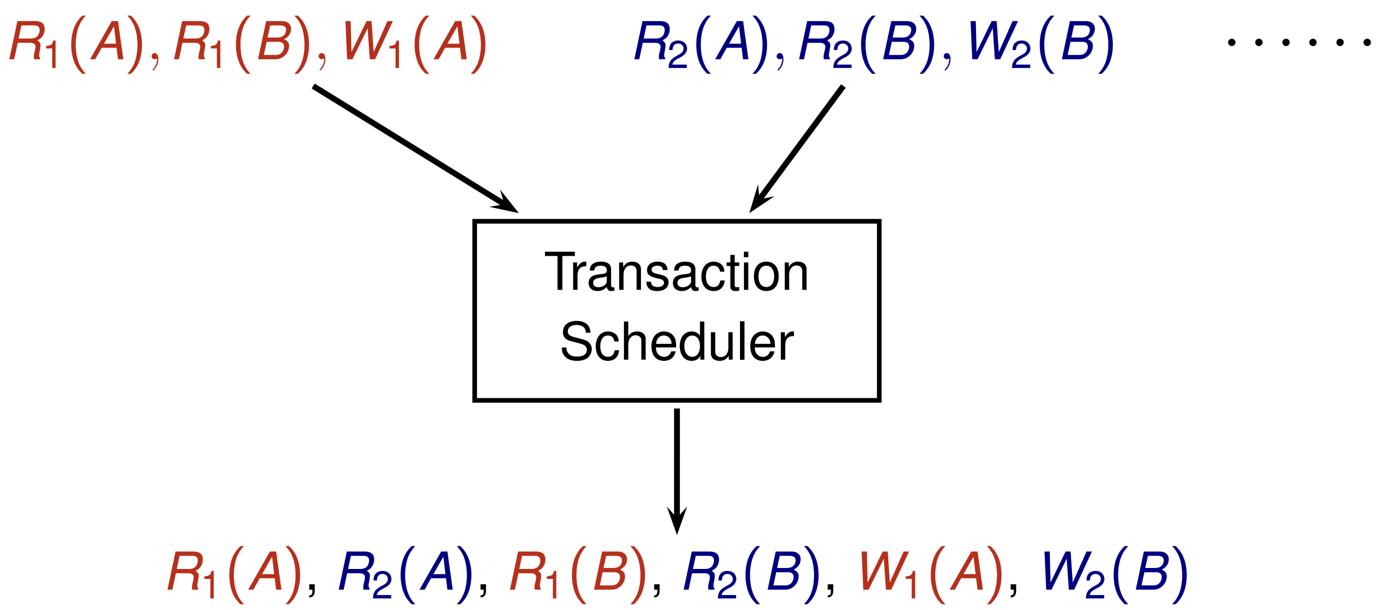
CS3223 Lecture 8
Concurrency Control

Transaction Scheduler



- For each input action (read, write, commit, abort) to the scheduler, the scheduler performs one of the following:
  - output the action to the schedule,
  - postpone the action by blocking the transaction, or
  - reject the action and abort the transaction

Concurrency Control Algorithms

- Lock-based concurrency control
- Timestamp-based concurrency control
- Multiversion concurrency control
- Optimistic concurrency control

Lock-Based Concurrency Control

- Each Xact needs to request for an appropriate lock on an object before the Xact can access the object
- Locking modes
  - Shared (S) locks for reading objects
  - Exclusive (X) locks for writing objects
- Lock compatibility:

Lock Requested	Lock Held: -	S	X
S	$\surd$	$\surd$	$\times$
X	$\surd$	$\times$	$\times$

$\surd$: Compatible
Lock request is granted

$\times$: Incompatible
Lock request is blocked

Lock-Based Concurrency Control (cont.)

