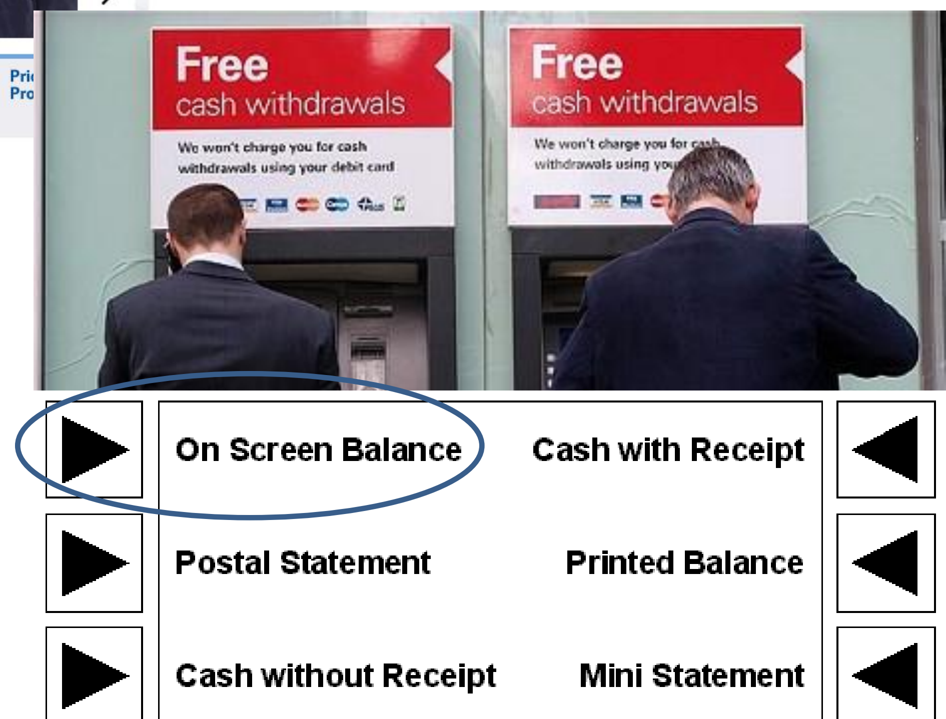


- No concurrency....

Simple Browse / Check Balance



- Should I lock all the resources (table/database)?



Locking Protocol - Better idea

- **Two kinds of Locks:**
- Shared (S) lock on an item enables a transaction to read the item only. *Allow others to read.*
- Exclusive (X) lock on a data item enables a transaction to read and write to the item. Exclude others from read/write.
- A transaction obtains a lock only if the lock is compatible with those already on the data item.

Summary:

Shared (S) lock – Read Only

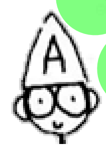
Exclusive (X) lock – Read and Write

Allow other transactions to obtain the lock?

