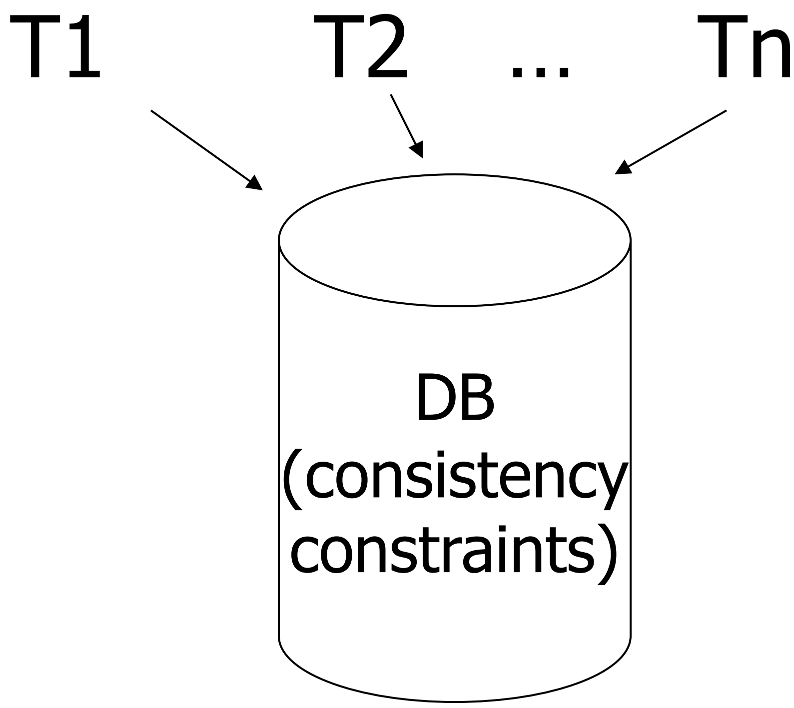
Ullman et al. : Database System Principles

Notes 09: Concurrency Control

Chapter 18 [18] Concurrency Control



Interactions among concurrently executing transactions can cause the database state to become inconsistent, even when the transactions individually preserve correctness of the state, and there is no system failure.

Thus, the timing of individual steps of different transactions needs to be regulated in some manner.

This regulation is the job of the *scheduler component of the DBMS, and the* general process of assuring that transactions preserve consistency when executing simultaneously is called *concurrency control.*

In most situations, the scheduler will execute the reads and writes directly, first calling on the buffer manager if the desired database element is not in a buffer.

However, in some situations, it is not safe for the request to be executed immediately.

The scheduler must delay the request.

In some concurrency-control techniques, the scheduler may even abort the transaction that issued the request

Example:

T1: Read(A)
$A \leftarrow A+100$
Write(A)
Read(B)
$B \leftarrow B+100$
Write(B)

T2: Read(A)
$A \leftarrow A\times2$
Write(A)
Read(B)
$B \leftarrow B\times2$
Write(B)

Constraint: $A=B$

A *schedule is a sequence of the important actions taken by one or more transactions.*

When studying concurrency control, the important read and write actions take place in the main-memory buffers, not the disk.

That is, a database element *A that is brought to a buffer by some transaction T may be read or* written in that buffer not only by *T but by other transactions that access A.*

A schedule is *serial if its actions consist of all the actions of one transaction,* then all the actions of another transaction, and so on.

No mixing of the actions is allowed.

Schedule A (serial)

T1	T2	A	B
		25	25
Read(A); A $\leftarrow$ A+100			
Write(A);		125	
Read(B); B $\leftarrow$ B+100;			
Write(B);			125
	Read(A);A $\leftarrow$ A$\times$2;		
	Write(A);	250	
	Read(B);B $\leftarrow$ B$\times$2;		
	Write(B);		250
		250	250

Schedule B (serial)

T1	T2	A	B
		25	25
	Read(A);A $\leftarrow$ A$\times$2;		
	Write(A);	50	
	Read(B);B $\leftarrow$ B$\times$2;		
	Write(B);		50
Read(A); A $\leftarrow$ A+100			
Write(A);		150	
Read(B); B $\leftarrow$ B+100;			
Write(B);			150
		150	150

A schedule *S is serializable if there is a serial* schedule *S' such that for every initial database state, the effects of S and S'* are the same.

Schedule C (serializable)

T1	T2	A	B
		25	25
Read(A); A $\leftarrow$ A+100			
Write(A);		125	
	Read(A);A $\leftarrow$ A$\times$2;		
	Write(A);	250	
Read(B); B $\leftarrow$ B+100;			
Write(B);			125
	Read(B);B $\leftarrow$ B$\times$2;		
	Write(B);		250
		250	250

Schedule D (not serializable)

T1	T2	A	B
		25	25
Read(A); A $\leftarrow$ A+100			
Write(A);		125	
	Read(A);A $\leftarrow$ A$\times$2;		
	Write(A);	250	
	Read(B);B $\leftarrow$ B$\times$2;		
	Write(B);		50
Read(B); B $\leftarrow$ B+100;			
Write(B);			150
		250	150

Schedule E

Same as Schedule D
but with new T2′

T1	T2′	A	B
		25	25
Read(A); A $\leftarrow$ A+100			
Write(A);		125	
	Read(A);A $\leftarrow$ A$\times$1;		
	Write(A);	125	
	Read(B);B $\leftarrow$ B$\times$1;		
	Write(B);		25
Read(B); B $\leftarrow$ B+100;			
Write(B);			125
		125	125

- Want schedules that are “good”, regardless of
  - initial state and
  - transaction semantics
- Only look at order of read and writes

It is not realistic for the scheduler to concern itself with the details of computation undertaken by transactions.

Any database element *A that a transaction T writes is given a value* that depends on the database state in such a way that no arithmetic coincidences occur.

To make the notation precise:

1. An *action is an expression of the form* $r_i(X)$ *or* $w_i(X)$, *meaning that* transaction T_i, reads or writes, respectively, the database element *X*.

2. A *transaction* T_i *is a sequence of actions with subscript i.*

3. A *schedule S of a set of transactions T is a sequence of actions, in which* for each transaction T_i *in T, the actions of* T_i *appear in S in the same* order that they appear in the definition of T_i itself. We say that *S is an interleaving of the actions of the transactions of which it is composed.*

Example:

$$Sc = r_1(A)w_1(A)\underbrace{r_2(A)w_2(A)}\underbrace{r_1(B)w_1(B)}r_2(B)w_2(B)$$

$$Sc' = \underbrace{r_1(A)w_1(A)\ r_1(B)w_1(B)}_{T_1}\underbrace{r_2(A)w_2(A)r_2(B)w_2(B)}_{T_2}$$

Sc is ”equivalent” to a serial schedule.

Sc is ”good”

However, for Sd:

$$Sd = \underbrace{r_1(A)w_1(A)}r_2(A)w_2(A)\ r_2(B)w_2(B)\underbrace{r_1(B)w_1(B)}$$

- as a matter of fact,

  T_2 must precede T_1

  in any equivalent schedule,

  i.e., $T_2 \rightarrow T_1$

- $T_2 \rightarrow T_1$
- Also, $T_1 \rightarrow T_2$

$$\begin{tikzcd} T_1 \arrow[r, bend left] & T_2 \arrow[l, bend left] \end{tikzcd}$$

⇨ Sd cannot be rearranged into a serial schedule

⇨ Sd is not "equivalent" to any serial schedule

⇨ Sd is "bad"

Returning to Sc

$$Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$$

$$T_1 \rightarrow T_2 \qquad\qquad T_1 \rightarrow T_2$$

☛ no cycles $\Rightarrow$ Sc is "equivalent" to a serial schedule (in this case T_1,T_2)

Concepts

Transaction: sequence of $r_i(x)$, $w_i(x)$ actions

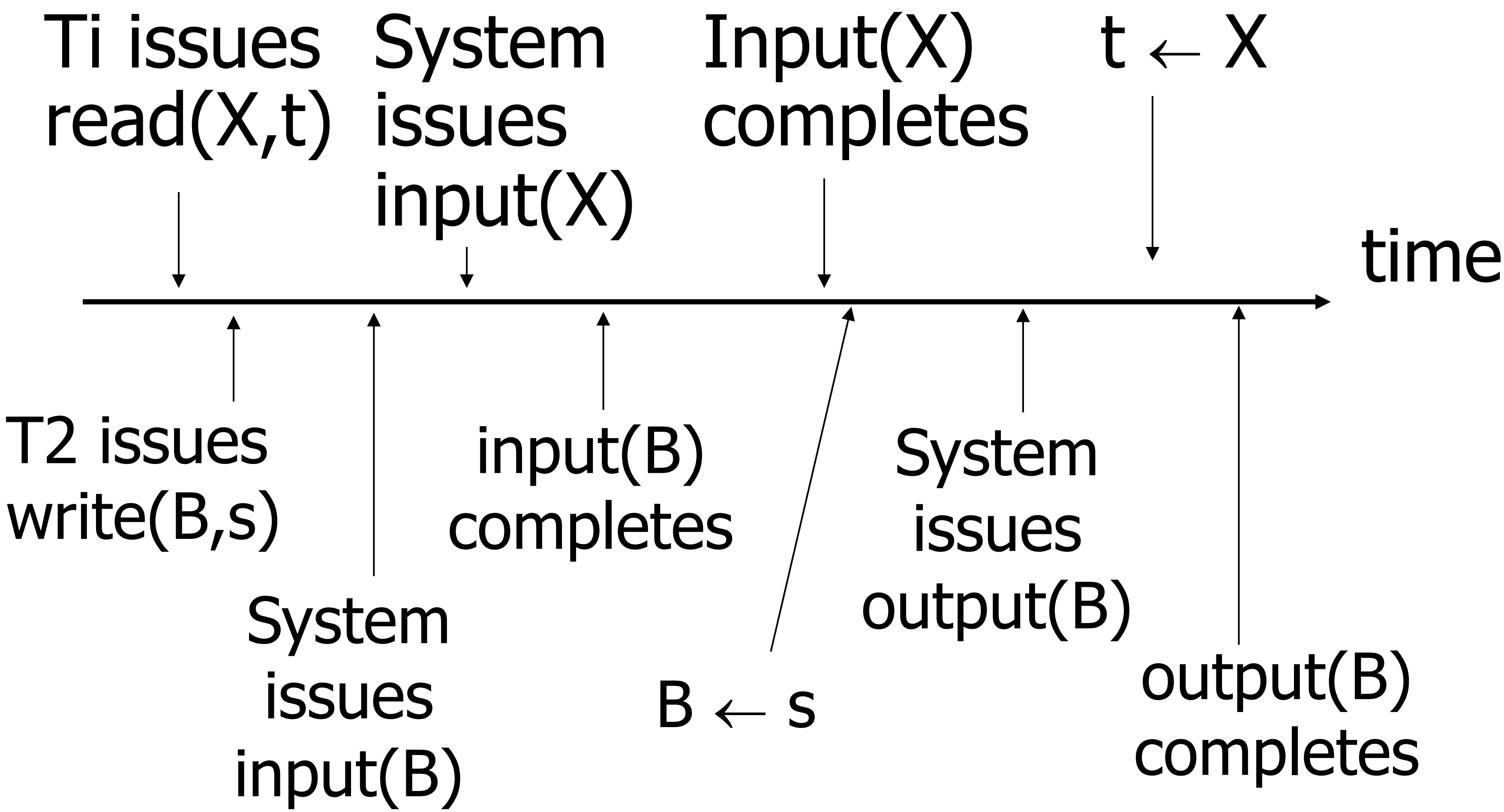
Conflicting actions: $r_1(A)$ $w_2(A)$ $w_1(A)$

$w_2(A)$ $r_1(A)$ $w_2(A)$

Schedule: represents chronological order in which actions are executed

Serial schedule: no interleaving of actions or transactions

What about concurrent actions?

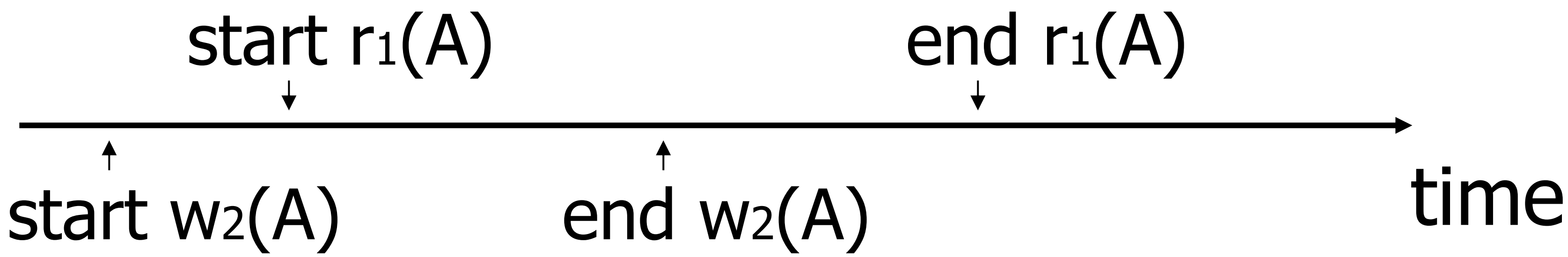


So net effect is either

- $S = \ldots r_1(X) \ldots w_2(B) \ldots$ or
- $S = \ldots w_2(B) \ldots r_1(X) \ldots$

We assume that elementary actions are atomic, and follow each other.

What about conflicting, concurrent actions on same object?



- Assume equivalent to either $r_1(A)\ w_2(A)$

  or $w_2(A)\ r_1(A)$

- $\Rightarrow$ low level synchronization mechanism
- Assumption called "atomic actions"

Definition

S_1, S_2 are conflict equivalent schedules
if S_1 can be transformed into S_2 by a series of non-conflicting swaps of adjacent actions.

A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

Example:

$r_1(A);\ w_1(A);\ r_2(A);\ \underline{w_2(A);\ r_1(B);}\ w_1(B);\ r_2(B);\ w_2(B);$

We claim this schedule is conflict-serializable.

1. $r_1(A);\ w_1(A);\ r_2(A);\ \underline{w_2(A);\ r_1(B);}\ w_1(B);\ r_2(B);\ w_2(B);$
2. $r_1(A);\ w_1(A);\ \underline{r_2(A);\ r_1(B);}\ w_2(A);\ w_1(B);\ r_2(B);\ w_2(B);$
3. $r_1(A);\ w_1(A);\ r_1(B);\ r_2(A);\ \underline{w_2(A);\ w_1(B);}\ r_2(B);\ w_2(B);$
4. $r_1(A);\ w_1(A);\ r_1(B);\ \underline{r_2(A);\ w_1(B);}\ w_2(A);\ r_2(B);\ w_2(B);$
5. $r_1(A);\ w_1(A);\ r_1(B);\ w_1(B);\ r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);$

- Conflict-serializability is a sufficient condition for serializability i.e., a conflict-serializable schedule is a serializable schedule.

- Conflict-serializability is not required for a schedule to be serializable.

- Schedulers in commercial systems generally use conflict-serializability when they need to guarantee serializability.

Precedence graph P(S) (S is schedule)

Nodes: transactions in S

Arcs: Ti $\rightarrow$ Tj whenever

- $p_i(A)$, $q_j(A)$ are actions in S
- $p_i(A) <_S q_j(A)$ (pi(A) precedes qj(A) in S)
- at least one of p_i, q_j is a write

Exercise:

- What is P(S) for
  $S = w_3(A)\ w_2(C)\ r_1(A)\ w_1(B)\ r_1(C)\ w_2(A)\ r_4(A)\ w_4(D)$

- Is S serializable?

Another Exercise:

- What is P(S) for
  $S = w_1(A)\ r_2(A)\ r_3(A)\ w_4(A)$?

Lemma

S_1, S_2 conflict equivalent $\Rightarrow$ $P(S_1)=P(S_2)$

Lemma

S_1, S_2 conflict equivalent $\Rightarrow P(S_1)=P(S_2)$

Proof:

Assume $P(S_1) \neq P(S_2)$

$\Rightarrow \exists\ T_i: T_i \rightarrow T_j$ in S_1 and not in S_2

$$\left.\begin{array}{l} \Rightarrow S_1 = \ldots p_i(A) \ldots\ q_j(A) \ldots \\ \quad\ S_2 = \ldots q_j(A) \ldots p_i(A) \ldots \end{array}\right\} \begin{array}{l} p_i, q_j \\ \text{conflict} \end{array}$$

$\Rightarrow S_1, S_2$ not conflict equivalent

Note: $P(S_1)=P(S_2) \not\Rightarrow S_1, S_2$ conflict equivalent

Counter example:

$S_1=w_1(A)\ r_2(A) \qquad w_2(B)\ r_1(B)$ (cannot swap)

$S_2=r_2(A)\ w_1(A) \qquad r_1(B)\ w_2(B)$

Theorem

$P(S_1)$ acyclic $\Longleftrightarrow$ S_1 conflict serializable

($\Leftarrow$) Assume S_1 is conflict serializable

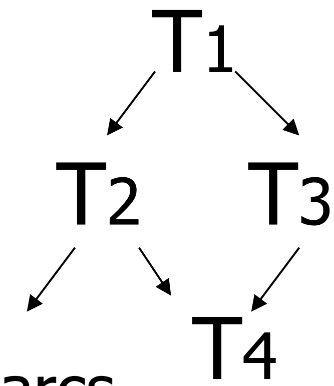
$\Rightarrow \exists\ S_s$: S_s, S_1 conflict equivalent

$\Rightarrow P(S_s) = P(S_1)$

$\Rightarrow P(S_1)$ acyclic since $P(S_s)$ is acyclic

Theorem

$P(S_1)$ acyclic $\Longleftrightarrow$ S_1 conflict serializable



($\Rightarrow$) Assume $P(S_1)$ is acyclic

Transform S_1 as follows:

(1) Take T_1 to be transaction with no incident arcs

(2) Move all T_1 actions to the front

$$S_1 = \ldots\ldots\ q_j(A) \ldots\ldots p_1(A) \ldots\ldots$$

(3) we now have S_1 = < T_1 actions >< ... rest ... >

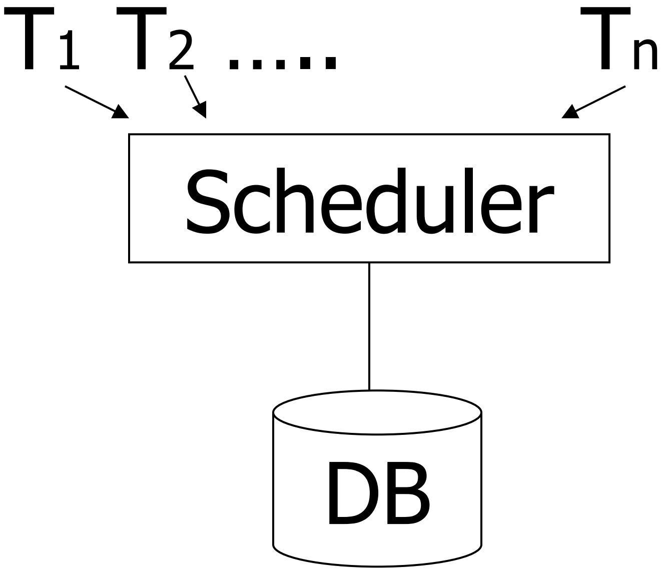
(4) repeat above steps to serialize rest!

