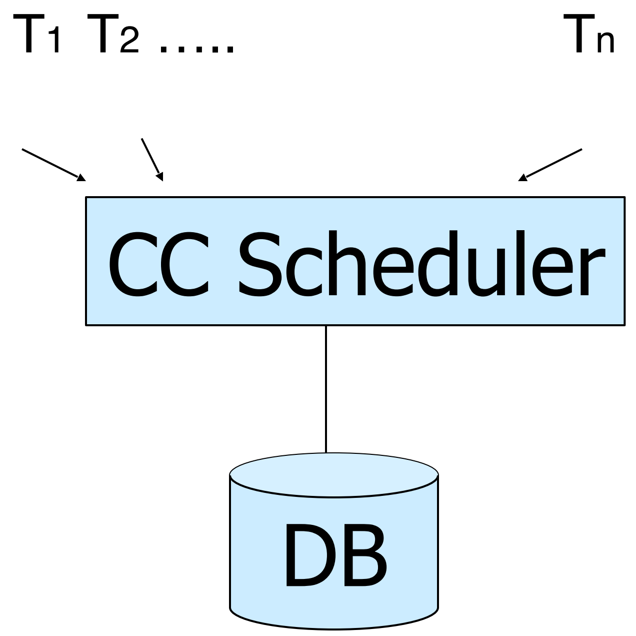
Concurrency Control

- Concurrency Control
  - Ensures interleaving of operations amongst concurrent transactions result in serializable schedules

- How?
  - transaction operations interleaved following a protocol

How to enforce serializable schedules?

Prevent P(S) cycles from occurring using a **concurrency control manager:** ensures interleaving of operations amongst concurrent transactions only result in serializable schedules.



Concurrency Via Locks

- Idea:
  - Data items modified by one transaction at a time
- Locks
  - Control access to a resource
    - Can block a transaction until lock granted
  - Two modes:
    - **S**hared (read only)
    - e**X**clusive (read & write)

Granting Locks

- Requesting locks
  - Must request before accessing a data item

- Granting Locks
  - No lock on data item?   Grant
  - Existing lock on data item?
    - Check compatibility:
      - Compatible? Grant
      - Not? Block transaction

	shared	exclusive
shared	Yes	No
exclusive	No	No

Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example:

T1	T2
lock-X(B)	lock-S(A)
read(B)	read(A)
B ←B-50	unlock(A)
write(B)	lock-S(B)
unlock(B)	read(B)
lock-X(A)	unlock(B)
read(A)	display(A+B)
A ←A + 50	
write(A)	
unlock(A)	

Locking Issues

- Starvation
  - T1 holds shared lock on Q
  - T2 requests exclusive lock on Q: blocks
  - T3, T4, ..., Tn request shared locks: granted
  - T2 is starved!

- Solution?

Do not grant locks if older transaction is waiting

Locking Issues

- No transaction proceeds:

Deadlock

- T1 waits for T2 to unlock A
- T2 waits for T1 to unlock B

Rollback transactions
Can be costly...

T1	T2
lock-X(B)	
read(B)	
B ← B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Locking Issues

- Locks do not ensure serializability by themselves:

T1	T2
lock-X(B)	
read(B)	
$B \leftarrow B-50$	
write(B)	
unlock(B)	
	lock-S(A)
	read(A)
	unlock(A)
	lock-S(B)
	read(B)
	unlock(B)
	display(A+B)
lock-X(A)	
read(A)	
$A \leftarrow A + 50$	
write(A)	
unlock(A)	

T2 displays 50 less!!

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock). Locks can be either X, or S/X.

