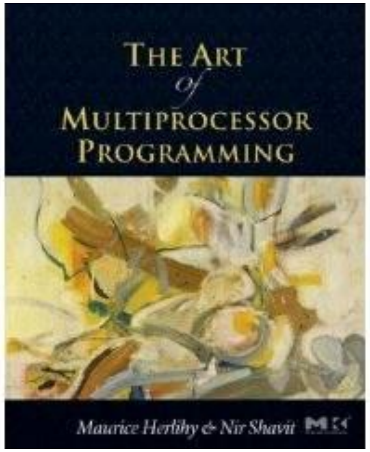
Programming Paradigms for Concurrency
Lecture 8 – Transactional Memory



Based on companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Modified by
Thomas Wies
New York University

Beyond the State of the Art

So far, we covered...

Best practices …

New and clever ideas …

And common-sense observations.

Beyond the State of the Art

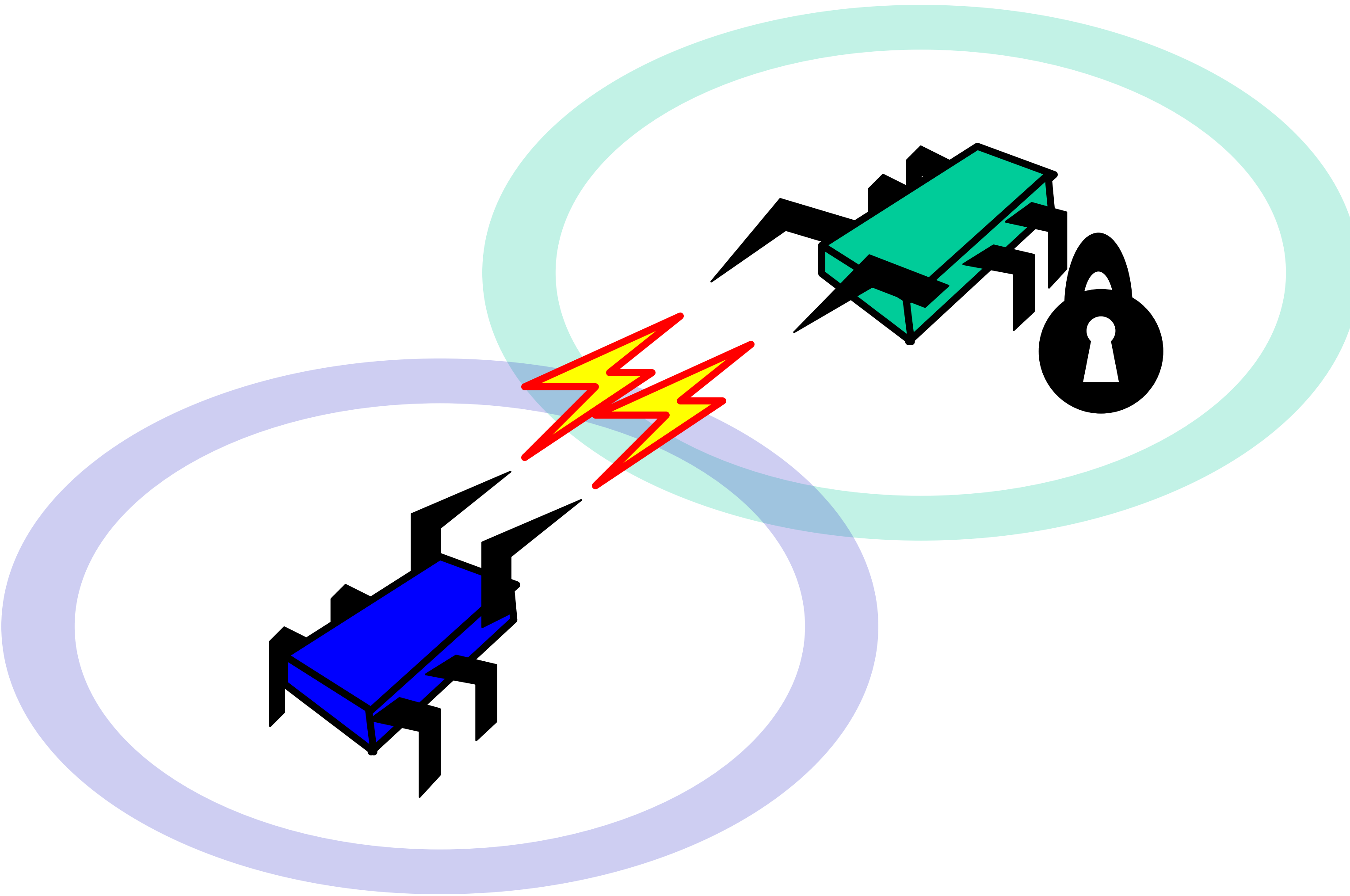
Nevertheless …

Concurrent programming is still too hard …

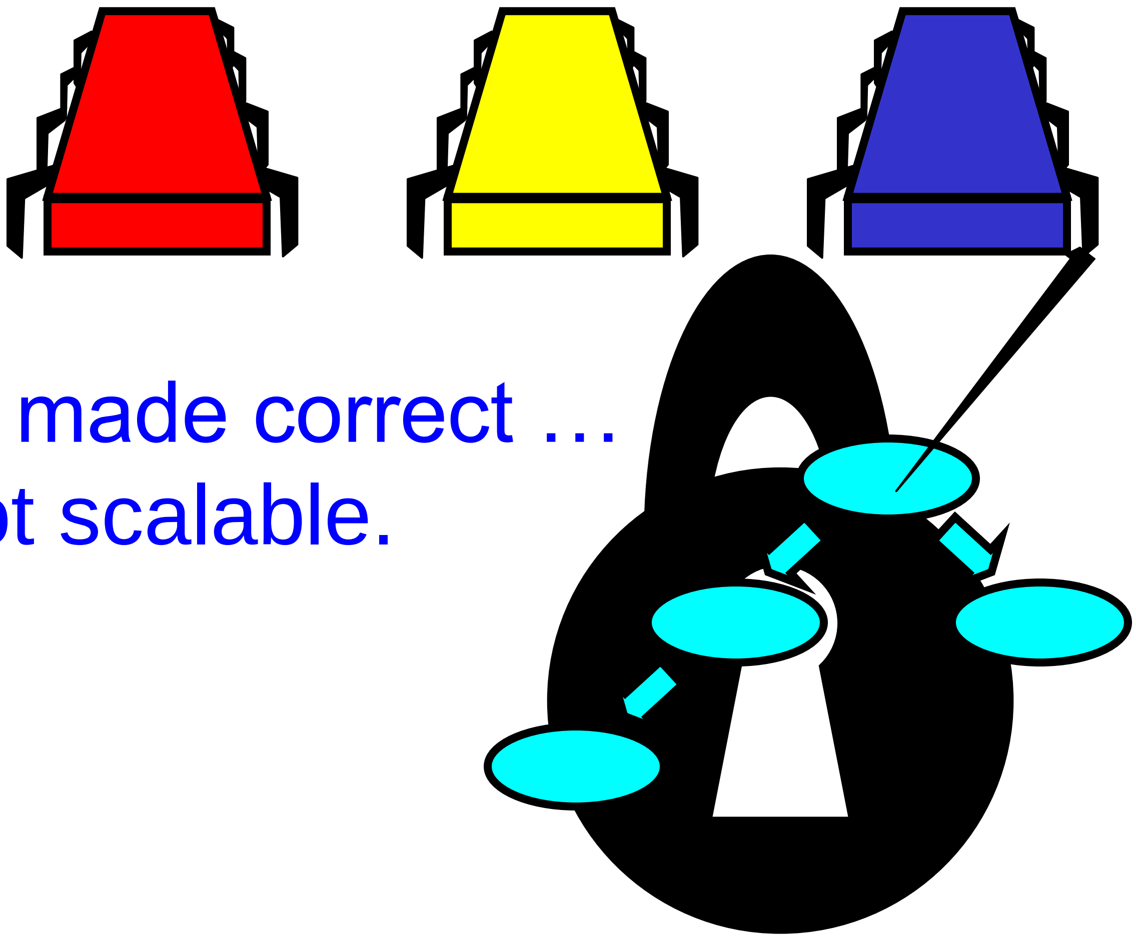
Next, we explore why this is ….

and what we can do about it.

Locking

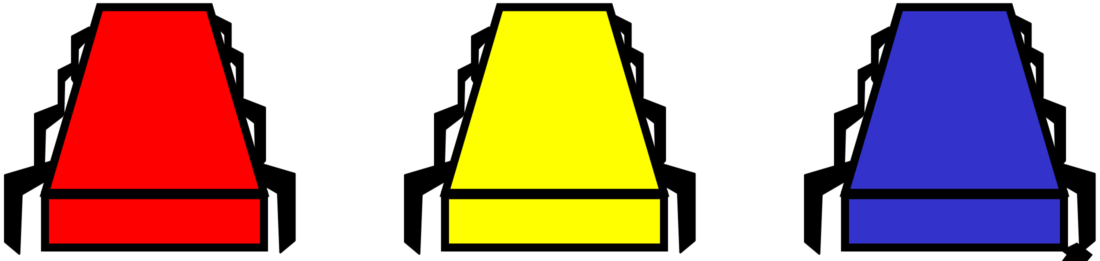


Coarse-Grained Locking

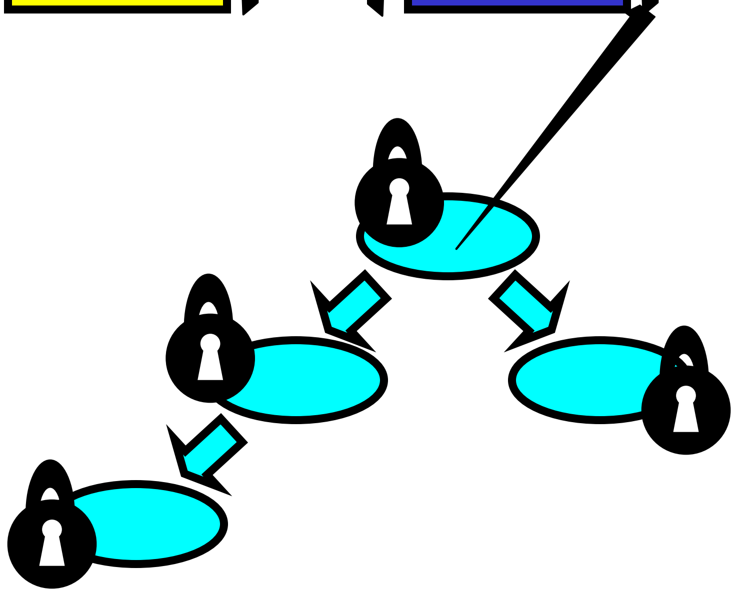


Easily made correct …
But not scalable.