Read & Write

Read Only

Lock already acquired by one transaction



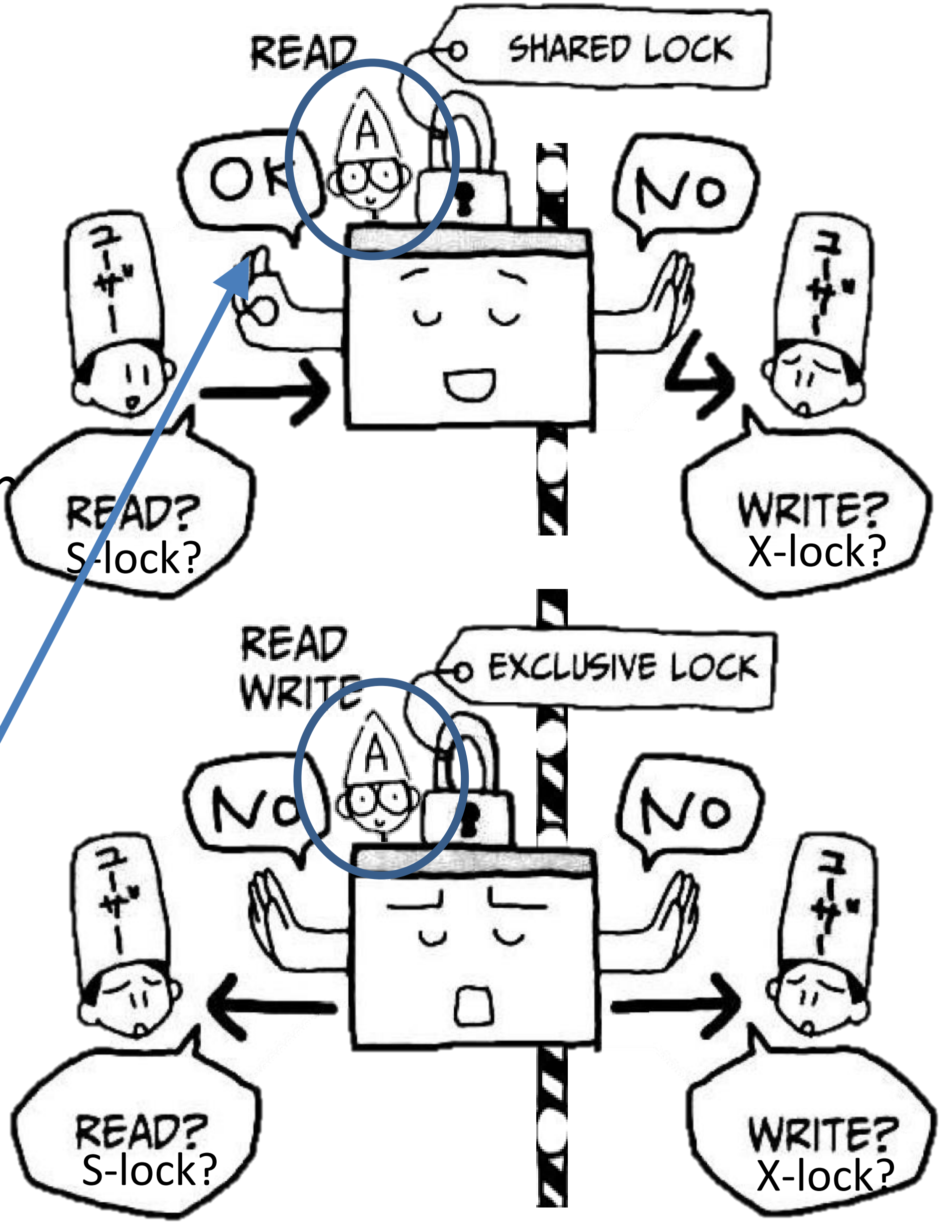
	X	S
Read & Write X	False	False
Read Only S	False	**True**

Lock compatibility on the **same resource**

Locking Protocol - Better idea

- **Two kinds of Locks:**
- Shared (S) lock on an item enables a transaction to read the item only. *Allow others to read.*
- Exclusive (X) lock on a data item enables a transaction to read and write to the item. Exclude others from read/write.
- A transaction obtains a lock only if the lock is compatible with those already on the data item.

	X	S
X	False	False
S	False	**True**



More Locking Problems

- ***Simple locking*** won't allow us to serialise all schedules.
- For example:

A:1000, B:2000

T1	T2
Read(A)	
A = A - 500	
Write(A)	
Read(B)	
B = B + 500	
Write(B)	
	Read(A)
	Read(B)
	A = A * 1.1
	B = B * 1.1
	Write(A)
	Write(B)

A:550, **B:2750** (expected)

Serial schedule

A:1000, B:2000

	T1	T2	
Lock-X(A)	Read(A)		
	A = A - 500		
	Write(A)		
Unlock (A)			Lock-X(A) Lock-X(B)
		Read(A)	
		Read(B)	
		A = A * 1.1	
		B = B * 1.1	
		Write(A)	
		Write(B)	
Lock-X(B)			Unlock (A) Unlock (B)
	Read(B)		
	B = B + 500		
	Write(B)		
Unlock (B)			

A:550, **B:2700** ✗

Two-Phase Locking

- A transaction follows two-phase locking protocol (2PL) if **all locking operations precede all unlocking operations**
  - Once a transaction releases a lock, it <u>cannot</u> apply for <u>any</u> lock in the future