How to enforce serializable schedules?

Option 1: run system, recording P(S);
at end of day, check for P(S) cycles and declare if execution was good

How to enforce serializable schedules?

Option 2: prevent P(S) cycles from occurring

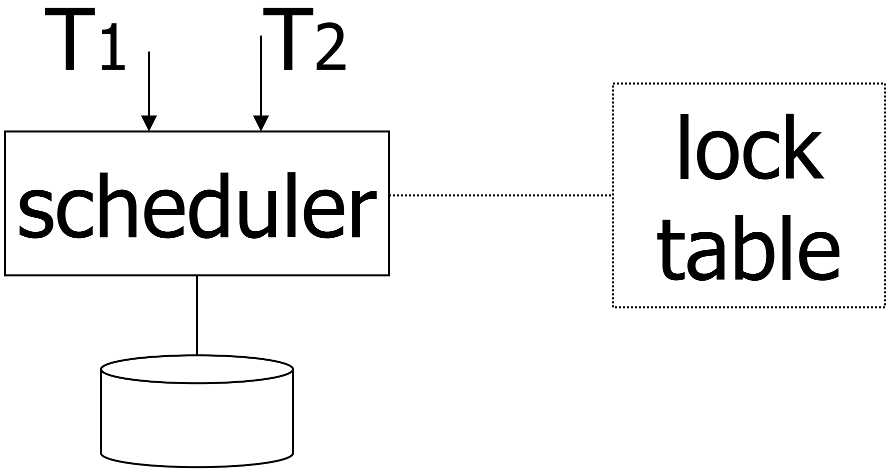


A locking protocol

Two new actions:

lock (exclusive): l_i (A)

unlock: u_i (A)



Rule #1: Consistency of transactions

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

1. A transaction can only read or write an element if it previously was granted a lock on that element and hasn't yet released the lock.
2. If a transaction locks an element, it must later unlock that element.

Rule #2 Legality of schedules

$S = \ldots\ldots\ldots \; l_i(A) \; \ldots\ldots\ldots\ldots\ldots \; u_i(A) \; \ldots\ldots\ldots$

no $l_j(A)$

Locks must have their intended meaning: no two transactions may have locked the same element without one having first released the lock.

Exercise:

- What schedules are legal?
  What transactions are consistent?

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

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

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

Exercise:

- What schedules are legal?
  What transactions are consistent?

S1 = $l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$
$r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$ (S1: not legal)

S2 = $l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$ (T1: not consistent)
$l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$ (S2: not legal)

S3 = $l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
$l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$
(S3: legal schedule, T1,T2,T3: consistent transactions)

Schedule F (legal schedule of consistent transactions)

(still not equivalent to a serial schedule)

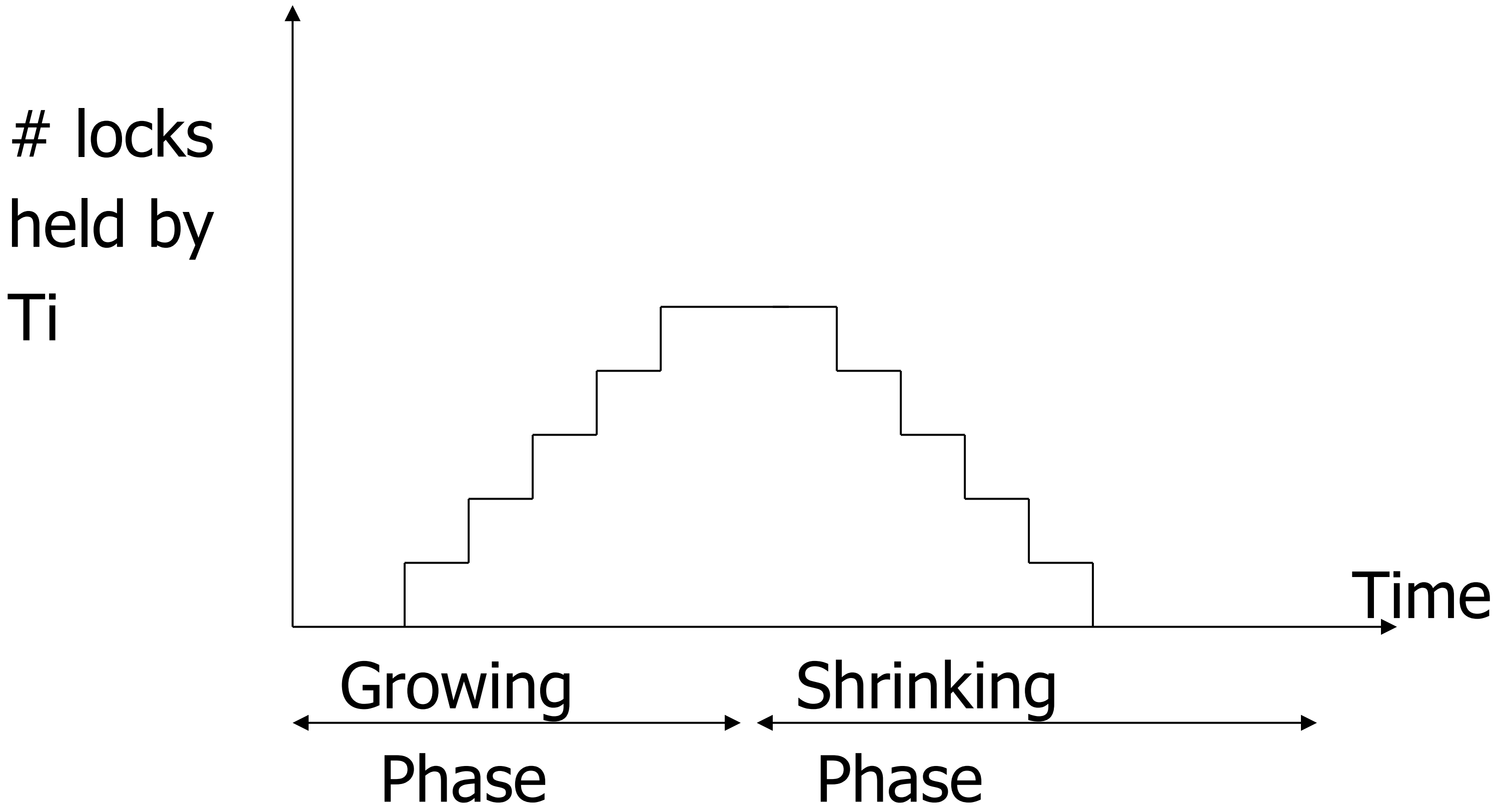
T1	T2	A	B
		25	25
$l_1(A)$;Read(A)			
$A \leftarrow A+100$;Write(A);$u_1(A)$		125	
	$l_2(A)$;Read(A)		
	$A \leftarrow Ax2$;Write(A);$u_2(A)$	250	
	$l_2(B)$;Read(B)		
	$B \leftarrow Bx2$;Write(B);$u_2(B)$		50
$l_1(B)$;Read(B)			
$B \leftarrow B+100$;Write(B);$u_1(B)$			150
		250	150

Rule #3 Two phase locking (2PL)

for transactions

$T_i = \ldots\ldots.\ l_i(A)\ \ldots\ldots\ldots\ldots\ u_i(A)\ \ldots\ldots\ldots$

no unlocks　　　　　　　　no locks



locks
held by
Ti
Time
Growing
Phase
Shrinking
Phase

Schedule G

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

delayed

Schedule G

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

delayed

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)
	A ← Ax2;Write(A); l_2(B) delayed
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 G "good" (equivalent to a serial)

		A	B
T1	T2	25	25
$l_1(A)$;Read(A); A←A+100			
Write(A); $l_1(B)$;$u_1(A)$		125	
	$l_2(A)$;Read(A)		
	A←Ax2;Write(A);$u_2(A)$	250	
	$l_2(B)$; delayed		
Read(B);			
B←B+100;Write(B);$u_1(B)$			125
	$l_2(B)$;Read(B)		
	B←Bx2;Write(B);$u_2(B)$	250	250

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

(consistency, legality, 2PL)

<u>Theorem</u> Rules #1,2,3 $\Longrightarrow$ conflict serializable schedule

(consistency, legality, 2PL)

Intuitively, each two-phase-locked transaction may be thought to execute in its entirety at the instant it issues its first unlock request. In a conflict-equivalent serial schedule transactions appear in the same order as their first unlocks.

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

Proof:

BASIS: If n = 1, there is nothing to do; S is already a serial schedule.

INDUCTION: Suppose S involves n transactions $T_1, T_2, \ldots, T_n$, and let T_i be the transaction with the first unlock action in the entire schedule S, say $u_i(x)$.

We claim it is possible to move all the read and write actions of T_i forward to the beginning of the schedule without passing any conflicting reads or writes.

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

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

Proof of lemma:

$T_i \rightarrow T_j$ 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

Schedule H (T$_2$ reversed)

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

Deadlock !!!

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

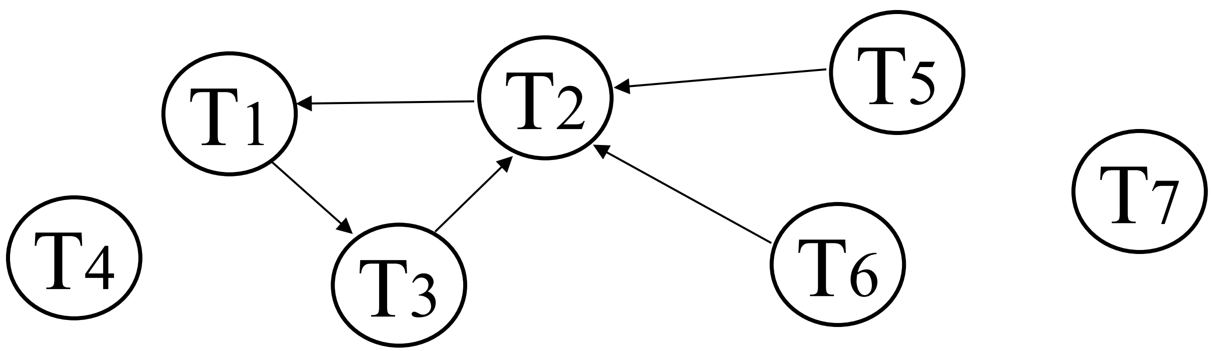
E.g., Schedule H = $\underbrace{\qquad\qquad\qquad\qquad}_{\text{This space intentionally left blank!}}$

Deadlocks

- Detection
  - Wait-for graph
- Prevention
  - Resource ordering
  - Timeout
  - Wait-die
  - Wound-wait

Deadlock Detection

- Build Wait-For graph (Ti -> Tj edge if Ti waits for Tj)
- Use lock table structures
- Build incrementally or periodically
- When cycle found, rollback victim



<u>Resource Ordering</u> (prevention)

- Order all elements $A_1, A_2, \ldots, A_n$
- A transaction T can lock A_i after A_j only if $i > j$

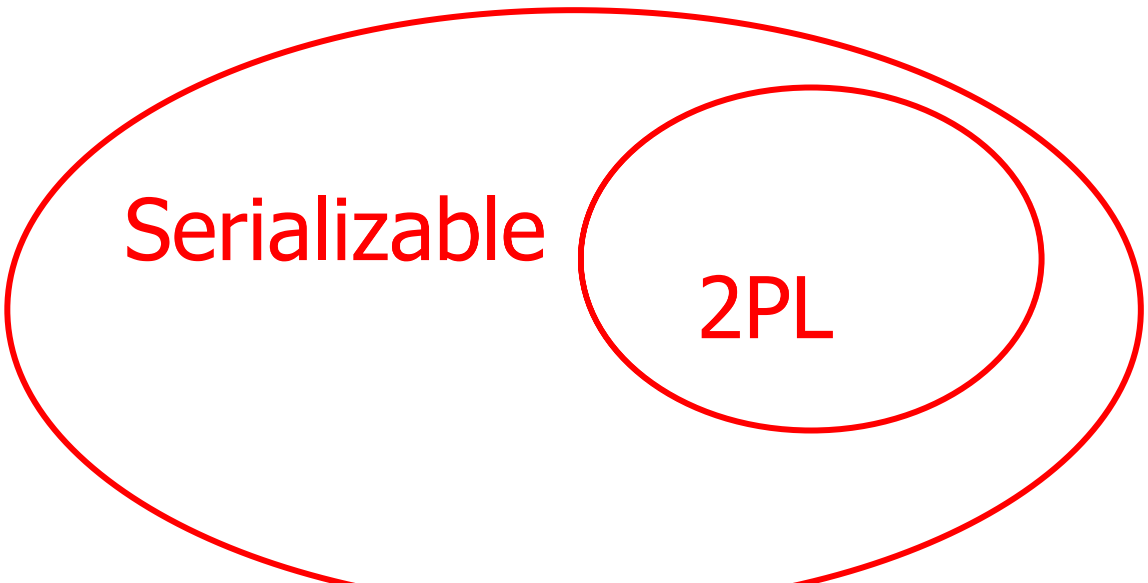
Problem : Ordered lock requests not realistic in most cases

Timeout

- If transaction waits more than L sec., roll it back!
- Simple scheme
- Hard to select L

2PL subset of Serializable

(schedules that can be implemented by 2PL locks are subset of serializable schedules)



S1: w1(x) w3(x) w2(y) w1(y)

- S1 cannot be achieved via 2PL:
  The lock by T1 for y must occur after w2(y), so the unlock by T1 for x must occur after this point (and before w3(x)). Thus, w3(x) cannot occur under 2PL where shown in S1 because T1 holds the x lock at that point.
- However, S1 is serializable (conflict-serializable) (equivalent to T2, T1, T3).

If you need a bit more practice:
Are our schedules S_C and S_D 2PL schedules?

S_C: w1(A) w2(A) w1(B) w2(B)

S_D: w1(A) w2(A) w2(B) w1(B)

- 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

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-t_i(A): lock A in t mode (t is S or X)

u-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 Consistency of transactions

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

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

- A transaction may not write without holding an exclusive lock, and may not read without holding some lock.
- All locks must be followed by an unlock of the same element.

Two-phase locking of transactions:

Locking must precede unlocking.

Legality of schedules:

An element may either be locked exclusively by one transaction or by several in shared mode, but not both.

- 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

$S = \ldots .l\text{-}S_i(A) \ldots \quad \ldots u_i(A) \ldots$

$\longleftrightarrow$

no $l\text{-}X_j(A)$

$S = \ldots\ l\text{-}X_i(A) \ldots \quad \ldots u_i(A) \ldots$

$\longleftrightarrow$

no $l\text{-}X_j(A)$

no $l\text{-}S_j(A)$

A way to summarize Rule #2

Compatibility matrix

Comp

	S	X
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!

