6.852 Lecture 23

- Transactional memory (continued)
- Shared memory vs networks
- Paxos
- Reading:
  - Herlihy, Luchangco, Moir, Scherer paper
  - Dice, Shalev, Shavit paper
  - Lynch, Chapter 17
  - Lamport: The Part-Time Parliament

Transactional memory

- Raise level of abstraction
  - programmer specifies atomicity boundaries: transactions
  - system guarantees atomicity
    - commits if it can
    - aborts if not (roll back any changes)
    - possibly retry on abort
  - system manages contention (possibly separable functionality)
  - compositional (due to nested transactions)
    - but large transactions may not commit
  - simplified interface: atomic blocks
    - atomic { code }
    - automatic retry

Using transactional memory

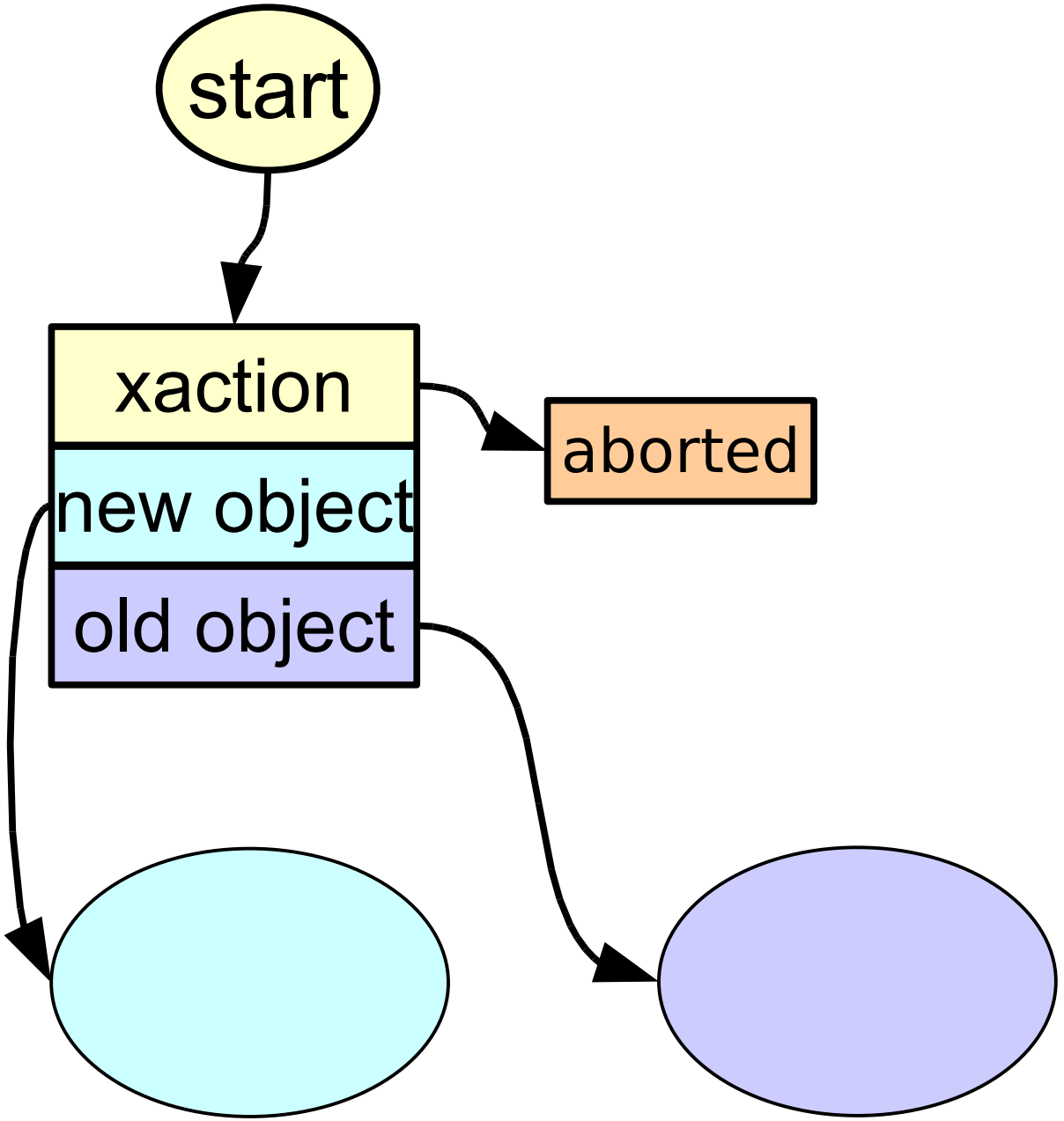
```
Q.enqueue(x)
  node = new Node(x)
  node.next := null
  atomic{
    oldtail = Q.tail
    Q.tail := node
    if oldtail = null then
      Q.head := node
    else
      oldtail.next := node
  }
```

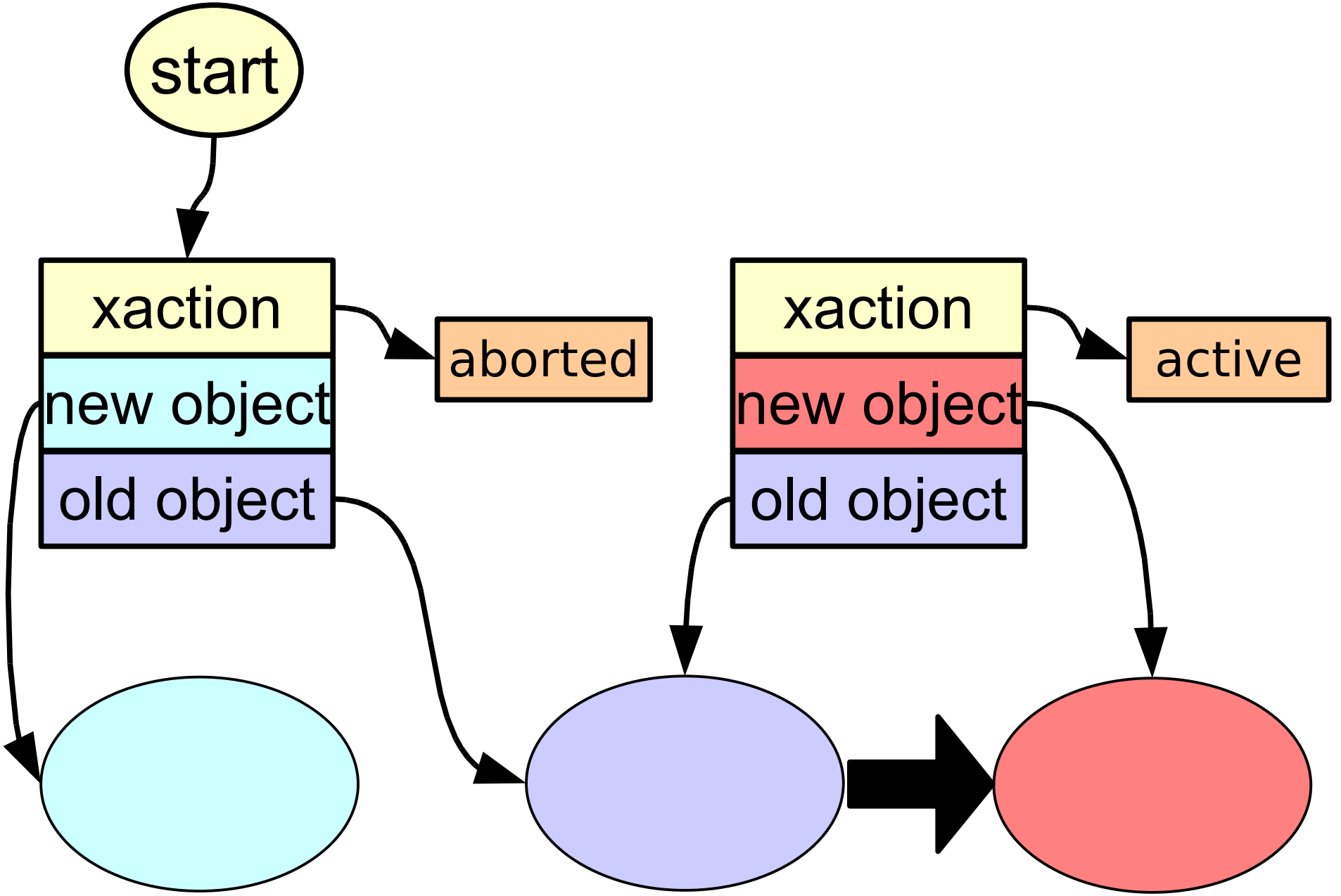
```
Q.dequeue()
  atomic{
    if Q.head = null then
      return null
    else
      node := Q.head
      Q.head := node.next
      if node.next = null then
        Q.tail := null
      return node.item
  }
```

