EEL-4736/5737
Principles of Computer System Design

Lecture Slides 22

Textbook Chapter 9

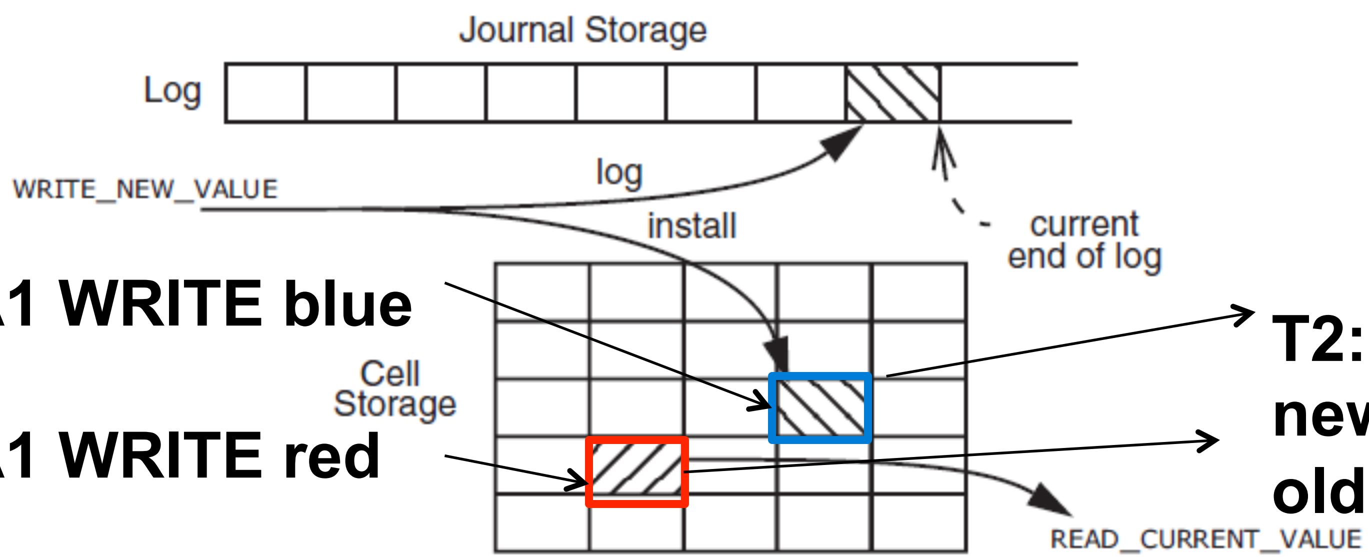
Atomicity Logs – Before-or-After and Locks

Introduction

- Before-or-after atomicity
  - Mark-point discipline
    - Commits to journal
    - Values not visible to other concurrent actions until committed
      - READ_CURRENT_VALUE gets data from journal when it is committed
  - Logs
    - Installs to cell storage
    - Cell storage updates are visible immediately to other threads
      - READ_CURRENT_VALUE gets data from cell storage when installed

Introduction

- Multiple threads installing to cell storage
  - A commit to the log is a single atomic operation – append
  - Installs can require multiple writes
  - Ok if single action; but concurrent actions, possible to see intermediate requests



T1: A1 WRITE blue

T3: A1 WRITE red

T2: A2 reads new blue, old red

System-wide lock

- Simple serialization
  - One lock used to serialize all actions
- Simple, correct – but conservative

Begin transaction

ACQUIRE(system-lock)

…

RELEASE(system-lock)

End transaction

Simple locking discipline

- Similar in spirit to mark-point discipline
- Rule #1:
  - Transaction must acquire a lock for *every* object it intends to *read or write, before* actually reading or writing
    - A “lock set”
- Rule #2:
  - It may release a lock only *after* it has installed its *last update* and committed (or completely restored its data and aborted)

Simple locking discipline

- Intuition:
  - Acquire locks of *all* objects before proceeding to *any* reads or writes
    - At this point, guaranteed that no other action will update any of these objects
  - Release *first* lock only after all updates are installed
    - Any other action that uses any of the objects in the "lock set" is blocked because it has not been able to acquire at least one lock
    - Hence, no other action using anything on lock set will "see" intermediate installs on cell storage – only after full set is installed