1. To **read an object** *O*, a Xact must request for a shared/exclusive lock on *O*
2. To **update an object** *O*, a Xact must request for an exclusive lock on *O*
3. A **lock request is granted** on *O* if the requesting lock mode is compatible with the lock modes of existing locks on *O*
4. If *T*'s **lock request is not granted** on *O*, *T* becomes blocked: its execution is suspended & *T* is added to *O*'s request queue
5. When a **lock is released** on *O*, the lock manager checks the request of the first Xact *T* in the request queue for *O*. If *T*'s request can be granted, *T* acquires its lock on *O* and resumes execution after its removal from the queue
6. When a Xact **commits/aborts**, all its locks are released & T is removed from any request queue it's in

Lock-Based Concurrency Control

- Notations
  - $S_i(O)$: Xact T_i is requesting S-lock on object O
  - $X_i(O)$: Xact T_i is requesting X-lock on object O
  - $U_i(O)$: Xact T_i releases lock on object O
- **Example**:

$$R_1(A), \quad W_2(A), \quad W_2(B), \quad W_1(B)$$

Lock-Based Concurrency Control

- Notations
  - $S_i(O)$: Xact T_i is requesting S-lock on object O
  - $X_i(O)$: Xact T_i is requesting X-lock on object O
  - $U_i(O)$: Xact T_i releases lock on object O
- **Example**:

$$\boxed{R_1(A),} \quad W_2(A), \quad W_2(B), \quad W_1(B)$$

$S_1(A), R_1(A),$

Lock-Based Concurrency Control

- Notations
  - $S_i(O)$: Xact T_i is requesting S-lock on object O
  - $X_i(O)$: Xact T_i is requesting X-lock on object O
  - $U_i(O)$: Xact T_i releases lock on object O
- **Example**:

$$R_1(A), \quad \boxed{W_2(A),} \quad W_2(B), \quad W_1(B)$$

$S_1(A), R_1(A), U_1(A), X_2(A), W_2(A),$

Lock-Based Concurrency Control

- Notations
  - $S_i(O)$: Xact T_i is requesting S-lock on object O
  - $X_i(O)$: Xact T_i is requesting X-lock on object O
  - $U_i(O)$: Xact T_i releases lock on object O
- **Example**:

$$R_1(A), \quad W_2(A), \quad \boxed{W_2(B),} \quad W_1(B)$$

$S_1(A)$,$R_1(A)$, $U_1(A)$,$X_2(A)$, $W_2(A)$, $X_2(B)$, $W_2(B)$,

Lock-Based Concurrency Control

- Notations
  - $S_i(O)$: Xact T_i is requesting S-lock on object O
  - $X_i(O)$: Xact T_i is requesting X-lock on object O
  - $U_i(O)$: Xact T_i releases lock on object O
- **Example**:

$$R_1(A), \quad W_2(A), \quad W_2(B), \quad \boxed{W_1(B)}$$

$S_1(A), R_1(A), U_1(A), X_2(A), W_2(A), X_2(B), W_2(B), U_2(A), U_2(B), X_1(B), W_1(B),$

Lock-Based Concurrency Control

- Notations
  - $S_i(O)$: Xact T_i is requesting S-lock on object O
  - $X_i(O)$: Xact T_i is requesting X-lock on object O
  - $U_i(O)$: Xact T_i releases lock on object O
- **Example**:

$$R_1(A), \quad W_2(A), \quad W_2(B), \quad W_1(B)$$

$S_1(A)$,$R_1(A)$, $U_1(A)$,$X_2(A)$, $W_2(A)$, $X_2(B)$, $W_2(B)$, $U_2(A)$,$U_2(B)$,$X_1(B)$, $W_1(B)$, $U_1(B)$

Two Phase Locking (2PL) Protocol

- 2PL Protocol:
  1. To read an object O, a Xact must hold a S-lock or X-lock on O
  2. To write to an object O, a Xact must hold a X-lock on O
  3. Once a Xact releases a lock, the Xact can't request any more locks
- Xacts using 2PL can be characterized into two phases:
  - Growing phase: before releasing 1^{st} lock
  - Shrinking phase: after releasing 1^{st} lock

Two Phase Locking (2PL) Protocol (cont.)

- **Example**: $R_1(A)$, $W_2(A)$, $W_2(B)$, $W_1(B)$

$S_1(A)$, $R_1(A)$, $\boxed{U_1(A)}$, $X_2(A)$, $W_2(A)$, $X_2(B)$, $W_2(B)$, $U_2(A)$, $U_2(B)$, $\boxed{X_1(B)}$, $W_1(B)$, $U_1(B)$

Not permitted by 2PL!

