Lecture 13:

Transactional Memory

Parallel Computing

Stanford CS149, Fall 2022

Raising level of abstraction for synchronization

- **Previous topic: machine-level atomic operations**
  - Fetch-and-op, test-and-set, compare-and-swap, load linked-store conditional

- **Then we used these atomic operations to construct higher level synchronization primitives in software:**
  - Locks, barriers
  - Lock-free data structures
  - We've seen how it can be challenging to produce correct programs using these primitives (easy to create bugs that violate atomicity, create deadlock, etc.)

- **Today: raising level of abstraction for synchronization even further**
  - Idea: transactional memory

What you should know

- What a transaction is
- The difference (in semantics) between an atomic code block and lock/unlock primitives
- The basic design space of transactional memory implementations
  - Data versioning policy
  - Conflict detection policy
  - Granularity of detection
- The basics of a software implementation of transactional memory
- The basics of a hardware implementation of transactional memory (consider how it relates to the cache coherence protocol implementations we've discussed previously in the course)

Between a Lock and a Hard Place

- Locks force trade-off between
  - Degree of concurrency ⇒ performance
  - Chance of races, deadlock ⇒ correctness

- Coarse grain locking
  - low concurrency, higher chance of correctness
    - E.g. single lock for the whole data structure

- Fine grain locking
  - high concurrency, lower chance of correctness
    - E.g. hand-over-hand locking

- Is there a better synchronization abstraction?

Review: ensuring atomicity via locks

```
void deposit(Acct account, int amount)
{
   lock(account.lock);
   int tmp = bank.get(account);
   tmp += amount;
   bank.put(account, tmp);
   unlock(account.lock);
}
```

- Deposit is a read-modify-write operation: want "deposit" to be atomic with respect to other bank operations on this account

- Locks are one mechanism to synchronize threads to ensure atomicity of update (via ensuring mutual exclusion on the account)

Programming with transactions

```
void deposit(Acct account, int amount)
{
   lock(account.lock);
   int tmp = bank.get(account);
   tmp += amount;
   bank.put(account, tmp);
   unlock(account.lock);
}
```

→

```
void deposit(Acct account, int amount)
{
   atomic {
     int tmp = bank.get(account);
     tmp += amount;
     bank.put(account, tmp);
   }
}
```

- **Atomic construct is declarative**
  - Programmer states what to do (maintain atomicity of this code), not how to do it
  - No explicit use or management of locks

- **System implements synchronization as necessary to ensure atomicity**
  - System could implement atomic { } using locks (see this later)
  - Implementation discussed today uses optimistic concurrency: maintain serialization only in situations of true contention (R-W or W-W conflicts)

Declarative vs. imperative abstractions

- Declarative: programmer defines <u>what</u> should be done
  - Execute all these independent 1000 tasks
  - Perform this set of operations atomically

- Imperative: programmer states <u>how</u> it should be done
  - Spawn N worker threads. Assign work to threads by removing work from a shared task queue
  - Acquire a lock, perform operations, release the lock

Transactional Memory (TM) Semantics

- **Memory transaction**
  - An atomic and isolated sequence of memory accesses
  - Inspired by database transactions
- **Atomicity (all or nothing)**
  - Upon transaction commit, all memory writes in transaction take effect at once
  - On transaction abort, none of the writes appear to take effect (as if transaction never happened)
- **Isolation**
  - No other processor can observe writes before transaction commits
- **Serializability**
  - Transactions appear to commit in a single serial order
  - But the exact order of commits is not guaranteed by semantics of transaction

Transactional Memory (TM)

In other words… many of the properties we maintained for a single address in a coherent memory system, we'd like to maintain for sets of reads and writes in a transaction.

Transaction:
- **Reads: X, Y, Z**
- **Writes: A, X**

These memory transactions will either all be observed by other processors, or none of them will. (the effectively all happen at the same time)

What is the consistency model for TM?

Motivating transactional memory

Another example: Java HashMap

Map: Key → Value

- Implemented as a hash table with linked list per bucket

```
public Object get(Object key)  {
  int idx = hash(key);            // compute hash
  HashEntry e = buckets[idx];     // find bucket
  while (e != null)  {            // find element in bucket
    if (equals(key, e.key))
      return e.value;
    e = e.next;
  }
  return null;
}
```

Bad: not thread safe (when synchronization needed)

Good: no lock overhead when synchronization not needed

Synchronized HashMap

- Java 1.4 solution: synchronized layer
  - Convert any map to thread-safe variant
  - Uses explicit, coarse-grained mutual locking specified by programmer

```
public Object get(Object key) {
  synchronized (myHashMap) {   // per-hashmap lock guards all
                               // accesses to hashMap
    return myHashMap.get(key);
  }
}
```

- Coarse-grain synchronized HashMap
  - Good: thread-safe, easy to program
  - Bad: limits concurrency, poor scalability

Review from earlier fine-grained sync lecture

What are solutions for making Java's HashMap thread-safe?

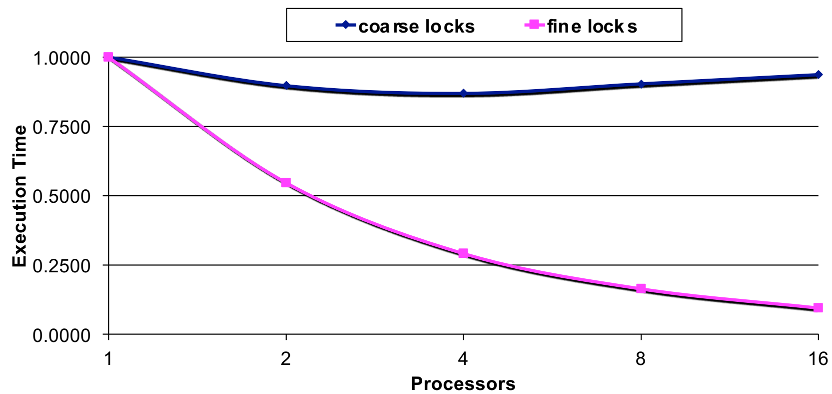
```
public Object get(Object key)  {
  int idx = hash(key);              // compute hash
  HashEntry e = buckets[idx];       // find bucket
  while (e != null)  {              // find element in bucket
    if (equals(key, e.key))
      return e.value;
    e = e.next;
  }
  return null;
}
```

- **One solution: use finer-grained synchronization (e.g., lock per bucket)**
  - **Now thread safe: but incurs lock overhead even if synchronization not needed**

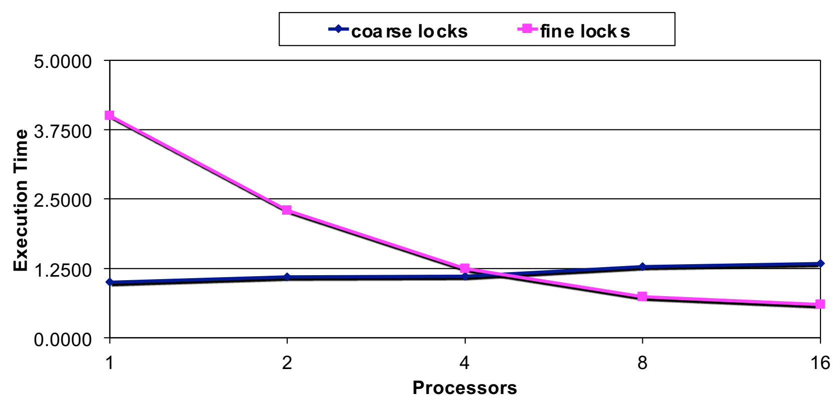
Review: performance of fine-grained locking

Reducing contention via fine-grained locking leads to better performance

Hash-Table



Balanced Tree



Transactional HashMap

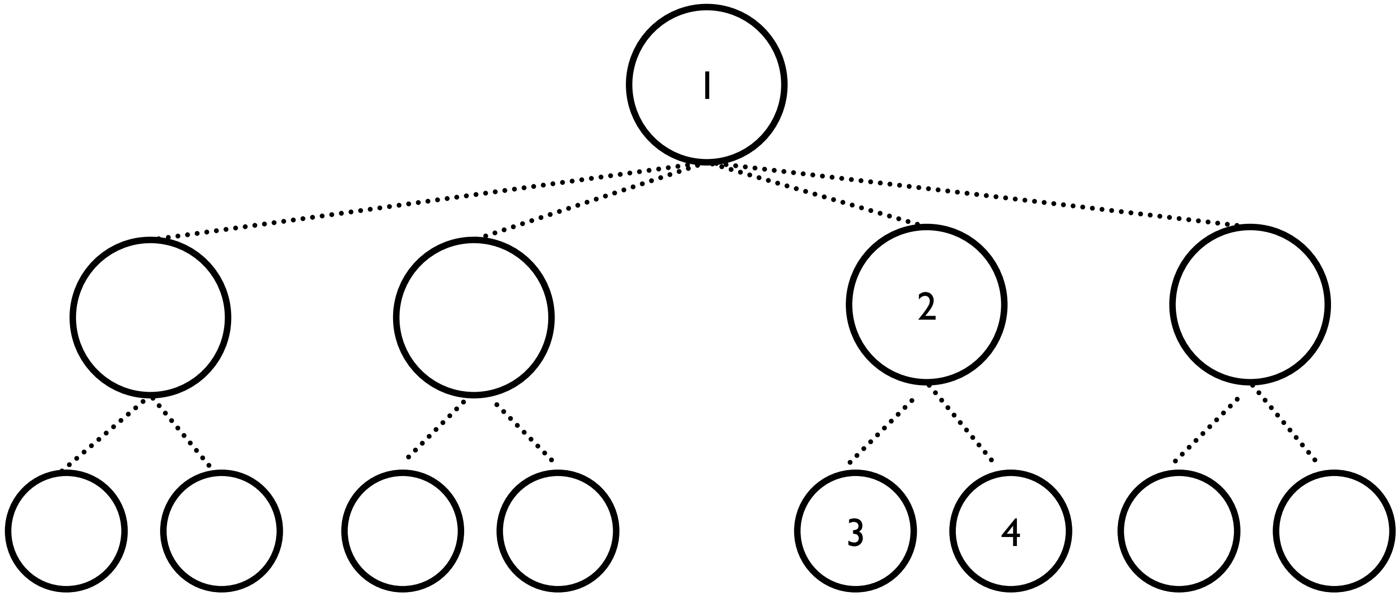
- Simply enclose all operation in atomic block
  - Semantics of atomic block: system ensures atomicity of logic within block

```
public Object get(Object key) {
      atomic {        // system guarantees atomicity
         return m.get(key);
      }
 }
```