Simple locking discipline

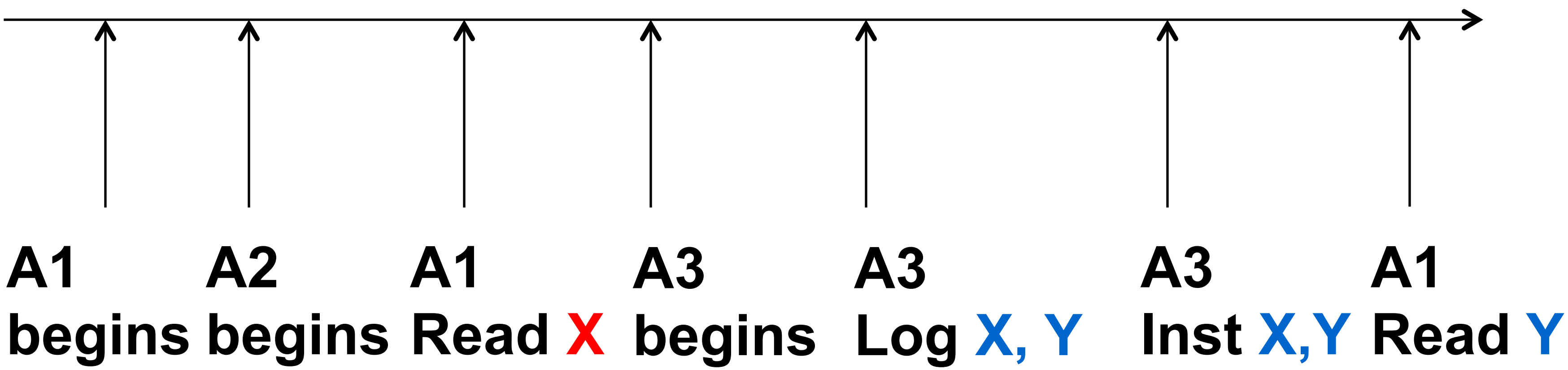
- A “lock manager” can expose lock set and enforce discipline without relying on application to write locks
  - Name all objects that will be used at the beginning of a transaction
  - Lock manager ensures begin_transaction starts by acquiring all locks in set (thus blocking until all acquired)
  - Use logging of END to release locks

Simple locking discipline

- Improved opportunities for concurrency compared to system-wide locking
  - E.g. consider transfer transactions working on completely disjoint sets of accounts
- However, there are still opportunities for concurrency that may be lost
  - E.g. consider two actions A1, A2 that read from the same object X, while no concurrent action intends to write to X
  - A1 or A2 will block waiting to acquire lock for X, but no real need for lock

Why locking of reads?

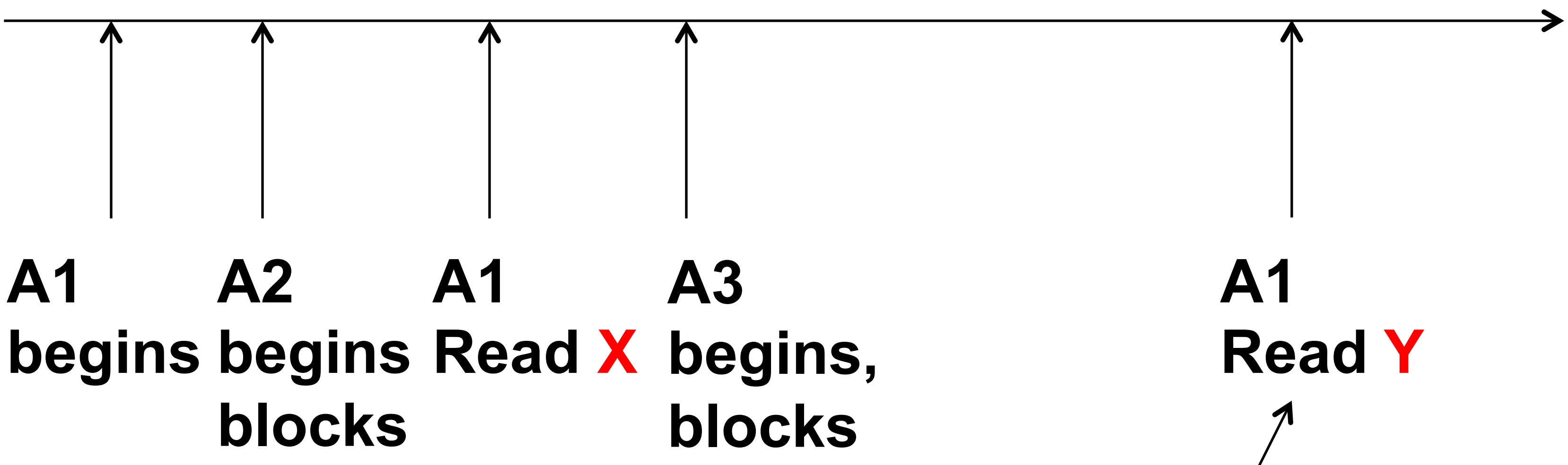
A1: read X, Y; write Z
A2: read X, Y; write W
A3: write X, Y



A1 reads “red” version of X, “blue” version of Y
Intermediate steps of A3 exposed

Why locking of reads?

A1: read X, Y; write Z
A2: read X, Y; write W
A3: write X, Y



Lock of X, Y blocks A2, A3 from reading/writing
A1 reads "red" version of Y

Simple locking discipline

- Need to know in advance all data that may be read or written so they can be locked
  - If not known in advance, it is a problem
  - What to read may depend on value that itself is read from the shared storage
  - E.g. consider traversing a pointer-based linked list – next pointer to read not known until current is read

Two-phase locking discipline

- Avoids requirement that transaction knows in advance which locks it needs to acquire
- Allows transaction to acquire locks as it progresses – before it uses an object
- May read/write an object as soon as it acquires its lock
- Primary constraint:
  - May only release *any locks* until it passes its *lock point*
    - First instance when it has acquired *all locks*

