

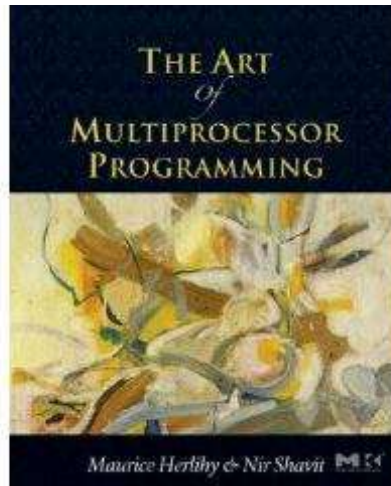
NON-BLOCKING DATA STRUCTURES AND TRANSACTIONAL MEMORY

Tim Harris, 17 Oct 2018

Lecture 3/3

- Problems with locks
- Atomic blocks and composition
- Hardware transactional memory
- Software transactional memory

Transactional Memory



Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Our Vision for the Future

In this course, we covered

Best practices ...

New and clever ideas ...

And common-sense observations.



Our Vision for the Future

In this course we covered
Nevertheless ...

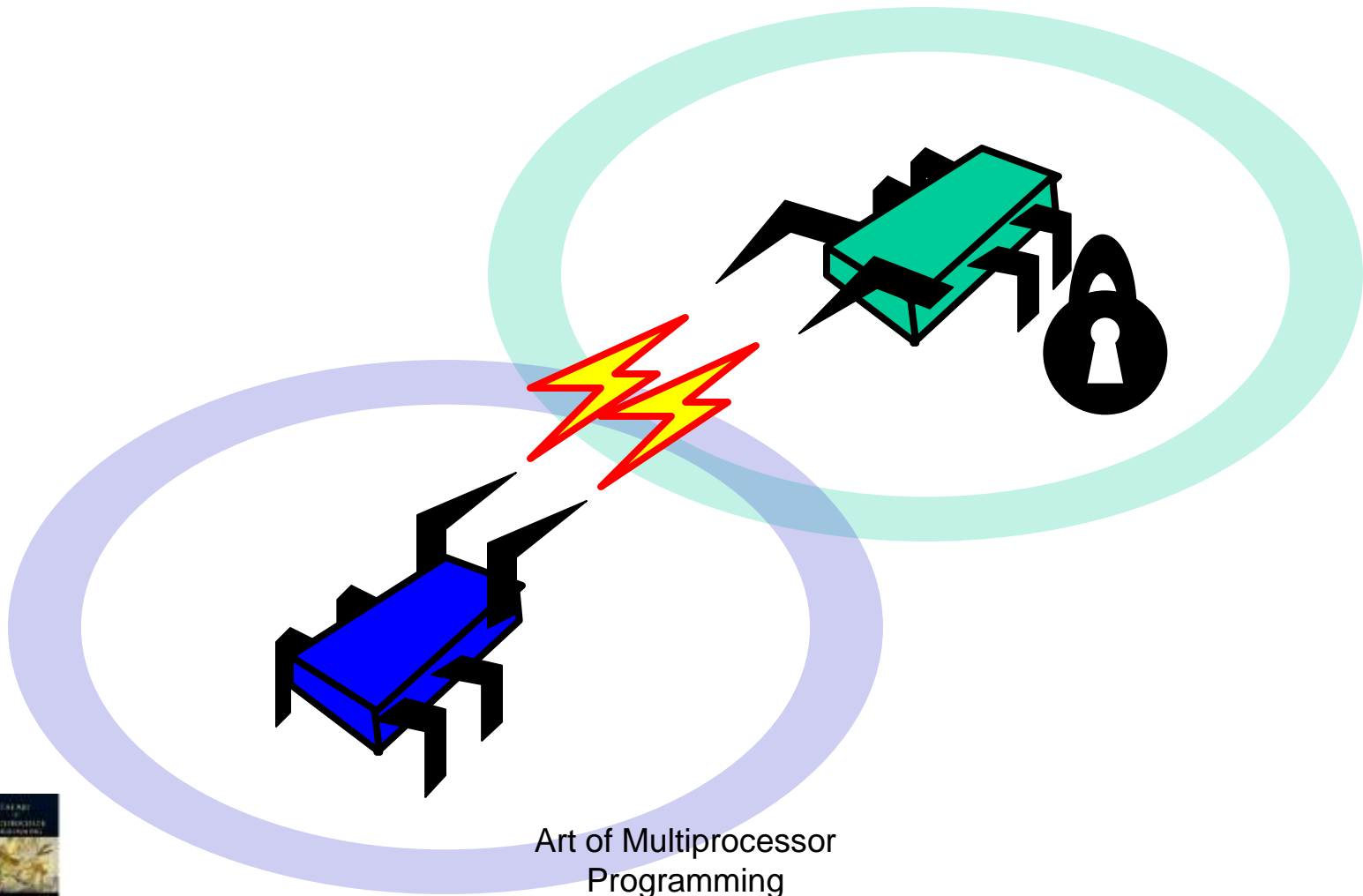
Concurrent programming is still too hard ...

Here we explore why this is

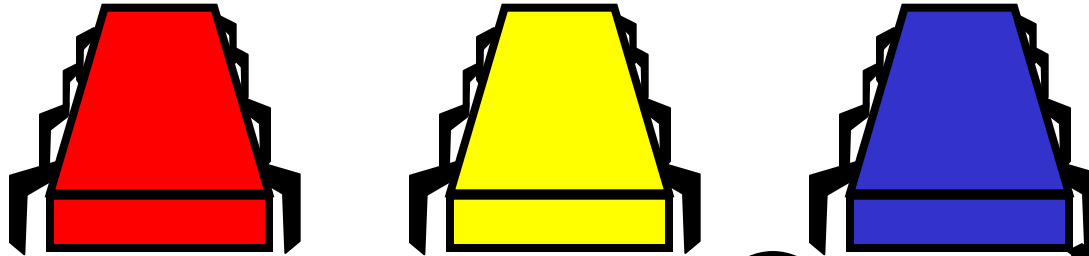
And what we can do about it.