(It is like a new lock, allowed only in the growing phase)

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

- can be allowed in growing phase

(This is not a real release or unlock)

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

Proof: similar to X locks case

Detail:

l-t_i(A), l-r_j(A) do not conflict if comp(t,r)

l-t_i(A), u-r_j(A) do not conflict if comp(t,r)

Lock types beyond S/X

Examples:

(1) increment lock

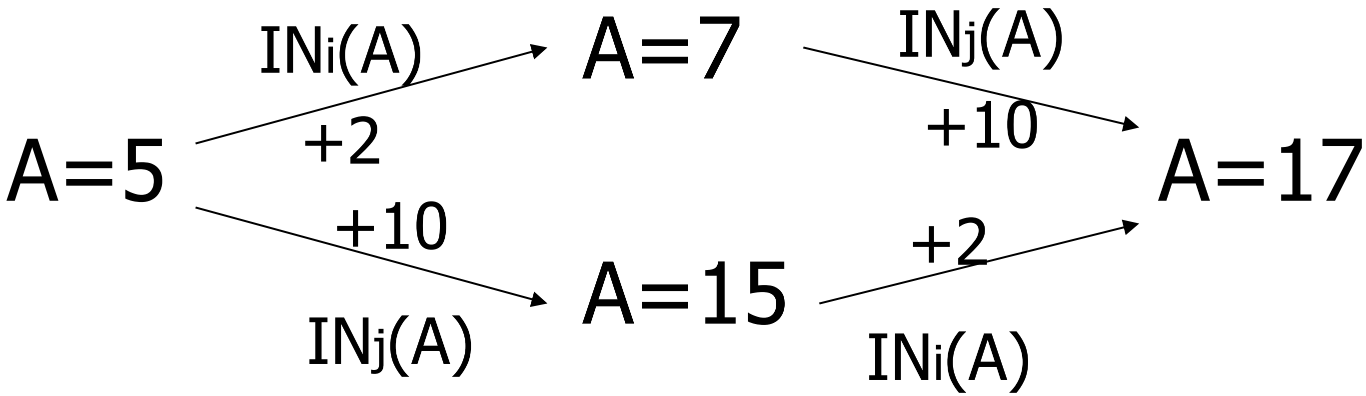
(2) update lock

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	T	F	T
X	F	F	F
U	F	F	F

-> not symmetric table

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

$$S_1 = \ldots l\text{-}S_1(A) \ldots l\text{-}S_2(A) \ldots l\text{-}U_3(A) \ldots \begin{cases} l\text{-}S_4(A) \ldots ? \\ l\text{-}U_4(A) \ldots ? \end{cases}$$

- 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

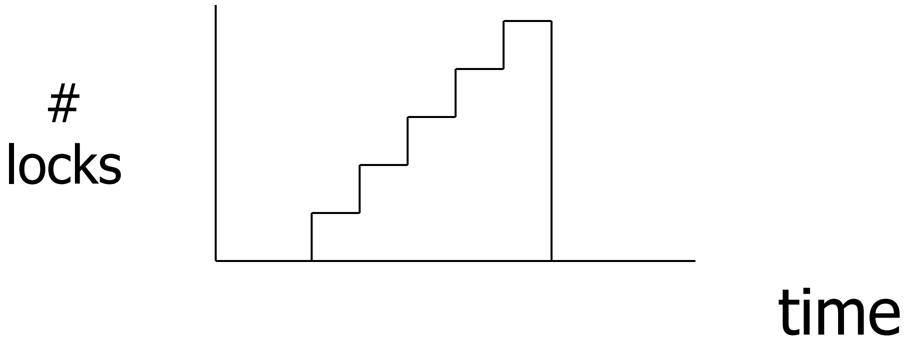
  (E.g., may not even provide
  CONFLICT-SERIALIZABLE schedules)

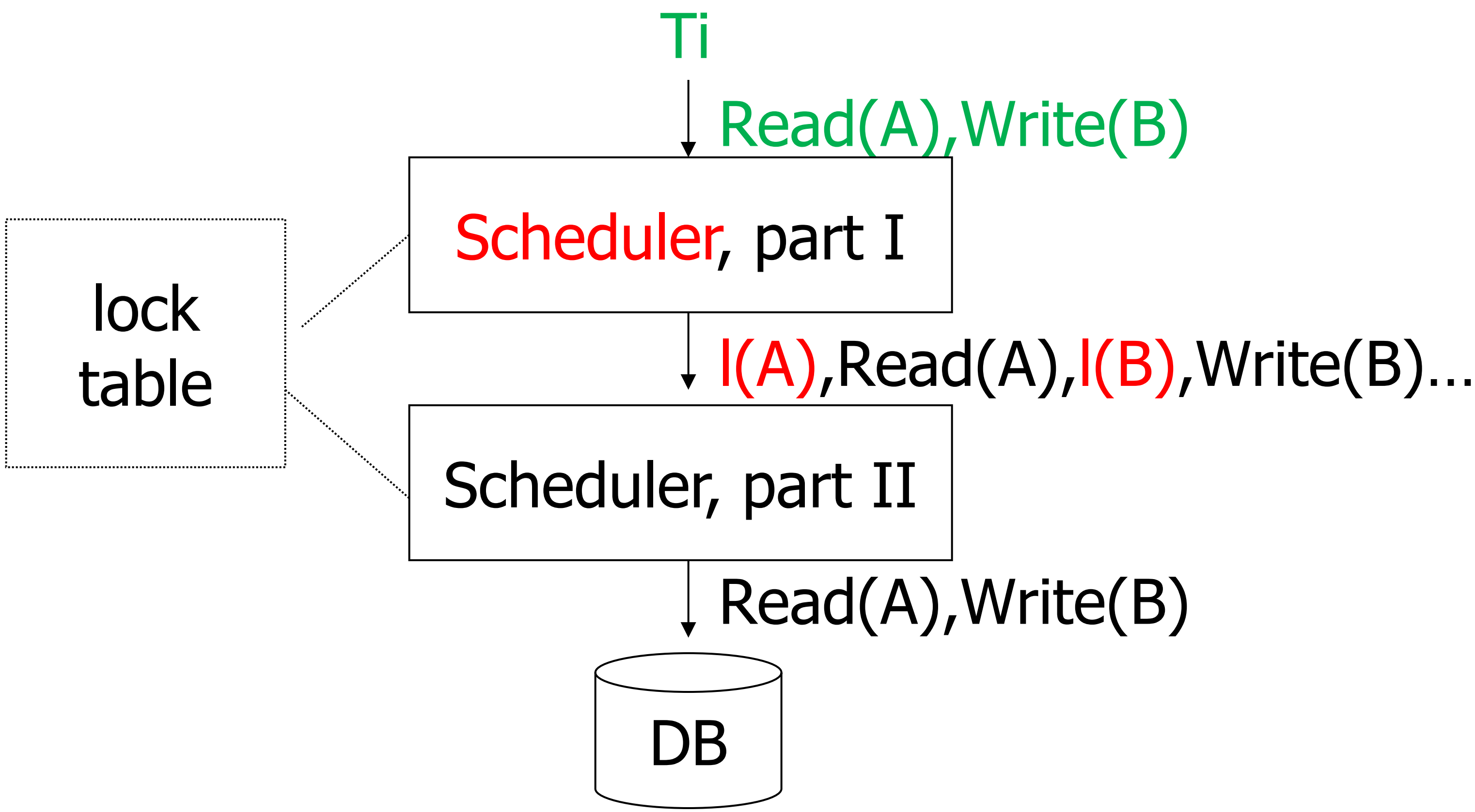
- But here is one (simplified) way ...

Sample Locking System:

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

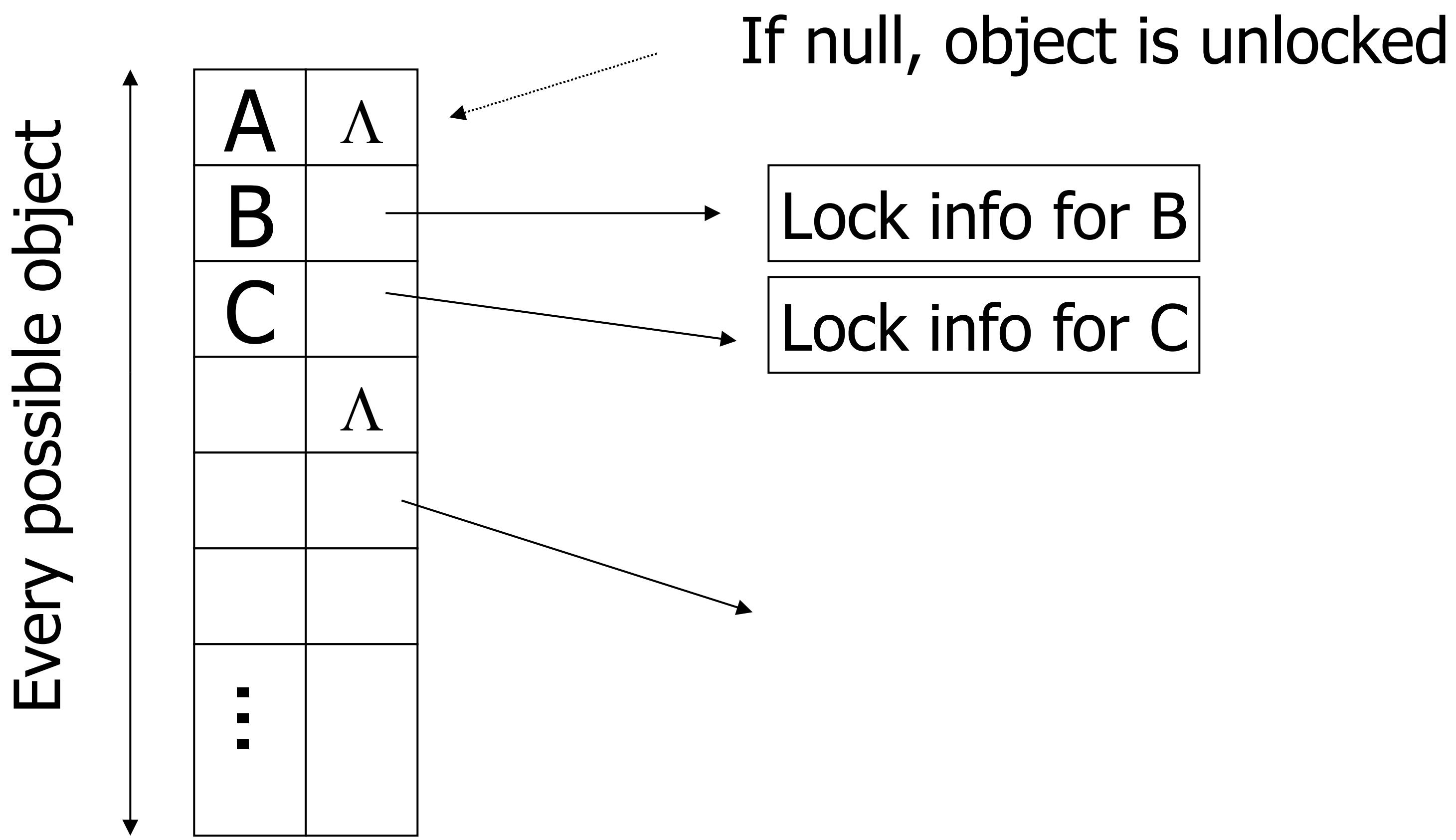
(2) Hold all locks until transaction commits



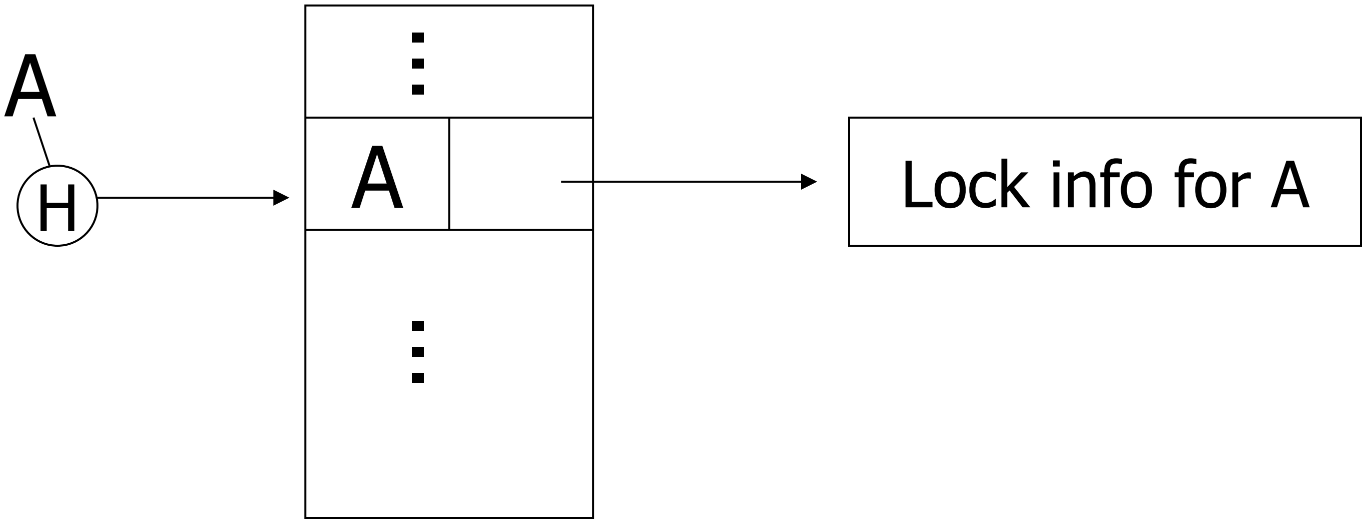


Scheduler requests locks (not transactions)

Lock table Conceptually

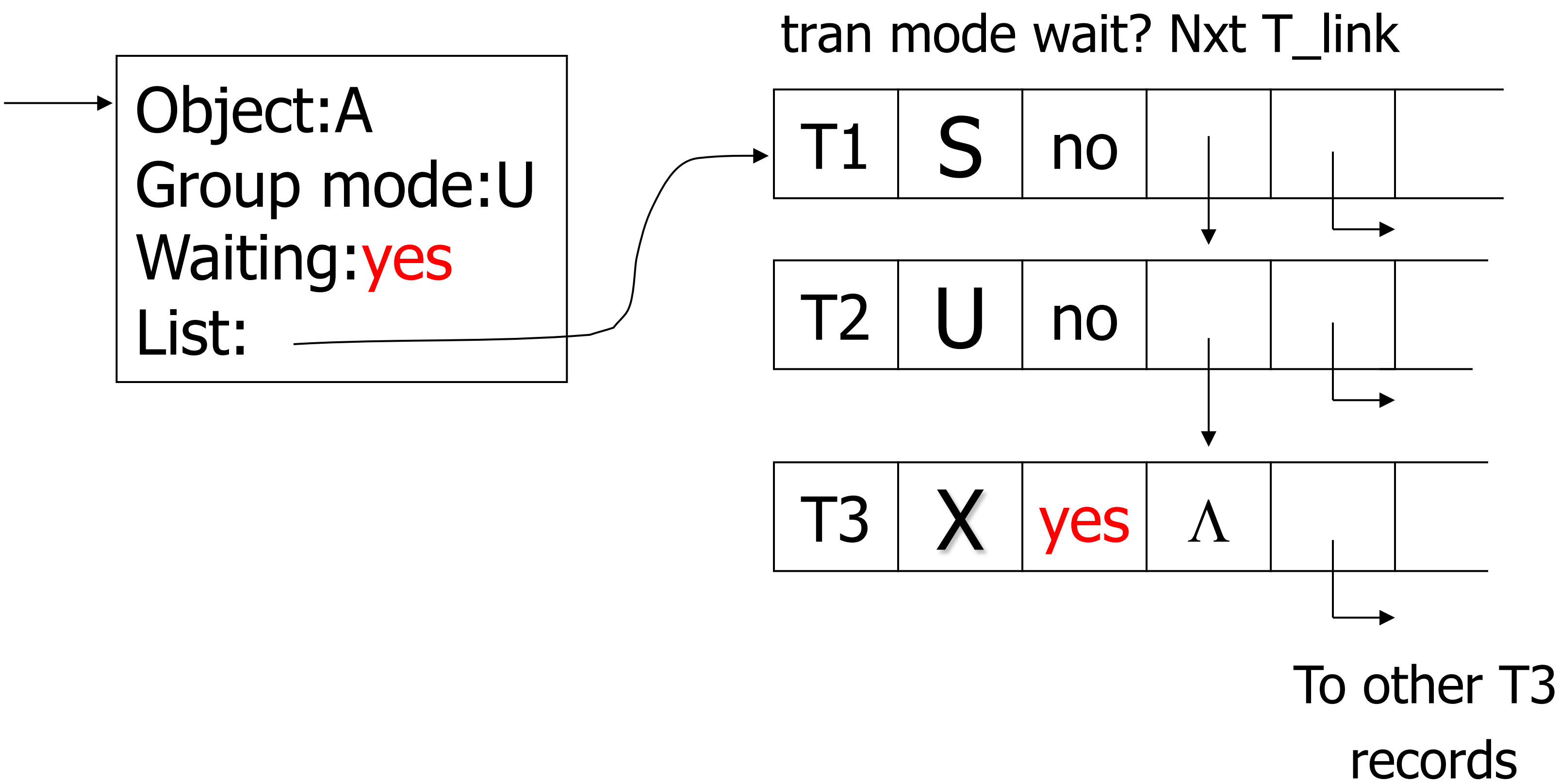


But use hash table:

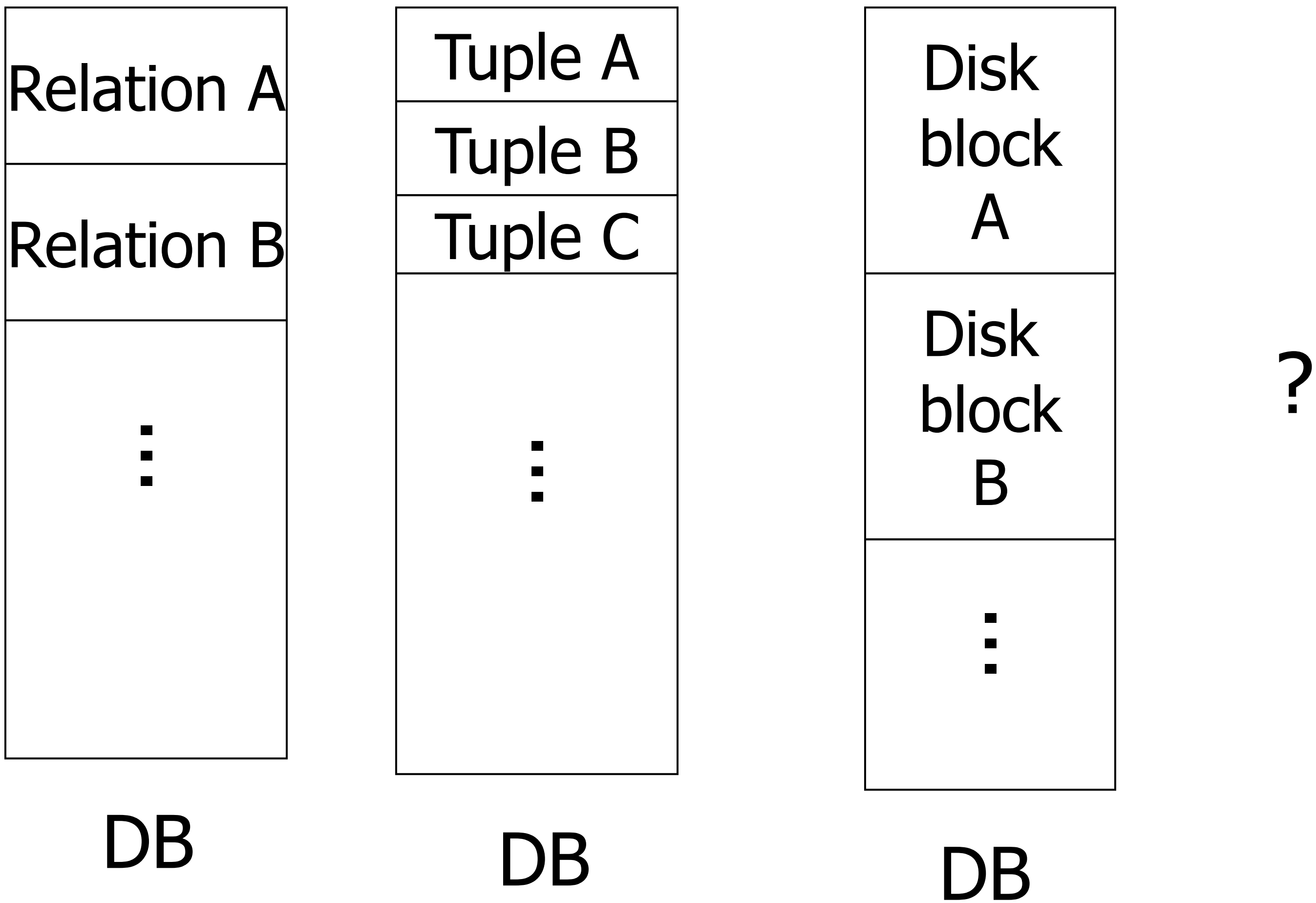


If object not found in hash table, it is unlocked

Lock info for A - example



What are the objects we lock?



