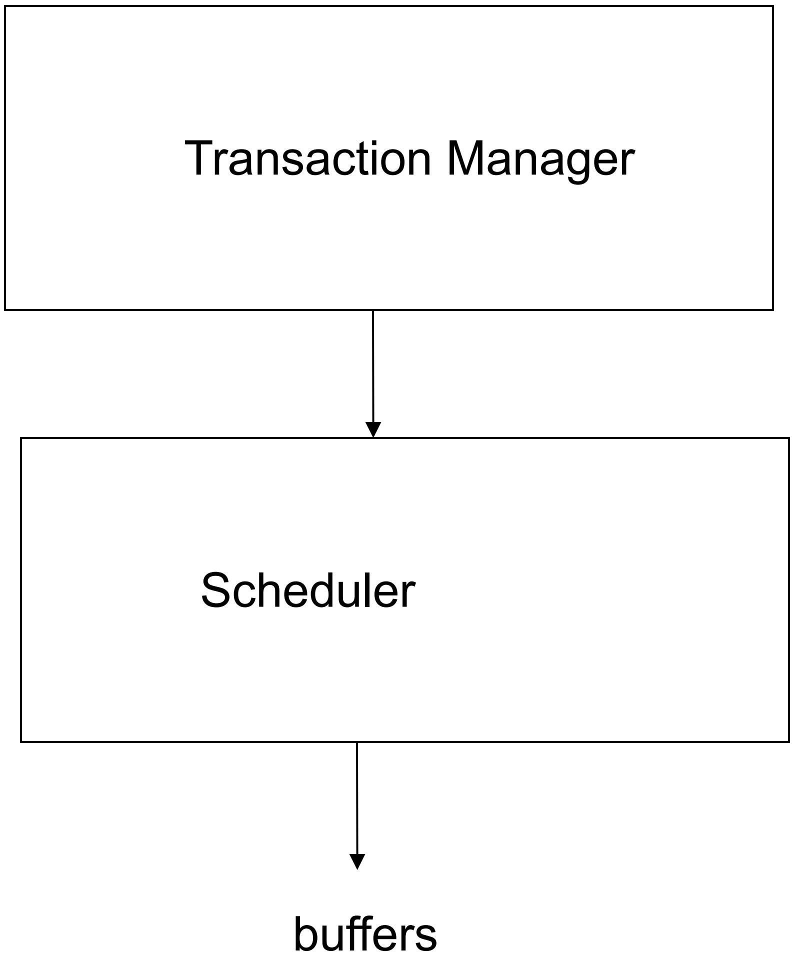
Concurrency Control

Introduction

- Interactions among the transactions can cause the database state inconsistent.
- Each transaction
  - Individually ensures consistent state with no system failure.
- When several transactions are processed in parallel, inconsistency may occur.
  - Needs regulation
    - Scheduler controls the accesses
    - Scheduler is a protocol
      - Question: how to design a protocol ?

Scheduler



- Transaction passes reads and write requests to scheduler
- Scheduler executes directly, if the elements are present in the buffer.
  - Otherwise, data is brought to memory.
- Scheduler may delay or abort the transaction.
  - What is the criteria ?
    - Read Uncommitted, Read committed, Repeatable read, Serializability.
  - What is the protocol ?
    - No locking, Locking, optimistic, timestamp, and so on

Dirty read, non-repeatable read and phantom problems

- Dirty read problem: No locks: dirty read problem occurs
  - A dirty read occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed.
- Non repeatable read problem: No read locks.
  - A *non-repeatable read* occurs, when during the course of a transaction, a row is retrieved twice and the values within the row differ between reads.
- Phantom problem: read locks are released as soon as possible
  - Execution two similar queries give different results.

Isolation levels and read phenomena

Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	X	X	X
Read Committed	-	X	X
Repeatable Read	-	-	X
Serializable	-	-	-

Note: "X" means that the isolation level suffers that phenomenon, while "-" means that it does not suffer it.

Refer: http://en.wikipedia.org/wiki/Isolation_(database_systems)

Outline

- Serial and serializable schedules
- Conflict serializability
- Enforcing serializability by locks
- Locking system with several lock modes
- An architecture of a locking scheduler
- Managing hierarchies of database elements
- The tree protocol
- Concurrency control by timestamps
- Concurrency control by validation

Serial and Serializable Schedules.

- Correctness principle.
  - If executed alone, it starts in a consistent state and leaves the database in a consistent state, when the transaction ends.
- What is the correctness principle if several transactions are executed concurrently ?
- What are the protocols which are to be followed by scheduler to ensure the correctness ?

Schedules (or History)

- A schedule is a time-ordered sequence of the important actions taken by one or more transactions.
  - We consider read and write operations that occur in the buffer.
  - Ignore INPUT and OUTPUT operations

Serial Schedules

- A schedule is serial, if its actions consist of all the actions of one transaction, another transaction and so on.
  - A schedule S is serial if any two transactions T, and T', if any action of T precedes any action of T', then all the actions of T precede all actions of T'.

Example:

T1:	Read(A)	T2:	Read(A)
	$A \leftarrow A+100$		$A \leftarrow A\times 2$
	Write(A)		Write(A)
	Read(B)		Read(B)
	$B \leftarrow B+100$		$B \leftarrow B\times 2$
	Write(B)		Write(B)

Constraint: $A=B$

Schedule A

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

Schedule A

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

Serial Schedule in which T1 precedes T2

Schedule B

T1	T2
	Read(A);A ← A×2;
	Write(A);
	Read(B);B ← B×2;
	Write(B);
Read(A); A ← A+100	
Write(A);	
Read(B); B ← B+100;	
Write(B);	

Another serial schedule: T2 precedes T1

Schedule B

T1	T2	A	B
		25	25
	Read(A);A ← A×2;		
	Write(A);	50	
	Read(B);B ← B×2;		
	Write(B);	50	
Read(A); A ← A+100			
Write(A);		150	
Read(B); B ← B+100;			
Write(B);		150	
		150	150

Serializable Schedules

- Correctness principle
  - Every serial schedule ensures the correctness.
- Are there other schedules which preserve the consistency ?
  - These are called serializable schedules which are equivalent to a serial schedule.

Schedule C

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

Schedule C

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

Serializable, but not serial

Schedule D

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

Not serial not serializable

Schedule D

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

The effect of transaction semantics

- The details of transactions matter (see the next example).
  - But it is very difficult to analyze the semantics.

Schedule E

Same as Schedule D
but with new T2′

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

A schedule is serializable due to behavior of transactions

Schedule E

Same as Schedule D
but with new T2′

T1	T2’	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×1;		
	Write(A);	125	
	Read(B);B ← B×1;		
	Write(B);	25	
Read(B); B ← B+100;			
Write(B);		125	
		125	125

Notations

- $r_T(X)$: transaction T reads database element X
- $w_T(X)$: transaction T writes database element X
- $r_i(X)$ is same as $r_{Ti}(X)$:
- $w_i(X)$ is same as $w_{Ti}(X)$:
- Example

  $T1=r_1(A); w_1(A); r_1(B); w_1(B);$

  $T2=r_2(A); w_2(A); r_2(B); w_2(B);$

- Want schedules , regardless of
  - initial state and
  - transaction semantics
- Only look at order of read and writes

Example: Consider serializable schedule

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

Notation for Transactions and Schedules

- An action is ri(X) or wi(X)
- Ti is a sequence of actions with subscript "i".
- A schedule S of a set of transactions T is a sequence of actions, in which for each T_i in T, the actions of T_i appear in S in the **same order** as in the definition of "T_i" itself.
- Example:

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

Example:

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

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

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., $T_2 \rightarrow T_1$

Conflict Serializability

- Scheduler ensures that schedules are serializable by ensuring a condition conflict-serializability.
- It is based on the idea of conflict
- Conflict: Let Ti and Tj are transactions
  - ri(X); rj(Y) is never conflict even X=Y as they do not change any value
  - ri(X); wj(Y) is not a conflict provided that X≠Y.
  - wi(X); rj(Y) is not a conflict if X≠Y.
  - Wi(X); wj(Y) is not a conflict if X≠Y

- The actions of the same transaction conflict.; ri(X); wi(Y) conflict; order is fixed by a transaction. DBMS can not reorder!
- Two writes of different transaction on the same element conflict: wi(X), wj(X) conflict.
- Read and write of the same database element by different transactions also conflict. ri(X); wj(X) conflict. Also, wi(X) and ri(X) conflict.