Fine-Grained Locking

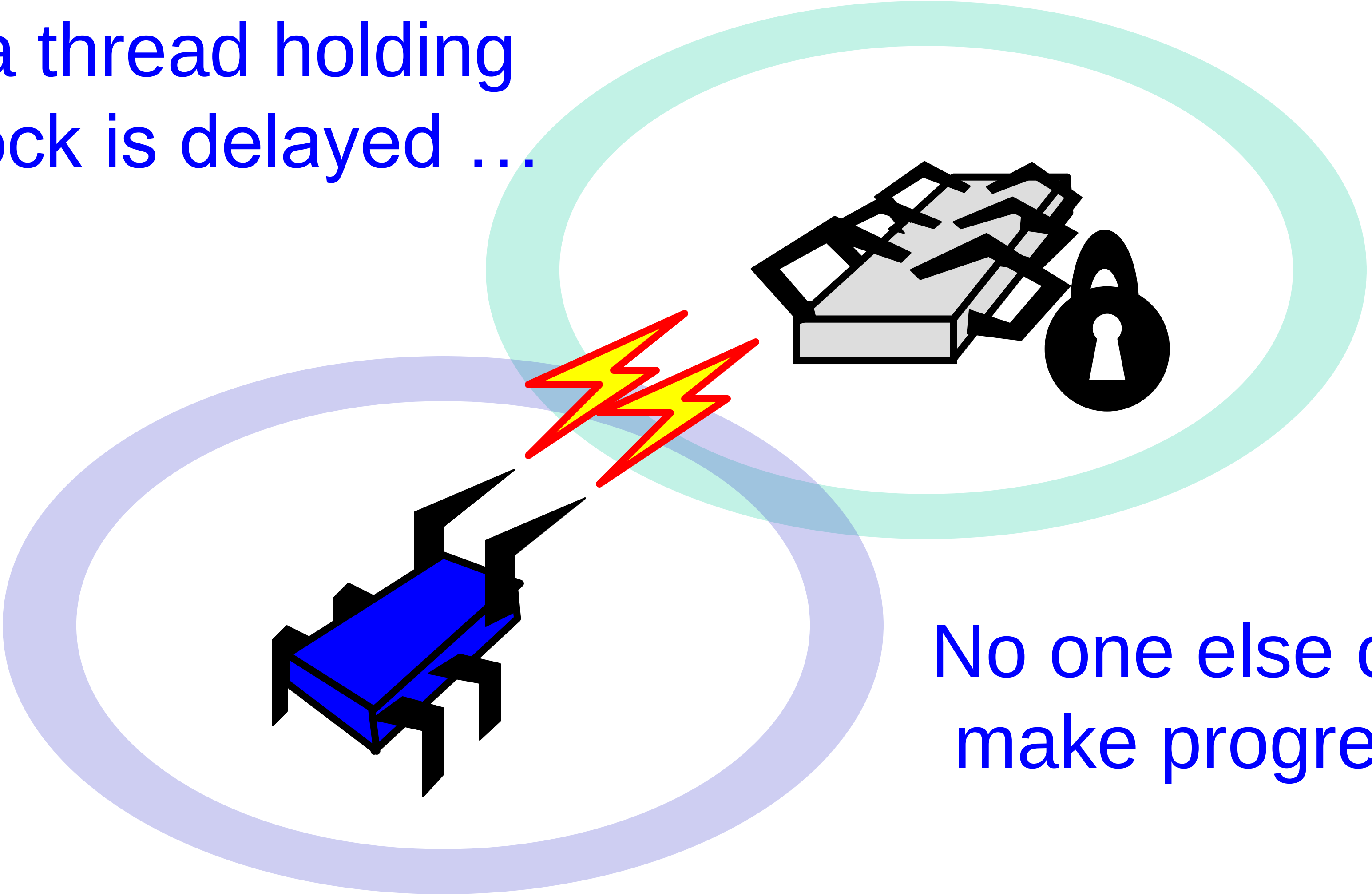


Can be tricky …



Locks are not Robust

If a thread holding a lock is delayed …



No one else can make progress

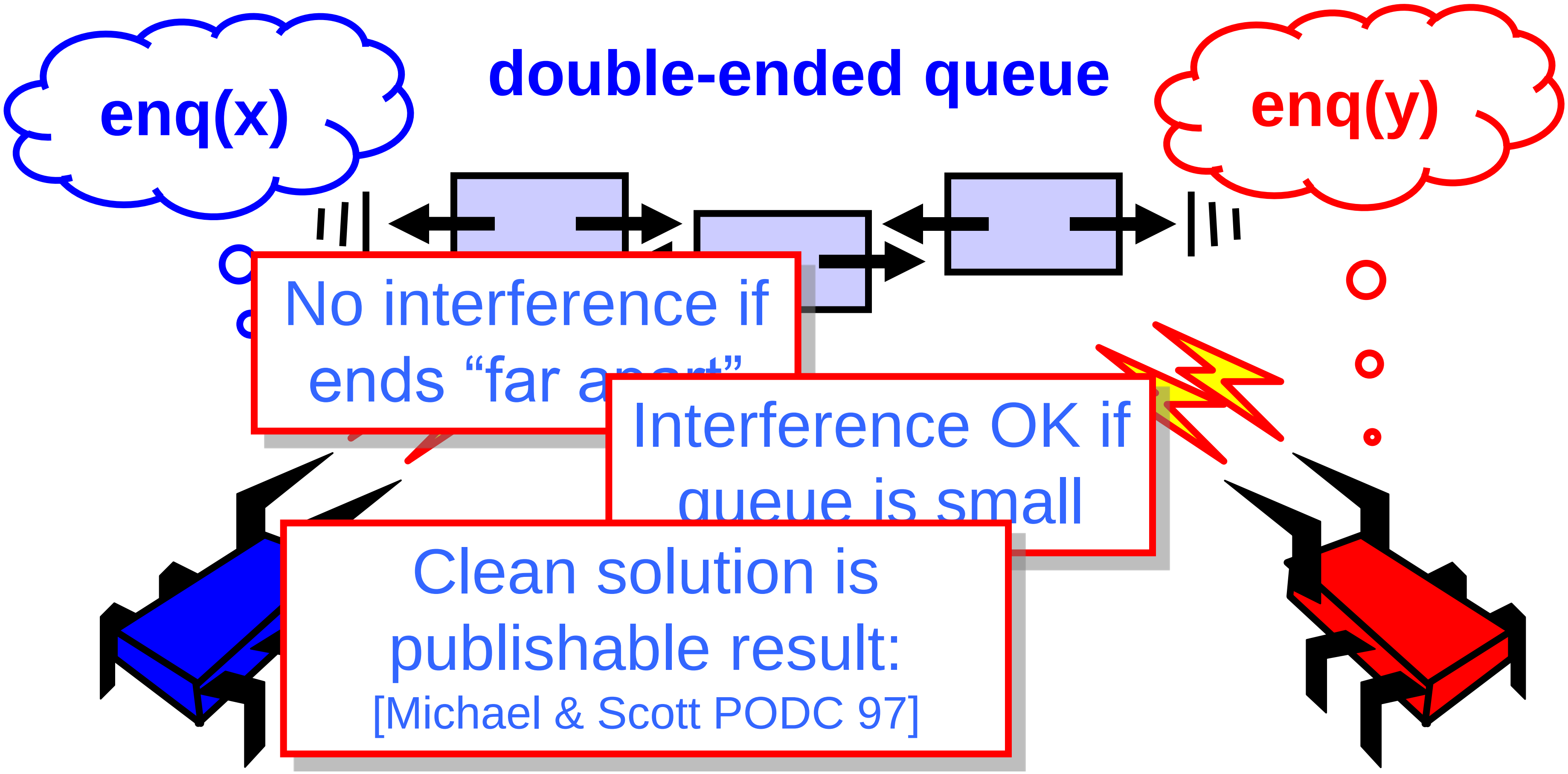
Locking Relies on Conventions

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

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

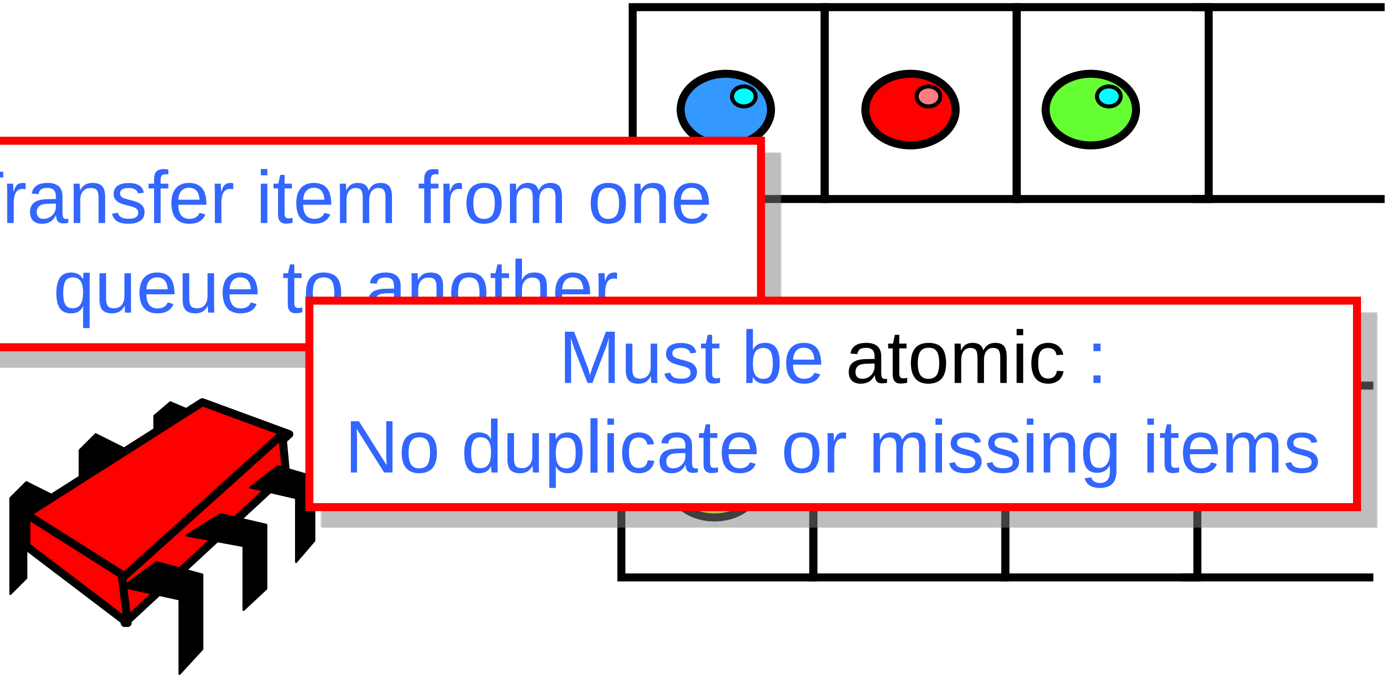


No interference if ends “far apart”

Interference OK if queue is small

Clean solution is publishable result:
[Michael & Scott PODC 97]