- Locking works in any case, but should we choose small or large objects?

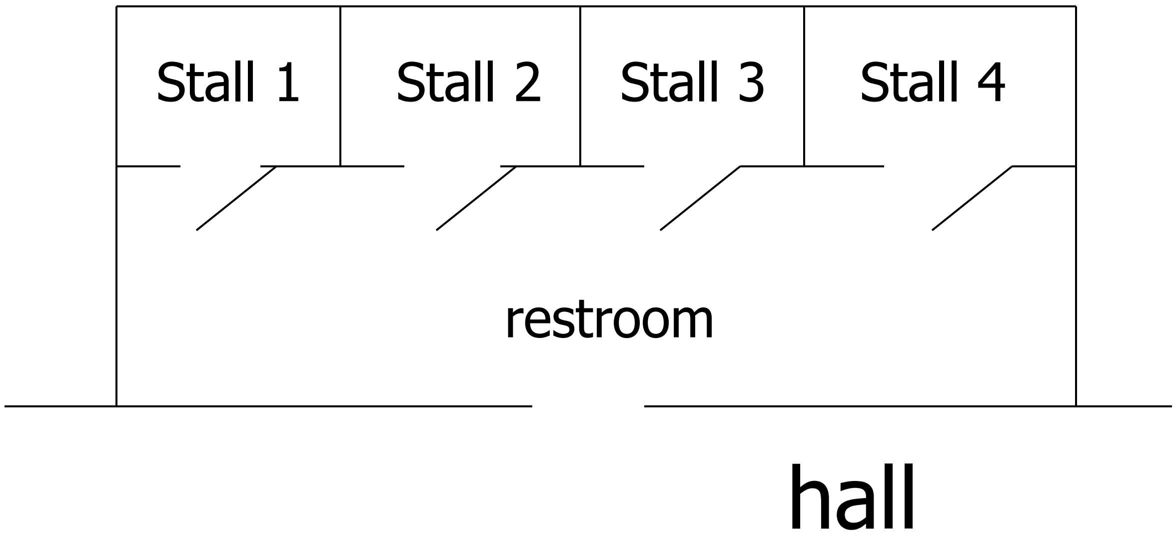
- Locking works in any case, but should we choose small or large objects?

- 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

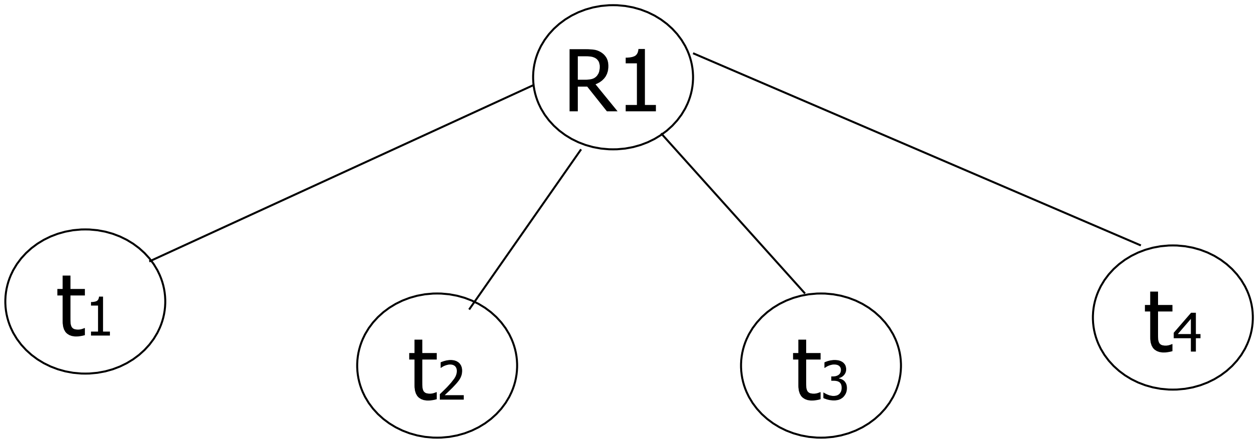
We <u>can</u> have it both ways!!

Ask any janitor to give you the solution...

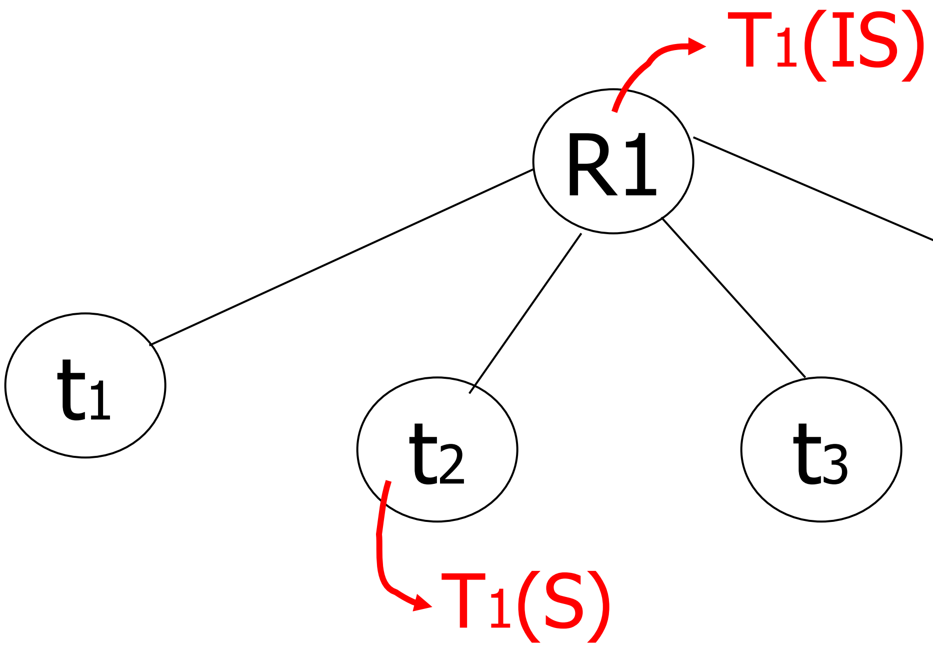
Should we close (lock) individual doors or restroom?



Example (R: relation, t: tuple)

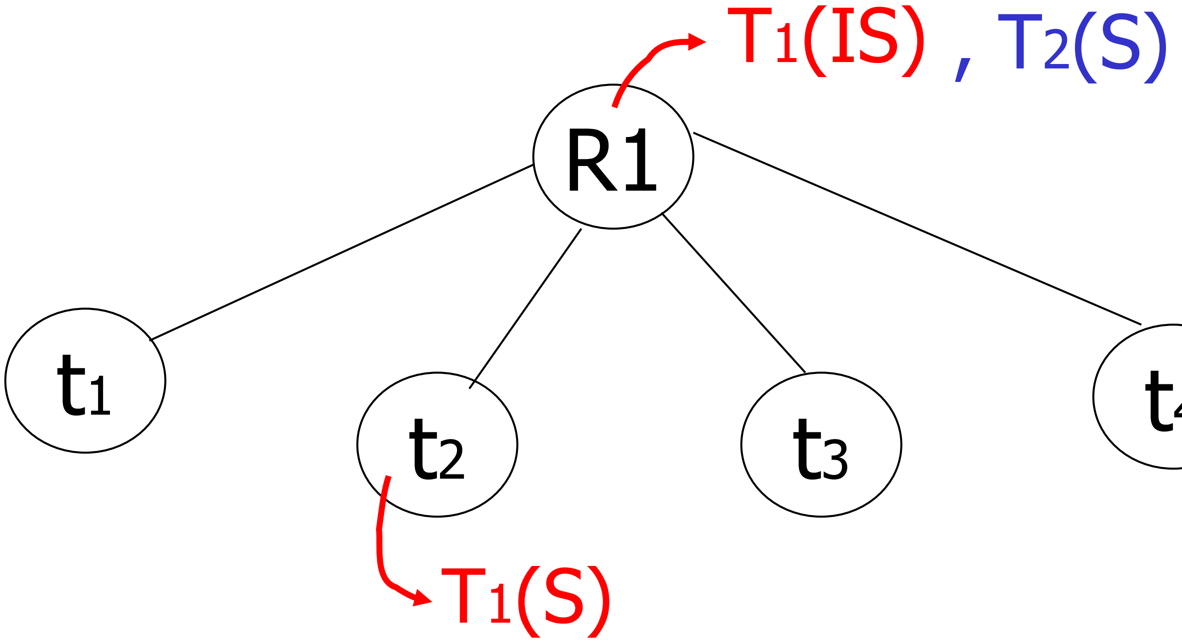


Example (Warning or Intention)

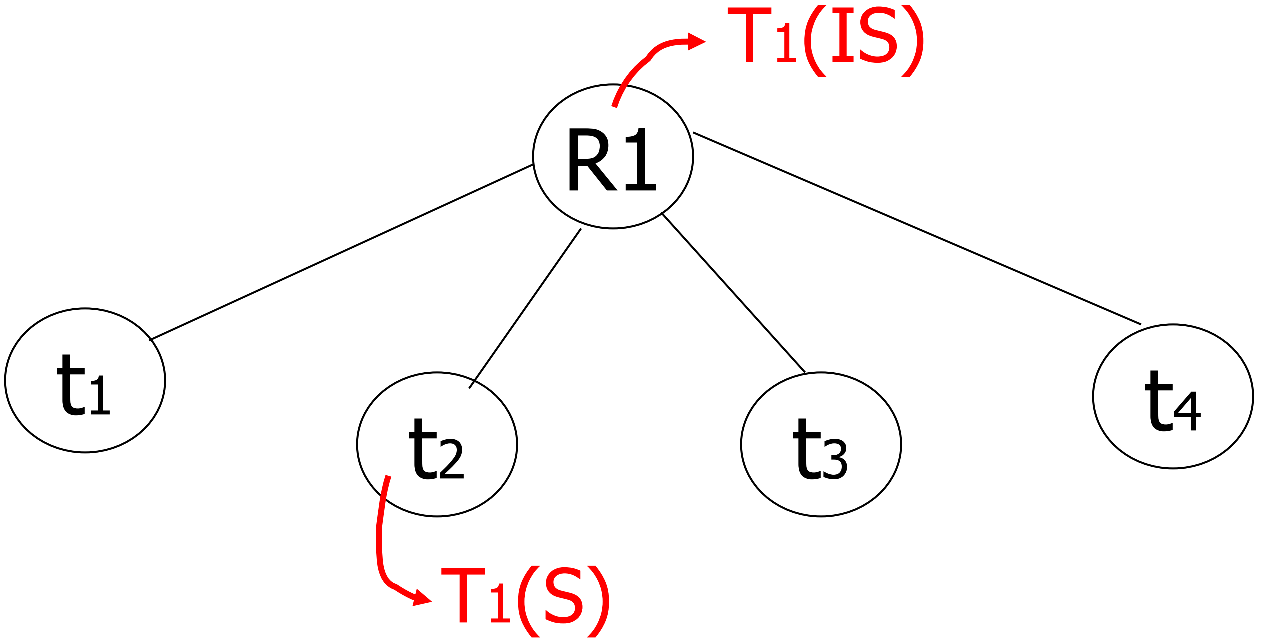


IS: Intention to obtain a shared lock on a subelement

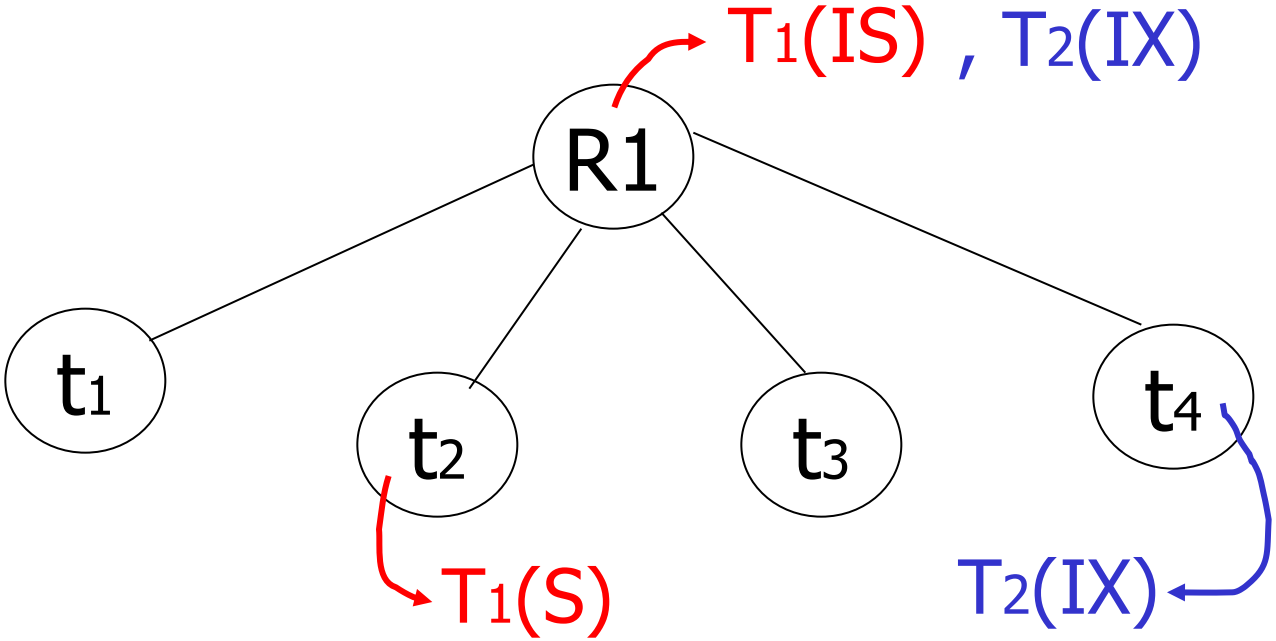
Example



Example (b)



Example (b)



Multiple granularity (SIX: goup mode)

Comp Requestor

Holder

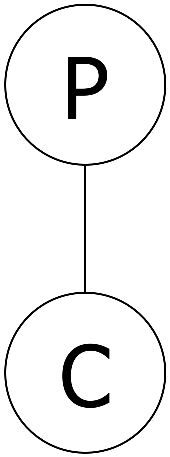
	IS	IX	S	SIX	X
IS					
IX					
S					
SIX					
X					

Multiple granularity (SIX: goup mode)

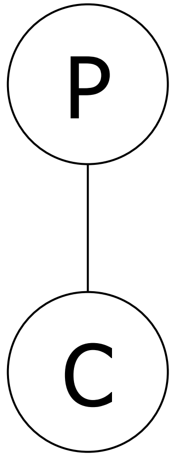
Comp

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

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



Parent locked in	Child can be locked by same transaction in
IS	IS, S
IX	IS, S, IX, X, SIX
S	none
SIX	X, IX, [SIX]
X	none



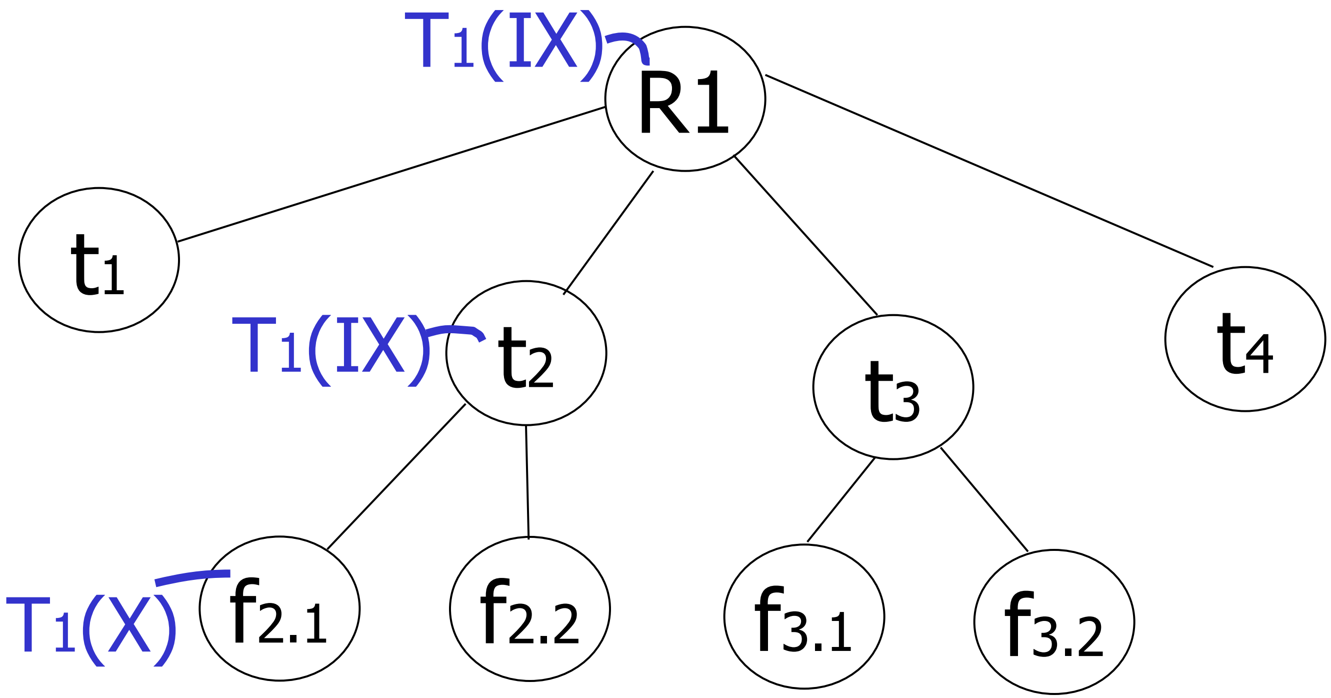
not necessary

Rules

(1) Follow multiple granularity compat. 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,SIX,IX only if parent(Q) locked by Ti in IX,SIX
(5) Ti is two-phase
(6) Ti can unlock node Q only if none of Q's children are locked by Ti