- The above example schedule is not a 2PL schedule

Strict Two Phase Locking (strict 2PL) Protocol

- 2PL Protocol:
  1. To read an object O, a Xact must hold a S-lock or X-lock on O
  2. To write to an object O, a Xact must hold a X-lock on O
  3. Once a Xact releases a lock, the Xact can't request any more locks
- **Theorem 1**: 2PL schedules are conflict serializable

Strict Two Phase Locking (strict 2PL) Protocol

- 2PL Protocol:
  1. To read an object O, a Xact must hold a S-lock or X-lock on O
  2. To write to an object O, a Xact must hold a X-lock on O
  3. Once a Xact releases a lock, the Xact can't request any more locks
- **Theorem 1**: 2PL schedules are conflict serializable
- Strict 2PL Protocol:
  1. To read an object O, a Xact must hold a S-lock or X-lock on O
  2. To write to an object O, a Xact must hold a X-lock on O
  3. A Xact must hold on to locks until Xact commits or aborts
- **Theorem 2**: Strict 2PL schedules are strict & conflict serializable

Lock Management

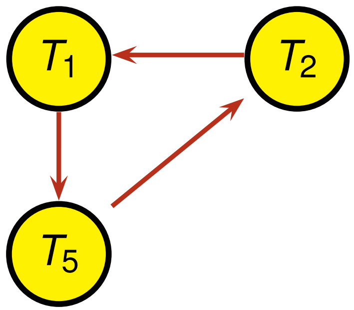
- Handling deadlocks
- Lock conversion

Deadlocks

- Deadlock: cycle of Xacts waiting for locks to be released by each other
- **Example**:
  T_1 requests X-lock on A and is granted;
  T_2 requests X-lock on B and is granted;
  T_1 requests X-lock on B and is blocked;
  T_2 requests X-lock on A and is blocked;
- Dealing with deadlocks:
  - deadlock detection
  - deadlock prevention

How to Detect Deadlocks?

- Waits-for graph (WFG)
  - Nodes represent active Xacts
  - Add an edge $T_i \rightarrow T_j$ if T_i is waiting for T_j to release a lock



- Lock manager
  - adds an edge when it queues a lock request
  - updates edges when it grants a lock request
- Deadlock is detected if WFG has a cycle
- Breaks a deadlock by aborting a Xact in cycle
- Alternative to WFG: timeout mechanism

How to Prevent Deadlocks?

- Assume older Xacts have higher priority than younger Xacts
  - Each Xact is assigned a timestamp when it starts
  - An older Xact has a smaller timestamp
- Suppose T_i requests for a lock that conflicts with a lock held by T_j
- Two possible deadlock prevention policies:
  - Wait-die policy: lower-priority Xacts never wait for higher-priority Xacts
  - Wound-wait policy: higher-priority Xacts never wait for lower-priority Xacts

Prevention Policy	T_i has higher priority	T_i has lower priority
Wait-die	T_i waits for T_j	T_i aborts
Wound-wait	T_j aborts	T_i waits for T_j

How to Prevent Deadlocks? (cont.)

- Wait-die policy
  - non-preemptive: only a Xact requesting for a lock can get aborted
  - a younger Xact may get repeatedly aborted
  - a Xact that has all the locks it needs is never aborted
- Wound-wait policy
  - preemptive
- To avoid starvation, a restarted Xact must use its original timestamp!

Lock Conversion

- Consider two Xacts:

  T_1: $R_1(A)$, $R_1(B)$, $W_1(A)$
  T_2: $R_2(A)$, $R_2(B)$

- Since T_1 needs to update A, T_1 requires an exclusive lock on A
- All possible schedules are serial

  S_1: $X_1(A)$, $R_1(A)$, $S_1(B)$, $R_1(B)$, $W_1(A)$, $U_1(A)$, $U_1(B)$, $S_2(A)$, $R_2(A)$, $S_2(B)$, $R_2(B)$, $U_2(A)$, $U_2(B)$
  S_2: $S_2(A)$, $R_2(A)$, $S_2(B)$, $R_2(B)$, $U_2(A)$, $U_2(B)$, $X_1(A)$, $R_1(A)$, $S_1(B)$, $R_1(B)$, $W_1(A)$, $U_1(A)$, $U_1(B)$

- Increase concurrency by allowing **lock conversions**
- Two types of lock conversions
  - $UG_i(A)$: T_i upgrades its S-lock on object A to X-lock
  - $DG_i(A)$: T_i downgrades its X-lock on object A to S-lock

Lock Conversion (cont.)