- Good: thread-safe, easy to program
- What about performance and scalability?
  - Depends on the workload and implementation of atomic (to be discussed)

Another example: tree update by two threads

Goal: modify nodes 3 and 4 in a thread-safe way

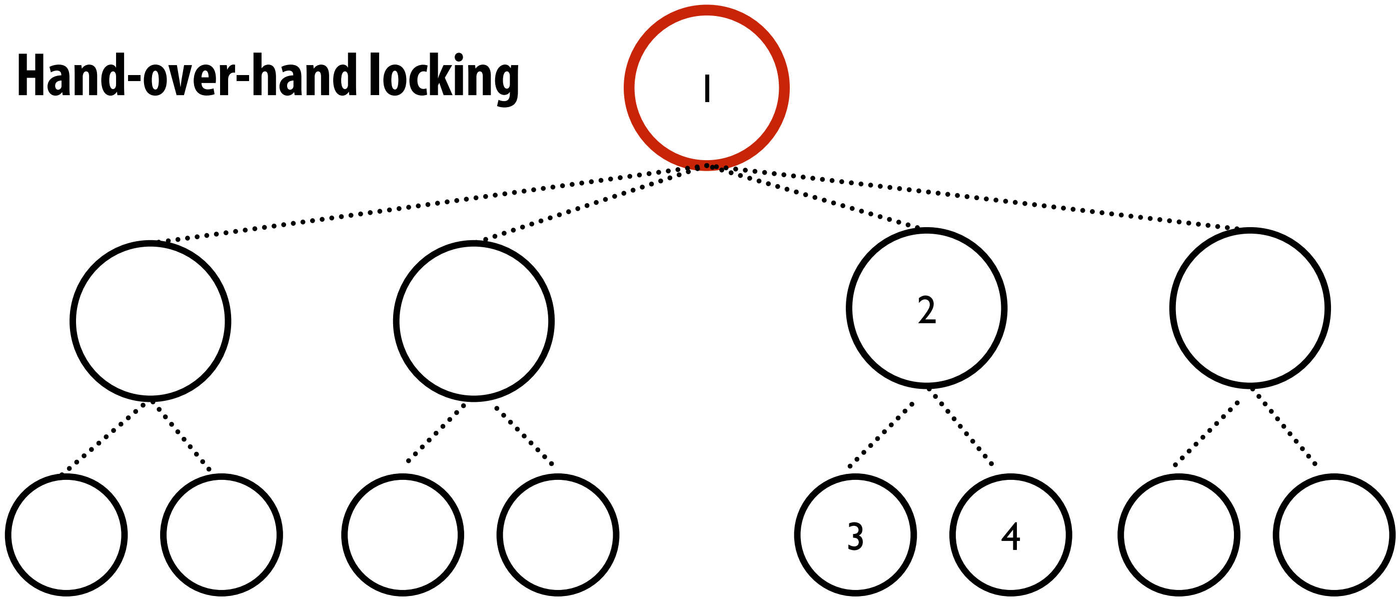


Slide credit: Austen McDonald

Fine-grained locking example

Goal: modify nodes 3 and 4 in a thread-safe way

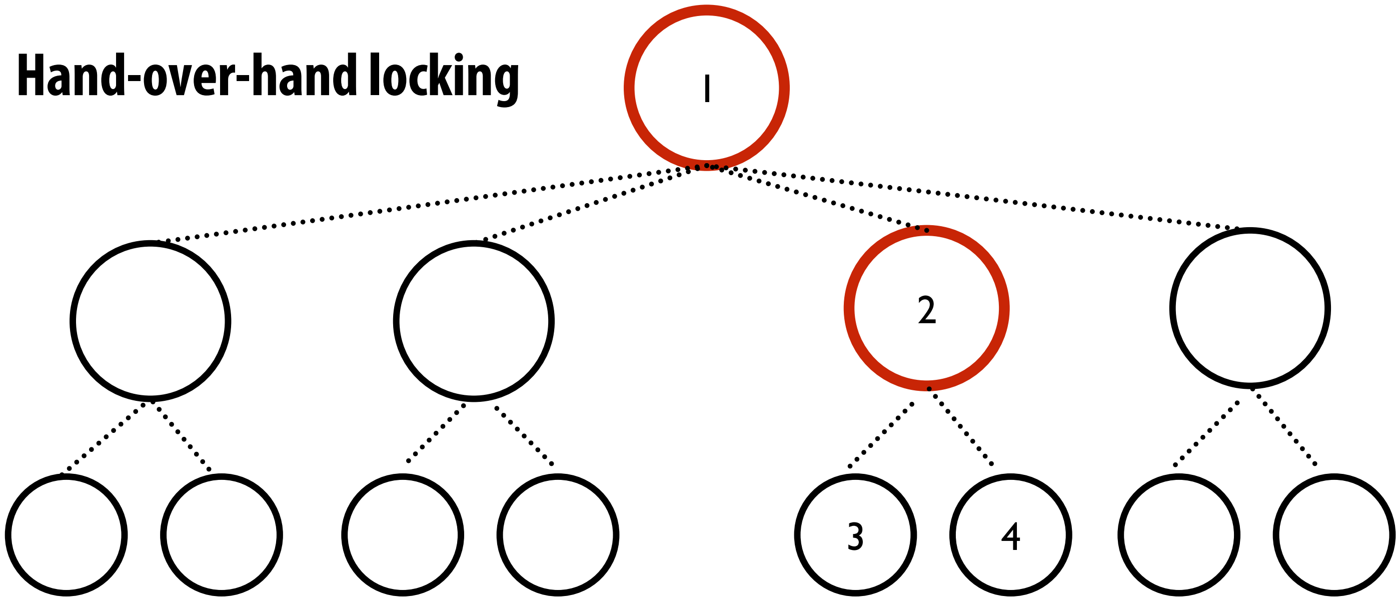
Hand-over-hand locking



Fine-grained locking example

Goal: modify nodes 3 and 4 in a thread-safe way

Hand-over-hand locking

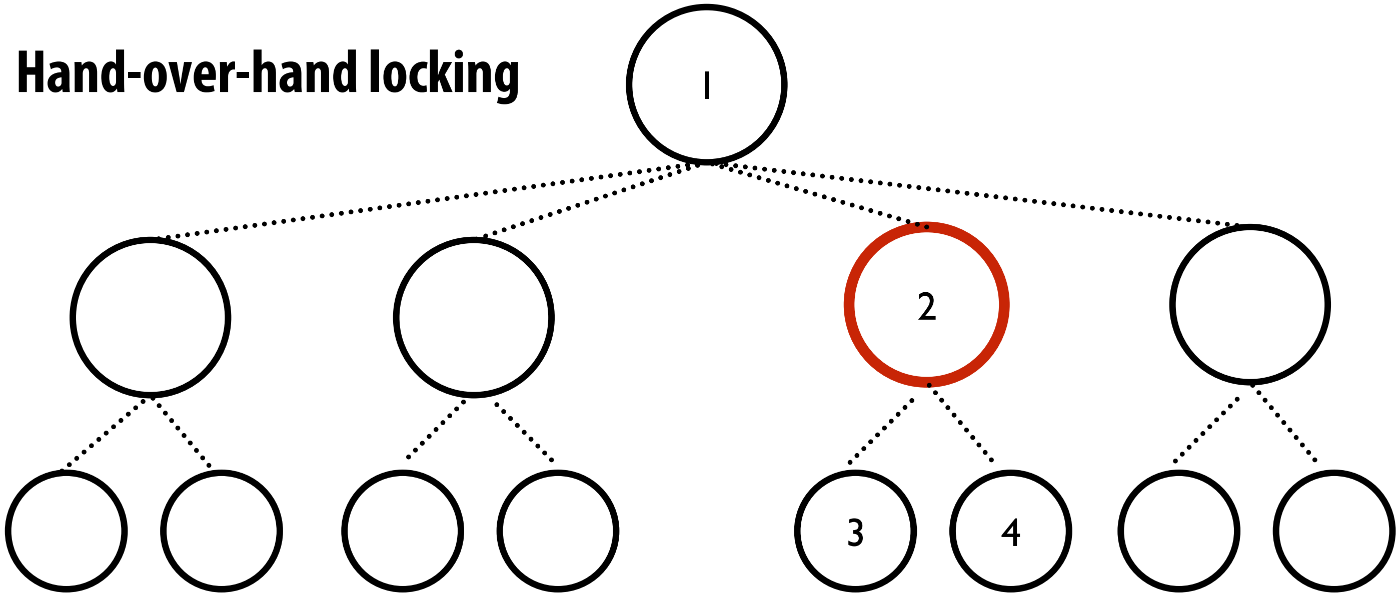


Slide credit: Austen McDonald

Fine-grained locking example

Goal: modify nodes 3 and 4 in a thread-safe way