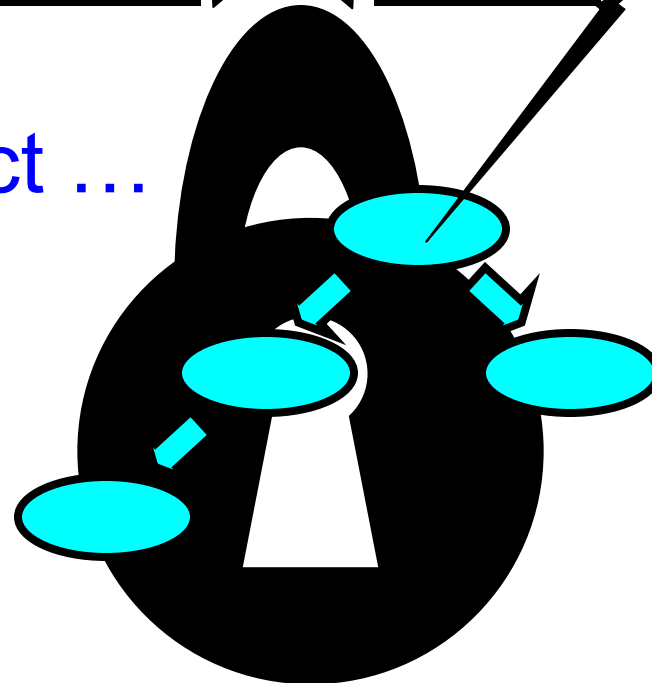
Locking



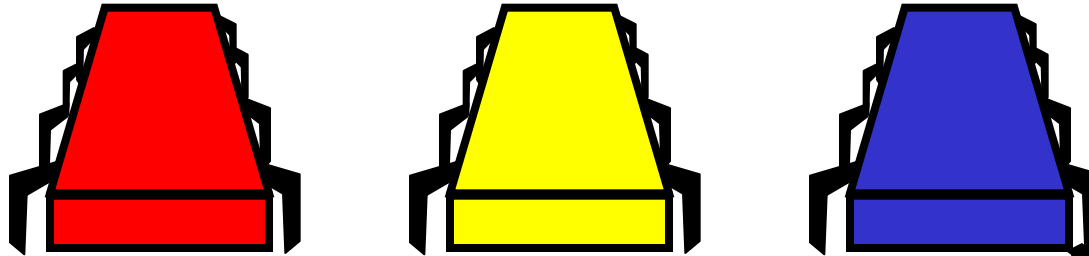
Coarse-Grained Locking



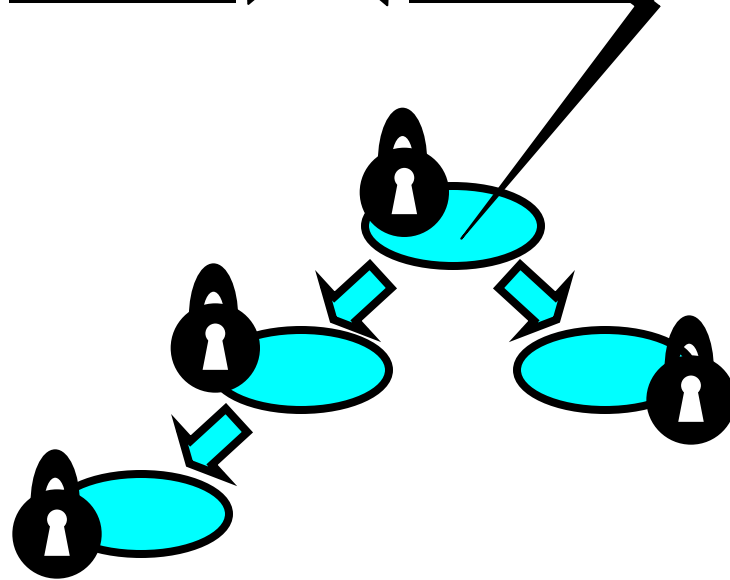
Easily made correct ...
But not scalable.



Fine-Grained Locking

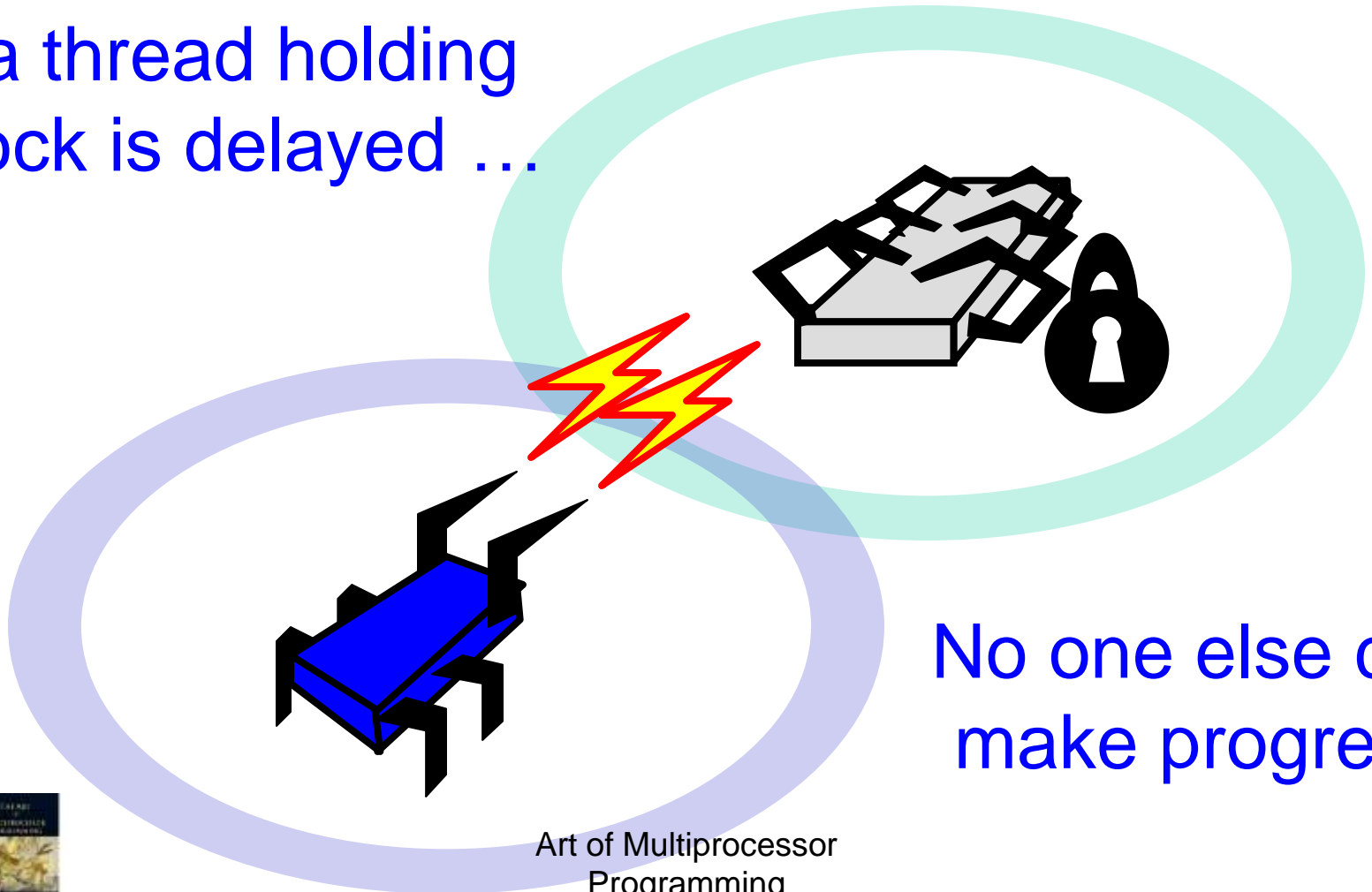


Can be tricky ...



Locks are not Robust

If a thread holding a lock is delayed ...



