>>

Lock-based Concurrency Control

^ >>

❖ Lock-based Concurrency Control

Requires read/write lock operations which act on database objects.

Synchronise access to shared DB objects via these rules:

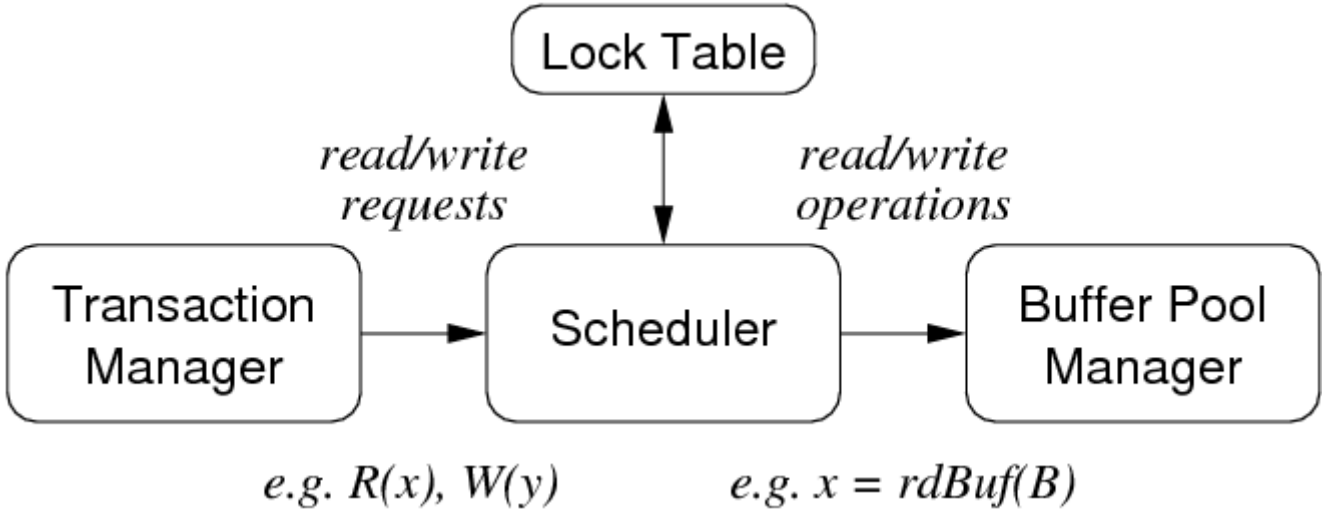
- before reading *X*, get read (shared) lock on *X*
- before writing *X*, get write (exclusive) lock on *X*
- a tx attempting to get a read lock on *X* is blocked
  if another tx already has write lock on *X*
- a tx attempting to get a write lock on *X* is blocked
  if another tx has any kind of lock on *X*

These rules alone do not guarantee serializability.

<< ^ >>

❖ Lock-based Concurrency Control (cont)

Locks introduce additional mechanisms in DBMS:



The Lock Manager manages the locks requested by the scheduler

<< ^ >>

❖ Lock-based Concurrency Control (cont)

Lock table entries contain:

- object being locked (DB, table, tuple, field)
- type of lock: read/shared, write/exclusive
- FIFO queue of tx's requesting this lock
- count of tx's currently holding lock (max 1 for write locks)

Lock and unlock operations *must* be atomic.

Lock upgrade:

- if a tx holds a read lock, and it is the only tx holding that lock
- then the lock can be converted into a write lock

<< ^ >>

❖ Lock-based Concurrency Control (cont)

Consider the following schedule, using locks:

```
T1(a): L_r(Y)      R(Y)                 continued
T2(a):       L_r(X)     R(X) U(X)       continued

T1(b):       U(Y)             L_w(X) W(X) U(X)
T2(b): L_w(Y)....W(Y) U(Y)
```

(where L_r = read-lock, L_w = write-lock, U = unlock)

Locks correctly ensure controlled access to **X** and **Y**.

Despite this, the schedule is not serializable. (Ex: prove this)

<< ^ >>

❖ Two-Phase Locking

To guarantee serializability, we require an additional constraint:

- in every tx, all *lock* requests precede all *unlock* requests

Each transaction is then structured as:

- growing phase where locks are acquired
- action phase where "real work" is done
- shrinking phase where locks are released

Clearly, this reduces potential concurrency ...

<< ^ >>

❖ Problems with Locking

Appropriate locking can guarantee serializability..

However, it also introduces potential undesirable effects:

- Deadlock
  - No transactions can proceed; each waiting on lock held by another.
- Starvation
  - One transaction is permanently "frozen out" of access to data.
- Reduced performance
  - Locking introduces delays while waiting for locks to be released.

<< ^ >>

❖ Deadlock

Deadlock occurs when two tx's wait for a lock on an item held by the other.

Example:

```
T1: L_w(A) R(A)                    L_w(B) ......
T2:               L_w(B) R(B)              L_w(A) .....
```

How to deal with deadlock?

- prevent it happening in the first place
- let it happen, detect it, recover from it

<< ^ >>

❖ Deadlock (cont)

Handling deadlock involves forcing a transaction to "back off"

- select process to roll back
  - choose on basis of how far tx has progressed, # locks held, ...
- roll back the selected process
  - how far does this it need to be rolled back?
  - worst-case scenario: abort one transaction, then retry
- prevent starvation
  - need methods to ensure that same tx isn't always chosen

<< ^ >>

❖ Deadlock (cont)

Methods for managing deadlock

- timeout : set max time limit for each tx
- waits-for graph : records T_j waiting on lock held by T_k
  - *prevent* deadlock by checking for new cycle ⇒ abort T_i
  - *detect* deadlock by periodic check for cycles ⇒ abort T_i
- timestamps : use tx start times as basis for priority
  - scenario: T_j tries to get lock held by T_k ...
  - wait-die: if $T_j < T_k$, then T_j waits, else T_j rolls back
  - wound-wait: if $T_j < T_k$, then T_k rolls back, else T_j waits

<< ^

❖ Deadlock (cont)

Properties of deadlock handling methods:

- both wait-die and wound-wait are fair
- wait-die tends to
  - roll back tx's that have done little work
  - but rolls back tx's more often
- wound-wait tends to
  - roll back tx's that may have done significant work
  - but rolls back tx's less often
- timestamps easier to implement than waits-for graph
- waits-for minimises roll backs because of deadlock

Produced: 14 Apr 2021