Enforcing Serializability
(implementing concurrency control)

- Objectives:
- Locking
  - two phase-locking protocol (with simple locking)
  - escalading locks (read/shared vs write/exclusive)
  - update locks
  - multigranular locks (tuple, block, table level locks)

How to enforce serializable schedules?

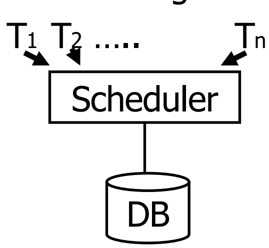
Option 1: Optimistic protocol

- run system, record P(S);
- at commit, check P(S) for cycles
  - if no cycles, commit
  - otherwise, abort offending transactions and restart.

Under what circumstances would this make sense?

How to enforce serializable schedules?

Option 2: prevent P(S) cycles from occurring



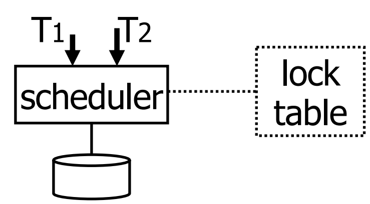
// a protocol decides whose turn

A locking protocol

Two new actions:

lock (exclusive): l_i (A)

unlock: u_i (A)



Rule #1: Well-formed transactions

T_i: … $l_i(A)$ … $p_i(A)$ … $u_i(A)$ …

Obvious, proper use of lock/unlock.

- Every read or write is preceded by a lock
- Every lock has a corresponding unlock
- Don't ask for a lock twice, not without an intervening unlock: $l_i(A)\ l_i(A)$

Rule #2 Legal scheduler

$S =$ ……. $l_i(A)$ ………… $u_i(A)$ ………

$\longleftrightarrow$

no $l_j(A)$

Blocks a transaction that asks for a lock on a locked object

Exercise:

- What schedules are legal?
  What transactions are well-formed?

  $S1 = l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$
  $r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

  $S2 = l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$
  $l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$

  $S3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
  $l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

Exercise:

- What schedules are legal?
  What transactions are well-formed?

  $S1 = l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$
  $r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

  $S2 = l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$
  $l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$

  $S3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
  $l_2(A)r_2(A)w_2(A)u_2(A)l_3(B)r_3(B)u_3(B)$

Exercise:

- What schedules are legal?
  What transactions are well-formed?

  S1 = $l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$
  $r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$ *illegal*

  S2 = $l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$
  $l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$ *not well formed*

  S3 = $l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
  $l_2(A)r_2(A)w_2(A)u_2(A)l_3(B)r_3(B)u_3(B)$

Revisit: Schedule F, add locking

T1	T2
$l_1(A)$;Read(A)	
$A \leftarrow A+100$;Write(A);$u_1(A)$	
	$l_2(A)$;Read(A)
	$A \leftarrow Ax2$;Write(A);$u_2(A)$
	$l_2(B)$;Read(B)
	$B \leftarrow Bx2$;Write(B);$u_2(B)$
$l_1(B)$;Read(B)	
$B \leftarrow B+100$;Write(B);$u_1(B)$	

Schedule F

T1	T2	A	B
		25	25
$l_1(A)$;Read(A)			
$A \leftarrow A+100$;Write(A);$u_1(A)$		125	
	$l_2(A)$;Read(A)		
	$A \leftarrow Ax2$;Write(A);$u_2(A)$	250	
	$l_2(B)$;Read(B)		
	$B \leftarrow Bx2$;Write(B);$u_2(B)$		50
$l_1(B)$;Read(B)			
$B \leftarrow B+100$;Write(B);$u_1(B)$			150
		250	150

still oops

Two phase locking (2PL)

for transactions

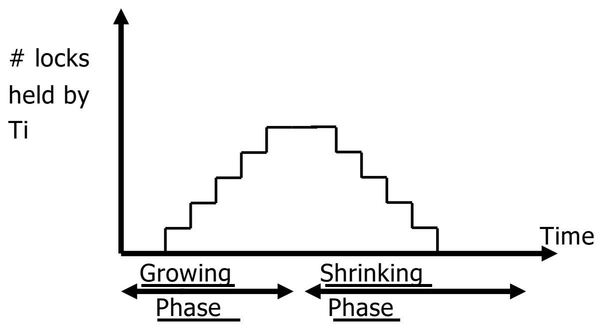
T_i = ……. $l_i(A)$ ………… $u_i(A)$ ………

no unlocks no locks

- no locks awarded/asked for

after the first unlock

--> collect locks, then release locks



Schedule G

T1	T2
l_1(A);Read(A)	
A ← A+100;Write(A)	
l_1(B); u_1(A)	
	l_2(A);Read(A) delayed
	A ← Ax2;Write(A);l_2(B)