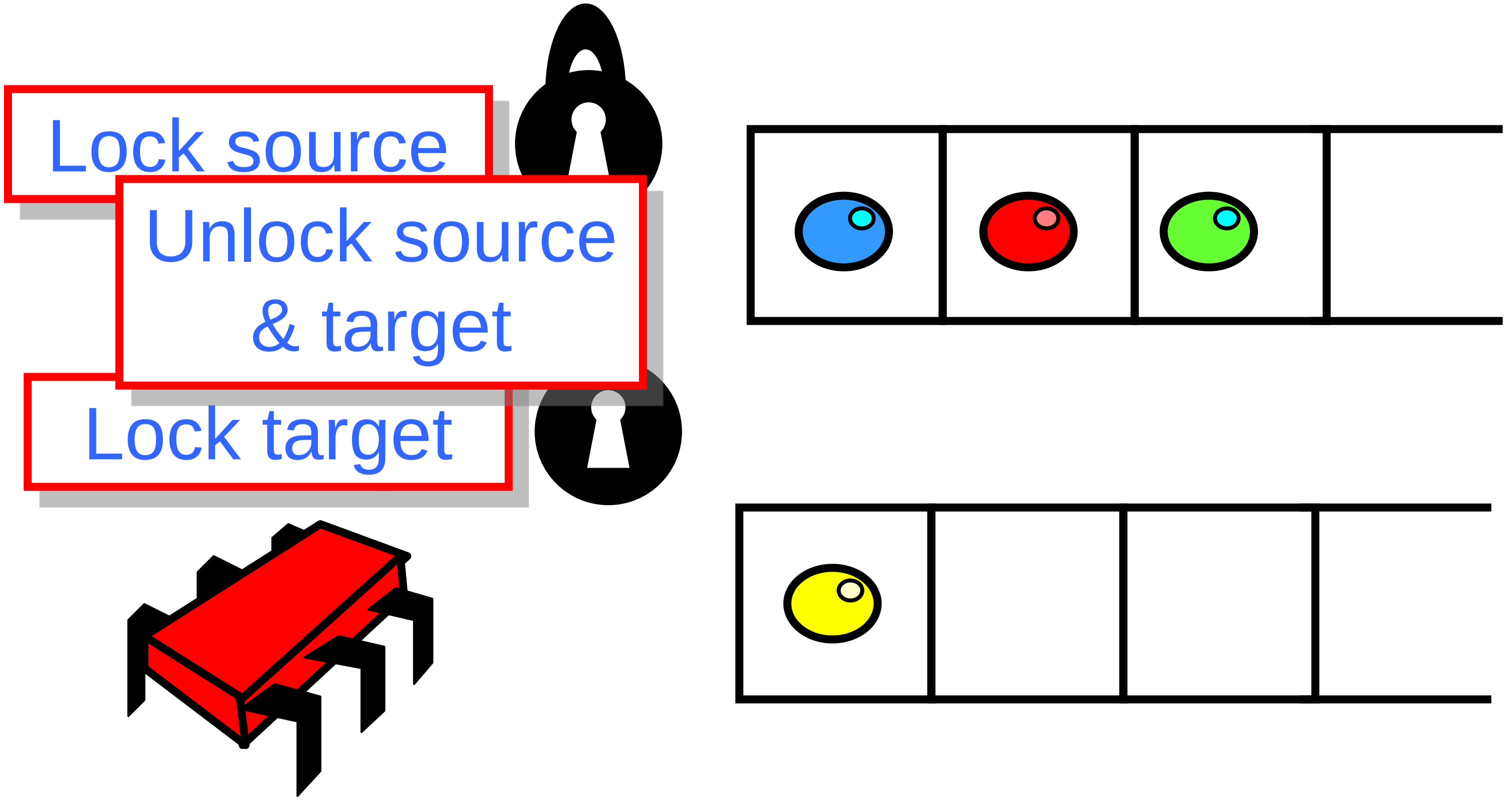
Locks Not Composable



Transfer item from one queue to another

Must be atomic : No duplicate or missing items

Locks Not Composable



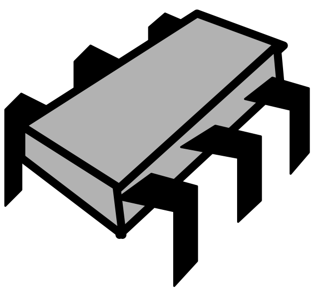
Locks Not Composable

Methods cannot provide internal synchronization

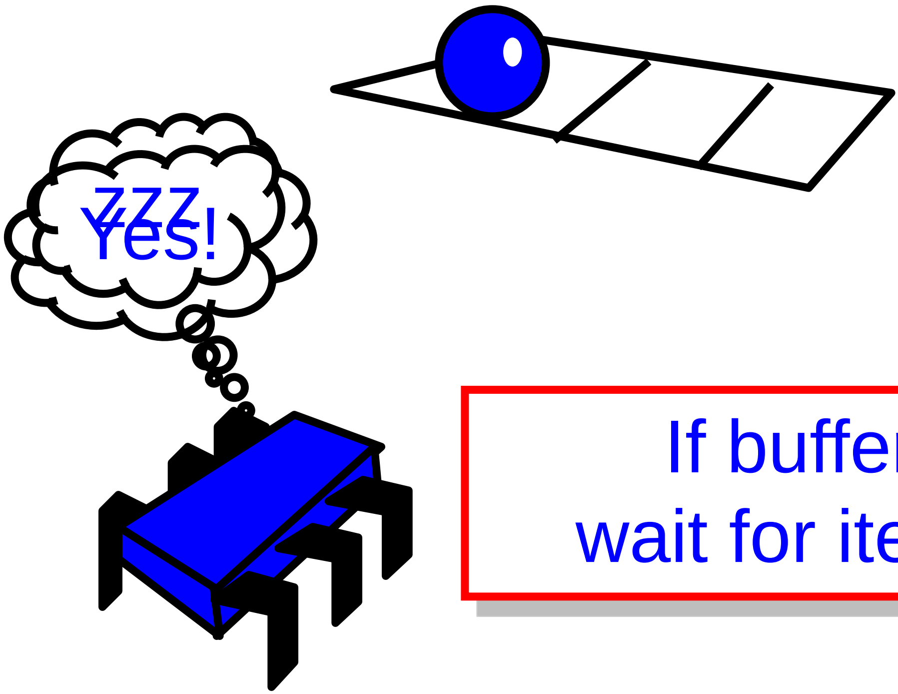
Objects must expose locking protocols to clients

Clients must devise and follow protocols

Abstraction broken!



Monitor Wait and Signal

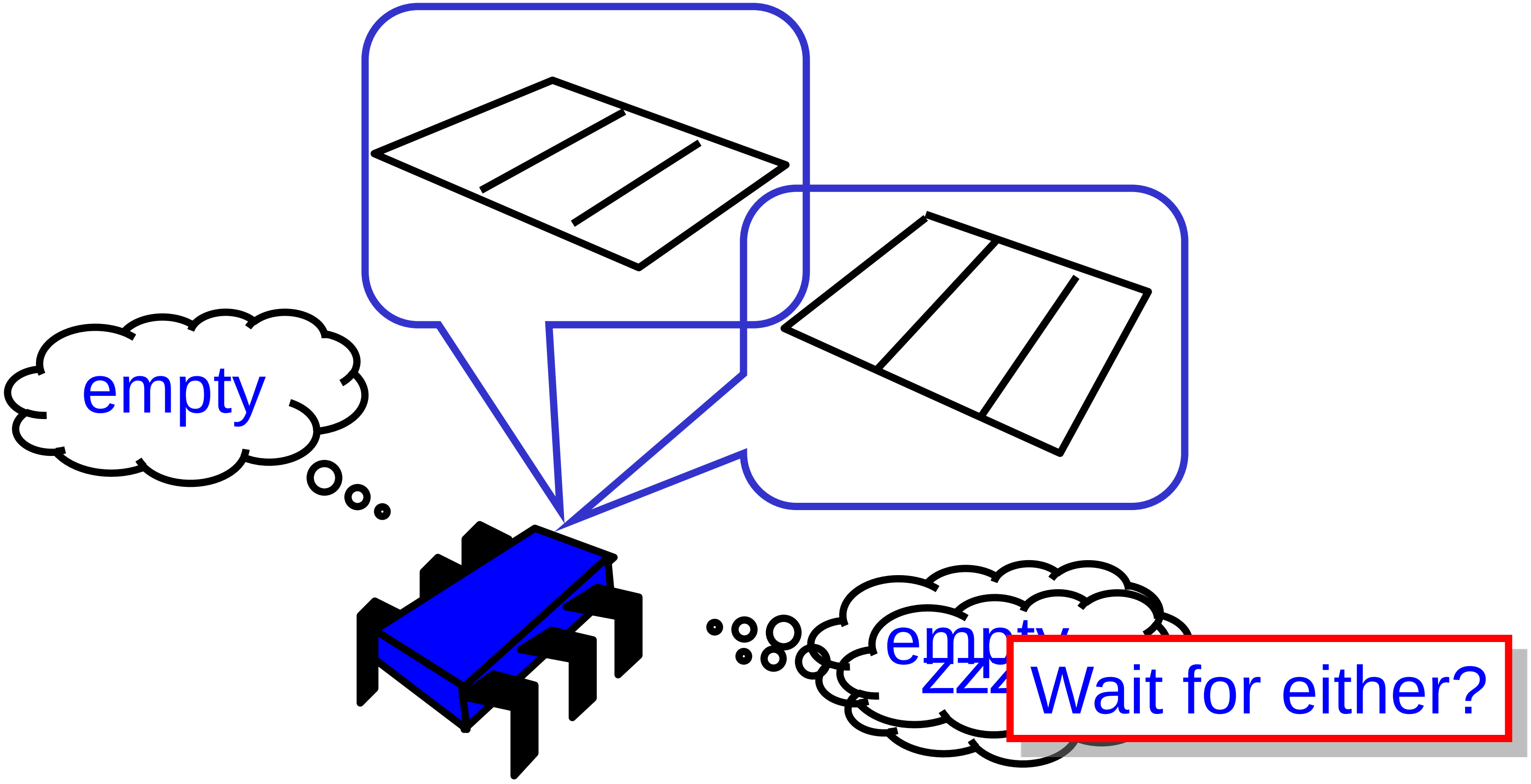


Empty

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
- Alternative Programming Paradigm
  - 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

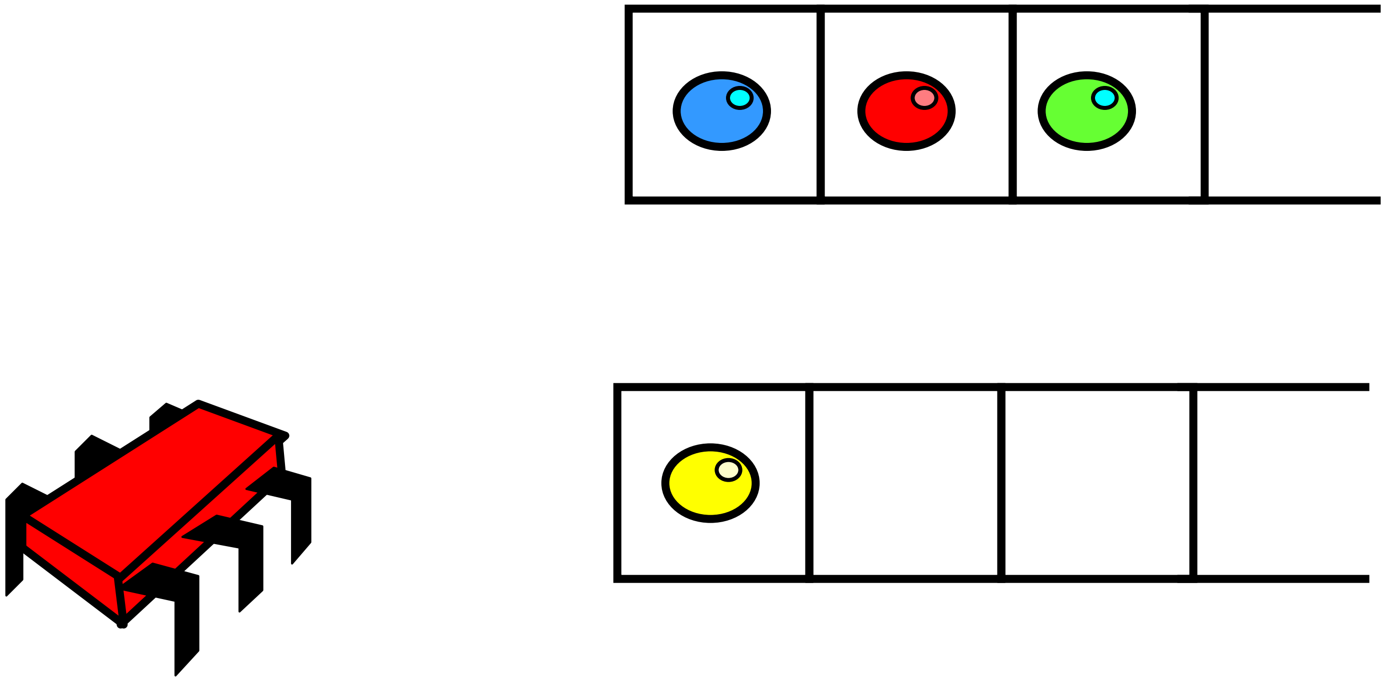
```
public void LeftEnq(item x) {
  atomic {
    Qnode q = new Qnode(x);
    q.left = this.left;
    this.left.right = q;
    this.left = q;
  }
}
```

Enclose in atomic block

Warning

- Not always this simple
  - Conditional waits
  - Enhanced concurrency
  - Complex patterns
- But often it is…

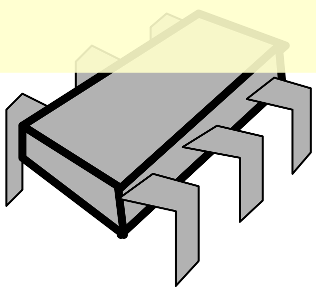
Composition?

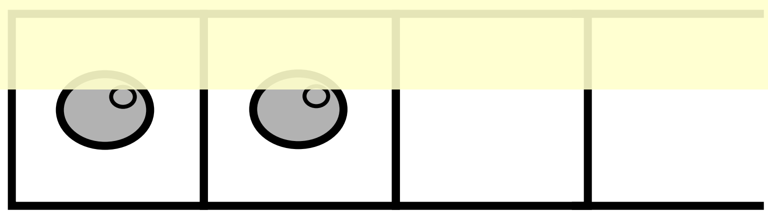


Composition?

```
public void Transfer(Queue<T> q1, q2)
{
 atomic {
  T x = q1.deq();
  q2.enq(x);
 }
}
```

Trivial or what?