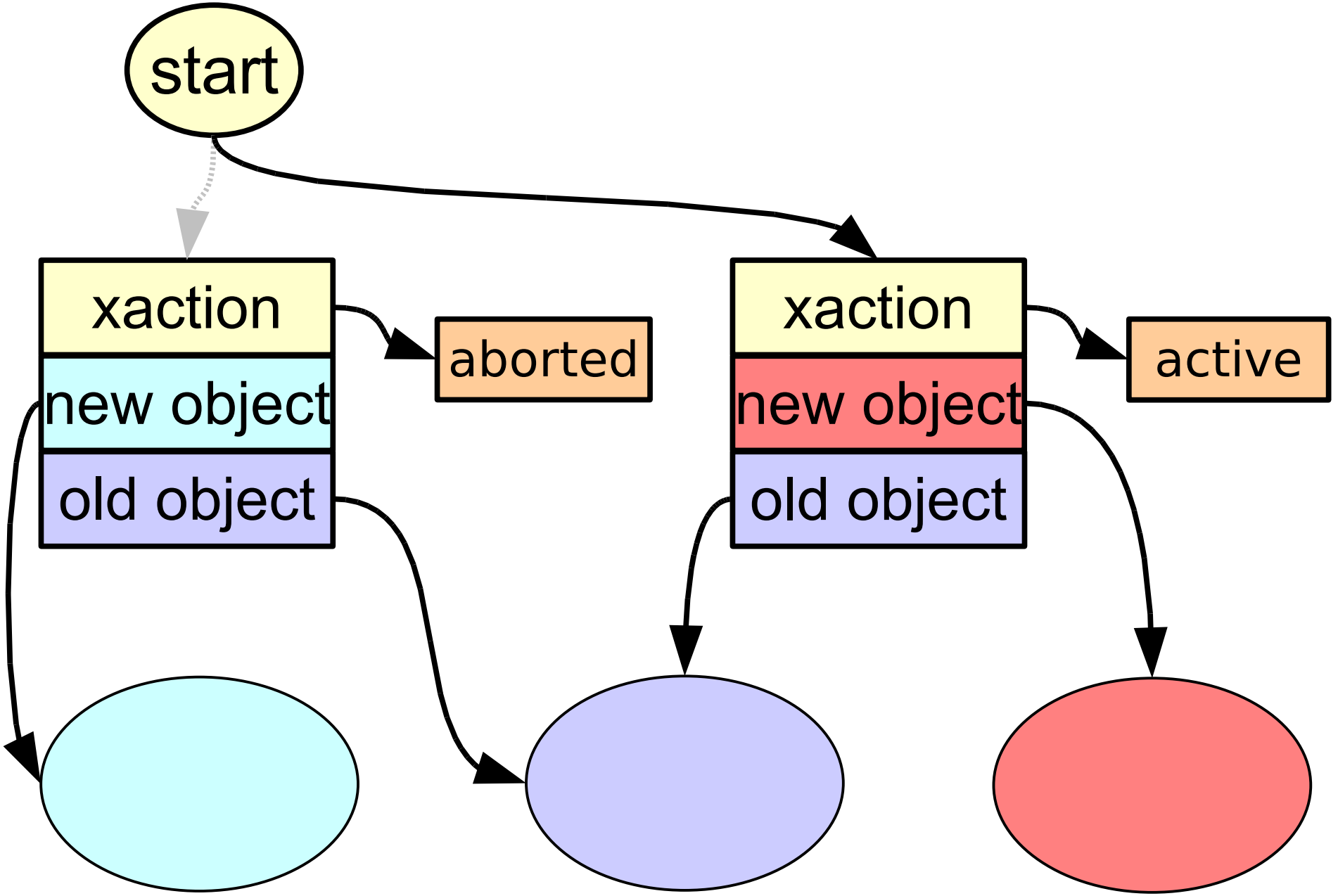
```
transfer(Q,Q')
  atomic{
    x = Q.dequeue()
    if x  null then
      Q'.enqueue(x)
  }
```