2PL

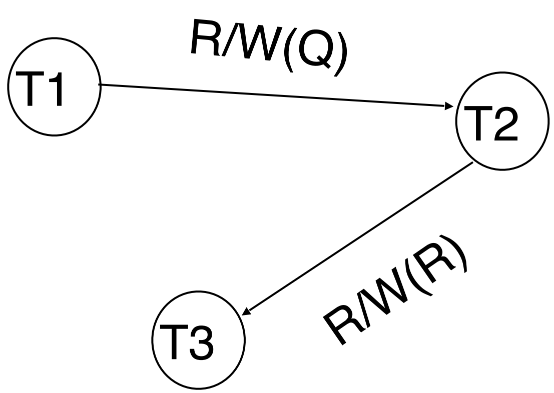
- Example: T1 in 2PL

	T1
Growing phase	lock-X(B)
	read(B)
	B ← B - 50
	write(B)
	lock-X(A)
	read(A)
	A ← A - 50
	write(A)
Shrinking phase	unlock(B)
	unlock(A)

2PL & Serializability

- Recall: Precedence Graph

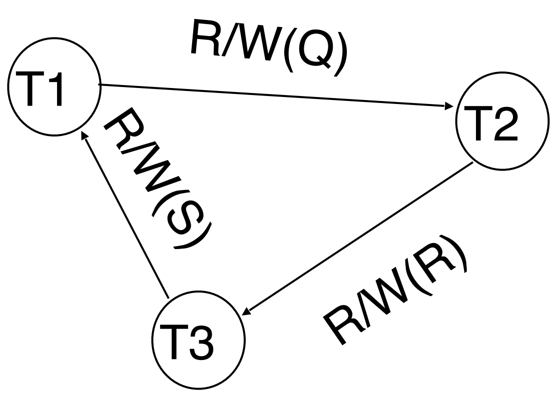
T1	T2	T3
read(Q)		
	write(Q)	
	read(R)	
		write(R)
		read(S)



2PL & Serializability

- Recall: Precedence Graph

T1	T2	T3
read(Q)		
	write(Q)	
	read(R)	
		write(R)
		read(S)
write(S)		



Cycle → Non-serializable

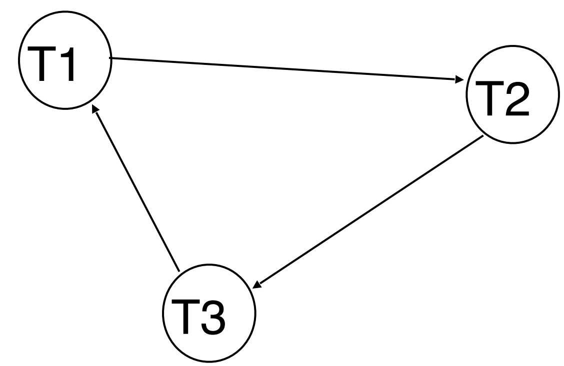
2PL & Serializability

Relation between Growing & Shrinking phase:

$T_1G < T_1S$

$T_2G < T_2S$

$T_3G < T_3S$



T1 must release locks for other to proceed

$T_1S < T_2G$

$T_2S < T_3G$

$\boxed{T_3S < T_1G}$

$T_1G < T_1S < T_2G < T_2S < T_3G < T_3S < T_1G$

Not Possible under 2PL!

It can be generalized for any set of transactions...

2PL Issues

- As observed earlier,
  2PL does not prevent deadlock

- > 2 transactions involved?
  - Rollbacks expensive.

- We will revisit later.

T1	T2
lock-X(B)	
read(B)	
B ← B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

2PL Variants

Strict two phase locking

- Exclusive locks must be held until transaction commits
- **Ensures data written by transaction can’t be read by others**
- **Prevents cascading rollbacks**

Strict 2PL

T1	T2	T3
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)
<xaction fails>		

Strict 2PL will not allow that

Strict 2PL & Cascading Rollbacks

- Ensures any data written by uncommited transaction not read by another

- Strict 2PL would prevent T2 and T3 from reading A
  - T2 & T3 wouldn't rollback if T1 does

Deadlock Handling

- Consider the following two transactions:

  T_1: write (X) $\quad$ T_2: write(Y)

  write(Y) $\quad$ write(X)

