Transactional Memory

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

Vision for the Future

In this course we covered ….

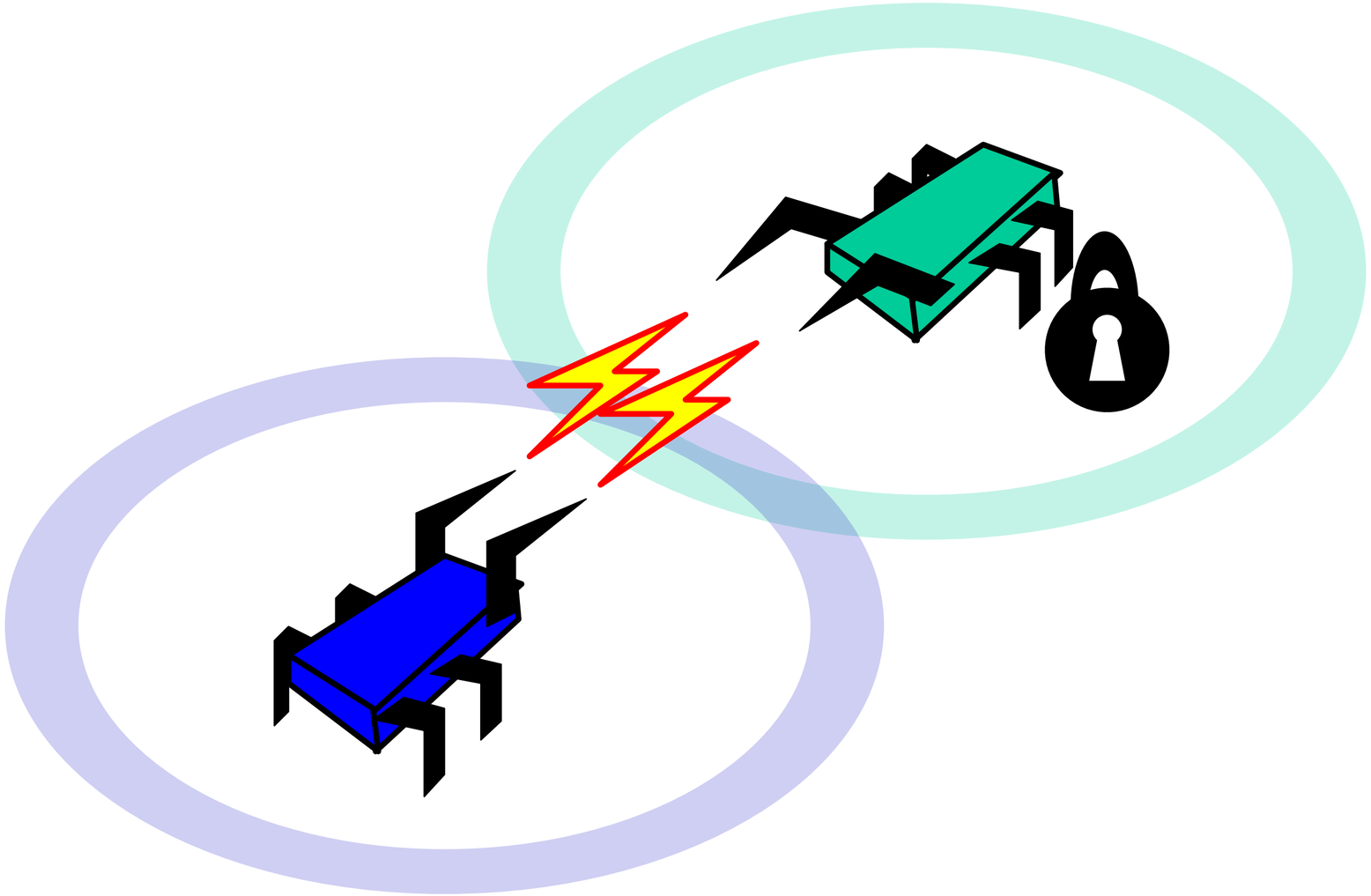
Nevertheless …

Concurrent programming is still too hard …

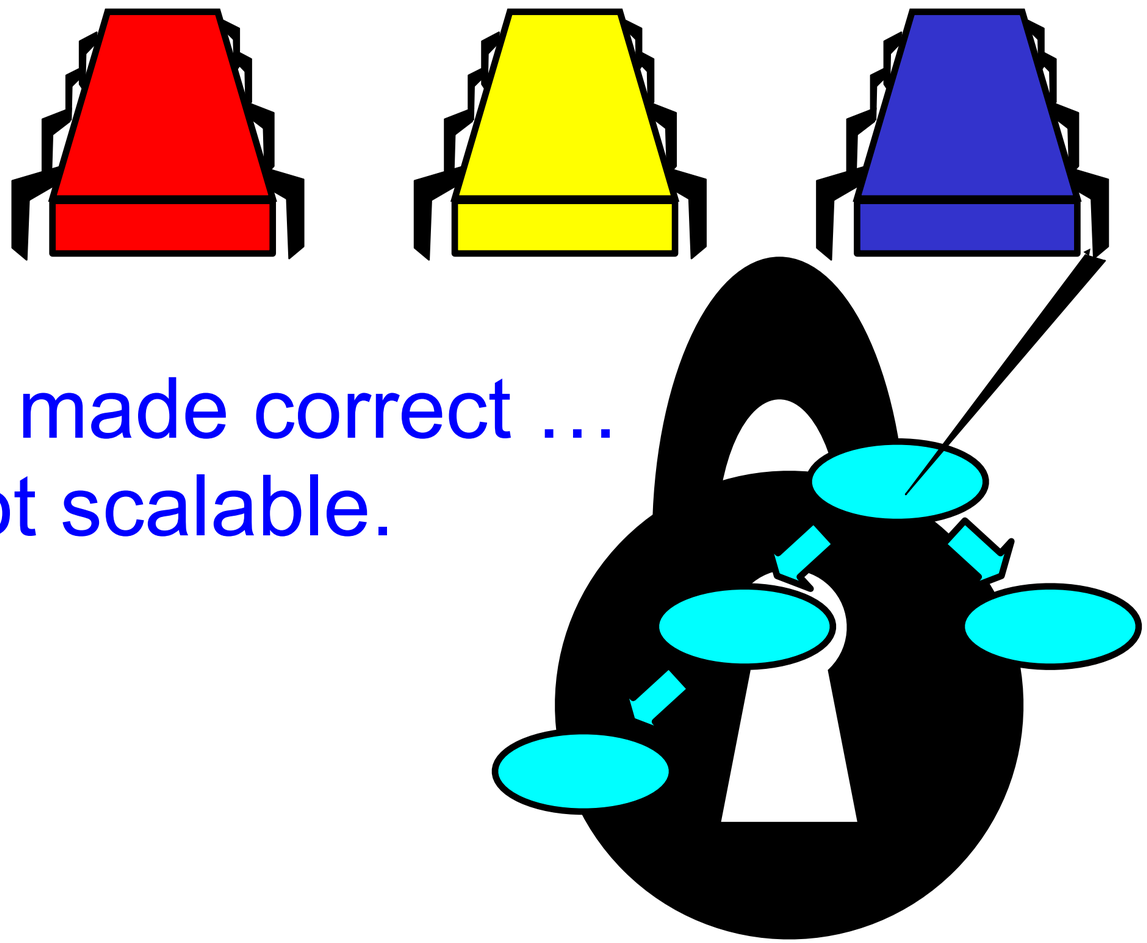
Here we explore why this is ….

And what we can do about it.

Locking

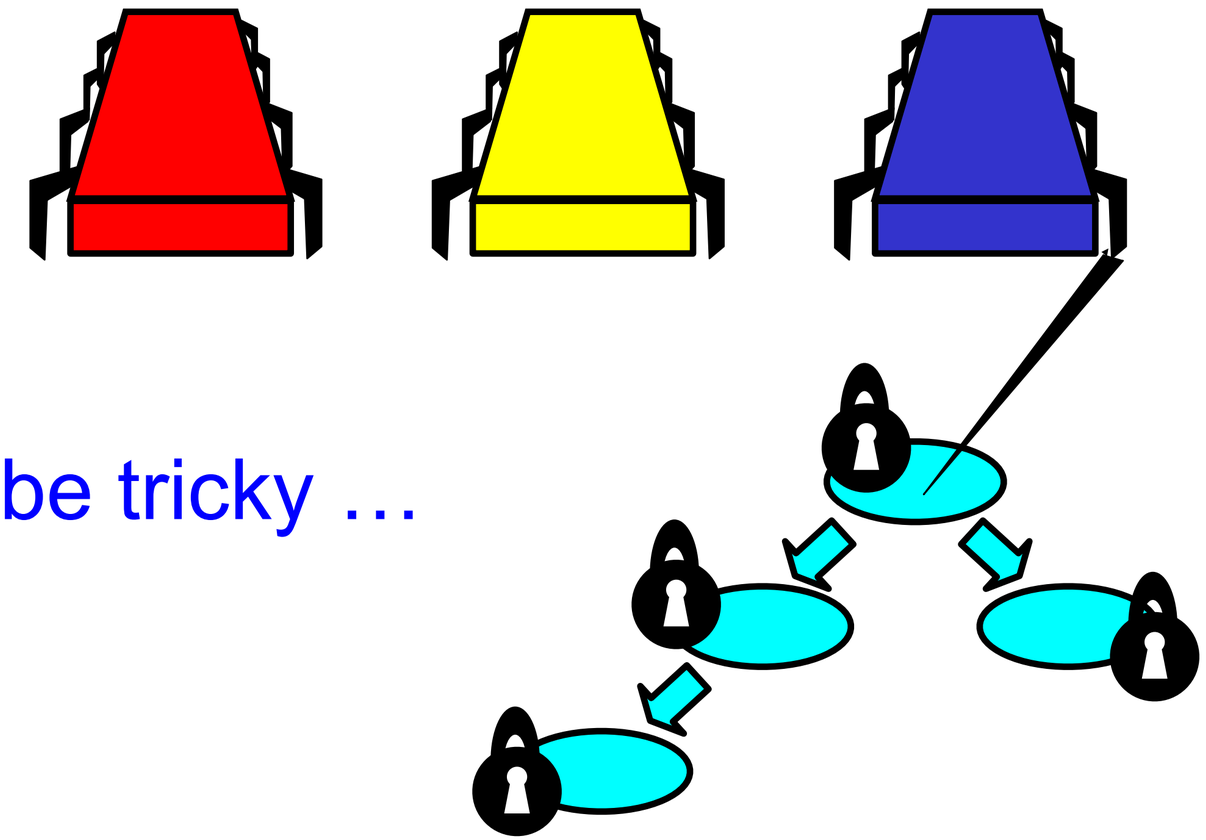


Coarse-Grained Locking



Easily made correct …
But not scalable.

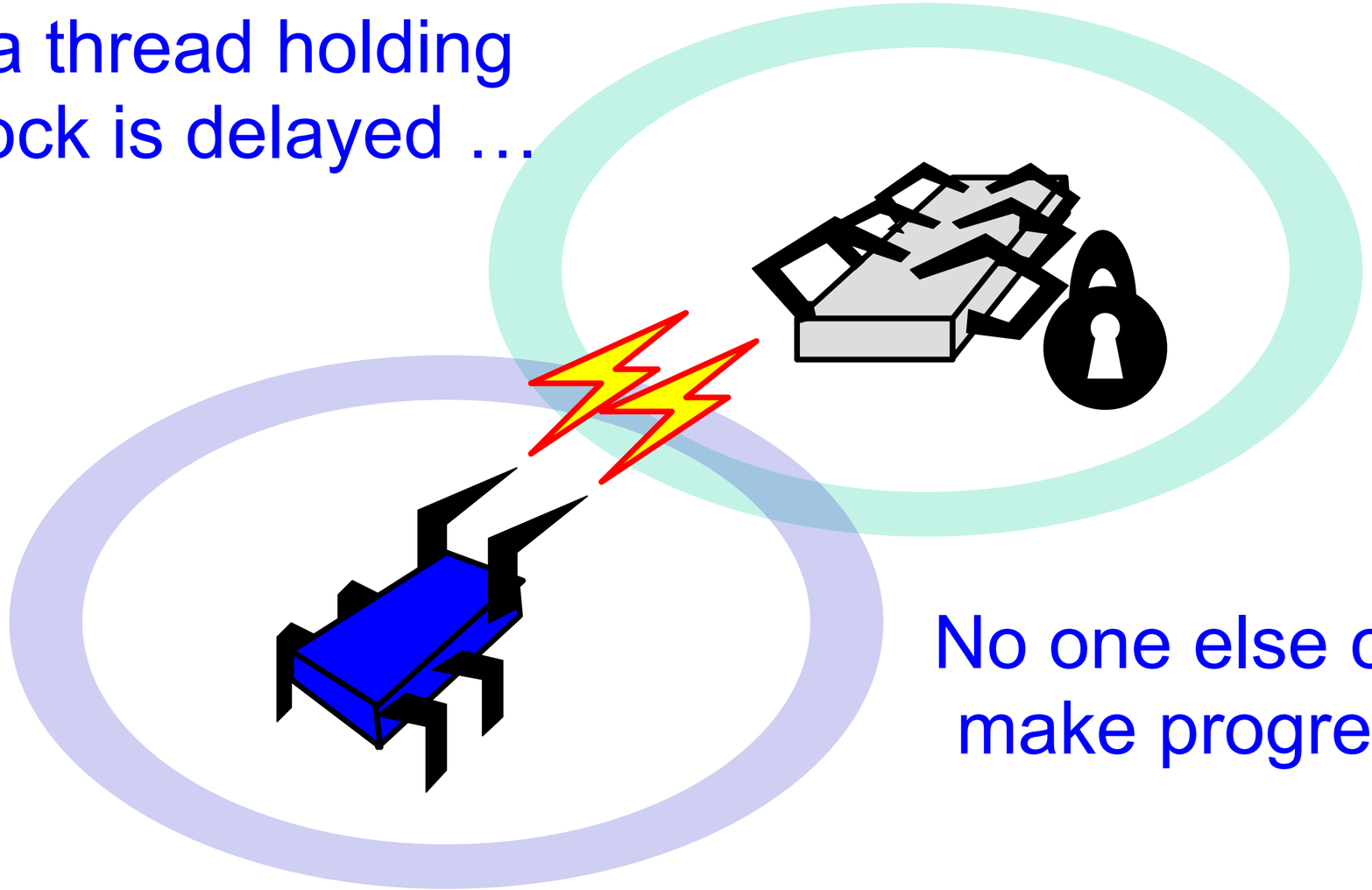
Fine-Grained Locking



Can be tricky …

Locks are not Robust

If a thread holding
a lock is delayed …



No one else can
make progress

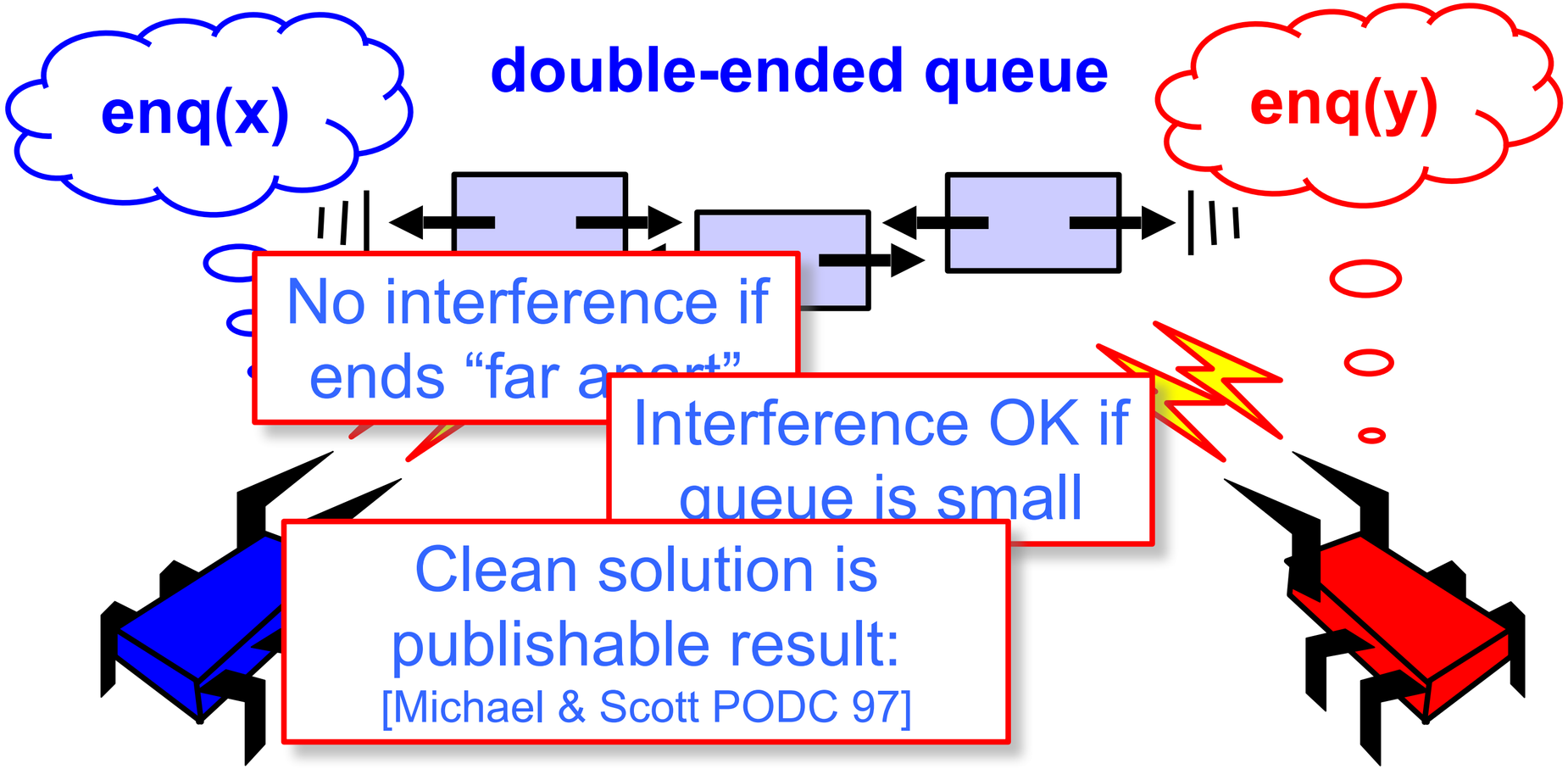
Locking Relies on Conventions

- Relation between
  - Lock bit and object bits
  - Exists only in programmer's m

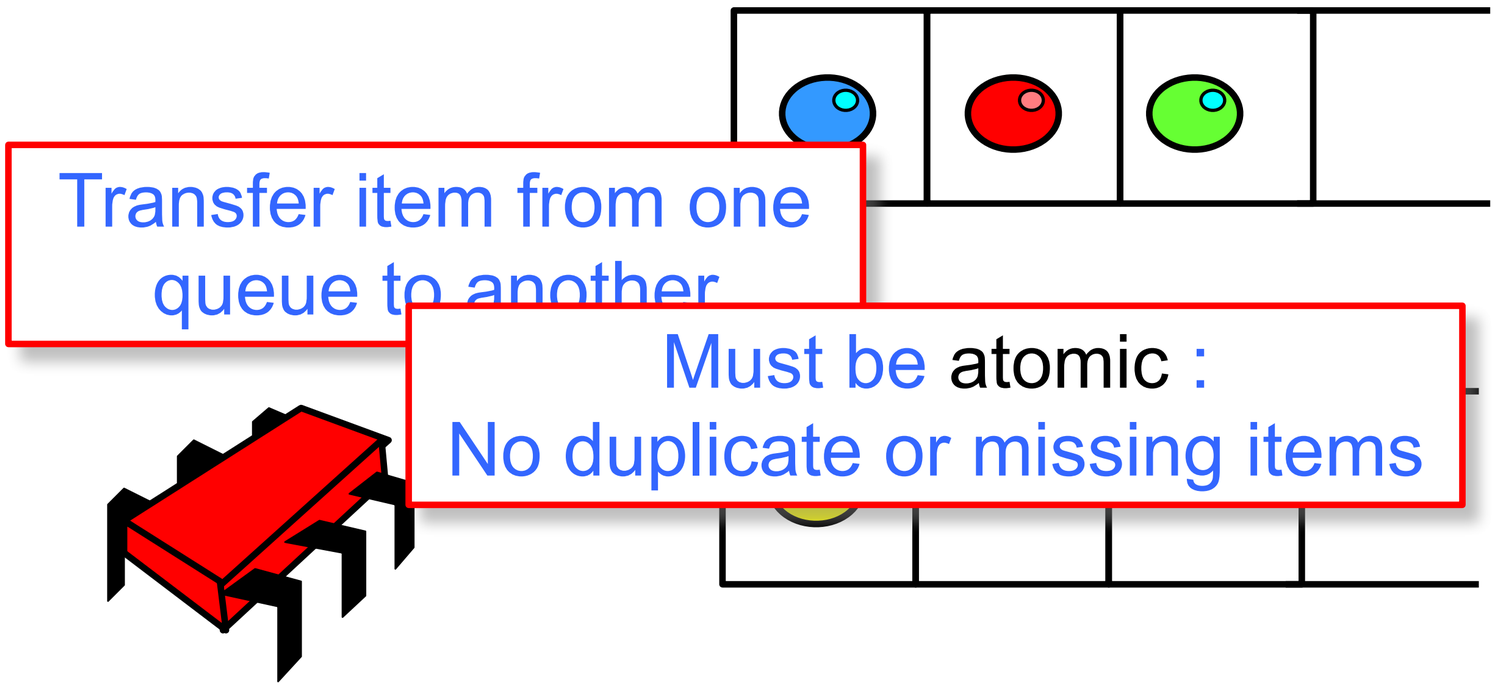
Actual comment from Linux Kernel
(hat tip: Bradley Kuszmaul)

```
/*
 * When a locked buffer is visible to the I/O layer
 * BH_Launder is set. This means before unlocking
 * we must clear BH_Launder,mb() on alpha and then
 * clear BH_Lock, so no reader can see BH_Launder set
 * on an unlocked buffer and then risk to deadlock.
 */
```