Conditional Waiting

```
public T LeftDeq() {
  atomic {
    if (this.left == null)
      retry;
    …

  }
}
```

Roll back transaction and restart when something changes

Composable Conditional Waiting

```
atomic {
 x = q1.deq();
} orElse {
 x = q2.deq();
}
```

Run 1st method. If it retries …

Run 2nd method. If it retries …

Entire statement retries

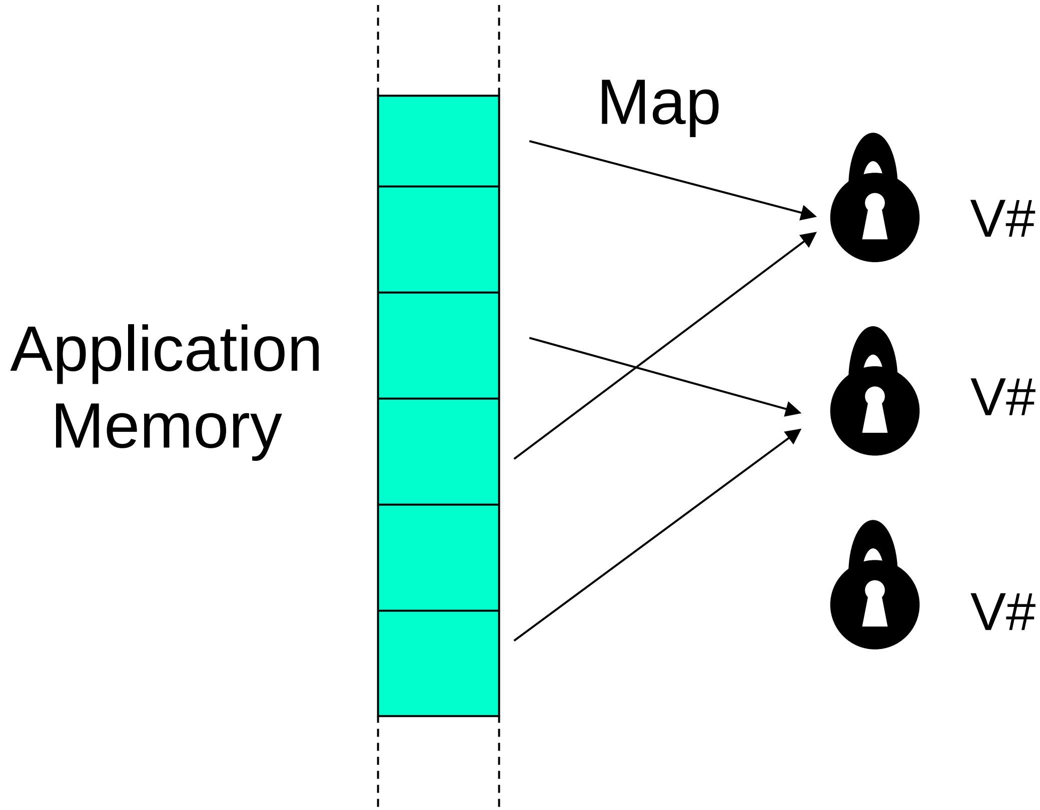
Simple Lock-Based STM

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

Synchronization

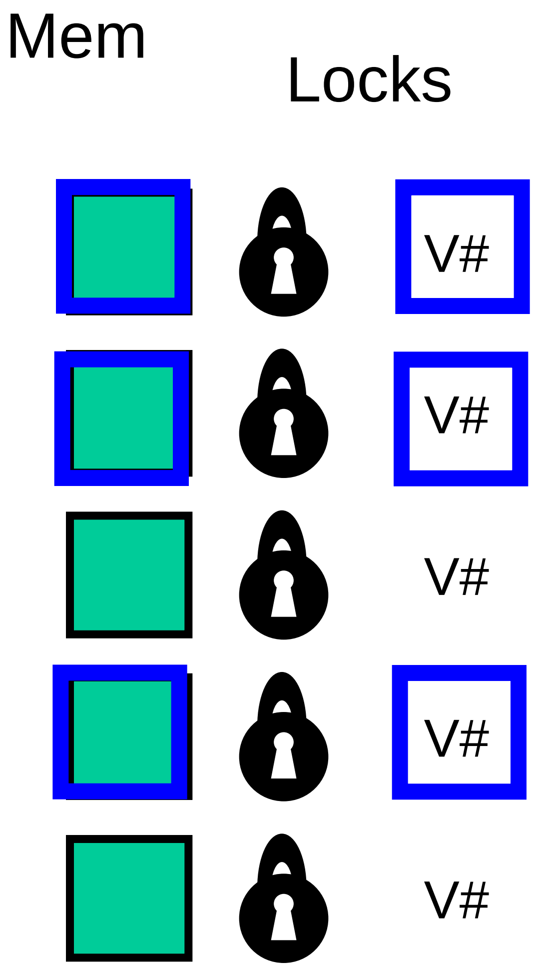
- Transaction keeps
  - **Read set**: locations & values read
  - **Write set**: locations & values to be written
- Deferred update
  - Changes installed at commit
- Lazy conflict detection
  - Conflicts detected at commit

STM: Transactional Locking



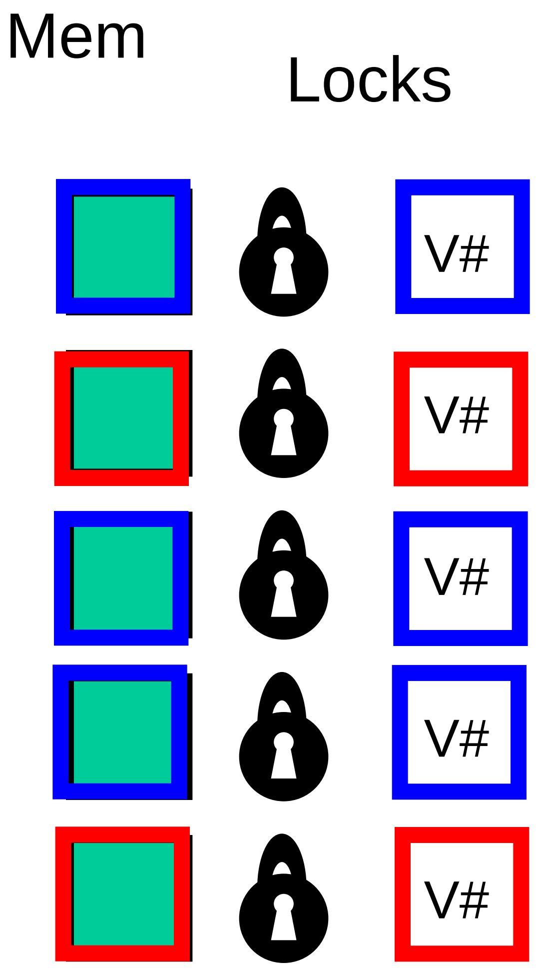
Array of version #s & locks

Reading an Object



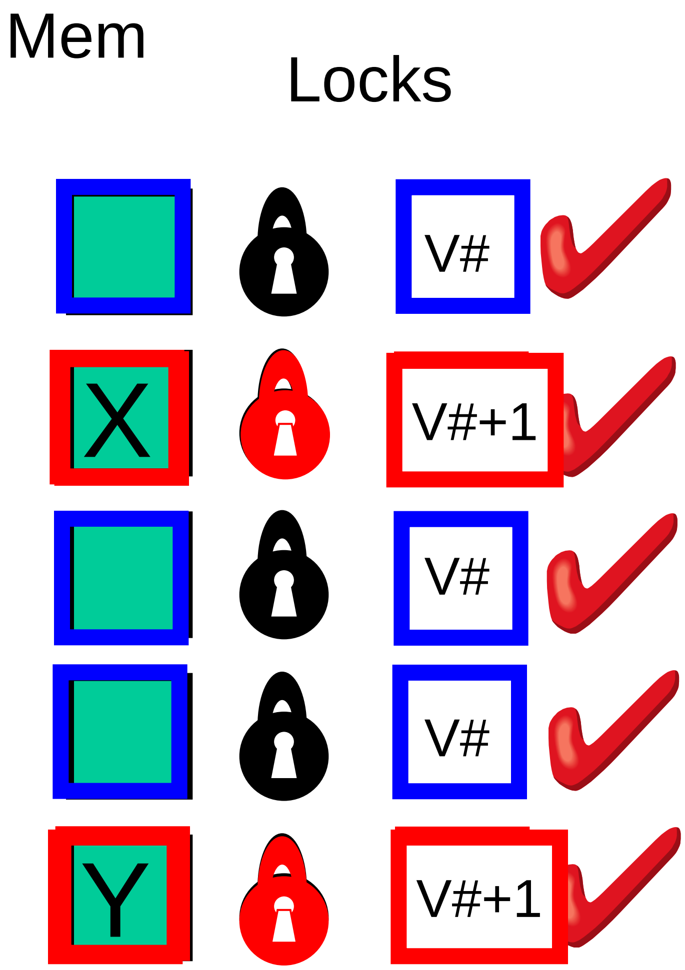
Add version numbers & values to read set

To Write an Object



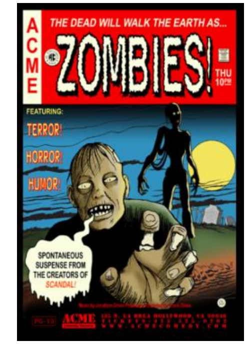
Add version numbers & new values to write set

To Commit



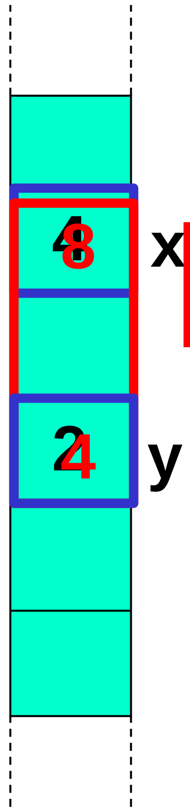
- Acquire write locks
- Check version numbers unchanged
- Install new values
- Increment version numbers
- Unlock.

Problem: Internal Inconsistency



- 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, 4 to y, aborts A)

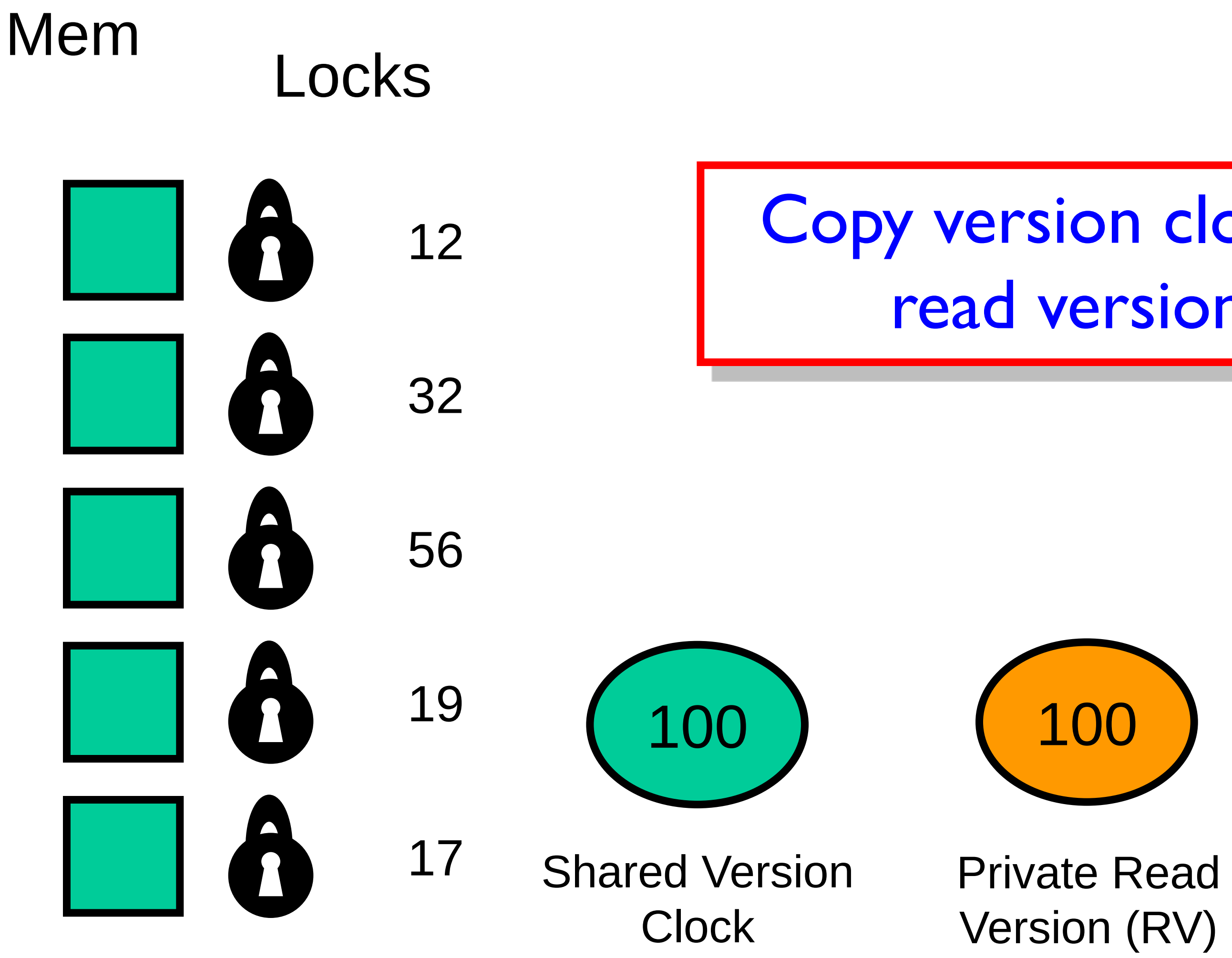
Transaction A: (zombie)
reads y = 4
computes 1/(x-y)

Divide by zero FAIL!

