## Transactions for Concurrency

1. Problem:
    a. Locks are hard to use, not always available
        i. Example: across system calls
2. Problem within an OS:
    a. Two classes of problems: in mutual exclusion
        i. Concurrency/isolation
            1. Want to hide updates until complete
        ii. Atomicity
            1. Want updates to persistent state to be complete or not happen, e.g. across a failure
    b. Atomic update to multiple files
        i. Add user to both /etc/shadow, /etc/passwd
        ii. If login between two changes, could get incorrect information
        iii. If system crashes between, left with inconsistent state that needs to be detected and repaired
    c. Atomic check permissions and open:
        i. Servers do setuid(user), access(filename), setuid(0), open() to check if a caller has access to a file
            1. Is possible to rename the file between the access() call and the open() call via symbolic link to some system file like /etc/shadow
    d. More general file updates
        i. Install application – want all or nothing
        ii. Update website – want all files update or none when some goes there (if live)
3. General solution: Transactions
    a. Designed for fault tolerance: reason about state of system after a failure
        i. Key property: atomicity,
            1. Atomicity means either the whole set of operations happened successfully, or none of the operations took place
            2. No clean up code needed
            3. Typically mplemented via logging operations during transaction
                a. Redo after commit if not complete
                b. Undo after failure it not complete
        ii. Key property: isolation
            1. Isolation means intermediate state not visible to other entities (could be threads, processes, transactions)
            2. Key rule: two **conflicting** transactions cannot overlap
                a. Access same location in memory, one is a write
                b. E.g. allow readers/writers locks or mutex locks.
            3. Implemented via;
                a. Locks: lock data when modifying it, block others from seeing it until commit/abort

            b. Speculation: make a copy of the data, only modify the copy, make copy visible ("publish") on commit
        4. "strong isolation" – prevent access from other transactions and code outside a transaction
        5. "Weak isolation" – only prevent conflicting access from other transactions.
        6. "Serializability" – outcome of transactions that overlap is equivalent to executing them in some serial order
            a. "serializing" means executing transactions in an order, rather than concurrently
    iii. Consistency: really an application property – it needs to ensure invariants hold at end of transaction
    iv. Durability: effects of a transaction, if committed, will survive a crash
        1. Implies saved on disk
        2. Not always needed

4. Transactional memory
    a. Use isolation properties of transactions instead of locks
    b. Two key needs:
        i. Version control:
            1. Need to keep both old version, for abort, and new version, for commit
            2. Can do eagerly: update in place, store old version someplace else
            3. Can do lazily: update someplace else, write back on commit
                a. Eager: faster commit, slower abort
                b. Lazy: faster abort, slower commit
        ii. Conflict detection
            1. Need to detect when two threads/transaction are modifying the same state
            2. Can do it pessimistically/eagerly: acquire locks as execute transaction to block other threads
                a. On a conflict, stall one transaction, or if a deadlock, abort one transaction to let other continue
            3. Can do it optimistically: acquire locks when ready to commit, do commit, release locks
                a. First to commit makes all other transactions accessing the same data abort
    c. Benefits:
        i. Not need to assign locks; automatically "locks" just the locations accessed
        ii. No deadlock: will abort & retry if would deadlock
        iii. Concurrent execution for non-conflicting transactions
            1. Like fine-grained locks, but easy of coarse-grained locking
    d. Implementation:
        i. Hardware:
            1. Version management : buffer new state in cache

        2. Isolation: abort transaction if another thread tries to access state accessed by a transaction in a conflicting way
    ii. Software:
        1. Instrument loads/stores
        2. Keep table of memory locations referenced for conflict detection
        3. Keep log of locations accessed for atomicity