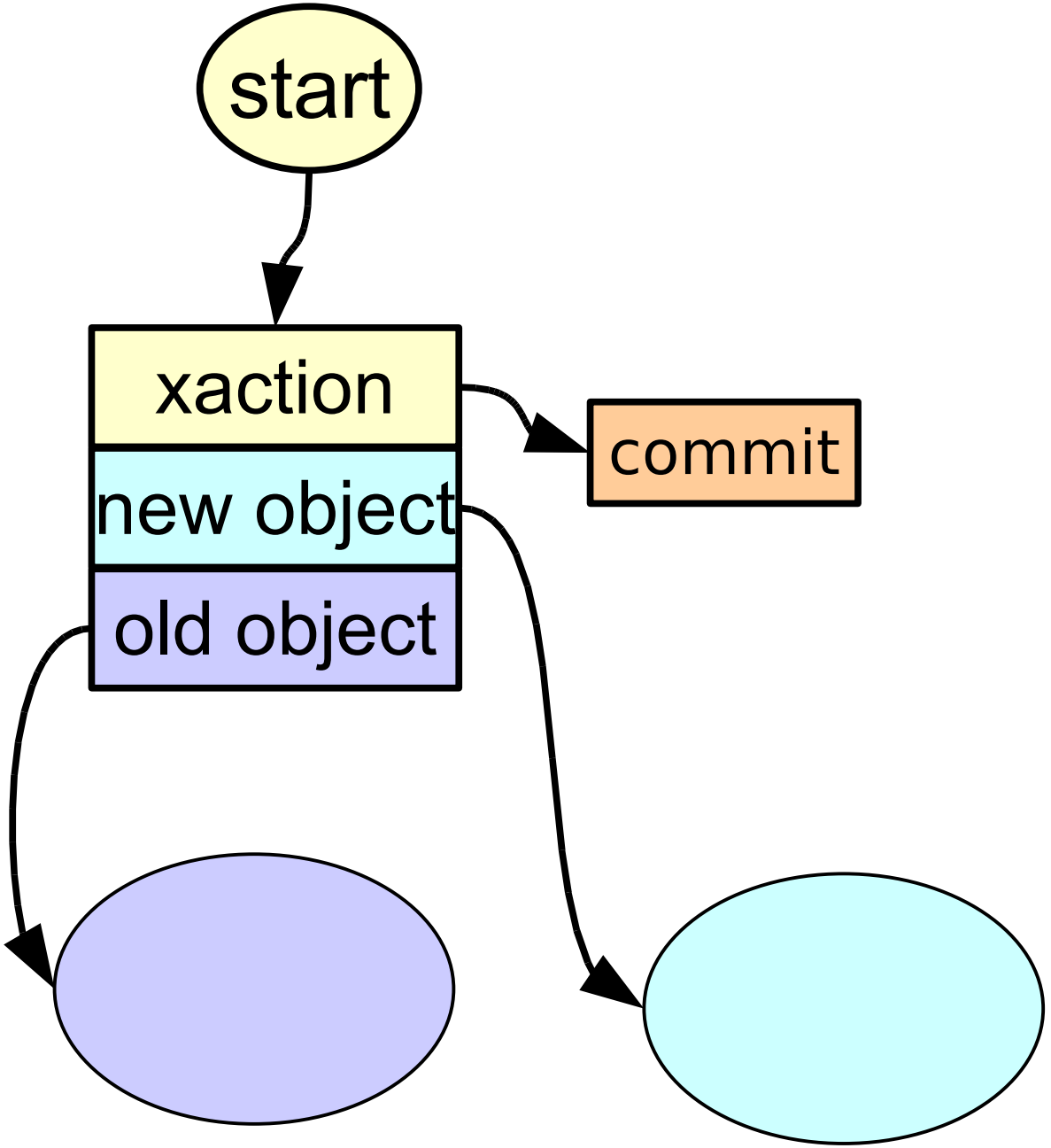
Dynamic STM (DSTM)

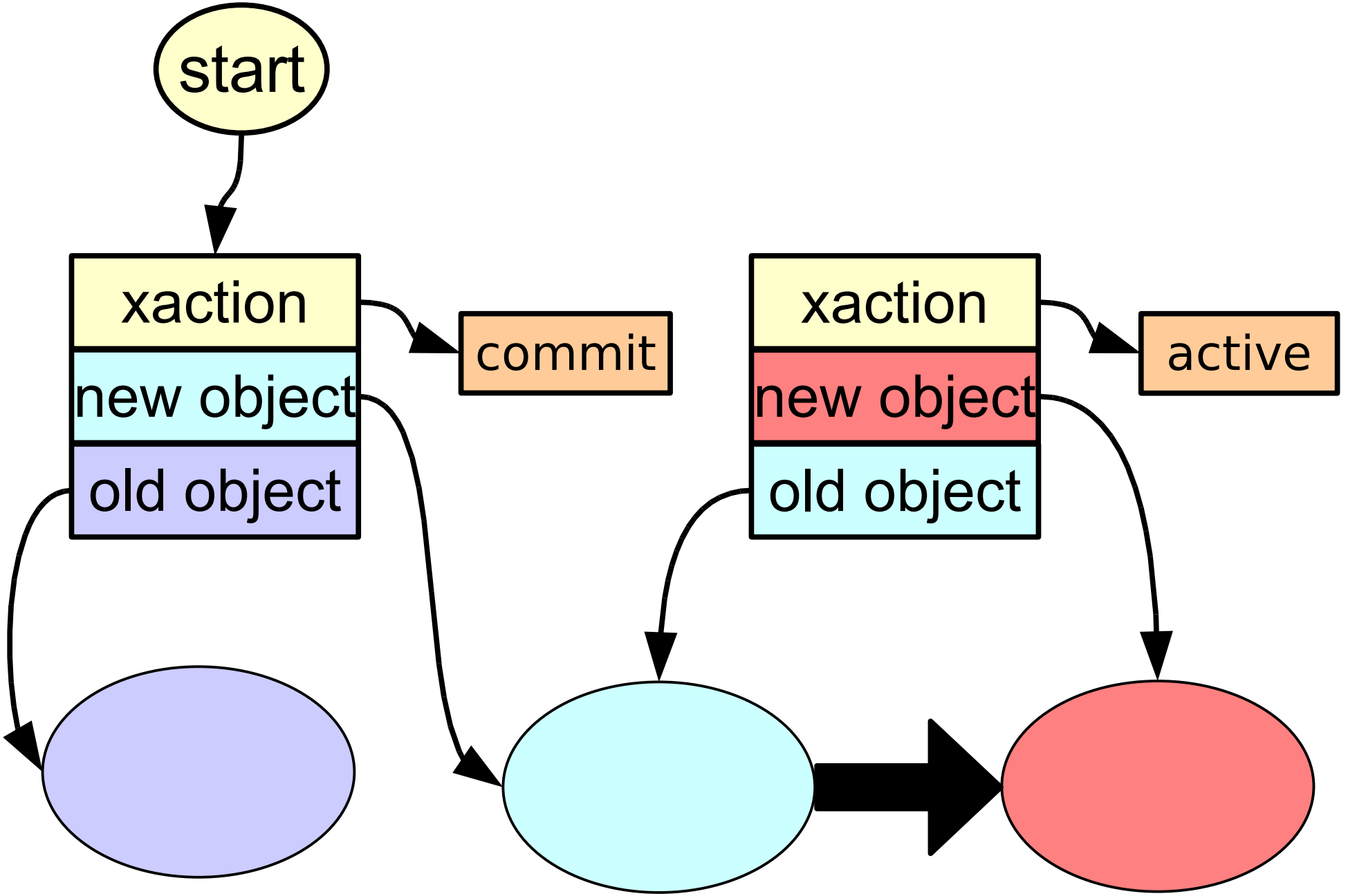
- object-based (Java™ library)
- no locks (obstruction-free)
- supports dynamic allocation and access
- separable contention management
- pluggable implementations (2nd release)
  - added blocking “shadow” implementation