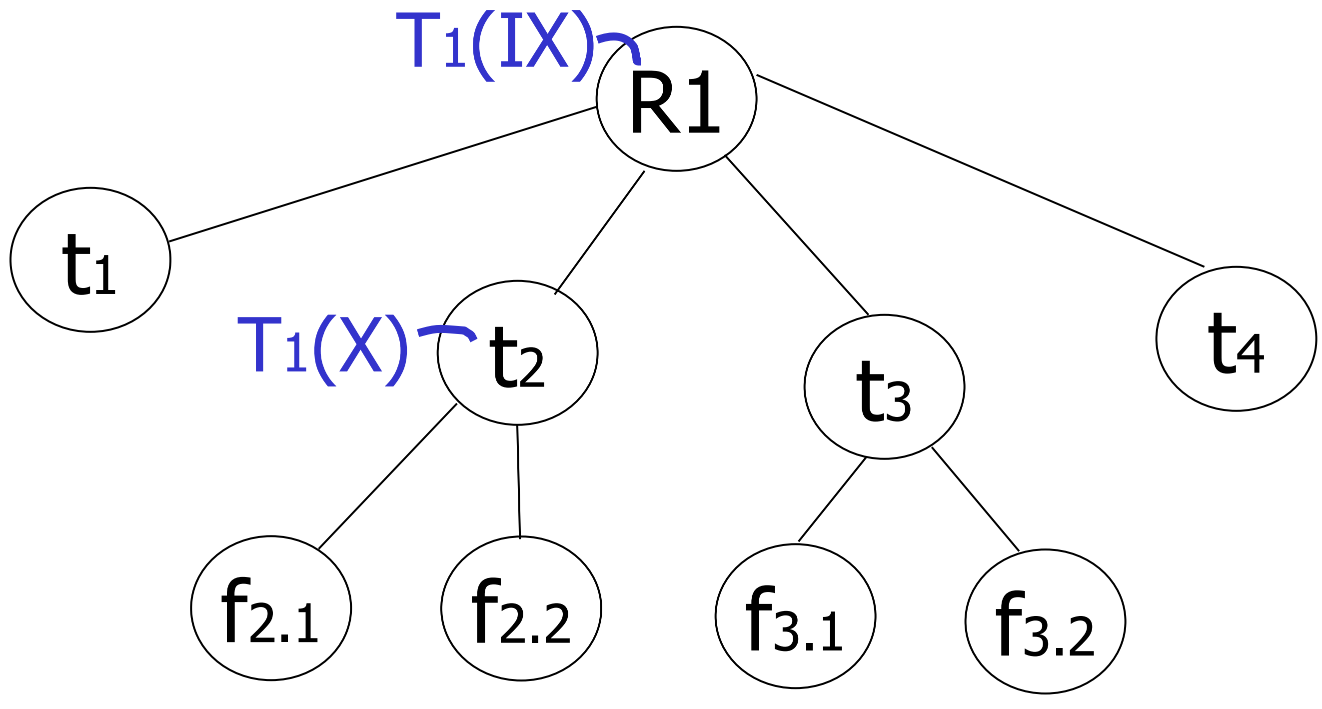
Exercise:

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



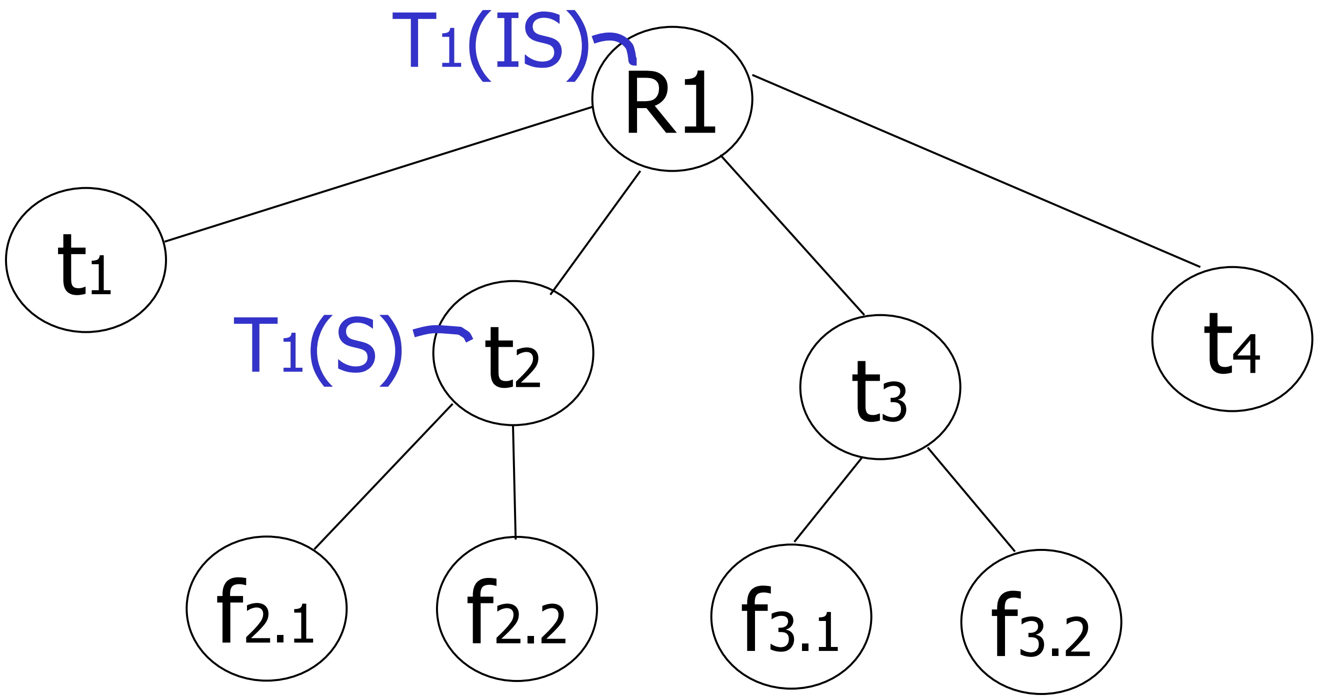
Exercise:

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



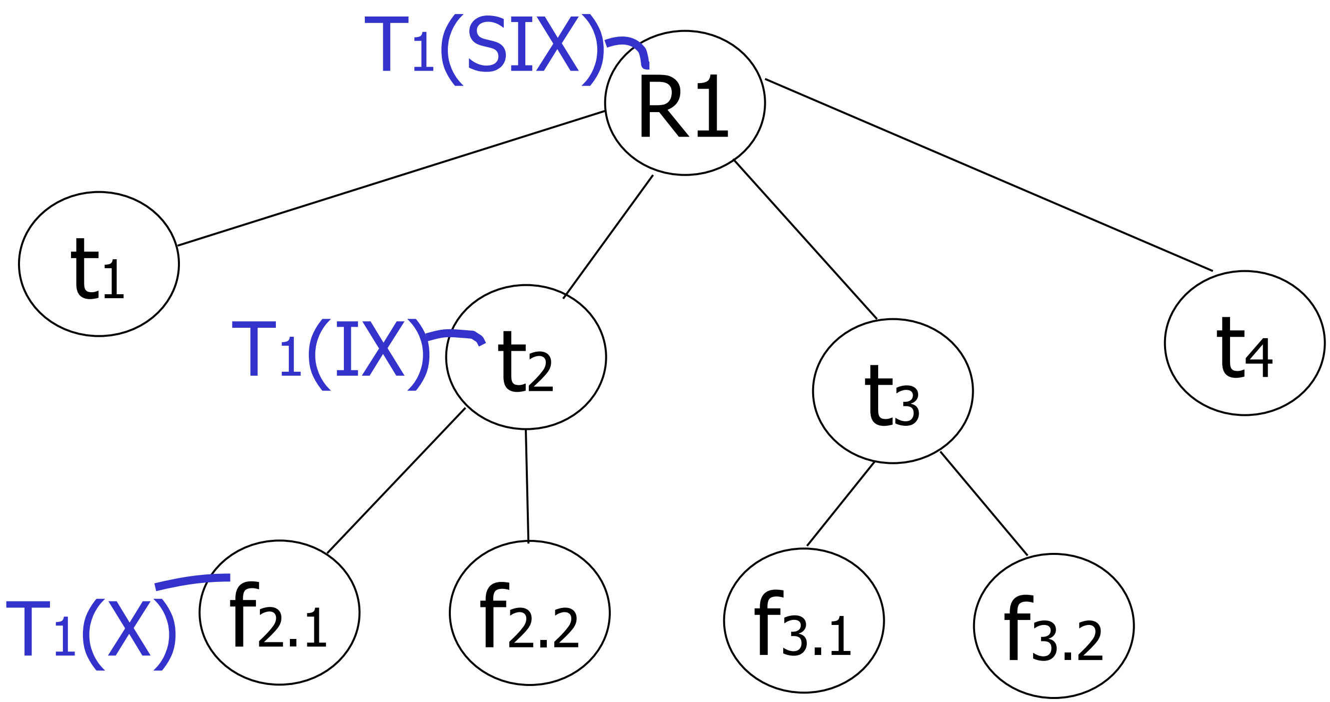
Exercise:

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



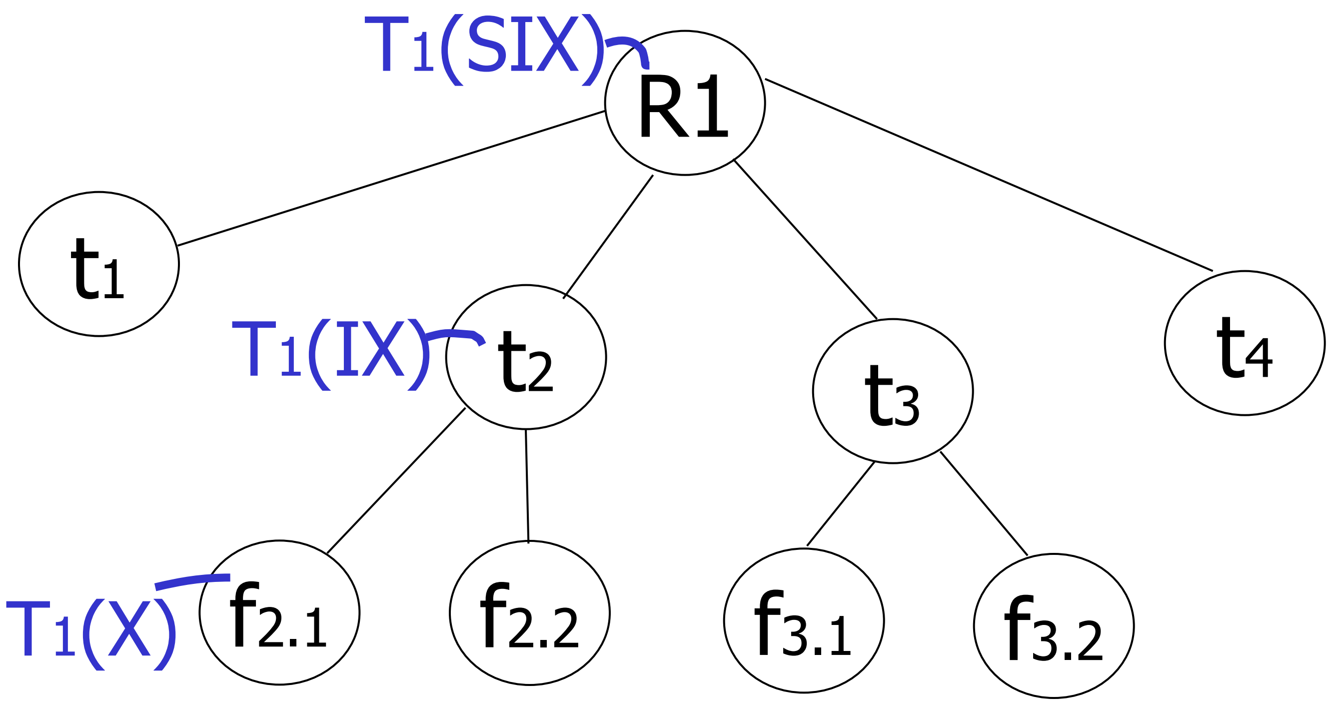
Exercise:

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

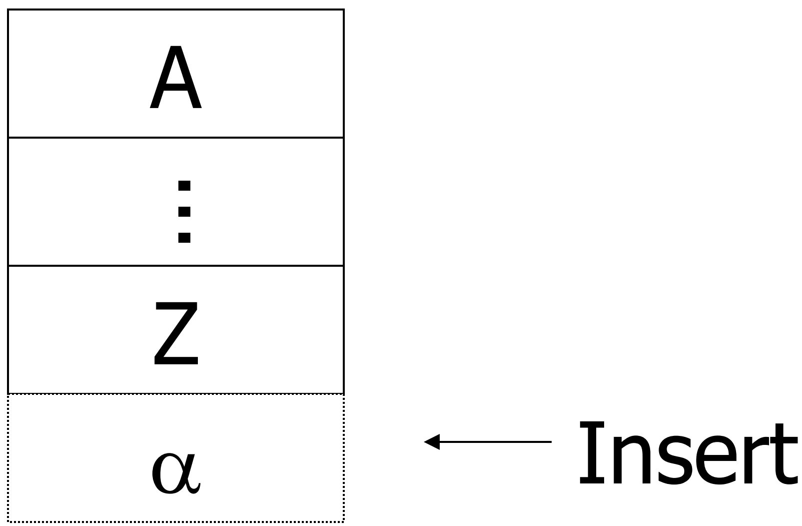


Exercise:

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



Insert + delete operations



Modifications to locking rules:

(1) Get exclusive lock on A before deleting A

(2) At insert A operation by Ti, Ti is given exclusive lock on A

Still have a problem: **Phantoms**

Example: relation R (E#,name,…)
constraint: E# is key
use tuple locking

R

	E#	Name	….
o1	55	Smith	
o2	75	Jones	

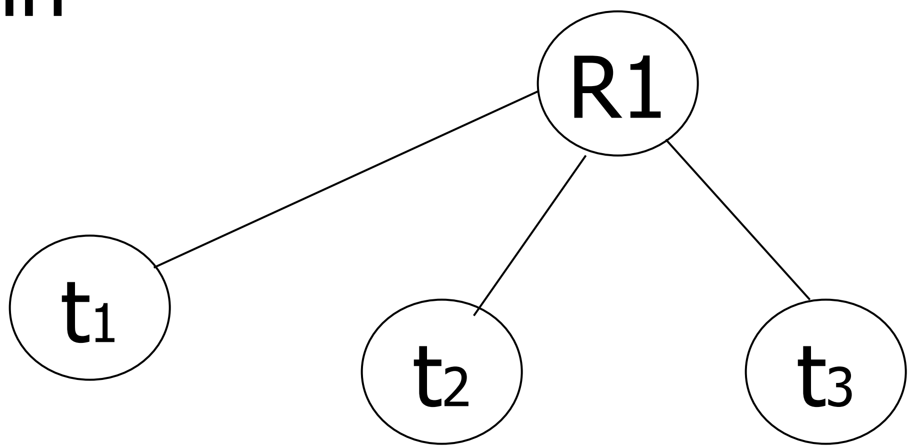
T_1: Insert <08,Obama,…> into R

T_2: Insert <08,McCain,…> into R

T_1	T_2
$S_1(o_1)$	$S_2(o_1)$
$S_1(o_2)$	$S_2(o_2)$
Check Constraint	Check Constraint
⋮	⋮
Insert o_3[08,Obama,..]	
	Insert o_4[08,McCain,..]