- Interleaved executions become possible with lock upgrading:

S_3: $S_1(A)$, $R_1(A)$, $S_2(A)$, $R_2(A)$, $S_2(B)$, $R_2(B)$, $U_2(A)$, $U_2(B)$, $S_1(B)$, $R_1(B)$, $UG_1(A)$, $W_1(A)$, $U_1(A)$, $U_1(B)$
S_4: $S_1(A)$, $R_1(A)$, $S_1(B)$, $R_1(B)$, $S_2(A)$, $R_2(A)$, $S_2(B)$, $R_2(B)$, $U_2(A)$, $U_2(B)$, $UG_1(A)$, $W_1(A)$, $U_1(A)$, $U_1(B)$
S_5: $S_1(A)$, $R_1(A)$, $S_2(A)$, $R_2(A)$, $S_1(B)$, $R_1(B)$, $S_2(B)$, $R_2(B)$, $U_2(A)$, $U_2(B)$, $UG_1(A)$, $W_1(A)$, $U_1(A)$, $U_1(B)$

- **Lock upgrade** $UG_i(A)$
  - Upgrade request is blocked if another Xact is holding a shared lock on A
  - Upgrade request is allowed if T_i has not released any lock
- **Lock downgrade** $DG_i(A)$
  - Downgrade request is allowed if
    1. T_i has not modified A, and
    2. T_i has not released any lock

Performance of Locking

- Resolve Xact conflicts by using blocking and aborting mechanisms
- Blocking causes delays in other waiting Xacts
- Aborting and restarting a Xact wastes work done by Xact
- How to increase system throughput?
  1. Reduce the locking granularity
  2. Reduce the time a lock is held
  3. Reduce hot spots - a hot spot is a DB object that is frequently accessed and modified

Concurrency Control Anomalies & Locking

- Dirty read problem: $W_1(x)$, $R_2(x)$
- Unrepeatable read problem: $R_1(x)$, $W_2(x)$, Commit_2, $R_1(x)$
- Lost update problem: $R_1(x)$, $R_2(x)$, $W_1(x)$, $W_2(x)$
- **Phantom read problem**

  A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently committed transaction.

Phantom Read Problem

Accounts

account	name	balance
100	Alice	5000
200	Bob	800
300	Carol	1000

Transaction 1	Transaction 2
begin transaction;	
select name **from** Accounts **where** balance $>$ 1000;	
	begin transaction;
	insert into Accounts **values** (400,'Dave',3000);
	commit;
select name **from** Accounts **where** balance $>$ 1000;	
commit;	

Phantom Read Problem (cont.)

Accounts

account	name	balance
100	Alice	5000
200	Bob	800
300	Carol	1000

$\longrightarrow$

Accounts

account	name	balance
100	Alice	5000
200	Bob	800
300	Carol	1000
400	Dave	3000

```
begin transaction;
select  name
from    Accounts
where   balance > 1000;
                                begin transaction;
                                insert into Accounts
                                   values (400,'Dave',3000);
                                commit;
select  name
from    Accounts
where   balance > 1000;
commit;
```

R_1(100,Alice,5000)
R_1(200,Bob,800)
R_1(300,Carol,1000)
Output: {Alice}

W_2(400,Dave,3000)
$Commit_2$

R_1(100,Alice,5000)
R_1(200,Bob,800)
R_1(300,Carol,1000)
R_1(400,Dave,3000)
$Commit_1$
Output: {Alice, Dave}

Phantom Read Problem (cont.)