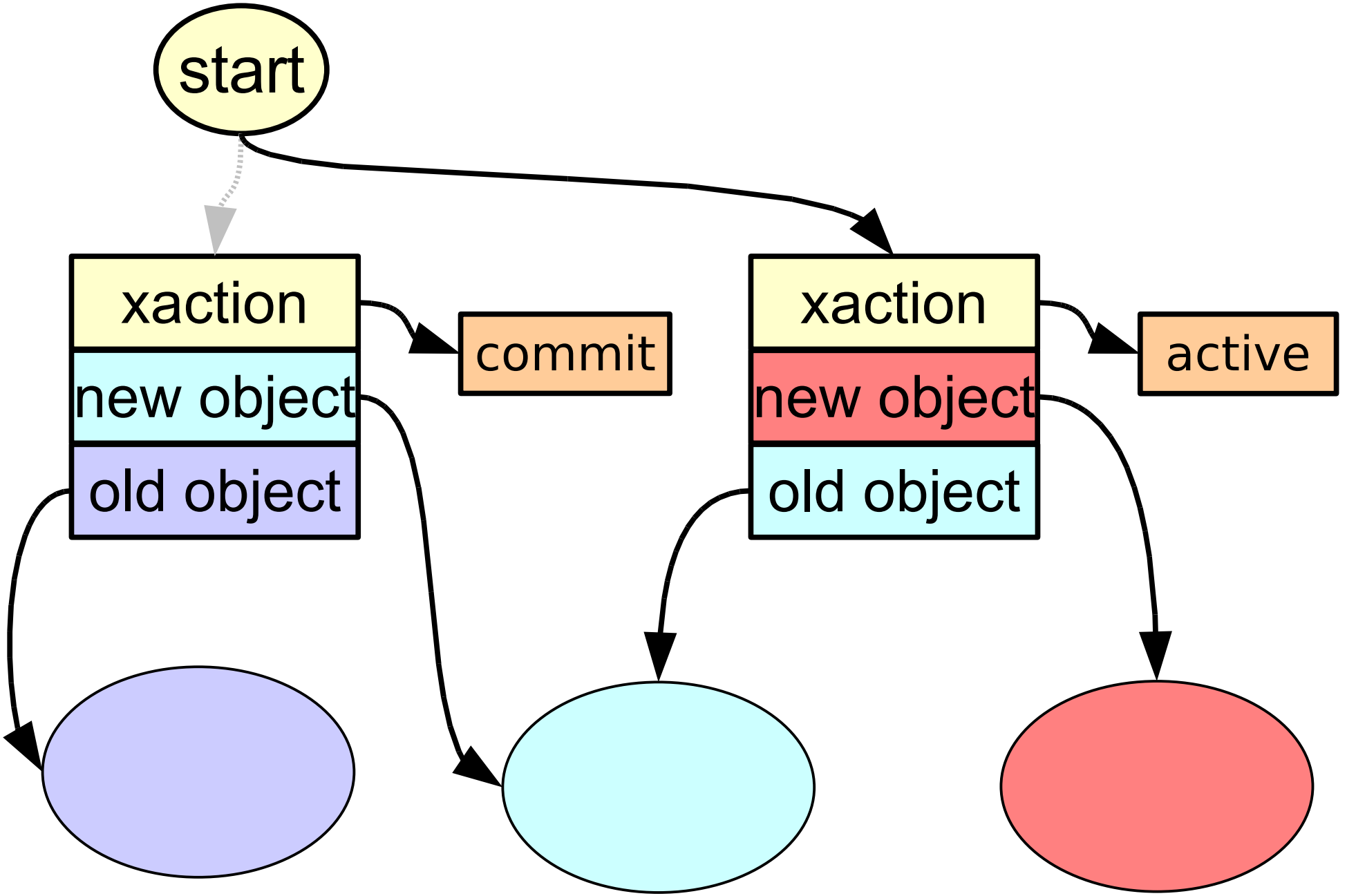

start
xaction
aborted
new object
old object

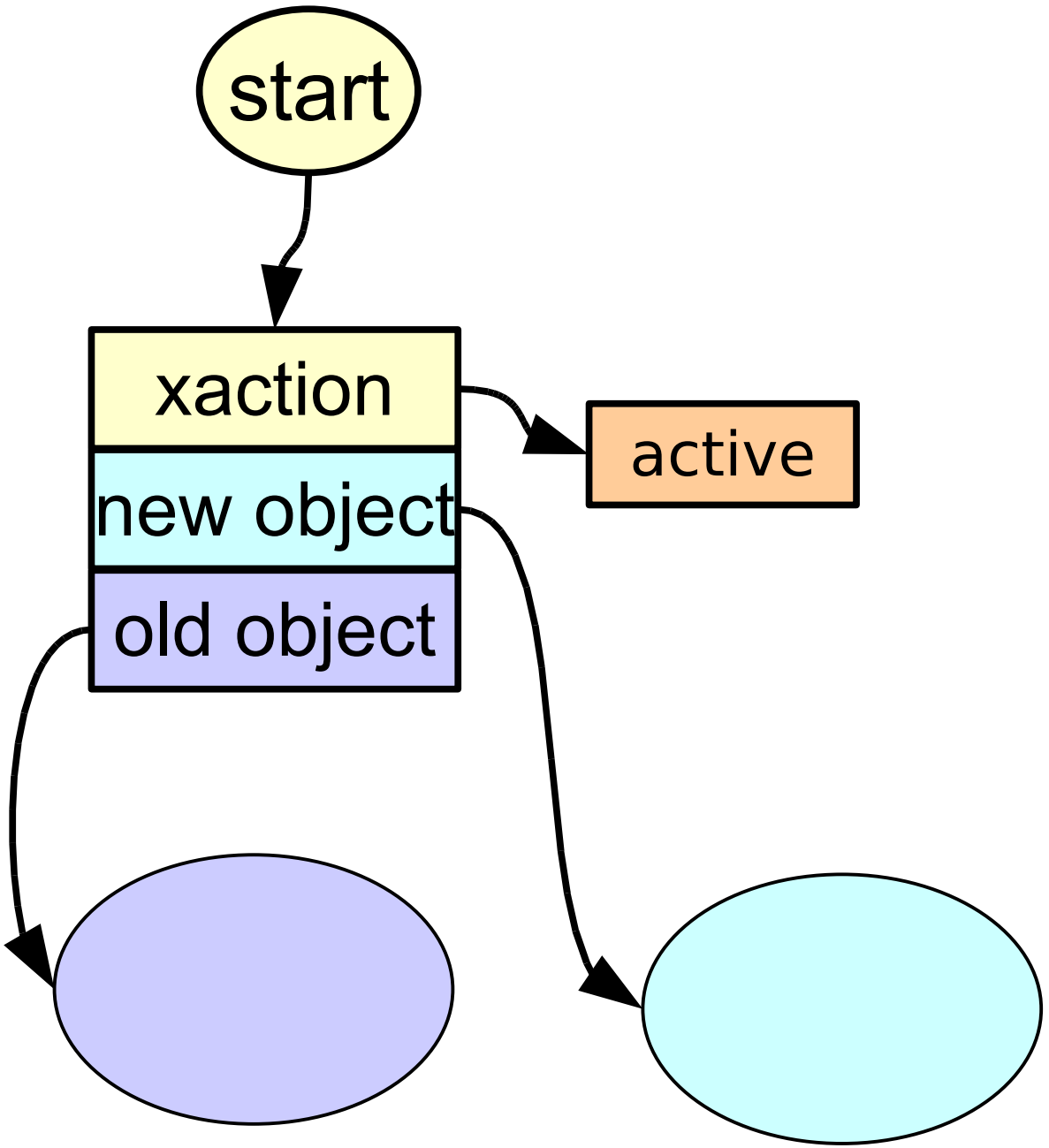

start
xaction
new object
old object
aborted
xaction
new object
old object
active

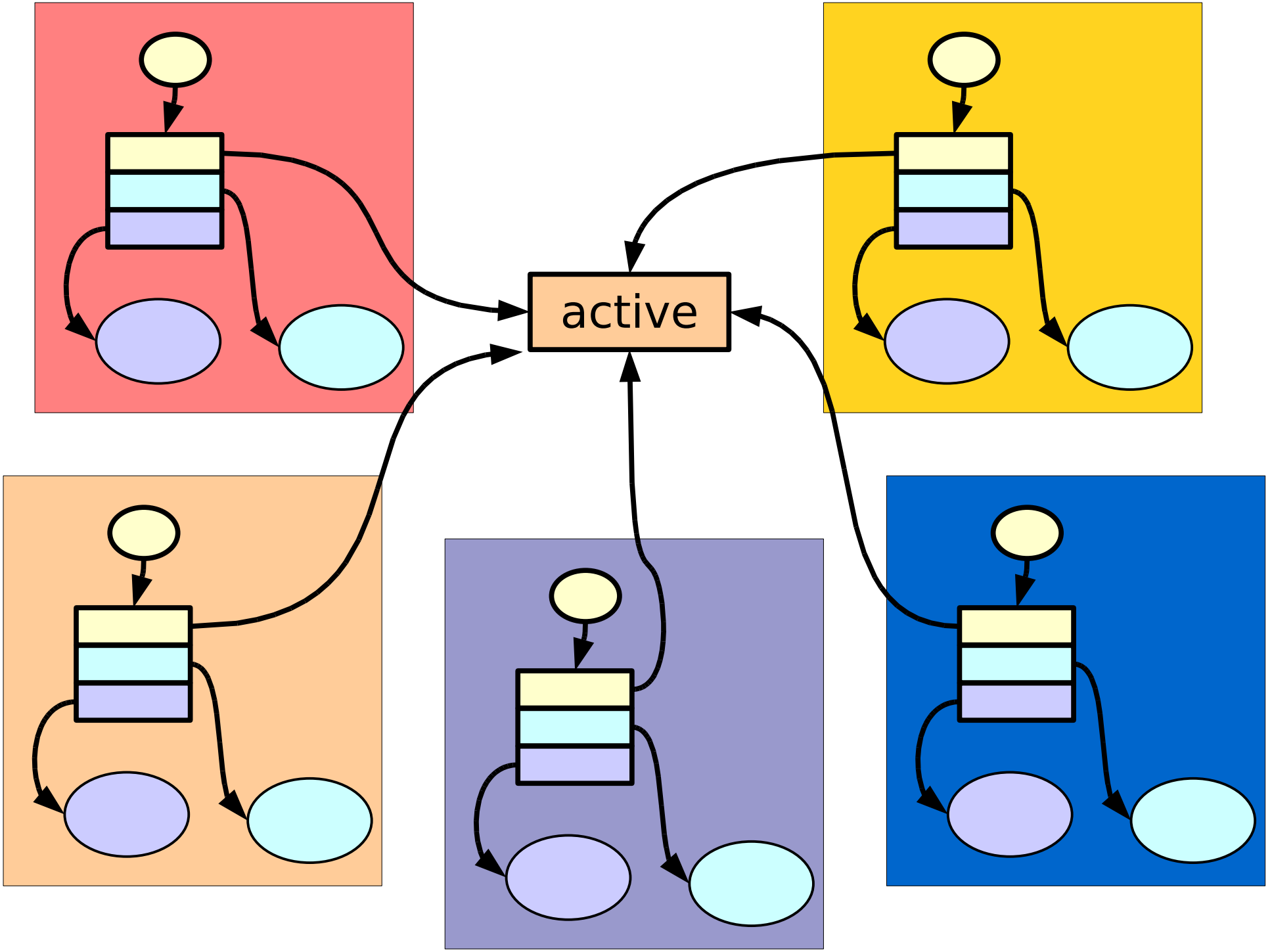

start
xaction
new object
old object
aborted
xaction
new object
old object
active


start
xaction
commit
new object
old object


start
xaction
new object
old object
commit
xaction
new object
old object
active

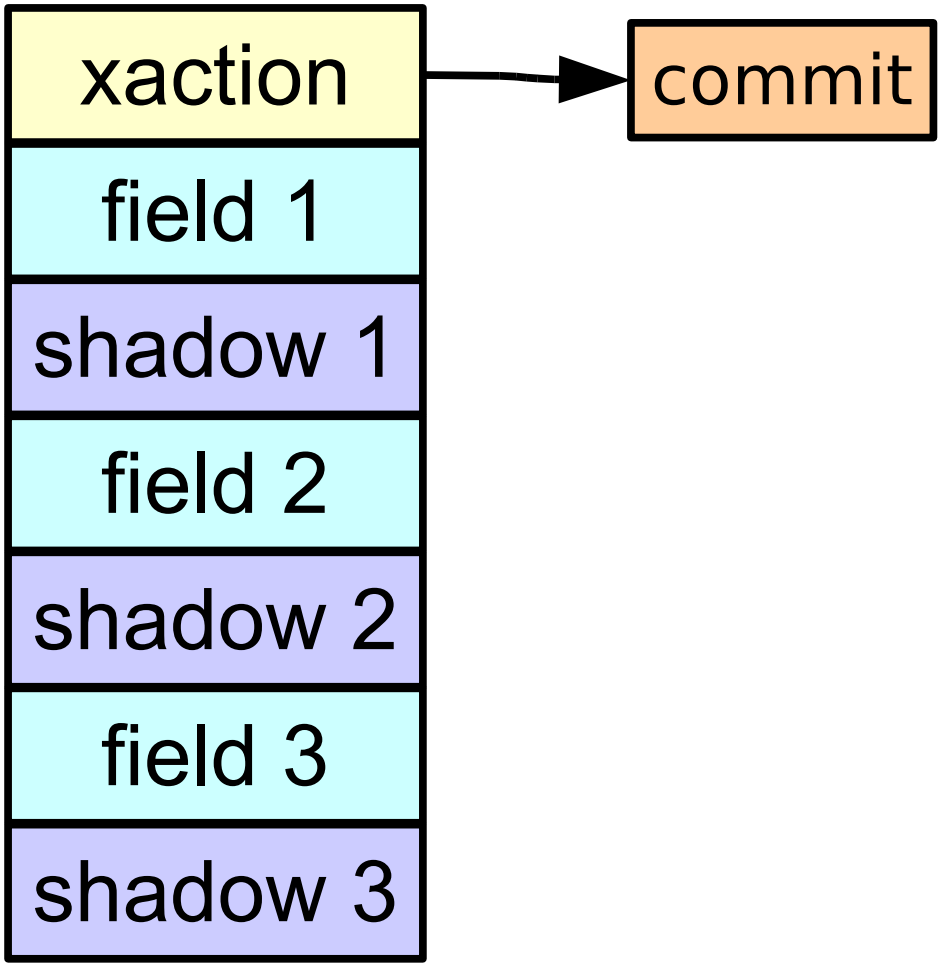

start
xaction
new object
old object
commit
xaction
new object
old object
active

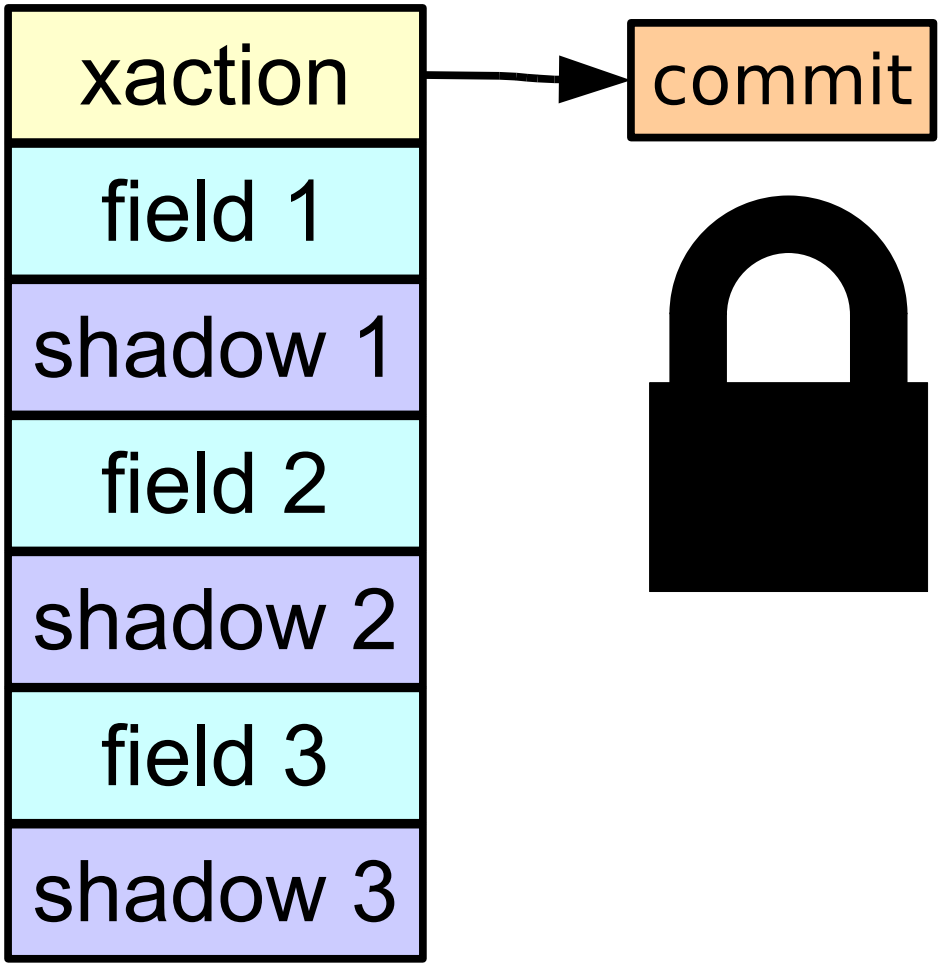

start
xaction
active
new object
old object

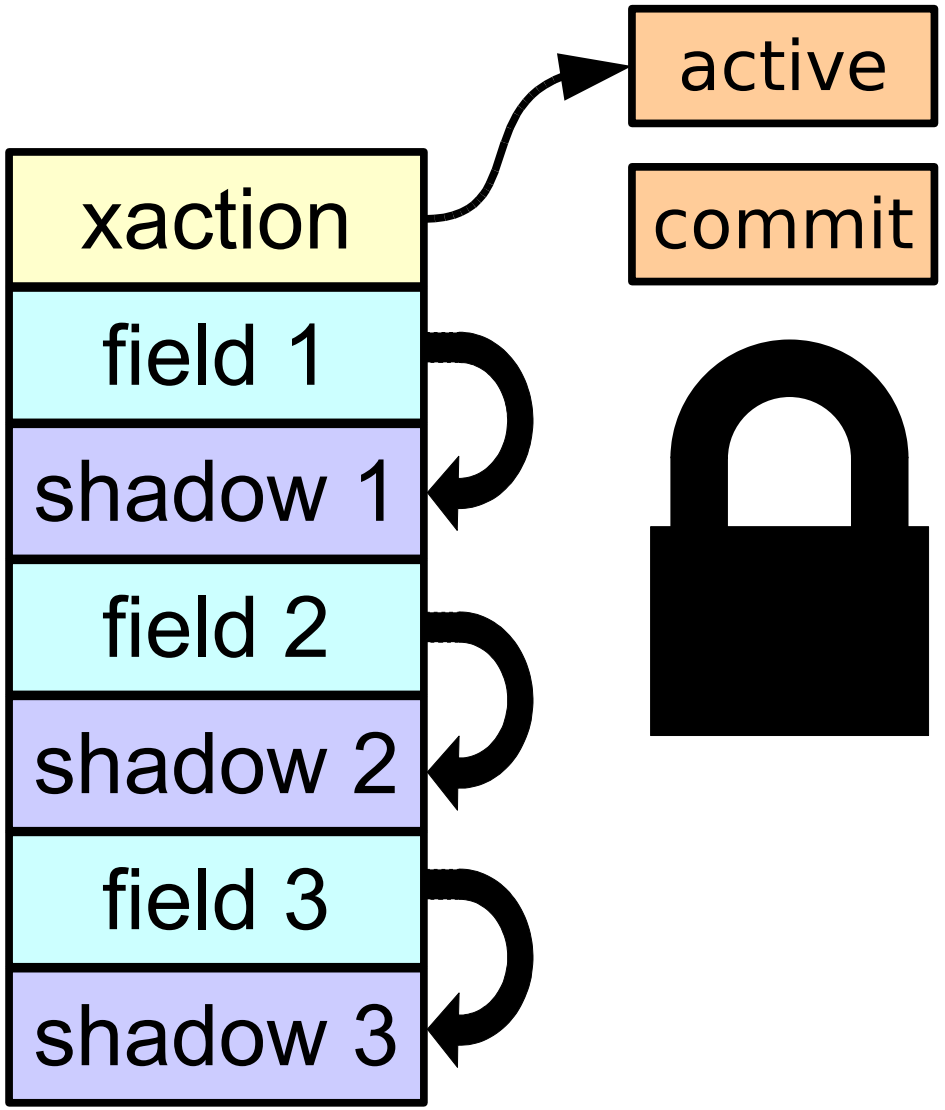

active

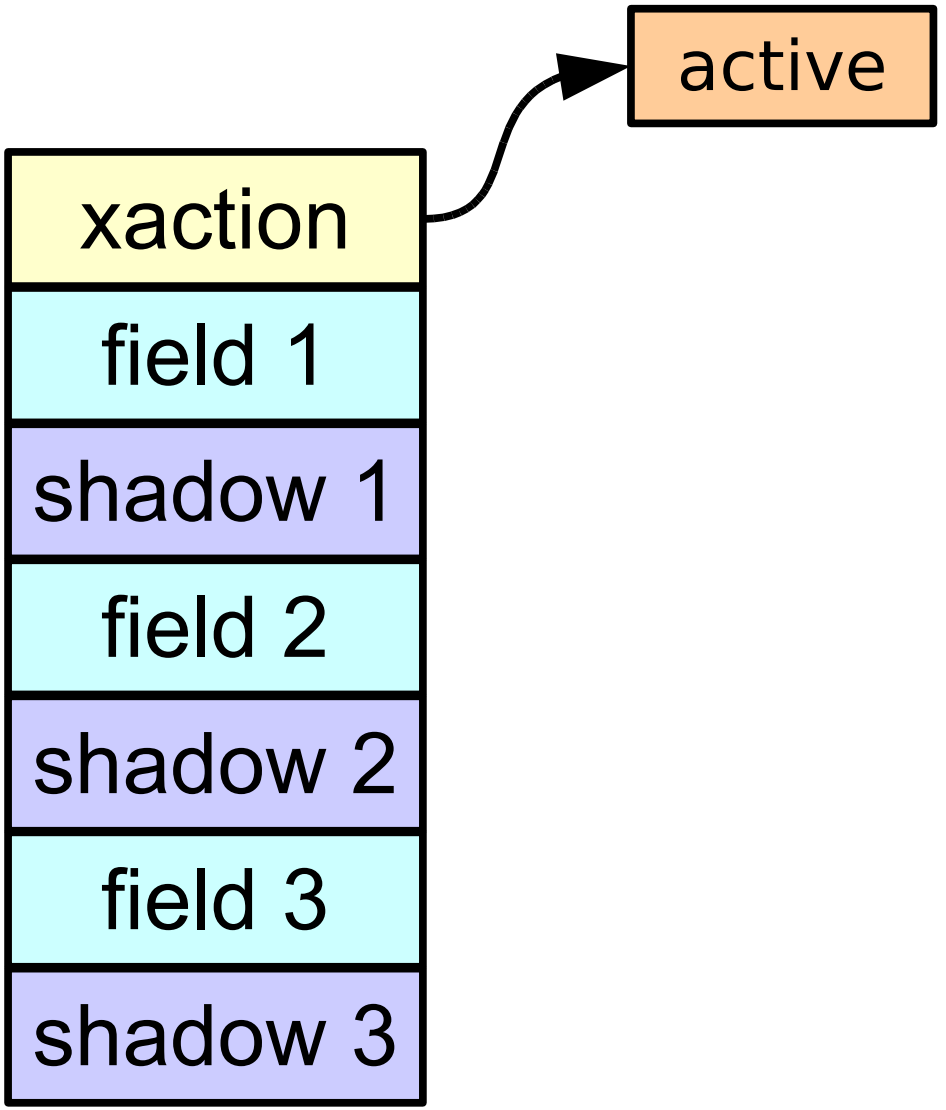
Enable “read-sharing”

- Invisible readers
  - keep thread-local “read set”: pairs of location and version read
  - validation checks no location in read set has changed
- Visible readers
  - keep set of active readers in “locator”
- Semi-visible readers
  - keep number of readers


xaction
commit
field 1
shadow 1
field 2
shadow 2
field 3
shadow 3


xaction
commit
field 1
shadow 1
field 2
shadow 2
field 3
shadow 3


active
xaction
commit
field 1
shadow 1
field 2
shadow 2
field 3
shadow 3


active
xaction
field 1