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

Serialisability Theorem

Any schedule of **two-phase locking** transactions is **conflict serialisable**

Two-Phase Locking Example

- T1 follows 2PL protocol
  - All locks in T1 are acquired ***before*** any are released
  - This happens even if the resource is no longer used

T1
Lock-X(A)
Read(A)
A = A + 100
Write(A)
Lock-X(B)
Unlock(A)
Read(B)
B = B – 100
Write(B)
Unlock(B)
COMMIT

Growing

Shrinking

- T2 does not follow 2PL
  - Releases a lock on A, which is no longer needed, before acquiring on B

T2
Lock-X(A)
Read(A)
A = A + 100
Write(A)
Unlock(A)
Lock-X(B)
Read(B)
B = B – 100
Write(B)
Unlock(B)
COMMIT

Strict Two-Phase Locking (**Strict** 2PL)

- Further requires that all locks be held by a transaction until it commits/aborts.

T1	T2	T3
$\boxed{\textit{lock-X(A)}}$		
read(A)		
lock-S(B)		
read(B)		
write(A)		
	lock-X(A) wait...	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A) wait...
		read(A)
abort		

Easier to implement by DBMS

- We now discuss strict 2PL in action.

Two-Phase Locking – Muddiest Points

- When to lock?
  - Tuple **retrieve** operation: Lock-S
  - Tuple **update** operation (i.e. insert, delete, update): Lock-X
- When to unlock?
  - 2PL locking: can unlock locks (that no longer in use) ***after*** all essential locks are acquired.
  - **Strict** 2PL Locking: unlock only at the ***end*** of transaction.
- Lock promotion in growing phase:
  - If a transaction T
    - holds a Lock-S on the tuple A, and
    - wishes to update tuple A
  - T must **promote/upgrade** Lock-S to Lock-X (according to lock compatibility).

Strict 2PL - Lost Update

- Lost Update

Lock	X	S
X	False	False
S	False	True

T1	T2
Read(X)	
X = X + 10	
	Read(X)
	X = X + 10
Write(X)	
	Write(X)
	COMMIT
COMMIT	

	T1	T2	
Lock-S(X)	Read(X)		
	X = X – 5		
		Read(X)	Lock-S(X)
		X = X + 5	
upgrade Lock-X(X)	Write(X)		
	WAIT	Write(X)	upgrade Lock-X(X)
	WAIT	WAIT	
	…	WAIT	
		…	
		…	

Shared lock on resource X from a different thread

Strict 2PL prevents Lost Update, and both not allow to continue.

That’s important: No inconsistency in database. Good!

Strict 2PL - Uncommitted Update

Lock	X	S
X	False	False
S	False	True

- Uncommitted Update

T1	T2
Read(X)	
X = X + 10	
Write(X)	
	Read(X)
	X = X + 10
	Write(X)
ROLLBACK	
	COMMIT

	T1	T2	
Lock-S(X)	Read(X)		
	X = X + 10		
upgrade Lock-X(X)	Write(X)		
Unlock (X) T2 continues	ROLLBACK	WAIT	Lock-S(X)
		Read(X)	
		X = X + 10	
		Write(X)	upgrade Lock-X(X)
		COMMIT	Unlock(X)

Uncommitted Update – problem solved.

Strict 2PL - Inconsistent Analysis

Lock	X	S
X	False	False
S	False	True

- Inconsistent Analysis

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

	T1	T2	
Lock-S(X)	Read(X)		
		Read(Y)	Lock-S(Y)
		Y=Y+5	
		Write(Y)	upgrade lock-X(Y)
		Read(X)	Lock-S(X)
		X = X - 5	
		Write(X)	upgrade lock-X(X)
Lock-S(Y)	Read(Y)	WAIT	
	WAIT	WAIT	
	WAIT	WAIT	
	WAIT	WAIT	
	WAIT	WAIT	
	WAIT	WAIT	

Strict 2PL prevent Inconsistent Analysis , and not allow to continue.
That’s important: No inconsistency in database. Good!