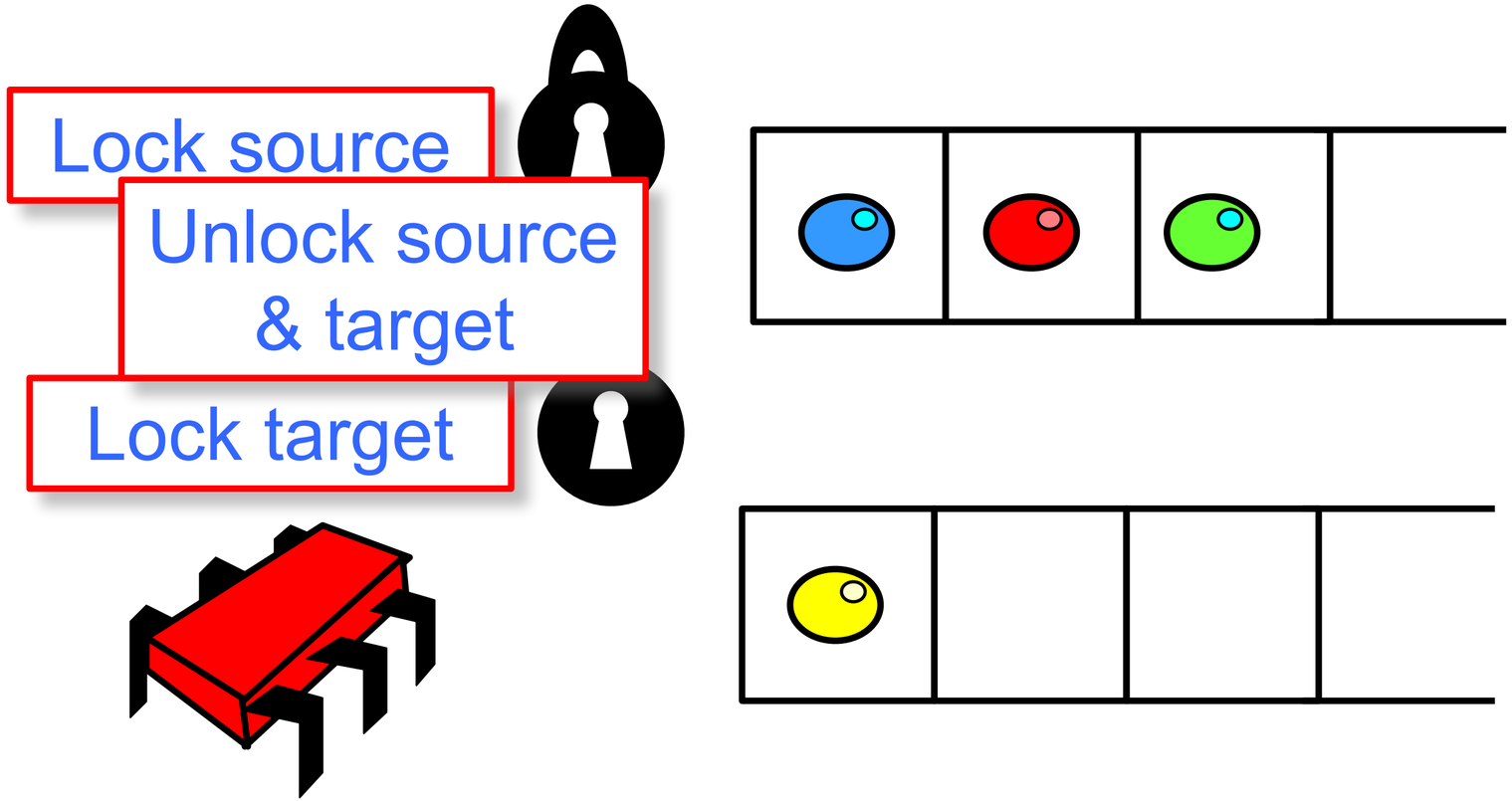
Simple Problems are hard



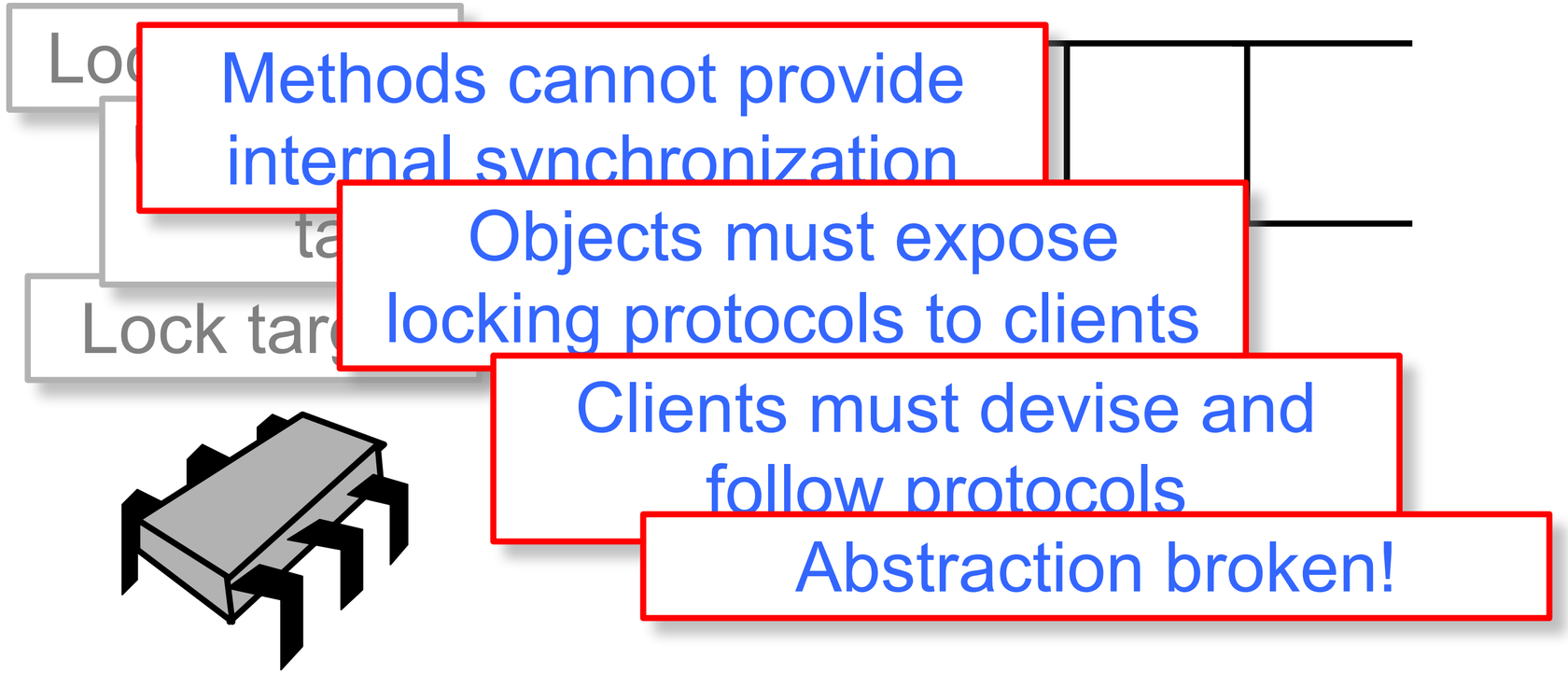
Locks Not Composable



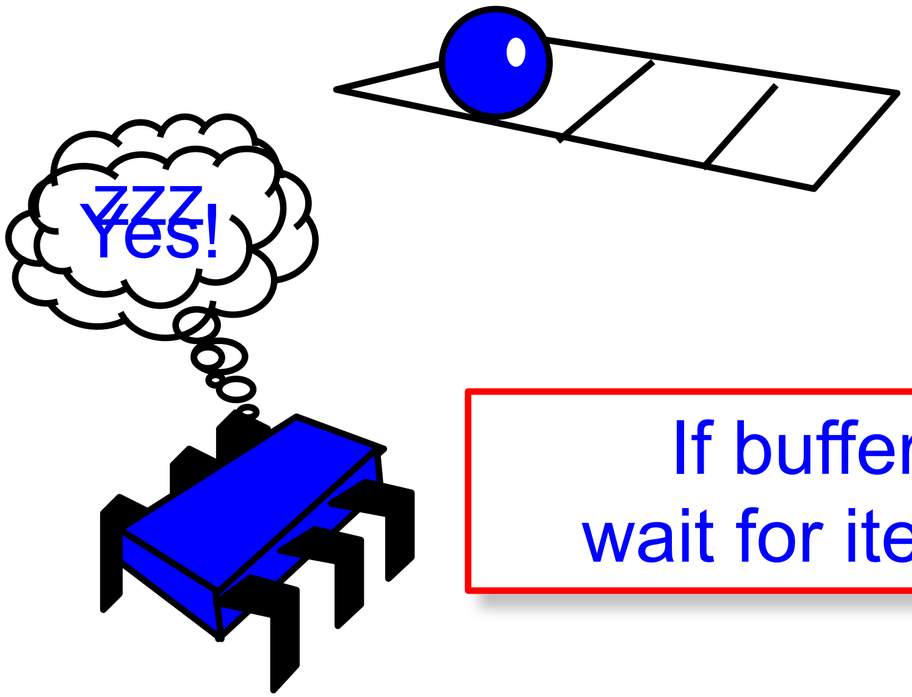
Locks Not Composable



Locks Not Composable



Monitor Wait and Signal

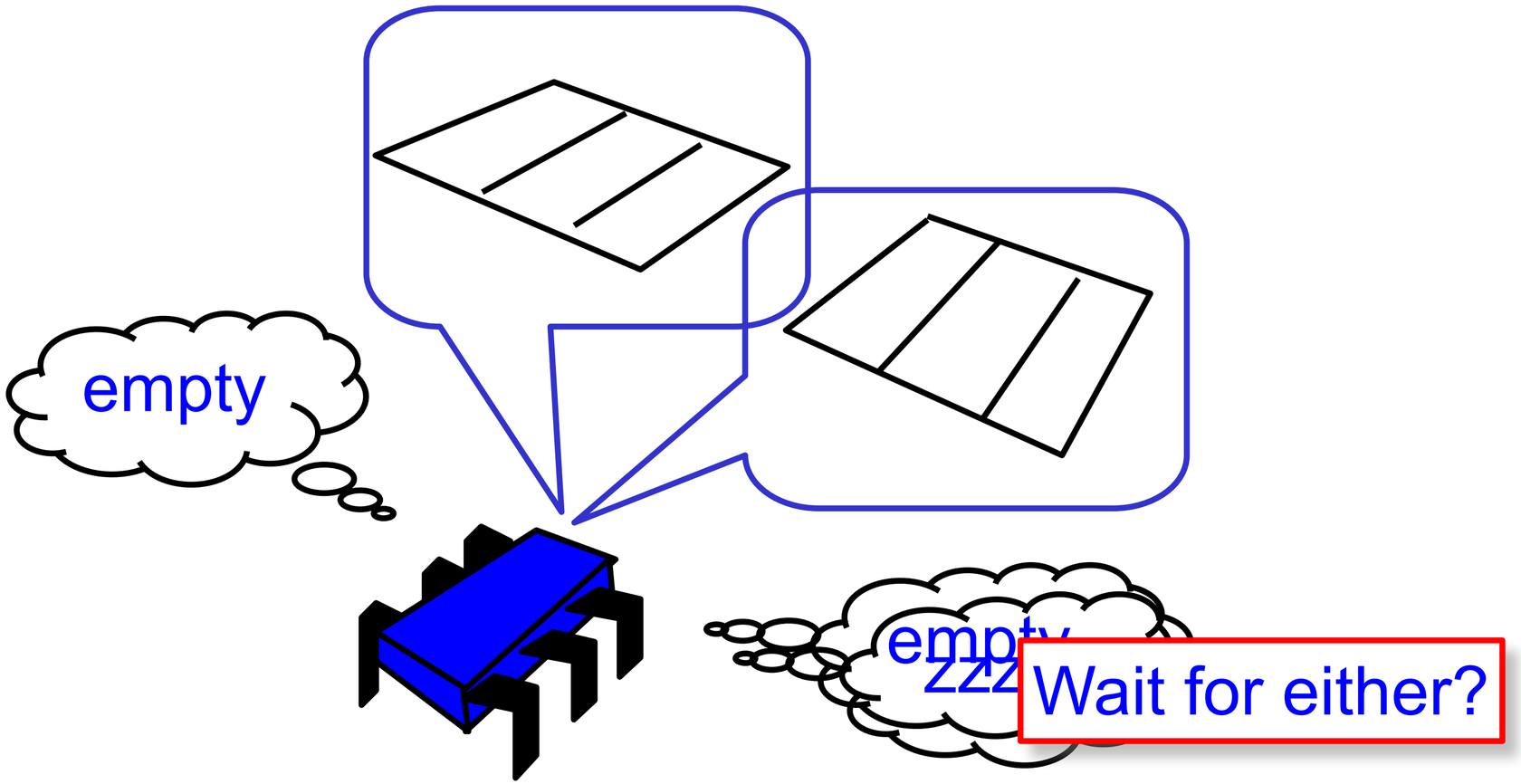


Empty

buffer

If buffer is empty,
wait for item to show up

Wait and Signal do not Compose



The Transactional Manifesto

- Current practice inadequate
  - to meet the multicore challenge
- Research Agenda
  - Replace **locking** with a **transactional** API
  - **Design** languages or libraries
  - **Implement** efficient run-times

Transactions

Block of code ….

Atomic: appears to happen instantaneously

Serializable: all appear to happen in one-at-a-time

Commit: takes effect (atomically)

Abort: has no effect (typically restarted)

Atomic Blocks

```
atomic {
 x.remove(3);
 y.add(3);
}

atomic {
 y = null;
}
```

Atomic Blocks

```
atomic {
 x.remove(3);
 y.add(3);
}

atomic {
 y = null;
}
```

No data race

A Double-Ended Queue

```
public void LeftEnq(item x) {
  Qnode q = new Qnode(x);
  q.left = this.left;
  this.left.right = q;
  this.left = q;
}
```

Write sequential Code

A Double-Ended Queue

```
public void LeftEnq(item x)
 atomic {
  Qnode q = new Qnode(x);
  q.left = this.left;
  this.left.right = q;
  this.left = q;
 }
}
```

A Double-Ended Queue

```
public void LeftEnq(item x) {
 atomic {
  Qnode q = new Qnode(x);
  q.left = this.left;
  this.left.right = q;
  this.left = q;
 }
}
```

Enclose in atomic block

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 global lock for shared memory

- Fine grain locking
  - high concurrency, lower chance of correctness

- 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 <u>what</u> to do (maintain atomicity of this code), not <u>how</u> to do it
  - No explicit use or management of locks

- System implements synchronization as necessary to ensure atomicity
  - System <u>could</u> implement atomic { } using locks (see this later)
  - Implementation discussed today uses optimistic concurrency: maintain serialization only in situations of true contention (R-W or WAW 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 Nworker threads. Assign work to threads by removing work from a shared task queue
  - Acquire a lock, perform operations, release the lock

Transactional Memory (TM)

- 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: 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

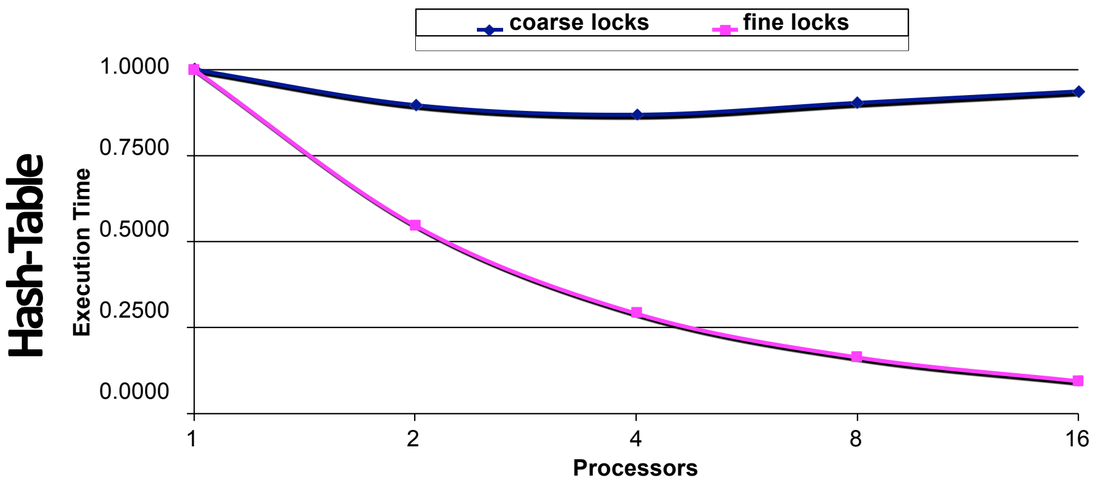
What are solutions for making Java's HashMap thread-safe?

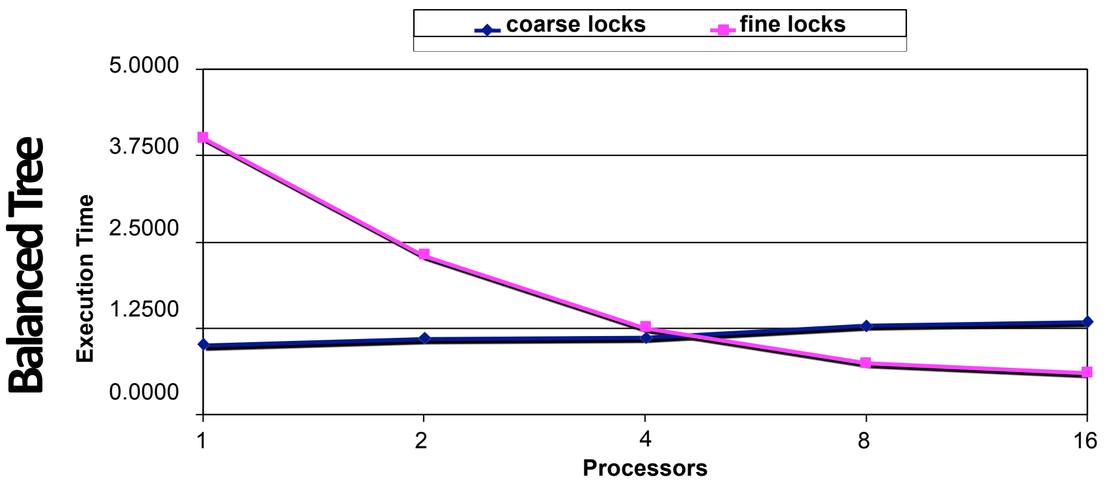
```
public Object get(Object key)  {
  int idx = hash(key);                  // compute hash
  HashEntry e = buckets[idx];           // find bucket
  while (e != null)  {                  // find element in bucket
    if (equals(key, e.key))
      return e.value;
    e = e.next;
  }
  return null;
}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

- 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





Transactional HashMap

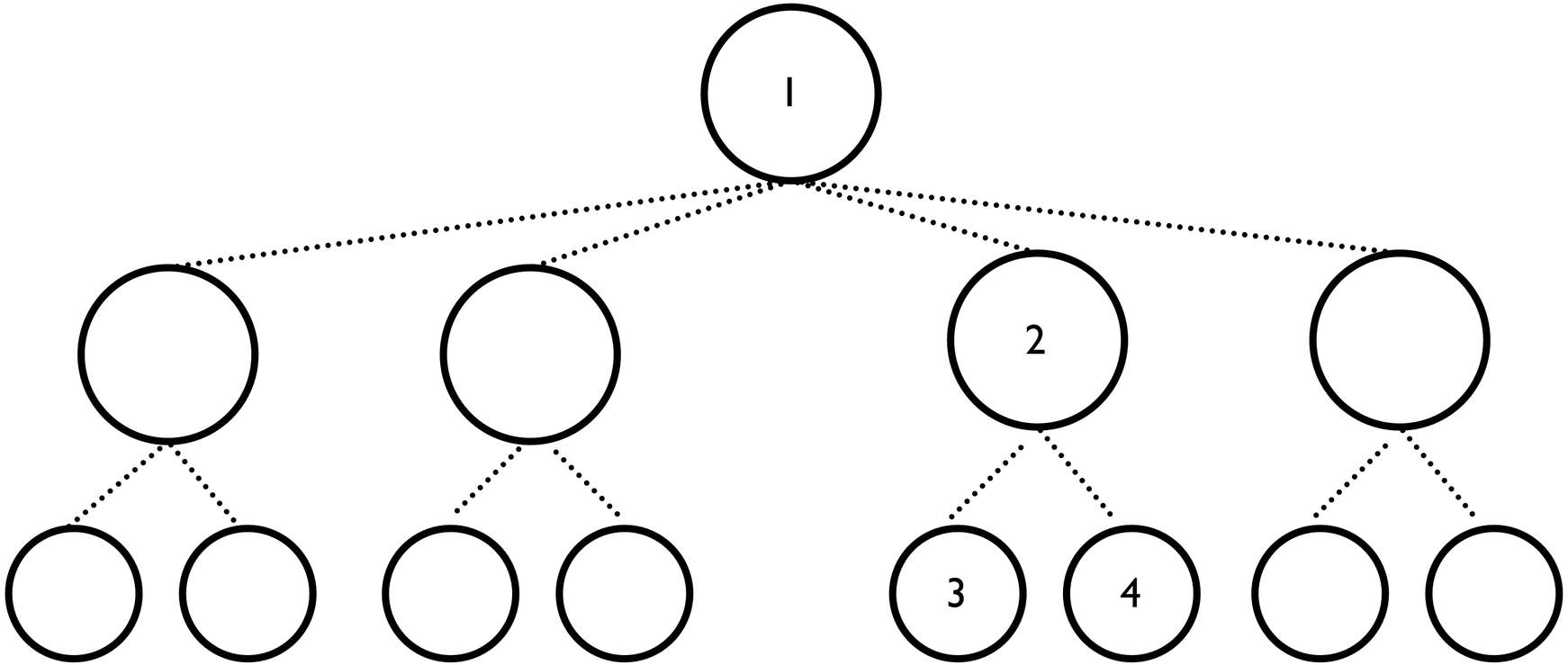
- Simply enclose all operation in atomic block
  - Semantics of atomic block: system ensures atomicity of logic within block

