- Standard SQL interface: each SQL statement is a transaction
- SQL allows the programmer to group statements into a transaction
- `START TRANSACTION` marks the beginning of a transaction
- No `END TRANSACTION` ! Two possibilities
    - `COMMIT` causes the transaction to end successfully. Changes made by the transaction are made permanent ("committed")
    - `ROLLBACK` causes the transaction to abort, or terminate unsuccessfully. Any changes made by the transaction are undone ("rolled back")
- Note that "temporary" changes could be read by other transactions. In case of rollback, these transactions must also be rolled back

Example

- Bank transfer example
- Execute `BEGIN TRANSACTION` before accessing the database
- If there are insufficient funds, execute `ROLLBACK`
- If there are sufficient funds, we execute the update statements and then execute `COMMIT`

Read-Only transactions

- Our examples had transactions that read and then wrote some data into the database
- We saw the problems that can arise
- If a transaction only reads data there is more freedom for parallel execution

Example

- A program that determines whether a certain seat is available
- We can execute this many times in parallel without problems
- We tell the SQL system that our current transaction is readonly, which makes more efficient execution possible:

  ```
  SET TRANSACTION READ ONLY;
  ```

- We can also write

  ```
  SET TRANSACTION READ WRITE;
  ```

  (but this is the default)

Dirty data

- *Dirty data*: data written by a transaction that has not yet committed
- *Dirty read*: read of dirty data written by another transaction
- Risk: The transaction that wrote the data may abort
- Then the dirty data will be removed from the database, as though it never existed
- The transaction has read the dirty data must then also abort and so on commit or take some other action that reflects its knowledge of the dirty data.

Allowing dirty reads

- Sometimes dirty read are not a serious problem
- In such a case, it makes sense to risk occasional dirty reads
- This avoids
  - The costly work by the DBMS to avoid dirty reads
  - The loss of parallelism from waiting until there is no possibility of a dirty read

Example

- Account transfer example
- Suppose transfers are implemented by a program P that does
  1. Add money to account 2
  2. Test if account 1 has enough money.
  3. If there is not enough money, remove the money from account 2 and end.
  4. If there is enough money, subtract the money from account 1 and end
- In a serializable execution, the money inserted into account 2 will not be seen by any transaction (until P ends)

When to avoid dirty reads

- Three accounts: A_1 (\$100), A_2 (\$200), and A_3 (\$300)
- T_1 transfers \$150 from A_1 to A_2
- At about the same time, transaction T_2 transfers \$250 from A_2 to A_3
- A possible sequence of events:
  1. T_2 adds \$250 to A_3, which now has \$550
  2. T_1 adds \$150 to A_2, which now has \$350
  3. T_2 finds that A_2 has enough funds (\$350) for the transfer of \$250 from A_2 to A_3
  4. T_1 finds that A_1 does not have enough funds (\$100) to allow the transfer of \$150 from A_1 to A_2
  5. T_2 subtracts \$250 from A_2 (\$100) and ends
  6. T_1 subtracts \$150 from A_2 (\$50) and ends.
- The total amount of money has not changed: Total is still \$600
- But one account is now negative

When dirty reads are not a problem

- A variation of the seat-choosing example
- Find an available seat and reserve it by setting `seatStatus` to "occupied" for that seat. If there are no seats, end
- Ask the customer for approval of the seat. If so, commit. If not, release the seat by setting `seatStatus` to "available" and repeat to get another seat
- If two transactions are executing this in parallel, one might reserve a seat S, which is later rejected
- If the second transaction executes the step when S is marked occupied, the customer for that transaction is not offered this seat (dirty read)
- It might not be so important here

SQL

- SQL specification

  ```
  SET TRANSACTION READ WRITE
  ISOLATION LEVEL READ UNCOMMITTED;
  ```

  "read-uncommitted": can read uncomitted, i.e., dirty, data

- Default for transactions is usually `READ WRITE`, but when dirty reads are allowed, the default becomes read only

- Other *isolation levels*

  ```
  SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
  ```

  The latter means that we will always read tuples we have read before, but may read additional tuples if they have been inserted meanwhile