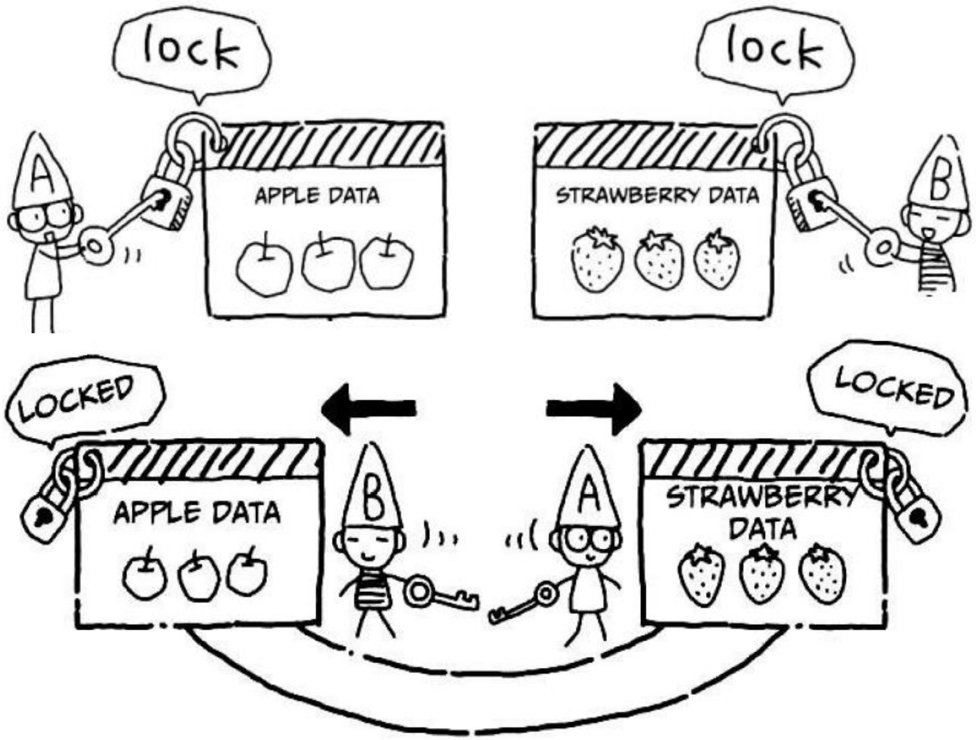
This lecture

Overview

- Concurrency Protocols
- Involves two parts:
  - Locking Protocol
  - Deadlock Detection and Handling

Deadlock

- 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.



Andy and Becky both waits for each other to release their locks, and do nothing...

	T1	T2	
Lock-S(X) →	Read(X)		
Lock-X(X) →	Write(X)		
		Read(Y)	← Lock-S(Y)
		Write(Y)	← Lock-X(Y)
Lock-X(Y) →	Read(Y)		
	WAIT	Read(X)	← Lock-X(X)
	WAIT	WAIT	
	WAIT	WAIT	
	WAIT	WAIT	
	WAIT	WAIT	
	WAIT	WAIT	
	...	WAIT	
		...	

Deadlock Detection

- Is there a deadlock?

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

	X	S
X	False	False
S	False	**True**

Deadlock Detection

- Is there a deadlock? First write down the locks

	T1	T2	T3	
Lock-S(X)	Read(X)			
		Read(Y)		Lock-S(Y)
Lock-X(X) (upgrade)	Write(X)			
		Read(X)		Lock-S(X)
			Read(Z)	Lock-S(Z)
			Write(Z)	Lock-X(Z)
Lock-S(Y)	Read(Y)			
			Read(X)	Lock-S(X)
Lock-X(Y) (upgrade)	Write(Y)			

	X	S
X	False	False
S	False	**True**

Deadlock Detection

- Is there a deadlock? Then, check locks above.