Schedule G

T1	T2
l_1(A);Read(A)	
A ← A+100;Write(A)	
l_1(B); u_1(A)	
	l_2(A);Read(A) delayed
	A ← Ax2;Write(A);l_2(B)
Read(B);B ← B+100	
Write(B); u_1(B)	

Schedule G

T1	T2
l_1(A);Read(A)	
A ← A+100;Write(A)	
l_1(B); u_1(A)	
	l_2(A);Read(A) delayed
	A ← Ax2;Write(A);l_2(B)
Read(B);B ← B+100	
Write(B); u_1(B)	
	l_2(B); u_2(A);Read(B)
	B ← Bx2;Write(B);u_2(B);

Schedule H (T$_2$ reversed)

T1	T2
$l_1(A)$; Read(A)	$l_2(B)$;Read(B)
A ← A+100;Write(A)	B ← Bx2;Write(B)
$l_1(B)$ delayed	$l_2(A)$ delayed

- Assume deadlocked transactions are rolled back
  - They have no effect
  - They do not appear in schedule

E.g., Schedule H = $\underbrace{\qquad\qquad\qquad}_{\text{This space intentionally left blank!}}$

Next step:

Show that rules #1,2,3 ⇒ conflict-serializable schedules

<u>Conflict rules for</u> $l_i(A)$, $u_i(A)$:

- $l_i(A)$, $l_j(A)$ conflict
- $l_i(A)$, $u_j(A)$ conflict

Note: no conflict < $u_i(A)$, $u_j(A)$>, < $l_i(A)$, $r_j(A)$>,...

Theorem Rules #1,2,3 $\Rightarrow$ conflict
(2PL) serializable
schedule

To help in proof:

Definition Shrink(Ti) = SH(Ti) =
first unlock action of Ti

Lemma

$Ti \rightarrow Tj$ in S $\Rightarrow$ SH(Ti) $<_S$ SH(Tj)

Proof of lemma:

$Ti \rightarrow Tj$ means that

S = … $p_i(A)$ … $q_j(A)$ …; p,q conflict

By rules 1,2:

S = … $p_i(A)$ … $u_i(A)$ … $l_j(A)$ **…** $q_j(A)$ …

By rule 3: SH(Ti) SH(Tj)

So, SH(Ti) $<_S$ SH(Tj)

Theorem Rules #1,2,3 $\Rightarrow$ conflict
(2PL) serializable
schedule

Proof:

(1) Assume P(S) has cycle

$T_1 \rightarrow T_2 \rightarrow \ldots T_n \rightarrow T_1$

(2) By lemma: $SH(T_1) < SH(T_2) < \ldots < SH(T_1)$

(3) Impossible, so P(S) acyclic

(4) $\Rightarrow$ S is conflict serializable

- Beyond this simple 2PL protocol, it is all a matter of improving performance and allowing more concurrency….
  - Shared locks
  - Multiple granularity
  - Inserts, deletes and phantoms
  - Other types of C.C. mechanisms

Shared locks

So far:

$S = \ldots l_1(A)\ r_1(A)\ u_1(A) \ldots l_2(A)\ r_2(A)\ u_2(A) \ldots$

Do not conflict

Instead:

$S = \ldots ls_1(A)\ r_1(A)\ ls_2(A)\ r_2(A) \ldots\ us_1(A)\ us_2(A)$

Lock actions

$l\text{-}t_i(A)$: lock A in t mode (t is S or X)

$u\text{-}t_i(A)$: unlock t mode (t is S or X)

Shorthand:

$u_i(A)$: unlock whatever modes T_i has locked A

Rule #1_ Well formed transactions

$T_i = \ldots l\text{-}S_1(A) \ldots r_1(A) \ldots u_1(A) \ldots$

$T_i = \ldots l\text{-}X_1(A) \ldots w_1(A) \ldots u_1(A) \ldots$

Locks on same object are now allowed…

but - really :

escalating locks on same object allowed

- What about transactions that read and write same object?