Conflict Serializability

- Two actions of different transactions may be swapped unless
  - They involve the same database element, and
  - at least one of them is write.
- Carry out the non conflicting swaps and try to convert the schedule into a serial schedule.
- Conflict-equivalent
  - Two schedules are conflict equivalent if they can be turned one into other by a sequence of non-conflicting swaps of **ADJACENT ACTIONS.**
- Conflict serializable
  - If a schedule is conflict equivalent to a serial schedule.
- Conflict serializable schedule is a serializable schedule.
- Note:
  - Conflict serializability is not required for a schedule to be serializable.
  - But many commercial system use serializability criteria to guarantee serializabilty

Precedence Graphs and a test for conflict serializability

- Nodes: transactions in S

Arcs: $T_i \rightarrow T_j$ whenever

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

- Insert the edge
  - $r_i(X)$, $w_j(X)$
  - $w_j(X)$, $r_i(X)$
  - $w_i(X)$, $w_j(X)$
  - $w_j(X)$, $w_i(X)$

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
  $S = w_1(A)\ r_2(A)\ r_3(A)\ w_4(A)$?

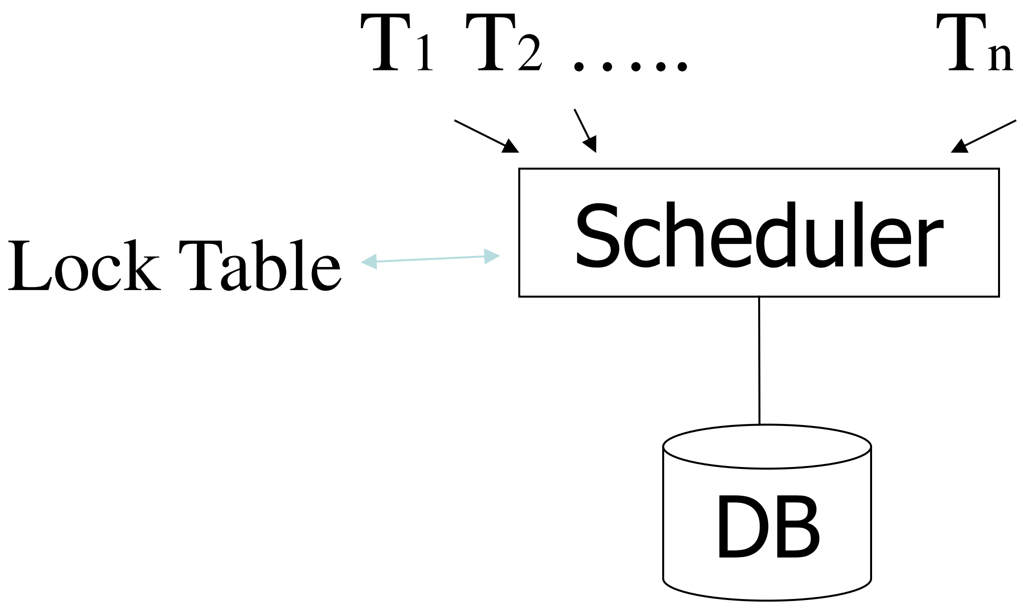
Enforcing serializability by locks

- 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



Serializability with Locks

- Arbitrary execution may result into non-serializable schedule.
- Job of scheduler is to prevent orders of actions that lead to an non-serializable schedule.
- We first discuss the lock-based scheduler with single lock
  - Lock must be obtained for a database element before accessing it.

Proper use of locks: Legal schedule

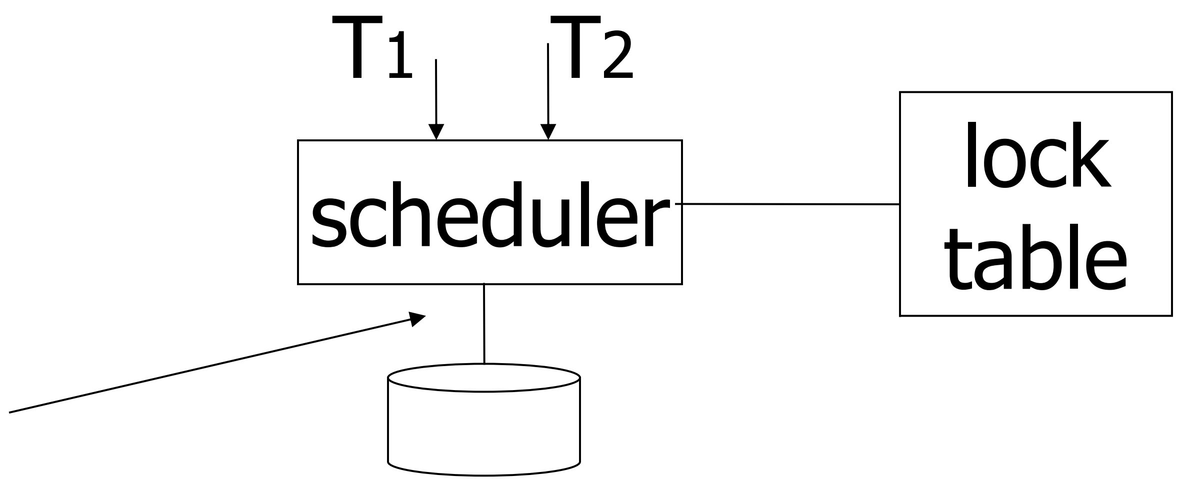
- Consistency of transactions
  - Transaction can only read/write an element if it has previously has requested the lock.
  - If transaction locks an element, it must unlock that element.
- No two transactions must not lock the given object simultaneously. Lock can be given after the first transaction unlocking it.

A locking protocol (with one kind of lock)

Two new actions:

lock (exclusive): l_i (A) Ti requests a lock

unlock: u_i (A) Ti releases the lock



Serializable Schedule of actions

Consistency Condition for transactions

- If there are actions li(X) followed by lj(X) in a schedule, there should be ui(X) action between these actions.

Rule #1: Well-formed transactions: transaction should request a lock and release the lock

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

<u>Rule #2</u> Legal scheduler: No two transactions should not lock the element without the first unlocked it.

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

$\longleftrightarrow$

no $l_j(A)$

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?

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

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

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

Schedule F

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

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

Note: Schedule F is a legal schedule of consistent transactions, but it is not serializable.

Two-phase locking

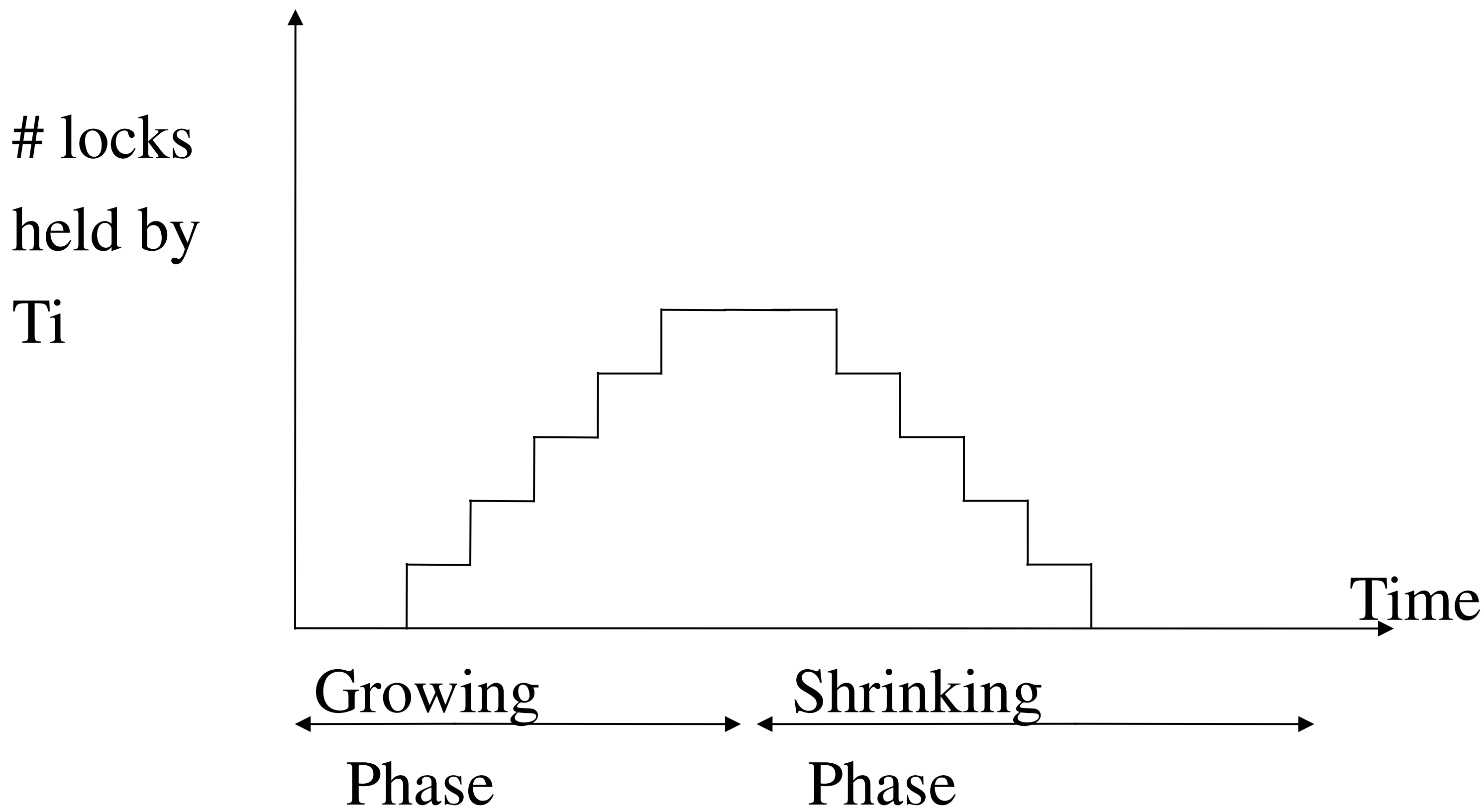
- Rule #3: In every transaction, all lock requests precede all unlock requests.