```
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

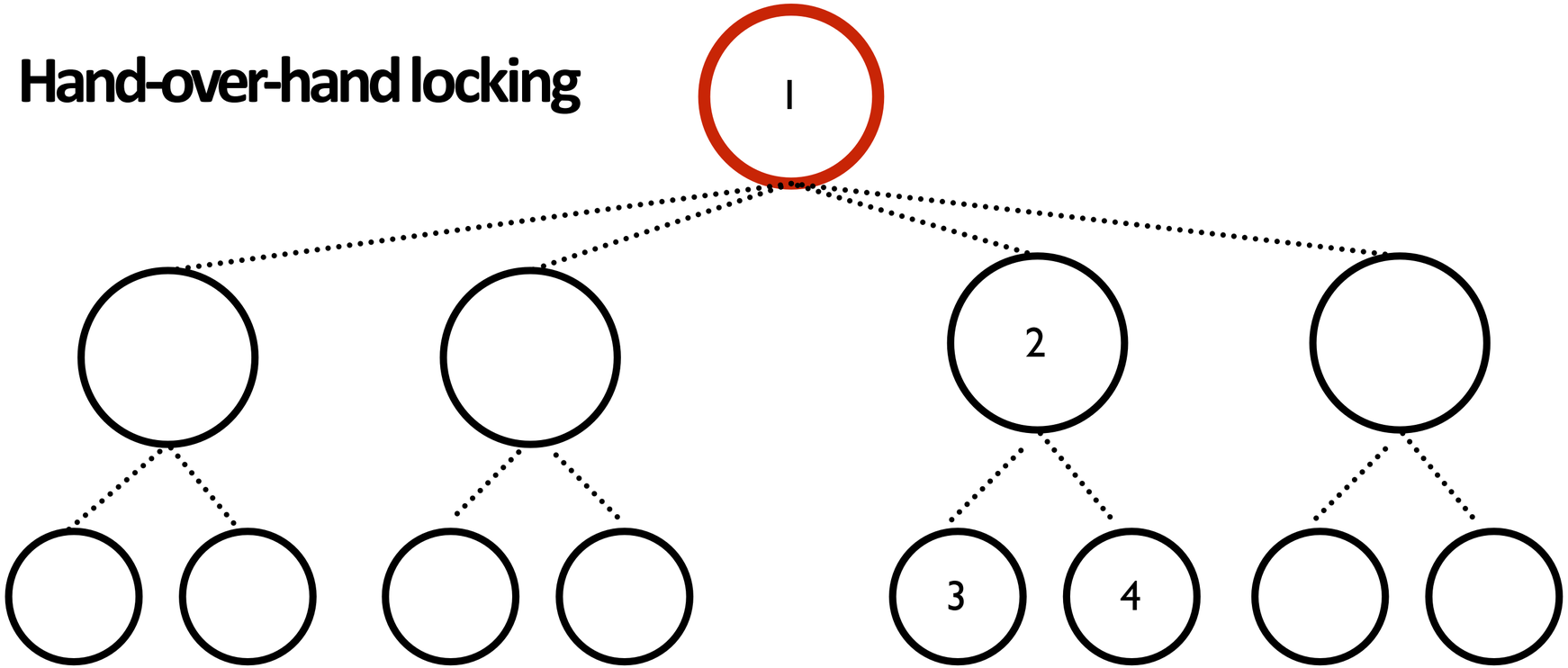
Goal: modify nodes 3 and 4 in a thread-safe way