Accounts

account	name	balance
100	Alice	5000
200	Bob	800
300	Carol	1000

$\longrightarrow$

Accounts

account	name	balance
100	Alice	5000
200	Bob	800
300	Carol	1000
400	Dave	3000

Xact 1	Xact 2
begin transaction;	
select name	
from Accounts	
where balance $>$ 1000;	
	begin transaction;
	insert into Accounts
	values (400,'Dave',3000);
	commit;
select name	
from Accounts	
where balance $>$ 1000;	
commit;	

- Phantom problem can be prevented by predicate locking
  - Xact 1 is granted a shared lock on the predicate "balance $>$ 1000"
  - Xact 2's request for an exclusive lock on the predicate "balance $=$ 3000" is blocked
- In practice, phantom problem is prevented via index locking

ANSI SQL Isolation Levels

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read
READ UNCOMMITTED	possible	possible	possible
READ COMMITTED	not possible	possible	possible
REPEATABLE READ	not possible	not possible	possible
SERIALIZABLE	not possible	not possible	not possible

- SQL's SET TRANSACTION ISOLATION LEVEL command

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL
      { READ UNCOMMITTED |
        READ COMMITTED |
        REPEATABLE READ |
        SERIALIZABLE };
......
COMMIT;
```

- In many DBMSs, the default isolation level is READ COMMITTED

Lock-based Implementation of Isolation Levels

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read
READ UNCOMMITTED	possible	possible	possible
READ COMMITTED	not possible	possible	possible
REPEATABLE READ	not possible	not possible	possible
SERIALIZABLE	not possible	not possible	not possible

Degree	**Isolation level**	**Write Locks**	**Read Locks**	**Predicate Locking**
0	Read Uncommitted	long duration	none	none
1	Read Committed	long duration	short duration	none
2	Repeatable Read	long duration	long duration	none
3	Serializable	long duration	long duration	yes