5. Transactional Memory
   a. What do locks give you?
      i. Atomicity: entire critical section is executed as a chunk from perspective of other threads
      ii. Isolation: don't see intermediate states of a thread in a critical section
   b. Problems:
      i. Deadlock: acquire locks out of order
      ii. Wrong lock: acquiring correct lock for data (see eraser)
      iii. Lock granularity:
         1. Fine grain – lots of time spent locking/unlocking, likely deadlock
         2. Coarse grain – easy, correct, but low concurrency with many processors
   c. Transactional memory: allow programmer to declare regions "atomic"
      i. No associating locks with code/data
         1. Just annotate code that should be executed atomically
      ii. Provides atomicity: executes either all the way to the end or not at all
         1. Either acquire all locks first, so can execute to end without waiting, or speculate and abort if got it wrong
      iii. Example:
         1. Transfer(queue x, queue y, obj z) {
```
  begin_tx
    x.remove(z);
    y.add(z);
  end_tx;
```
         2. What happens if called on (x,y) and (y,x)?
            a. System detects a conflict, aborts one of them
         3. What if called on (x,y) and (a,b)?
            a. Can execute in parallel (fine grained locking)
   d.
   e. Implementation
      i. Version control: for atomicity/aborts/deadlock
         1. Need to keep both old version, for abort, and new version, for commit
         2. Can do eagerly: update in place, store old version someplace else
         3. Can do lazily: update someplace else, write back on commit
            a. **Eager**: faster commit, slower abort
            b. **Lazy**: faster abort, slower commit
         4.

    ii. Provides isolation: internal state not visible
        1. Detect concurrent memory accesses from transactions in other threads
        2. Stall/abort/wait on lock if someone tries to access same data
    iii. Automatically detects conflicts
        1. Value written by one transaction is read/written by another transaction
        2. Prevents serializability: execution as if a global lock held for duration of transaction
        3. Solution is to abort one of the two transactions
    iv. Conflict detection
        1. Need to detect when two threads/transaction are modifying the same state
        2. Can do it pessimistically/**eagerly**: acquire locks as execute transaction to block other threads
            a. On a conflict, stall one transaction, or if a deadlock, abort one transaction to let other continue
        3. Can do it optimistically,**lazily**: acquire locks when ready to commit, do commit, release locks
            a. First to commit makes all other transactions accessing the same data abort
    v. Tradeoff:
        1. Memory for time; buffers state in memory for atomicity to solve deadlocks.
    vi. Compared to locks:
        1. Only detects conflicts when two threads access the **same memory locations**
            a. Like a perfectly fine-grained lock; only protects memory actually accessed
        2. No need to select the lock to protect data; always detects concurrent access to same memory locations
    vii. Contention: what happens when applications conflict?
        1. Contention manager (in hw?) applies a policy to decide which transaction gets to keep executing.
        2. Common policies:
            a. Oldest wins: ensures liveness
            b. Committer wins: only detect at commit, long tx gets starved
            c. SizeMatters: tx that has read/written more data wins
f. What does it make easier?
    i. No longer remember which lock protects which data
        1. Only use transactions
    ii. No longer have to create lots of locks
        1. Write coarse grained locks, get benefit of fine-grained locks

        2. Just transactions
    iii. Avoid the cost of acquiring/releasing a lock
        1. Atomic instructions are expensive
    iv. No deadlock between pure transactions
        1. Detected by TM system, resolved automatically by abort
        2. If call from tx 1 into tx2, which calls back into code accessing data from tx1, what happens?
            a. 
```
F() {
  begin_tx;
    x = 1;
    A();
  end_tx;
}
A() {
  begin_tx;
  G();
  end_tx;
}
G() {
  x = 2;
}
```
            b. In a monitor, this will deadock when recursively acquiring monitor lock
            c. With a transaction, this is just fine
    v. What happens instead of deadlock?
        1. Aborts
    vi. What happens where you might have lock contention?
        1. Repeated aborts; even worse than lock contention