Fine-grained locking example

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

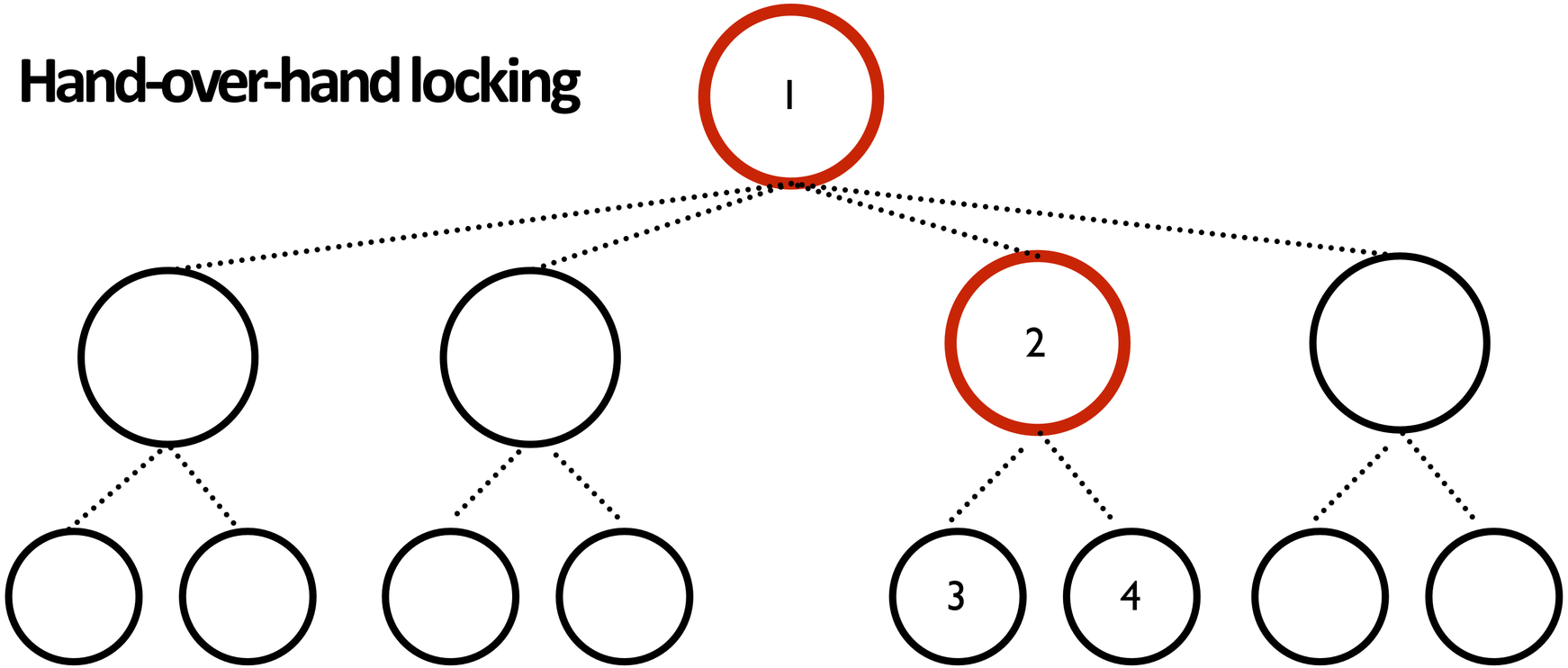
Hand-over-hand locking



Fine-grained locking example

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

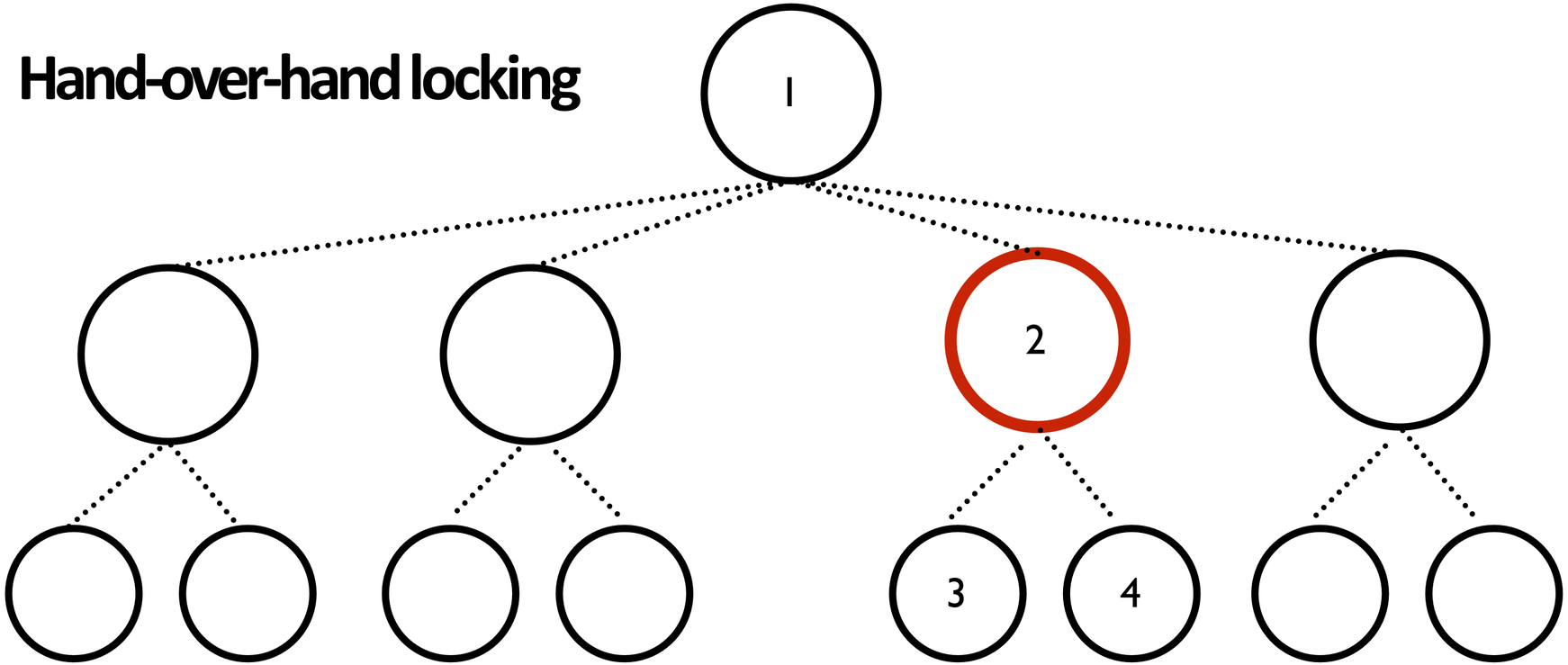
Hand-over-hand locking



Fine-grained locking example

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

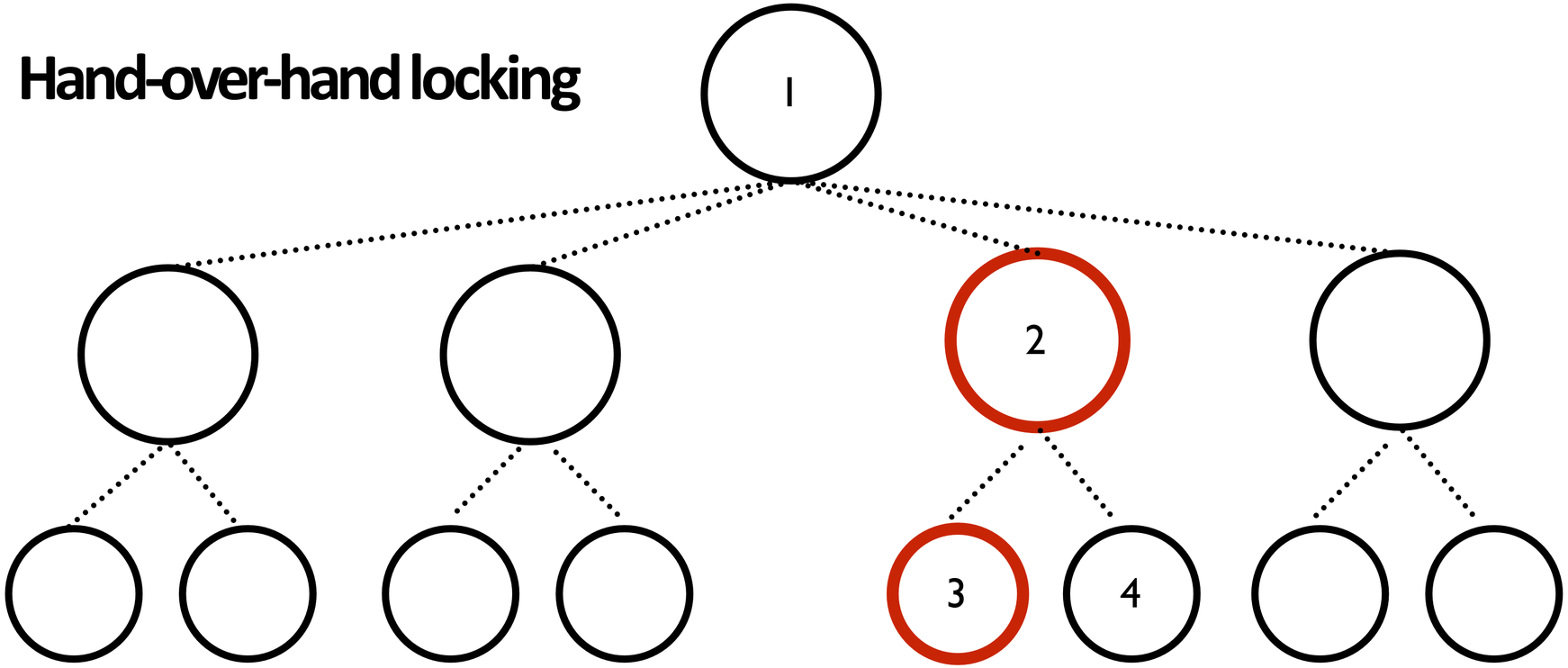
Hand-over-hand locking



Fine-grained locking example

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

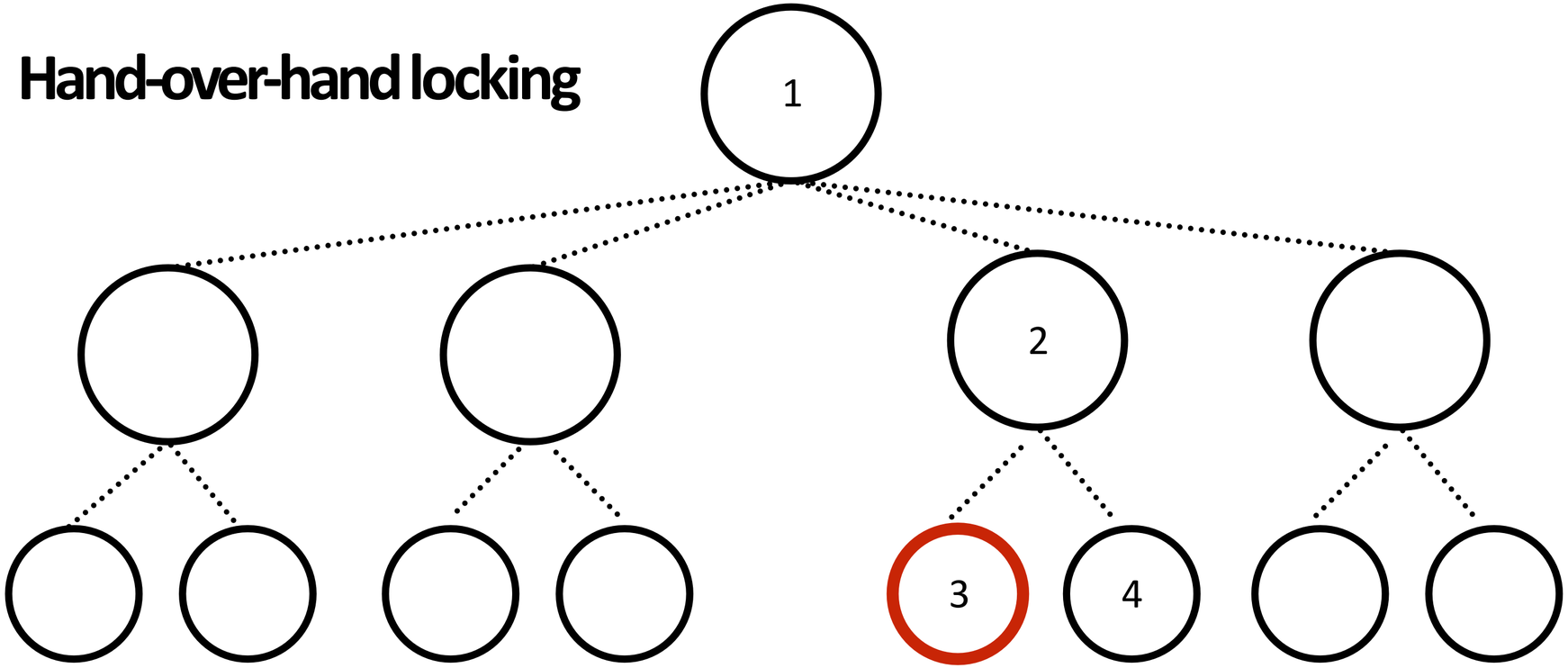
Hand-over-hand locking



Fine-grained locking example

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

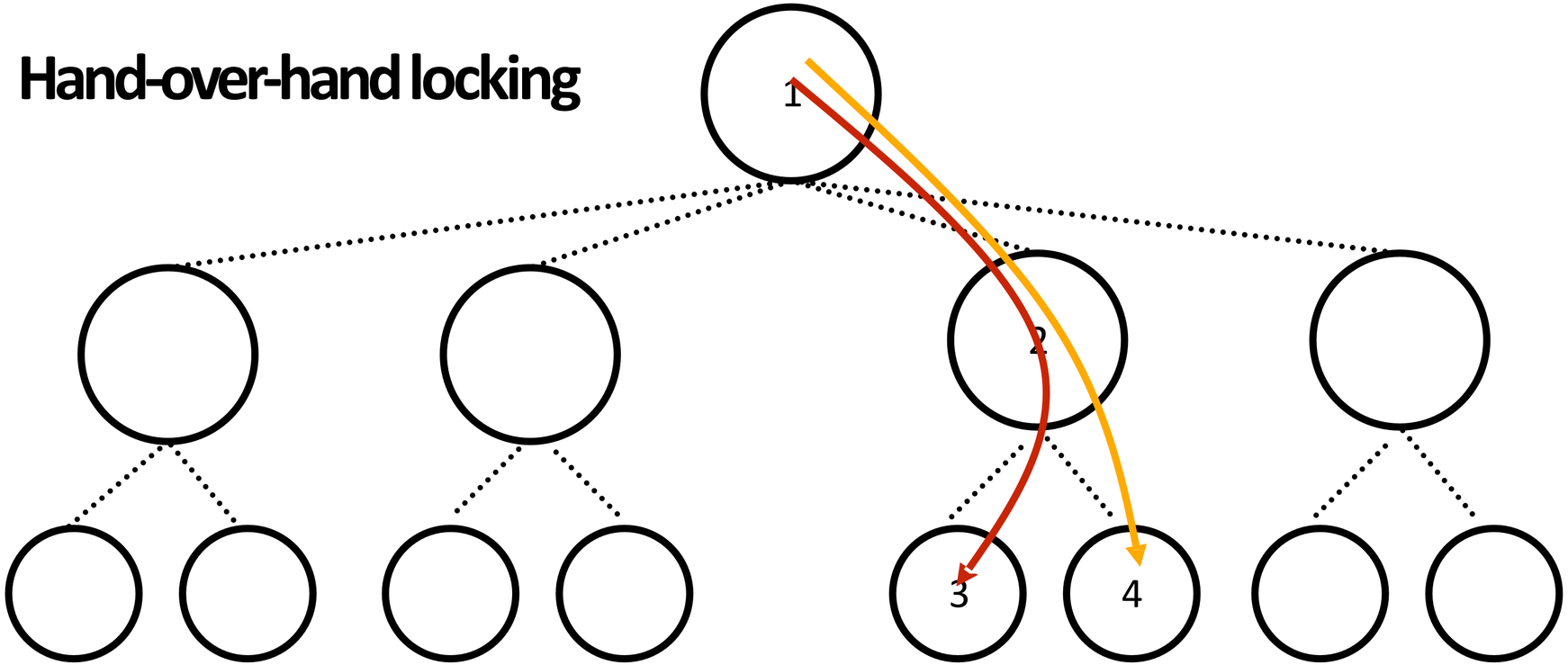
Hand-over-hand locking



Fine-grained locking example

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

Hand-over-hand locking

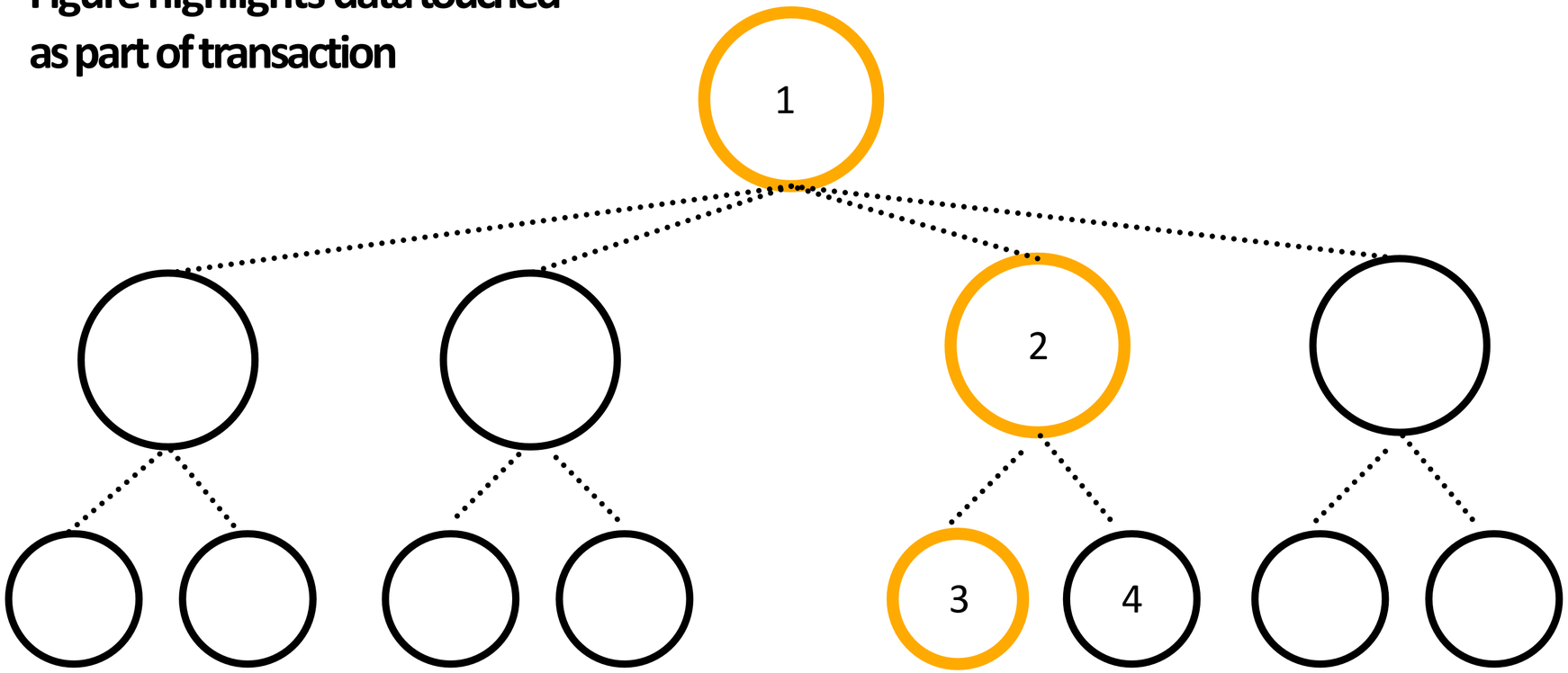


Locking can prevent concurrency

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

Transactions example

Figure highlights data touched as part of transaction

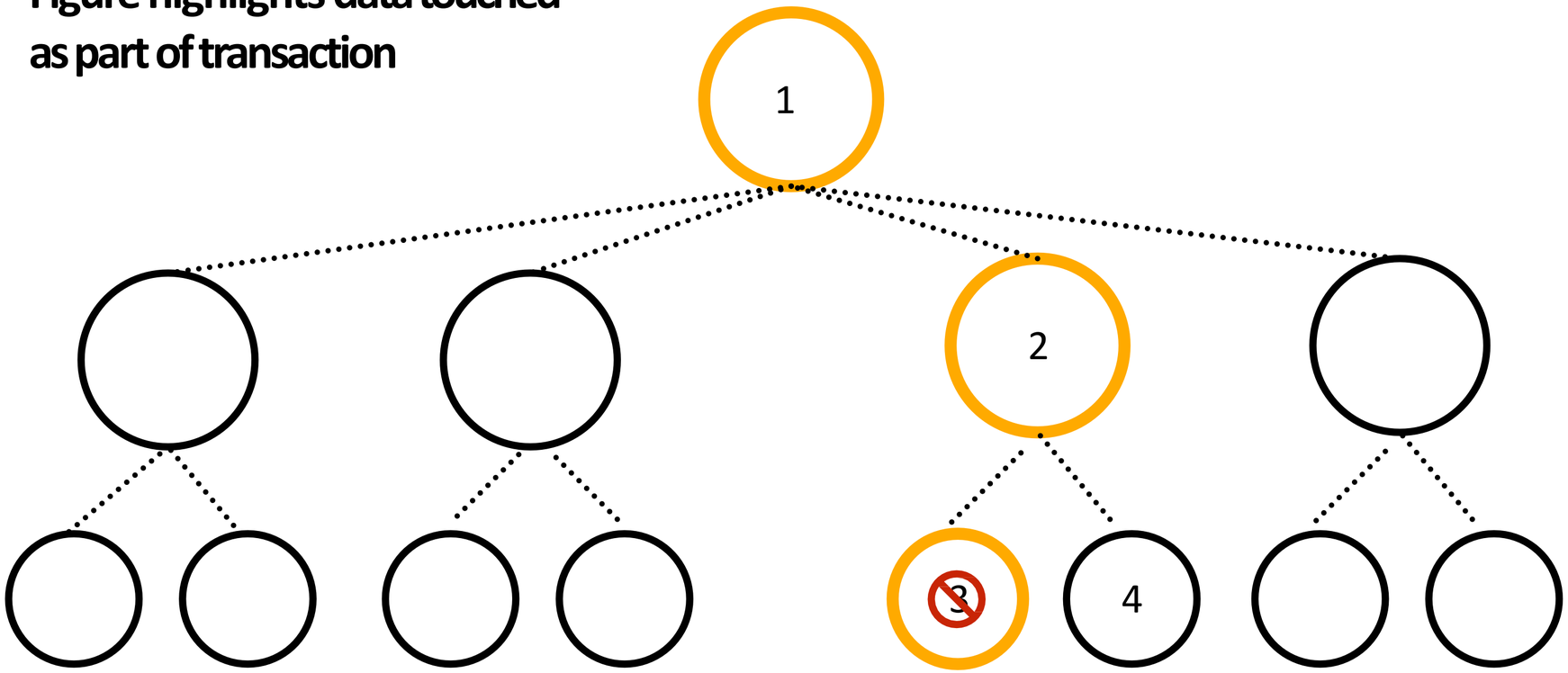


Transaction A

READ: 1, 2, 3

Transactions example

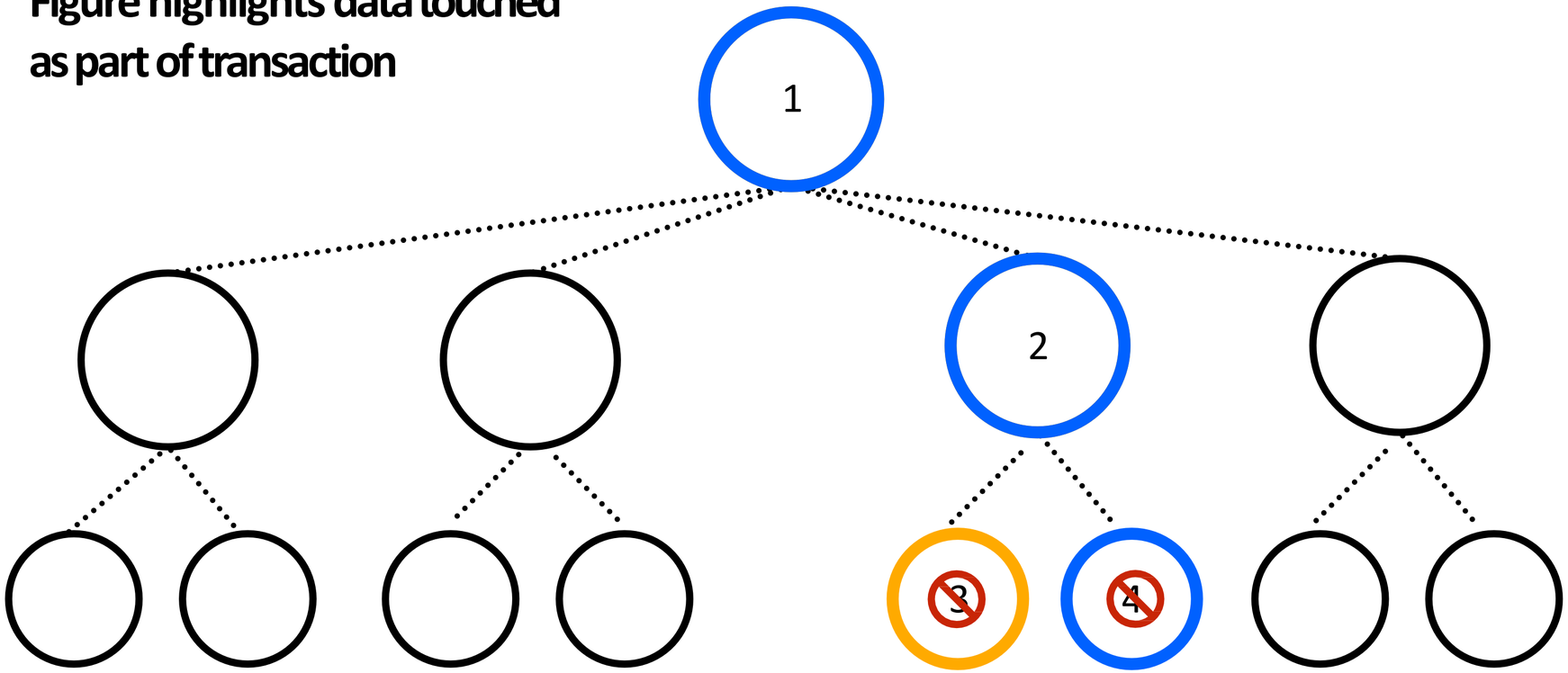
Figure highlights data touched as part of transaction



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

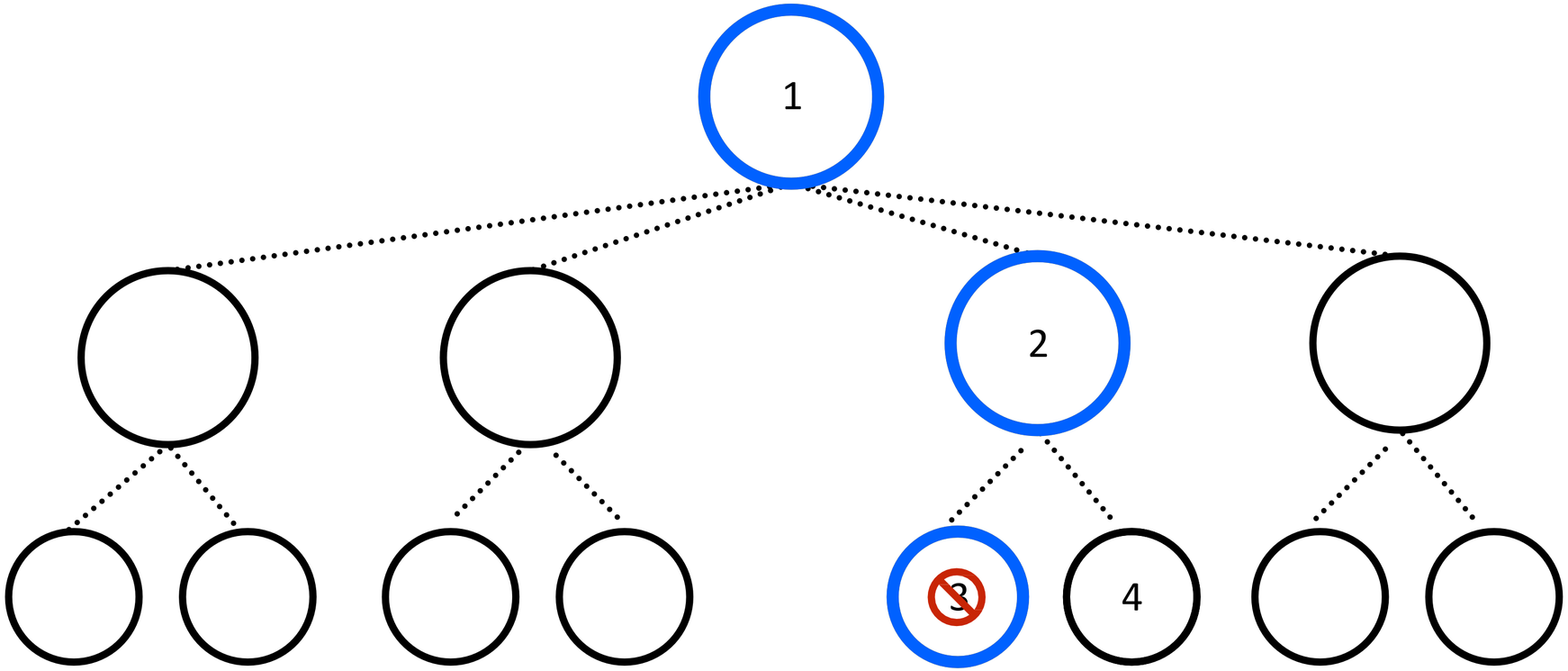
Transaction B
READ: 1, 2,4
WRITE: 4

NO READ-WRITE or WRITE-WRITE conflicts!

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

Transactions example #2

(Both transactions modify node 3)



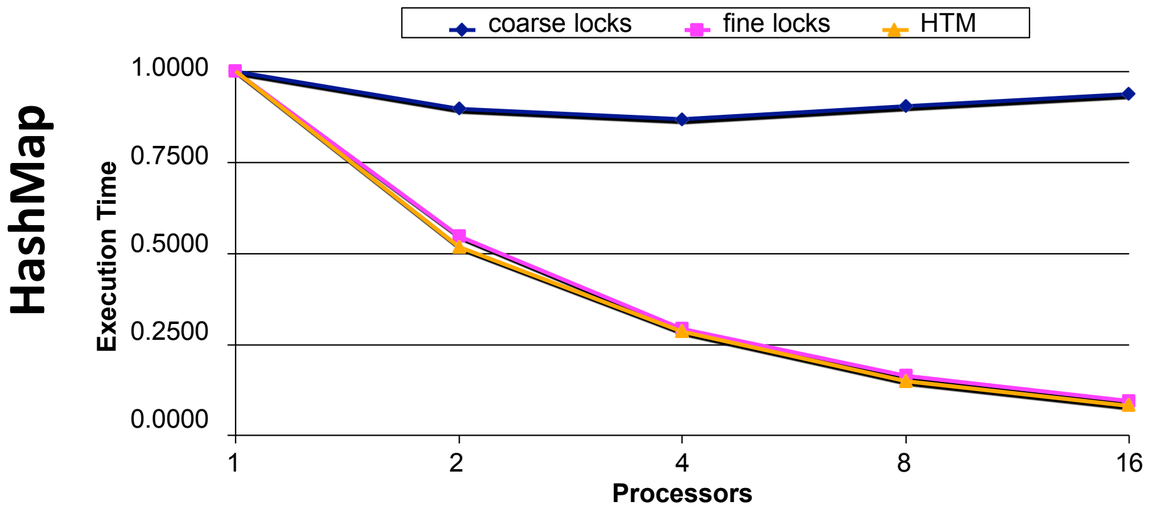
Transaction A
READ: 1, 2, 3
WRITE: 3

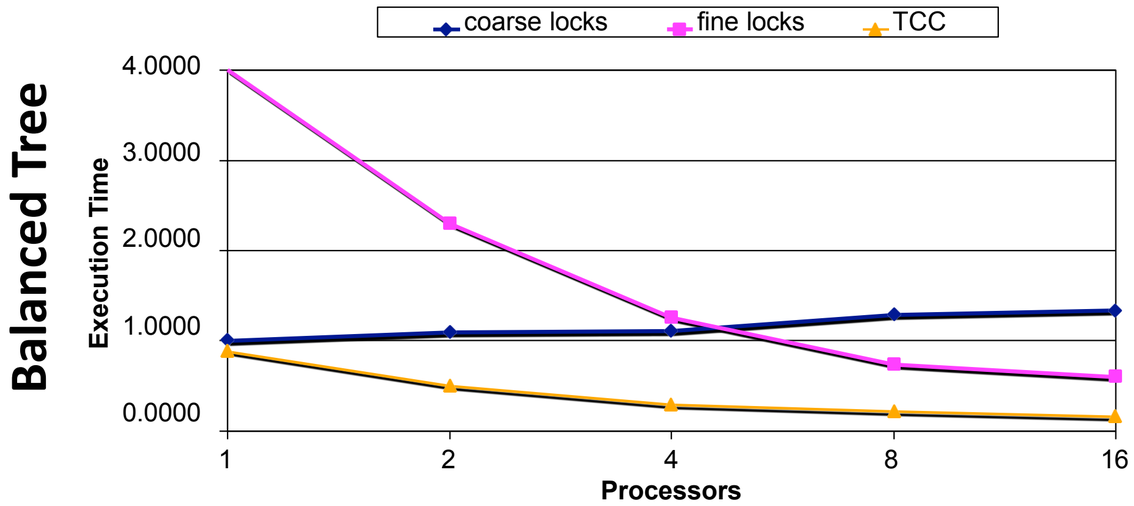
Transaction B
READ: 1, 2,3
WRITE: 3

Conflicts exist: transactions must be serialized

(both transactions write to node 3)

Performance: locks vs. transactions





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(x, y, 100);
```

Thread 1:

```
transfer(y, x, 100);
```

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(A, B, 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 an 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(x, y, 100)`

Thread 1:
`transfer(y, x, 100);`

- **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**
- **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

Make sure you understand this difference in semantics!

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 concurrency as much 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?**

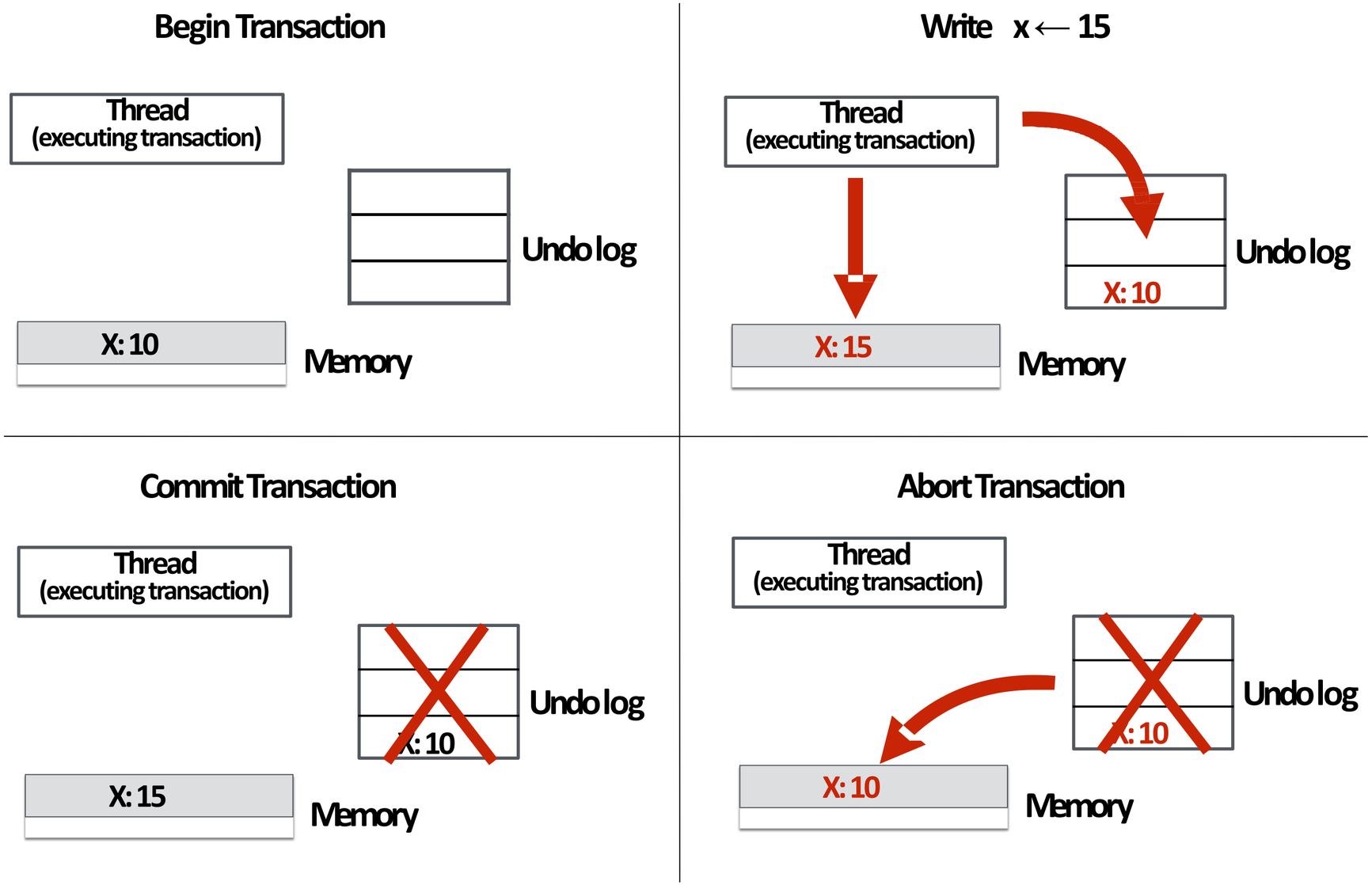
Data versioning policy

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)

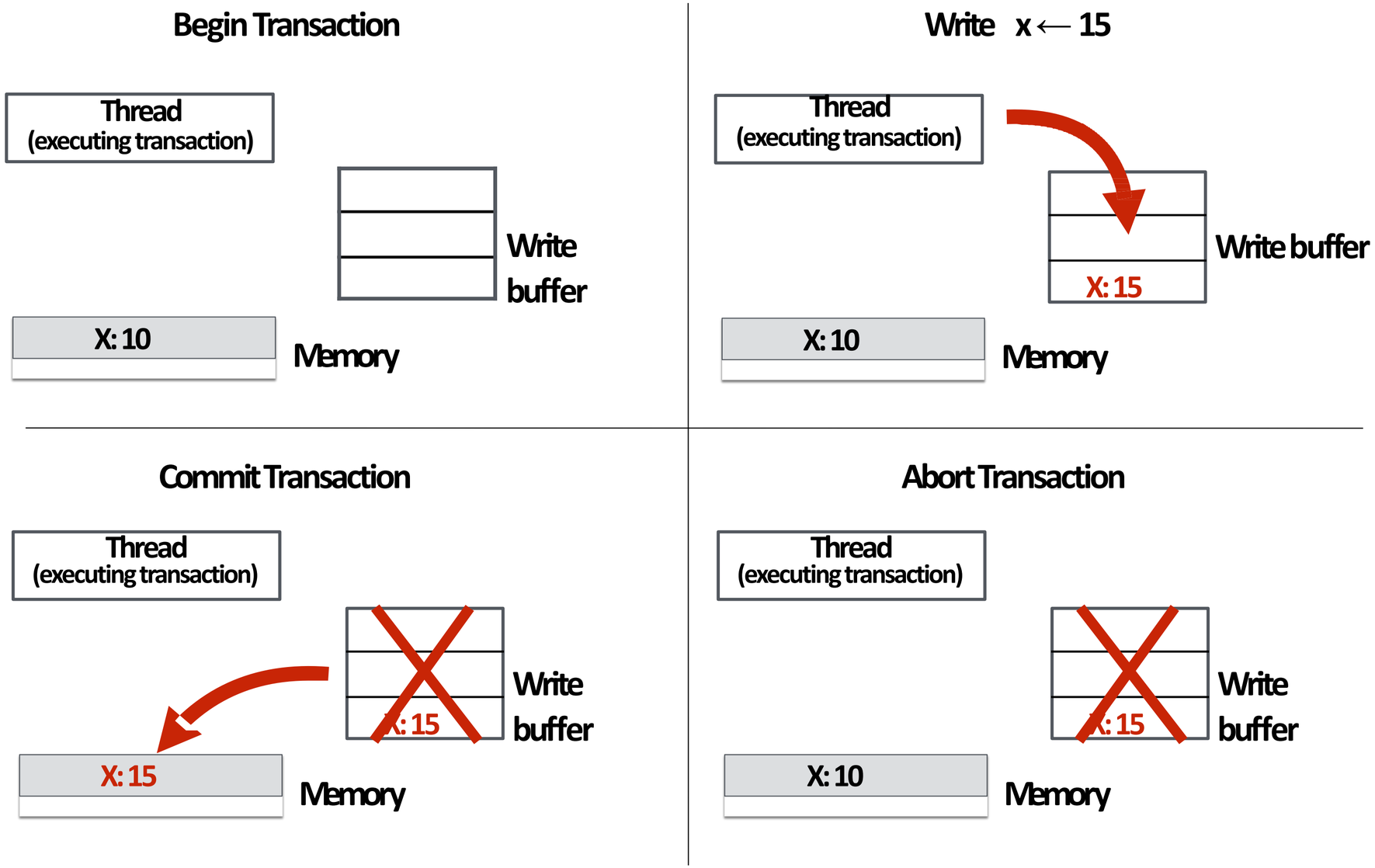
Eager versioning

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

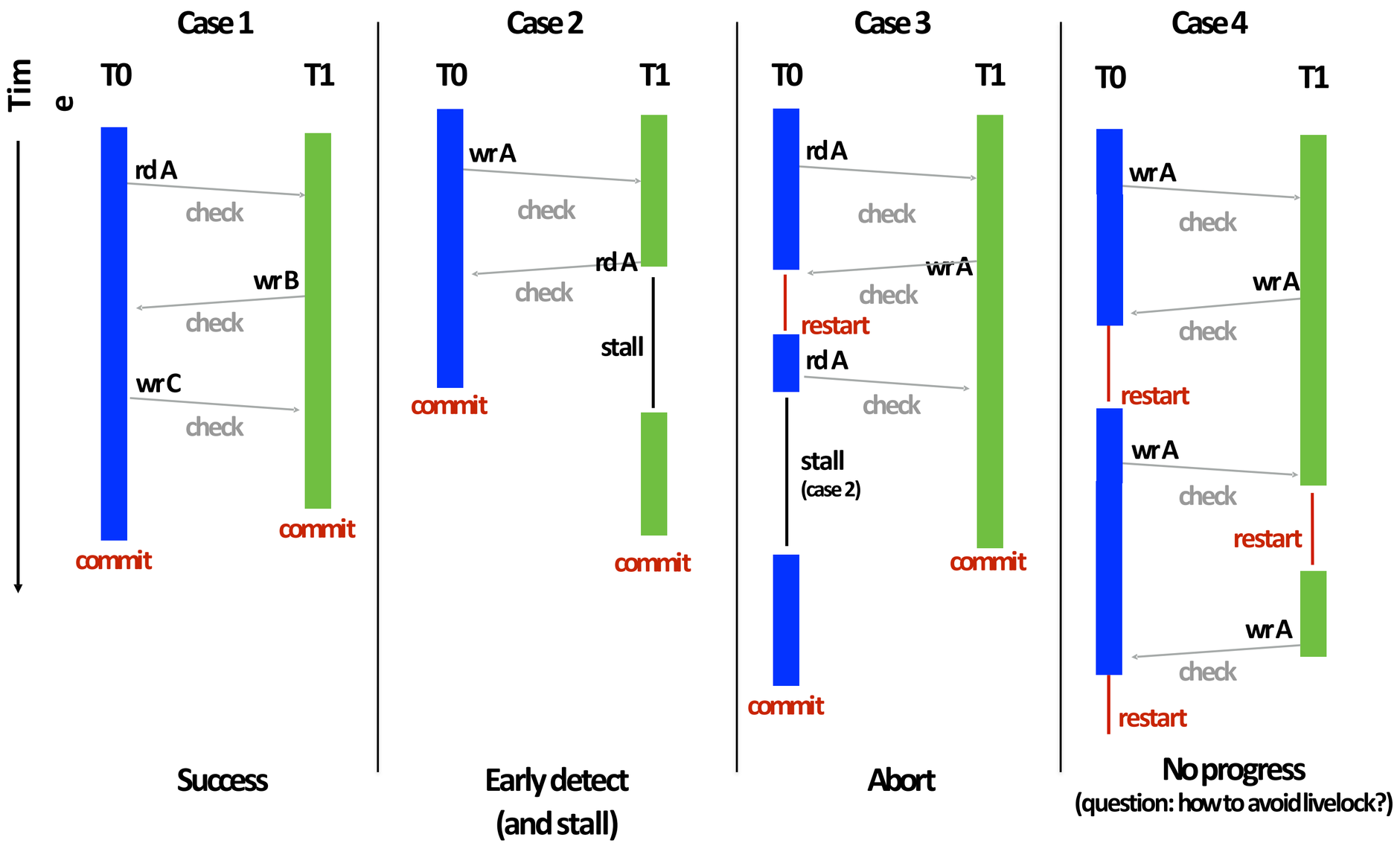
- System must track a transaction's read set and write set
  - Read-set: addresses read during the transaction
  - Write-set: addresses written during the transaction

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

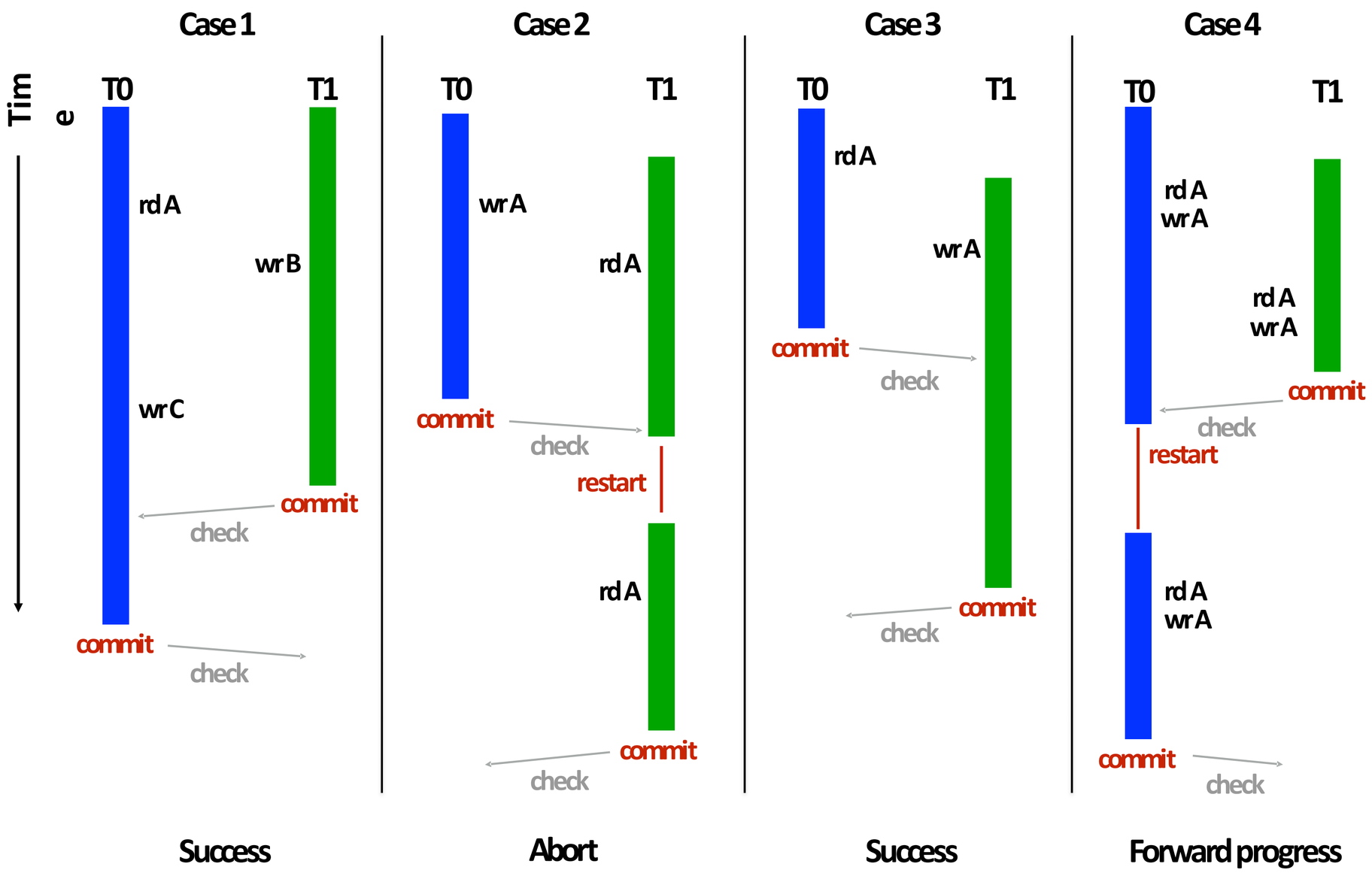
Note: diagrams assume "aggressive" contention manager on writes: writer wins, so other transactions abort)



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

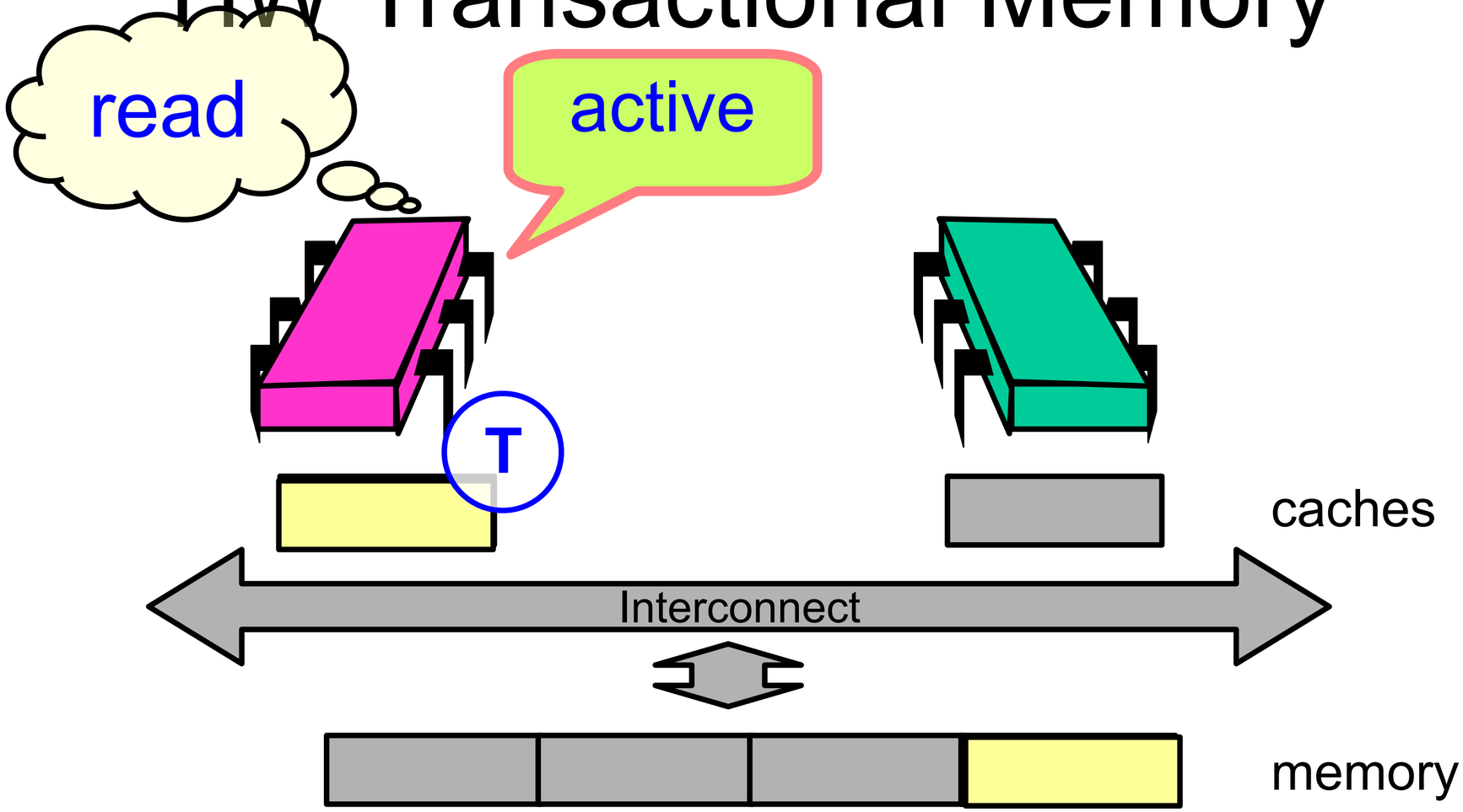
- **Pessimistic conflict detection (a.k.a. "eager")**
  - Good: detect conflicts early (undo less work, turn some aborts to stalls)
  - Bad: no forward progress guarantees, more aborts in some cases
  - Bad: fine-grained communication (check on each load/store)
  - Bad: detection on critical path

- **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

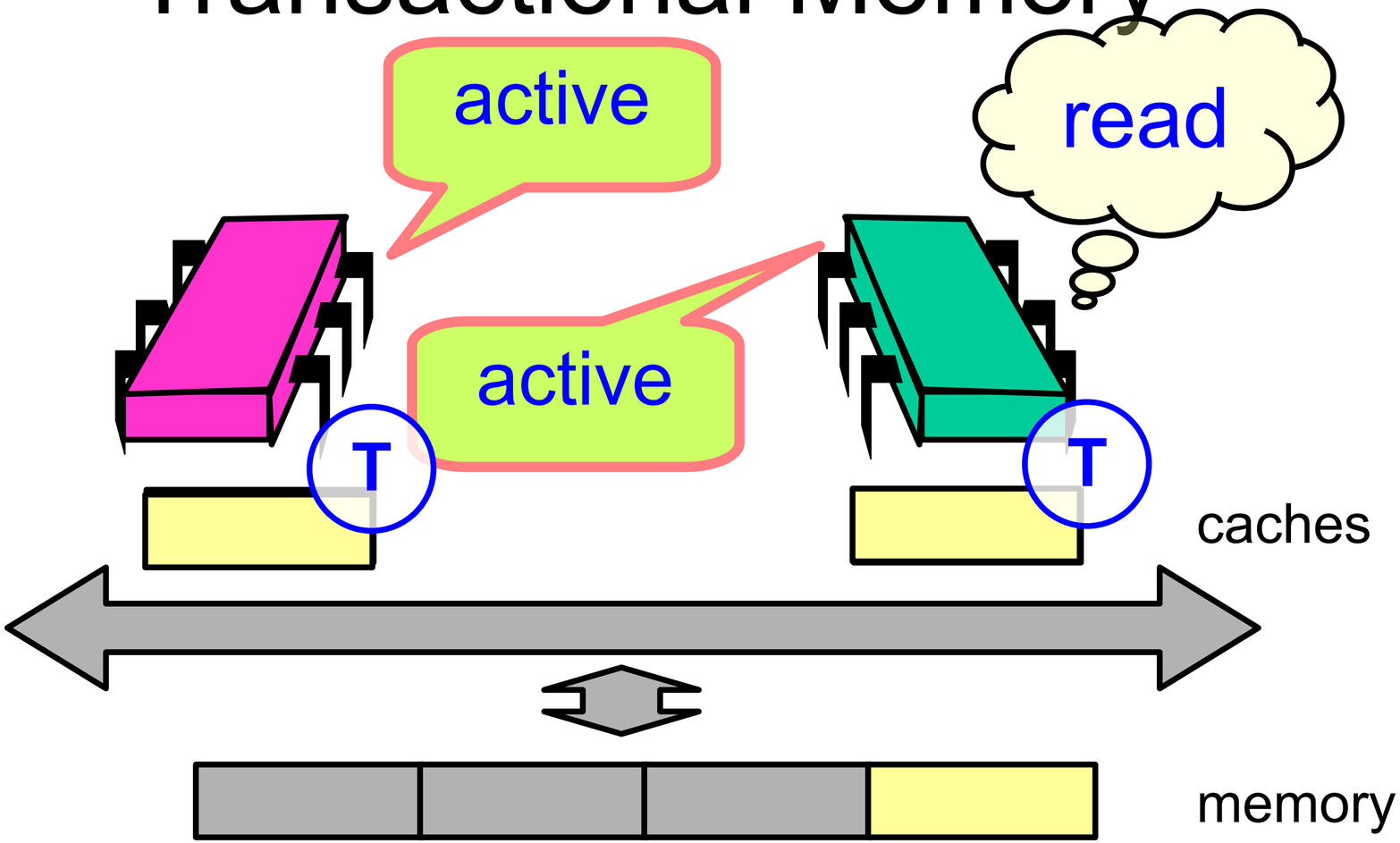
Hardware Transactional Memory

- Exploit Cache coherence
- Already almost does it
  - Invalidation
  - Consistency checking
- Speculative execution
  - Branch prediction = optimistic synch!

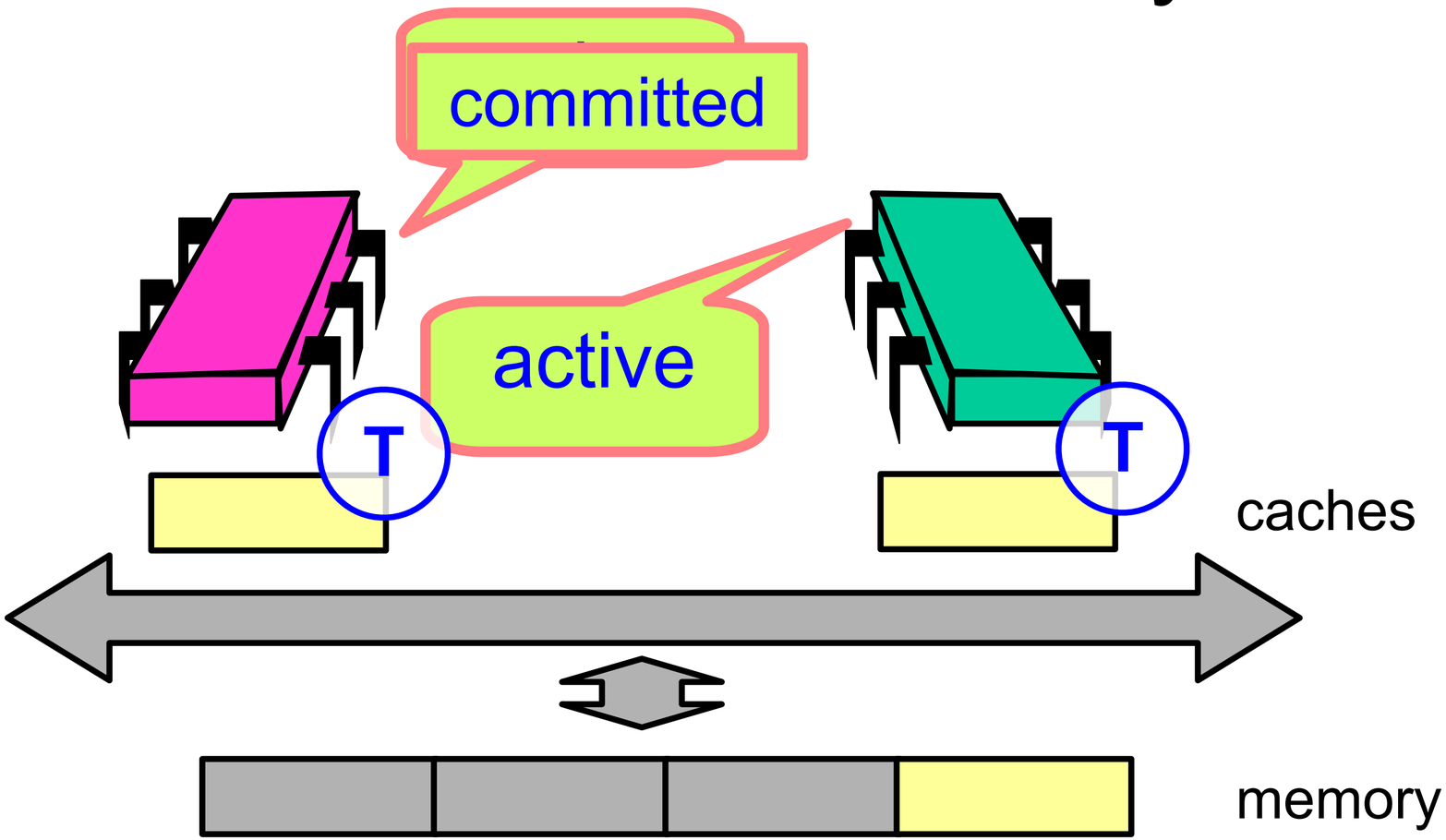
HW Transactional Memory



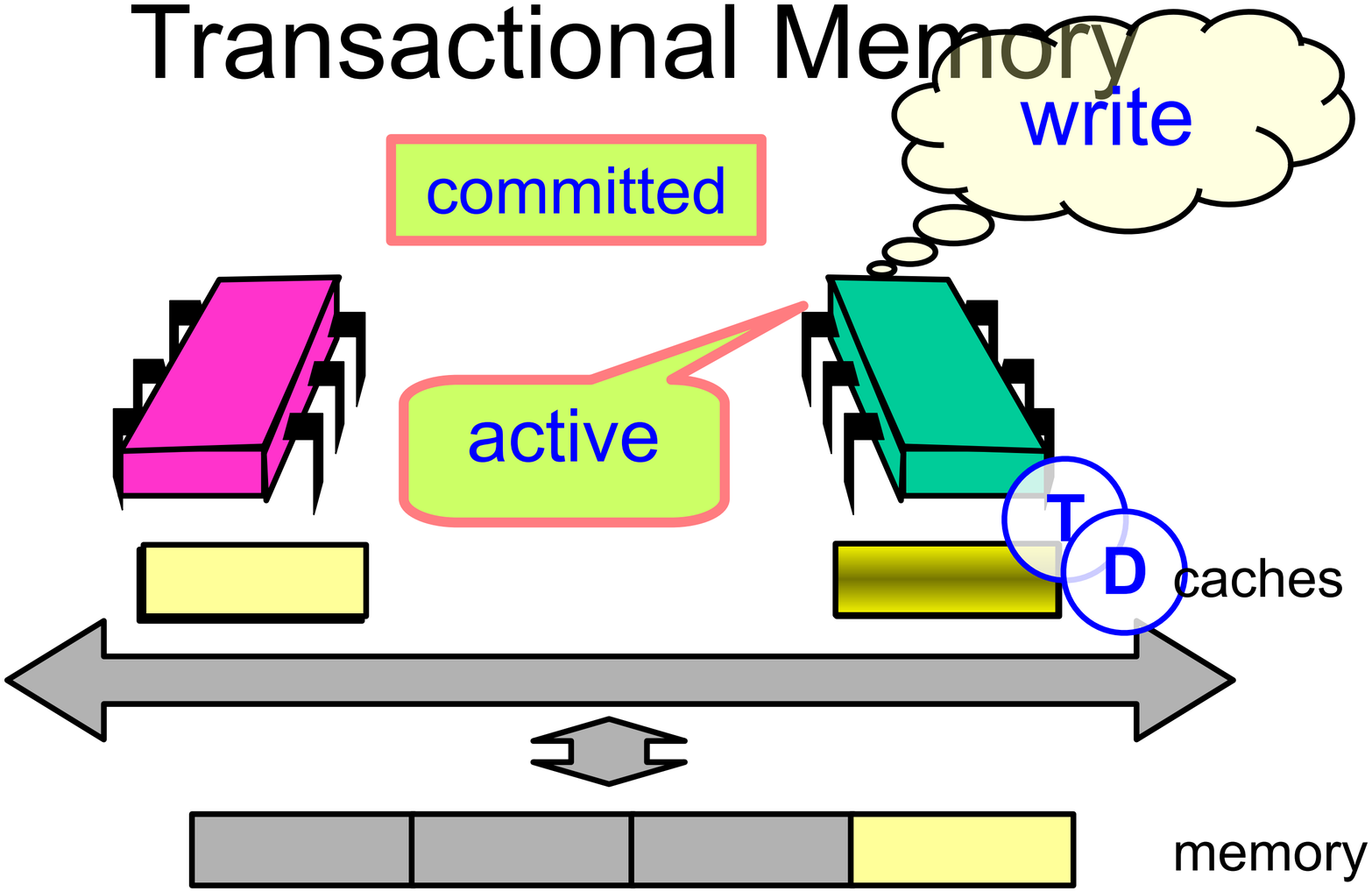
Transactional Memory



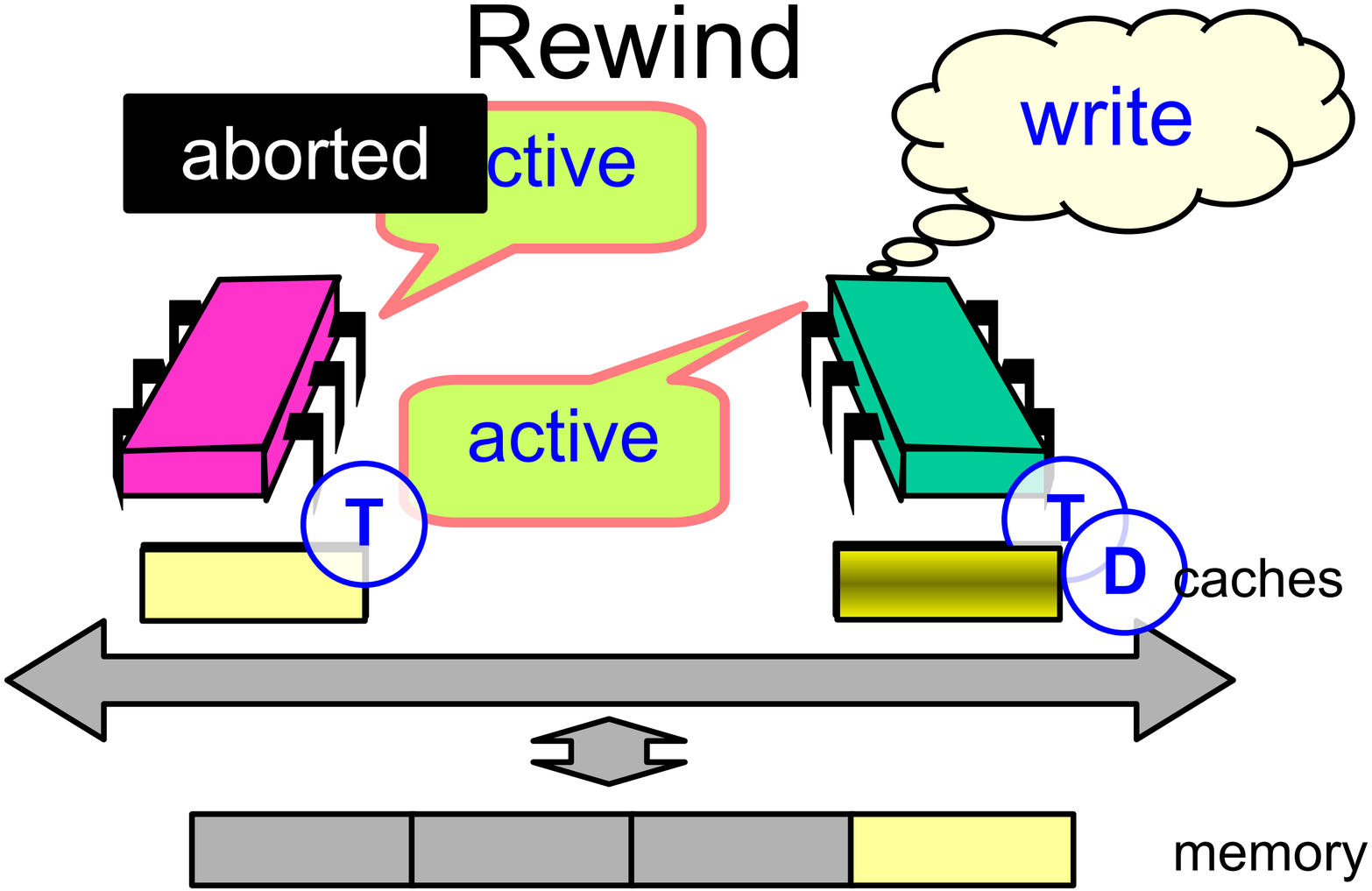
Transactional Memory



Transactional Memory



Rewind



Transaction Commit

- At commit point
  - If no cache conflicts, we win.
- Mark transactional entries
  - Read-only: valid
  - Modified: dirty (eventually written back)
- That's all, folks!
  - Except for a few details …

Not all and

- Limits to
  - Transactional cache size
  - Scheduling quantum
- Transaction cannot commit if it is
  - Too big
  - Too slow
  - Actual limits platform-dependent

HTM Strengths & Weaknesses

- Ideal for lock-free data structures

HTM Strengths & Weaknesses

- Ideal for lock-free data structures
- Practical proposals have limits on
  - Transaction size and length
  - Bounded HW resources
  - Guarantees vs best-effort

HTM Strengths & Weaknesses

- Ideal for lock-free data structures
- Practical proposals have limits on
  - Transaction size and length
  - Bounded HW resources
  - Guarantees vs best-effort
- On fail
  - Diagnostics essential
  - Retry in software?

Compos

Locks don’t compose, transactions do.

Composition necessary for Software Engineering.

But practical HTM doesn’t really support composition!

Why we need STM

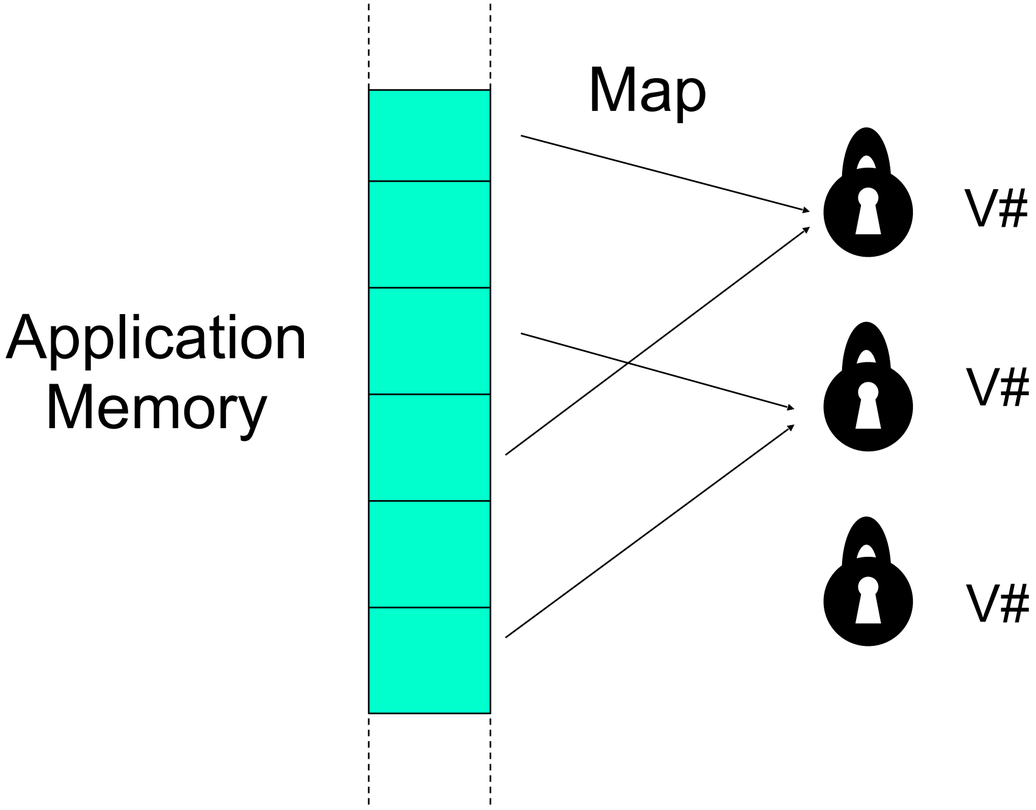
Simple Lock-Based STM

- STMs come in different forms
  - Lock-based
  - Lock-free
- Here : a simple lock-based STM

Synchronization

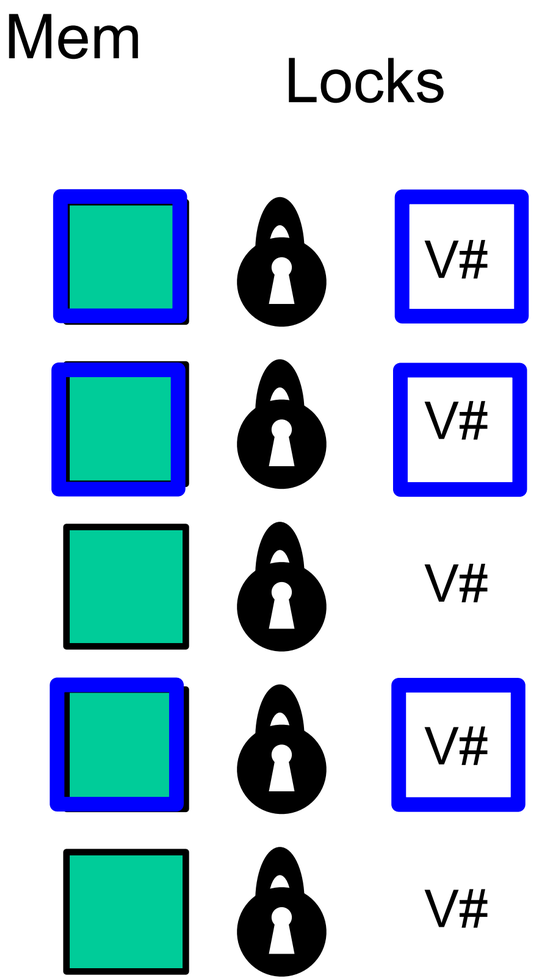
- Transaction keeps
  - **Read set**: locations & values read
  - **Write set**: locations & values to be written
- Deferred update
  - Changes installed at commit
- Lazy conflict detection
  - Conflicts detected at commit

STM: Transactional Locking



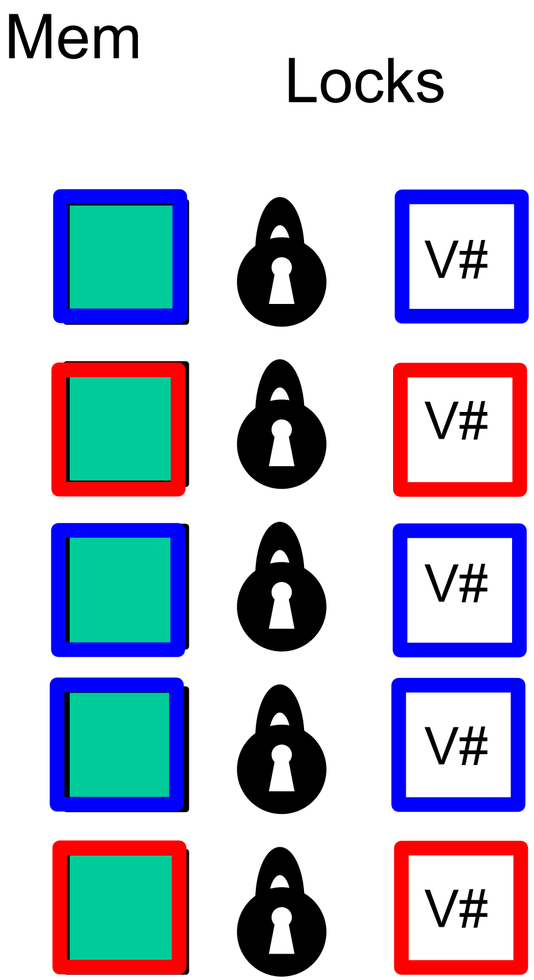
Array of version #s & locks

Reading an Object



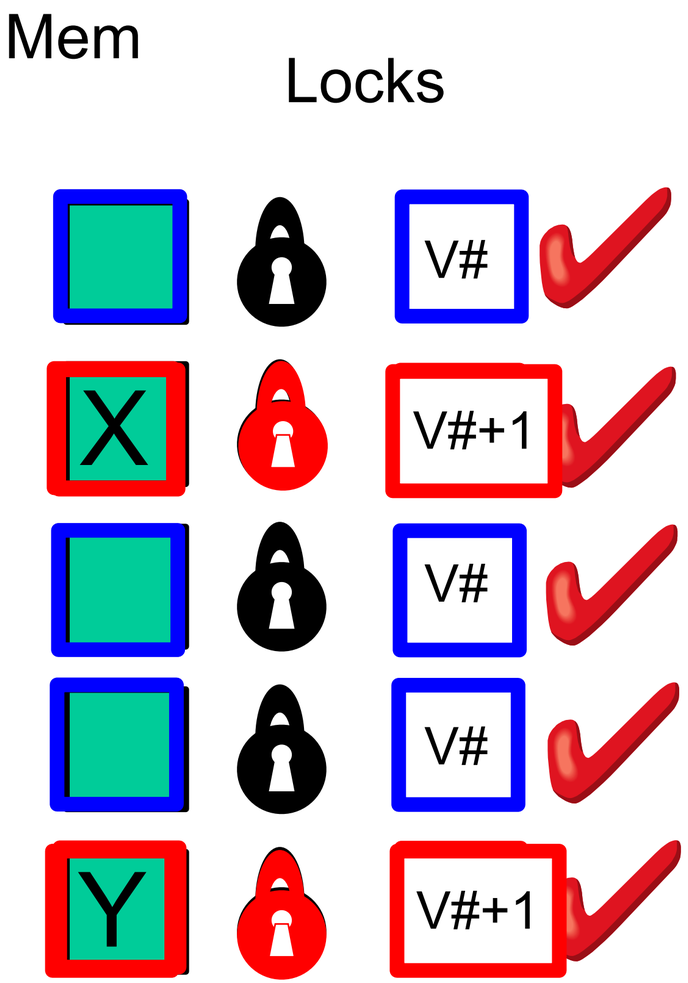
Add version numbers & values to read set

To Write an Object



Add version numbers & new values to write set

To Commit



Acquire write locks

Check version numbers unchanged

Install new values

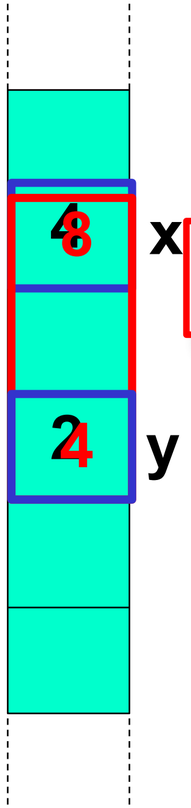
Increment version numbers

Unlock.

blem: Internal Inconsiste

