Goals

* Atomic
* Consistent
* Isolated - Serializable
* Durable

Before-or-After Atomicity

* **sequence coordination** = an action must happen before another one
* before-or-after atomicity
  + 2 concurrent actions A and B have this property if from point of view of invokers, result is same as if A occurred completely before or after B
  + coordination among concurrent actions correct if every result is guaranteed to be one that could have been obtained by some purely serial application of those same actions
  + When concurrent actions have before-or-after property, they are **serializable**
    - exists some serial order of those concurrent transactions that if followed lead to same ending state
* External time consistency
  + if external evidence that action T1 ended before T2 began, serialization order should make T1 precede T2
* sequential consistency
  + result should be as if instructions executed in original order specified (by programmer for example)

Simple Locking

* each transaction must acquire lock for every shared data object it intends to read/write before doing the reading/writing
* may release locks only after installs last update and commits or completely restores data and aborts
* lock point
  + first instant in which transaction has acquired all of its locks
  + collection of locks called lock set
* each transaction needs to supply its intended lock set as argument to begin\_transaction operation
  + at that point it either waits to get all locks, or get all of them

Two-Phase Locking

* transaction can acquire locks as it proceeds, and transaction may read/write as soon as acquires lock on object
* transaction may not release any locks until passes its lock point
* transaction can release lock on objects that it only reads any time after reaches lock point if never need to read that object again, even to abort
* lock manager can implement this
  + acquire lock on first used of each shared variable
  + hold the locks until commit, abort or end
* If locks in volatile memory, after a crash, all locks disappear
  + this is fine, because any recovery algorithm can just replay the log and roll back any changes without an end message
    - no concurrent modifications since all non-committed transactions never share any locks

two-phase commit

* want atomicity of a multiple site transaction
* correctness = all sites commit or all site abort
* Example
  + Alice, coordinator, wants Bob do X, Charles do Y, Dawn do Z
  + Alice creates top-layer outcome record for overall transaction
  + Alice sends messages to B,C,D
    - ex: Alice tells B to do X
    - Keeps sending messages if do not receive response
  + Everyone sends back message saying "ready to commit"
  + Phase 1
    - Alice sends message saying "prepare to commit"
    - Bob, for example, commits tentatively (or aborts)
      * sends back message "I am prepared to commit. Have you committed"
      * Bob is now in prepared state
      * Bob waits for a reply indefinitely
    - If all workers send back prepared messages, phase 1 is complete
  + Phase 2
    - Alice commits transaction: marks her outcome record committed
    - Then sends message to B,C,D saying she committed
    - Each worker, upon receiving commit message, changes state from prepared to committed
    - If worker site crashes and recovers
      * it must classify any prepared transaction as tentative winner that it should restore to the prepared state
      * must reacquire any locks the prepared transaction was holding at time of failure
      * Lastly restart persistent sender to learn status of Alice's transaction
    - Alice only certain of eventual completion of her transaction
  + 3N messages to commit N transactions