No one else can make progress



Locking Relies on Conventions

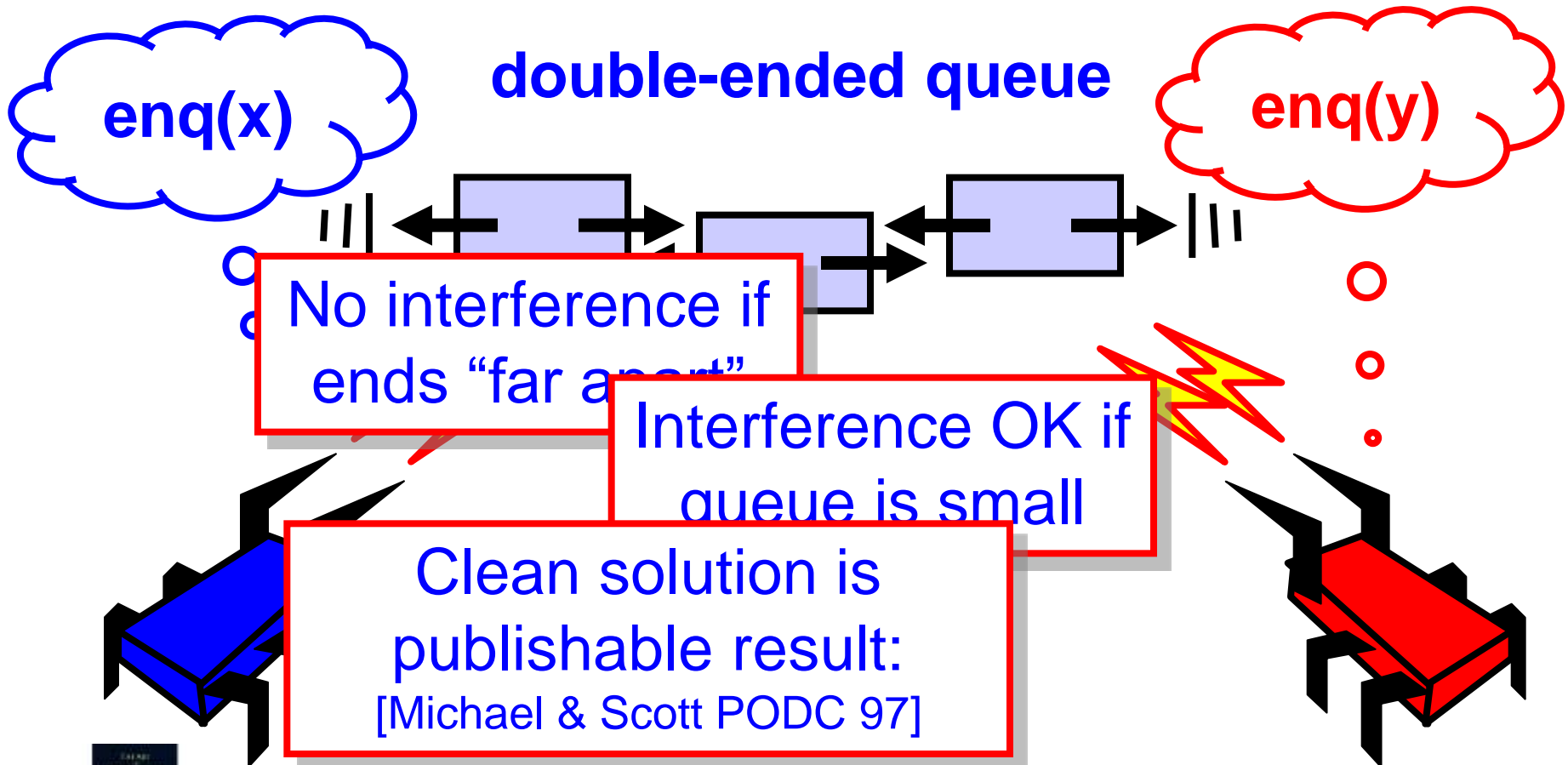
- Relation between
 - Locks and objects
 - Exists only in programmer's

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

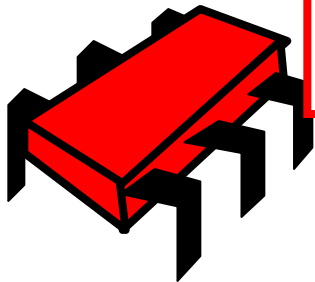
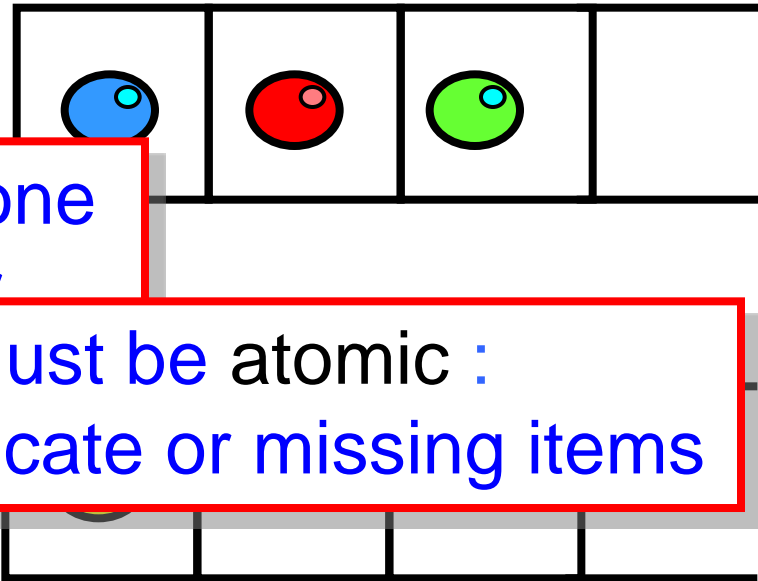
```
/*  
 * When a locked buffer is visible to the I/O layer  
 * BH_Laundry is set. This means before unlocking  
 * we must clear BH_Laundry, mb() on alpha and then  
 * clear BH_Lock, so no reader can see BH_Laundry set  
 * on an unlocked buffer and then risk to deadlock.  
 */
```



Simple Problems are hard



Locks Not Composable



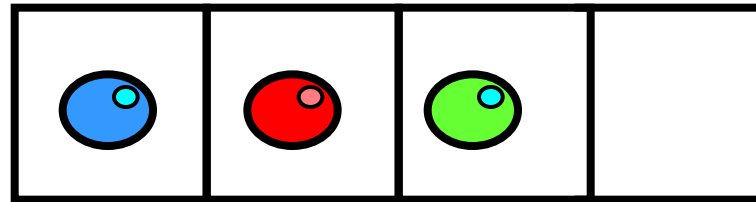
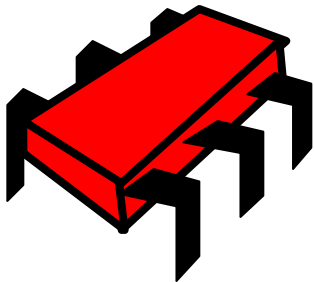
Locks Not Composable



Lock source

Unlock source
& target

Lock target



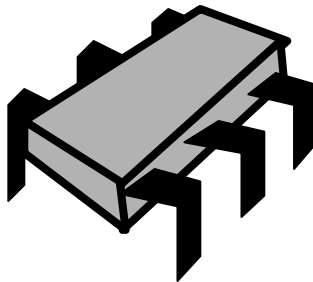
Locks Not Composable

Methods cannot provide
internal synchronization

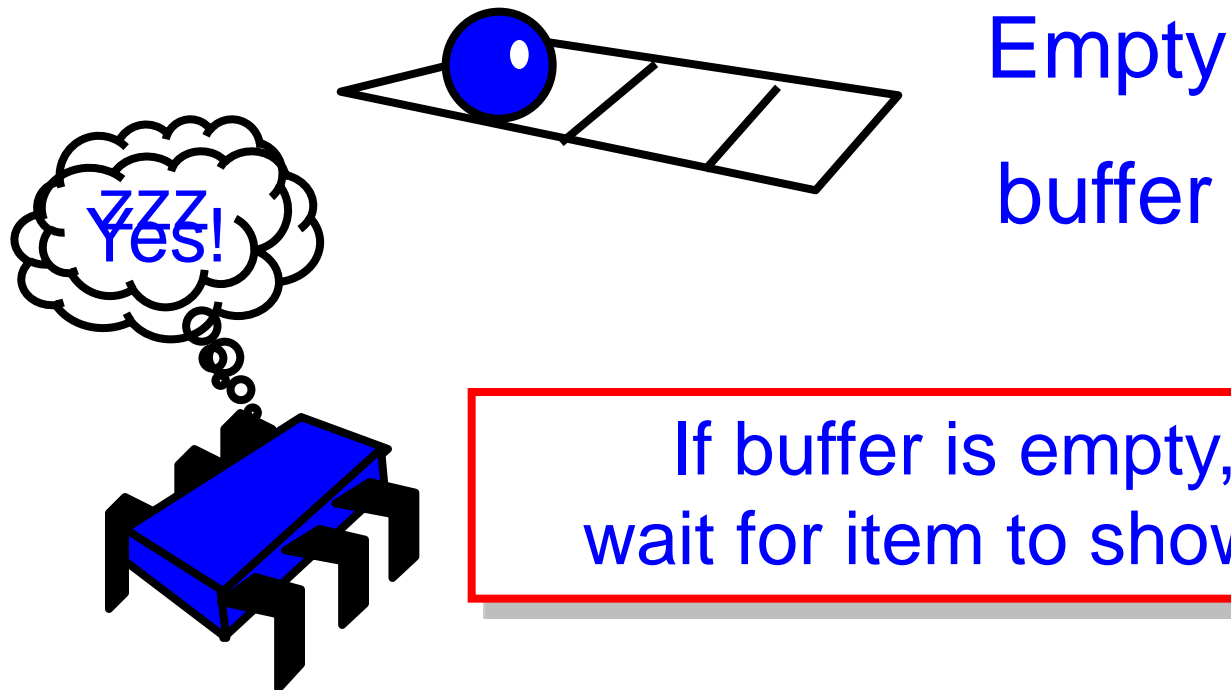
Objects must expose
locking protocols to clients