Hand-over-hand locking

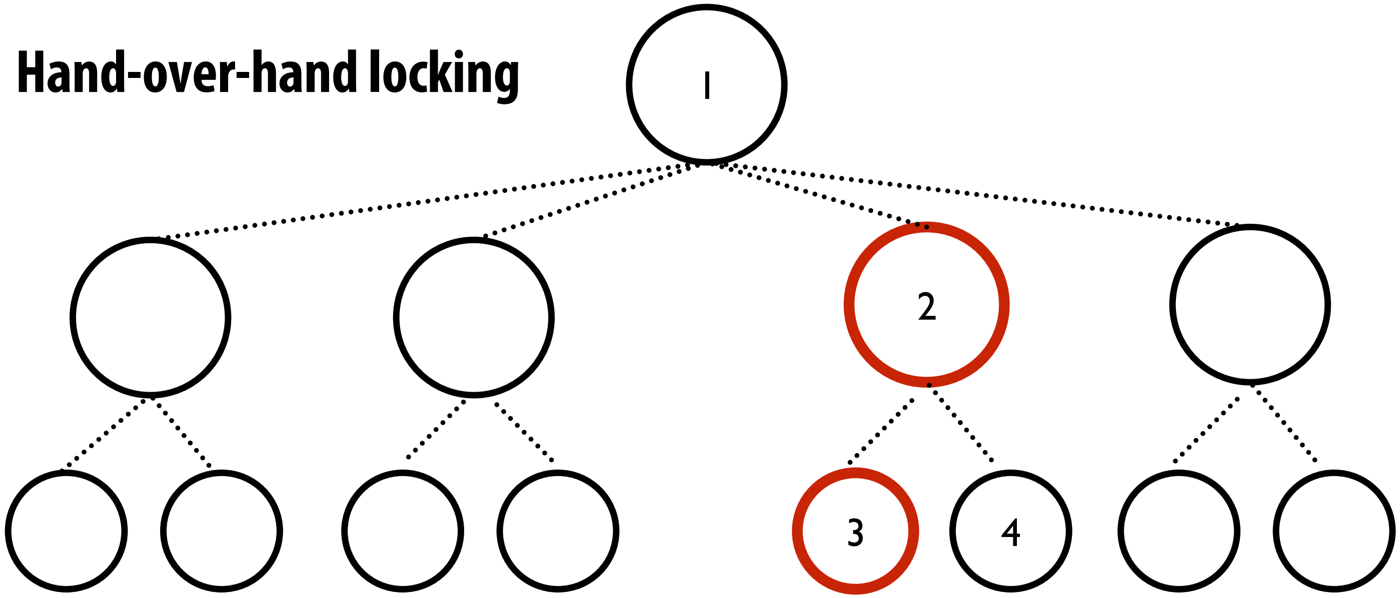


Slide credit: Austen McDonald

Fine-grained locking example

Goal: modify nodes 3 and 4 in a thread-safe way

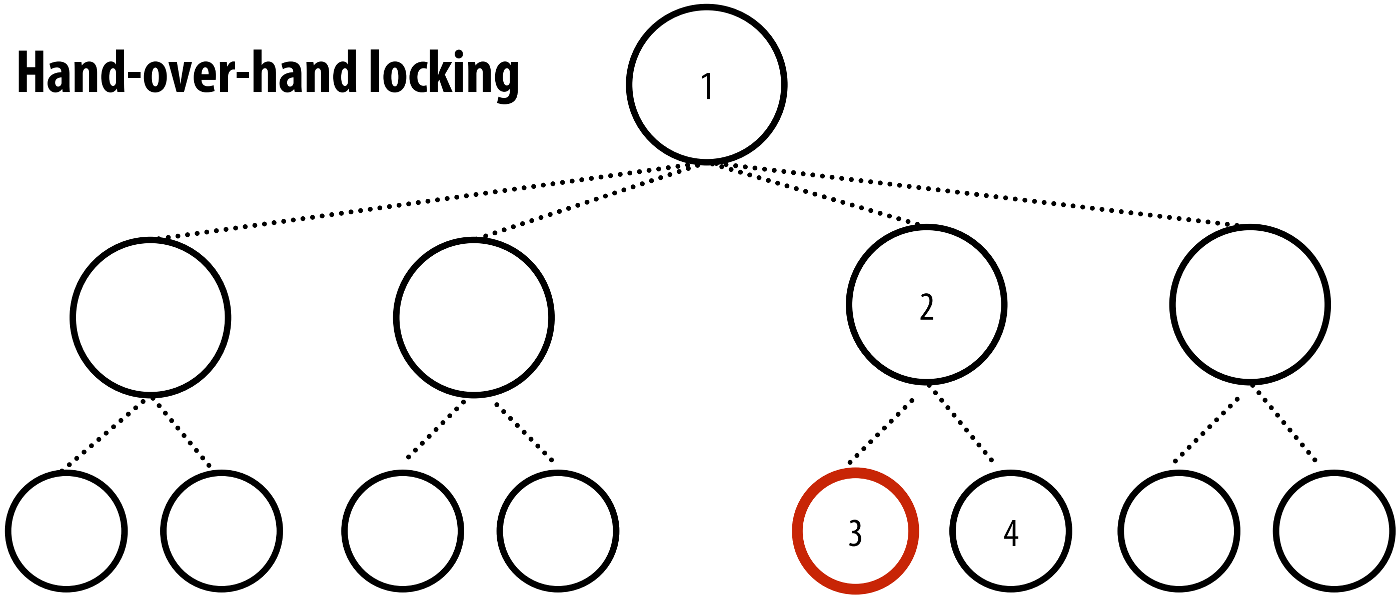
Hand-over-hand locking



Fine-grained locking example

Goal: modify nodes 3 and 4 in a thread-safe way

Hand-over-hand locking

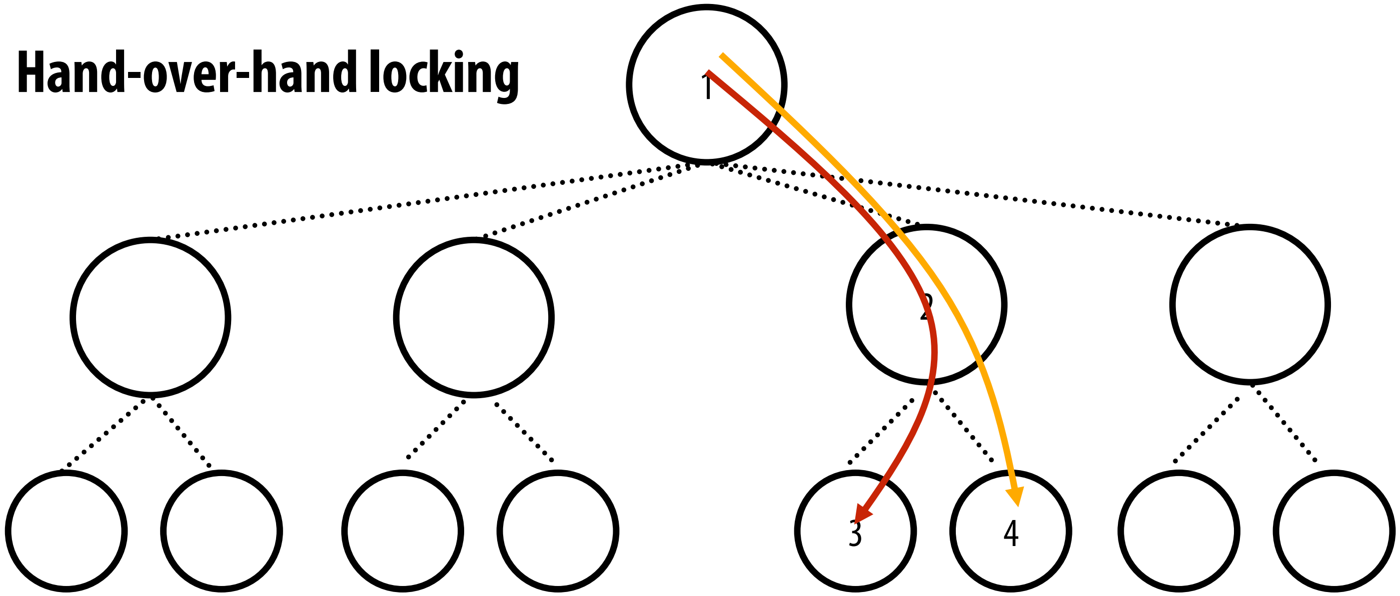


Slide credit: Austen McDonald

Fine-grained locking example

Goal: modify nodes 3 and 4 in a thread-safe way

Hand-over-hand locking



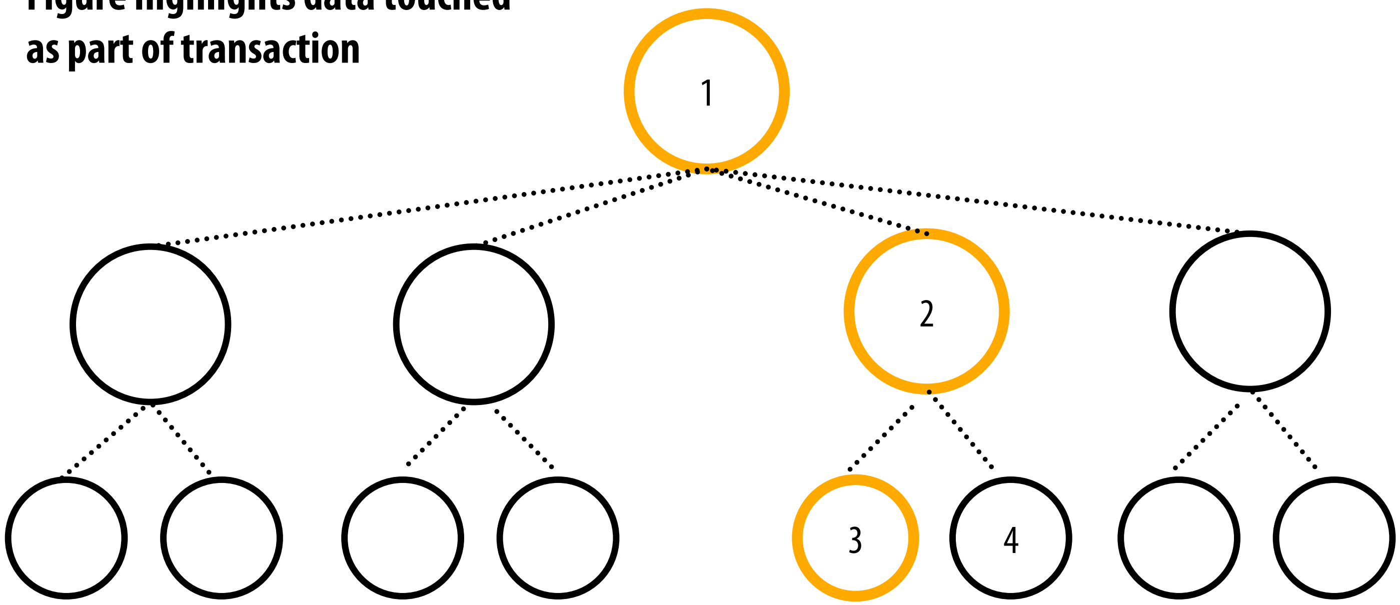
Locking can prevent concurrency

(here: locks on node 1 and 2 during update to node 3 could delay update to 4)