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); $l_2(B)$ delayed
Read(B);B $\leftarrow$ B+100	
Write(B); $u_1(B)$	

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

Deadlock !

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

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

Next step:

Show that rules #1,2,3 $\Rightarrow$ 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

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

Lemma

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

Proof of lemma:

$Ti \rightarrow Tj$ means that

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

By rules 1,2:

$S = \ldots p_i(A) \ldots u_i(A) \ldots l_j(A) \ldots q_j(A) \ldots$

Lemma

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

Proof of lemma:

$Ti \rightarrow Tj$ means that

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

By rules 1,2:

$S = \ldots\ p_i(A)\ \ldots\ u_i(A)\ \ldots\ l_j(A)\ \ldots\ q_j(A)\ \ldots$

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

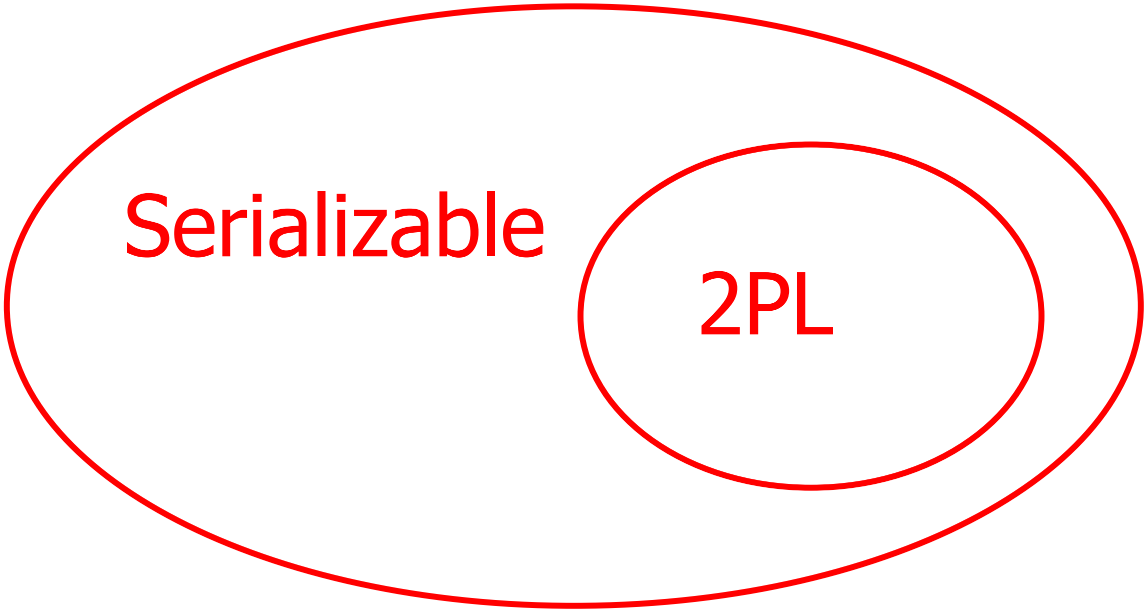
$$T_1 \rightarrow T_2 \rightarrow \ldots T_n \rightarrow T_1$$

(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

2PL subset of Serializable



Locking Systems with Several Lock Modes

- In one lock scheme even “reading” action also requires a lock.
  - Several transactions can read X in parallel.
- We can use two kinds of locks
  - Shared lock or read lock
  - Exclusive lock or write lock
- sli(X): requests a shared lock on X
- xli(X): requests an exclusive lock on X
- ui(X): relinquishes the locks.

Requirements

- 1. For writing, exclusive lock is required and for reading any lock is OK.
  - ri(X) must be preceded by sli(X) or xli(X) with no intervening ui(X).
  - wi(X) must be preceded by xli(X) with no intervening ui(X).
- 2: Two phase rule: locking must precede unlocking
  - No action sli(X) or xli(X) can be preceded by an action ui(X)
- 3. An object can be locked by several on a shared mode or exclusively by one transaction.
  - If xli(X) appears in a schedule, there will be no xlj(X) or slj(X) for some j other than i, without an intervening ui(X).
  - If sli(X) appears in a schedule, then there can not be following xlj(X) without intervening ui(X).

Compatible matrix

Lock Requested

Lock Held

	S	X
S	true	false
X	false	false

Example

T1	T2
sl1(A);r1(A)	
	sl2(A);r2(A)
	sl2(B); r2(B)
sl1(B); r1(B);	
xl1(B) denied	
	u2(A);u2(B)
xl1(B); w1(B);	
u1(A); u2(B)	

Note: If T1 would have requested exclusive lock initially, it would have been rejected.

Upgrading Locks

- If Ti has a shared lock on X can upgrade to exclusive lock.

Upgrading locks: deadlock

T1	T2
sl1(A)	
	sl2(A)
xl1(A) denied	
	xl2(A) denied

More concurrency Update locks

- To avoid deadlock problem
- Update lock can only read and not to write
- Only update lock will be upgraded to write lock later.
- We can grant update lock even though transactions have shared lock on X but, Once we have an update lock, other locks (shared, excusive, update) are denied.

Update Locks: Compatibility Matrix

	S	X	U
S	Yes	No	Yes
X	No	No	No
U	No	No	No

Upgrading locks: No deadlock Problem

T1	T2
ul1(A); r1(A);	
	ul2(A) denied
xl1(A), w1(A); u1(A)	
	ul2(A); r2(A);
	xl2(A); w2(A); u2(A);

Increment Locks

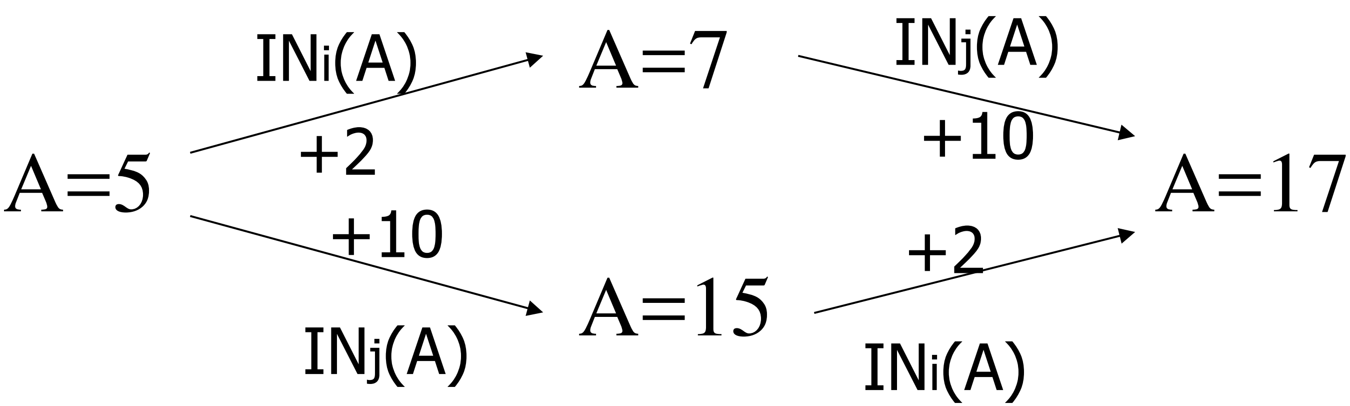
- For operations that commute each other; two transactions add constants to each other; it does not matter which goes first.

Example: 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!