- Short duration lock: lock acquired for an operation could be released after the end of operation before Xact commits/aborts
- Long duration lock: lock acquired for an operation is held until Xact commits/aborts

Locking Granularity

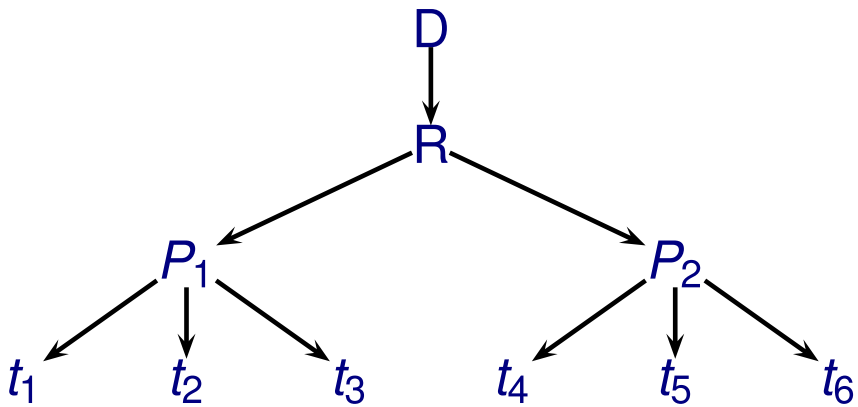
- What to lock?

$$\begin{array}{c}\text{database}\\ \downarrow \\ \text{relation}\\ \downarrow \\ \text{page}\\ \downarrow \\ \text{tuple}\end{array}$$

- Locking granularity = size of data items being locked
  highest (coarsest) granularity = database
  lowest (finest) granularity = tuple

Locking Granularity (cont.)

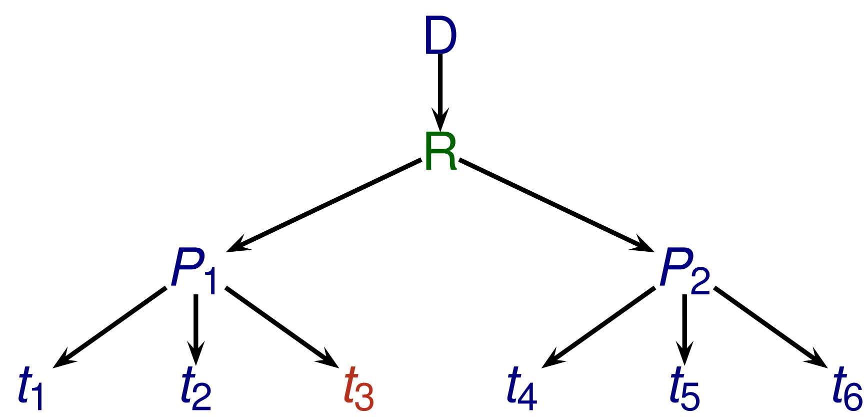
- Allow multi-granular lock instead of fixed granule locking
- If Xact T holds a lock mode M on a data granule D, then T implicitly also holds lock mode M on granules finer than D
- **Example**: Consider database D containing relation R consisting of pages P_1 and P_2 each with 3 tuples



- If T is holding a S-lock on P_1, then T is implicitly holding a S-lock on t_1, t_2, & t_3
- If T is holding a X-lock on R, then T is implicitly holding a X-lock on P_1, P_2, t_1, t_2, t_3, t_4, t_5, & t_6

Locking Granularity (cont.)

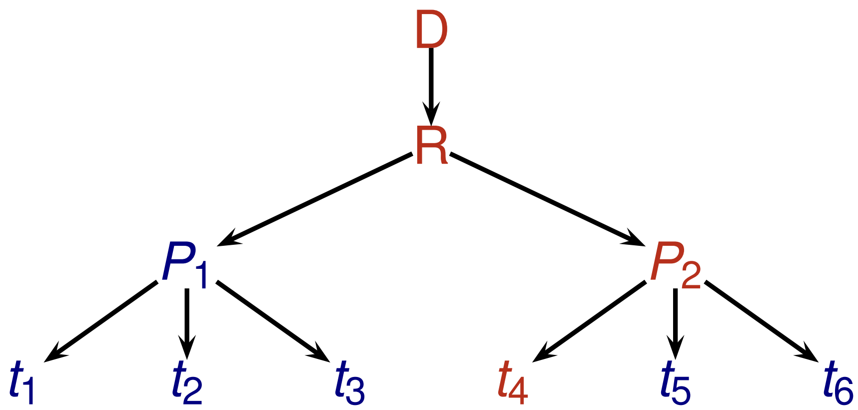
- Problem: How to detect locking conflicts?
- **Example**:



- Suppose T_1 is holding a S-lock on t_3
- If T_2 requests for X-lock on relation R, should this request be granted or blocked?

Multigranularity Locking

- Idea: Use a new intention lock (I-lock) mode
- Protocol: Before acquiring S-lock/X-lock on a data granule G, need to acquire I-locks on granules coarser than G in a top-down manner
- **Example**: Xact T wants to request X-lock on tuple t_4



T requests for I-lock on D
T requests for I-lock on R
T requests for I-lock on P_2
T requests for X-lock on t_4

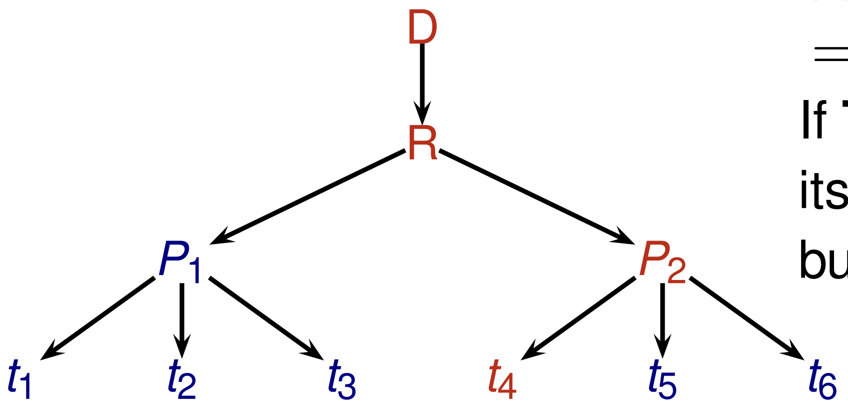
Multigranularity Locking (cont.)