Two-phase locking discipline

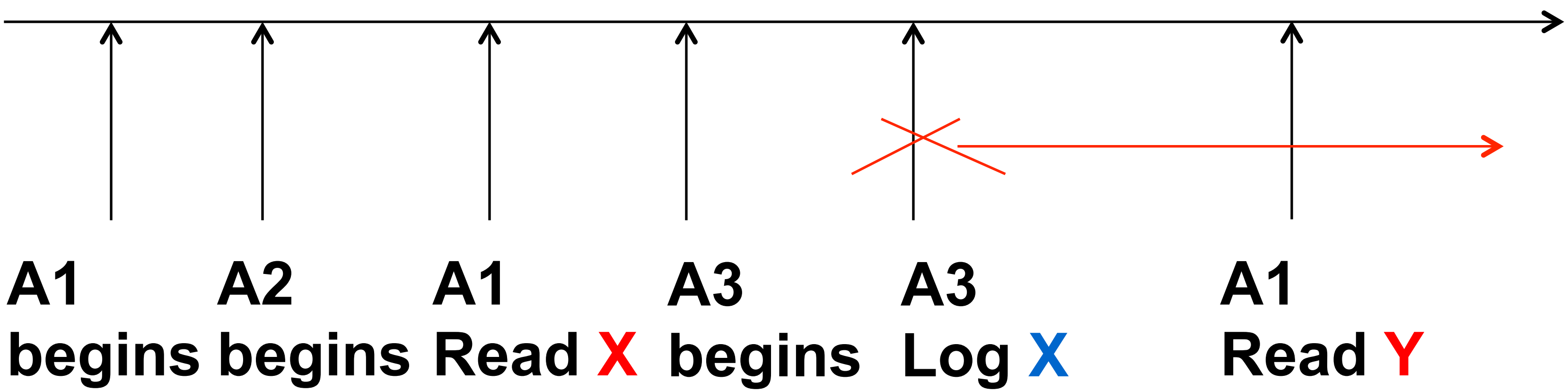
- Second constraint:
  - May only release a lock (after lock point) for an object that it only reads *if* it will never need to read the object again (even if it needs to abort)
- First phase
  - Number of locks monotonically increase as they are acquired until reaching lock point
- Second phase
  - Number of locks monotonically decrease as they are released

Two-phase locking discipline

- Lock manager can simplify use and enforce correctness
  - Interpose reads/writes
    - First use of any object: precede by a lock acquire
  - Interpose logging the end of transaction
    - Releases all locks

Two-phase locking

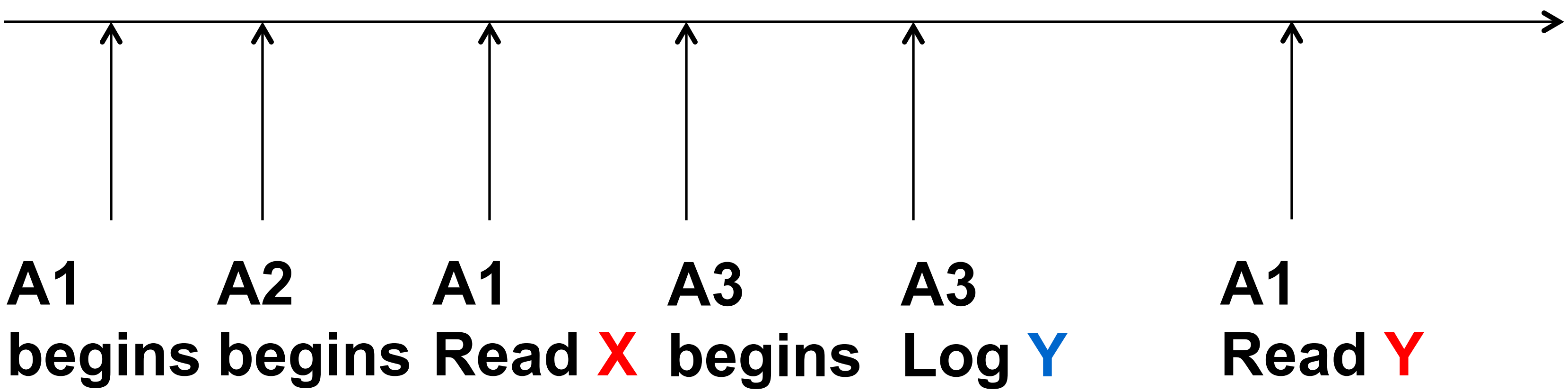
A1: read X, Y; write Z
A2: read X, Y; write W
A3: write X then Y



If A1 reads "red" version of X, it has locked X; Thus, A3 cannot write "blue" version of X until after A1 releases lock; A1 only releases lock after lock point, hence it also has a lock for Y – so it must Also read "red" Y

Two-phase locking

A1: read X, Y; write Z
A2: read X, Y; write W
A3: write Y then X



If A1 reads "red" version of X, it has locked X; A3 may log "blue" version of Y, but it cannot install Y and commit until it has reached its "lock point": it'd need X's lock to reach lock point (and X's lock is still held by A1)

Deadlocks can happen; detect, abort victim thread