Concurrency Control

Chapter 18.1, 18.2, 18.5, 18.7

Locks

- A lock is associated with each (lockable) item in the DB
  - describes status of that item

Main idea: transaction must lock an item before using it.

- Con: added overhead; decreased parallelism
- Pro: used properly, locks guarantee that schedules are serializable

Binary Locks

- Binary lock: statuses are "locked" and "unlocked"
- Rules for use (just common sense, or common courtesy):
  - T must lock item before reading or writing it.
  - T must unlock item when finished with it.
  - T will not try to lock an item it's already locked.
  - T will not try to unlock an item it doesn't already have locked.
- Implementation: Queue of waiting T's.

Multimode Locks

- Only slightly more complicated
- Three states:
  - read-locked ("shared-locked")
  - write-locked ("exclusive-locked)
  - unlocked
- Multiple T's can hold the item read-locked, only one can write-lock it.
- Rules of common courtesy similar to binary locks

Are Locks Enough?

- ☹ Sadly… no.
- Locks alone do not guarantee serializability.
- ☺ Happily… there is a simple protocol (2PL) which uses locks to guarantee serializability.
  - Easy to explain
  - Easy to implement
  - Easy to enforce

The Two-Phase Locking Protocol

T is said to follow the 2PL if all lock operations precede the first unlock.

- T must also follow the courtesy rules, of course
- Such a T has two phases:
  - Expanding: locks but no unlocks
  - Shrinking: unlocks but no locks

2PL is Good Stuff

- 2PL is widely used
  - Can be adapted to variety of situations, such as distributed processing
- Variations:
  - Basic 2PL (as described)
  - Conservative 2PL: T locks all items before execution
  - Strict 2PL: T doesn't unlock any items until after COMMIT or ABORT.

But is it the last word?

Is there a snake in the Garden of Eden?

- 2PL reduces concurrency ☹
  - Conservative and strict variants reduce it even more, because they hold locks longer.
- Basic and strict 2PL are both subject to deadlock ☹

Deadlock

- *Deadlock* occurs when two transactions are each waiting for the other
  - e.g., each waiting for a lock that the other holds
  - can also be deadlocks in larger circles of T's
- 2PL is not deadlock-free
- Two possible approaches:
  - Deadlock *prevention* (don't let it happen)
  - Deadlock *detection* (notice when it's happened and take recovery action)

Deadlock Prevention Using Timestamps

- *Transaction timestamps*: unique numerical identifiers, same order as the starting order of the T's.
  - Notation: $TS(T_i)$
  - Not necessarily a system clock value
  - Could be simply integer++ for each T
- Interesting factoid: global timestamp techniques can be used in distributed systems, even when each system has its own clock.

Two TS Schemes

- Suppose T_j has locked X, and now T_i wants to lock X.
- ***Wait-die***:

  if $TS(T_i)<TS(T_j)$ then T_i is allowed to wait

  else T_i dies and is restarted with its same TS.
- ***Wound-wait***:

  if $TS(T_i)<TS(T_j)$ then abort T_j and restart with its same TS

  else Ti is allowed to wait.

TS Schemes

- In both schemes, the older T has a certain preferential treatment.
  - An aborted and restarted T always keeps it original TS.
  - Thus it gets older with respect to the other Ts, so eventually it gets the preference.

Deadlock Prevention Without Timestamps

- ***No waiting***: If T needs X and X is already locked, immediately abort T, restart later.
  - May cause lots of restarting
- ***Cautious waiting***: Suppose T_i needs X, and X is already locked by T_j.

  If T_j is waiting for anything, then abort T_j
  else abort T_i.
- ***Timeout***: If T waits longer than some fixed time, abort and restart T.

11/13/97 N-13

Deadlock Detection

- Simple idea:
  - Do nothing to prevent deadlocks
  - Make a check periodically to see if there are deadlock; if so, chose victim(s) to abort
- Pro: no elaborate protocols needed
- Con: deadlock might go undetected for a while
- Deadlocks tend to be rare on lightly loaded systems, frequent on heavily loaded systems

11/13/97 N-14

Detection Algorithm

- "Wait-for" graph:
  - One node per T
  - Directed edge $T_i \rightarrow T_j$ if T_i wants to lock X which is already locked by T_j.

Theorem: There is a deadlock iff the wait-for graph has a cycle.

- Issues
  - When to run the algorithm
  - How to select the victims

11/13/97 N-15

Starvation and Livelock

- Another snake in the Garden of Eden
- A T might be aborted and restarted indefinitely: *Starvation*
- A T might wait indefinitely, not for the same other T (deadlock), but for different reasons at different times: *Livelock*
- Footnote: Some people use "starvation" for either situation.

11/13/97 N-16

Concurrency Control Using Timestamp Ordering

Main idea: insure that conflicting operations occur in timestamp order

- Each item X must have a read TS and a write TS
  - = TS's of the T which last read or wrote X
- If T wants to use X, then X's TSs are checked. For a conflicting operation, TS(T) must be later. If not, T is aborted and restarted with a new (later) TS.

11/13/97 N-17

TO Properties

- Guarantees serializability
  - in fact, guarantees conflict serializability
- Deadlock free
- Not starvation free ☹
  - You *hope* that eventually T becomes "late enough" to slide through, but that might never happen.

11/13/97 N-18