Option 1: Request exclusive lock

$T_i = \ldots l\text{-}X_1(A) \ldots r_1(A) \ldots w_1(A) \ldots u(A) \ldots$

- What about transactions that read and write same object?

Option 2: Upgrade

(E.g., need to read, but don't know if will write…)

T_i=... l-S_1(A) … r_1(A) ... l-X_1(A) …w_1(A) ...u(A)…

Think of
- Get 2nd lock on A, or
- Drop S, get X lock

Rule #2 Legal scheduler

S = ...l-S_i(A) … … u_i(A) …

$\longleftrightarrow$

no l-X_j(A)

S = ... l-X_i(A) … … u_i(A) …

$\longleftrightarrow$

no l-X_j(A)
no l-S_j(A)

A way to summarize Rule #2

Compatibility matrix

		request	
		S	X
held	S	true	false
	X	false	false

Rule # 3 2PL transactions

No change except for upgrades:

(I) If upgrade gets more locks
(e.g., S $\rightarrow$ {S, X}) then no change!

(II) If upgrade releases read (shared) lock (e.g., S $\rightarrow$ X)
- beginning of shrinking

<u>Theorem</u> Rules 1,2,3 $\Rightarrow$ Conf.serializable for S/X locks schedules

<u>Proof:</u> similar to X locks case

<u>Detail:</u>

$l\text{-}t_i(A)$, $l\text{-}r_j(A)$ do not conflict if comp(t,r)

$l\text{-}t_i(A)$, $u\text{-}r_j(A)$ do not conflict if comp(t,r)

Lock types beyond S/X

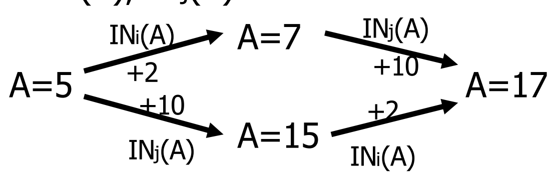
Examples:

(1) increment lock

(2) update lock

Example (1): increment lock

- Atomic increment action: $IN_i(A)$
  {Read(A); A $\leftarrow$ A+k; Write(A)}
- $IN_i(A)$, $IN_j(A)$ do not conflict!



Comp

	S	X	I
S			
X			
I			

Comp

	S	X	I
S	T	F	F
X	F	F	F
I	F	F	T

Update locks

A common deadlock problem with upgrades:

T1	T2
$l\text{-}S_1(A)$	
	$l\text{-}S_2(A)$
$l\text{-}X_1(A)$	
	$l\text{-}X_2(A)$

--- Deadlock ---

Solution

If T_i wants to read A and knows it may later want to write A, it requests update lock (not shared)

Comp

New request

Lock already held in	S	X	U
S			
X			
U			

New request

Comp

Lock already held in

	S	X	U
S	T	F	T
X	F	F	F
U	F	F	F

Note: object A may be locked in different modes at the same time...

S_1=...l-S_1(A)...l-S_2(A)...l-U_3(A)... { l-S_4(A)...? / l-U_4(A)...?

- To grant a lock in mode t, mode t must be compatible with all currently held locks on object

How does locking work in practice?

- Every system is different

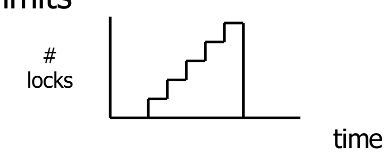
  (E.g., may not even provide CONFLICT-SERIALIZABLE schedules)

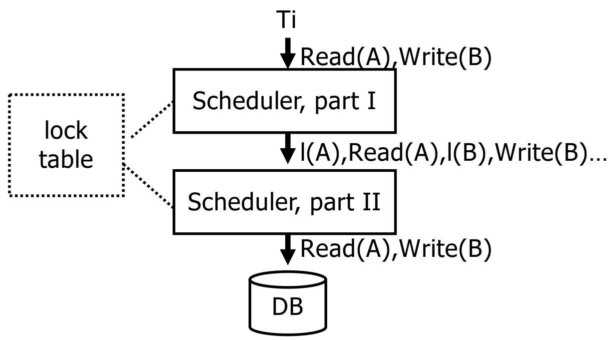
- But here is one (simplified) way ...