Solution

- Use multiple granularity tree
- Before insert of node Q,
  lock parent(Q) in
  X mode

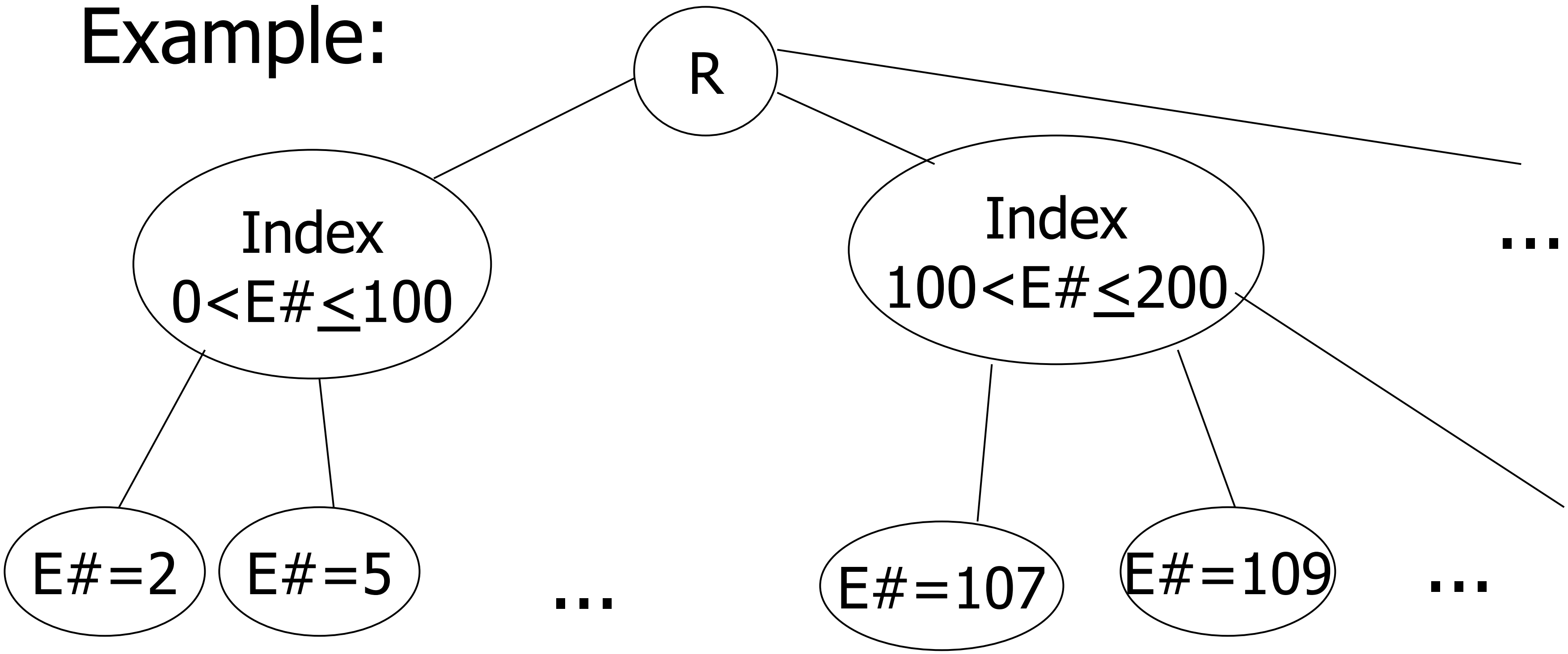


Back to example

T_1: Insert<08,Obama>	T_2: Insert<08,McCain>
T_1	T_2
$X_1(R)$	
	$X_2(R)$ *delayed*
Check constraint	
Insert<08,Obama>	
$U(R)$	
	$X_2(R)$
	Check constraint
	Oops! e# = 08 already in R!

Instead of using R, can use index on R:

Example:



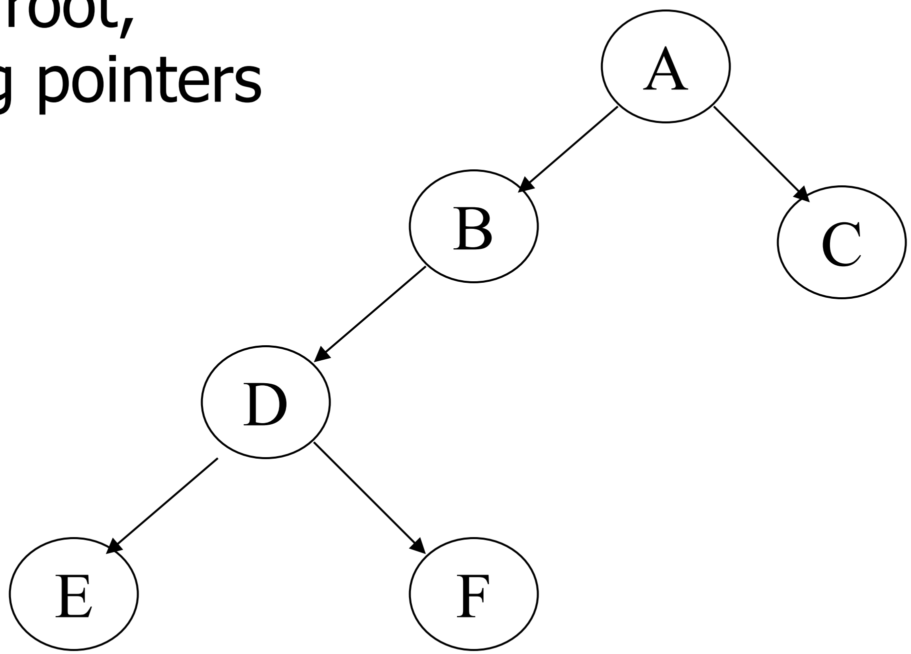
- This approach can be generalized to multiple indexes...

Next:

- Tree-based concurrency control
- Validation concurrency control

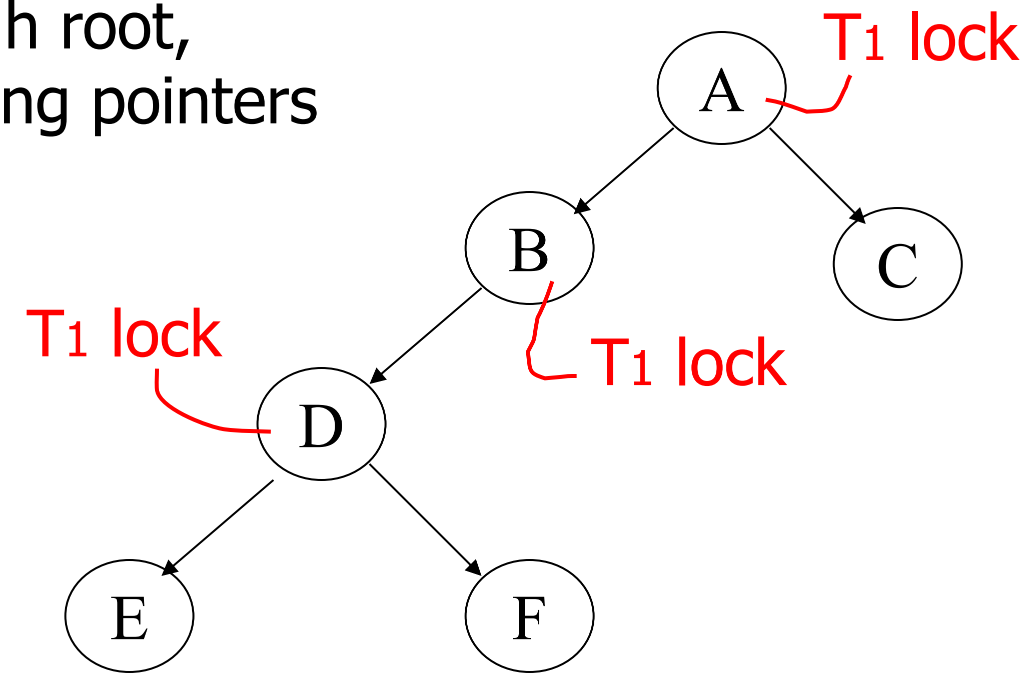
Example

- all objects accessed through root, following pointers



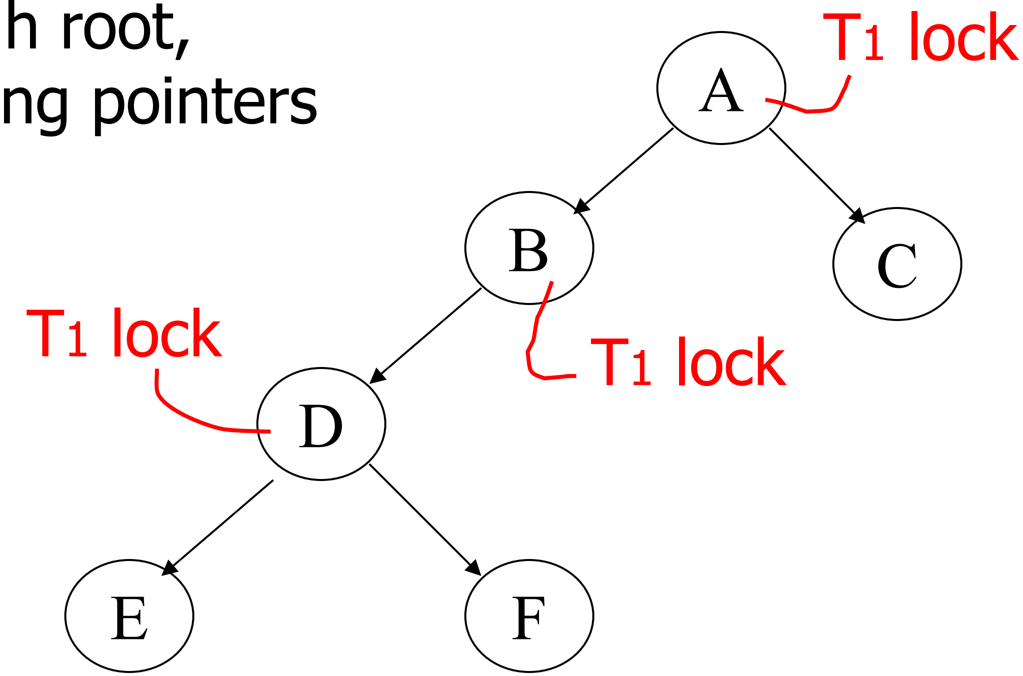
Example

- all objects accessed through root, following pointers



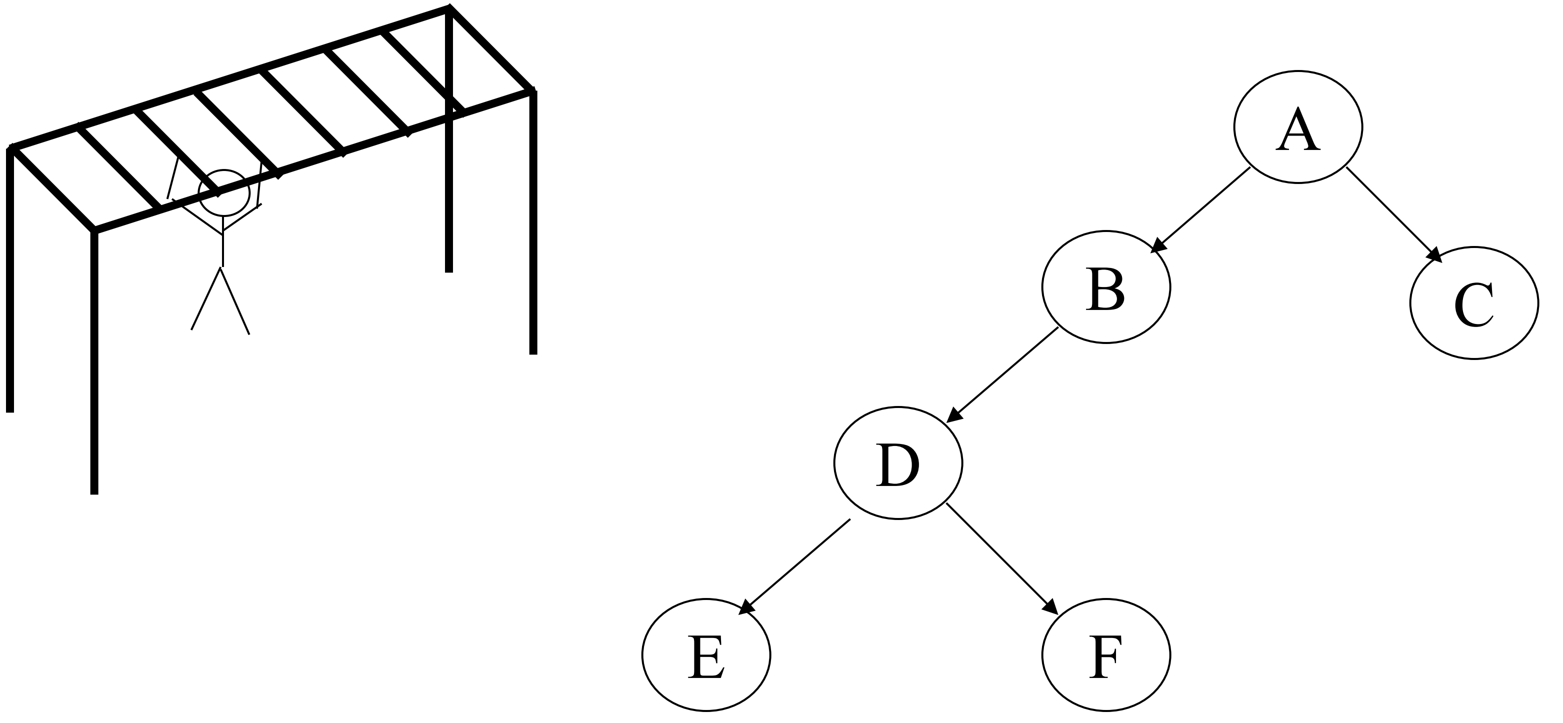
Example

- all objects accessed through root, following pointers

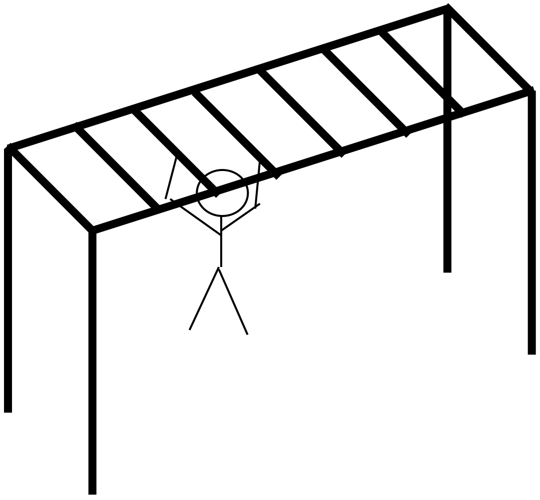


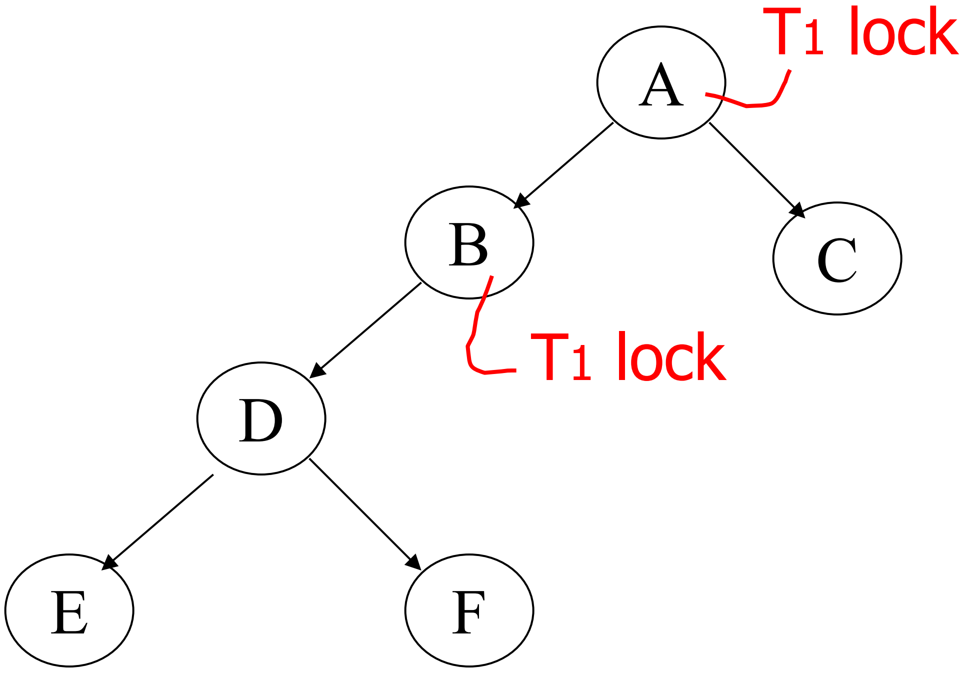
☛ can we release A lock if we no longer need A??