Increment Locks: Compatibility Matrix

	S	X	I
S	Yes	No	No
X	No	No	No
I	No	No	Yes

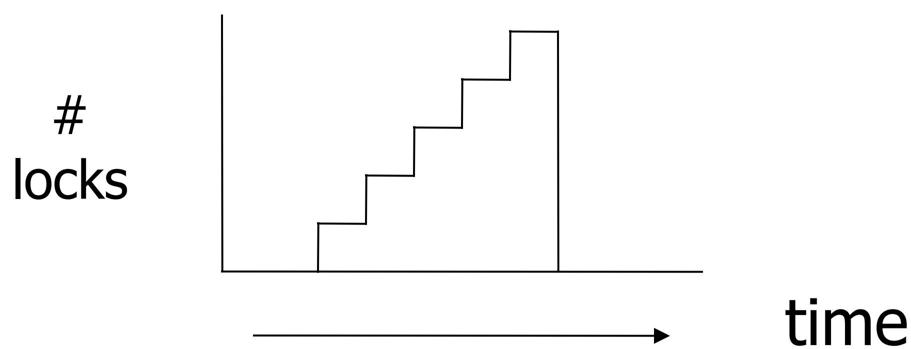
Increment locks:

T1	T2
sl1(A); r1(A);	
	sl2(A);r2(A);
	il2(B);inc2(B);
il1(B), inc1(B);	
	u2(A); u2(B);
u1(A); u1(B);	

Architecture of a Lock Scheduler

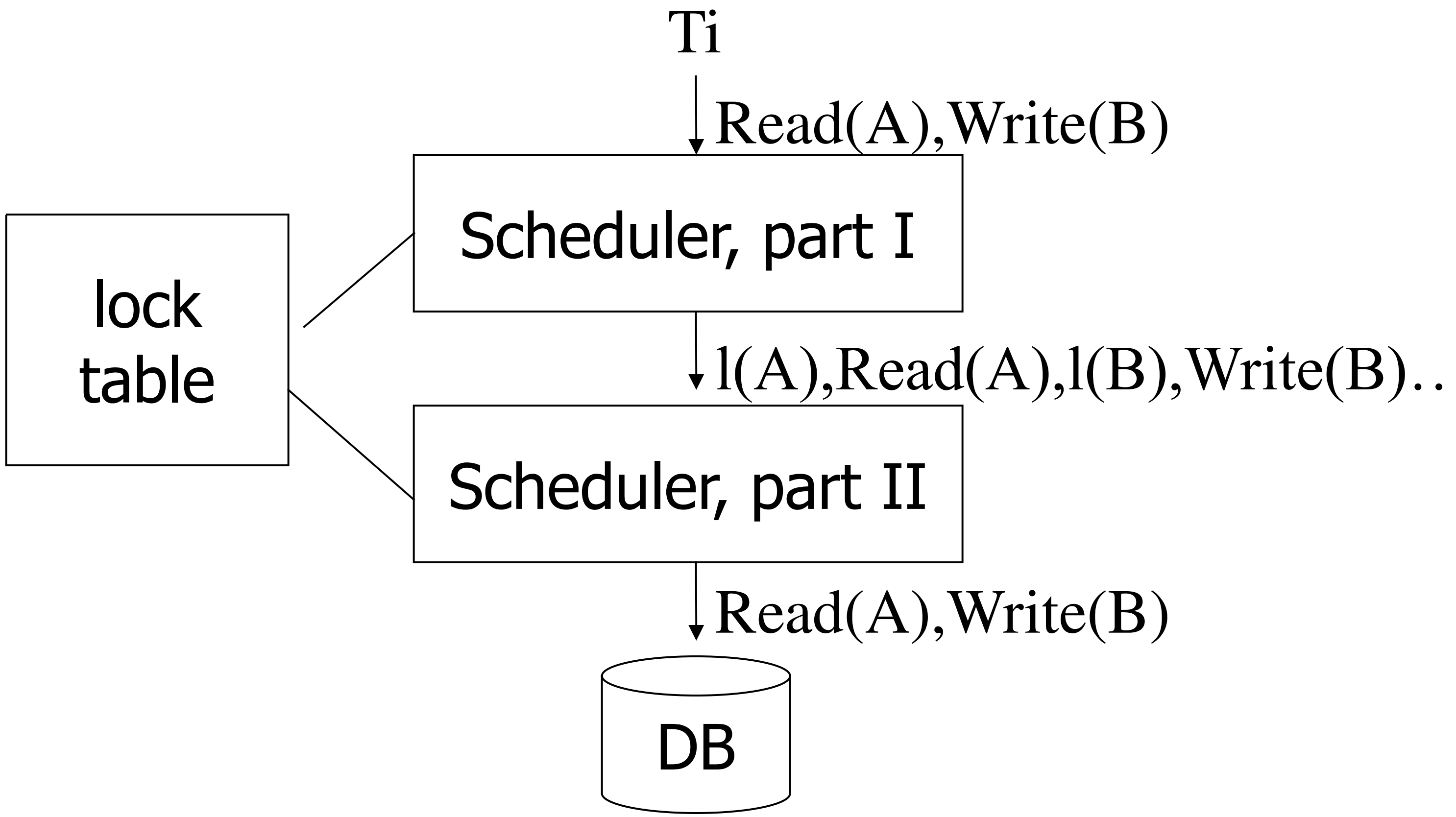
(1) Don’t trust transactions to request/release locks

(2) Hold all locks until transaction commits



Principles

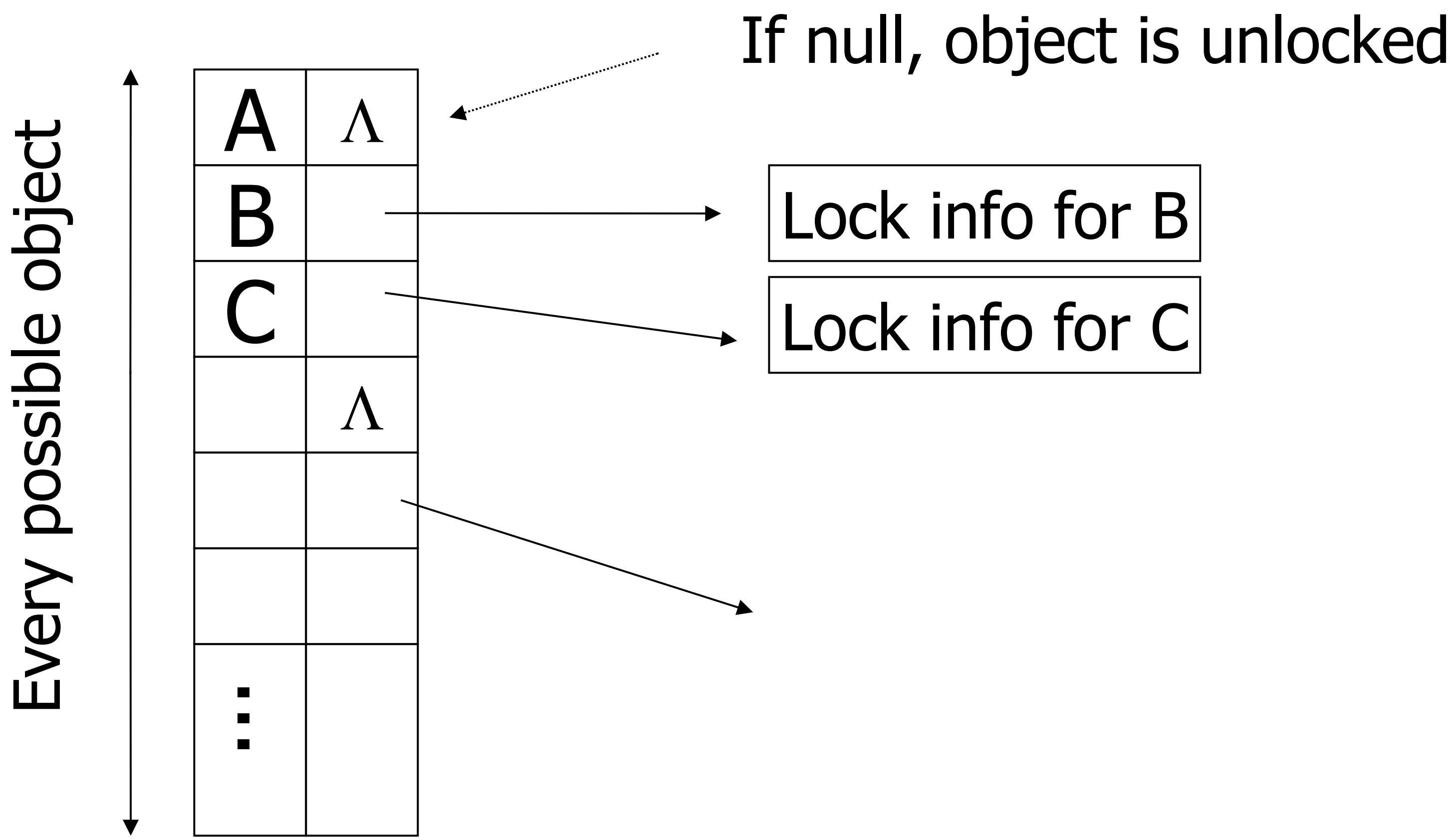
- Transactions do not request locks
  – Lock scheduler should insert the locks
- Transactions do not release the locks, the scheduler releases the locks based on commit or abort command.