Lock	T1	T2	T3	Lock
Lock-S(X)	Read(X)			
		Read(Y)		Lock-S(Y)
Lock-X(X) (upgrade)	Write(X)			
		Read(X)		Lock-S(X)
		WAIT	Read(Z)	Lock-S(Z)
		...	Write(Z)	Lock-X(Z)
Lock-S(Y)	Read(Y)	...		
		...	Read(X)	Lock-S(X)
Lock-X(Y) (upgrade)	Write(Y)	...		
		...		
		...		

	X	S
X	False	False
S	False	**True**

Deadlock Detection

- Is there a deadlock?

	T1	T2	T3	
Lock-S(X)	Read(X)			
		Read(Y)		Lock-S(Y)
Lock-X(X) (upgrade)	Write(X)			
		Read(X)		Lock-S(X)
		WAIT	Read(Z)	Lock-S(Z)
			Write(Z)	Lock-X(Z)
		...		
Lock-S(Y)	Read(Y)	**...**		
		...	Read(X)	Lock-S(X)
Lock-X(Y) (upgrade)	Write(Y)	**...**	**WAIT**	
		...	**...**	
		...	**...**	

	X	S
X	False	False
S	False	**True**

Deadlock Detection

- Is there a deadlock?

	T1	T2	T3	
Lock-S(X)	Read(X)			
		Read(Y)		Lock-S(Y)
Lock-X(X) (upgrade)	Write(X)			
		Read(X)		Lock-S(X)
		WAIT	Read(Z)	Lock-S(Z)
		...	Write(Z)	Lock-X(Z)
Lock-S(Y)	Read(Y)	...		
		...	Read(X)	Lock-S(X)
		...	**WAIT**	
Lock-X(Y) (upgrade)	Write(Y)	...		
	WAIT	...	...	
	...	...	...	

	X	S
X	False	False
S	False	**True**

Yes, deadlock! T1 and T2 wait one another to release locks.

Wait-for graph

- Computer needs an algorithm

- Given a schedule, we can ***detect*** deadlocks which will happen in this schedule using a wait-for graph (WFG).

Locks operations:
- Look above
- Look at another transaction

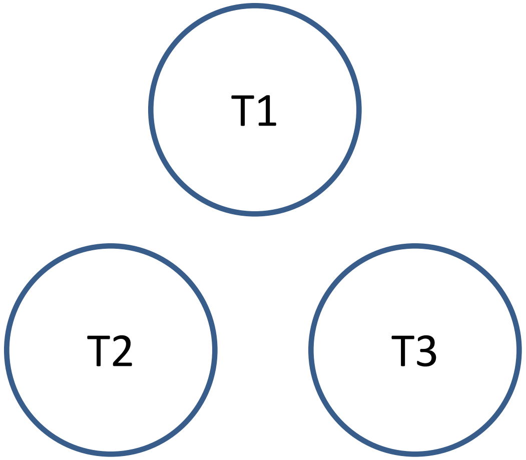
- Each transaction is a vertex
- Edge from T2 to T1 if
  - T1 Lock-S(X) then T2 tries to Lock-X(X) it
  - T1 Lock-X(X) then T2 tries to Lock-S(X) it
  - T1 Lock-X(X) then T2 tries to Lock-X(X) it

Lock	X	S
X	False	False
S	False	True

Example – Wait For Graph

- Deadlock?

	T1	T2	T3	
Lock-S(X)	Read(X)			
		Read(Y)		Lock-S(Y)
Lock-X(X) (upgrade)	Write(X)			
		Read(X)		Lock-S(X)
			Read(Z)	Lock-S(Z)
			Write(Z)	Lock-X(Z)
Lock-S(Y)	Read(Y)			
			Read(X)	Lock-S(X)
Lock-X(Y) (upgrade)	Write(Y)			

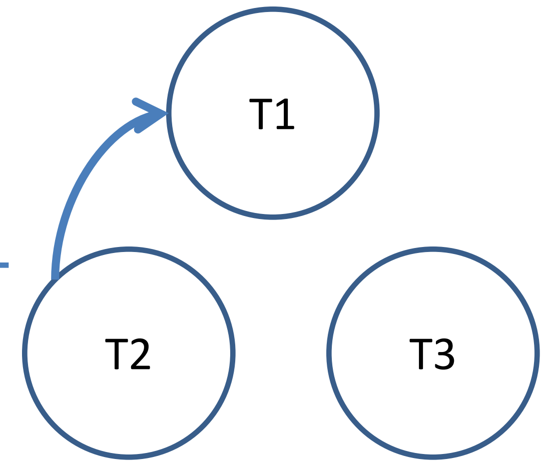


Wait For Graph

Example – Wait For Graph

- Deadlock?