Sample Locking System:

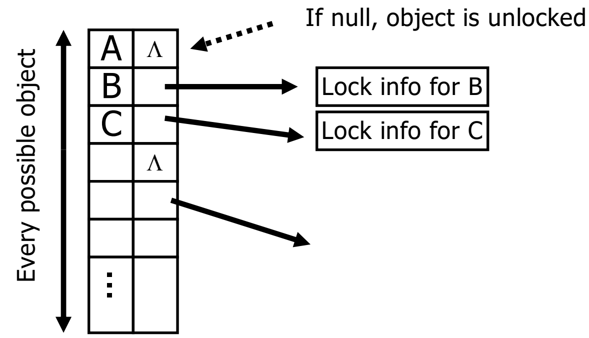
(1) Don't trust transactions to request/release locks

(2) Hold all locks until transaction commits

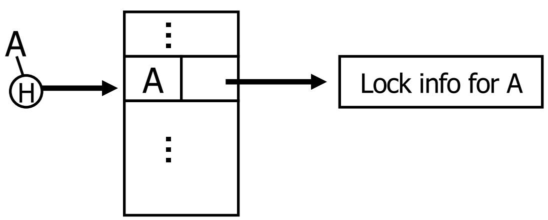




Lock table Conceptually

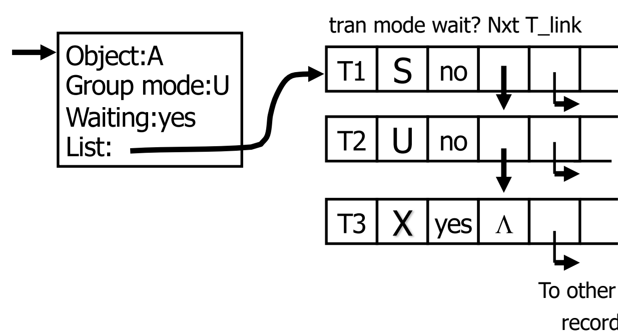


But use hash table:

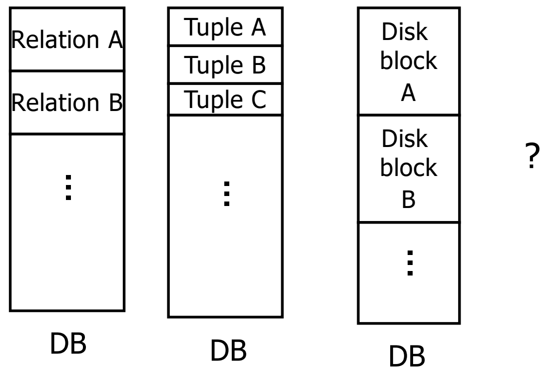


If object not found in hash table, it is unlocked

Lock info for A - example



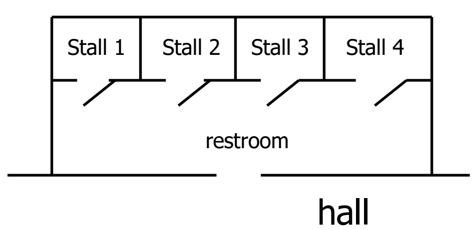
What are the objects we lock?



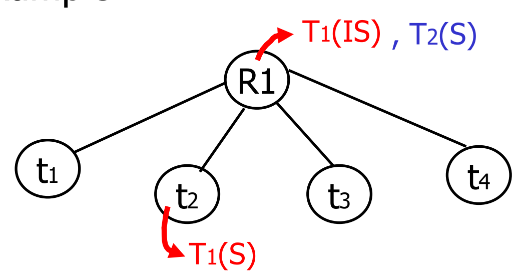
- Locking works in any case, but should we choose small or large objects?
- If we lock large objects (e.g., Relations)
  - Need few locks
  - Low concurrency
- If we lock small objects (e.g., tuples,fields)
  - Need more locks
  - More concurrency

We can have it both ways!!

