# Logistics

- Programming assignment 1: All graded but 2.
  - For next assignment.  Please turn in your files in a single archive (tar, zip, rar) that contains a directory with your last name, under which the sources and documents go.
  - So far, no problems.
- Today:
  - Onwards through transactions.
- Tomorrow:
  - Reminder: Deschutes rm. 200, 6pm.  Pizza.
    - Pizza preferences?  Any constraints?  (e.g.: Veg?)

# Abort

- The abort operation is the important thing that distinguishes a transaction from a critical section protected by mutual exclusion.
- Abort states that anything that a transaction has done up to the abort is not visible.  The transaction is to have effectively not happened from the perspective of other clients.
- The abort may require cleanup to occur.
- There are many reasons why an abort may occur.

# Reasons for an abort

- Crashes
  - Both server and client side.
- Deadlock prevention
  - Upon detection of deadlock, or potential deadlock, abort a transaction to un-wedge things.
- Internal logic
  - Transaction makes decision that it should not continue the transaction.

# Crashes

- Say a server crashes, and a new one is started in it's place.
  - Remember that we're assuming that we have a permanent, recoverable store on the server.
  - So, the new server can abort any outstanding open transactions that were around when the first server died.
    - Restore state of objects before the crash occurred. We assume that these were stashed away somewhere safe at the openTransaction() step.
- If a client crashes, the server can detect it (via a timeout or lease expiration) and abort the defunct transaction.

# Crashes

- What about the client perspective?
- Obviously nothing to do if the client crashed.
  - It's the server's problem.
- But if the server crashes, and a new one arrives, the server will have aborted the transaction.
- Client must be informed.
  - We assume that during a transaction, the client issues operations with an associated transaction ID.
  - The server could recognize this ID as defunct and issue an exception.
  - The client can act accordingly, such as starting at the openTransaction() step again.

# Correctness problems

- We can see concurrency-related correctness problems in different forms.
  - **Lost update**: An update is missed by interleaving a read/update/write across two processes such that one update is lost.
    - R1;R2;U1;U2;W1;W2 = Only update 2 would be committed!
  - **Inconsistent retrievals**: Retrieval of a partially updated value during a transaction.
    - Bank transfer example.
- So, we design transaction protocols to prevent these.

# What do we desire: serial equivalence

- Ultimately what we want is a set of concurrent operations that have exactly the same effect as if they had executed serially.
- This does **not** prohibit interleaving or concurrent execution though.

- It turns out only certain combinations of operations can cause a conflict that would result in a deviation from execution equivalent to the serial case.

# Read and Write: The troublemakers

- The operations that cause all of the trouble are those that access the store.
- Consider two threads and the three possible combinations of these operations.
- **Read/Read**: Clearly no problem. The data isn't modified so it doesn't matter if the two processes read it.
- **Read/Write**: These can conflict. The result of the read depends on it's relative position with the write.
- **Write/Write**: Similarly, the final value is dependent on which write occurs last.

# Conflicting operations

- Dirty reads
  - Reading the intermediate values of a transaction from the outside.
  - What if the transaction aborts?  Then the read value should have never existed.

- Premature writes
  - Writing to a value being used by another transaction.
  - What if the writer aborts?  Then it caused a perturbation that should never have existed.

# Methods for avoiding conflicts

- Locking
- Optimistic control
- Timestamp ordering

# Locking

- Elements that a transaction uses are locked as the transaction requires them.
- Other transactions will block if they try to access these elements while another transaction is executing.
- Locks removed when transaction completes.

- Potential for deadlock.
- Similar to basic semaphore or locking to protect critical sections, except lock acquisition and release is typically hidden by the transaction software layer.

# Optimistic control

- Run through the transaction without locking.
- At the end, check to see if any other transactions have accessed data in a conflicting way.
- If so, abort, clean up, and let the client restart.

- This type of scheme avoids locking overhead.
- Works best when conflicts aren't probable.
- Depending on how the intermediate transaction data is stored the mechanism to abort and retry may vary.

# Timestamp ordering

- Data is owned by a server.
- Server records access times (for both reads and writes) of objects that transactions touch.