Slide credit: Austen McDonald

Transactions example

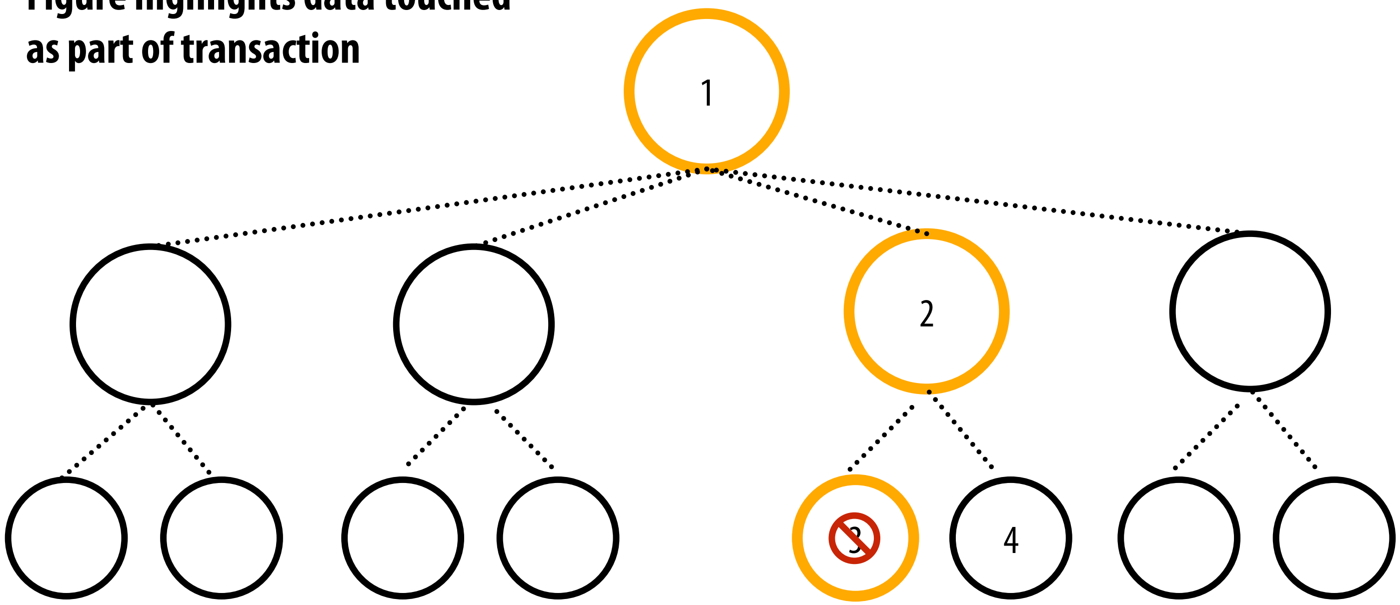
Figure highlights data touched as part of transaction



Transaction A
READ: 1, 2, 3

Transactions example

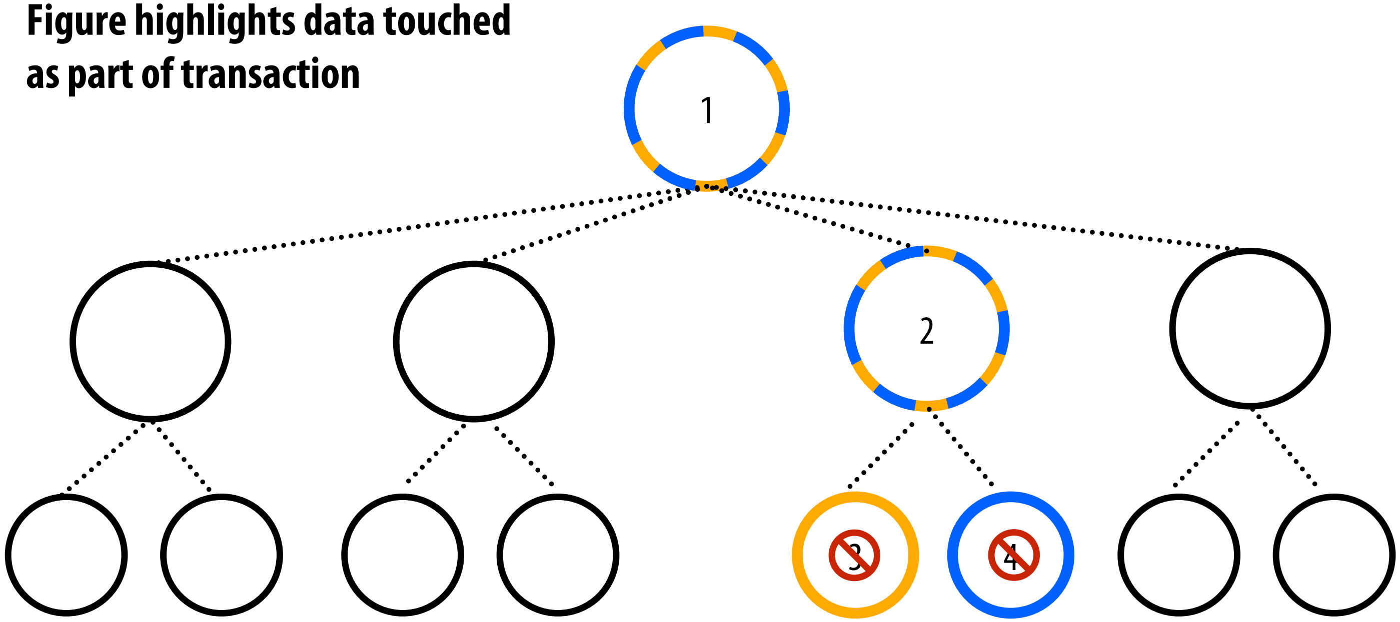
Figure highlights data touched as part of transaction



Transaction A
READ: 1, 2, 3
WRITE: 3

Transactions example

Figure highlights data touched as part of transaction



Transaction A
READ: 1, 2, 3
WRITE: 3

Transaction B
READ: 1, 2, 4
WRITE: 4

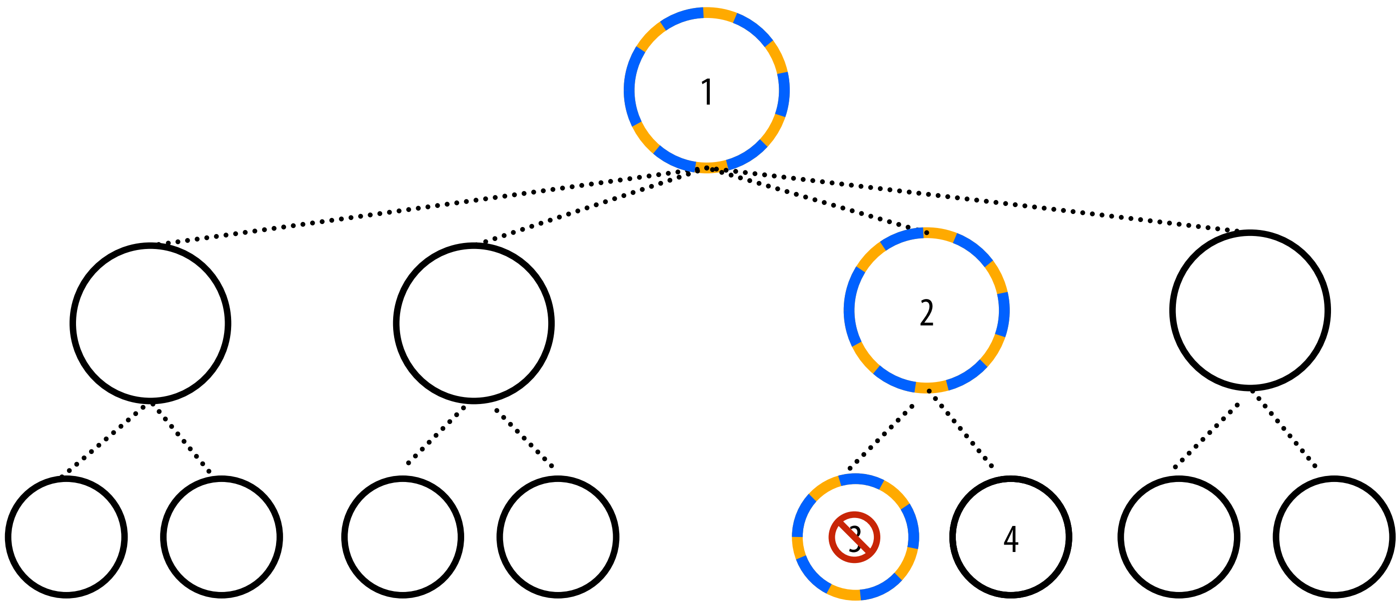
NO READ-WRITE or WRITE-WRITE conflicts!