Clients must devise and
follow protocols

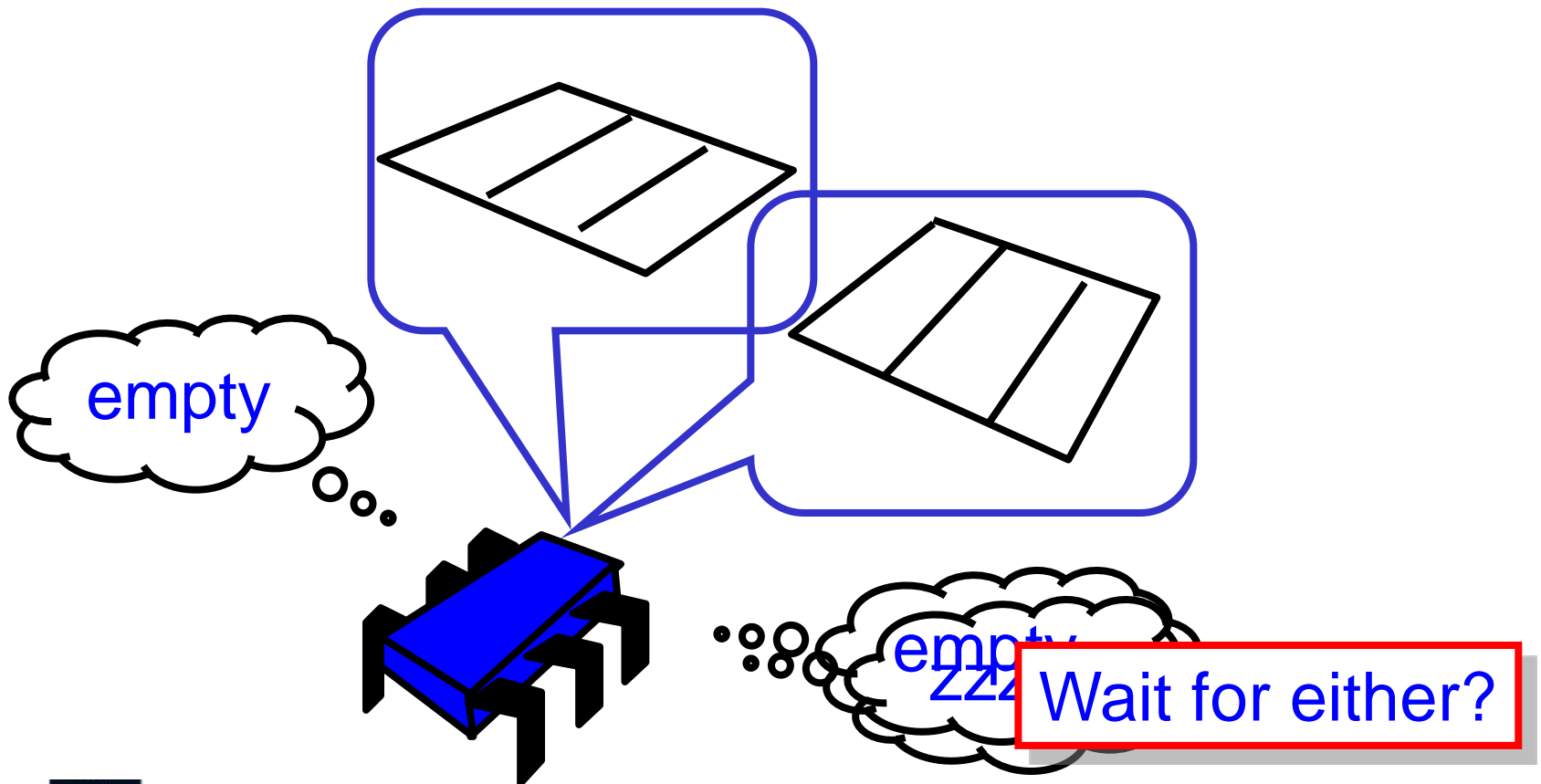
Abstraction broken!



Monitor Wait and Signal



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-time systems



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);  
}
```

y.add(3);