Solution: The Global Clock

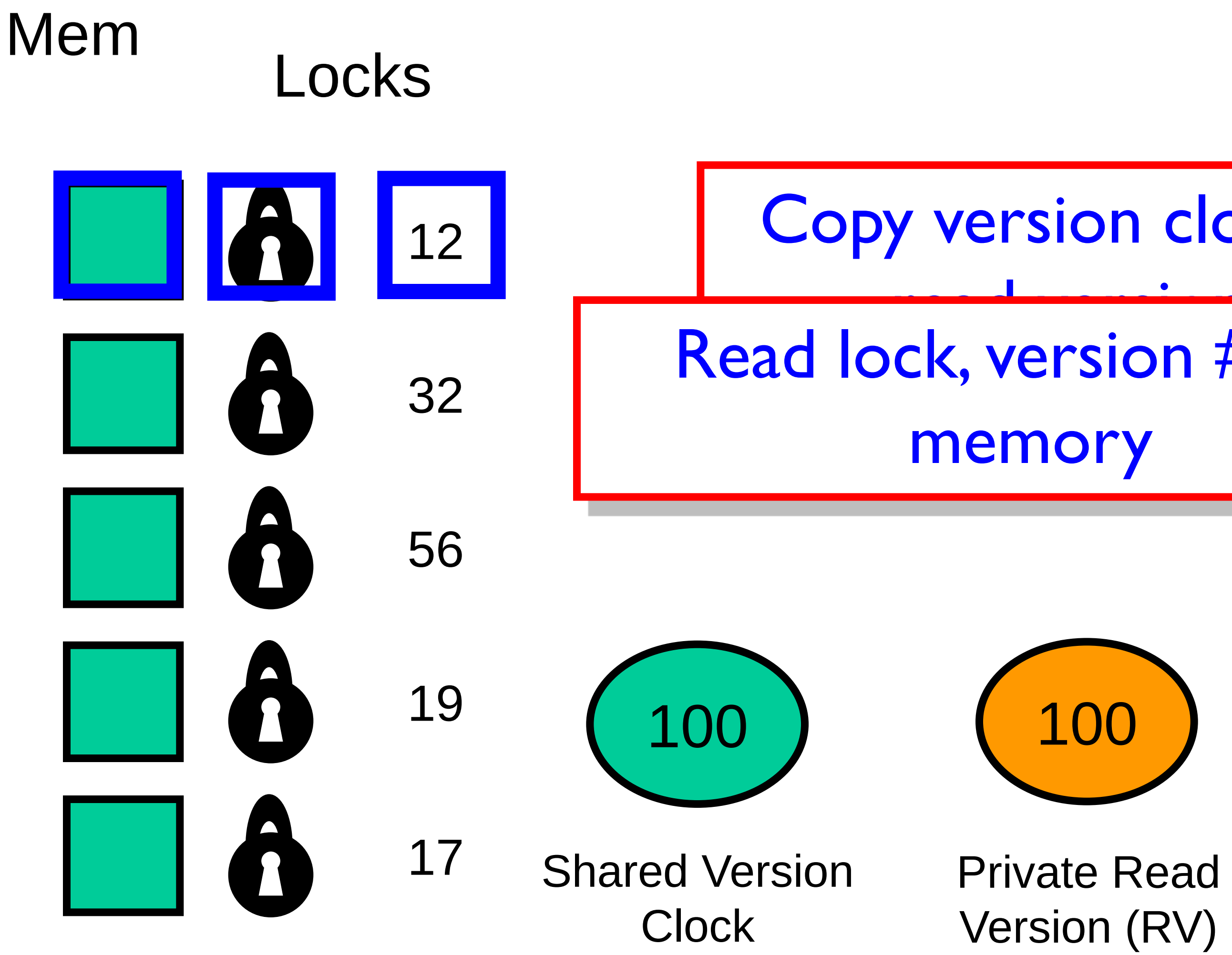
- Have one shared global clock
- Incremented by (small subset of) writing transactions
- Read by all transactions
- Used to validate that state worked on is always consistent