- A Zombie is an active transaction destined to abort.
- If Zombies see inconsistent states bad things can happen

Internal Consistency



Invariant: x = 2y

Transaction A: reads x = 4

Transaction B: writes
8 to x, 16 to y, aborts A)

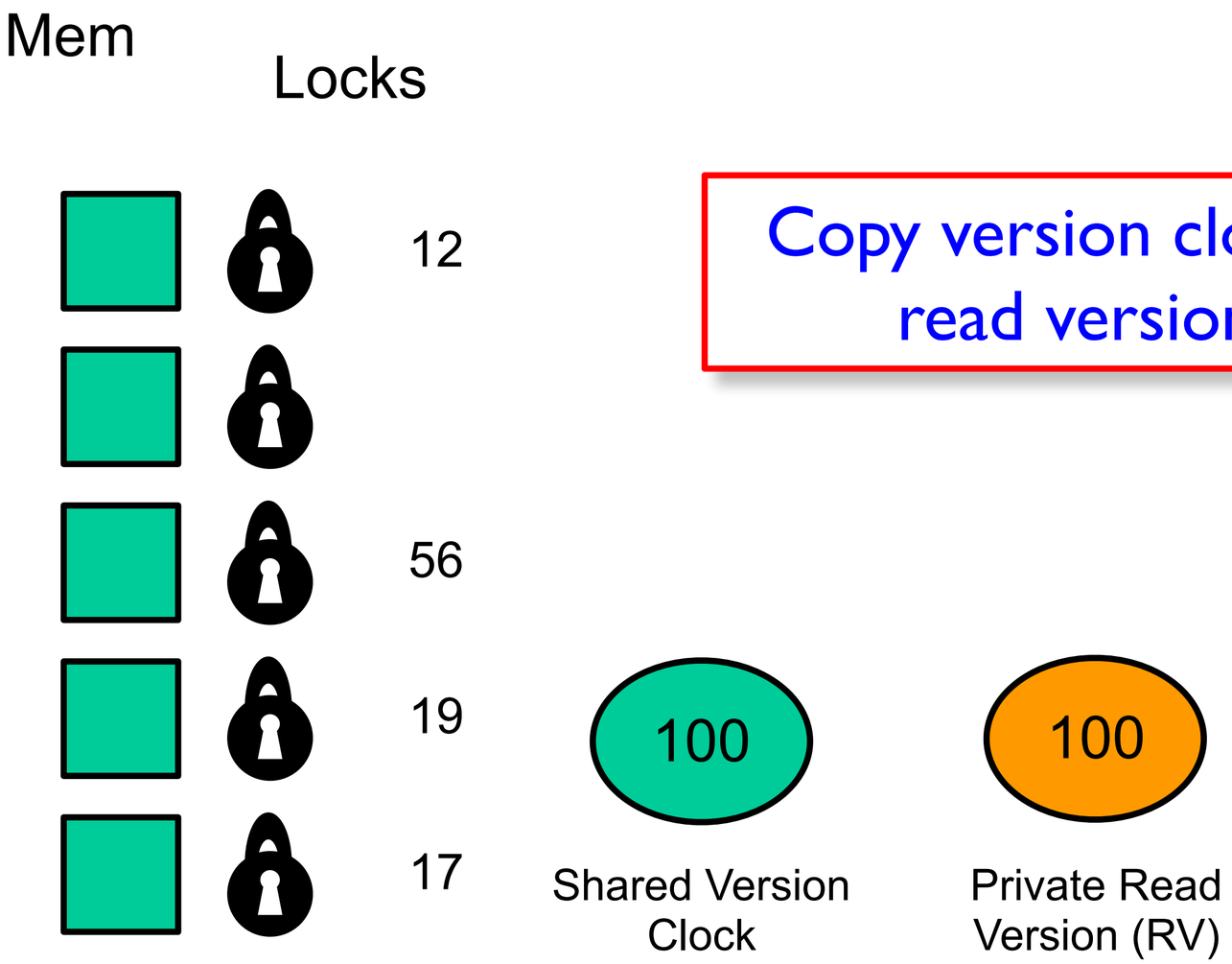
Transaction A: (zombie)
reads y = 4
computes 1/(x-y)

Divide by zero FAIL!

Solution: The Global Clock

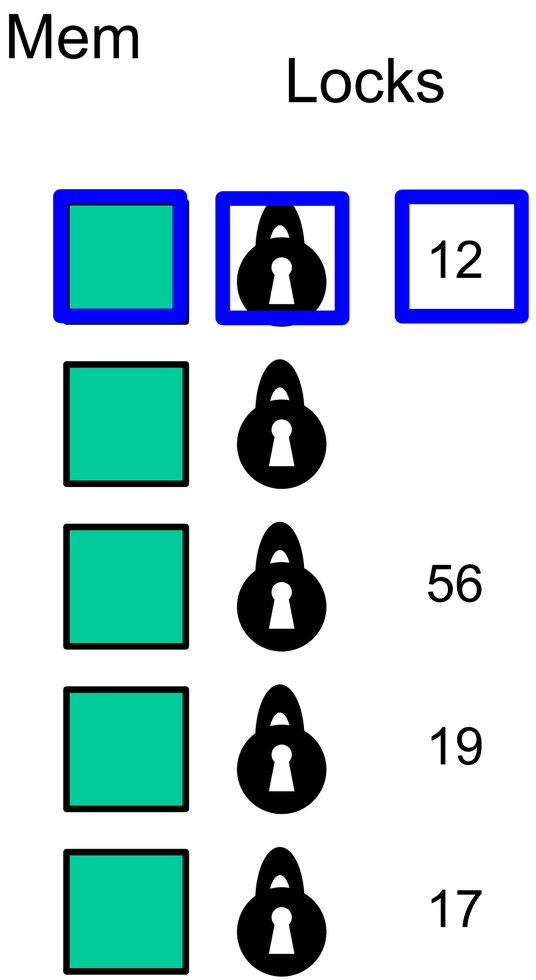
- Have one shared global clock
- Incremented by (small subset of) writing transactions
- Read by all transactions
- Used to validate that state worked on is always consistent

Read-Only Transactions



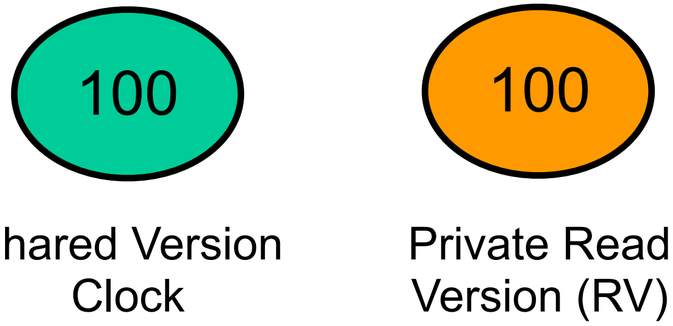
Copy version clock to local read version clock

Read-Only Transactions

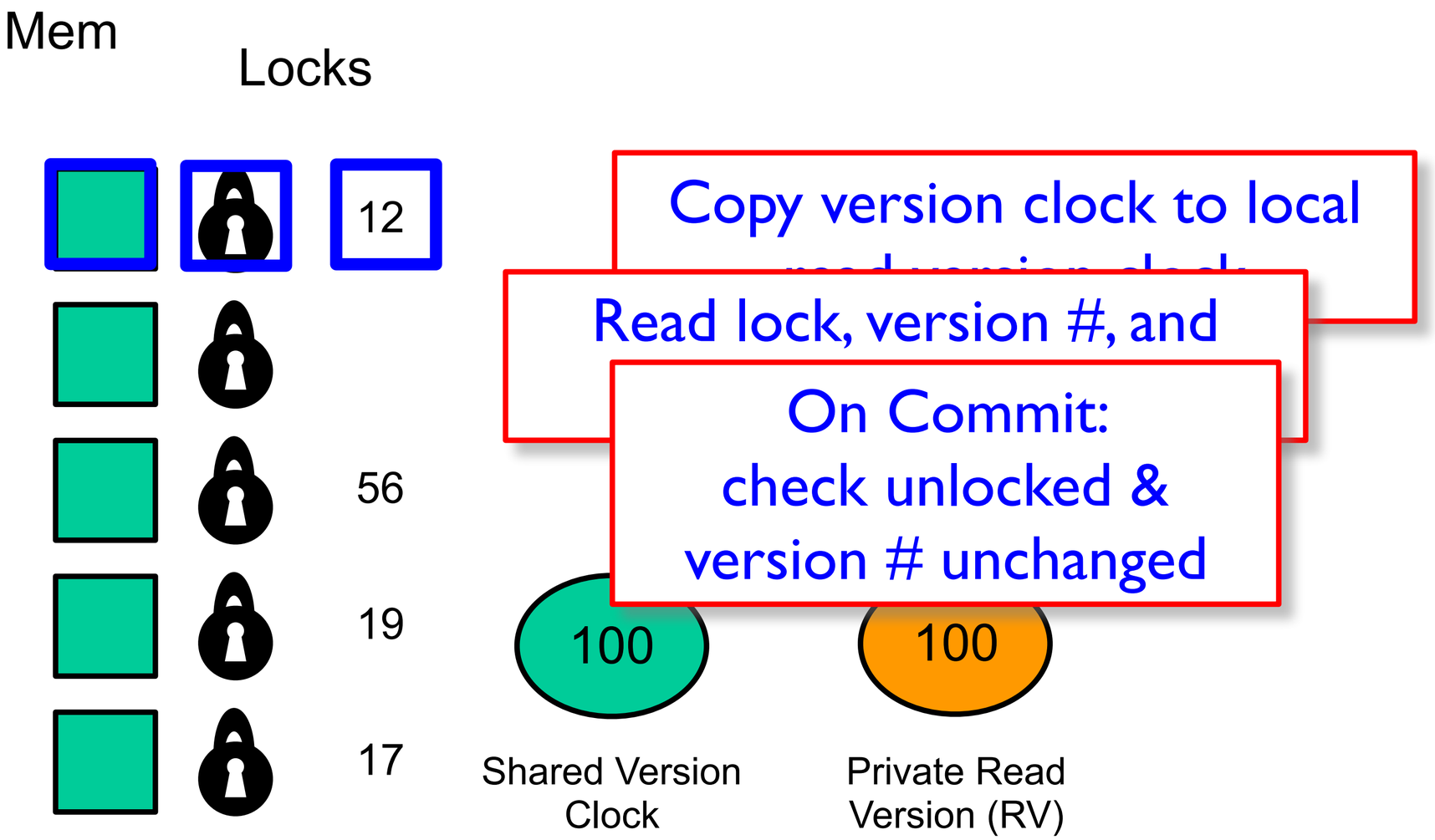


Copy version clock to local read version clock

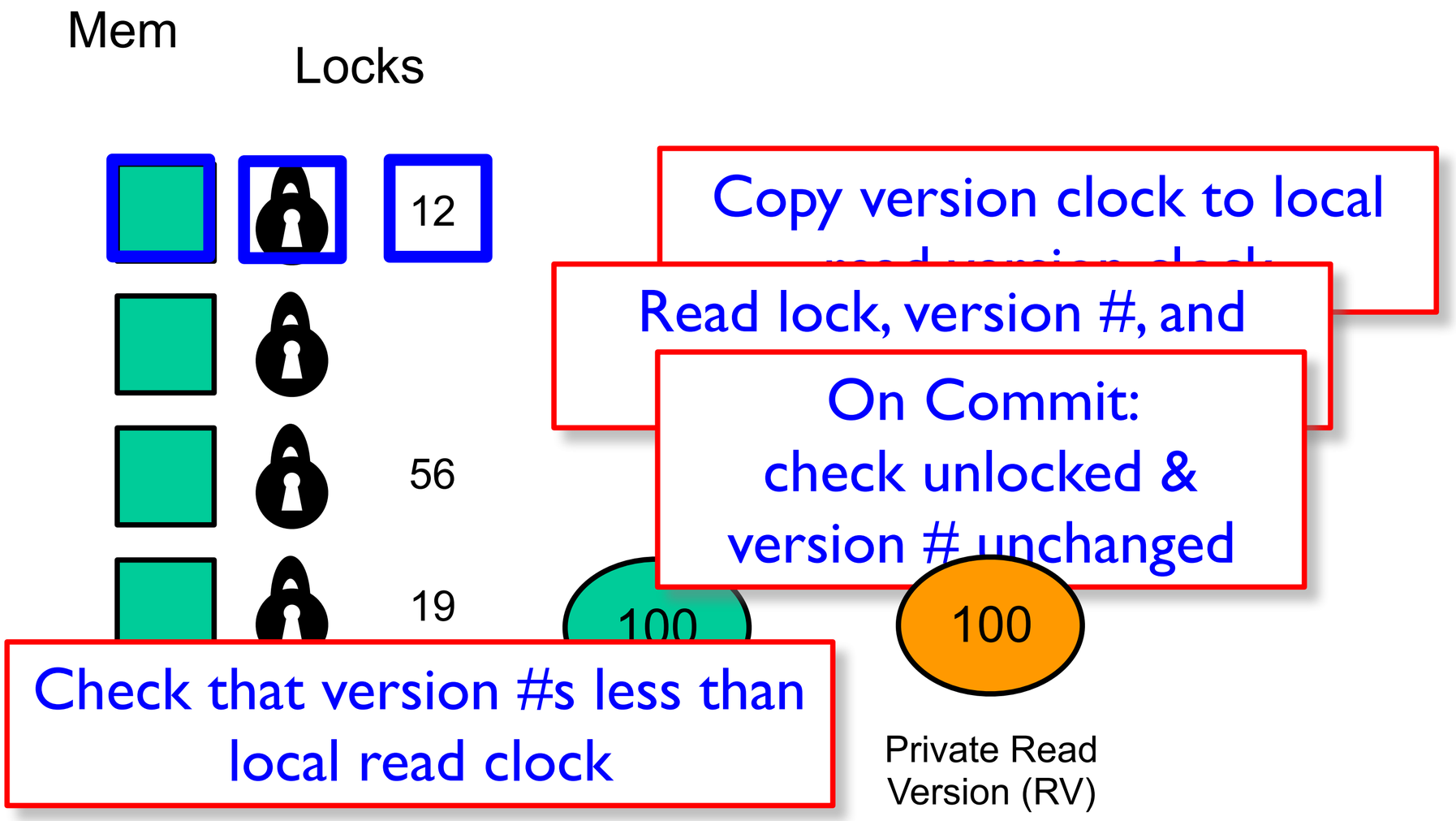
Read lock, version #, and memory



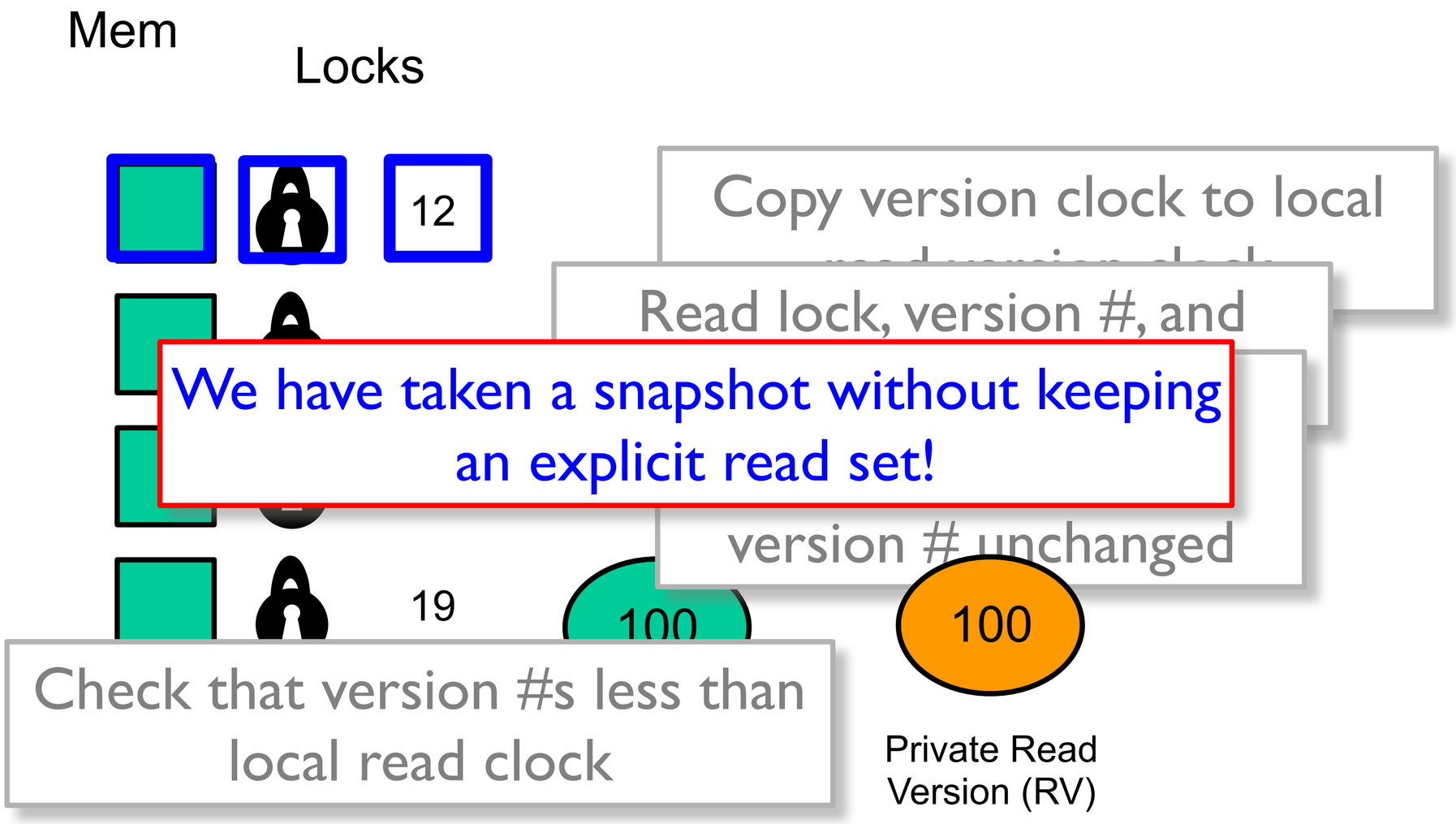
Read-Only Transactions



Read-Only Transactions

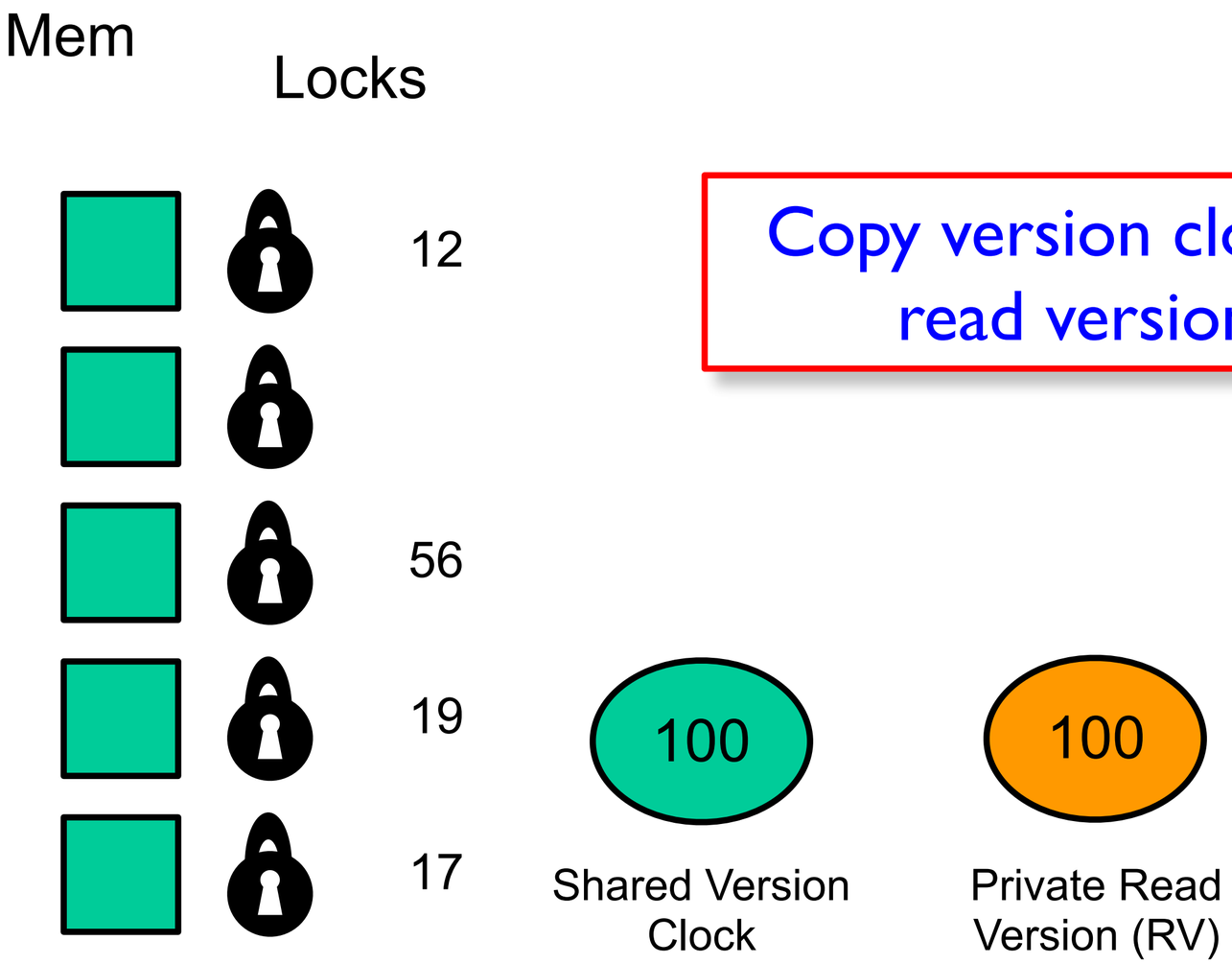


Read-Only Transactions



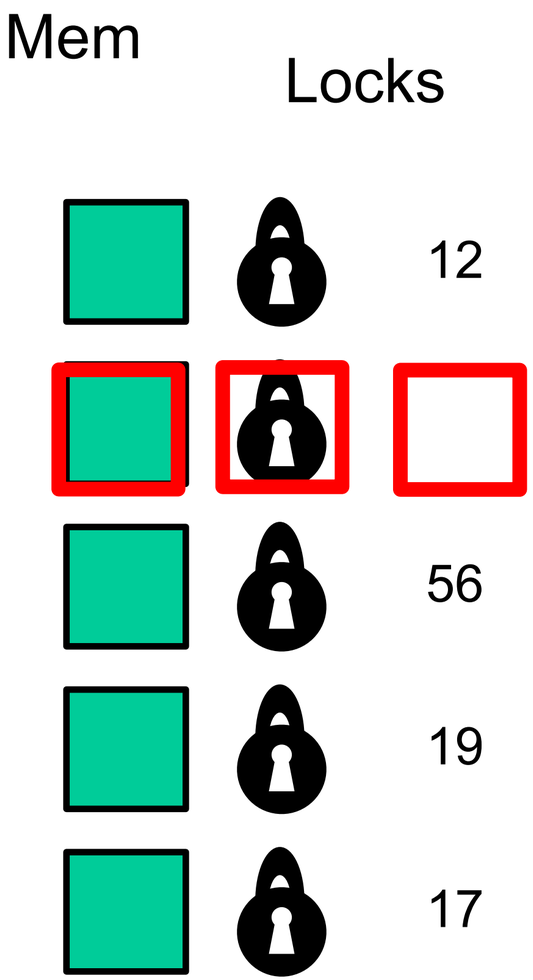
We have taken a snapshot without keeping an explicit read set!

Regular Transactions



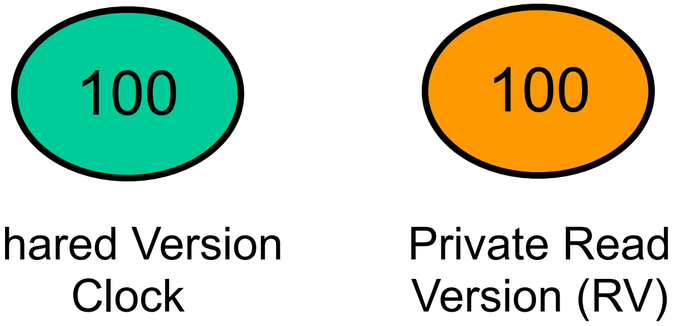
Copy version clock to local read version clock

Regular Transactions

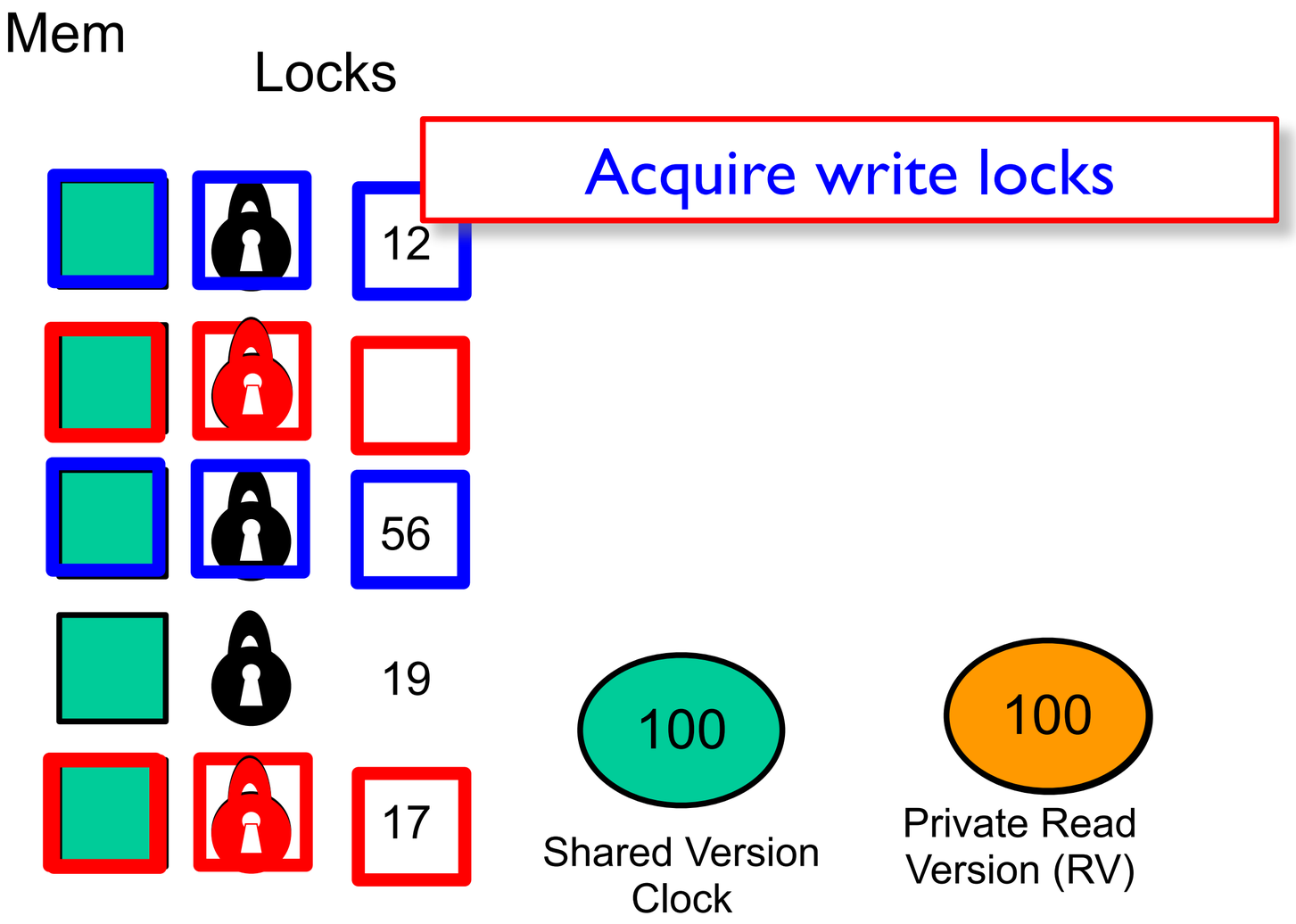


Copy version clock to local read version clock

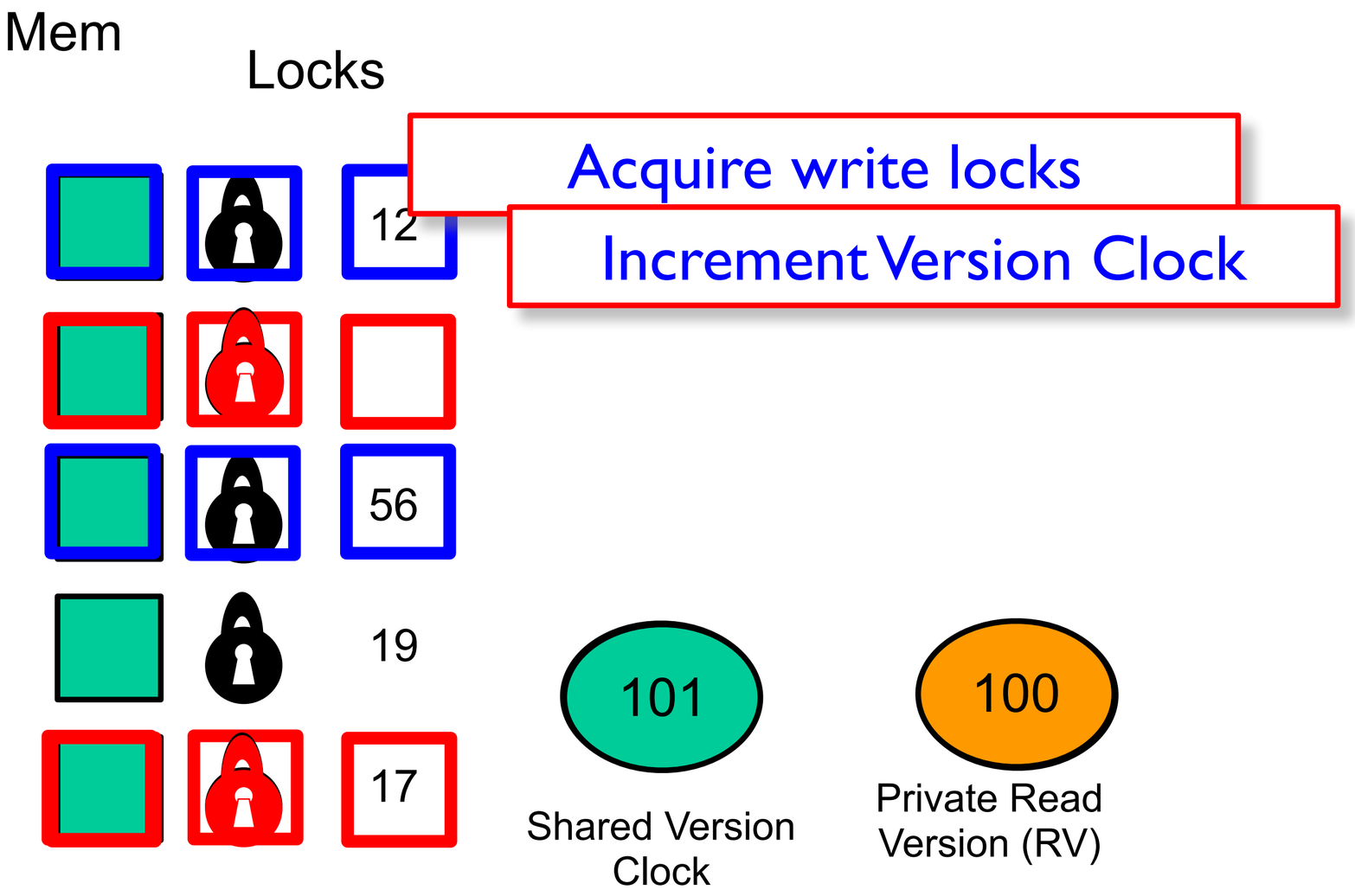
On read/write, check:
Unlocked & version # < RV
Add to R/W set



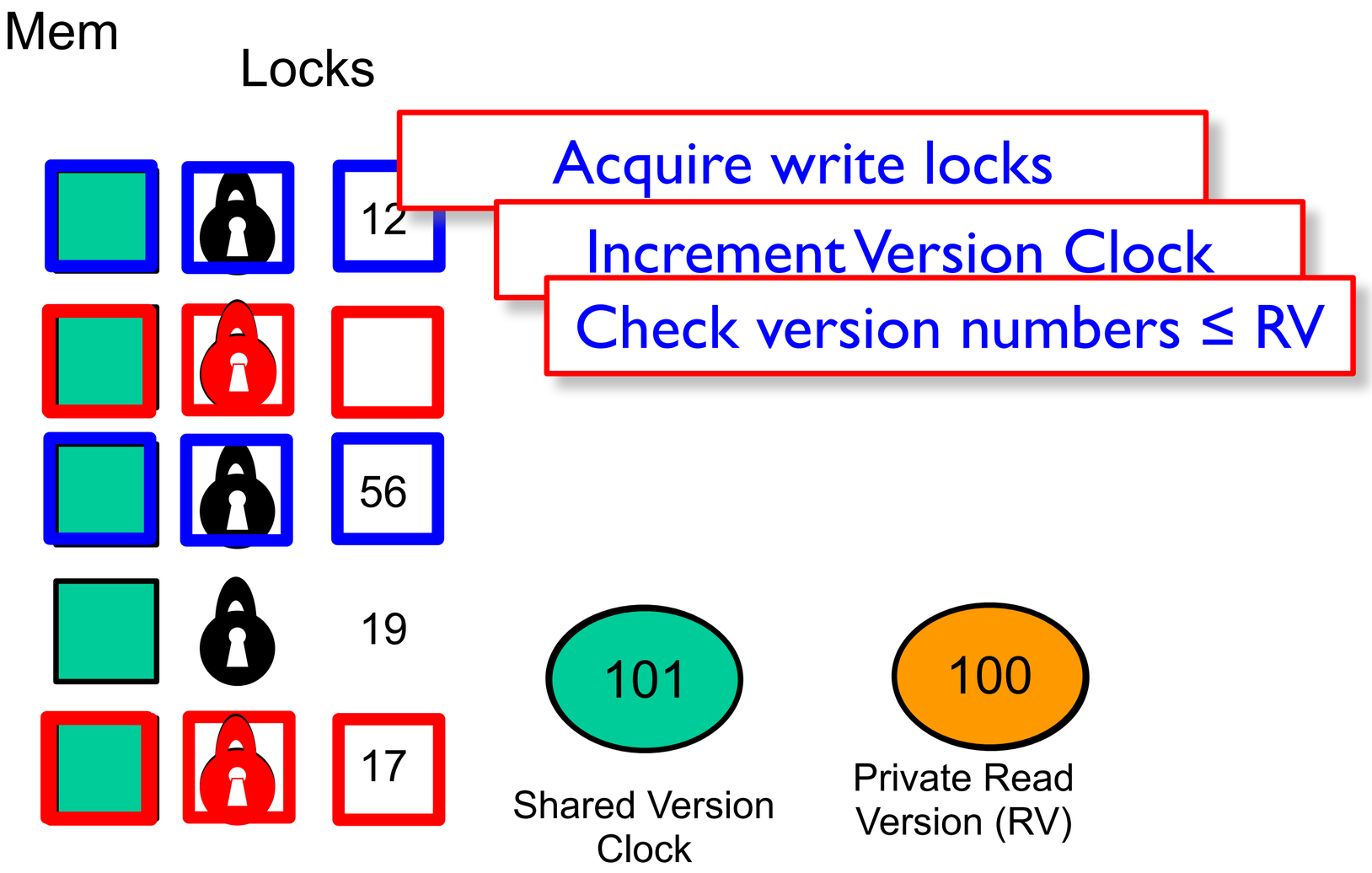
On Commit


Mem
Locks
Acquire write locks
12
56
19
17
100
Shared Version Clock
100
Private Read Version (RV)

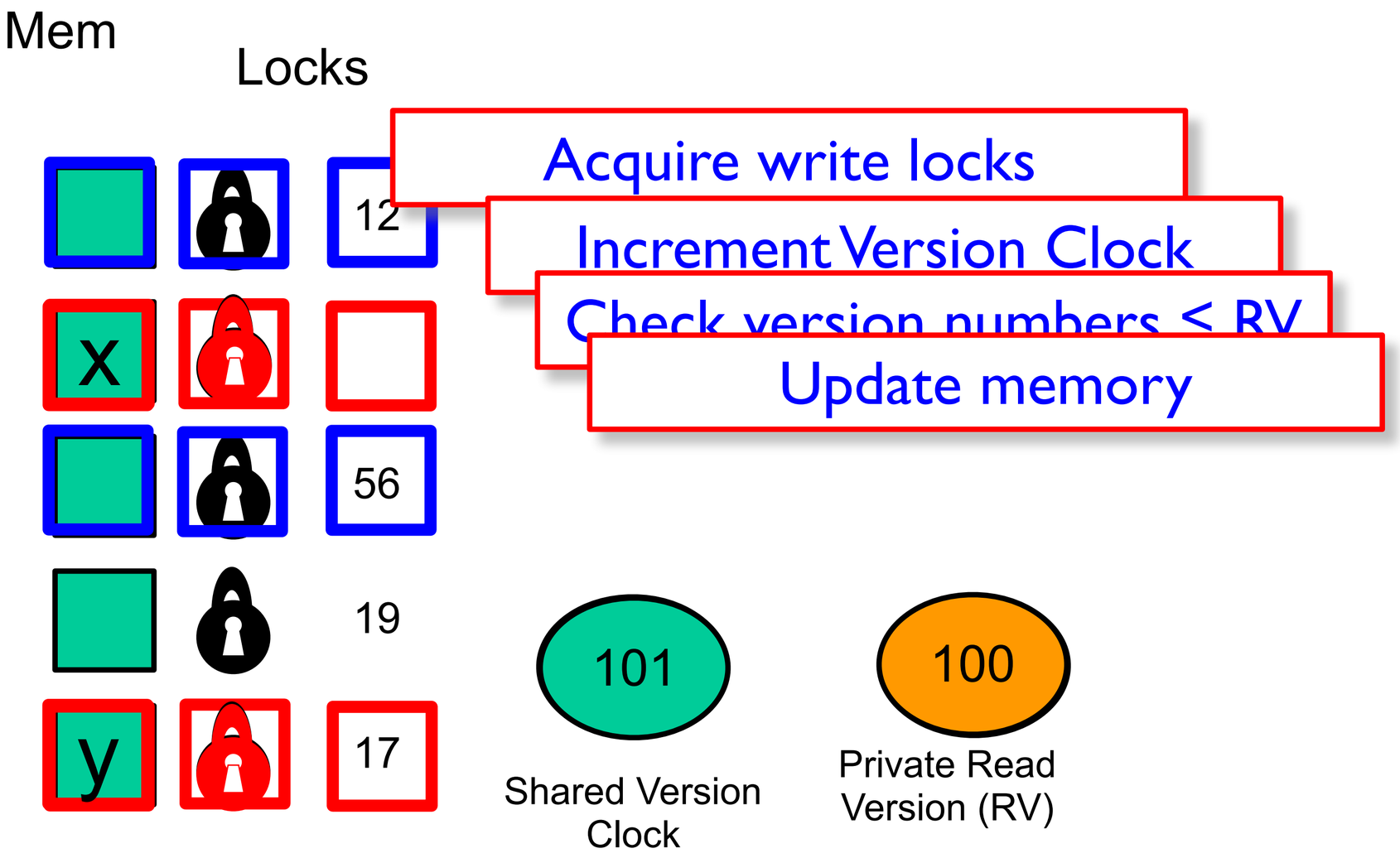
On Commit


Mem
Locks
12
56
19
17
Acquire write locks
Increment Version Clock
101
Shared Version Clock
100
Private Read Version (RV)

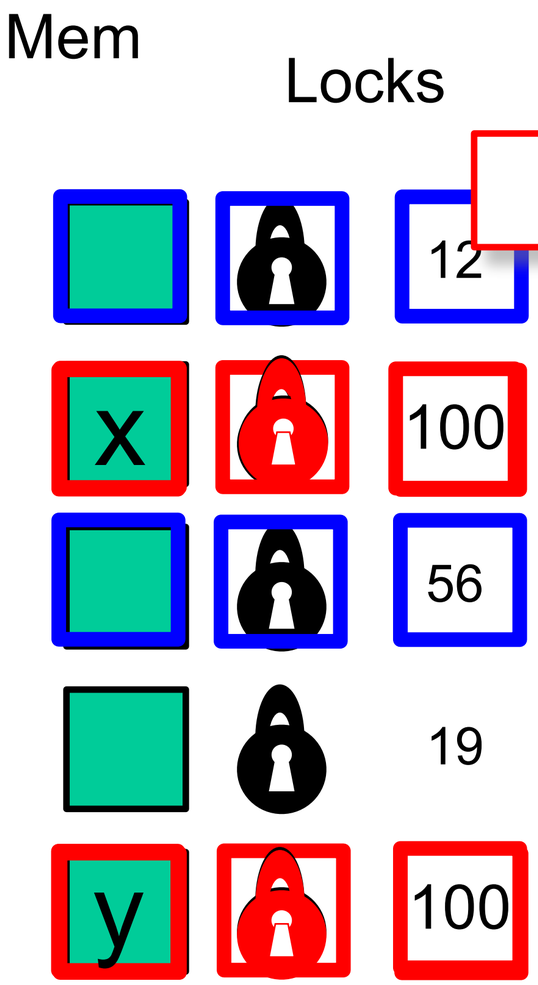
On Commit


Mem
Locks
12
56
19
17
Acquire write locks
Increment Version Clock
Check version numbers ≤ RV
101
Shared Version Clock
100
Private Read Version (RV)

On Commit


Mem
Locks
12
x
56
19
y
17
Acquire write locks
Increment Version Clock
Check version numbers < RV
Update memory
101
Shared Version Clock
100
Private Read Version (RV)

On Commit



Acquire write locks

Increment Version Clock

Check version numbers < RV

Update memory

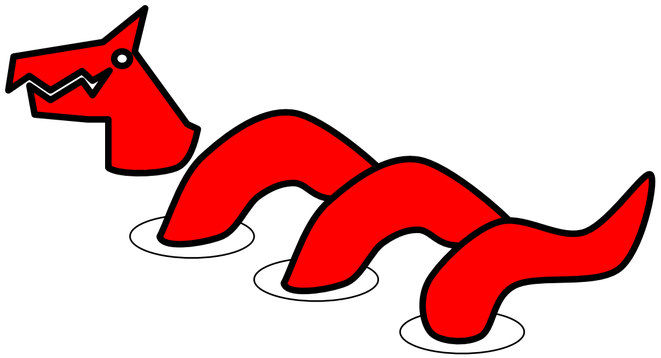
Update write version #s

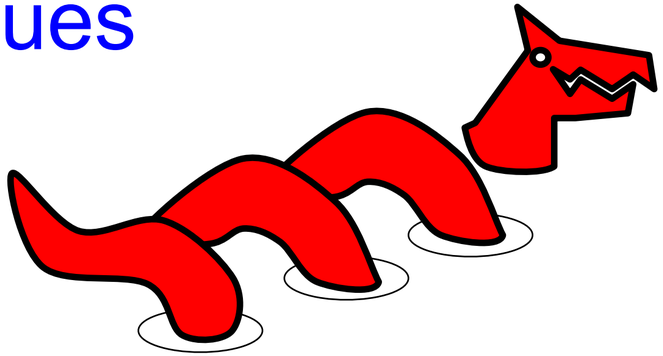
Shared Version Clock

Private Read Version (RV)

TM Design Issues

- Implementation choices
- Language design issues
- Semantic issues





Granularity

- Object
  - managed languages, Java, C#, …
  - Easy to control interactions between transactional & non-trans threads
- Word
  - C, C++, …
  - Hard to control interactions between transactional & non-trans threads

Direct/Deferred Update

- *Deferred*
  - modify private copies & install on commit
  - Commit requires work
  - Consistency easier
- *Direct*
  - Modify in place, roll back on abort
  - Makes commit efficient
  - Consistency harder

Conflict Detection

- Eager
  - Detect before conflict arises
  - “Contention manager” module resolves
- Lazy
  - Detect on commit/abort
- Mixed
  - Eager write/write, lazy read/write …

Conflict Detection

- Eager detection may abort transactions that could have committed.
- Lazy detection discards more computation.

Contention Management & Scheduling

- How to resolve conflicts?
- Who moves forward and who rolls back?
- Lots of empirical work but formal work in infancy

Contention Manager Strategies

- Exponential backoff
- Priority to
  - Oldest?
  - Most work?
  - Non-waiting?
- None Dominates
- But needed anyway

Judgment of Solomon

I/O & System Calls?

- Some I/O revocable
  - Provide transaction-safe libraries
  - Undoable file system/ DB calls
- Some not
  - Opening cash drawer
  - Firing missile

I/O & System Calls

- One solution: make transaction irrevocable
  - If transaction tries I/O, switch to irrevocable mode.
- There can be only one …
  - Requires serial execution
- No explicit aborts
  - In irrevocable transactions

Exceptions