shadow 1
field 2
shadow 2
field 3
shadow 3

Transactional Locking II (TL2)

- Best (or one of the best) performing STM
  - also other nice properties beyond the scope of this lecture
- Lock-based, word-based STM
  - locks only held during commit phase (not executing user code)
- Uses global version number (potential bottleneck)
  - updated by each writing transaction (but could relax this)
- Every location also has version number
  - transaction that last wrote it

Transactional Locking II (TL2)

- Read global version counter
  - store locally: rv (for read version)
- Run transaction “speculatively”
  - track which locations are read and written
  - when location first accessed, check version counter
  - write values into write set
  - read values into read set (so transaction gets consistent reads)
    - if value was written by transaction, get value from write set
- At commit
  - lock write set
  - increment global version counter
  - validate read set
  - write-back values
  - release locks

Combining hardware and software

- Hardware-assisted transactional memory
  - new required hardware
  - hardware support can accelerate implementation
- Hybrid transactional memory
  - STM that can work with HTM
    - hardware transactions and software transactions must “play nicely”
    - can be used now (with no hardware support)
    - can exploit HTM support with little change
  - phased transactional memory
    - can switch dynamically between using STM and HTM

Shared memory vs networks

- Network algorithm to shared memory algorithm
  - "implement" network using shared memory
  - easy because shared memory is "stronger" than network
  - useful primarily for lower bounds (e.g., impossibility of consensus)
- Shared memory algorithm to network algorithm
  - "implement" shared memory over network (DSM)
  - easy if no failures
  - impossible if more than n/2 failures
  - Attiya-BarNoy-Dolev fault-tolerant algorithm
- Replicated state machine over network
  - Paxos

Network on shared memory

- State variables for each i (written by i)
  - $pstate_i$: states(P_i)
  - $sent(j)_i$ for each out-neighbor j: sequence of M, initially empty
  - $msgs\text{-}rcvd(i)_j$ for each in-neighbor j: **N**, initially 0
- Transitions for i
  - $send(m,j)_i$:
    - pre: $send(m)_{i,j}$ in $pstate_i$
    - eff: append m to $sent(j)_i$; update $pstate_i$ as for $send(m)_{i,j}$
  - $receive(m,j)_i$
    - pre: $|sent(i)_j| > msgs\text{-}rcvd(j)_i$; $m = sent(i)_j[msgs\text{-}rcvd(j)_i+1]$
    - eff: increment $msgs\text{-}rcvd(j)_i$; update $pstate_i$ as for $receive(m)_{j,i}$
  - all others: precondition/effect as in P_i with state $pstate_i$

Network on shared memory

- Impossibility of consensus on network
  - with reliable FIFO channels
  - with reliable broadcast
    - similar transformation

Shared memory over network

- Assume shared memory system A
  - n ports; user U_i for i = 1..n interacts with process i on port i
  - for each i, either user's turn, or process's turn (not both)
    - so we can use atomic objects instead of shared variables
- Design asynchronous network system B
  - same ports/user interface
  - execs of B are indistiguishable (to users) from execs of A
    - and same processes fail (if applicable)
- Non-fault-tolerant strategies: single-copy, multiple copy
- Impossibility of tolerating majority failures
- ABD fault-tolerant algorithm for read/write registers

Single-copy DSM

- Each shared variable is “owned” by some process
  - owner(X) known by all
  - handle each shared variable independently
- All actions other than shared-memory access as before
- To invoke op on X, send “inv(op,X)” to owner(X)
  - wait for response
  - continue to handle invocation msgs received
- Each process applies operations in order received
  - send response of each operation to appropriate process