Ti
Read(A),Write(B)
Scheduler, part I
lock
table
l(A),Read(A),l(B),Write(B)…
Scheduler, part II
Read(A),Write(B)
DB

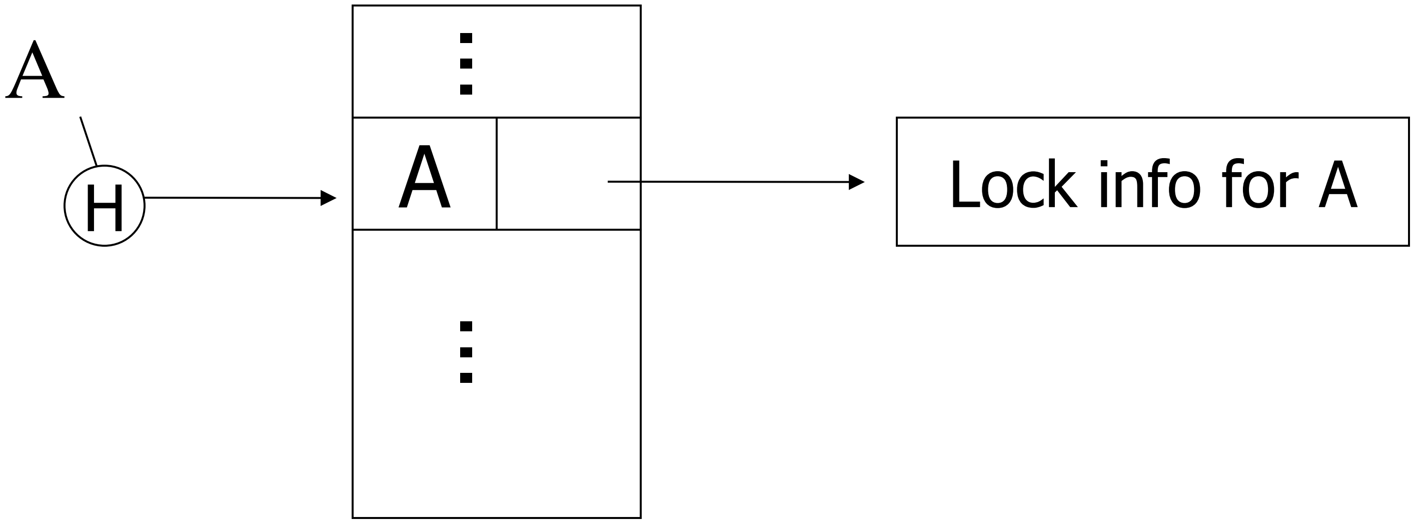
Scheduler

- Part I selects appropriate mode of lock requests
- Part II executes the operations.
- When part I receives commit/abort by transaction manager, it releases the lock held by T. When a transaction is waiting for a lock, part I notifies part II.
- Part II starts executing waiting transactions.

<u>Lock table</u>   Conceptually

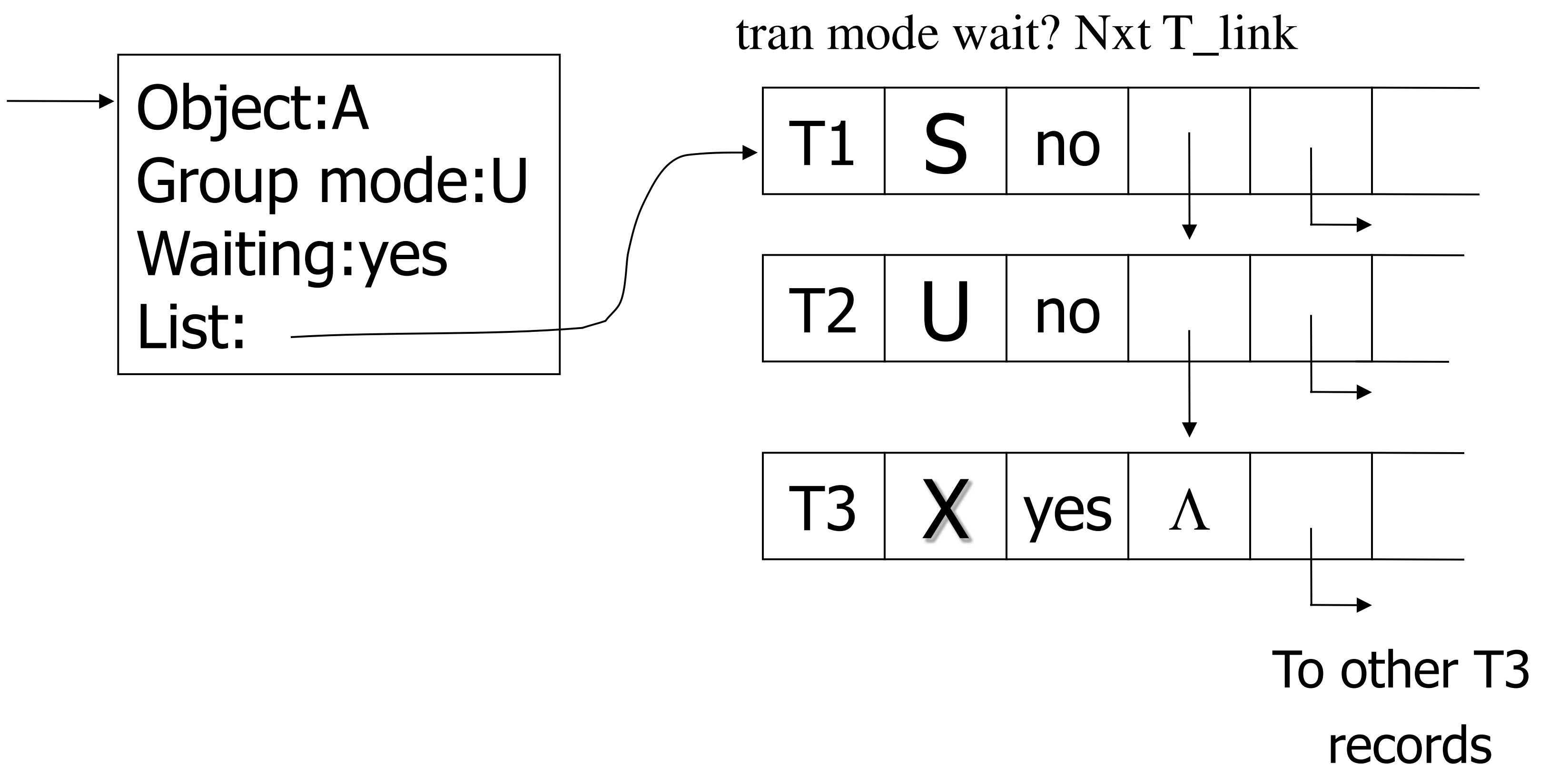


But use hash table:



If object not found in hash table, it is unlocked

Lock info for A - example



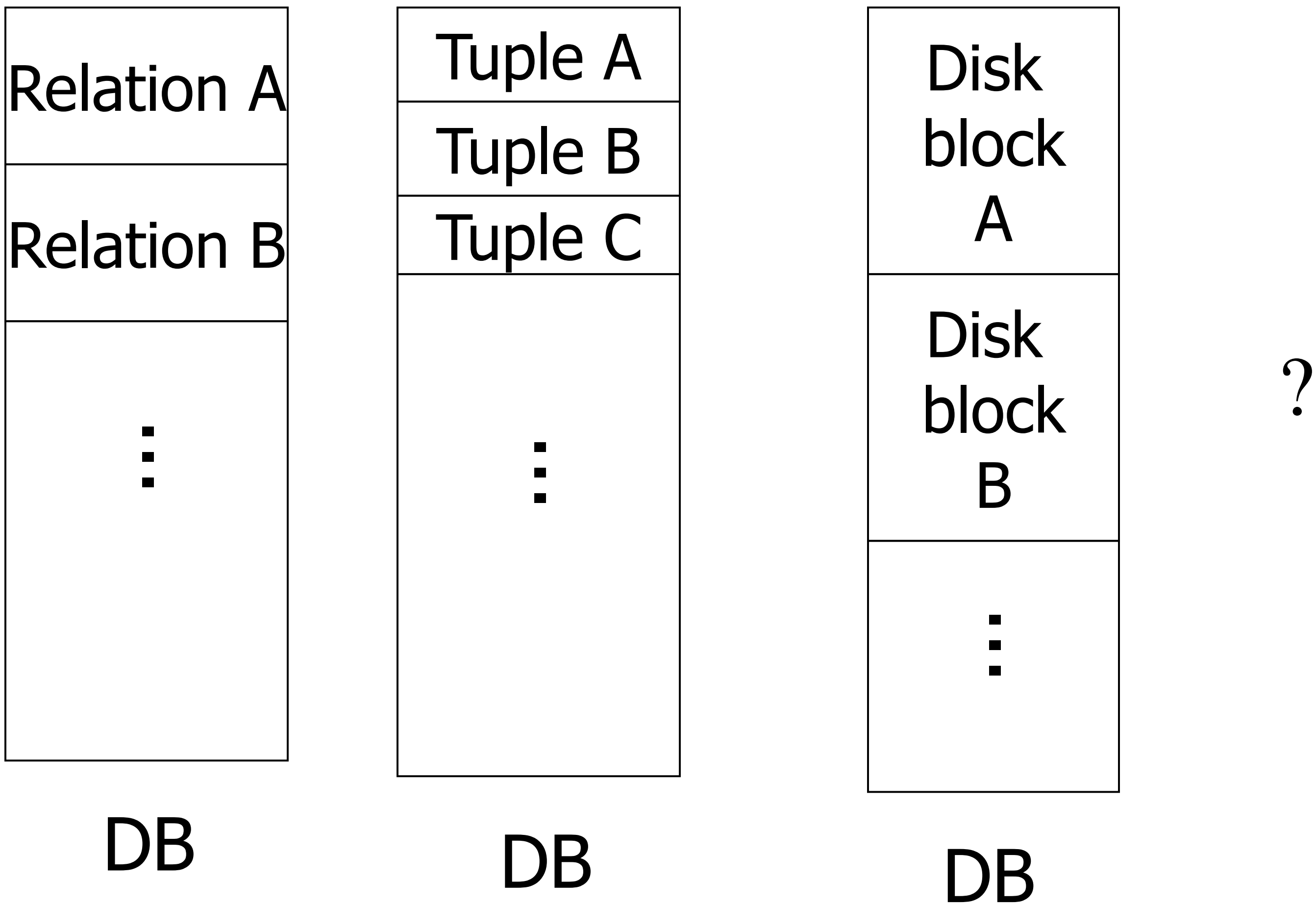
Selecting the list of requested locks

- FCFS
- Priority to shared locks
- Priority to upgrading

Managing hierarchies of database elements

- Two kinds of hierarchies
  - Hierarchy of lockable elements
- B-tree indexes

What are the objects we lock?



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

Locks with multiple granularity

- What is database element ?
  - One lock for each relation
  - One lock for each tuple
  - One lock for each page

Warning locks
Hierarchy Locking Protocol

- Relations are largest lockable elements
- Each relation consists of blocks
- Each block consists of tuples.
- The rules for managing hierarchy locking protocol involves warning locks and ordinary locks.

Warning protocol

1. To place an ordinary S or X lock on any element begin from the root
2. If we are at the element request S or X.
3. If the element is further down in the hierarchy, place a warning on this node.
   - If we want to request S lock further down, place IS lock on this node.
     - Here, “IS” means intention to obtain shared lock on the sub-element.
   - If we want to request X lock further down, place IX lock on this node. After grating the lock, proceed further and repeat 1,2, or 3.

Warning protocol: Compatibility matrix

	IS	IX	S	X
IS	Yes	Yes	Yes	No
IX	Yes	Yes	No	No
S	Yes	No	Yes	No
X	No	No	No	NO

Phantoms and Handling insertions Correctly

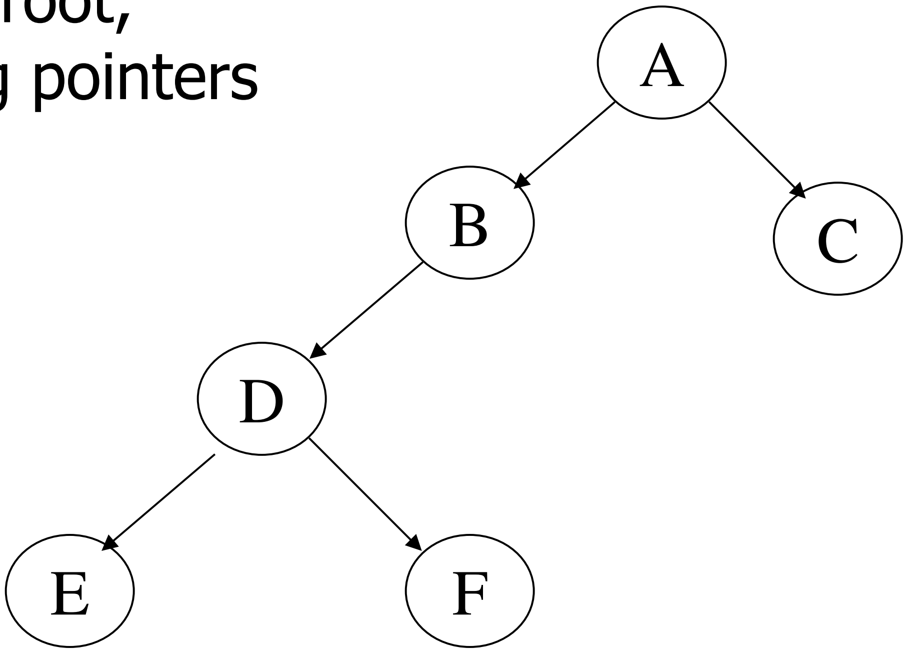
- We can only lock the existing elements!
- Find all Disney movies.
- But new Disney movie is inserted, in between. Does not require the lock.
- Solution: lock the relation.

The Tree Protocol

- So far we have discussed about the elements organized in an hierarchical order
- Linked pattern of trees
  - Btrees
- Btrees allows the locking of individual nodes.
  - Treating entire B-tree as one element reduces the concurrency.
  - If we use S,X,Update, we can not design a CC protocol for Btree.
- If the transaction moves to child, and observes that split does not propagate upward, we can release the locks of higher level nodes.
  - Violates 2PL, but ensures serializability.