Lock compatability matrix

Lock Requested	Lock Held: -	I	S	X
I	√	√	×	×
S	√	×	√	×
X	√	×	×	×

√: Compatible
Lock request is granted

×: Incompatible
Lock request is blocked



Suppose **T_1 has a X-lock on t_4**

$\Longrightarrow$ T_1 has I-locks on D, R & P_2

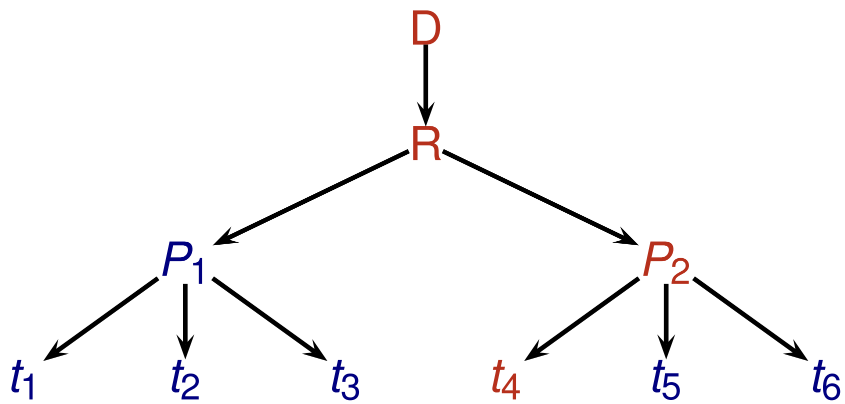
If **T_2 wants to read P_2**,

its I-lock requests on D & R will be granted,

but its S-lock request on P_2 will be blocked

Multigranularity Locking (cont.)

- Problem: Limited concurrency with lock modes I, S, and X
- **Example**:



- Suppose **T_1 has a S-lock on t_4**
  $\Longrightarrow$ T_1 has I-locks on D, R & P_2
- If **T_2 wants to read P_2**,
  - its I-lock requests on D & R will be granted,
  - but its S-lock request on P_2 will be blocked

Multigranularity Locking (cont.)

- Refine *intention lock* idea with IS & IX lock modes
  - intention shared (IS): intent to set S-locks at finer granularity
  - intention exclusive (IX): intent to set X-locks at finer granularity

Lock compatability matrix

Lock Requested	Lock Held: -	IS	IX	S	X
IS	√	√	√	√	×
IX	√	√	√	×	×
S	√	√	×	√	×
X	√	×	×	×	×

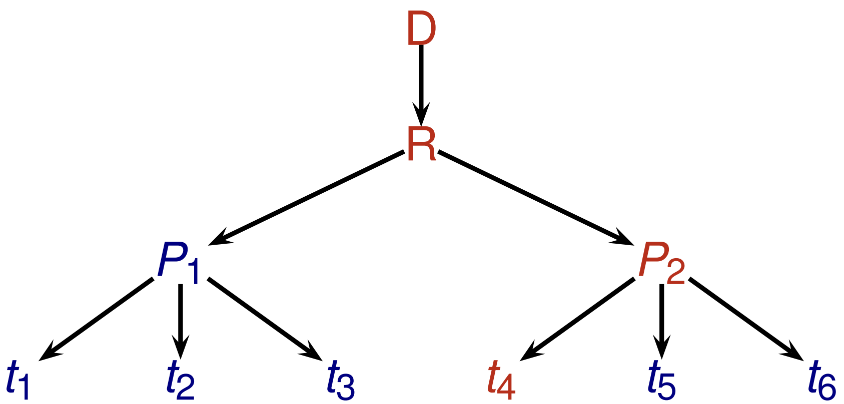
Multi-granular locking protocol:

- Locks are acquired in top-down order
- To obtain S or IS lock on a node, must already hold IS or IX lock on its parent node
- To obtain X or IX lock on a node, must already hold IX lock on its parent node
- Locks are released in bottom-up order

Multigranularity Locking (cont.)

Lock compatability matrix

Lock Requested	Lock Held: -	IS	IX	S	X
IS	√	√	√	√	×
IX	√	√	√	×	×
S	√	√	×	√	×
X	√	×	×	×	×



- For T_1 to read t_4:
  1. T_1 acquires IS-lock on D
  2. T_1 acquires IS-lock on R
  3. T_1 acquires IS-lock on P_2
  4. T_1 acquires S-lock on t_4
- For T_2 to read P_2:
  1. T_1 acquires IS-lock on D
  2. T_1 acquires IS-lock on R
  3. T_1 acquires S-lock on P_2