Given techniques of chapter 19.

- Implement mechanism to control concurrent access to data.
- Lock - permission associated with data.
- Most restrictive - binary locks

**Binary locks**

- Two states - locked and unlocked
- Implements mutual exclusion (only one has access at any time.)
- Logged: \(<X, \text{LOCK}, T>\)
- Given item \(X\),
  - `lock_item(x)` - sets lock on \(x\) to 1
  - `unlock_item(x)` - sets lock on \(x\) to 0
  - `lock(x)` - returns 1 if locked, 0 if unlocked.

**Binary lock implementation**

```
Lock_item(x)
while (lock (x) == 0) wait;
Lock(x) <- 1
Unlock_item(x)
Lock(x) <- 0
Wake up any waiting transactions
```

**Lock processes**

1. \(T\) must `lock_item(x)` before read or write \(x\).
2. \(T\) must unlock\(x\) after read or write operations and before exiting \(T\).
3. \(T\) must `lock_item(x)` if it already has a lock on \(x\).
4. \(T\) must not `unlock_item(x)` if it didn’t lock \(x\).

**Generally**

Binary locks are too restrictive (limit amount of parallel access to data)

E.g. If majority of database accesses are reads, then binary locks on read-only data are not very useful.
Read/Write locks

- `read_lock (x)` - adds one to number of readers
- `write_lock (x)` - exclusive access to write a value
- `unlock(x)` - removes the lock

Entered in log as

<X, type-of-lock, number-of-reads, T>

R/W locks

Only need to keep lock information for existing locks (don’t need to keep history of locks.)

Rules for R/W locks

1. T must issue r or w lock before read of x.
2. T must issue w lock before writing x.
3. T must issue unlock after all R or W ops are done.
4. No double locking.
5. T cannot unlock(x) if it doesn’t have a lock on X.

R/W lock upgrades

If T is only process with read lock on X, it can upgrade to write lock on X.
Similarly
If T is only process with write lock on X, it can downgrade to read lock on X.

Locks are only part of the solution

Locks are the mechanism to help implement a serializable schedule.

Still need a protocol…

Two-phase locking

All read/write locking must occur before first unlocking in a transaction.

Phase 1. Growing or acquiring locks phase.
Phase 2. Shrinking or releasing lock phase.
Example transactions

T1: rl(y); r(y); ul(y); w(x); r(x); x := x + y; w(x); ul(x)

T1': rl(y); r(y); w(x); ul(y); r(x); x := x + y; w(x); ul(x)

More examples

T2: rl(x); r(x); ul(x); w(y); r(y); y := y + x; w(y); ul(y)

T2': rl(x); r(x); w(y); ul(x); r(y); y := y + x; w(y); ul(y)

Two-phase locking

Guarantees serializability.

Problems:
- Can still have deadlock.
- Limits amount of concurrency: T must lock all the variables it needs at some point before releasing any of them.

Variations of two phase locking

Basic 2PL - just covered.
Conservative 2PL - must predeclare read and write sets (locks vars before transaction starts).
Strict 2PL - write locks not released until commit or abort. Strict sched for recoverability.
Rigorous 2PL - same as strict + read locks.

Problems with locks

Deadlock - each transaction T in a set of two or more transactions is waiting for a resource locked by some other T' in the set.

Some simple protocols

Protocol 1: T must lock all X before starting transaction.
Problem: limits concurrency

Protocol 2: Well-known locking order for X
Problem: lots of overhead (programmer or system)
Timestamp-based methods

TS(T) - time stamp associated with T (when was T started).
Assuming if TS(Ti) < TS(Tj) then Ti was started before Tj. (Ti is older than Tj)

In all these cases, Ti trying to acquire a lock held by Tj.

Wait-die protocol

If TS(Ti)<TS(Tj)
   Ti waits for X
else
   Ti is killed and restarted later with same TS.

Older transactions can wait for a younger transactions.