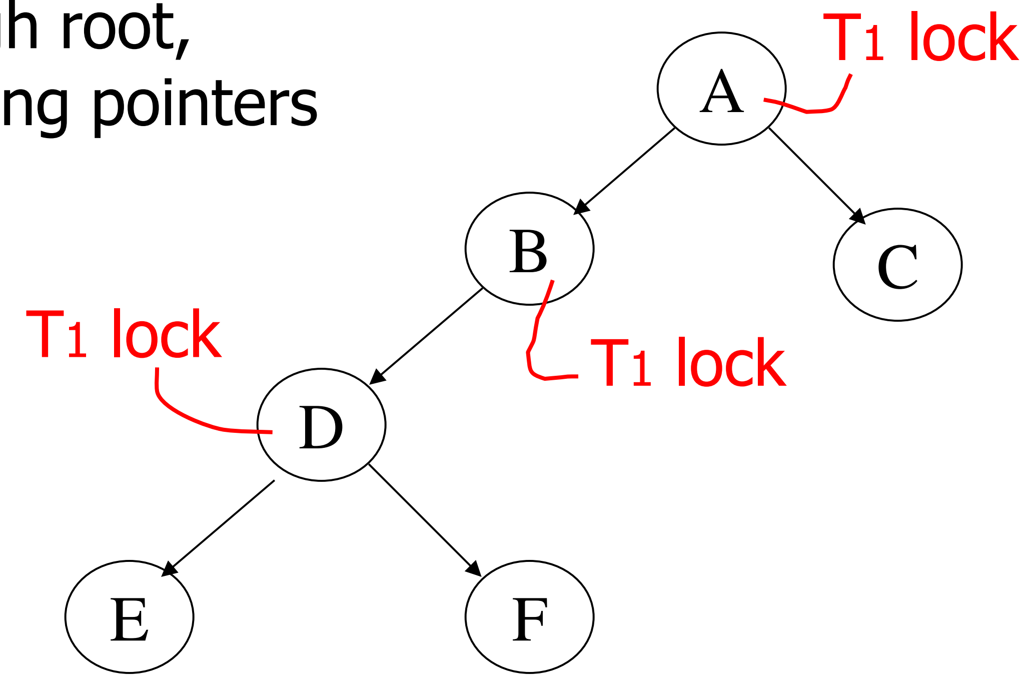
Example

- all objects accessed through root, following pointers



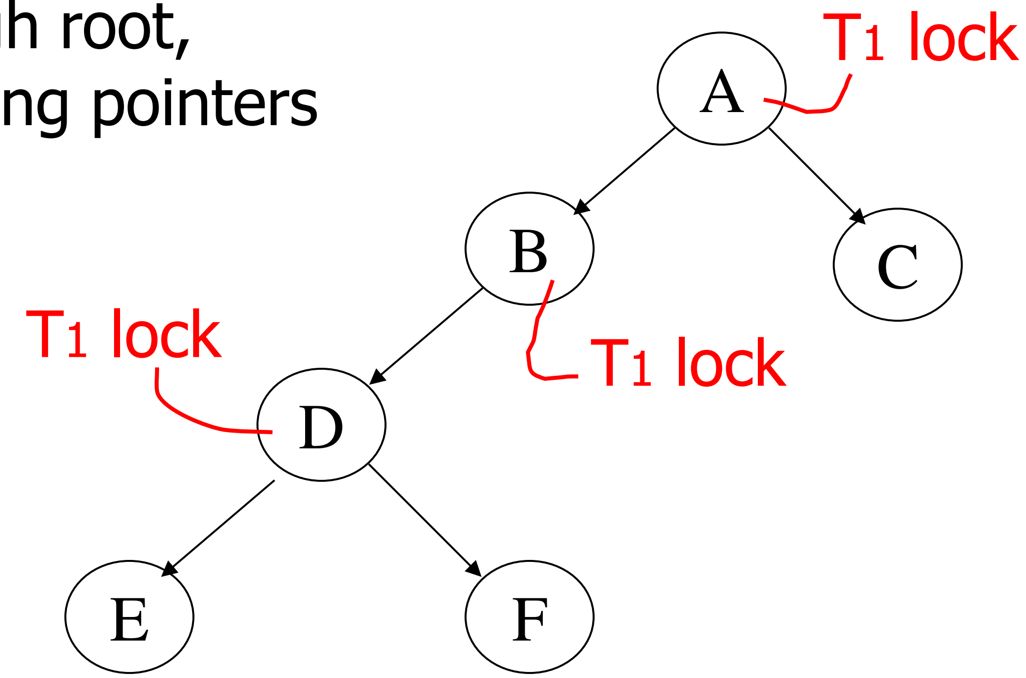
Example

- all objects accessed through root, following pointers



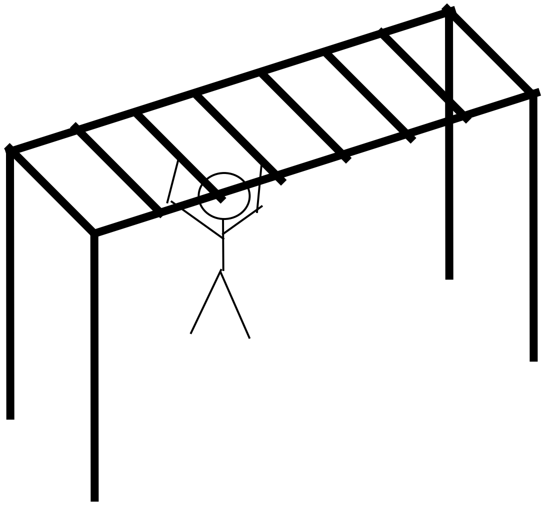
Example

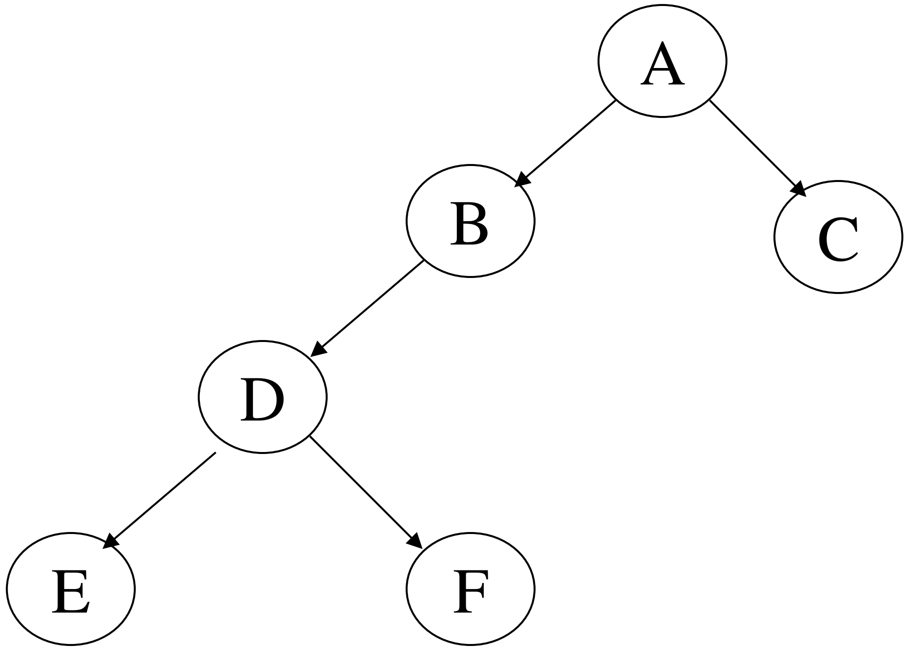
- all objects accessed through root, following pointers



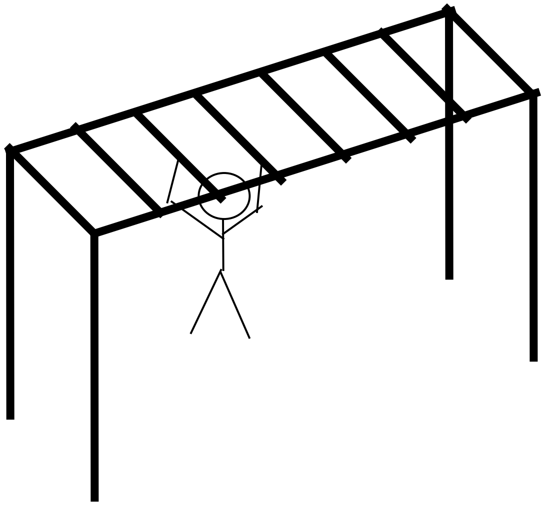
☛ can we release A lock if we no longer need A??

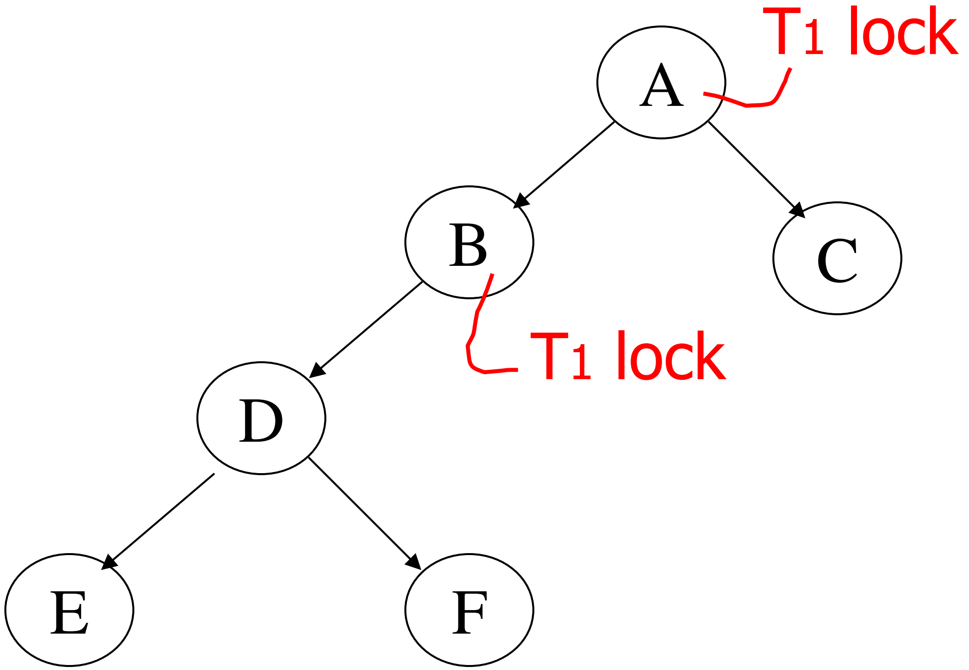
Idea: traverse like “Monkey Bars”