- Default is

  ```
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  ```

Practical aspects

We cannot use our algorithm since:

- We would have to wait (forever?) until all the transactions have finished
- We need to keep all the data about every transaction
- The algorithm is too inefficient

This issues can be addressed

- Algorithms such as timestamps let us check each transaction when it finishes
- They also only have to maintain limited information about each transaction
- Much more efficient

A more serious problem

What do we do if we discover a problem?

- We have to cancel this transaction (and maybe restart it again later)
- This means undoing any changes it might have made to the database
- But some other transaction might have already read data written by this one
- In that case, this transaction will have to be cancelled as well
- And so on

This works if problems are very infrequent

Otherwise, we prefer to ensure that transactions are only allowed to proceed if serializability can be guaranteed

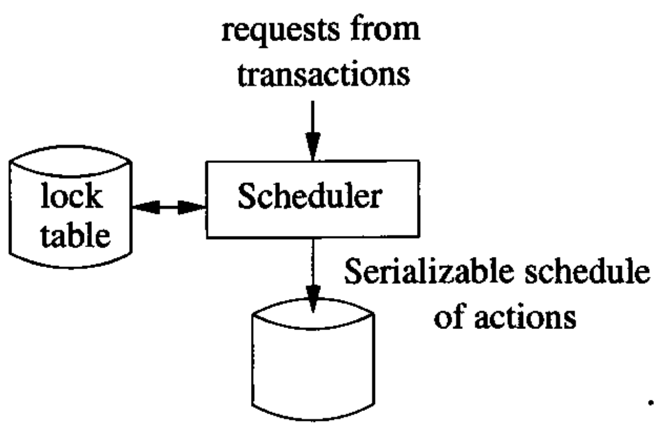
Locks

- *Locks* are maintained on database elements to stop unserializable behavior
- Intuitively, a transaction obtains locks on the database elements it accesses to prevent other transactions from accessing them at the same time
- This will guarantee serializability
- We start with a very simple locking scheme
- This guarantees serializability, but may prohibit many serializable schedules
- We then study variants that are more complex but allow more serializable schedules

Objects

- Our algorithms access “objects”
- What are objects?
- Relations, tuples, part of a relation, disk block etc.
- The smaller the objects, the more we can run in parallel
- But the cost of keeping track of what we lock could be prohibitive
- Intuition: Assume that objects are tuples, even if this is unrealistic in practice

Locks



- Scheduler uses lock table
- Scheduler takes requests from transactions and allow them to run, or blocks them until it is safe
- Scheduler uses a *lock table* to guide decisions
- Ideally: scheduler forwards a request whever its execution can never lead to an inconsistent database state

- A locking scheduler should guarantee conflict-serializability
- Transactions must now request and release locks, as well as reading and writing database objects
- They must satisfy the two properties:
- *Consistency of Transactions:*
  - A transaction can read or write an element only if it previously was granted a lock on that element and hasn't yet released it
  - If a transaction locks an element, it must later unlock that element ("release" it)
- *Legality of Schedules:* Locks must have their intended meaning. In other words, no two transactions can have a lock on the same object at the same time

Notation

- $l_i(X)$: T_i requests a lock on database element X
- Note that it is up to the scheduler to decide whether to grant the request or not
- $u_i(X)$: T_i releases ("unlocks") its lock on X
- Consistency:

  Whenver T contains an action $r_i(X)$ or $w_i(X)$, there must be a previous action $l_i(X)$ with no intervening action $u_i(X)$, and there must be a subsequent $u_i(X)$

- Legality of schedules:

  If a schedule contains an action $l_i(X)$ followed by $l_j(X)$ (where $i \neq j$), there must be an action $u_i(X)$ between them

Example

- Our standard example
- We add lock and unlock actions
- $T1$: $l_1(A); r_1(A); A := A + 100; w_1(A); u_1(A); l_1(B); r_1(B); B := B + 100; w_1(B); u_1(B);$
- T_2: $l_2(A); r_2(A); A := A * 2; w_2(A); u_2(A); l_2(B); r_2(B); B := B * 2; w_2(B); u2(B)$
- Next slide: a legal schedule

T_1	T_2	A	B
		25	25
$l_1(A); r_(A)$			
$A \leftarrow A + 100$			
$w_1(A); u_1(A)$		125	
	$l_2(A); r_2(A)$		
	$A \leftarrow A * 2$		
	$w_2(A); u_2(A)$		250
	$l_2(B); r_2(B)$		
	$B \leftarrow B * 2$		
	$w_2(B); u_2(B)$		50
$l_1(B); r_1(B)$			
$B \leftarrow B + 100$			
$w_1(B); u_1(B)$		150	

This is legal, but not serializable!

We shall later study the *two-phase locking protocol* that prevents schedules like that

Locking scheduler

- Locking scheduler: Grants requests such as $l_i(A)$ if and only if the request results in a legal schedule
- If a request is not granted, the transaction is delayed
- The transaction waits until the scheduler grants its request later
- The scheduler uses a *lock table* that says, for every database element, which transaction or transactions (if any) hold a lock on that element
- We discuss the structure of the lock table later

Two-phase locking

- Two-phase locking (2PL): In every transaction, all lock actions precede all unlock actions
- This guarantees serializability
- First phase of a transaction: the phase where locks are obtained
- Second phase: where locks are released
- A transaction that obeys the 2PL condition is called a *two-phase-locked transaction*, or 2PL transaction

Example

Our other example does obey 2PL

Note how one transaction has a lock on B denied and has to wait for T_1 to release it

T_1	T_2	A	B
		25	25
$l_1(A); r_(A)$			
$A \leftarrow A + 100$			
$w_1(A); l_1(B); u_1(A)$		125	
	$l_2(A); r_2(A)$		
	$A \leftarrow A * 2$		
	$w_2(A)$	250	
	$l_2(B)$ **denied**		
$r_1(B); B \leftarrow B + 100$			
$w_1(B); u_1(B)$			125
	$l_2(B); u_2(A); r_2(B)$		
	$B \leftarrow B * 2$		
	$w_2(B); u_2(B)$		250

Why 2PL works

Assume that S is a legal 2PL schedule

- We show how to convert S to a conflict-equivalent serial schedule
- Induction on n, the number of transactions in S
- We use the lock and unlock operations for the proof, but show how to arrange only the reads and writes in a serial order (we can then easily restore the locks and unlocks)
- $n = 1$: S is already a serial schedule

Inductive proof

Assume the result holds for schedules with $n-1$ transactions

- Let T_i be the transaction with the first unlock action in the schedule, say $u_i(X)$
- We show that we can move all the actions of T_i to the beginning of the schedule without conflicts
- Consider some action of T_i, say $w_i(Y)$. Can it be preceded in S by some conflicting action, say $w_j(Y)$?
- If so, in order for the schedule to be legal, j must lock the item and i must then lock it
- So S must look like:

$$\cdots w_j(Y); \cdots ; u_j(Y); \cdots ; l_i(Y); \cdots ; w_i(Y); \cdots$$

Proof, continued

- But $u_i(X)$ is the first unlock operation, by definition
- So it must be before $u_j(Y)$
- S therefore looks like

$$\cdots w_j(Y); \cdots ; u_i(X); \ldots ; u_j(Y); \cdots ; l_i(Y); \cdots ; w_i(Y); \cdots$$

- But then $u_i(X)$ appears before $l_i(Y)$ contrary to the 2PL protocol
- We can therefore rewrite S as

  (Actions of T_i)(Actions of the other $n-1$ transactions)

- The result now follows from the induction hypothesis

Deadlock

Deadlock is studied in operating system courses and the same techniques apply for databases. We just show that even with 2PL, deadlocks are possible

Consider the transactions

- T_1: $l_1(A); r_1(A); A := A+100; w_1(A); l_1(B); u_1(A); r_1(B); B := B+100; w_1(B); u_1(B)$
- T_2: $l_2(B); r_2(B); B := B * 2; w_2(B); l_2(A); u_2(B); r_2(A); A := A * 2; w_2(A); u_2(A);$

Deadlock

T_1	T_2	A	B
		25	25
$l_1(A); r_(A)$			
	$l_2(B); r_2(B)$		
$A \leftarrow A + 100$			
	$B \leftarrow B * 2$		
$w_1(A)$		125	
	$w_2(B)$		50
$l_1(B)$ **Denied**			
	$l_2(A)$ **Denied**		

Lock modes

- Our system is too simple
- T must lock a database element even if it only wants to read it
- This is necessary, because some other transaction may want to write it
- But there's no problem with more than one transaction reading the object at the same time
- We study several extended locking schemes, with different types of locks
- Our first model: Two different kinds of locks
- One for reading ("shared" or "read" lock)
- One for writing ("exclusive" or "write" lock)

Shared and Exclusive

- Read lock: allows us to read (other transactions can read, i.e., can have read locks, but cannot write)
- Write lock: we (and nobody else) is allowed to write (and to read)
- Notation:
  - *sl*$_j(X)$: T_j requests a shared lock on X
  - *xl*$_j(X)$: T_j requests an exclusive lock on X
  - $u_i(X)$: T_i releases the lock on X (doesn't matter what kind, as T_i can only have one)
- We now extend 2PL, consistency and legality to these type of locks

Consistency and 2PL

- Consistency: T_i may not write without holding an exclusive lock, and may not read without holding some lock
  - A read action $r_i(X)$ must be preceded by *sl*$_i(X)$ or by *xl*$_i(X)$, with no intervening $u_i(X)$
  - A write action $w_i(X)$ must be preceded by *xl*$_i(X)$, with no intervening $u_i(X)$
  - All locks must be followed by an unlock of the same element
- Two-phase locking: No action *sl*$_i(X)$ or *xl*$_i(X)$ can be preceded by any action $u_i(Y)$

Legality of schedules

- An object may either be (a) locked exclusively by one transaction or (b) by several in shared mode, but not both. Formally,
  - If $xl_i(X)$ is in a schedule, it cannot be followed by $xl_j(X)$ or by $sl_j(X)$, $j \neq i$, without an intervening $u_i(X)$
  - If $sl_i(X)$ is in a schedule, it cannot be followed by $xl_j(X)$, $j \neq i$, without an intervening $u_i(X)$
- A transaction can request and hold both shared and exclusive locks on the same element
- If the transaction knows its needs in advance, it would only need to request the exclusive lock

Example

$$T_1 \quad : \quad \textsf{\textit{sl}}_1(A); r_1(A); \textsf{\textit{xl}}_1(B); r_1(B); w_1(B); u_1(A); u_1(B)$$
$$T_2 \quad : \quad \textsf{\textit{sl}}_2(A); r_2(A); \textsf{\textit{sl}}_2(B); r_2(B); u_2(A); u_2(B)$$

T_1 and T_2 read A and B, but only T_1 writes B. Neither writes A

T_1	T_2
$\textsf{\textit{sl}}_1(A); r_1(A);$	
	$\textsf{\textit{sl}}_2(A); r_2(A);$
	$\textsf{\textit{sl}}_2(B); r_2(B);$
$\textsf{\textit{xl}}_1(B)$ **Denied**	
	$u_2(A); u_2(B)$
$\textsf{\textit{xl}}_1(B); r_1(B); w_1(B);$	
$u_1(A); u_1(B)$	

- T_1 gets a shared lock on A
- T_2 gets shared locks on both A and B
- T_1 now needs an exclusive lock on B
- But T_2 already has a shared lock on B
- The scheduler therefore forces T_1 to wait
- Once T_2 releases the lock, T_1 can complete

The schedule is conflict-serializable, with the order (T_2, T_1) even though T_1 started first

The proof that 2PL guarantees conflict-serializability is the same

Compatibility Matrices

- The specification for consistency and legality is quite complicated, and becomes even more complicated if we allow more types of locks
- *Compatibility matrices* are a simpler way to specify such rules
- Example: Matrix for shared and exclusive locks

		Lock requested	
		S	X
Lock held	S	Yes	No
in mode	X	No	No

- We can grant the lock on X in a mode only if the row for that mode has a "Yes" in the appropriate column

Upgrading Locks

- A transaction that only wants to read should take shared lock to allow more concurrency
- What if a transaction wants to read, and, later on, to write?
- The transaction can take a shared lock, and, later, upgarde the lock to an exclusive one
- This means later issuing a request for an exclusive lock (so that it holds both types of locks)
- We write $u_i(X)$ to mean release all locks on X help by the transaction

Example

Same example, but with different locks for A

$$
\begin{aligned}
T_1 &: \textsf{\textit{sl}}_1(A); r_1(A); \textsf{\textit{sl}}_1(B); r_1(B); \textsf{\textit{xl}}_1(B); w_1(B); u_1(A); u_1(B) \\
T_2 &: \textsf{\textit{sl}}_2(A); r_2(A); \textsf{\textit{sl}}_2(B); r_2(B); u_2(A); u_2(B)
\end{aligned}
$$

This allows more concurrency. T_1 is still blocked, but at a later point

T_1	T_2
$\textsf{\textit{sl}}_1(A); r_1(A);$	
	$\textsf{\textit{sl}}_2(A); r_2(A);$
	$\textsf{\textit{sl}}_2(B); r_2(B);$
$\textsf{\textit{sl}}_1(B); r_1(B);$	
$\textsf{\textit{xl}}_1(B)$ **Denied**	
	$u_2(A); u_2(B)$
$\textsf{\textit{xl}}_1(B); w_1(B);$	
$u_1(A); u_1(B)$	

Problems with upgrading

- Use of upgrading makes deadlock a serious problem
- Simple example (ignoring details of reads and writes) where both transactions get a read lock and then upgrade

T_1	T_2
$sl_1(A)$	
	$sl_2(A)$
$xl_1(A)$ **Denied**	
	$xl_2(A)$ **Denied**

Update locks

- Solution: A third lock mode, *update locks*
- *ul*$_i(X)$ lets T_i read X
- But the update lock can be upgraded later to a write lock; a read lock cannot
- An update lock on X can be granted when there are shared locks on X, but not if any transaction has a lock of any other kind
- Furthermore, once a transaction has once an update lock on X, no other transaction can get *any* lock on X

Compatibility matrix

Compatibility matrix with shared (S), exclusive (X) and update (U) lock modes

		Lock requested		
		S	X	U
Lock held	S	Yes	No	Yes
in mode	X	No	No	No
	U	No	No	No

Example

$$\begin{aligned} T_1 &: \mathit{ul}_1(A); r_1(A); \mathit{xl}_1(A); w_1(A); u_1(A) \\ T_2 &: \mathit{ul}_2(A); r_2(A); \mathit{xl}_2(A); w_2(A); u_2(A) \end{aligned}$$

Execution:

T_1	T_2
$\mathit{ul}_1(A); r_1(A)$	
	$\mathit{ul}_2(A)$ **Denied**
$\mathit{xl}_1(A); w_1(A); u_1(A)$	
	$\mathit{ul}_2(A); r_2(A)$
	$\mathit{xl}_2(A); w_2(A); u_2(A)$

Result: In this example, we allow almost no concurrency (but concurrency would result in deadlock)

Increment locks

- Many transactions only by increment or decrement stored values
- Example: Bank transfer
- Increment actions commute with each other
- Do not commute with reading or writing
- We have a new type of action, called increment: *inc*(A, c)
- Abbrevation of: $r(A, t); t := t + c; w(A, t)$
- But *inc*(A, c) is interpreted as an atomic operation, and the transaction does not see the value of A
- The fact that the transaction does not know the value of A (before or after) makes it easier to guarantee serializability

Increment locks

- *il*$_i(X)$: T_i requests an increment lock on X
- *inc*$_i(X)$: T_i increments X (for our purposes, the amount doesn't matter)
- Rules:
  - T_i must have an increment lock on X in order to do *inc*$_i(X)$ (nothing stops T_i from getting a write lock and doing the increment explicitly, though it's better to avoid when possible)
  - An increment lock does *not* allow either reads or any other type of write
  - Any number of transactions can hold an increment lock on X at any time
  - For $i \neq j$, *inc*$_i(X)$ conflicts with $r_j(X)$ and $w_j(X)$, but not with *inc*$_j(X)$

Example

$$T_1 \;:\; \textit{sl}_1(A); r_1(A); \textit{il}_1(B); \textit{inc}_1(B); u_1(A); u_1(B)$$
$$T_2 \;:\; \textit{sl}_2(A); r_2(A); \textit{il}_2(B); \textit{inc}_2(B); u_2(A); u_2(B)$$

Scheduler does not have to delay requests

T_1	T_2
$\textit{sl}_1(A); r_1(A);$	
	$\textit{sl}_2(A); r_2(A);$
	$\textit{il}_2(B); \textit{inc}_2(B);$
$\textit{il}_1(B); \textit{inc}_1(B);$	
	$u_2(A); u_2(B)$
$u_1(A); u_1(B)$	

This means that the schedule

$$r_1(A); r_2(A); \textit{inc}_2(B); \textit{inc}_1(B)$$

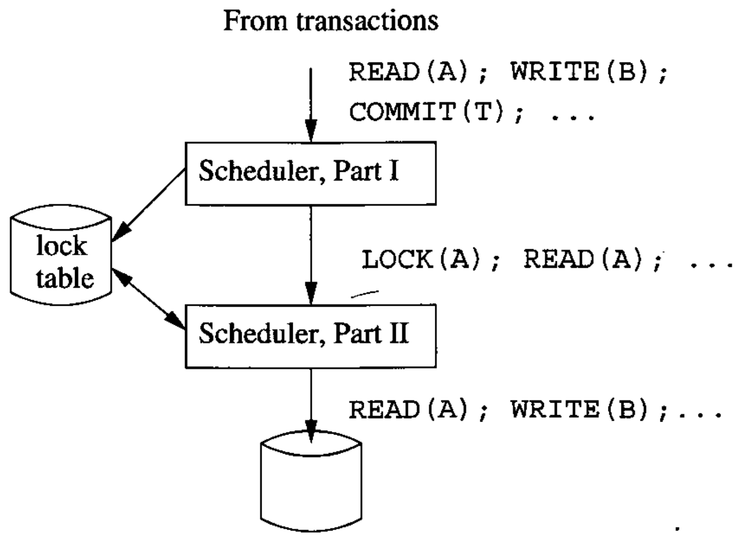
is conflict-equivalent to

$$r_1(A); \textit{inc}_1(B); r_2(A); \textit{inc}_2(B)$$

Architecture of locking scheduler

In our architecture

- The transactions do not request or release locks (or cannot be relied upon to do so)
- The scheduler inserts lock and unlock actions into the stream of read and write actions that it receives



- Scheduler gets requests (e.g., read, write, commit, and abort) from transactions
- Scheduler uses lock table
- Actions are usually transmitted through the scheduler and executed
- In some cases a transaction is delayed, waiting for a lock, and its requests are not yet transmitted to the database

- Part I: takes the stream of requests and inserts appropriate lock actions ahead of all database-access operations
- Part II: takes the sequence of lock and database-access actions it receives, and executes them appropriately:
  - Determines whether the transaction is already delayed: In this case it delays this action as well
  - If the transaction is not delayed:
    - Action is a database access: send to database
    - Action is a lock action: Examine the lock table to see if the lock can be granted.
      If so, modify the lock table
      If not, add to the lock table the fact that a lock has been requested, and delay the transaction
- Transaction commits
  - Part I releases the locks. If transactions are waiting for any of these, notifies Part II
  - Part II notified that X is available. Determine which transaction can be granted a lock on X and restart this transaction

Example

- Only one type of lock: Part I is simple
- Several types of locks: Scheduler needs advance notive of future requests
- For example, an SQL query does not write. For an SQL update, the processor can determine which elements will be written
- T_1: $r_1(A); r_1(B); w_1(B)$
- T_2: $r_2(A); r_2(B)$
- Actions arrive in order:

$$r_1(A); r_2(A); r_2(B); r_1(B); w_1(B)$$

- Locks used: S, U, X

Part I

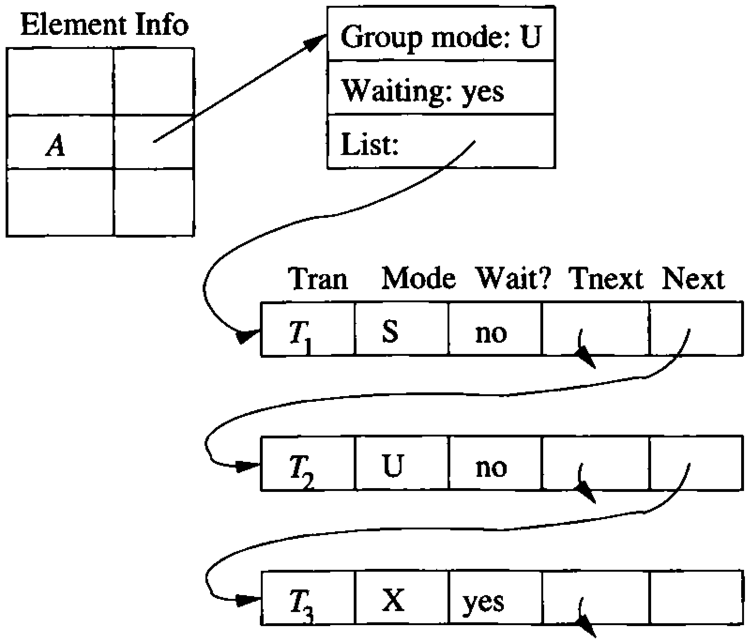
- $r_1(A)$. Scheduler inserts $sl_1(A)$ before it
- $r_2(A)$. Scheduler inserts $sl_2(A)$ before it
- $r_2(B)$. Scheduler inserts $sl_2(B)$ before it
- $r_1(B)$. Scheduler knows that a write follows, so inserts $ul_2(B)$ before it
- $w_1(B)$. Scheduler inserts $xl_1(B)$ before it

Unlock statements must also be added at the end

Part II:

- $sl_1(A)$. Check lock table. No lock on A, so add shared-lock information to A
- $sl_2(A)$. Check lock table and then compatibility matrix. Grant lock and add to lock table the fact that this transaction also has a shared lock on A
- $ul_2(B)$. Grant lock, according to compatibility matrix
- $xl_1(B)$. Cannot be granted.
- Transaction 2 commits. Lock is released and scheduler determines that transaction 1 is waiting, so restarts it

Lock table



We always assume S, X, U lock modes

Lock table contains one row for each element

Lock table

- Group mode: Strongest type of lock held on A:
  - S: only shared locks
  - U: one update lock and maybe some shared locks
  - X: one exclusive lock and nothing else
- Waiting: One bit saying whether transactions are waiting for a lock
- List of transactions that hold or a waiting for a lock on A. List may include
  - Name of transaction
  - Lock mode
  - Whether transaction holds or is waiting for this lock

Handling lock requests

T requests a lock on A:

- No lock table entry for A. Create the entry and grant the request
- Lock-table entry for A exist: Find the group mode
  - U or X: Deny the request, and add T to the list with the lock mode requested, "wait" equal to "yes"
  - S: Grant the request, change the mode to that requested, and add T to the list with "wait" equal to "no"

Handling unlocks

T unlocks A:

- Delete the list entry for T
- If the lock held by T is different from the group mode, there is no need to change the group mode
- If the lock held by T is equal to the group mode, examine the list to see what the new group mode should be (if the list is empty, delete the row from the lock table)

- If the value of “waiting” is “yes”, we must decide which transactions (s) can proceed. There are several possibilities
  - First-come-first-served: Chose the request that has been waiting the longest. This guarantees no “starvation” where a transaction can wait forever for a lock
  - Priority to shared locks: Grant first all the shared locks waiting, otherwise grant one update lock, if there are any waiting and only then grant exclusive locks. (this may allow starvation, if a transaction is waiting for a U or X lock)
  - Priority to upgrading. If a transaction with a U lock is waiting for an upgrade it to an X lock, grant that. Otherwise, follow one of the other strategies above