Single-copy DSM

- Can implement any shared variable type
- Location of shared variables
- Elimination of busy-waiting
  - send “condition” to owner
  - works for multiple variables if they have the same owner
- Process stopping means no access to variables it owns

Multicopy DSM

- Goal: improve performance of read
  - maintain several copies
  - can be good if reading is more common than writing
  - still no fault tolerance
- Concern: “coherence” problems
  - inconsistency among various copies
  - problem even for single-writer shared variables
- Use transactions to maintain coherence
- See book for more details

Impossibility of n/2-fault-tolerance

- Theorem: In asynchronous reliable broadcast model with n = m+p processes, no implementation of m-writer p-reader atomic registers guarantees f-failure termination for f ≥ n/2.
- Proof: (same structure for many proofs against n/2-fault-tolerance)
  - Partition processes into two halves (each with size at most f)
    - some writer w in first half, some reader r in second half
  - Consider
    - α_w: fair exec starting with $write_w(1)$, all processes in second half fail
    - α_r: fair exec starting with $read_r$, all processes in first half fail
  - By f-failure-termination each must get a response (read gets 0)
    - Do α_w upto response, then α_r up to response, without failures and delaying all messages between two halves.
  - Violates atomicity because read comes after write but gets 0

ABD algorithm

- Implements atomic single-writer multireader register
- Tolerates $f < n/2$ stopping faults
- Assume reliable channels

ABD algorithm

STATE VARIABLES per process
val: V, initially v_0
tag: **N**, initially 0
readtag: **N**, initially 0
lots of "bookkeeping" variables

WRITER
on write(v)
  (val,tag) := (v,tag+1)
  send "write(val,tag)" to all readers
    - wait for ack from majority
  return ack

READERS
on receiving "write(v,t)" from writer
 if t > tag then
    (val,tag) := (v,t)
  send "write-ack(t)" to writer

READERS
on read
  readtag := readtag+1
  send "read(readtag)" to all other processes
    - wait for ack from majority
  let t be largest tag received
  if t > tag then (val,tag) := (v,t)
    where v is value received with t
  send "propagate(val,tag,readtag)" to all readers
    - wait for ack from majority
  return val

ALL PROCESSES
on receiving "read(rt)" from $_j$
  send "read-ack(val,tag,rt)" to j

READERS
on receiving "propagate(v,t)" from j
  if t > tag then (val,tag) := (v,t)
  send "prop-ack(t)" to j

ABD algorithm

- Correctness
  - well-formedness
  - f-failure-termination
  - atomicity
    - linearization point of write with tag t
      - when majority of processes have tag ≥ t
      - may linearize multiple writes at same point
    - linearization point of read returning value associated with tag t
      - immediately after linearization point of write with tag t, or
      - immediately after invocation of read, (why do we need this?)
      - whichever is later
    - book uses partial order method

ABD algorithm

- What kind of channels are required?
  - reliable?
  - nonduplicating?
  - FIFO?
  - Byzantine?
- What kind of faults are tolerated?
  - stopping?
  - omission?
  - Byzantine?

Agreement in asynchronous networks

- Impossible to reach agreement in asynchronous networks, even if we know that at most one failure will occur.
- But what if we really need to?
  - For transaction commit.
  - For agreeing on order of operations to perform.
  - …
- Some possibilities:
  - Randomized algorithm (Ben-Or), terminates with high probability.
  - Approximate agreement.
  - Use a failure detector service, implemented by timeouts.
- Best strategy:
  - Guarantee agreement, validity in all cases.
  - Guarantee termination only if the system eventually “stabilizes”:
    - Failures, recoveries stop.
    - Timing of messages, process steps within known “normal” bounds.
  - Actually, stable behavior need not continue forever, just long enough for termination to occur.

Eventually stable approach

- [Dwork, Lynch, Stockmeyer] first to present an algorithm with these properties.
- [Lamport, Part-Time Parliament]
  - Introduced the Paxos algorithm.
  - Relationship with [DLS]:
    - Achieves similar guarantees.
    - Paxos allows more concurrency, tolerates more kinds of failures.
    - Basic strategy for assuring safety similar to [DLS].
  - Background:
    - Paper unpublished for 10 years because of nonstandard style.
    - Eventually published “as is”, because others began recognizing its importance and building on its ideas.

Paxos consensus protocol

- Called “Single-Decree Synod” protocol.
- Assumptions:
  - Asynchronous processes, stopping failures, also recovery.
  - Messages may be lost.
- Paper also describes how to cope with crashes, where volatile memory is lost in a crash (skip this).
- We’ll present in two stages:
  - Describe a very nondeterministic algorithm that guarantees the safety properties (agreement, validity).
  - Constrain this to get termination soon after stabilization.

The nondeterministic “safe” algorithm: Ballots

- Uses ballots, each of which represents an attempt to reach consensus.
- Ballot = (id, value) pair.
  - id $\in$ BId, a totally ordered set of ballot identifiers.
  - Value in $V \cup \{ \perp \}$, where V is the consensus domain.
- Somehow, ballots get started, and get values assigned to them.
- Processes can vote for, or abstain from, particular ballots.
  - Abstention from a ballot is a promise never to vote for it.

The safe algorithm: Quorums

- The fate of a ballot depends on the actions of quorums of processes on that ballot.
- Quorum configuration:
  - A set of read-quorums, finite subsets of process index set I.
  - A set of write-quorums, finite subsets of I, such that
  - R ∩ W nonempty for every read-quorum R and write-quorum W.
- Generalization of majorities.
- Ballot becomes dead if every node in some read-quorum abstains from it.
- A ballot can succeed only if every node in a write-quorum votes for it.

Safe algorithm, centralized version

- Anyone can create a new ballot with id b:
  - make-ballot(b)
  - provided no ballot with id b has yet been created.
  - val(b) is set to $\perp$.
- A process i can abstain, in one step, from an entire set of ballots:
  - abstain(B,i), $B \subseteq BId$
  - provided i has not previously voted for any ballot in B.
  - B may be any set of ballot ids, not necessarily associated with already-created ballots.
    - For example, B = all ballot ids in some range $[b_{min}, b_{max}]$.
    - This will be important in the algorithm…

Safe algorithm, centralized version

- Anyone can assign a value v to a ballot id b, assign-val(b,v), provided:
  - A ballot with id = b has been created.
  - val(b) is undefined.
  - v is someone's consensus input.
    - ** For every b' < b, either val(b') = v or b' is dead.
- Recall: b' dead means read-quorum R has abstained from b'.
- Refers to every b' ∈ BId, not just created ones.
  - relies on "set abstentions".
- Thus, we can assign a value to a ballot b only if we know it won't make b conflict with lower-numbered ballots b'.
- Motivation:
  - several ballots can be created, collect votes.
  - more than one might succeed in collecting a write quorum of votes.
  - but we don't want successful ballots to conflict.

Next time

- Continue Paxos algorithm
- Self-stabilization
- Reading:
  - Lamport: Part-Time Parliament
  - Dijkstra paper on self-stabilization
  - Dolev book on self-stabilization, Chapter 2