Idea: traverse like "Monkey Bars"

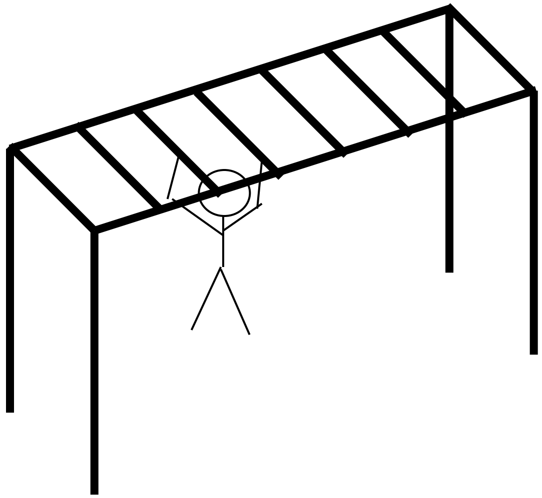


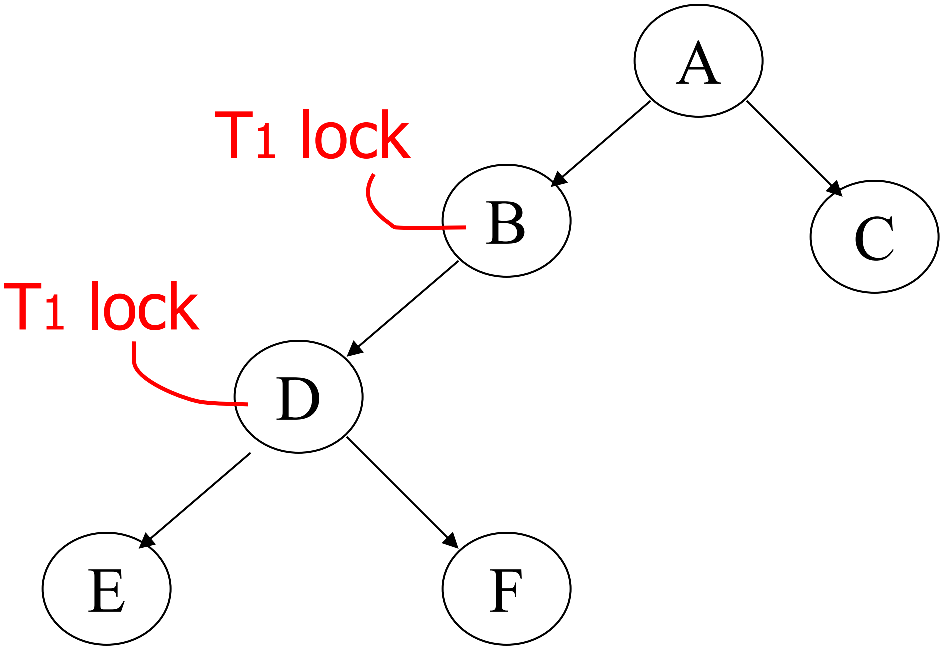
Idea: traverse like “Monkey Bars”





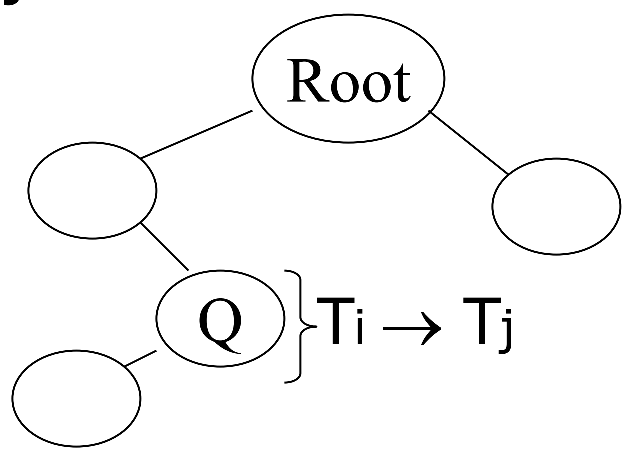
Idea: traverse like "Monkey Bars"





Why does this work?

- Assume all T_i start at root; exclusive lock
- $T_i \rightarrow T_j \;\Rightarrow\; T_i$ locks root before T_j



- Actually works if we don’t always start at root

Rules: tree protocol (exclusive locks)

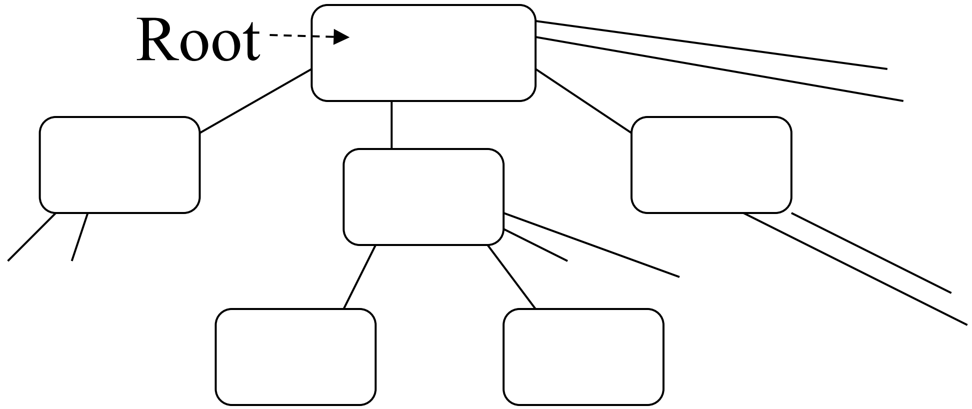
(1) First lock by T_i may be on any item

(2) After that, item Q can be locked by T_i only if parent(Q) locked by T_i

(3) Items may be unlocked at any time

(4) After T_i unlocks Q, it cannot relock Q

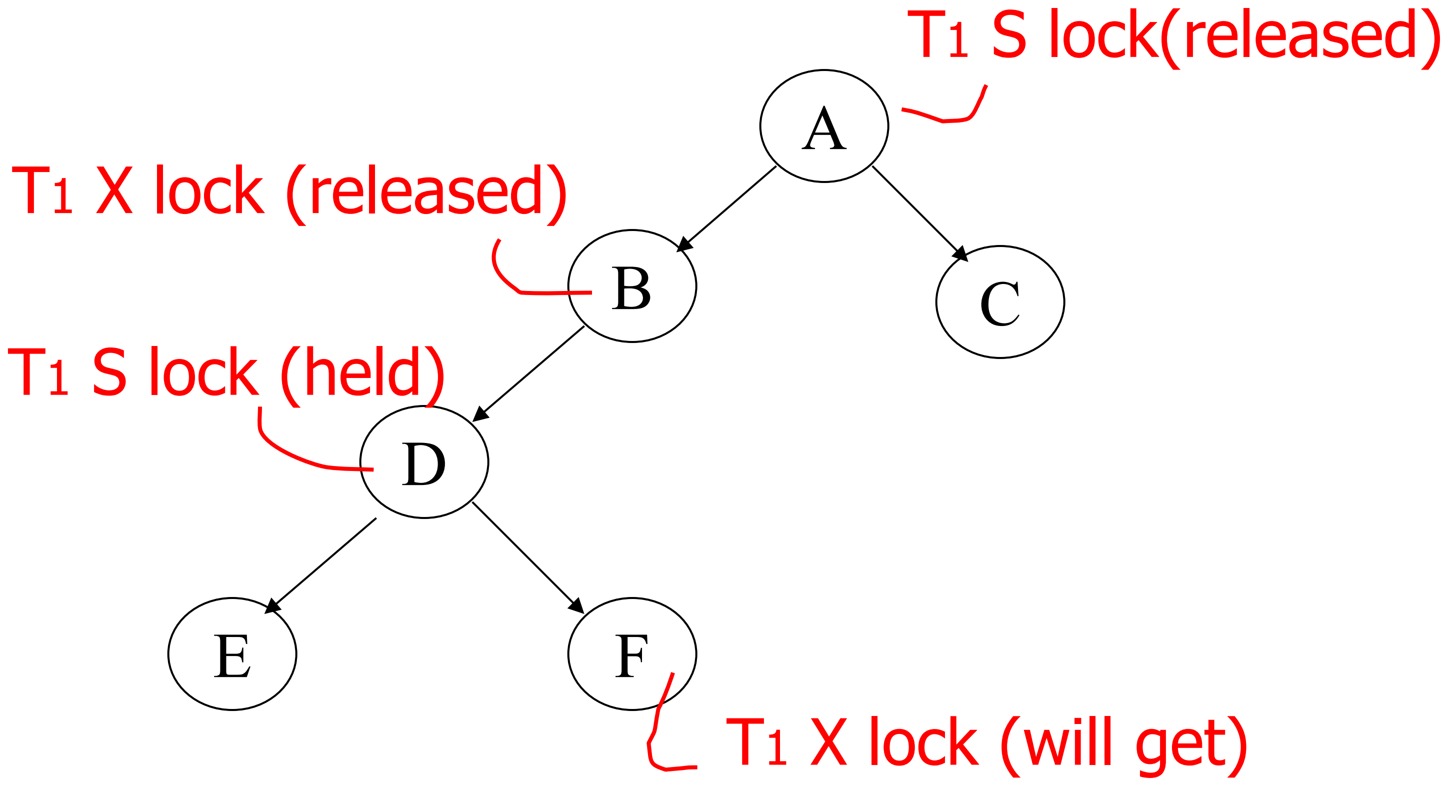
- Tree-like protocols are used typically for B-tree concurrency control



E.g., during insert, do not release parent lock, until you are certain child does not have to split

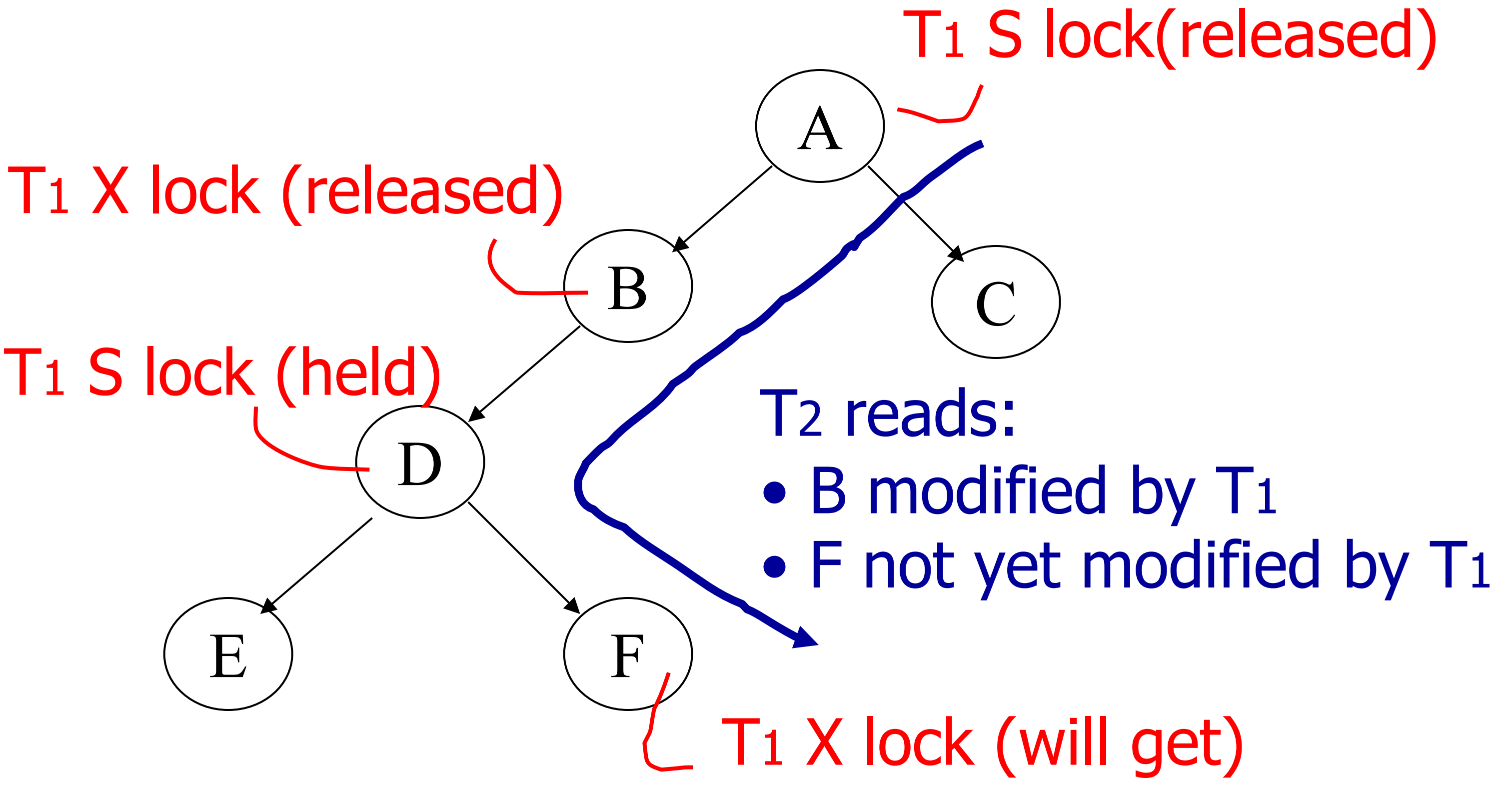
Tree Protocol with Shared Locks

- Rules for shared & exclusive locks?



Tree Protocol with Shared Locks

- Rules for shared & exclusive locks?



Tree Protocol with Shared Locks

- Need more restrictive protocol
- Will this work??
  - Once T_1 locks one object in X mode, all further locks down the tree must be in X mode

Validation

Transactions have 3 phases:

(1) Read

- all DB values read
- writes to temporary storage
- no locking

(2) Validate

- check if schedule so far is serializable

(3) Write

- if validate ok, write to DB

Key idea

- Make validation atomic
- If $T_1, T_2, T_3, \ldots$ is validation order, then resulting schedule will be conflict equivalent to $S_s = T_1\, T_2\, T_3 \ldots$

To implement validation, system keeps two sets:

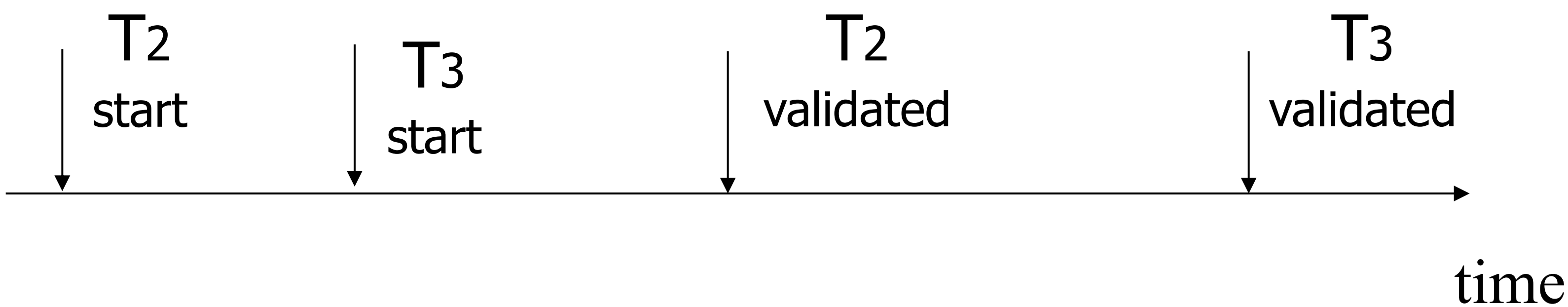
- FIN = transactions that have finished phase 3 (and are all done)
- VAL = transactions that have successfully finished phase 2 (validation)

Example of what validation must prevent:

$RS(T_2)=\{B\}$ $\qquad$ $RS(T_3)=\{A,B\}$

$WS(T_2)=\{B,D\}$ $\qquad$ $WS(T_3)=\{C\}$

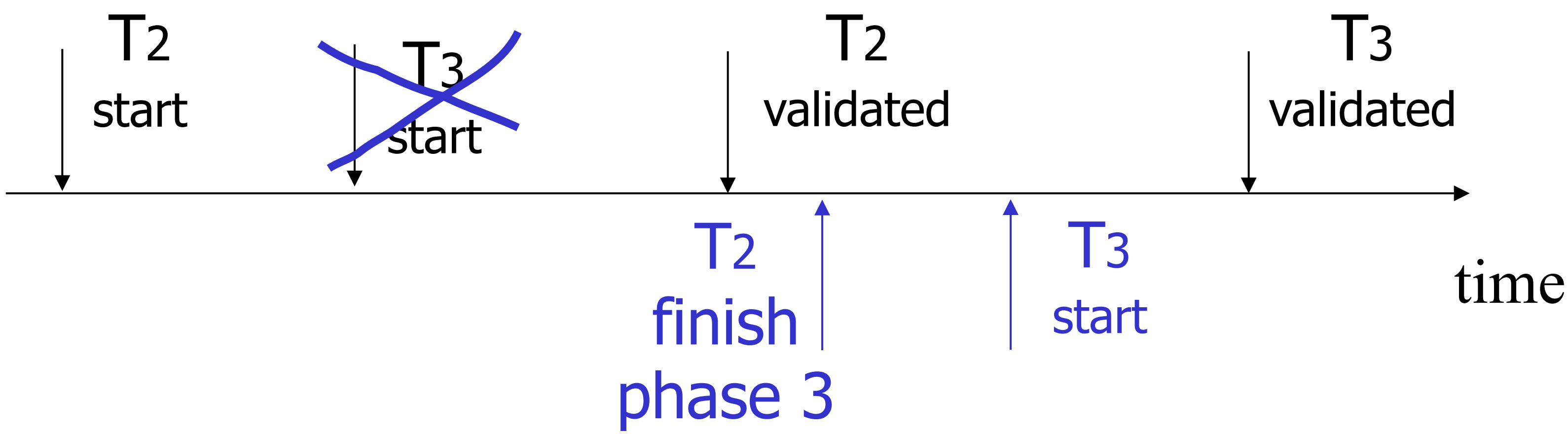
$WS(T_2) \cap RS(T_3) \neq \phi$



Example of what validation must ~~prevent~~ allow:

$RS(T_2)=\{B\}$ $\cap$ $RS(T_3)=\{A,B\} \neq \phi$

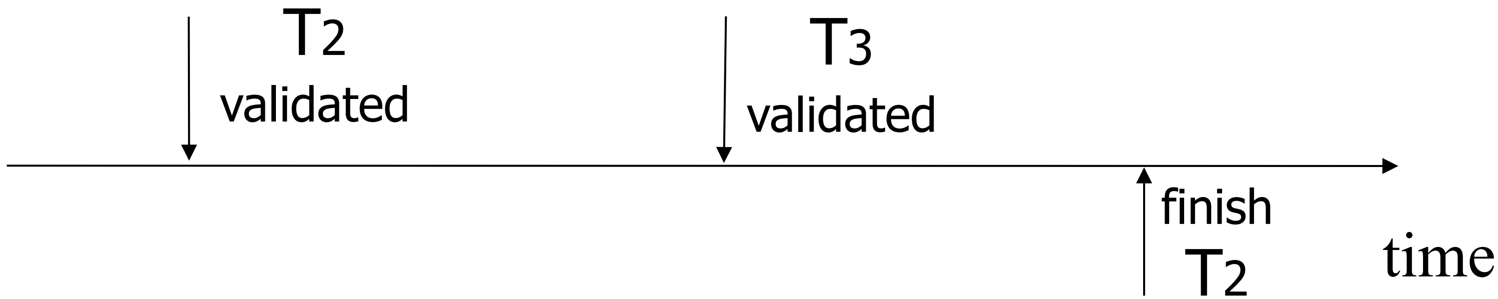
$WS(T_2)=\{B,D\}$ $WS(T_3)=\{C\}$



Another thing validation must prevent:

$RS(T_2)=\{A\}$ $RS(T_3)=\{A,B\}$

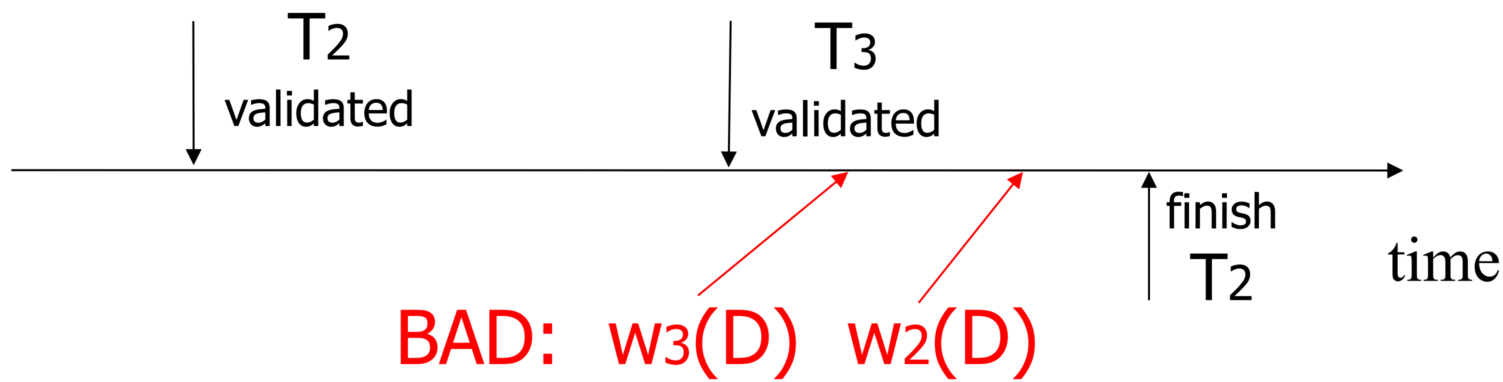
$WS(T_2)=\{D,E\}$ $WS(T_3)=\{C,D\}$



Another thing validation must prevent:

$RS(T_2)=\{A\}$ $\qquad$ $RS(T_3)=\{A,B\}$

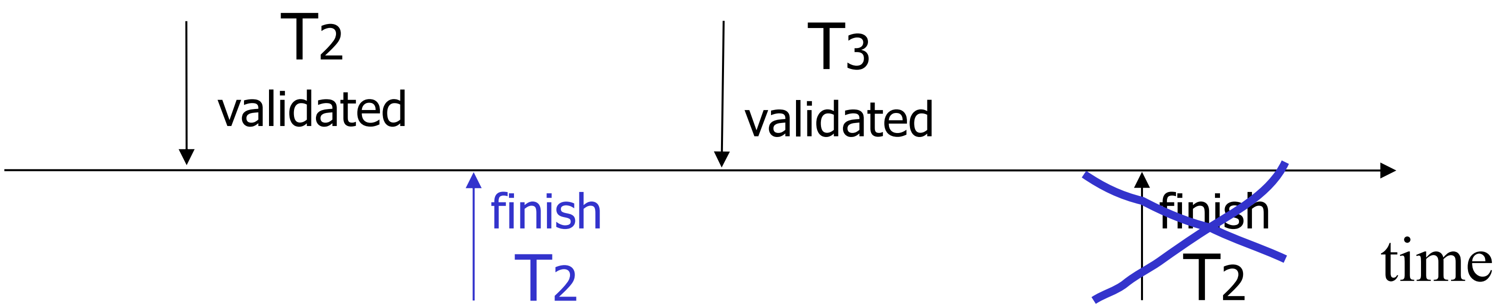
$WS(T_2)=\{D,E\}$ $\qquad$ $WS(T_3)=\{C,D\}$



Another thing validation must ~~prevent:~~ allow

$RS(T_2)=\{A\}$ $RS(T_3)=\{A,B\}$

$WS(T_2)=\{D,E\}$ $WS(T_3)=\{C,D\}$



Validation rules for T_j:

(1) When T_j starts phase 1:

ignore(T_j) $\leftarrow$ FIN

(2) at T_j Validation:

if check (T_j) then

[VAL $\leftarrow$ VAL U {T_j};

do write phase;

FIN $\leftarrow$ FIN U {T_j}]

Check (T_j):

For $T_i \in$ VAL - IGNORE (T_j) DO

IF [WS(T_i) $\cap$ RS(T_j) $\neq \emptyset$ OR

$T_i \notin$ FIN] THEN RETURN false;

RETURN true;

Check (T_j):

For $T_i \in$ VAL - IGNORE (T_j) DO

IF [$WS(T_i) \cap RS(T_j) \neq \emptyset$ OR

$T_i \notin$ FIN] THEN RETURN false;

RETURN true;

Is this check too restrictive ?

Improving Check(T_j)

For $T_i \in$ VAL - IGNORE (T_j) DO

IF [$WS(T_i) \cap RS(T_j) \neq \emptyset$ OR

$(T_i \notin$ FIN AND $WS(T_i) \cap WS(T_j) \neq \emptyset)]$

THEN RETURN false;

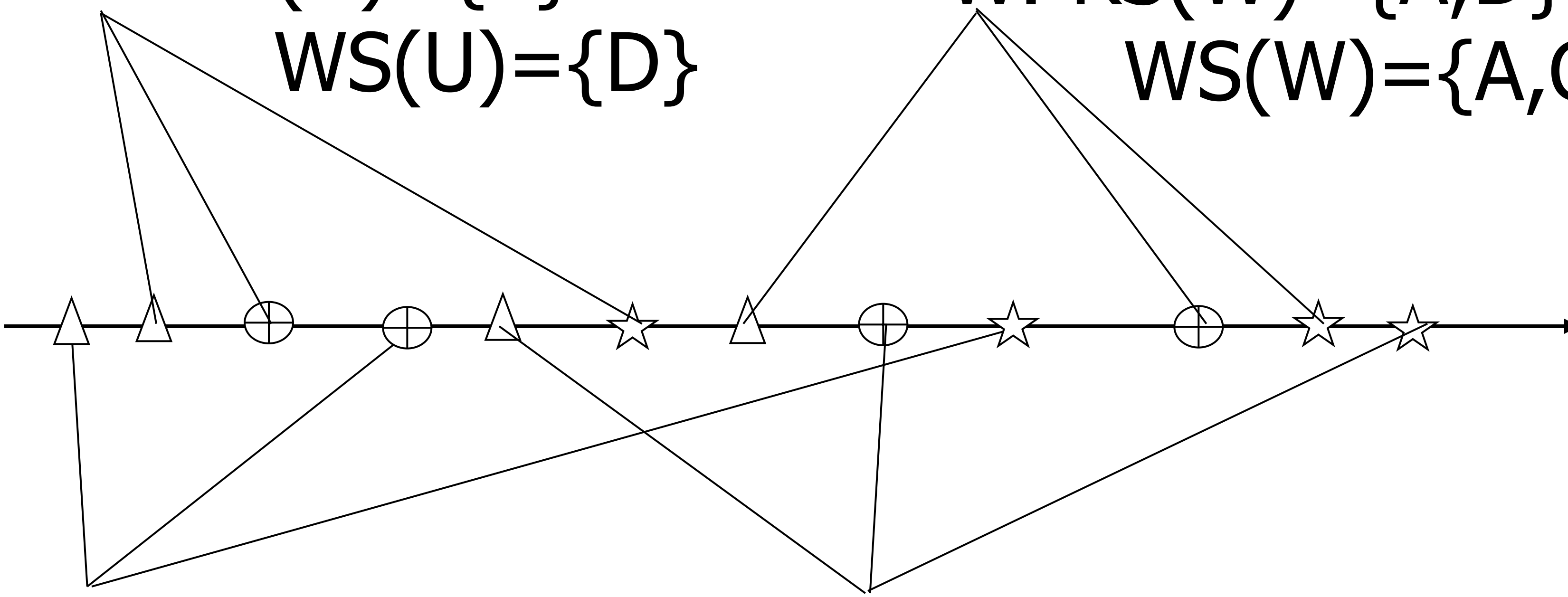
RETURN true;

Exercise:

start
validate
finish

U: RS(U)={B}
WS(U)={D}

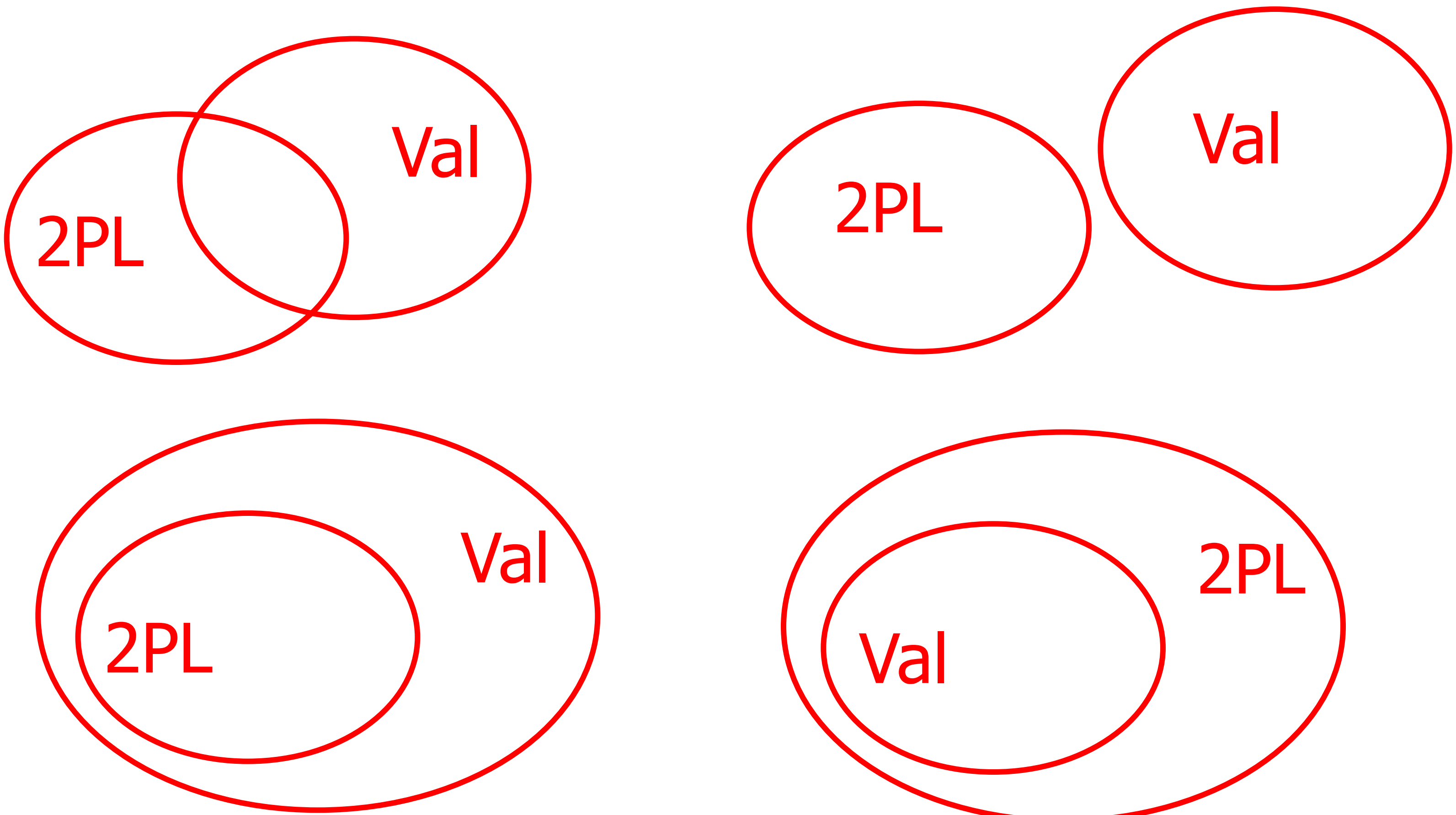
W: RS(W)={A,D}
WS(W)={A,C}



T: RS(T)={A,B}
WS(T)={A,C}

V: RS(V)={B}
WS(V)={D,E}

Is Validation = 2PL?



S2: w2(y) w1(x) w2(x)

- Achievable with 2PL?
- Achievable with validation?

S2: w2(y) w1(x) w2(x)

- S2 can be achieved with 2PL:
  l2(y) w2(y) l1(x) w1(x) u1(x) l2(x) w2(x) u2(y) u2(x)
- S2 cannot be achieved by validation:
  The validation point of T2, val2 must occur before w2(y) since transactions do not write to the database until after validation. Because of the conflict on x, val1 < val2, so we must have something like

  S2: val1 val2 w2(y) w1(x) w2(x)

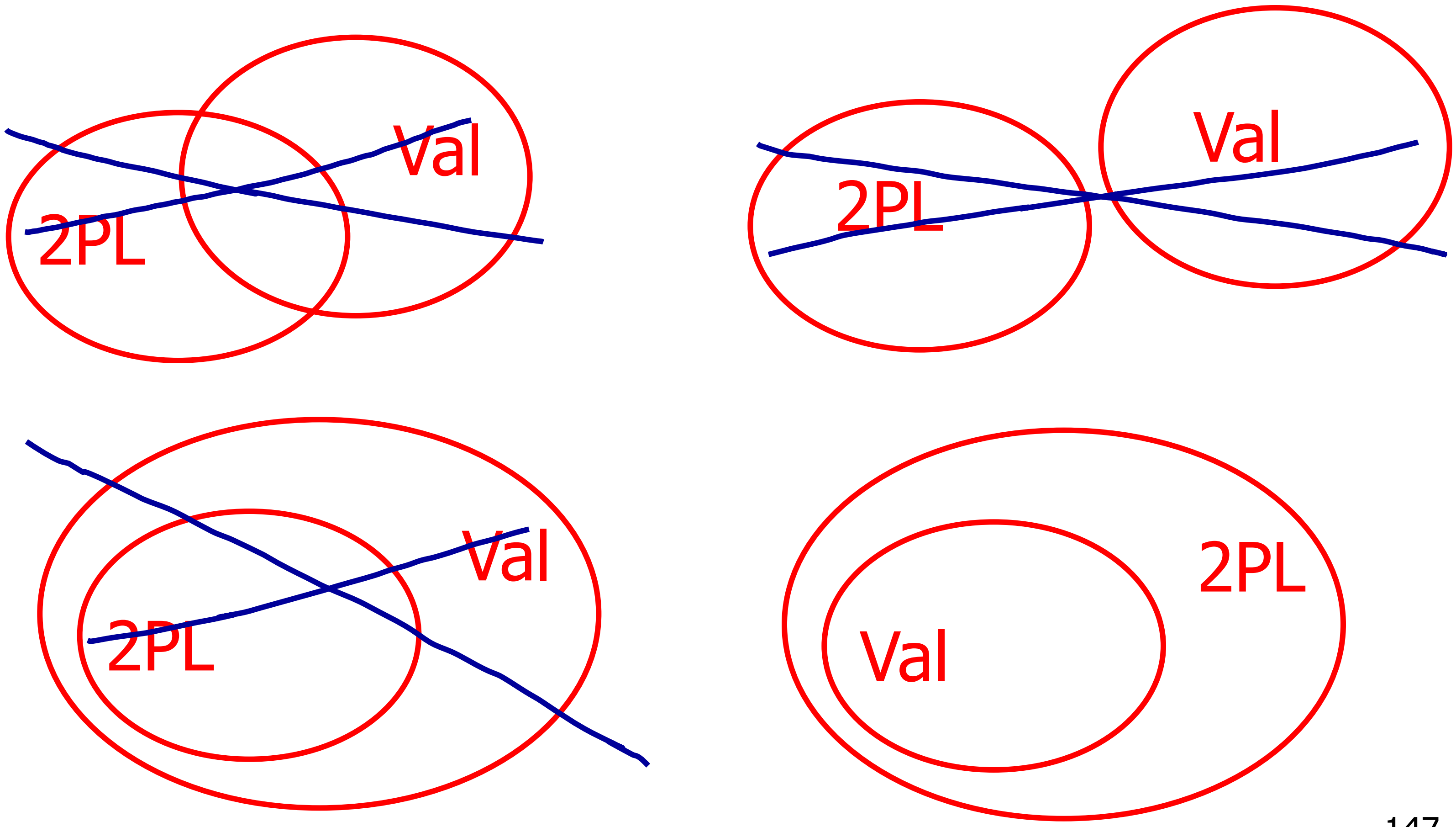
  With the validation protocol, the writes of T2 should not start until T1 is all done with its writes, which is not the case.

Validation subset of 2PL?

- Possible proof (Check!):
  - Let S be validation schedule
  - For each T in S insert lock/unlocks, get S':
    - At T start: request read locks for all of RS(T)
    - At T validation: request write locks for WS(T); release read locks for read-only objects
    - At T end: release all write locks
  - Clearly transactions well-formed and 2PL
  - Must show S' is legal (next page)

- Say S′ not legal (due to w-r conflict):
  S′: ... l1(x)   w2(x) r1(x)   val1 u1(x) ...
  - At val1: T2 not in Ignore(T1); T2 in VAL
  - T1 does not validate: WS(T2) $\cap$ RS(T1) $\neq \emptyset$
  - contradiction!
- Say S′ not legal (due to w-w conflict):
  S′: ... val1 l1(x)   w2(x) w1(x)   u1(x) ...
  - Say T2 validates first (proof similar if T1 validates first)
  - At val1: T2 not in Ignore(T1); T2 in VAL
  - T1 does not validate:
    T2 $\notin$ FIN AND WS(T1) $\cap$ WS(T2) $\neq \emptyset$)
  - contradiction!

Conclusion:
Validation subset 2PL



Validation (also called optimistic concurrency control) is useful in some cases:

- Conflicts rare
- System resources plentiful
- Have real time constraints

Summary

Have studied C.C. mechanisms used in practice

- 2 PL
- Multiple granularity
- Tree (index) protocols
- Validation