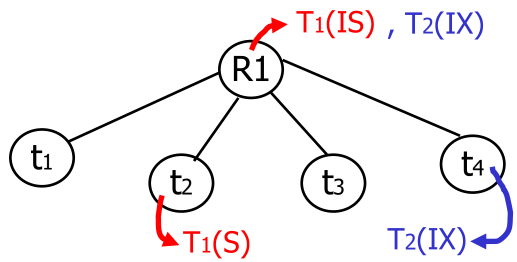
Ask any janitor to give you the solution...



Example



Example



Multiple granularity

Comp

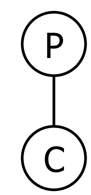
Holder \ Requestor	IS	IX	S	SIX	X
IS					
IX					
S					
SIX					
X					

Multiple granularity

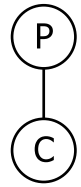
Comp

Holder \ Requestor	IS	IX	S	X
IS	T	T	T	F
IX	T	T	F	F
S	T	F	T	F
X	F	F	F	F

Parent locked in	Child can be locked in
IS	
IX	
S	
SIX	
X	



Parent locked in	Child can be locked in
IS	IS, S
IX	IS, S, IX, X
S	[S, IS] not necessary
X	none

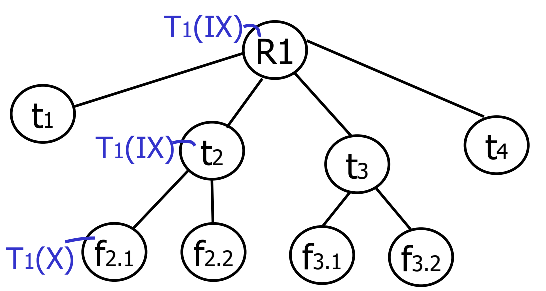


Rules

(1) Follow multiple granularity comp function
(2) Lock root of tree first, any mode
(3) Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
(4) Node Q can be locked by Ti in X,IX only if parent(Q) locked by Ti in IX
(5) Ti is two-phase
(6) Ti can unlock node Q only if none of Q's children are locked by Ti

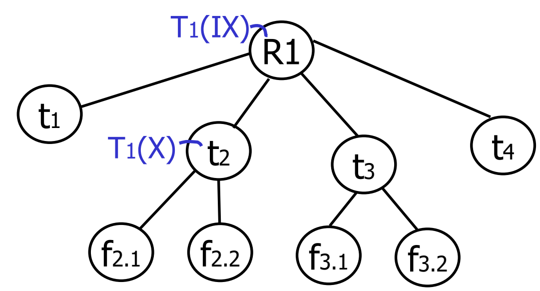
Exercise:

- Can T_2 access object $f_{2.2}$ in X mode? What locks will T_2 get?



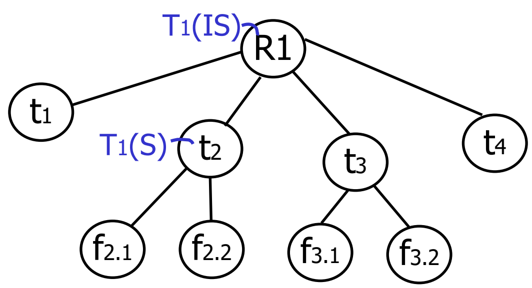
Exercise:

- Can T_2 access object $f_{2.2}$ in X mode? What locks will T_2 get?



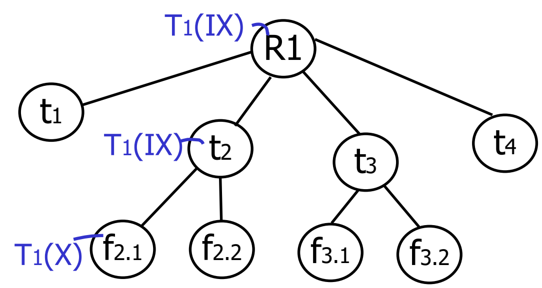
Exercise:

- Can T_2 access object $f_{3.1}$ in X mode? What locks will T_2 get?



Exercise:

- Can T_2 access object $f_{2.2}$ in S mode? What locks will T_2 get?



Isolation Levels

Hard and Soft Transactions in RDBMS

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time.
- Only one level (“serializable”) = ACID transactions.
- Each DBMS implements transactions in its own way.

Choosing the Isolation Level

- Within a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL *X*

where *X* =

1. SERIALIZABLE
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED