CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

CONCURRENCY CONTROL

Lecture D

PESSIMISTIC CONCURRENCY

Two Approaches

- Preventing isolation from being violated can be done in two ways
  1. Pessimistic concurrency control
  2. Optimistic concurrency control

Pessimistic vs. Optimistic

- Pessimistic: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking
- Optimistic: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time, multi-version approaches

Pessimistic: Exclusive Locking

- Each object has a lock
- At most one transaction can be inside lock
- Before reading or writing object O, transaction T must call lock(O)
  - Blocks if another transaction already inside lock
- After entering lock T can read and write O multiple times
- When done (or at commit point), T calls unlock(O)
  - If other transactions waiting at lock(O), allows one of them in
- Sound familiar? (This is mutual exclusion!)

Can We Improve Concurrency?

- More concurrency => more transactions per second => more revenue ($$$)
- Real-life workloads have a lot of read-only or read-mostly transactions
  - Exclusive locking reduces concurrency
  - Hint: Ok to allow two transactions to concurrently read an object, since read-read is not a conflicting pair

Another Approach: Read-Write Locks

- Each object has a lock that can be held in one of <u>two modes</u>
  - Read mode: multiple transactions allowed in
  - Write mode: exclusive lock
- Before first reading O, transaction T calls read_lock(O)
  - T allowed in only if *all* transactions inside lock for O all entered via read mode
  - Not allowed if *any* transaction inside lock for O entered via write mode

Read-Write Locks (2)

- Before first writing O, call write_lock(O)
  - Allowed in only if no other transaction inside lock
- If T already holds read_lock(O), and wants to write, call write_lock(O) to *promote* lock from read to write mode
  - Succeeds only if no other transactions in write mode or read mode
  - Otherwise, T blocks
- Unlock(O) called by transaction T releases any lock on O by T

Guaranteeing Serial Equivalence with Locks

- Two-phase locking
  - A transaction cannot acquire (or promote) any locks after it has started releasing locks
  - Transaction has two phases
    1. Growing phase: only acquires or promotes locks
    2. Shrinking phase: only releases locks
       - Strict two phase locking: releases locks only at commit point

Why Two-phase Locking => Serial Equivalence?

- Proof by contradiction
- Assume two-phase locking system where serial equivalence is violated for some two transactions T1, T2
- Two facts must then be true:
  - (*A*) For some object O1, there were conflicting operations in T1 and T2 such that the time ordering pair is (T1, T2)
  - (*B*) For some object O2, the conflicting operation pair is (T2, T1)
  - (*A*) => T1 released O1's lock and T2 acquired it after that

    => T1's shrinking phase is before or overlaps with T2's growing phase
- Similarly, *(B)* => T2's shrinking phase is before or overlaps with T1's growing phase
- But both these cannot be true!

Downside of Locking

- Deadlocks!

Downside of Locking – Deadlocks!

<u>Transaction T1</u>

Lock(ABC123);

x = write(10, ABC123);

Lock(ABC789);

// Blocks waiting for T2

…

<u>Transaction T2</u>

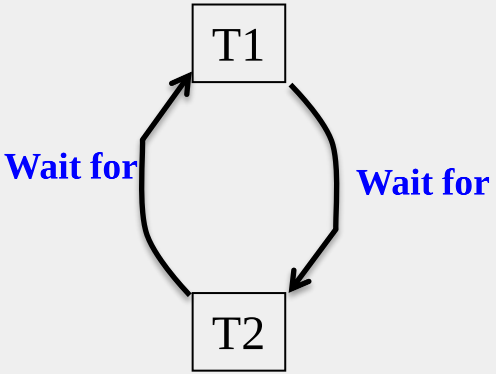
Lock(ABC789);

y = write(15, ABC789);

Lock(ABC123);

// Blocks waiting for T1

…



When Do Deadlocks Occur?

- 3 <u>necessary</u> conditions for a deadlock to occur
  1. Some objects are accessed in exclusive lock modes
  2. Transactions holding locks cannot be preempted
  3. There is a circular wait (cycle) in the Wait-for graph
- "Necessary" = if there's a deadlock, these conditions are all definitely true
- (Conditions not sufficient: if they're present, it doesn't imply a deadlock is present.)

Combating Deadlocks

1. Lock timeout: abort transaction if lock cannot be acquired within timeout

   ☹ Expensive, wasted work

2. Deadlock Detection:

   –Keep track of Wait-for graph (e.g., via Global Snapshot algorithm), and

   –Find cycles in it (e.g., periodically)

   –If find cycle, there’s a deadlock => Abort one or more transactions to break cycle

   ☹ Still allows deadlocks to occur

Combating Deadlocks (2)

3. Deadlock Prevention

- Set up the system so one of the *necessary conditions* is violated
  1. *Some objects are accessed in exclusive lock modes*
     - Fix: Allow read-only access to objects
  2. *Transactions holding locks cannot be preempted*
     - Fix: Allow preemption of some transactions
  3. *There is a circular wait (cycle) in the Wait-for graph*
     - Fix: Lock all objects in the beginning; if fail any, abort transaction => No cycles in Wait-for graph

Next

- Can we allow more concurrency?
- Optimistic Concurrency Control