	T1	T2	T3	
Lock-S(X)	Read(X)			
		Read(Y)		Lock-S(Y)
Lock-X(X) (upgrade)	Write(X)			
		Read(X)		**Lock-S(X)**
			Read(Z)	Lock-S(Z)
			Write(Z)	Lock-X(Z)
Lock-S(Y)	Read(Y)			
			Read(X)	Lock-S(X)
Lock-X(Y) (upgrade)	Write(Y)			

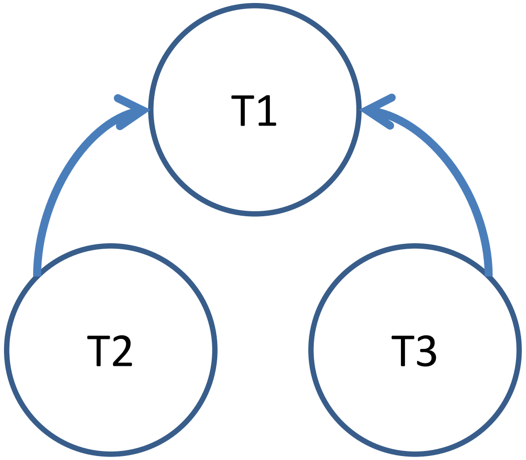


Wait For Graph

Example – Wait For Graph

- Deadlock?

	T1	T2	T3	
Lock-S(X)	Read(X)			
		Read(Y)		Lock-S(Y)
Lock-X(X) (upgrade)	Write(X)			
		Read(X)		Lock-S(X)
			Read(Z)	Lock-S(Z)
			Write(Z)	Lock-X(Z)
Lock-S(Y)	Read(Y)			
			Read(X)	**Lock-S(X)**
Lock-X(Y) (upgrade)	Write(Y)			

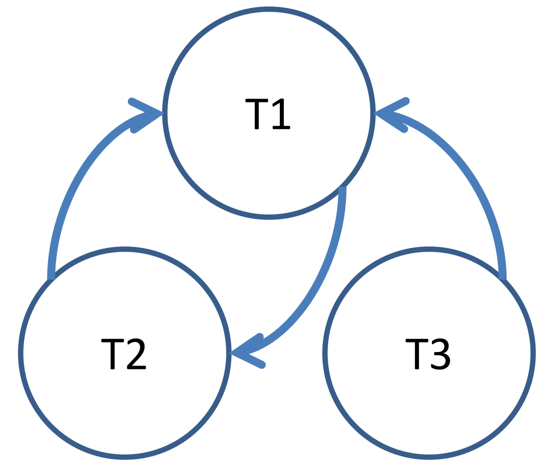


Wait For Graph

Example – Wait For Graph

- Deadlock?

	T1	T2	T3	
Lock-S(X)	Read(X)			
		Read(Y)		**Lock-S(Y)**
Lock-X(X) (upgrade)	Write(X)			
		Read(X) WAIT		Lock-S(X)
		...	Read(Z)	Lock-S(Z)
		...	Write(Z)	Lock-X(Z)
Lock-S(Y)	Read(Y)	...		
		...	Read(X)	Lock-S(X)
Lock-X(Y) (upgrade)	Write(Y) WAIT	...		
	...	...		



Wait For Graph

Run DFS, Cycle detected : Deadlock!

Deadlock Recovery

- Deadlock is less of a problem than an inconsistent DB
- Most DBMSs (include Oracle, MySQL)
  - Uses Strict 2PL for locking
  - Detect deadlocks with a wait-for graph
  - Recovery : choose a single transaction as a 'victim' to rollback and restart
  - Which transaction?
    - has been running the longest/shortest?
    - have made the most/least updates?
    - have the most/least updates still to make?