(no transaction writes to data that is accessed by other transactions)

Slide credit: Austen McDonald

Transactions example #2

(Both transactions modify node 3)



Transaction A
READ: 1, 2, 3
WRITE: 3

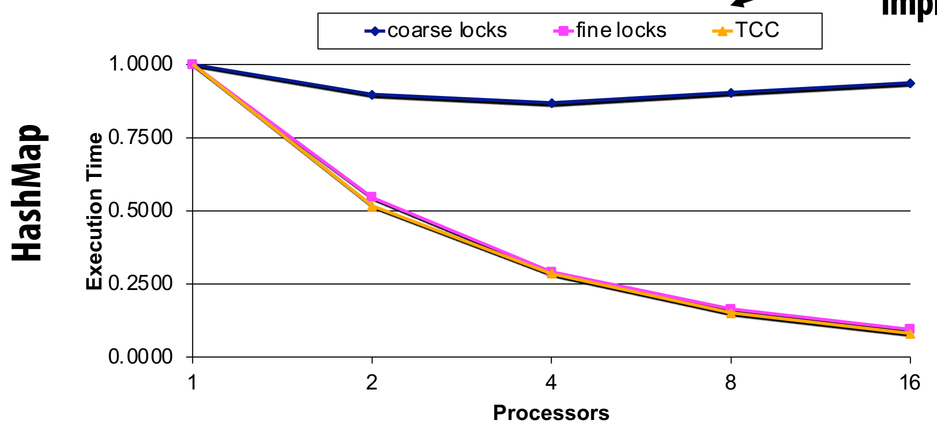
Transaction B
READ: 1, 2, 3
WRITE: 3

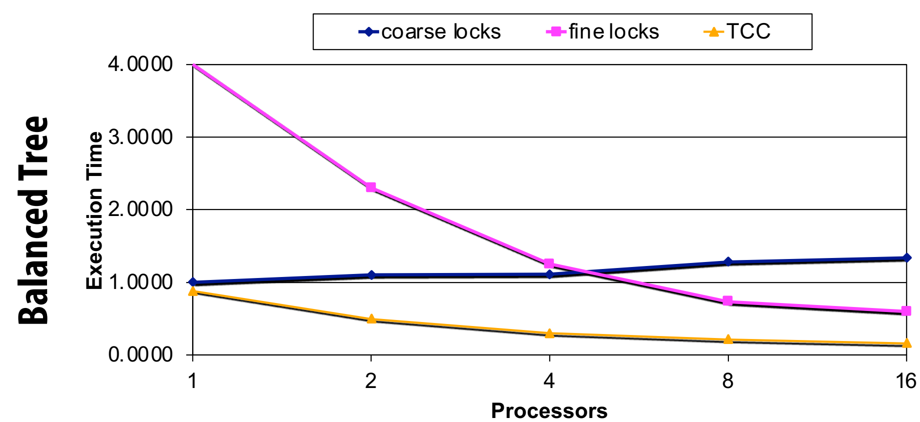
Conflicts exist: transactions must be serialized
(both transactions write to node 3)

Slide credit: Austen McDonald

Performance: locks vs. transactions

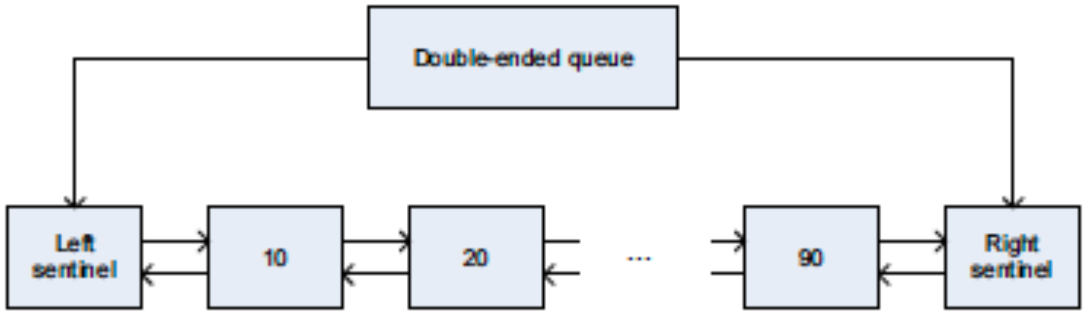
"TCC" is a TM system implemented in hardware





Atomic and Doubly-Linked List

- **Make PushLeft method on a doubly-linked list thread safe using atomic()**




```
void PushLeft(DQueue *q, int val) {
   QNode *qn = malloc(sizeof(QNode));
   qn->val = val;
   QNode *leftSentinel = q->left;
   QNode *oldLeftNode = leftSentinel->right;
   qn->left = leftSentinel;
   qn->right = oldLeftNode;
   leftSentinel->right = qn;
   oldLeftNode->left = qn;
}
```

Another motivation: failure atomicity

```
void transfer(A, B, amount) {
   synchronized(bank)
   {
      try {
        withdraw(A, amount);
        deposit(B, amount);
      }
      catch(exception1) { /* undo code 1*/ }
      catch(exception2) { /* undo code 2*/ }
    …
   }
}
```

- **Complexity of manually catching exceptions**
  - Programmer provides "undo" code on a case-by-case basis
  - Complexity: must track what to undo and how…
  - Some side-effects may become visible to other threads
    - E.g., an uncaught case can deadlock the system…

Failure atomicity: transactions

```
void transfer(A, B, amount)
{
   atomic {
      withdraw(A, amount);
      deposit(B, amount);
   }
}
```

- System now responsible for processing exceptions
  - All exceptions (except those explicitly managed by the programmer)
  - Transaction is aborted and memory updates are undone
  - Recall: a transaction either commits or it doesn't: no partial updates are visible to other threads
    - E.g., no locks held by a failing threads…

Another motivation: composability

```
void transfer(A, B, amount)
{
   synchronized(A) {
     synchronized(B) {
       withdraw(A, amount);
       deposit(B, amount);
     }
   }
}
```

Thread 0:
```
transfer(A, B, 100)
```

Thread 1:
```
transfer(B, A, 200)
```

DEADLOCK!

- **Composing lock-based code can be tricky**
  - Requires system-wide policies to get correct
  - System-wide policies can break software modularity

- **Programmer caught between a lock and a hard (to implement) place !**
  - Coarse-grain locks: low performance
  - Fine-grain locking: good for performance, but mistakes can lead to deadlock

Composability: locks

```
void transfer(A, B, amount) {
   synchronized(A) {
     synchronized(B) {
       withdraw(A, amount);
       deposit(B, amount);
     }
   }
}
```

DEADLOCK!

```
void transfer2(B, A, amount) {
   synchronized(B) {
     synchronized(A) {
       withdraw(A, 2*amount);
       deposit(B, 2*amount);
     }
   }
}
```

- **Composing lock-based code can be tricky**
  - Requires system-wide policies to get correct
  - System-wide policies can break software modularity

- **Programmer caught between and lock and a hard (to implement) place**
  - Coarse-grain locks: low performance
  - Fine-grain locking: good for performance, but mistakes can lead to deadlock

Composability: transactions

```
void transfer(A, B, amount) {
   atomic {
     withdraw(A, amount);
     deposit(B, amount);
   }
}
```

Thread 0:

`transfer(A, B, 100)`

Thread 1:

`transfer(B, A, 200)`

- **Transactions compose gracefully (in theory)**
  - Programmer declares global intent (atomic execution of transfer)
    - No need to know about global implementation strategy
  - Transaction in `transfer` subsumes any defined in `withdraw` and `deposit`
    - Outermost transaction defines atomicity boundary

- **System manages concurrency as well as possible**
  - Serialization for transfer(A, B, 100) and transfer(B, A, 200)
  - Concurrency for transfer(A, B, 100) and transfer(C, D, 200)

Advantages (promise) of transactional memory

- **Easy to use synchronization construct**
  - It is difficult for programmers to get synchronization right
  - Programmer declares need for atomicity, system implements it well
  - Claim: transactions are as easy to use as coarse-grain locks
- **Often performs as well as fine-grained locks**
  - Provides automatic read-read concurrency and fine-grained concurrency
  - Performance portability: locking scheme for four CPUs may not be the best scheme for 64 CPUs
  - Productivity argument for transactional memory: system support for transactions can achieve 90% of the benefit of expert programming with fined-grained locks, with 10% of the development time
- **Failure atomicity and recovery**
  - No lost locks when a thread fails
  - Failure recovery = transaction abort + restart
- **Composability**
  - Safe and scalable composition of software modules

Self-check: atomic { } ≠ lock() + unlock()

- **The difference**
  - Atomic: high-level declaration of atomicity
    - Does not specify implementation of atomicity
  - Lock: low-level blocking primitive
    - Does not provide atomicity or isolation on its own

Make sure you understand this difference in semantics!