```
atomic {  
  y = null;  
}
```

y = null;

No data race



A Double-Ended Queue

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

Write sequential Code



A Double-Ended Queue

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



A Double-Ended Queue

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

Enclose in atomic block

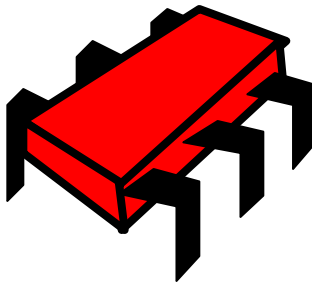
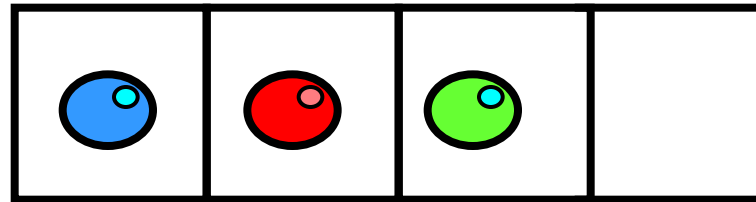


Warning

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



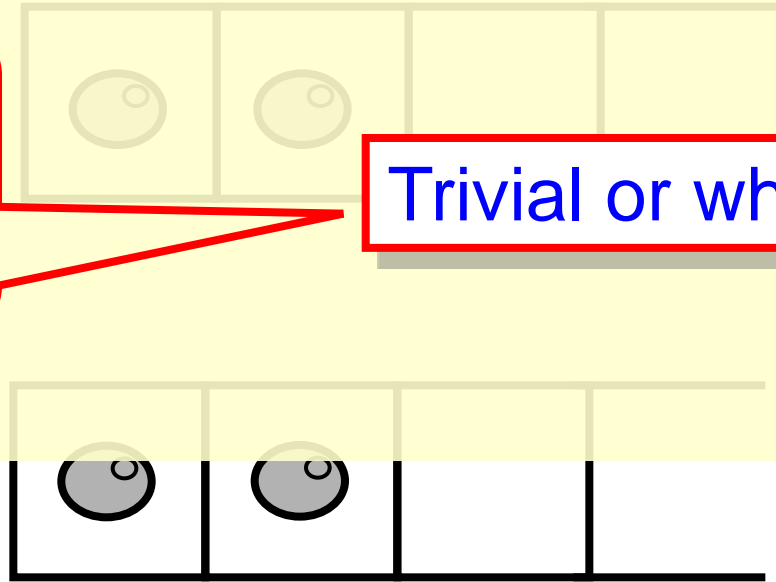
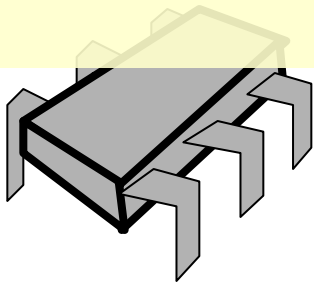
Composition?



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 (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

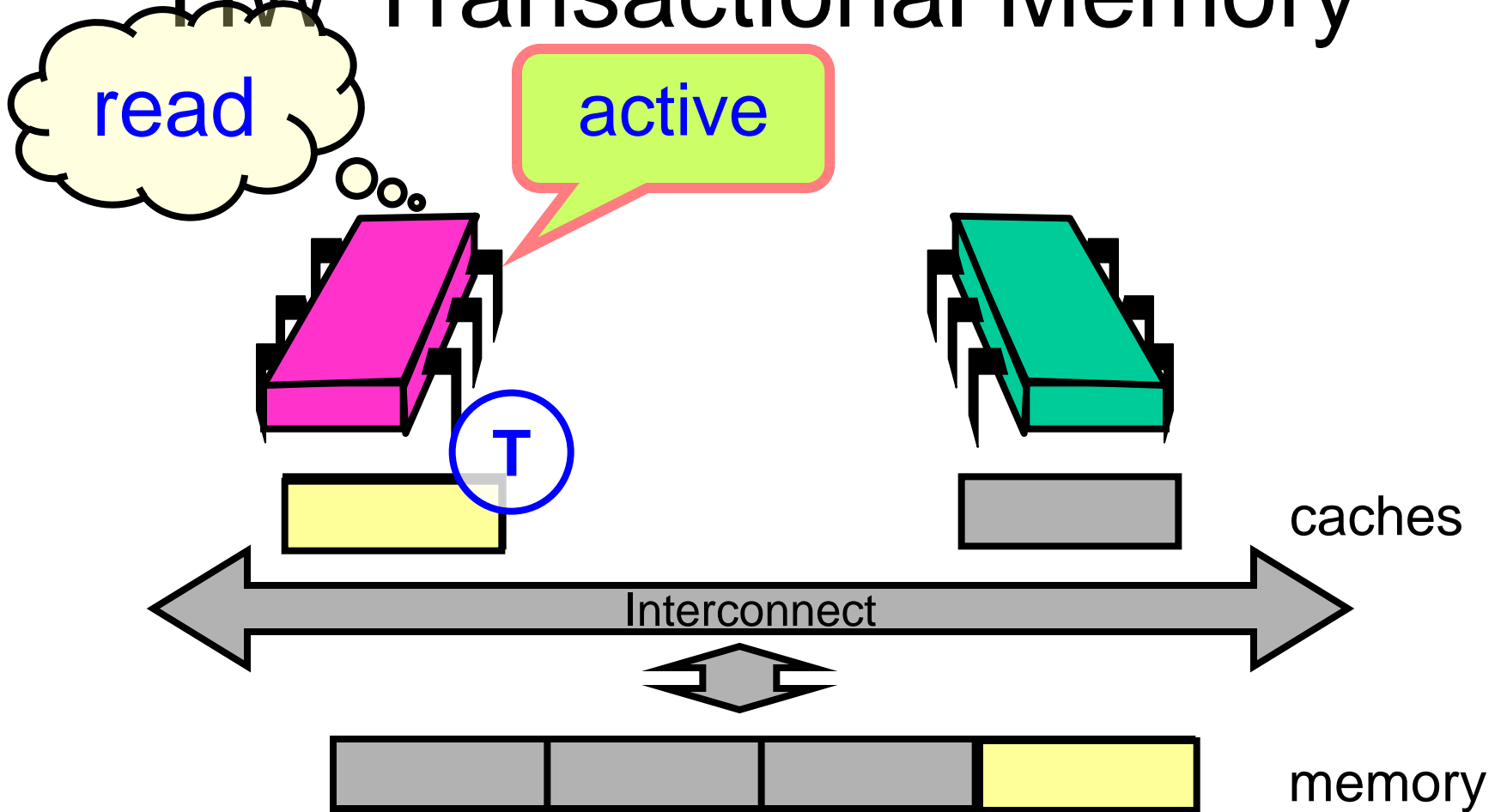


Hardware Transactional Memory

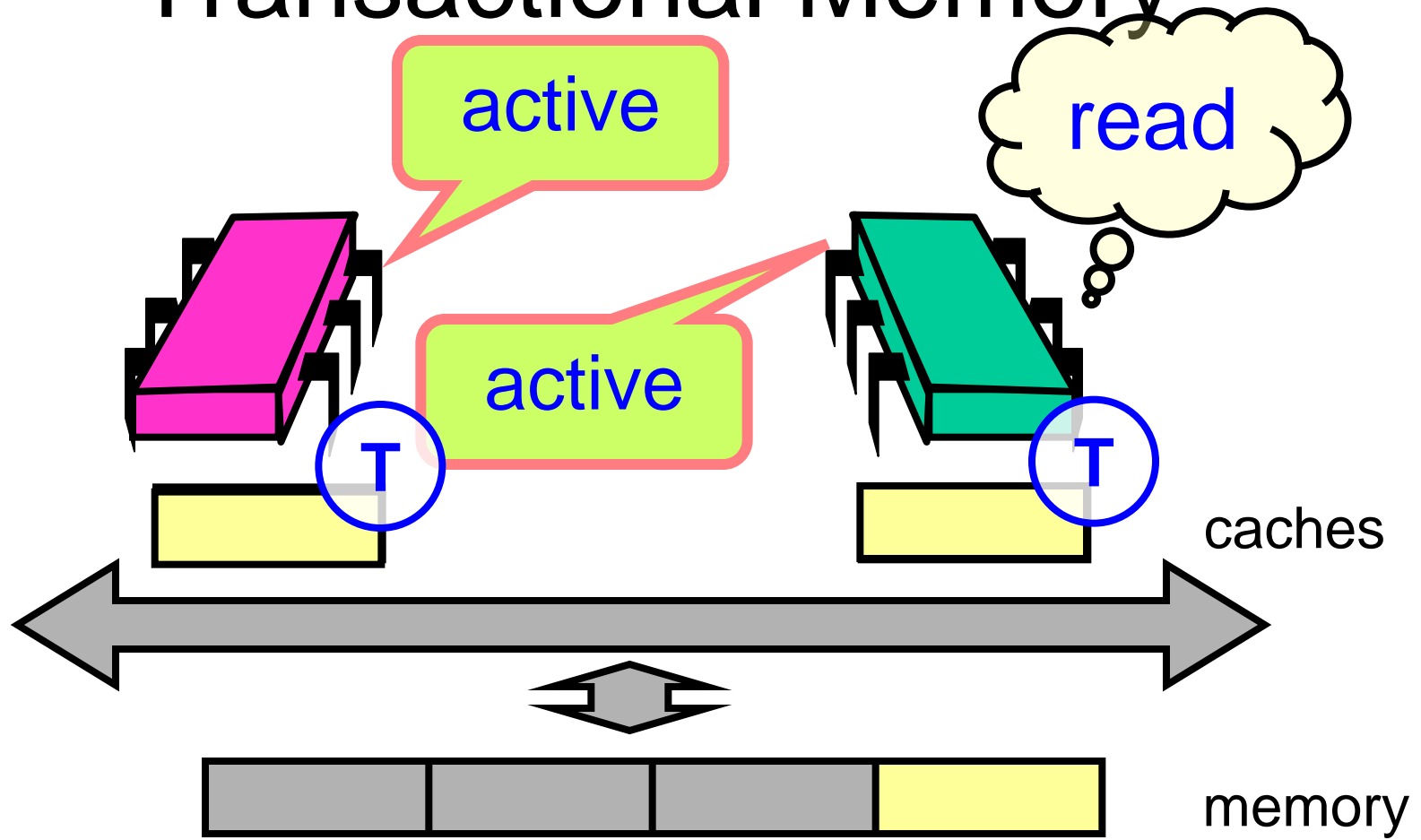
- Exploit Cache coherence
- Already almost does it
 - Invalidation
 - Consistency checking
- Speculative execution