6. TM Implementation
    a. Hardware:
        i. Save registers
        ii. Buffer state accessed by a transaction in cache
        iii. Detect coherence request from another core as a conflict, abort transactions in either thread
        iv. Note: faster than locks (no atomic instructions)
    b. Software
        i. Instrument code to note begin/end of transaction
            1. Save registers
        ii. Note all memory accesses and record
        iii. Compare accesses against concurrent transactions from other threads
            1. On conflict, abort one transaction
        iv. Note: 3-10x slower than normal code
    c. What gets harder?
        i. High contention: rather than queuing, tx all try, get aborted, restart

                1. May have mutual death
                2. May have backoff (Ethernet style) to make progress, causing longer delays
            ii. Dealing with non-transactional code
                1. System calls
                2. I/O
                    a.
            iii. Synchronization
                1. How do you deal with waiting, signaling?
                2. A: no answer – doesn't help
            iv. Modularity/correctness
                1. Not much better than locks
                2. Can enforce in language to be lexically scoped, to ensure you end transaction
                3. Take away points
7. System transactions
    a. Overview:
        i. Big picture: apply transactions to system calls and kernel state
            1. Abort/block conflicting accesses while transaction in progress
            2. Intuition: most system calls execute like mini 1-operation transactions
                a. E.g. two processes try to create a file with the same name
        ii. Only applies to system state
            1. Aborts do not roll back user-level state
        iii. Not safe to communicate two-ways
            1. Outside entity learns of state inside transaction, cannot roll back or might deadlock waiting for response
    b. General idea
        i. Buffer modifications in transaction-local structures until commit ("lazy version management")
            1. Example: file write : data goes to buffer
    c. Implementation
        i. Version management
            1. Multiple versions can exist
            2. Create private copy – a shadow – when accessed
                a. All subsequent system calls access shadow – protect against external change
            3. Split objects into headers and data
                a. Header is stable – destination of pointers, identity information (inode number)
                b. Data is versioned
                c. Code that needs versioned data takes a diferent type; identifyable statically in compiler
                d. Split data portion of an object if has disjoint use

            i. Inode metadata has both mapping information and owner/access time/permissions
      4. Support read-only objects to avoid expensive copies; code has to guarantee it will never be written
   ii. Isolation/conflict detection
      1. Need to record who is using an object in a transaction
         a. Embed on object header – tx_data field
         b. Existence of a list of readers or a writers could trigger a conflict
      2. Use normal locks to detect conflicts with non-tx code
         a. Tries to get lock while TX in progress
      3. Resolving conflicts
         a. Go by OS priority to prefer high-prio threads, or by older TX (to assure progress)
         b. For non-tx code, use preemption to suspend non-tx thread until tx completes
   iii. Aborts
      1. Can abort back to beginning of a system call (before anything modified) by storing registers there
         a. Discard shadow objects
   iv. Commits
      1. Defer some operations until commit
         a. Free memory – may need it back if abort
         b. Notify of file change – inotify, dnotify
            i. Only on commit does it become permanent
         c. Store a list of deferred operations – "commit handlers" to run at commit
      2. Protocol
         a. Go through all objects, get kernel lock protecting object
         b. If get all locks, can then apply updates
d. Integration with user-mode TX
   i. User TX gets ready to commit, asks system TX to commit
   ii. If successful, user Tx follows system TX
      1. Requres user TX not required to abort once asks system TX to commit

8. TxOS subsystems
   a. File system:
      i. All updates written as a single file system transaction to disk; ensures atomicity & durability
   b. Processes:
      i. Allow transactional processes that access internal transaction state
      ii. All tasks in process have to call sys_xend() or exit() to commit; not just any one thread
   c. Signales:

        i. Defer until commit if possible
        ii. Allows signal handlers to be transactions themselves

9. Challenges:
    a. How do networking/communication?
    b. What happens if there is a failure during commit? Write some blocks to disk but not all?
    c. What if you run out of memory to buffer state, e.g. for the file system?