- Use this to determine if an operation can occur right away, be delayed, or cause a problem resulting in an abort.
  - E.g.: A write occurs with a timestamp before the most recent write – abort because that means the transaction is out of date.

# Nested transactions

- Transactions within transactions.
- This would appear in a library-like context, where complex transactions are built out of simpler pieces that themselves use transactions.

- The key issue is how commits and aborts interact between the different transaction levels.

# Nested transactions

- A transaction may commit or abort ONLY after it's child transactions have completed.
- When subtransactions complete, they make an independent decision to commit provisionally or abort. Aborts are final.
- When a transaction aborts, all of it's subtransactions abort. Even those that have committed must abort.
  - Hence use of term "commit provisionally"
- When a subtransaction aborts, the parent may choose whether or not to abort. It is legal for the parent to commit in some cases.
  - E-mail example from book.

# Locking

- One approach to transaction implementation is to acquire a lock for any object used during a transaction.
- Exclusive locking allows a transaction to put a lock on a critical piece of data before it uses it, unlocking it when the transaction is done.
- This will force other transactions to block on accesses to this critical data if another transaction already has them.
  - This is not deadlock proof.

# Two-phase locking

- A disciplined way of maintaining serial equivalence.
- Locks are acquired but not released until the end of the transaction.
  - Growing and shrinking phases.
- If this was not true, then unintended interleavings of transactions could occur, violating serial equivalence.

  - *Reminder*: Serial equivalence means the transactions running concurrently behave as if they had been run in a serial fashion without concurrency.

# Read vs write locks

- Locks can be separated into classes with different rules.

- **Read lock**: Acquire a lock when attempting to read an object.  If no write lock exists for the object, other transactions can be granted a read lock too.
  - Shared lock.
- **Write lock**: Not shared.

# Read/write lock interactions

- Say a transaction has a read lock.  Then another transaction must wait until the first commits before acquiring a write lock on the same object.

- Similarly, assume a transaction has a write lock. Other transactions must wait until it commits before acquiring a read or write lock on the object.

# Exclusive locking vs read/write

- The reason for distinguishing between read vs write locks is to prevent unnecessary blocking.

- Say transactions will read lots of data objects, but modify only a small number of them.  Why block on harmless shared reads?

# Locks and nested transactions

- Subtransactions inherit locks from their parent.
- Parents inherit locks from children when they complete.
  - Why? Prevents partial results from being observable to the outside world.
  - Say a subtransaction modifies data that it's parent does not. If the lock was released when the subtransaction finished, the modification would be visible even though the full nested set of transactions wasn't done yet.
    - Bad.

# Deadlock

- Deadlock occurs when a set of processes each hold a partial set of locks and are each blocked pending the release of one of the locks by a peer.
- We mentioned earlier that the growing phase of a transaction can lead to deadlock.
  - E.g.: transaction 1 acquires lock for U and then V.  Transaction 2 acquires lock for V and then U.  Deadlock.

# Livelock

- Livelock is similar to deadlock in that processes in livelock cease to make progress.
- Deadlock is characterized by blocking and sitting.
- Livelock is characterized by constant execution with no tangible progress.
  - Example:
    - TestLock(A) > Lock(A) > TestLock(B) > Fail > Unlock(A) > Repeat
- Livelock is one reason why protocols with retries will sometimes force a random delay in when the retry starts again to try and break this sort of behavior.

# Prevention

- One method is to enumerate all objects to lock upon entering a transaction, and acquire all of the locks in a single atomic step before starting.
  - Atomicity of acquisition prevents deadlock-prone ordering.
- This will work, but may be too restrictive.
  - E.g.: If an access to an object occurs within a clause of an if-else block, and the clause is improbable, the lock will needlessly protect the data most of the time.