- Schedule with deadlock

T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

Deadlock Handling

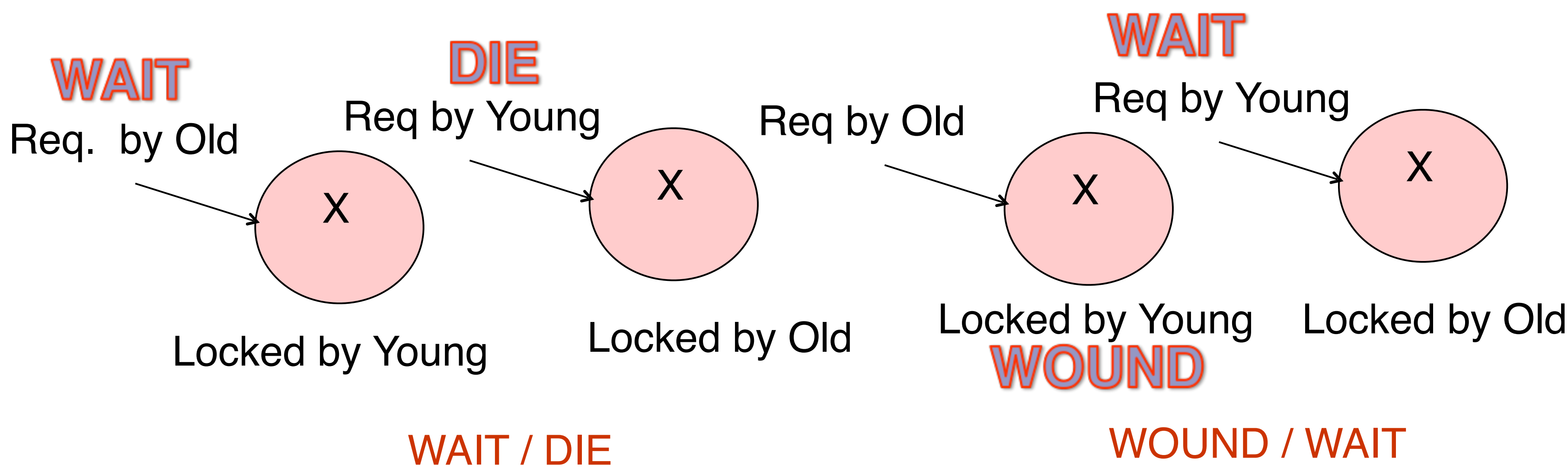
- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction 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 (graph-based protocol).

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. 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.

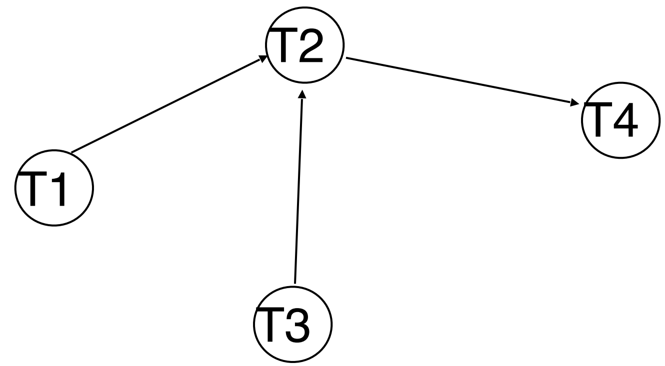
Deadlock Prevention

	Wait / Die	Wound / Wait
O Needs a resource held by Y	O Waits	Y Dies
Y needs a resource held by O	Y Dies	Y Waits



Dealing with Deadlocks

- How do you detect a deadlock?
  - Wait-for graph
  - Directed edge from Ti to Tj
    - If Ti waiting for Tj



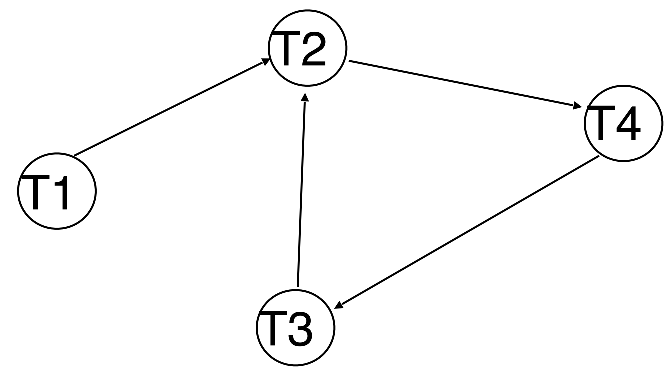
T1	T2	T3	T4
		X(Z)	
	X(V)		
			X(W)
S(V)			
	S(W)		
		S(V)	

Suppose T4 requests lock-S(Z)....

