Why do we lock access to data?

• Locking (leasing) provides mutual exclusion
  – Only one process at a time can access the data (or service)
• Allows us to achieve isolation
  – Other processes will not see or be able to access intermediate results
  – Important for consistency

Example:

```
Lock(table=checking_account, row=512348)
Lock(table=savings_account, row=512348)
checking_account.total = checking_account.total - 5000
savings_account.total = savings_account.total + 5000
Release(table=savings_account, row=512348)
Release(table=checking_account, row=512348)
```

Schedules

Transactions must be scheduled so that data is serially equivalent

How?
  – Use mutual exclusion to ensure that only one transaction executes at a time or...
  – Allow multiple transactions to execute concurrently
    • but ensure serializability
      ⇒ concurrency control

schedule: valid order of interleaving

Two-Phase Locking (2PL)

• Transactions run concurrently until they compete for the same resource
  – Only one will get to go … others must wait
• Grab exclusive locks on a resource
  – Lock data that is used by the transaction (e.g., fields in a DB, parts of a file)
  – Lock manager = mutual exclusion service

• Two-phase locking
  – phase 1: growing phase: acquire locks
  – phase 2: shrinking phase: release locks

• Transaction is not allowed new locks after it has released a lock
  – This ensures serial ordering on resource access

Without 2-phase locking

With 2-phase locking
Strong Strict Two-Phase Locking (SS2PL)

Problem with two-phase locking
• If a transaction aborts
  – Any other transactions that have accessed data from released locks (uncommitted data) have to be aborted
  – Cascading aborts
  – Otherwise, serial order is violated
• Avoid this situation:
  – Transaction holds all locks until it commits or aborts
⇒ Strong strict two-phase locking

Increasing concurrency: locking granularity
• Typically there will be many objects in a system
  – A typical transaction will access only a few of them (and is unlikely to clash with other transactions)
• Granularity of locking affects concurrency
  – Smaller amount locked → higher concurrency
• Example:
  – Lock an entire database vs. a table vs. a record in a table vs. a field in a record

Multiple readers/single writer
• Improve concurrency by supporting multiple readers
  – There is no problem with multiple transactions reading data from the same object
  – But only one transaction should be able to write to an object
  – and no other transactions should read that data
• Two types of locks: read locks and write locks
  – Set a read lock before doing a read on an object
  – Set a write lock before doing a write on an object
  – A write lock prevents others from reading or writing
  – Block (wait) if transaction cannot get the lock

Two-Version Based Concurrency Control
• A transaction can write tentative versions of objects
  – Others read from the original (previously-committed) version
• Read operations wait only when another transaction is committing the same object
• Allows for more concurrency than read-write locks
  – Transactions with writes risk waiting or rejection at commit
  – Transactions cannot commit if other uncompleted transactions have read the objects and committed

Multiple readers/single writer
If a transaction has
• No locks for an object:
  – Other transactions may obtain a read or write lock
• A read lock for an object:
  – Other transactions may obtain a read lock but must wait for a write lock
• A write lock for an object:
  – Other transactions will have to wait for a read or a write lock
Two-Version Based Concurrency Control

Three types of locks:
1. read lock
2. write lock
3. commit lock
Transaction cannot get a read or write lock if there is a commit lock

When the transaction coordinator receives a request to commit
- Write locks convert to commit locks
- Read locks wait until the transactions that set these locks have completed and locks are released

Compare with read/write locks:
- Read operations are delayed only while transactions are being committed
- BUT read operations of one transaction can cause a delay in the committing of other transactions

Problems with locking

- Locks have an overhead: maintenance, checking
- Locks can result in deadlock
- Locks may reduce concurrency
  - Transactions hold the locks until the transaction commits (strong strict two-phase locking)
- But … If data is not locked
  - A transaction may see inconsistent results
  - Locking solves this problem … but incurs delays

Optimistic concurrency control

- In many applications the chance of two transactions accessing the same object is low
- Allow transactions to proceed without obtaining locks
- Check for conflicts at commit time
  - Check versions of objects against versions read at start
  - If there is a conflict then abort and restart some transaction
- Phases:
  - Working phase: write results to a private workspace
  - Validation phase: check if there’s a conflict with other transactions
  - Update phase: make tentative changes permanent

Timestamp ordering

- Assign unique timestamp to a transaction when it begins
- Each object two timestamps associated with it:
  - Read timestamp: updated when the object is read
  - Write timestamp: updated when the object is written
- Each transaction has a timestamp = start of transaction
- Good ordering:
  - Object’s read and write timestamps will be older than the current transaction if it wants to write an object
  - Object’s write timestamps will be older than the current transaction if it wants to read an object
- Abort and restart transaction for improper ordering

Multiversion Concurrency Control (MVCC)

- We can use timestamp ordering AND multiple versions of an object to achieve even greater concurrency
- When a transaction wants to modify data, it creates a new version
- Store multiple versions of each object

Multiversion Concurrency Control (MVCC)

- Snapshot isolation
  - Each transaction sees the versions of data in the state when the transaction started
  - Data is consistent for that point in time
- Timestamps
  - Similar to timestamp ordering:
    - Each instance of an object has associated timestamps
      - Read timestamp = when the object was last read
      - Write timestamp = when the object was last modified
    - Transaction timestamp = start of transaction
  - Reads never block but read a version < timestamp(transaction)
    - Writes cannot complete if there are active transactions with earlier read timestamps for the object
    - This means a later transaction is dependent on an earlier value of the object
    - The transaction will be aborted and restarted
- Old versions of objects will have to be cleaned up periodically
Leasing versus Locking

- Common approach:
  - Get a lock for exclusive access to a resource
- But locks are not fault-tolerant
  - What if the process that has the lock dies?
  - It's safer to use a lock that expires instead
  - Lease = lock with a time limit
- Lease time: trade-offs
  - Long leases with possibility of long wait after failure
  - Or short leases that need to be renewed frequently
- Danger of leases
  - Possible loss of transactional integrity

Hierarchical Leases

- For fault tolerance, leases should be granted by consensus
- But consensus protocols aren't super-efficient
- Compromise: use a hierarchy
  - Use consensus as an election algorithm to elect a coordinator
  - Coordinator is granted a lease on a large set of resources
    - Coarse-grained locking: large regions; long time periods
    - Coordinator hands out sub-leases on those resources
    - Fine-grained locking: small regions (objects); short time periods
- When the coordinator's lease expires
  - Consensus algorithm is run again

The end