- **Keep in mind**
  - Locks can be used to implement an `atomic` block but...
  - Locks can be used for purposes beyond atomicity
    - Cannot replace all uses of locks with atomic regions
  - `Atomic` eliminates many data races, but programming with atomic blocks can still suffer from atomicity violations: e.g., programmer erroneous splits sequence that <u>should be</u> atomic into two atomic blocks

What about replacing synchronized with atomic in this example?

```
// Thread 1
synchronized(lock1)
{
  …
  flagA = true;
  while (flagB == 0);
  …
}
```

```
// Thread 2
synchronized(lock2)
{
  …
  flagB = true;
  while (flagA == 0);
  …
}
```

Atomicity violation due to programmer error

```
// Thread 1
atomic
{
  …
  ptr = A;
  …
}

atomic
{
  B = ptr->field;
}
```

```
// Thread 2
atomic
{
  …
  ptr = NULL;
}
```

- **Programmer mistake: logically atomic code sequence (in thread 1) is erroneously separated into two atomic blocks (allowing another thread to set pointer to NULL in between)**

Implementing transactional memory

Recall transactional semantics

- **Atomicity (all or nothing)**
  - At commit, all memory writes take effect at once
  - In event of abort, none of the writes appear to take effect
- **Isolation**
  - No other code can observe writes before commit
- **Serializability**
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed though

TM implementation basics

- TM systems must provide atomicity and isolation
  - While maintaining as much concurrency as possible

- Two key implementation questions
  - Data versioning policy: How does the system manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions?

  - Conflict detection policy: how/when does the system determine that two concurrent transactions conflict?

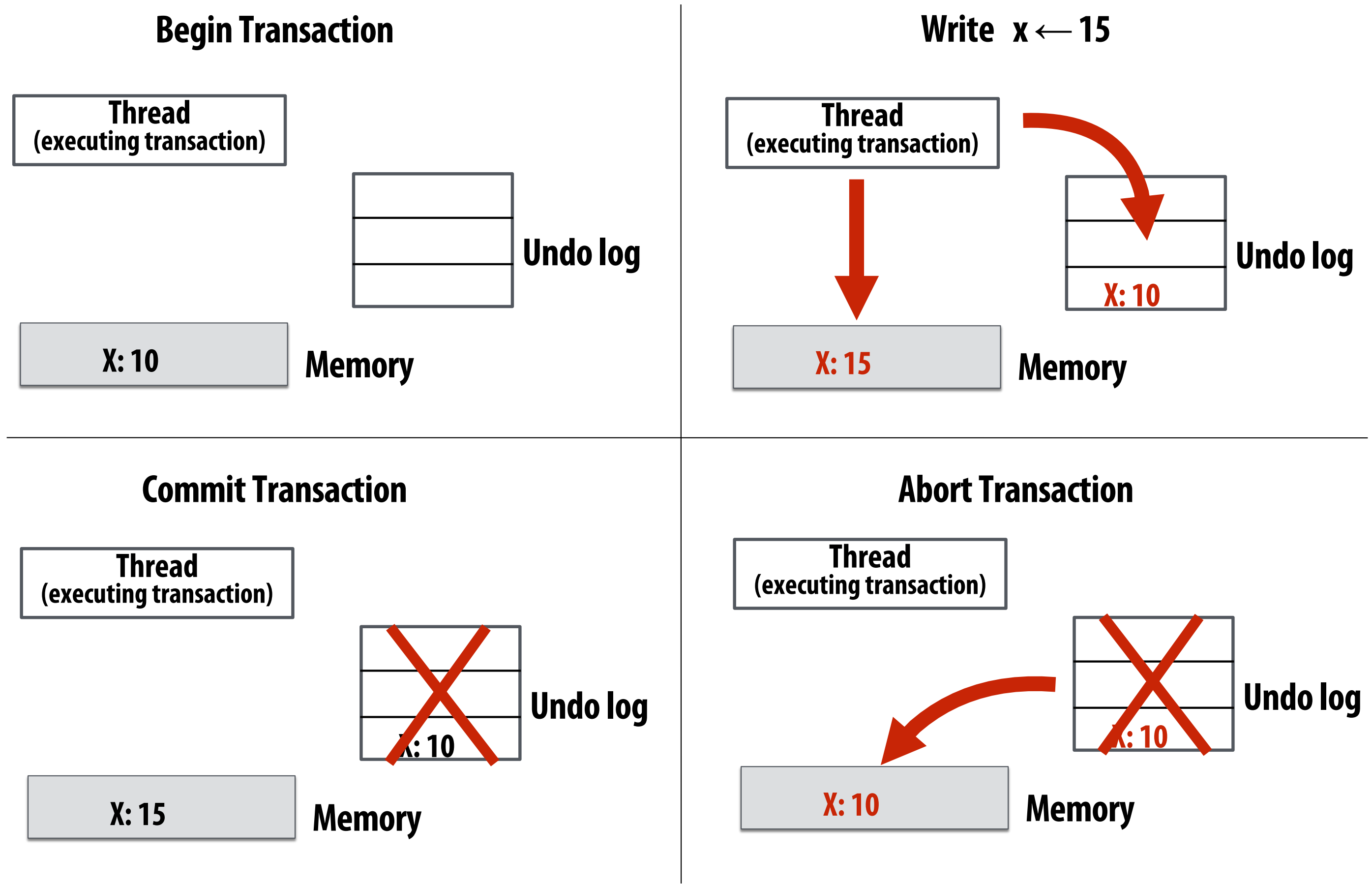
Data versioning policy

Manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions

1. **Eager versioning (undo-log based)**
2. **Lazy versioning (write-buffer based)**

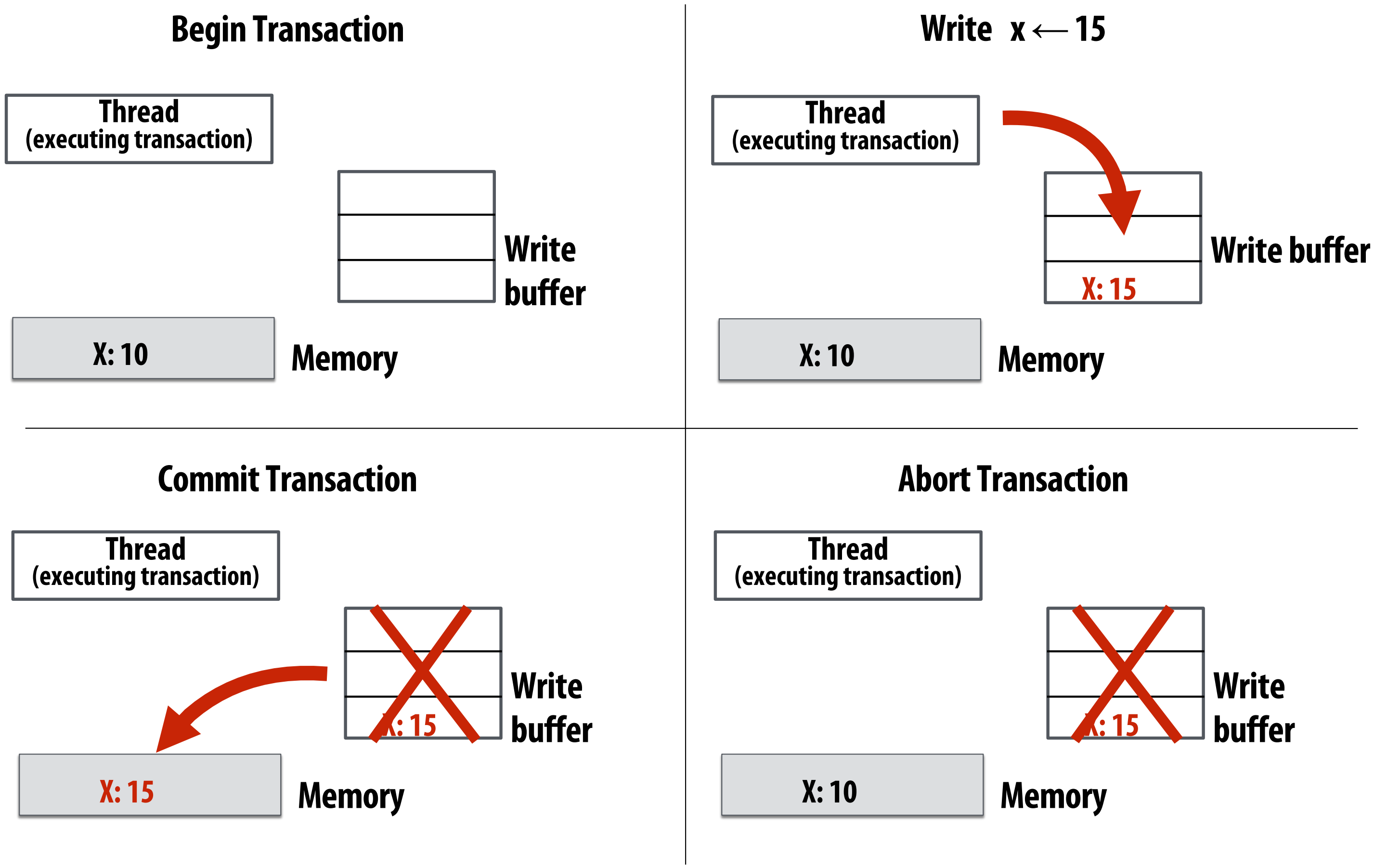
Eager versioning

Update memory immediately, maintain "undo log" in case of abort



Lazy versioning

Log memory updates in transaction write buffer, flush buffer on commit



Data versioning

- Goal: manage uncommitted (new) and committed (old) versions of data for concurrent transactions
- Eager versioning (undo-log based)
  - Update memory location directly on write
  - Maintain undo information in a log (incurs per-store overhead)
  - Good: faster commit (data is already in memory)
  - Bad: slower aborts, fault tolerance issues (consider crash in middle of transaction)

Eager versioning philosophy: write to memory immediately, hoping transaction won't abort (but deal with aborts when you have to)

- Lazy versioning (write-buffer based)
  - Buffer data in a write buffer until commit
  - Update actual memory location on commit
  - Good: faster abort (just clear log), no fault tolerance issues
  - Bad: slower commits

Lazy versioning philosophy: only write to memory when you have to

Conflict detection

- **Must detect and handle conflicts between transactions**
  - Read-write conflict: transaction A reads address X, which was written to by pending (but not yet committed) transaction B
  - Write-write conflict: transactions A and B are both pending, and both write to address X