# Deadlock detection

- You can write down a directed graph of processes that are waiting for each other.
- Deadlock occurs when cycles exist in this graph.
- A lock manager can maintain this list of dependencies and check the graph for cycles each time a lock is requested.
- If a cycle is detected, it can break the deadlock situation by choosing a transaction and forcing it to abort.
  - Choice of which to abort is system dependent.
  - Could be age of transaction, could be time spent in blocked state, could be based on number of locks held, etc…

# Deadlock prevention

- Another method for deadlock prevention is to timeout when waiting to acquire a lock.
- If a transaction holds a lock for beyond some timeout period, the lock may be broken by other processes wishing to acquire it.
  - The transaction that owns the lock that was broken will be aborted.
- This isn't necessarily ideal.  Heavily loaded systems that cause transactions to run very slowly can cause a transaction to be aborted even though deadlock doesn't actually exist.

# More sophisticated schemes

- Novel schemes exist for locking that increase concurrency (reducing serialization) by introducing new types of locks.

- Two-version locking:
  - Read/write/commit lock.  Introduces this third type of lock.
- Hierarchic locks

# Two-version locking

- Read operations: read lock acquisition successful unless a commit lock exists on the object.
- Write operations: write lock acquisition successful unless a write or commit lock exists on the object.
  - Non-success in the above two cases means the transaction waits.
- Commit attempts to convert write locks into commit locks. Any outstanding reads will cause the committer to wait until they finish.
- Consequences:
  - Exclusive locking is restricted to the commit phase only. This can reduce waiting time experienced by other transactions.
  - Read operations can cause commit delays. This is not so good.

# Hierarchic locks

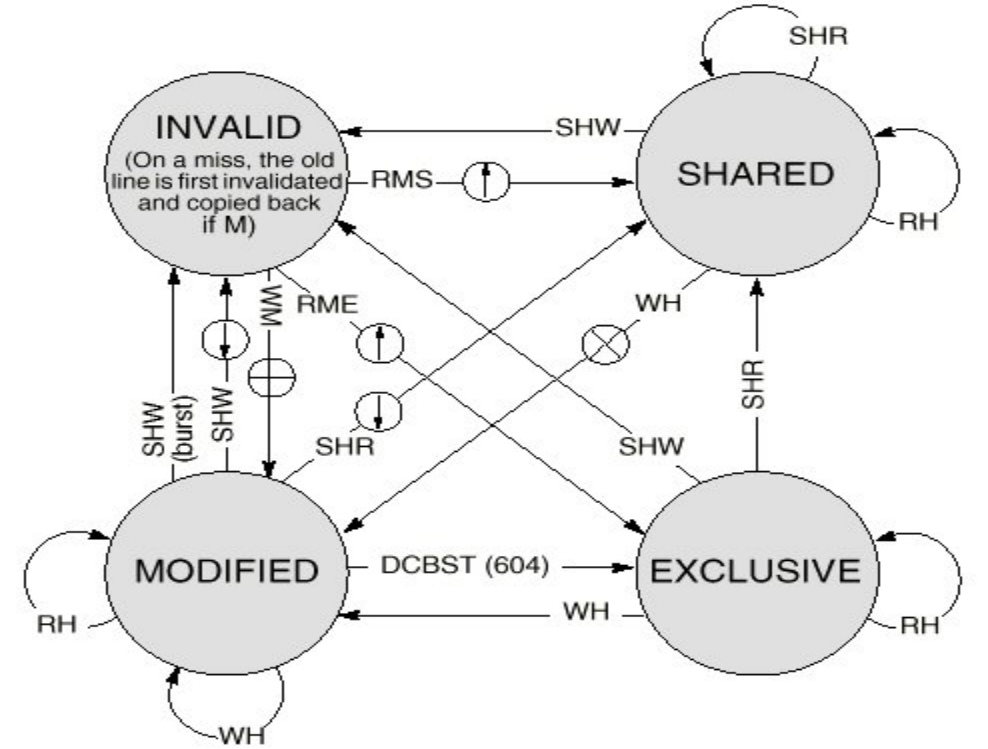
- Mixed granularity locking.
- Consider a tree relationship with the objects in a system:
  - E.g.: matrix contains rows contains elements.
- Lock acquisition at a point in the tree applies to all children.
- Gives transactions flexibility to only lock what they need, taking advantage of structure in the data.