Detecting Deadlocks

- Wait-for graph has a cycle → deadlock

T2, T3, T4 are deadlocked



•Build wait-for graph, check for cycle

•How often?
- Tunable
IF expect many deadlocks or many transactions involved
run often to reduce aborts
ELSE run less often to reduce overhead

Recovering from Deadlocks

- Rollback one or more transaction
  - Which one?
    - Rollback the cheapest ones
    - Cheapest ill-defined
      - Was it almost done?
      - How much will it have to redo?
      - Will it cause other rollbacks?
    - How far?
      - May only need a partial rollback
    - Avoid starvation
      - Ensure same xction not always chosen to break deadlock

Timestamp-Based Protocols

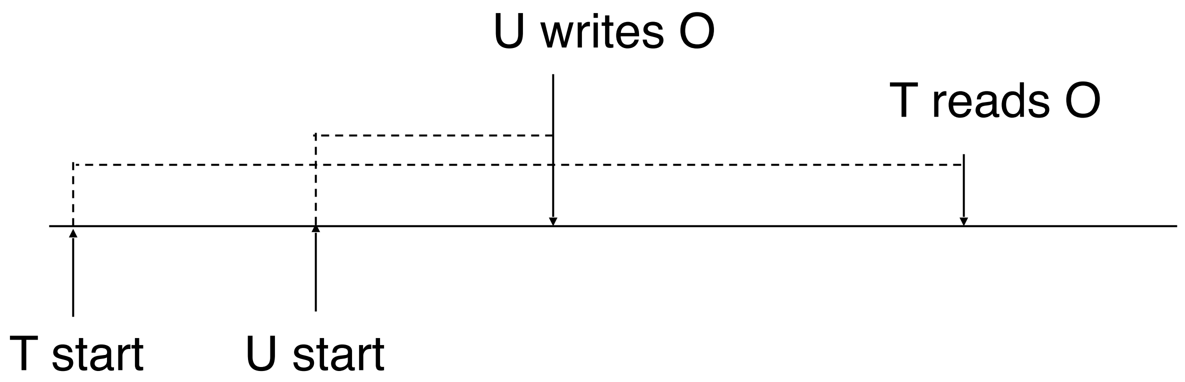
- Idea:
  - Decide in advance ordering of transactions.
  - Ensure concurrent schedule serializes to that serial order.
- Timestamps
  1. $TS(T_i)$ is time T_i entered the system
  2. Data item timestamps:
     1. W-TS(Q): Largest timestamp of any xction that wrote Q
     2. R-TS(Q): Largest timestamp of any xction that read Q
- Timestamps -> serializability order

Timestamp CC

Idea: If action p_i of Xact T_i conflicts with action q_j of Xact T_j, and $TS(T_i) < TS(T_j)$, then p_i must occur before q_j. Otherwise, restart violating Xact.