- **System must track a transaction's read set and write set**
  - Read-set: addresses read during the transaction
  - Write-set: addresses written during the transaction

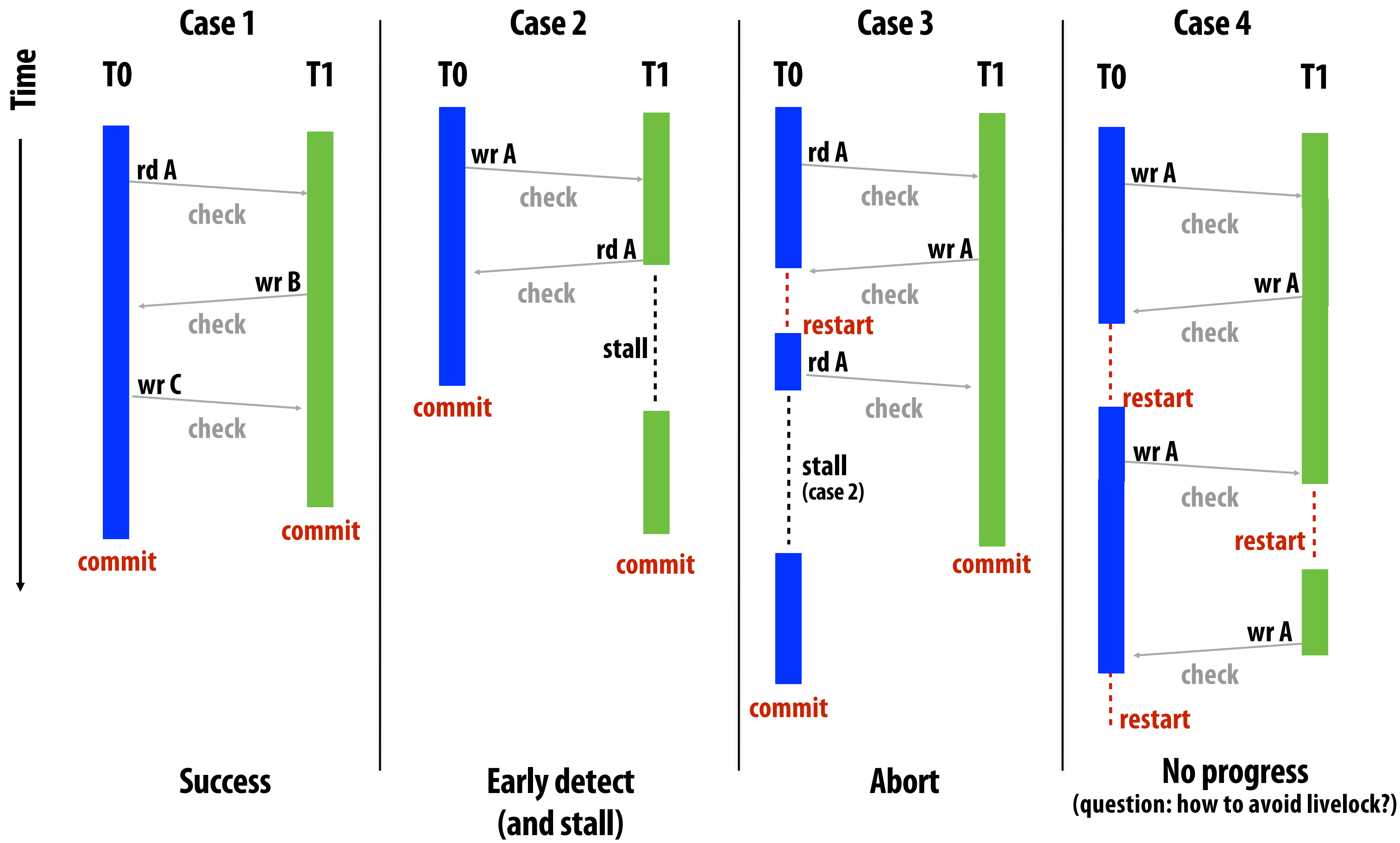
Pessimistic detection

- Check for conflicts (immediately) during loads or stores
  - Philosophy: "I suspect conflicts might happen, so let's always check to see if one has occurred after each memory operation… if I'm going to have to roll back, might as well do it now to avoid wasted work."

- "Contention manager" decides to stall or abort transaction when a conflict is detected
  - Various policies to handle common case fast

Pessimistic detection examples

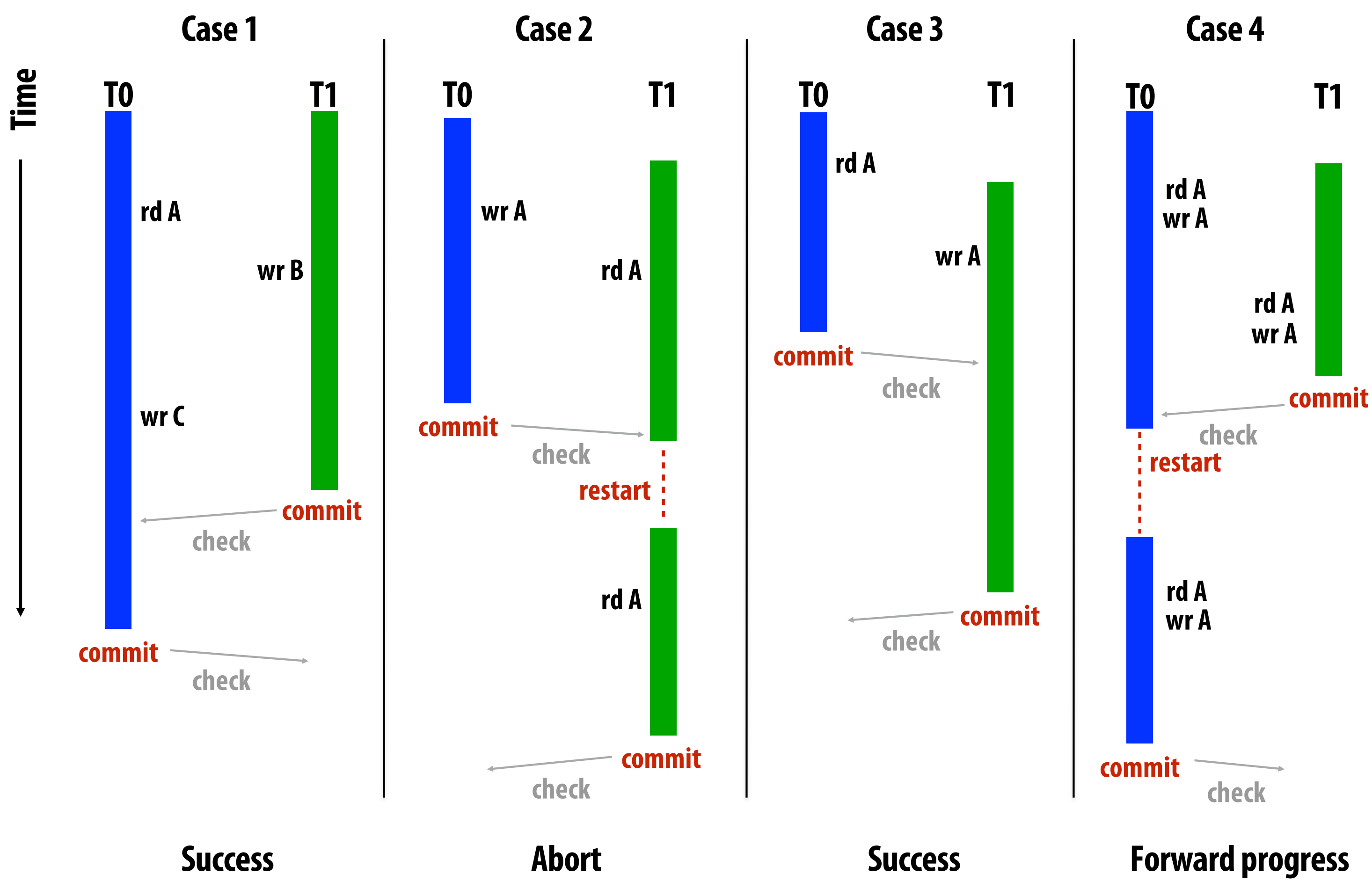
Note: diagrams assume "aggressive" contention manager on writes: writer wins, so other transactions abort)



Optimistic detection

- Detect conflicts when a transaction attempts to commit
  - Intuition: "Let's hope for the best and sort out all the conflicts only when the transaction tries to commit"
- On a conflict, give priority to committing transaction
  - Other transactions may abort later on

Optimistic detection



Conflict detection trade-offs

- **Pessimistic conflict detection (a.k.a. "eager")**
  - Good: detect conflicts early (undo less work, turn some aborts to stalls)
  - Bad: no forward progress guarantees, more aborts in some cases
  - Bad: fine-grained communication (check on each load/store)
  - Bad: detection on critical path
- **Optimistic conflict detection (a.k.a."lazy" or "commit")**
  - Good: forward progress guarantees
  - Good: bulk communication and conflict detection
  - Bad: detects conflicts late, can still have fairness problems

TM implementation space (examples)

- Hardware TM systems
  - Lazy + optimistic: Stanford TCC
  - Lazy + pessimistic: MIT LTM, Intel VTM
  - Eager + pessimistic: Wisconsin LogTM
  - Eager + optimistic: not practical

- Software TM systems
  - Lazy + optimistic (rd/wr): Sun TL2
  - Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
  - Eager + optimistic (rd)/pessimistic (wr): Intel STM
  - Eager + pessimistic (rd/wr): Intel STM

- Optimal design remains an open question
  - May be different for HW, SW, and hybrid