```
int i = 0;
try {
  atomic {
    i++;
    node = new Node();
  }
} catch (Exception e) {
  print(i);
}
```

Exceptions

Throws OutOfMemoryException!

```
int i = 0;
try {
  atomic {
    i++;
    node = new Node();
  }
} catch (Exception e) {
  print(i);
}
```

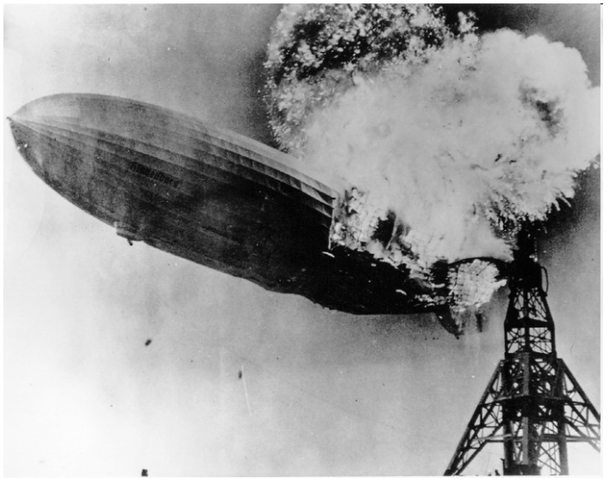
Exceptions

Throws OutOfMemoryException!