# An aside. Coherence protocols.

- In transaction protocols, we see a set of rules that define how reads and writes interact with each other to maintain correctness amongst a set of interacting transactions.

- While not exactly the same, this always reminds me of the protocols implemented in hardware to ensure coherence in shared memory machines that contain caches.
  - The difference is that blocking doesn't occur – instead of blocking, we see cache invalidations and flushes to memory.

# Coherence protocol: MESI example




BUS TRANSACTIONS

RH = Read Hit
RMS = Read Miss, Shared
RME = Read Miss, Exclusive
WH = Write Hit
WM = Write Miss
SHR = Snoop Hit on a Read
SHW = Snoop Hit on a Write or Read-with-Intent-to-Modify

= Snoop Push
= Invalidate Transaction
= Read-with-Intent-to-Modify
= Cache Block Fill

# More CC methods

- Why do I bring these up?
- MESI looks like a transaction scheme, except instead of aborting, flushes occur.
  - Sort of like a conflict forcing a transaction to close immediately.
- Other schemes exist that aren't so strict.
  - MOESI (used in AMD multicore processors by the way)
  - "O"wned state: Allows caches to provide values.
    - Why is this relevant? It just shows that with more tedious and sophisticated protocols, we can get away with being a bit more flexible in the interest of performance.

# Optimistic concurrency control

- Take advantage of low probability of conflict.
  - In other words, the likelihood of two transactions operating at the same time on the same data is very low.
- Split transaction into three phases.
  - Working, validation, update.
- Key point: Avoid lock overhead in working phase by adding validation step.

# Three phases

- **Working phase**: Transaction runs as though no conflicts were occurring. Writes occur to a transaction-local store (not visible to outside world).
- **Validation phase**: When the transaction is done and issues a closeTransaction request, the system checks to see if any conflicts occurred.
  - If so, either abort or attempt to resolve conflicts.
- **Update phase**: Once successfully validated, the written data is committed.

# Timestamp ordering

- Operations validated when they occur.
  - If operation can't be validated, abort.
- Transactions assigned timestamps when they start.
  - Acquired from a single clock (e.g.: server) or globally consistent logical counter.
  - Allows for total order to be established, avoids skew/drift issues.
- Ordering rule:
  - A request to write is valid if the object to write was last read and written by earlier transactions. A request to read is valid only if that object was last written by an earlier transaction.

- See fig. 13.30 / 13.31 to see examples of valid orderings and orderings that result in an abort for both reads and writes.

# Software transactional memory

- Transactions at the code (or hardware) level.
- **Many** implementations arising out in the wild.
  - Some in the form of libaries
  - Some at the compiler level. E.g.: Intel has a “preview release” of their STM C++ compiler.
- The key is the designation (typically via keywords or annotations) of atomic blocks.
- How you deal with shared data varies between implementations.
  - Haskell “Tvars”
  - RSTM smart objects/smart pointers
  - Compiler analysis
  - Etc…

# Lock implementation

- Hardware support
  - Lock instructions, test-and-set instructions, archaic interlocked instructions.

- Software support
  - Dekker's algorithm (1964). Provides locking in a shared memory setting when all that is available is an atomic read and write.
    - Disadvantage: Busy waits, and can be broken in out-of-order CPUs if not careful. Aggressive compiler optimizations can also break it.
    - But, it is very portable.

# Distributed transactions

- Moving on to ch. 14, we look at how concepts related to transactions translate to the distributed context.

# Distributed transactions

- Key difference from basic transactions:
  - Objects involved in a transaction may be distributed amongst a set of servers.
- Commit of transaction requires set of servers to decide if they all commit or all abort.
- “Two-phase commit protocol”
  - One server elected to be coordinator.
  - Servers communicate to determine if abort or commit is performed.

# Reminder

- 6pm tomorrow.
- Topic:
  - General concurrency issues.
  - Thread programming in Java.
  - Possibly talk about consequences of multicore and SMPs in distributed systems.
    - Cross-box / in-box interactions and interference.
  - Language-level concurrency topics.
    - With an eye towards distributed systems.