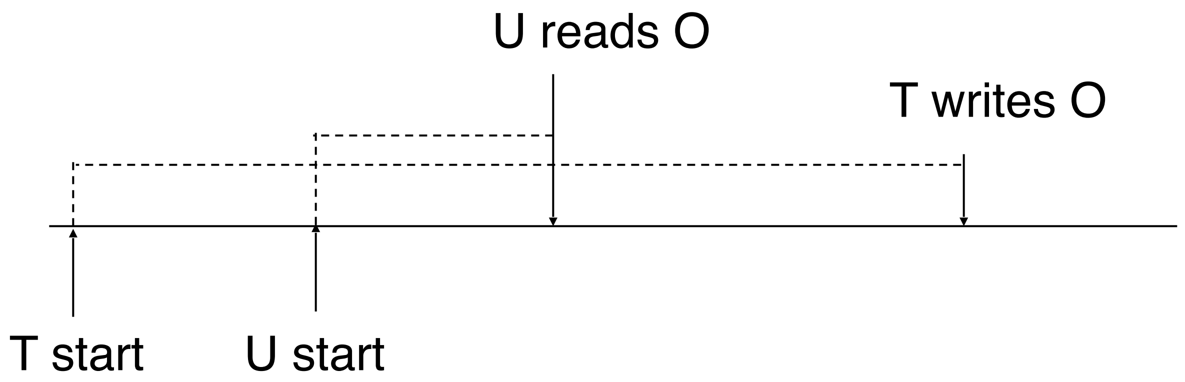
When Xact T wants to read Object O

- If $TS(T) < W\text{-}TS(O)$, this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again!)
- If $TS(T) > W\text{-}TS(O)$:
  - Allow T to read O.
  - Reset $R\text{-}TS(O)$ to $\max(R\text{-}TS(O), TS(T))$
- Change to $R\text{-}TS(O)$ on reads must be written to disk! This and restarts represent overhead.



When Xact T wants to Write Object O

- If TS(*T*) < R-TS(*O*), then the value of *O* that *T* is producing was needed previously, and the system assumed that that value would never be produced. **write** rejected, *T* is rolled back.
- If TS(*T*) < W-TS(*O*), then *T* is attempting to write an obsolete value of *O*. Hence, this **write** operation is rejected, and *T* is rolled back.
- Otherwise, the **write** operation is executed, and W-TS(*O*) is set to TS(*T*).



Timestamp-Ordering Protocol

- Rollbacks still present
  - On rollback, new timestamp & restart

T1 rollback since TS(T1) < W-TS(O)=TS(T2)

T1	T2
Read(O)	
	Write(O)
Write(O)	

Can reduce one rollback situation

When transaction writes an obsolete value, ignore it:

Thomas' write-rule does not rollback T1

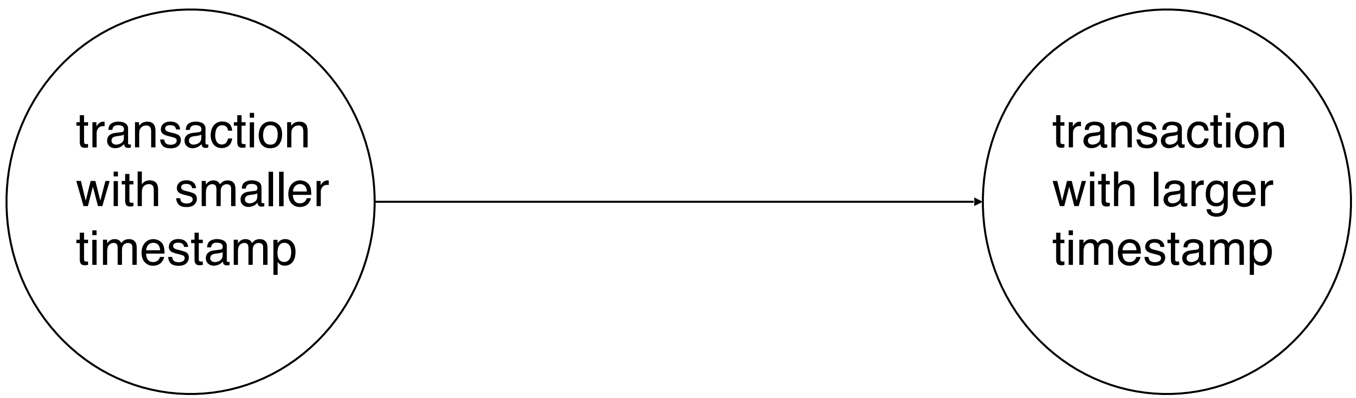
Example Use of the Protocol

A partial schedule for several data items for transactions with initial timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5	T_6	T_7
				write(X)		
	read(Y)					
read(Y)						
		write(Y) write(Z)				
				read(Z)		
	read(X) abort					
read(Q)						
					read(Y)	
		write(Z) abort				
				write(Y) write(Z)		
					read(X)	

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.