Leasing versus Locking

- Common approach:
  - Get a lock for exclusive access to a resource
- But: locks are not fault-tolerant
- It’s safer to use a lock that expires instead
  - Lease – lock with a time limit
- Trade-off
  - Long leases with possibility of long wait after failure
  - Or short leases that need to be renewed frequently

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
  - 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

Why do we lock access?

- 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

Example:

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

Schedules

- Transactions must have scheduled so that data is serially equivalent
- How?
  - Use mutual exclusion to ensure that only one transaction executes at a time
  - Allow multiple transactions to execute concurrently
    - but ensure serializability
    - concurrency control
- schedule: valid order of interleaving

Locking

- Serialize with exclusive locks on a resource
  - lock data that is used by the transaction (e.g., parts of a file)
  - lock manager
- Conflicting operations of two transactions must be executed in the same order
  - transaction not allowed new locks after it has released a lock
- Two-phase locking
  - phase 1: growing phase: acquire locks
  - phase 2: shrinking phase: release locks
- This ensures serial ordering on resource access
Strict 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
- Avoid this situation:
  - transaction holds all locks until it commits or aborts
- Strict two-phase locking

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

Multiple readers/single writer

- Improve concurrency by supporting multiple readers
  - there is no problem with multiple transactions reading data from the same object
  - only one transaction should be able to write to an object
  - and no other transactions should read that data
- Two locks: read locks and write locks
  - set a read lock before doing a read on an object
  - a write lock prevents writing
  - set a write lock before doing a write on an object
  - A write lock prevents reading and writing
  - block (wait) if transaction cannot get the lock

Multiple readers/single writer

If a transaction has
- no locks for an object:
  - another transaction may obtain a read or write lock
- a read lock for an object:
  - another transaction may obtain a read lock but must wait for a write lock
- a write lock for an object:
  - another transaction will have to wait for a read or a write lock

Increasing concurrency: two-version locking

- A transaction can write tentative versions of objects
  - Others read from the original (committed) version
- Read operations wait if another transaction is committing the same object
- Allows for more concurrency than read-write locks
  - Writing transactions risk waiting or rejection when the commit
  - Transactions cannot commit if other uncompleted transactions have read the objects
  - These transactions must wait until the reading transactions have committed

Two-version locking

- Three types of locks: read lock, write lock, 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 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 by having transactions hold the locks until the transaction commits (strict two-phase locking)

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
- 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

The End