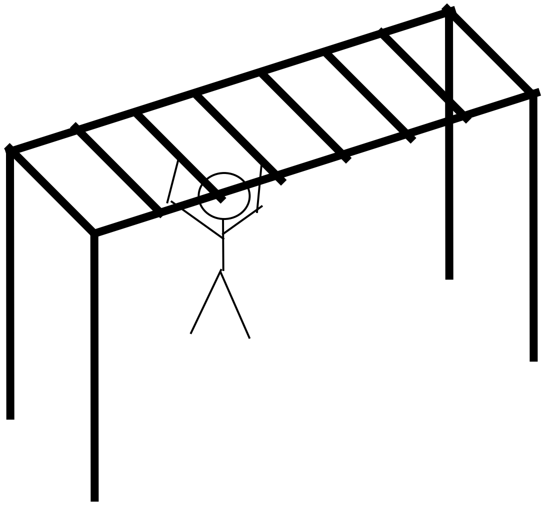


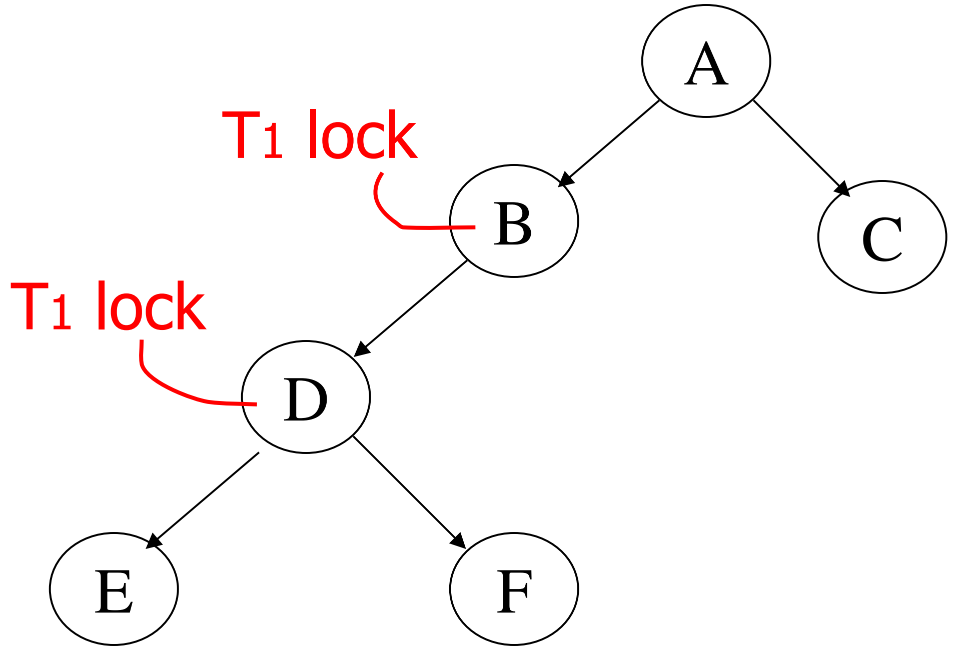
Idea: traverse like "Monkey Bars"





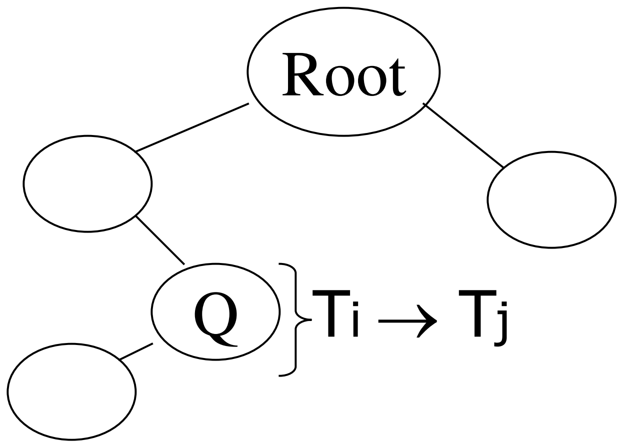
Idea: traverse like "Monkey Bars"





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)

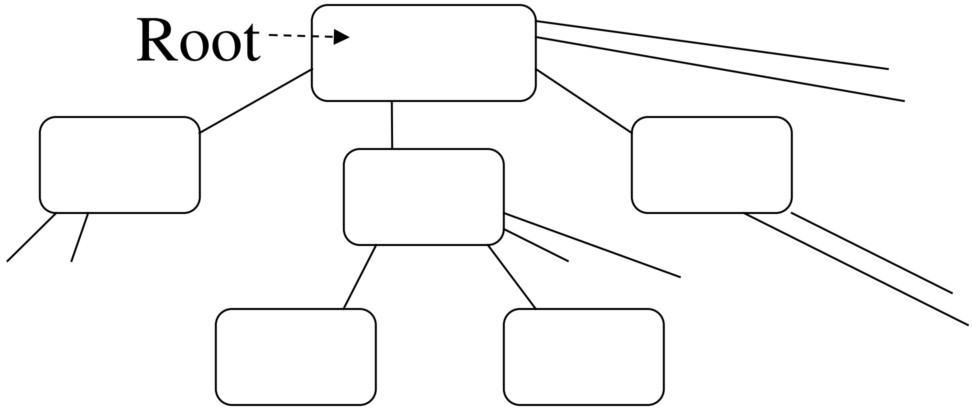
(1) First lock by T_i may be on any item

(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,
even it has a lock on node's parent.

- 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

Concurrency Control By Timestamps Optimistic protocols

- Two methods
  - Timestamping
    - Serial schedule is according to timestamps.
  - Validation
    - Serial schedule is according to validation times.
- These are Optimistic protocols
  - Assume that conflicts are rare
- Locking protocols are pessimistic protocols
  - Assume that conflicts are frequent.

Timestamps

- It is a unique number
  - Transaction which starts later has higher timestamp.
- Each object X has two time stamps
  - RT(X): read time of X: highest timestamp of the transaction which has read X.
  - WT(X): write time of X: highest timestamp of the transaction which has written X.
  - C(X), commit bit of X, true only most recent transaction to write X has already committed.
    - Suppose U writes X and aborts; dirty read problem can be avoided.

Timestamp-based Scheduling: Rules

- Suppose the scheduler receives $r_T(X)$
  - If TS(T) >= WT(X), read is possible
    - If C(X) is true, grant the request
    - If TS(T) > RT(X), set RT(X):=TS(T); otherwise do not change RT(X).
    - If C(X) is false, delay T until C(X) becomes true or the transaction which wrote X aborts
  - If TS(T) < WT(X) rollback T and restrat with new time stamp
- Suppose the scheduler receives $w_T(X)$
  - If TS(T) >= RT(X) and TS(T) >= WT(X), write is possible
    - Write X
    - Set WT(X) := TS(T)
    - Set C(X) is false.
  - If TS(T) >= RT(X) and TS(T) < WT(X), write is possible, but there is a later value of X. If C(X) is true, ignore T. Otherwise delay T.
  - If TS(T) < RT(X) then write is not possible
- If the scheduler receives Commit(T), make C(X) true.
- If the scheduler receives Abort(T), take appropriate action.

Multiversion Timestamping

- Store the old versions
- Allow reads $r_T(X)$ that otherwise cause transaction T to abort

Multiversion Protocol

- When a new $w_T(X)$ occurs, new version of X is created. It write time is TS(T), let it be Xt.
- When a rT(X) occurs, the scheduler finds version Xt of X such that t <= TS(T)
- Write times are associated with the versions and they do not change
- Read times are associated with versions. Certain writes are rejected one whose time is less than the read time of the previous version.
- If there is no active transaction less than t, delete any version of X previous to Xt.

Timestamps and locking

- Timestamp is better of most of the transactions are readonly.
- Locking is better in high conflict situations.
- Approach followed by commercial systems
  - Read only transactions are executed using multiversion timestamping.
  - Read/write transactions are executed using 2PL.

Concurrency Control by validation

- It is another type of optimistic control.
- Different from timestamping
  - Scheduler maintains the record of active transactions (not the timestamps of elements)
  - Before writing, a transaction goes through validation phase.
    - The items it wants to write are compared with write sets of other active transactions. If there is any problem, the transaction is rolled back.

Optimistic protocol: phases

- Read: reads the all the elements in read-set and computes the result in local memory.
- Validate: validates the transaction by comparing read and write sets with those of other transactions. If the validation fails, the transaction is rolled back.
- Write: transaction writes the values in the write set.

Sets maintained by Scheduler for validation

- START: set of transactions that have started, but not yet completed validation.
- VAL: set of transactions validated and not finished the writing.
- FIN: set of transactions that have completed in Phase 3.
- For each transaction, the scheduler maintains START(T), VAL(T) and FIN(T) timestamp.
- FIN is frequently purged.

Validation Rules for T

When we are trying to validate T

- Suppose there is U
  - If U is in VAL or FIN; that is, U has validated.
  - FIN(U) > START(T); U did not finish before T started.
  - RS(T) $\cap$ WS(U) not empty. Risky, So rollback T.
- Suppose there is U
  - U is in VAL, U is successfully validated
  - FIN(U) > VAL(T), U did not finish before T entered its validation phase.
  - WS(T) $\cap$ WS(U) not empty. Risky, So rollback T.

Comparison of Three CC protocols

- Locks:
  - Space is proportional to the number of database elements locked.
  - Delays transactions but avoids rollbacks
- Timestamps:
  - Space is needed for timestamps (naïve implementation)
  - Similar to the lock table, we can store the timestamps.
  - Rollback problem (detects earlier than validation)
- Validation
  - Space is used for timestamps and read/write sets.
  - Roll back problem

Summary

Have studied C.C. mechanisms used in practice

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

System Structure