Wound-wait protocol

If TS(Ti)<TS(Tj)
   Abort Tj and start Tj later with same TS.
Else
   Ti waits for X.

Older transaction causes younger transaction to abort and give up resource.

Two additional algorithms

No waiting algorithm - aborts if it can’t get a lock.

Cautious waiting: If Ti requires resource Tj is holding, Ti can wait as long as Tj is not blocked.

More practical approaches

Deadlock detection - construct a wait-for graph. If there is a cycle, must chose a victim to die.
Timeout - transaction is given a time quota, if time expires, abort and restart.

Another problem with concurrent systems

Starvation - some transactions cannot proceed while other transactions progress normally.

Possible places - victim selection always picks this transaction, priority weighting schemes.
Timestamp-based ordering

Transactions are scheduled based on timestamp order. (equivalent schedule)
For conflicting operations, maintain serializeability order.

Uses 2 timestamps for each X.
- Read_TS(x) - last timestamp to successfully read x
- Write_TS(x) - last TS(T) to successfully write X.

Basic Timestamp Ordering algo

(note: these are not-recoverable schedules)
May have cascading rollback.
1. Verify that TS(T) > write_TS(X) and TS(T) > read_TS(x), if not, abort and get T a new TS.
2. If T is read operation, if write_TS(x)>TS(t) then abort and rollback T.

Basic TO

Gives serializable schedules
- not the same as 2PL
- no algorithm gives all serializable schedules
Gives deadlock free schedules.
Problems with cyclic restarts and starvation.

Multi-version concurrency control

Maintains multiple versions of a tuples.
Tied into view serializeable techniques.
In extreme case, use temporal database which tracks all changes over time

Time stamp ordering for multi-version

Keep a read_TS and write_TS for each version of a tuple.

If T tries to read X, find the right version based on timestamps.

If T tries to write X, more complicated (need to verify that no one has read data could have read this data.)

Two phase locking for multi-version

Read, write and certify locks.

Allows T to read old data with no problem.

New data must go through two phase write/certification process.
Optimistic concurrency control

No check is done as transactions are processing.

All operations are performed on local copies of data.
When copies are written back, they are verified (make sure no serialization problems.)

Phases

1. Read data - into local space, mods are done on local vars.
2. Validate data - verify that serializability will not be violated if written back to DB.
3. Write data

Optimism - very few will need to be restarted or rolled back due to interference.

Optimistic CC uses

Time stamps
Read sets and write sets.

Verification algorithm
1. Tj finishes writing before T starts reading
2. T starts writing after Tj writes and T’s read set is independent of Tj’s write set.
3. T’s read and write set do not overlap Tj’s write set, and Tj’s reads are before Ts.

Granularity of locking

Database record
Field
Disk block
File
Whole DB.
Larger data (coarse granularity) - lower concurrency.
Smaller data (finer granularity) - more overhead for locks and time stamps.

Multiple granularity locking

Use intension locks
IS - intension shared lock
IX - intension exclusive lock
SIX - shared intension exclusive lock

Draw a typical locking structure

```
Db
  F1
  F2
  P11 p12 p1n  p21 p22 p2n
  R11...R11j
```

Show read all and write 1 r.
Multi-granularity locking protocol
1. Must adhere to 2PL.
2. Locking nodes must be compatible with parent nodes.
3. Root must be locked regardless.
4. Can only unlock a node if all children are unlocked.

Good for short transactions and transactions accessing entire file.

Locks for indices.
Don’t use raw 2PL. (holds too long till release phase.)

Use tree-based locking.
For reads, only lock leaf.
For write, use ix for path to leaf and x for leaf. (you can then do the split.)

B-link trees
Similar to B+ trees except internal nodes have pointers to siblings.
Allows searches to happen concurrently with inserts.

Other problems
Phantom records -
T inserts a new employee
T’ sums all the salaries of employee.

Logically these two transactions conflict, but they don’t share a common record (until T instantiates employee!)
Use index locking to solve these problems.