Read-Only Transactions



Copy version clock to local read version clock

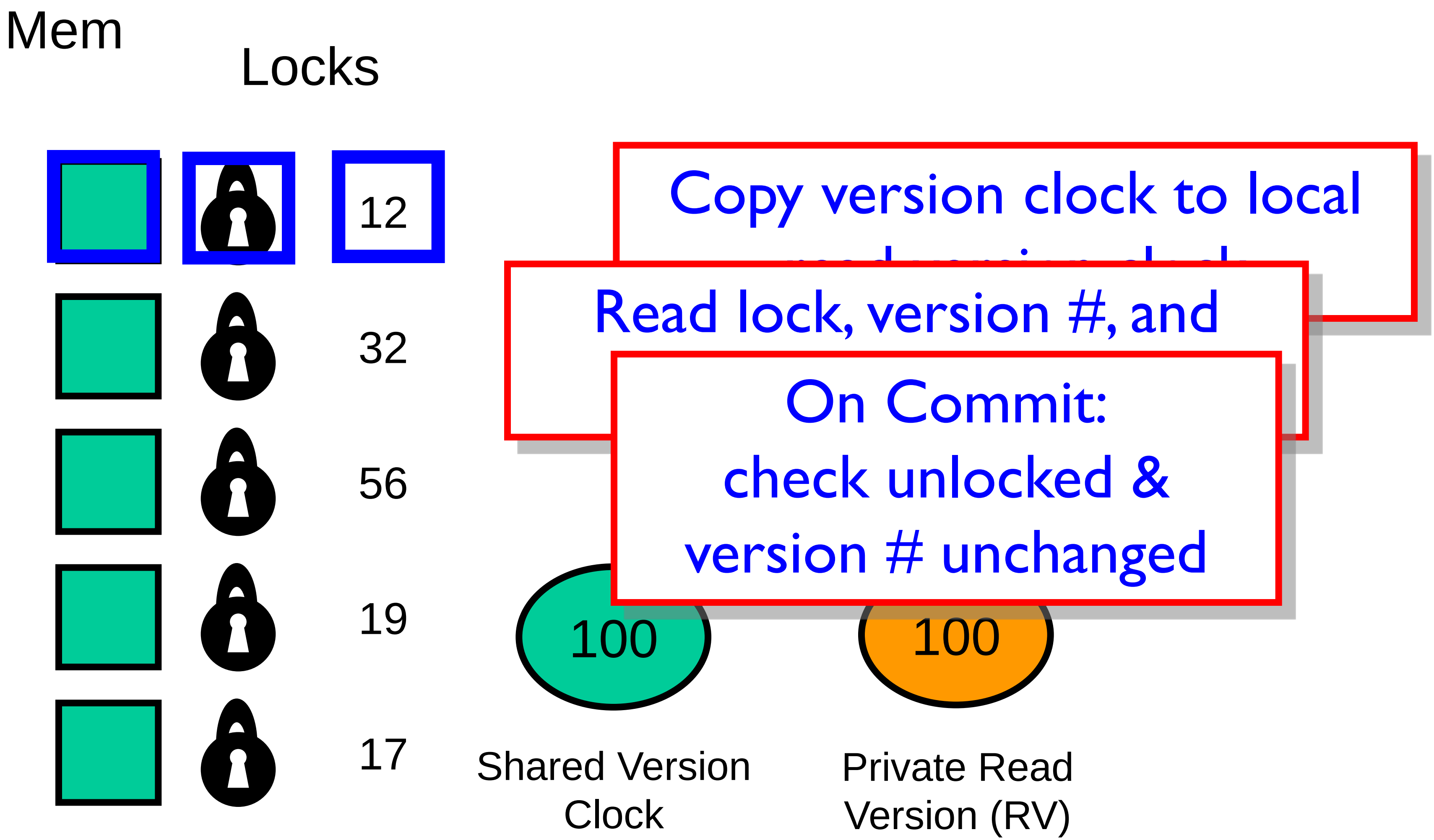
Read-Only Transactions



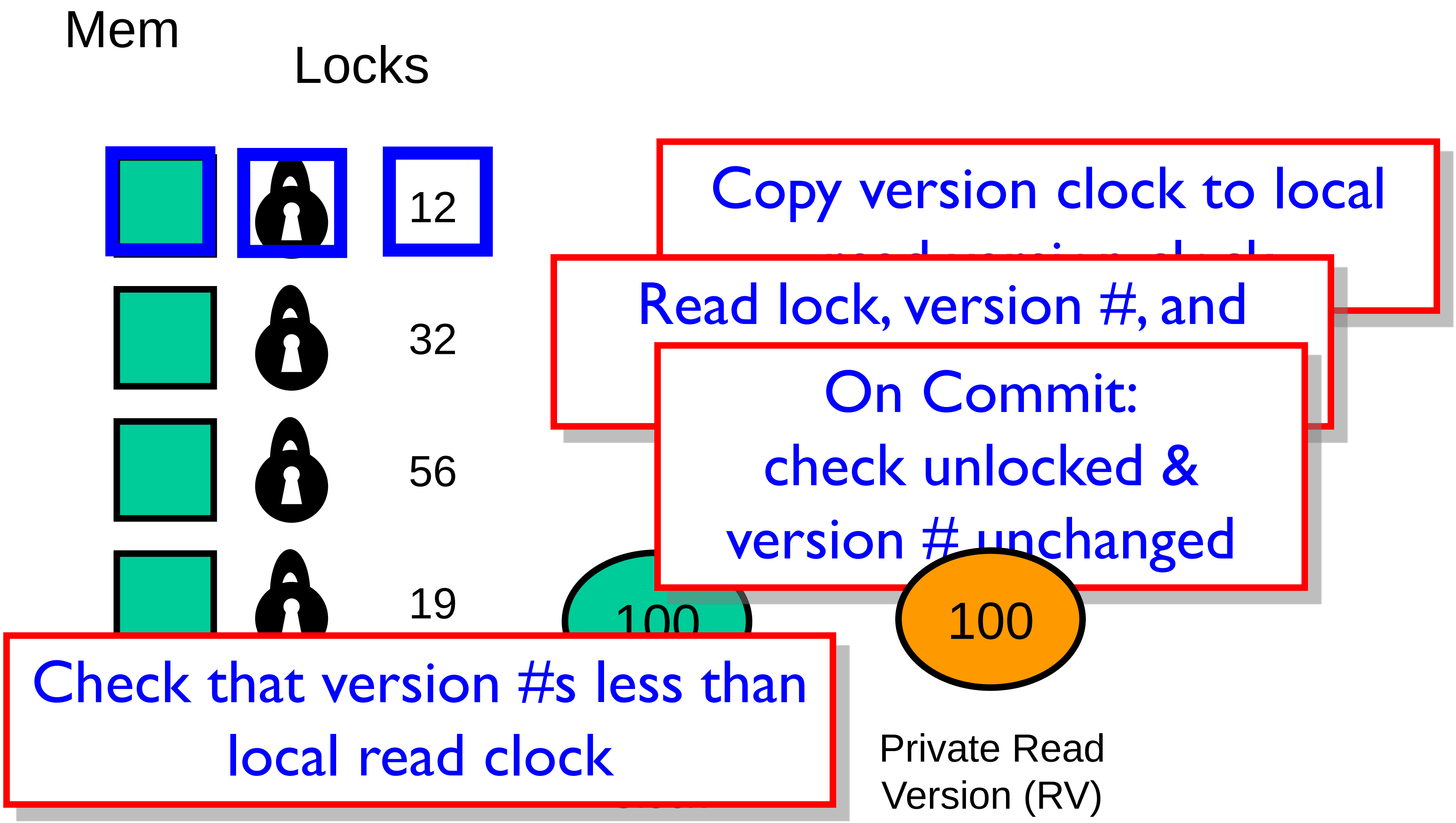
Copy version clock to local

Read lock, version #, and memory

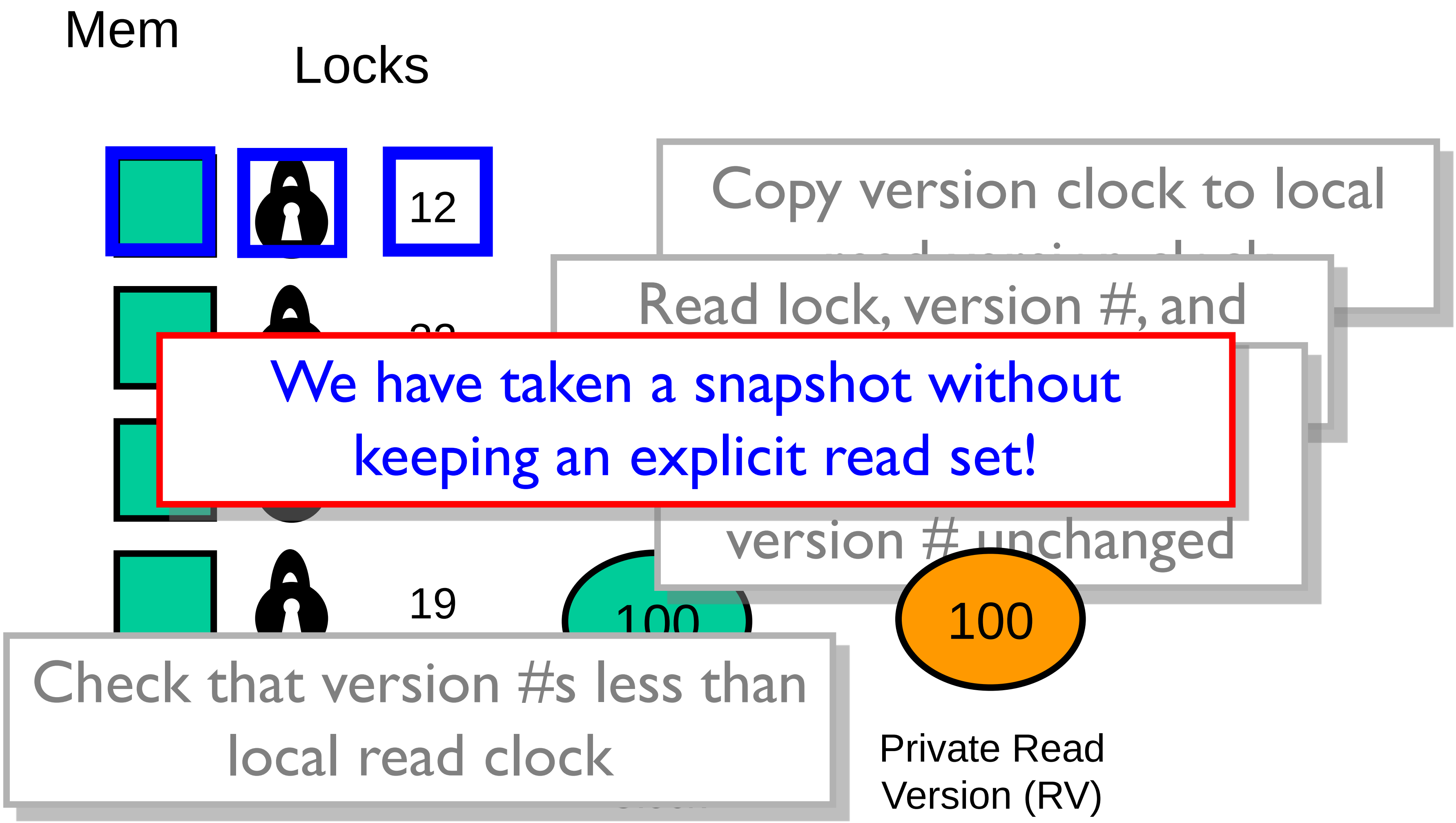
Read-Only Transactions



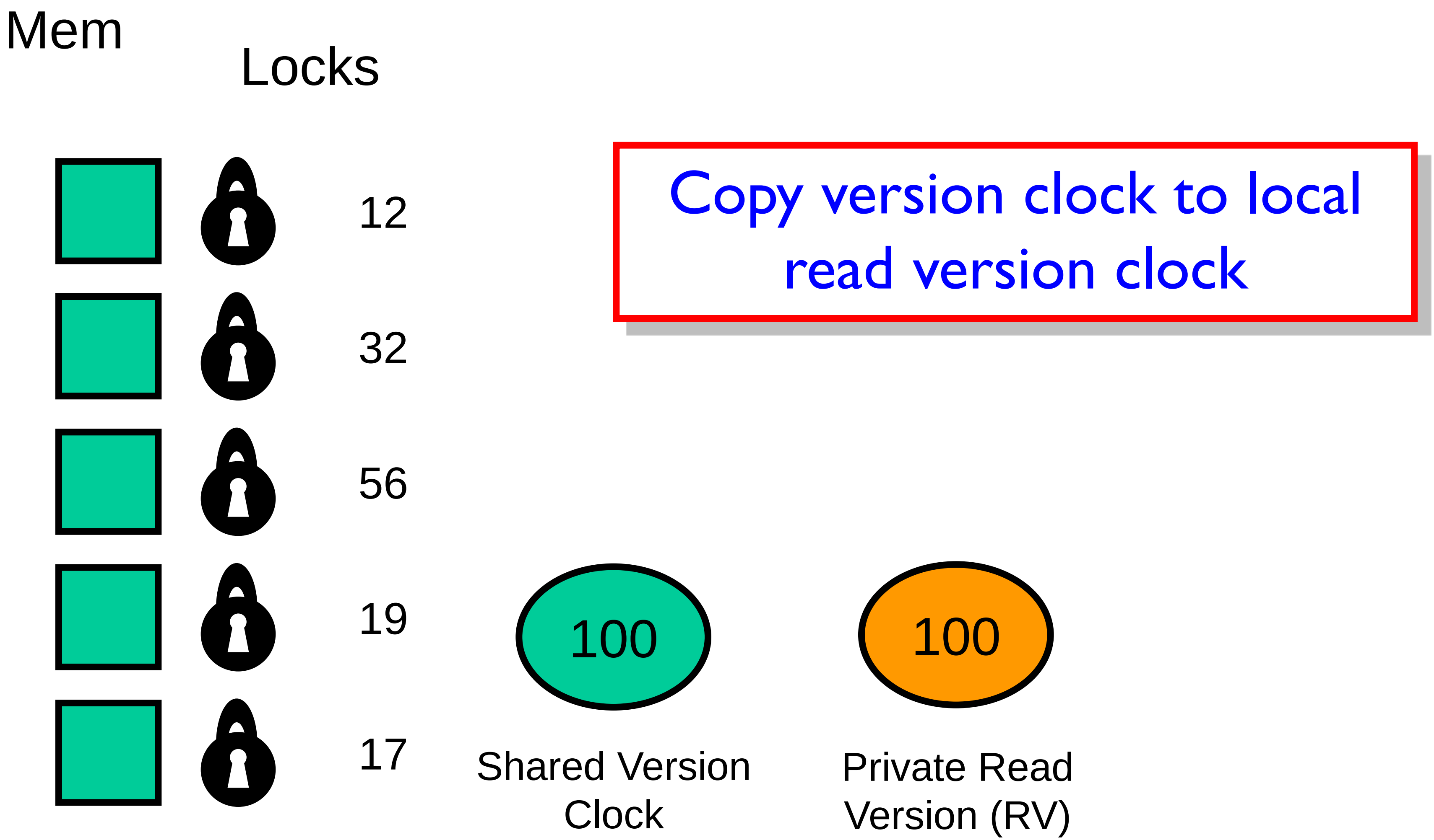