```
int i = 0;
try {
  atomic {
    i++;
    node = new Node();
  }
} catch (Exception e) {
  print(i);
}
```

What is printed?

Unhandled Exceptions



- Aborts transaction
  - Preserves invariants
  - Safer
- Commits transaction
  - Like locking semantics
  - What if exception object refers to values modified in transaction?

Nested Transactions

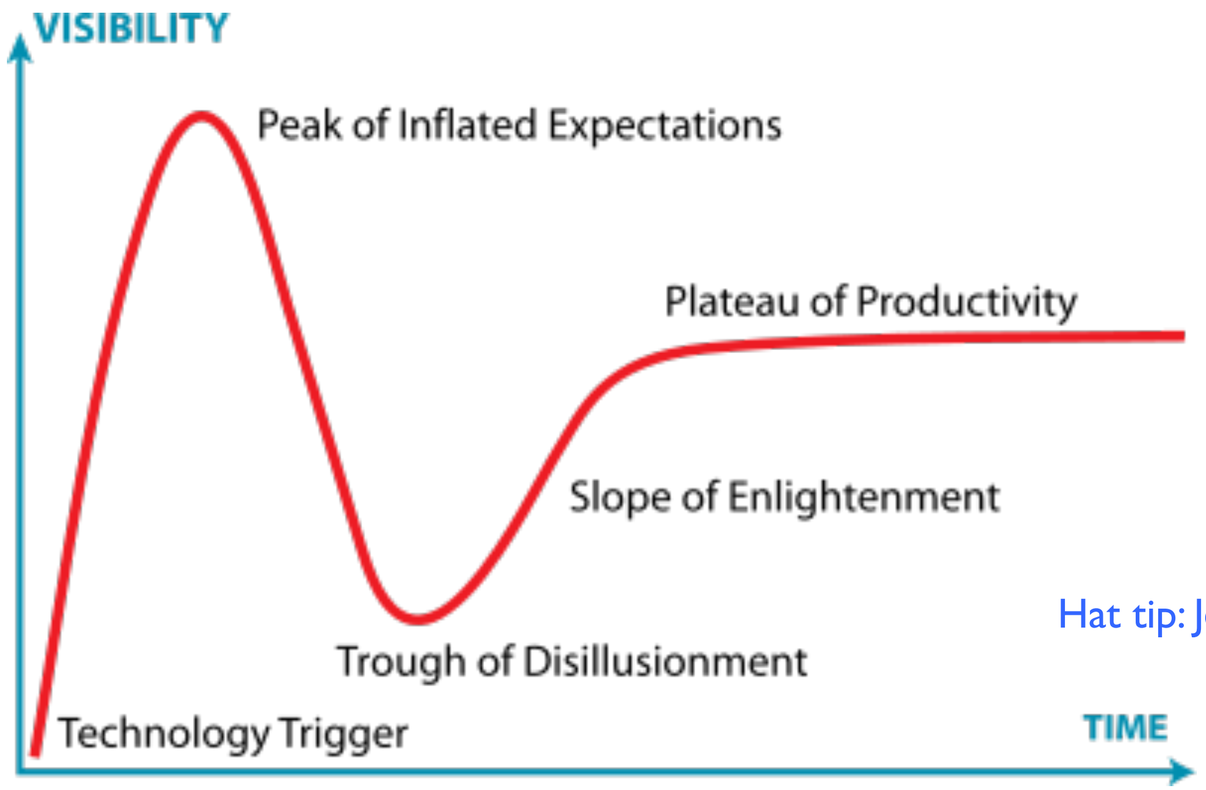
```
atomic void foo() {
  bar();
}

atomic void bar() {
 …
}
```

Nested Transactions

- Needed for modularity
  - Who knew that cosine() contained a transaction?
- Flat nesting
  - If child aborts, so does parent
- First-class nesting
  - If child aborts, partial rollback of child only

Gartner Hype Cycle



Hat tip: Jeremy Kemp