 - Branch prediction = optimistic synch!



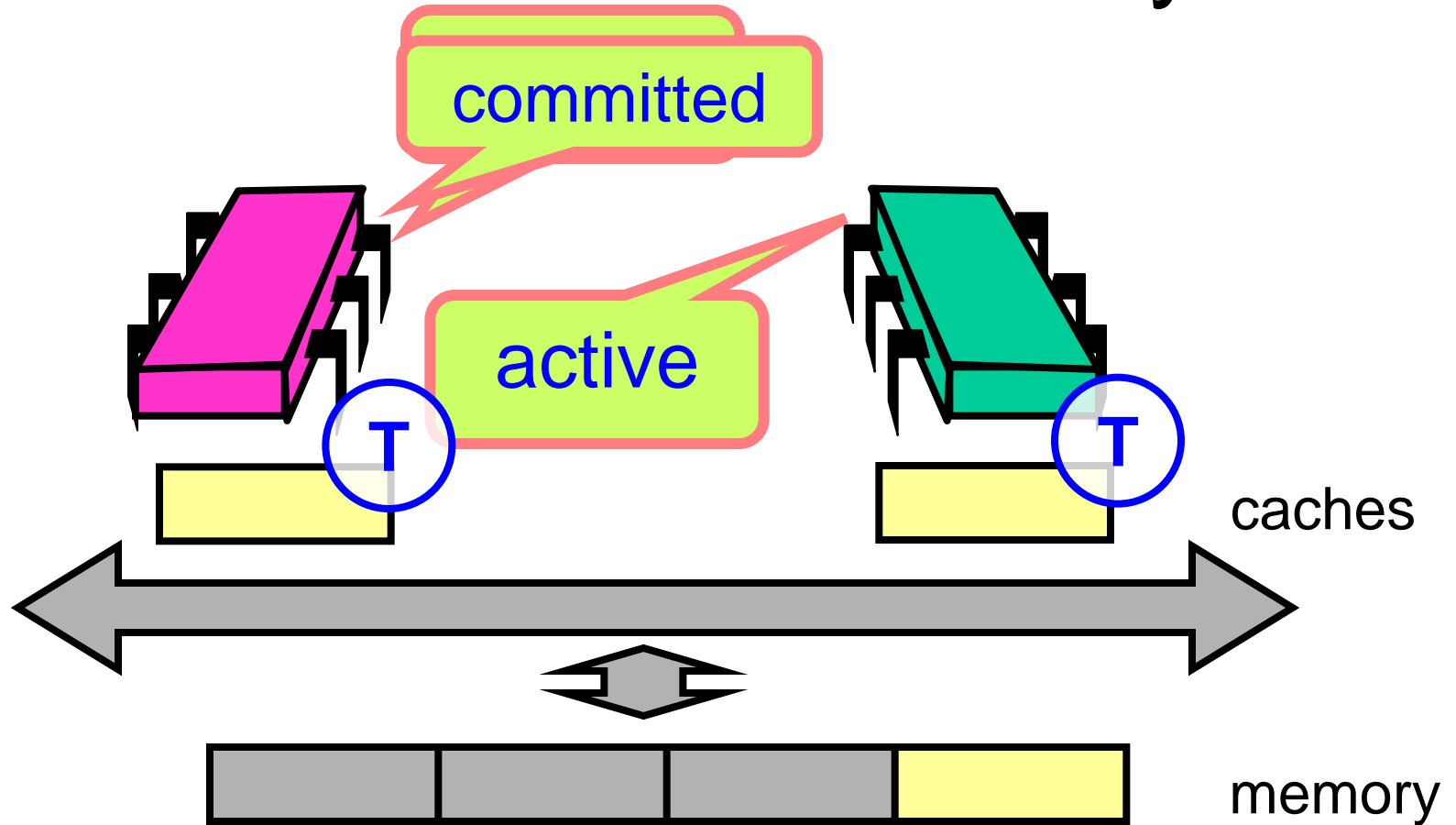
HW Transactional Memory



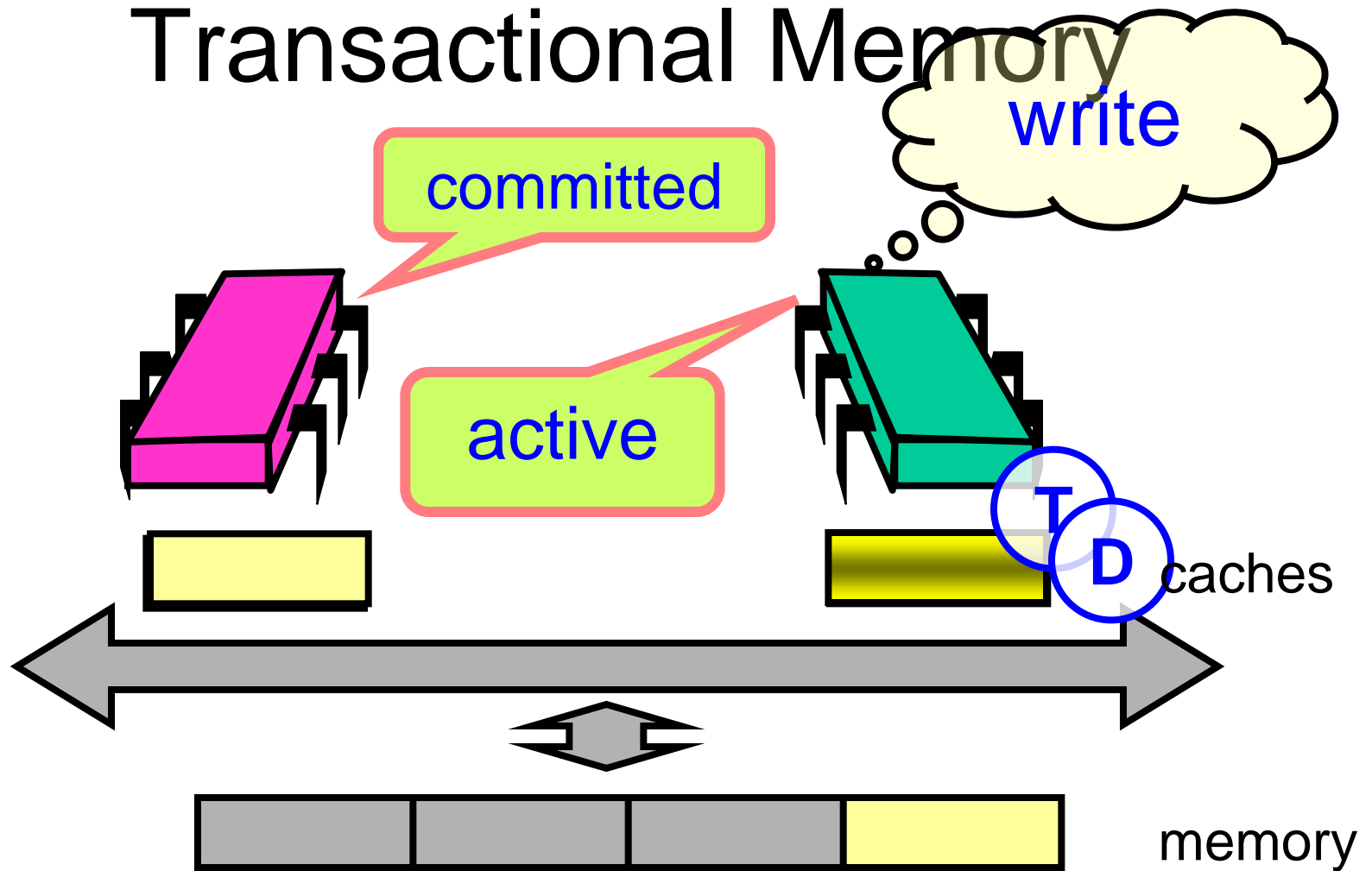
Transactional Memory

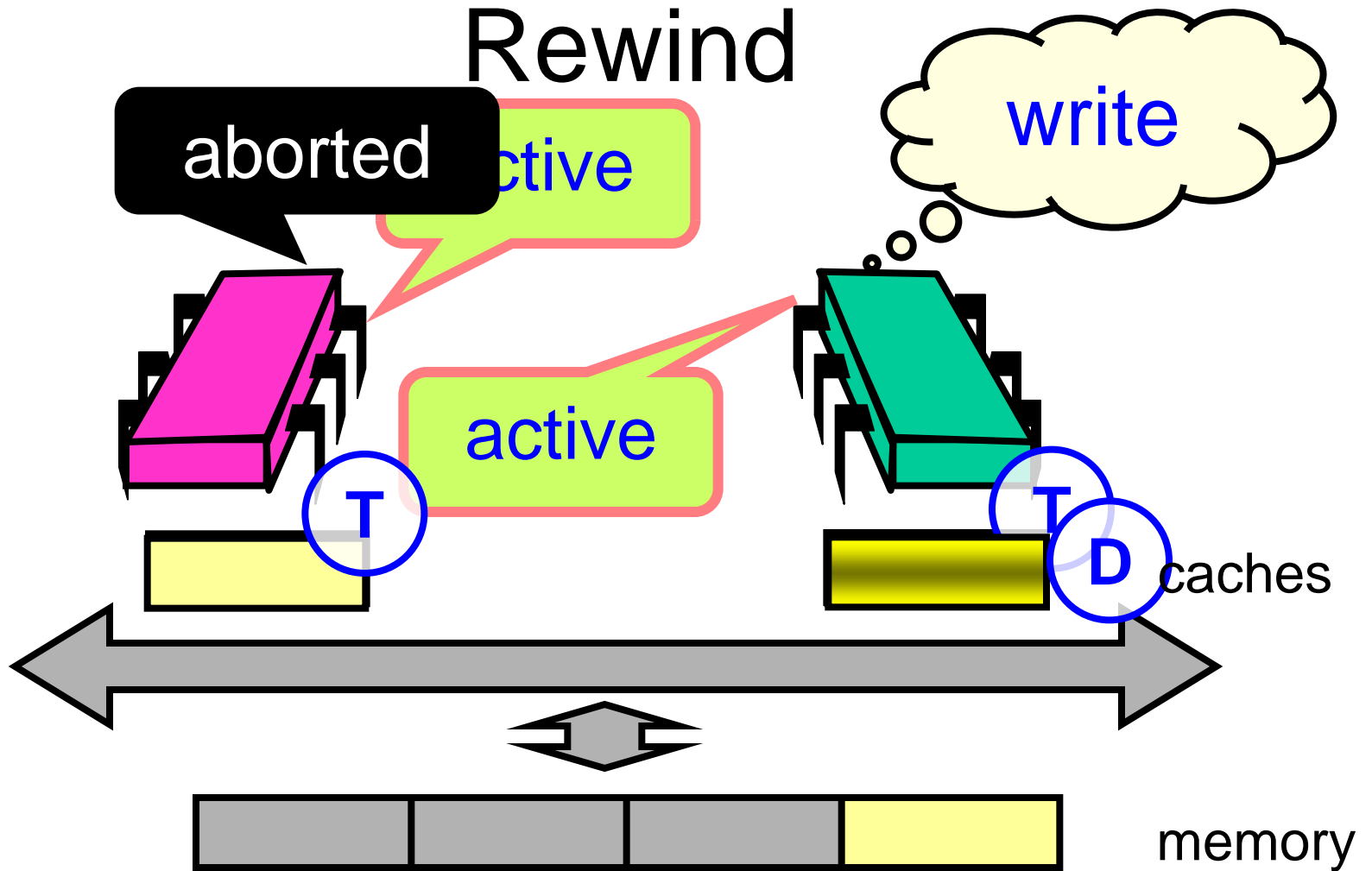


Transactional Memory



Transactional Memory





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
 - Try again in software?



Composition

Locks don't compose, transactions do.

Composition necessary for Software Engineering.

But practical HTM doesn't *really* support composition!

Why we need STM



Composability in Transactions
publication version 2006
Simon Peyton Jones
Cambridge
[email address]
Herlihy
interfaces and it did not
block. In this
make the
in the setting
more than a
can be ex-
and we
stronger guaran-
more transactions
together to
which apper
retry (S

Note: this post-
publication cited in the
been fixed, and an Appendix

Abstract
Writing concurrent programs is notoriously difficult.