Read-Only Transactions



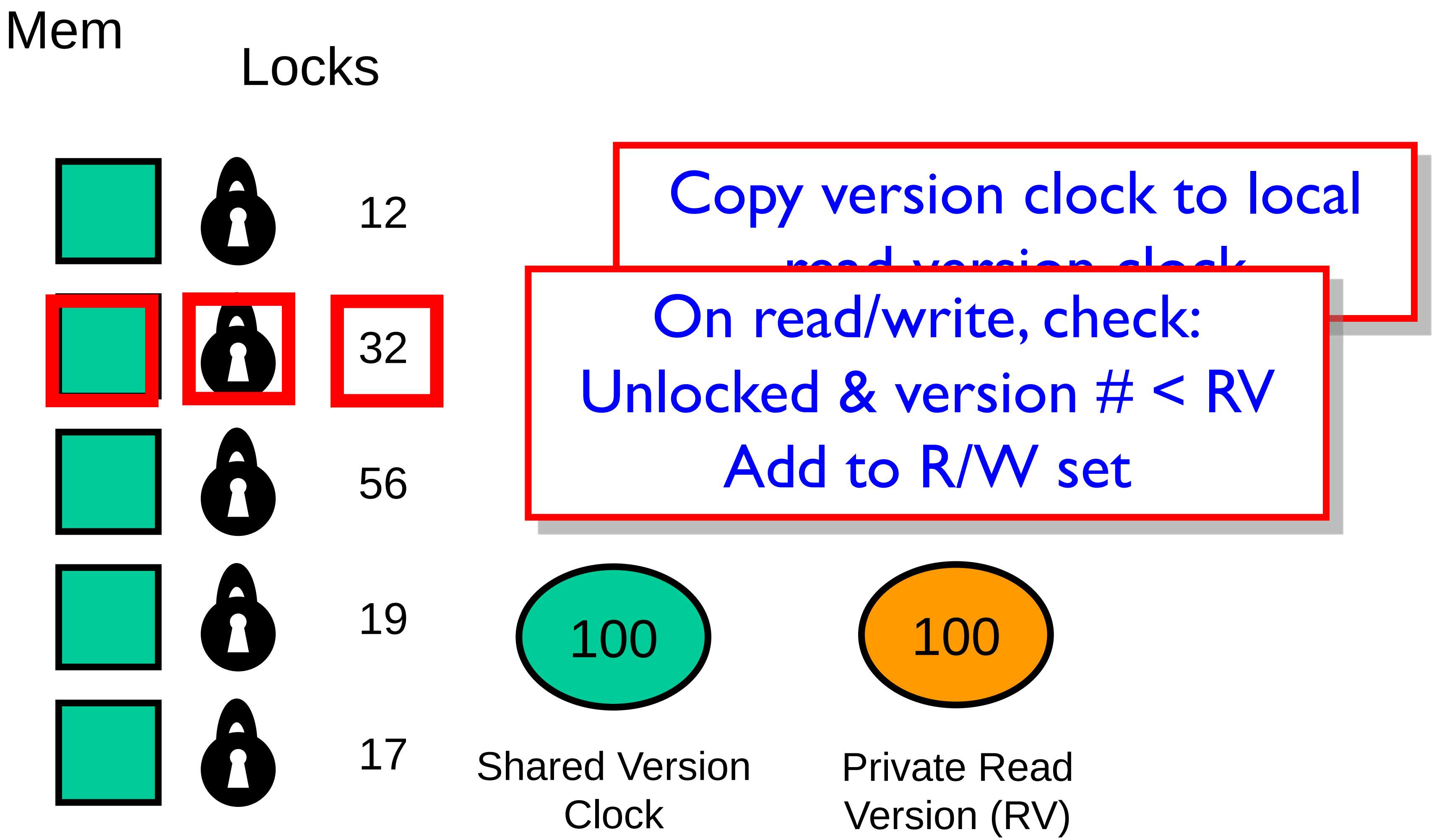
Read-Only Transactions



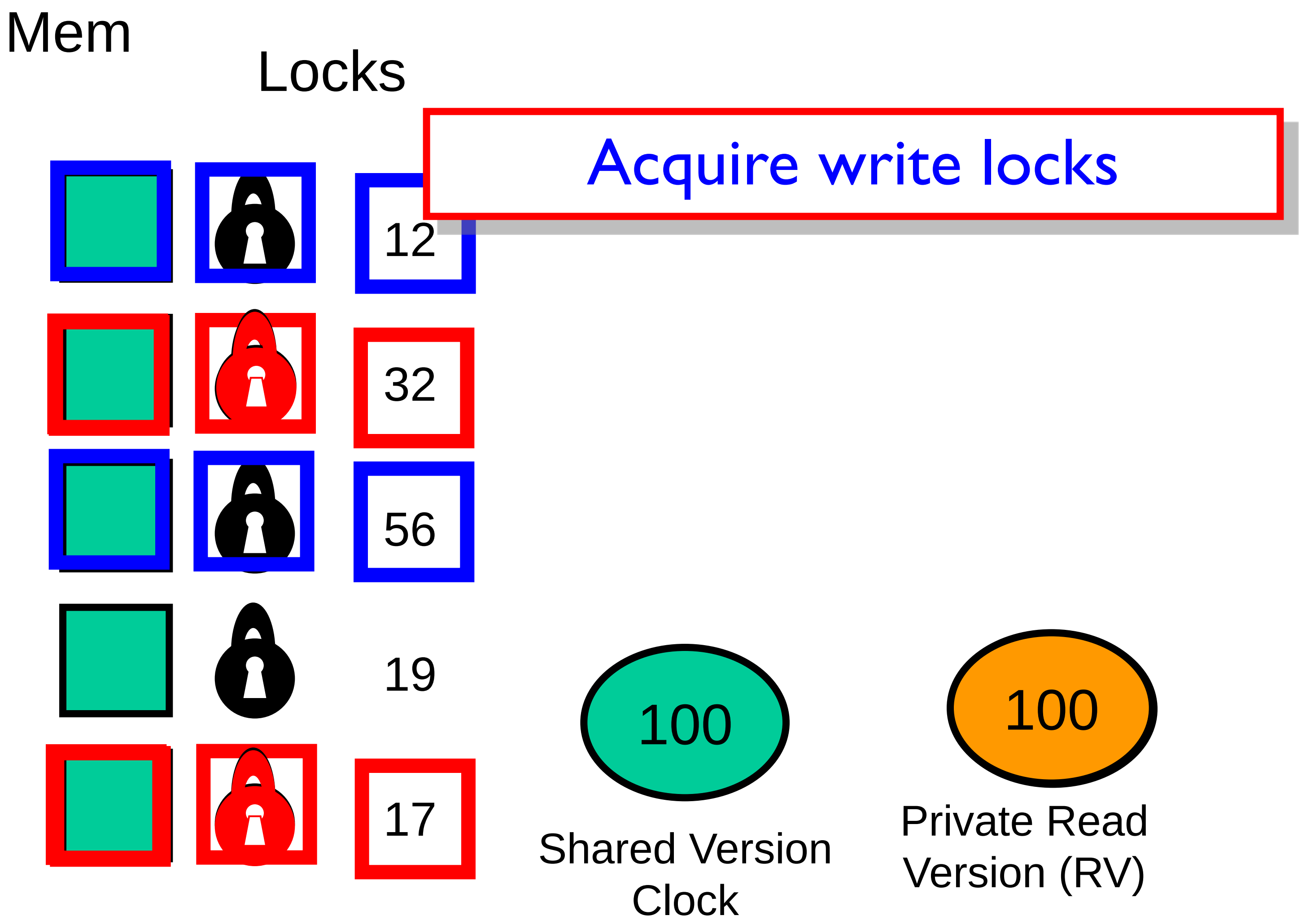
Regular Transactions



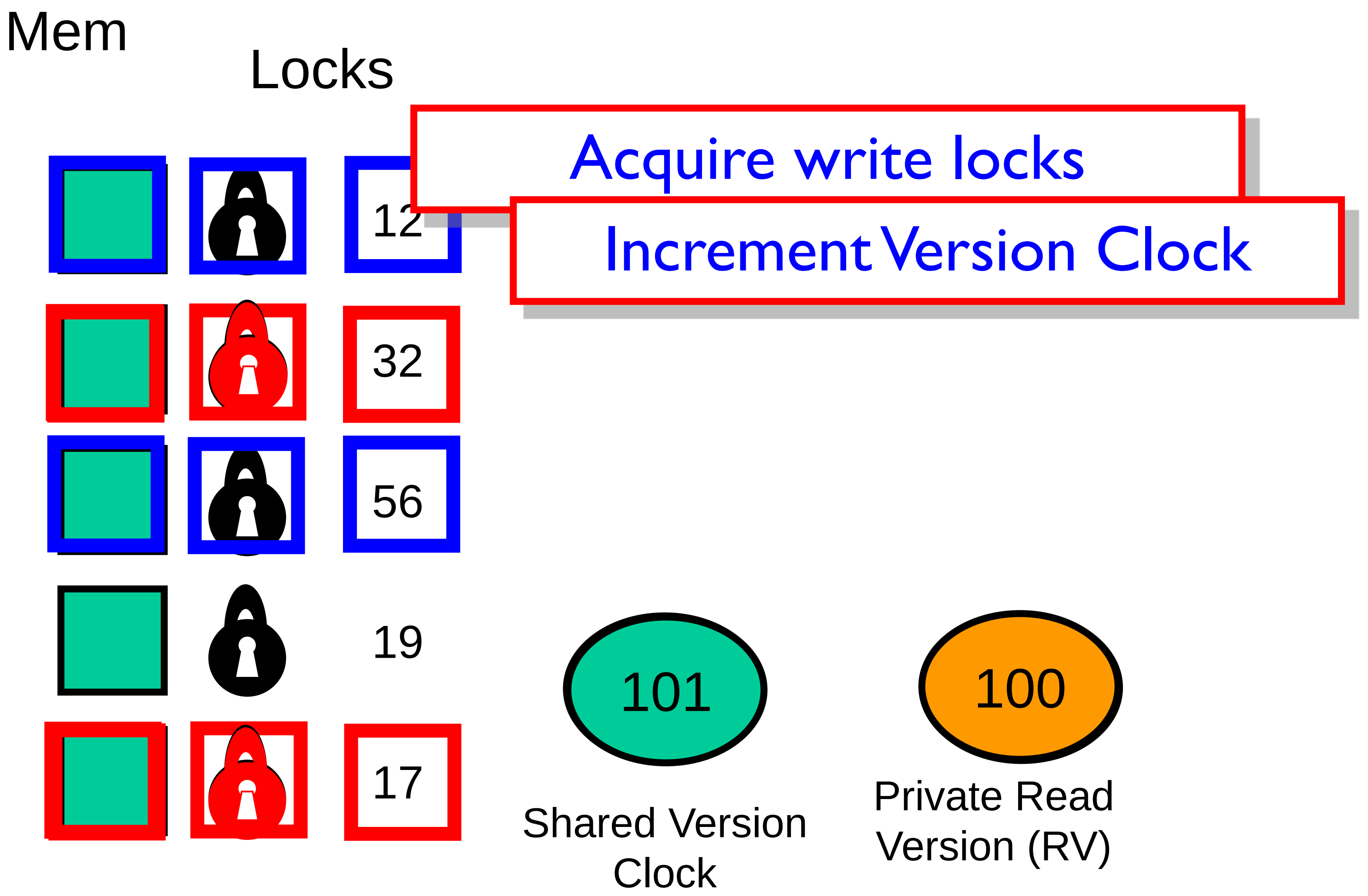
Regular Transactions



On Commit

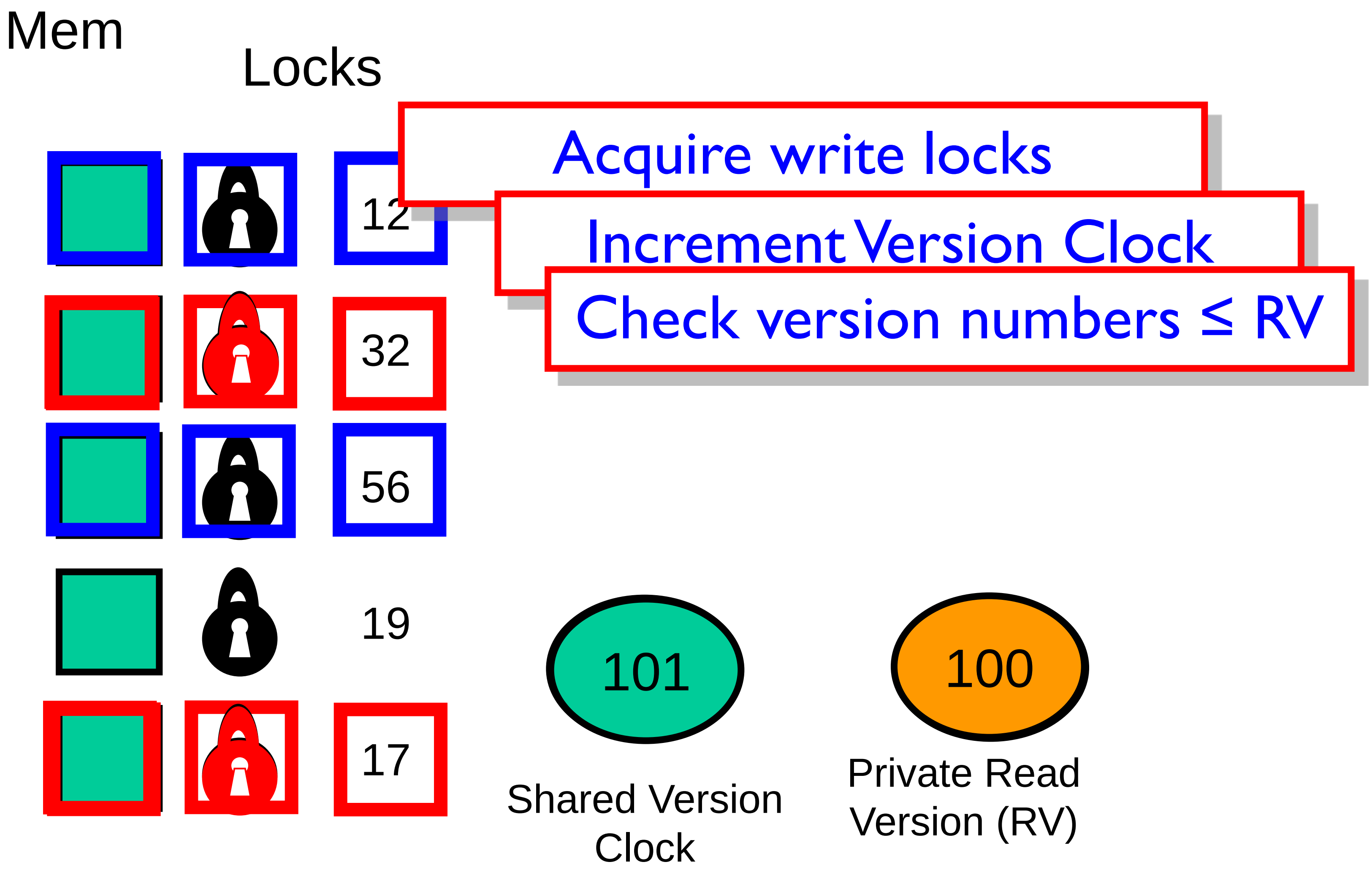

Mem
Locks
Acquire write locks
12
32
56
19
17
100
Shared Version Clock
100
Private Read Version (RV)