Software Transactional Memory

```
atomic {
   a.x = t1
   a.y = t2
   if (a.z == 0) {
   a.x = 0
   a.z = t3
   }
}
```

```
tmTxnBegin()
tmWr(&a.x, t1)
tmWr(&a.y, t2)
if (tmRd(&a.z) != 0) {
   tmWr(&a.x, 0);
   tmWr(&a.z, t3)
}
tmTxnCommit()
```

- Software barriers (STM function call) for TM bookkeeping
  - Versioning, read/write-set tracking, commit, . . .
  - Using locks, timestamps, data copying, . . .
- Requires function cloning or dynamic translation
  - Function used inside and outside of transaction

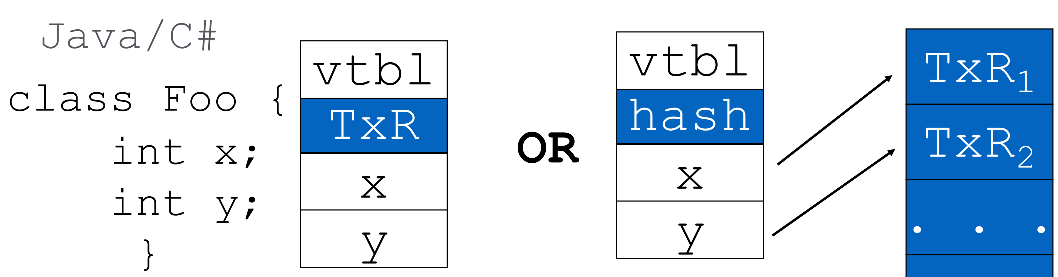
STM Runtime Data Structures

- Transaction descriptor (per-thread)
  - Used for conflict detection, commit, abort, …
  - Includes the read set, write set, undo log or write buffer

- Transaction record (per data)
  - Pointer-sized record guarding shared data
  - Tracks transactional state of data
    - Shared: accessed by multiple readers
      - Using version number or shared reader lock
    - Exclusive: access by one writer
      - Using writer lock that points to owner
    - BTW: same way that HW cache coherence works

Mapping Data to Transaction Records

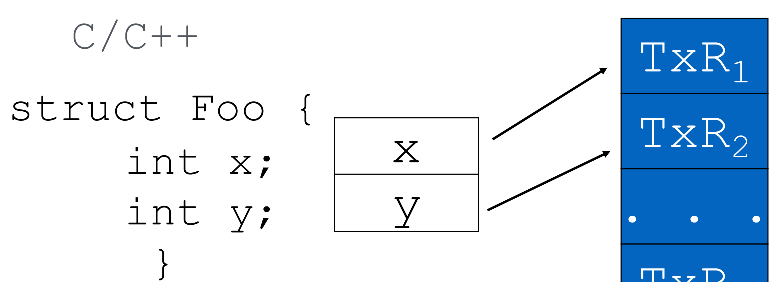
Every data item has an associated transaction record



Embed in each object

Hash fields or
array elements to global table

f(obj.hash, field.index)



Address-based hash
into global table

Cache-line or word
granularity

What's the tradeoff?

Conflict Detection Granularity

- **Object granularity**
  - Low overhead mapping operation
  - Exposes optimization opportunities
  - False conflicts (e.g. Txn 1 and Txn 2)
- **Element/field granularity (word)**
  - Reduces false conflicts
  - Improves concurrency (e.g. Txn 1 and Txn 2)
  - Increased overhead (time/space)
- **Cache line granularity (multiple words)**
  - Matches hardware TM
  - Reduces storage overhead of transactional records
  - Hard for programmer & compiler to analyze
- **Mix & match per type basis**
  - E.g., element-level for arrays, object-level for non-arrays

Txn 1

a.x = …

a.y = …

Txn 2

… = … a.z …

An Example STM Algorithm

- Based on Intel's McRT STM [PPoPP'06, PLDI'06, CGO'07]
  - Eager versioning, optimistic reads, pessimistic writes

- Based on timestamp for version tracking
  - Global timestamp
    - Incremented when a writing xaction commits
  - Local timestamp per xaction
    - Global timestamp value when xaction last validated

- Transaction record (32-bit)
  - LS bit: 0 if writer-locked, 1 if not locked
  - MS bits
    - Timestamp (version number) of last commit if not locked
    - Pointer to owner xaction if locked

STM Operations

- **STM read (optimistic)**
  - Direct read of memory location (eager)
  - Validate read data
    - Check if unlocked and data version ≤ local timestamp
    - If not, validate all data in read set for consistency
  - Insert in read set
  - Return value

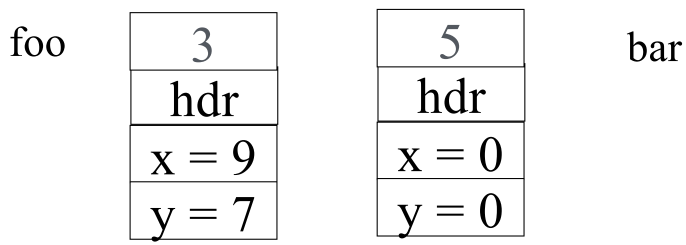
- **STM write (pessimistic)**
  - Validate data
    - Check if unlocked and data version ≤ local timestamp
  - Acquire lock
  - Insert in write set
  - Create undo log entry
  - Write data in place (eager)

STM Operations (cont)

- Read-set validation
  - Get global timestamp
  - For each item in the read set
    - If locked by other or data version > local timestamp, abort
  - Set local timestamp to global timestamp from initial step

- STM commit
  - Atomically increment global timestamp by 2 (LSb used for write-lock)
  - If preincremented (old) global timestamp > local timestamp, validate read-set
    - Check for recently committed transactions
  - For each item in the write set
    - Release the lock and set version number to global timestamp

STM Example



X1

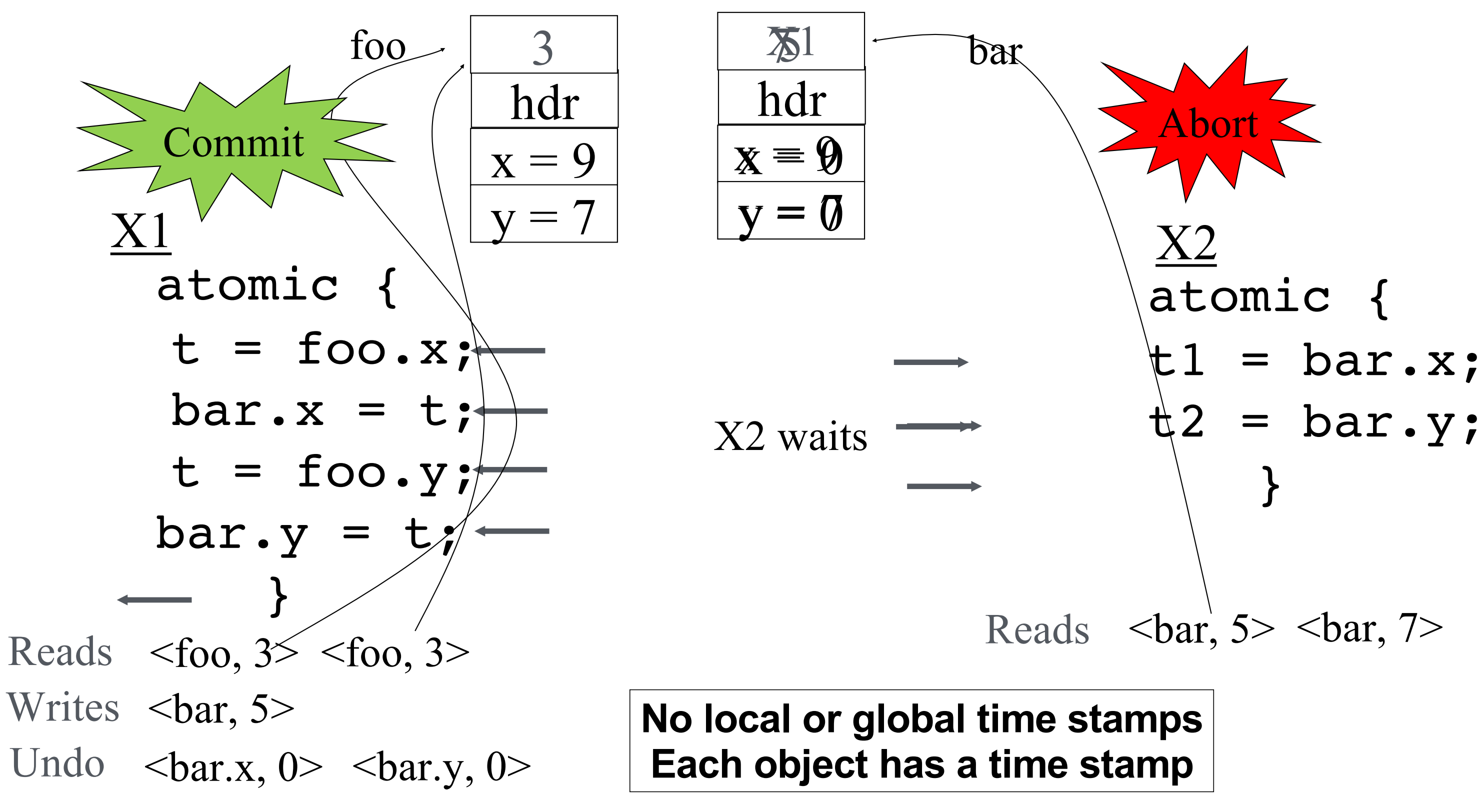
```
atomic {
 t = foo.x;
 bar.x = t;
 t = foo.y;
 bar.y = t;
    }
```

X2

```
atomic {
t1 = bar.x;
t2 = bar.y;
    }
```

- **X1 copies object foo into object bar**
- **X2 should read bar as [0,0] or [9,7]**

STM Example



No local or global time stamps
Each object has a time stamp