On Commit

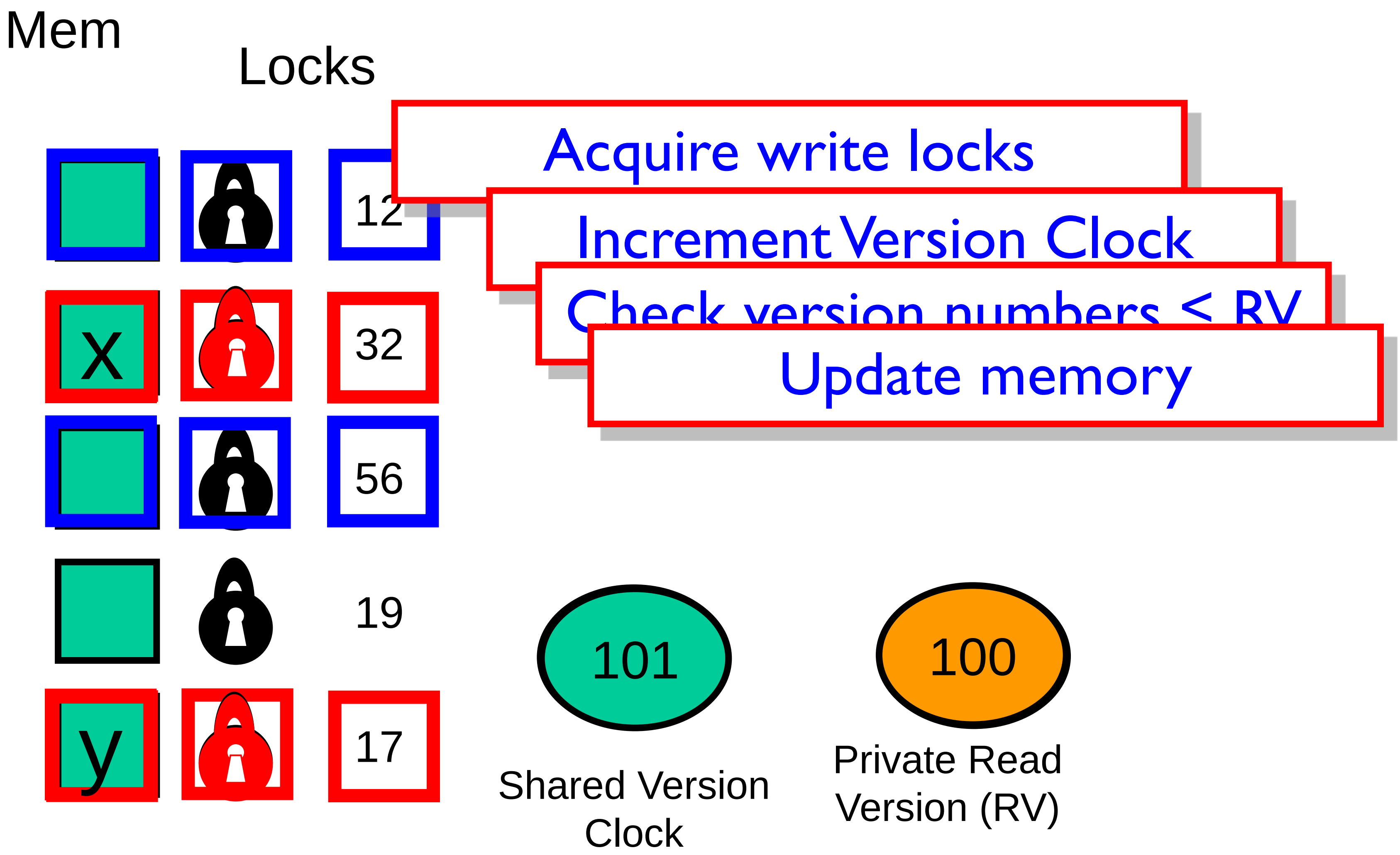


Mem
Locks
12
32
56
19
17
Acquire write locks
Increment Version Clock
101
Shared Version Clock
100
Private Read Version (RV)

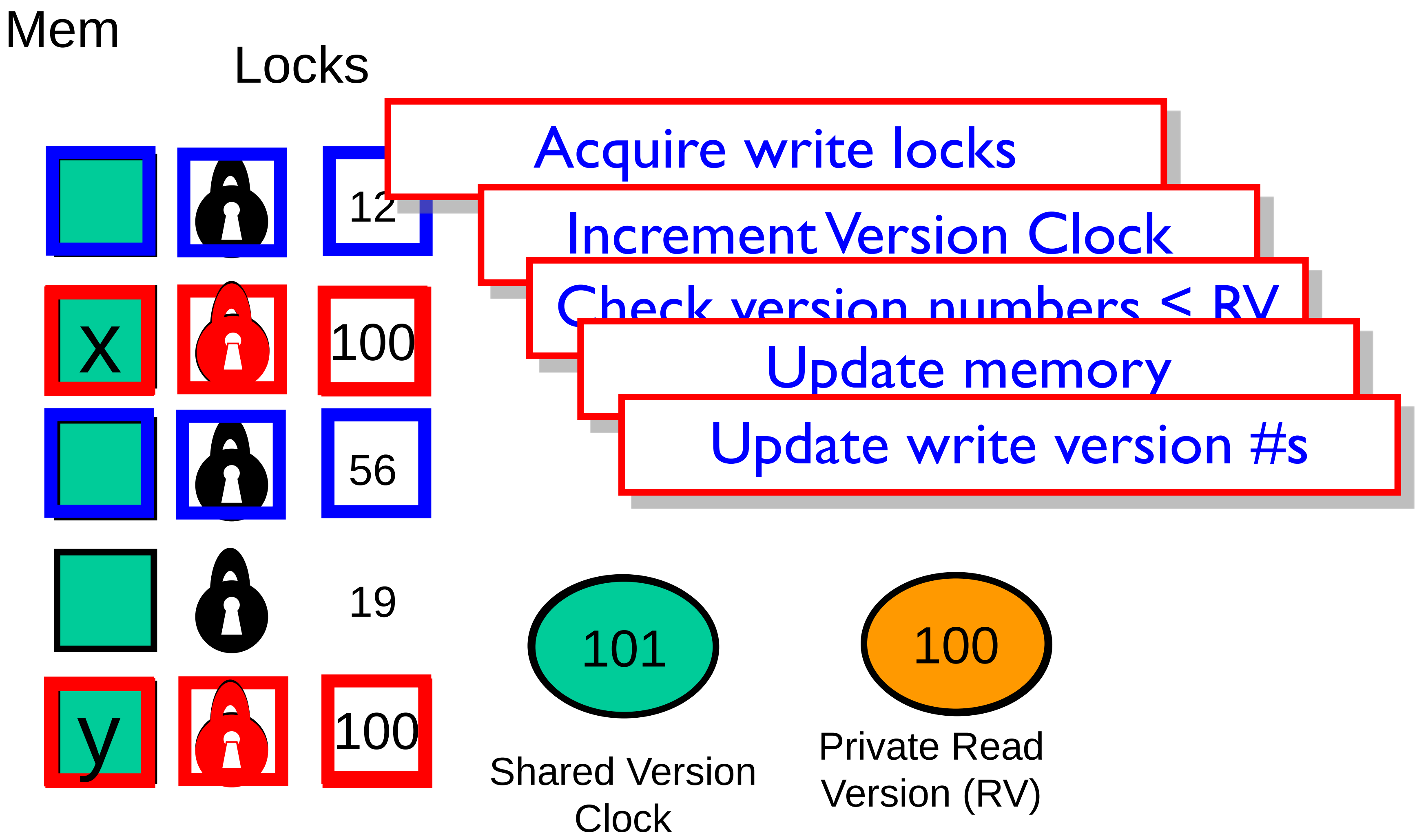
On Commit


Mem
Locks
12
32
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
x
y
12
32
56
19
17
Acquire write locks
Increment Version Clock
Check version numbers ≤ RV
Update memory
101
Shared Version Clock
100
Private Read Version (RV)

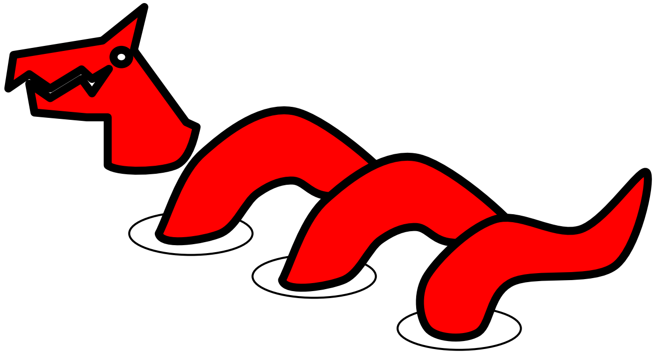
On Commit

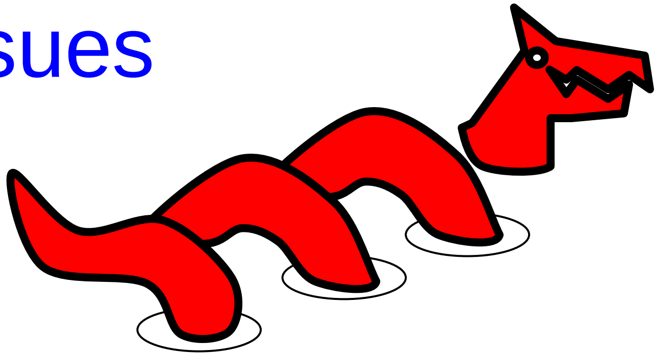


Mem
Locks
12
x
100
56
19
y
100
Acquire write locks
Increment Version Clock
Check version numbers ≤ RV
Update memory
Update write version #s
101
Shared Version Clock
100
Private Read Version (RV)

TM Design Issues

- Implementation choices
- Language design issues
- Semantic issues





Granularity

- Object
  - managed languages, Java, C#, Scala, …
  - 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

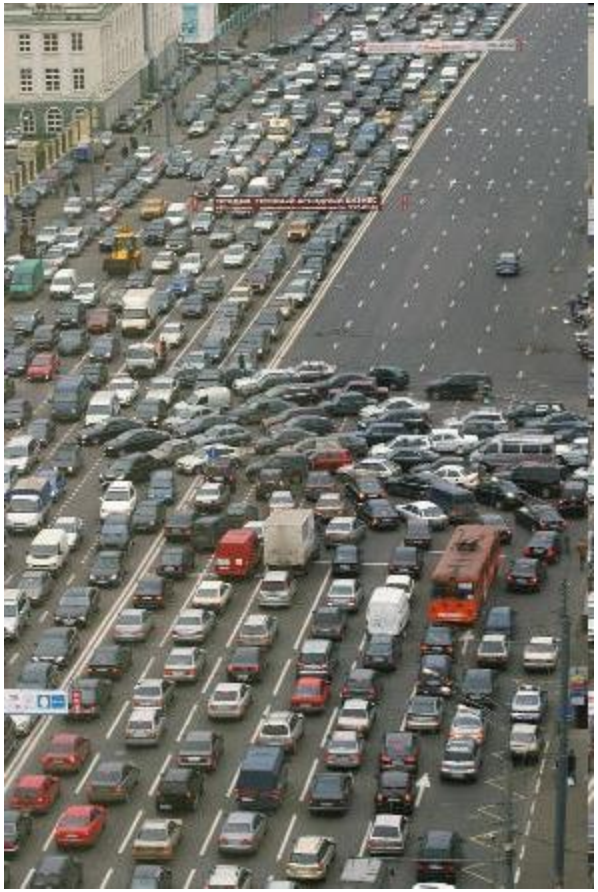
- Eager
  - Detect before conflict arises
  - "Contention manager" module resolves
- Lazy
  - Detect on commit/abort
- Mixed
  - Eager write/write, lazy read/write …

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

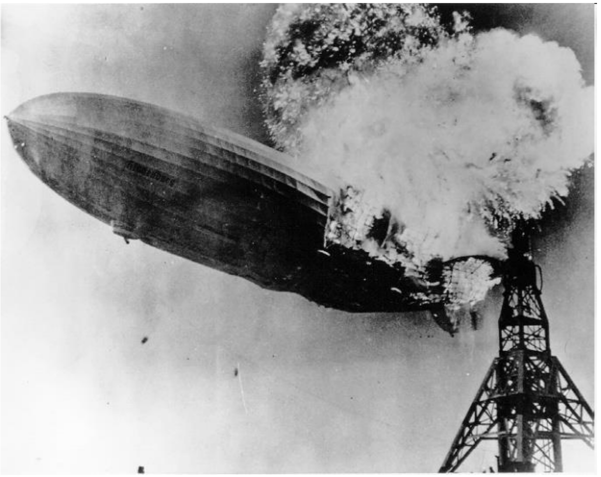
Throws OutOfMemoryException!

```
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