creasing practical importance. A particular source of con-
even correctly-implemented concurrency abstractions cannot
posed together to form larger abstractions. In this paper we
ncurrency model, based on *transactional memory*
composition. All the usual benefits of trans-
composition (e.g. freedom from deadlock), but in
forms of blocking and choice

vent threads from
provide a convincing story
er we resolve these shortc
contributions:
the ideas of transactional me
11 (Section 3). This is much
setting. As we show, STM can be ex-
declarative language, and we
ive far stronger guaran-
more transactions
together to

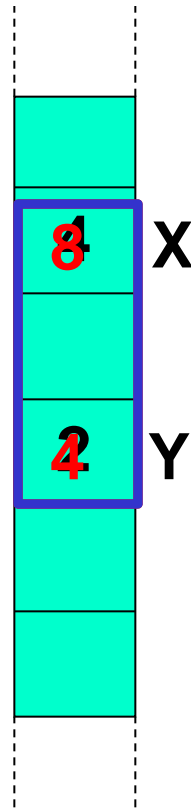
We
to the pro

Transactional Consistency

- Memory Transactions are collections of reads and writes executed atomically
- They should maintain *consistency*
 - *External*: with respect to the interleavings of other transactions (*linearizability*).
 - *Internal*: the transaction itself should operate on a consistent state.



External Consistency



Invariant $x = 2y$

Transaction A:

Write x

Write y

Transaction B:

Read x

Read y

Compute $z = 1/(x-y) = 1/2$

Application
Memory



A Simple Lock-Based STM

- STMs come in different forms
 - Lock-based
 - Lock-free
- Here : a simple lock-based STM
- Lets start by Guaranteeing External Consistency

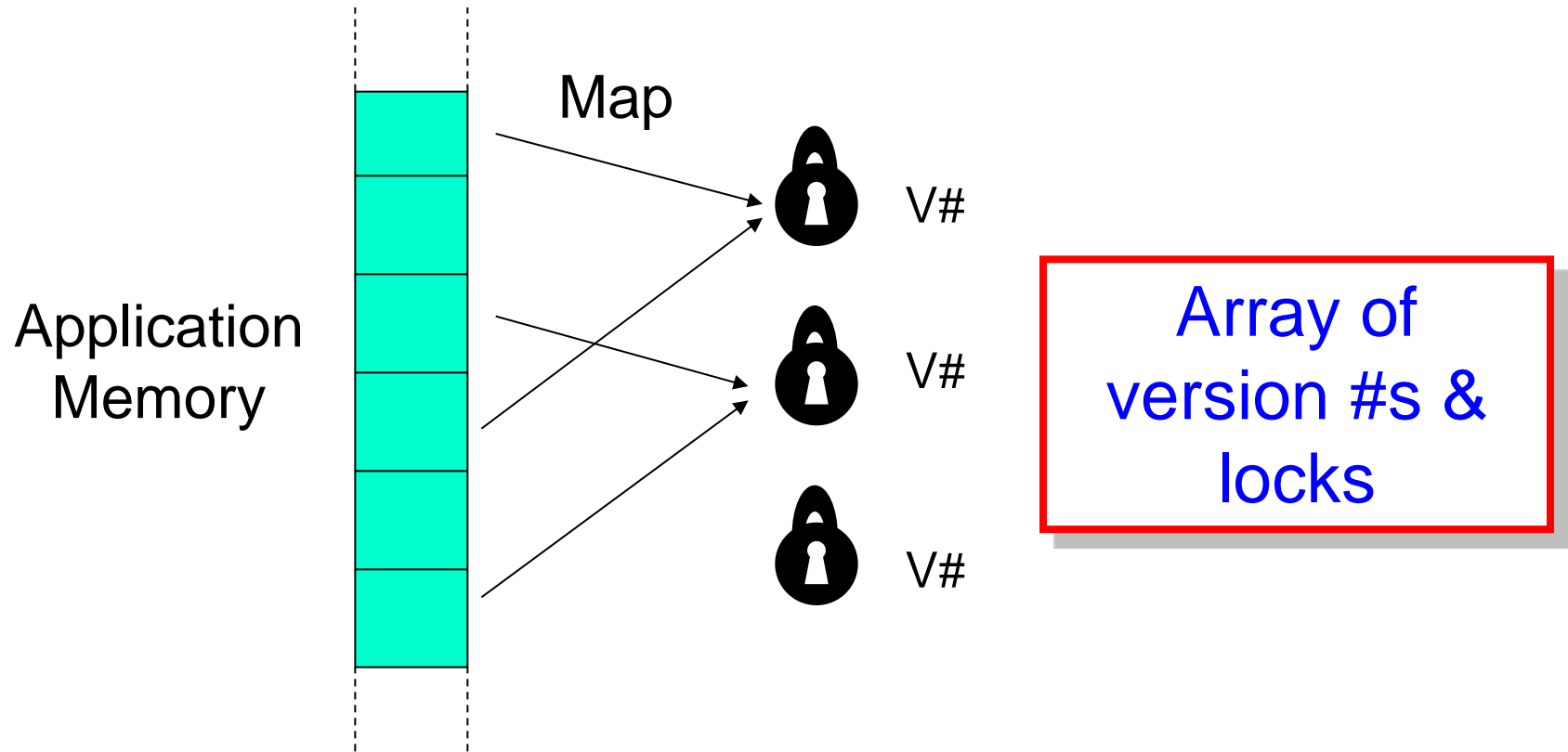


Synchronization

- 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



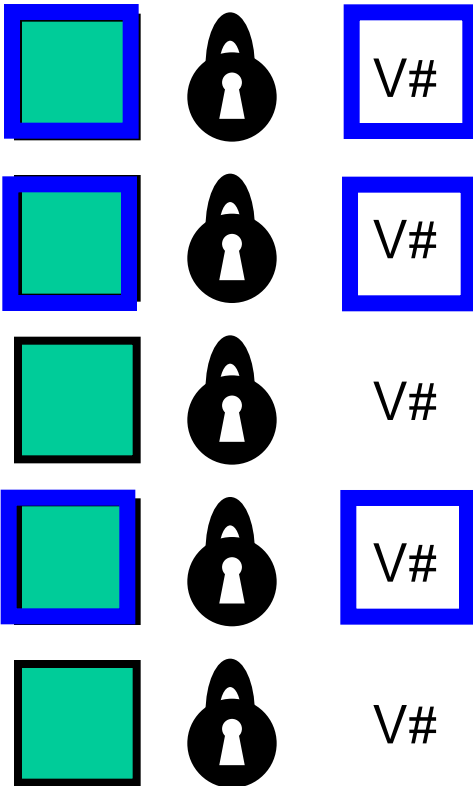
STM: Transactional Locking



Reading an Object

Mem

Locks



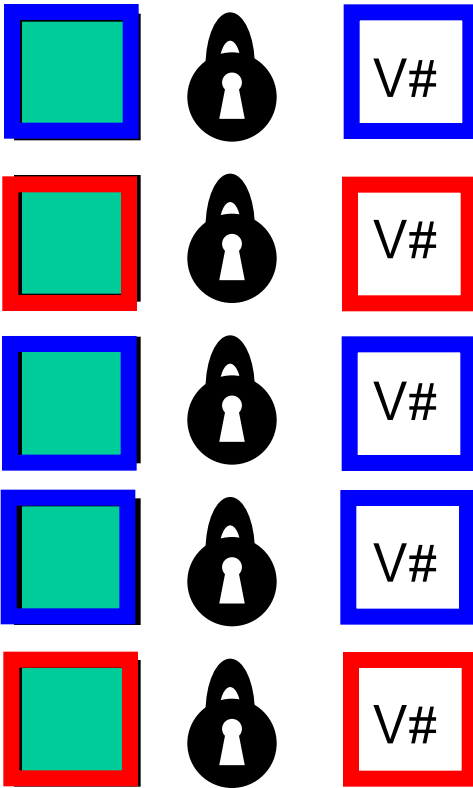
Add version numbers
& values to **read set**



To Write an Object

Mem

Locks



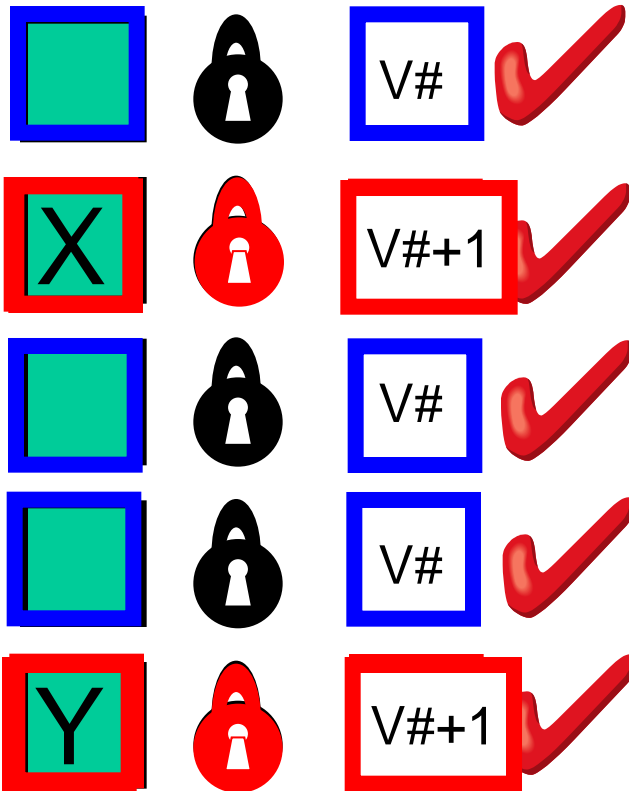
Add version numbers & new values to write set



To Commit

Mem

Locks



Acquire write locks

Check version numbers
unchanged

Install new values

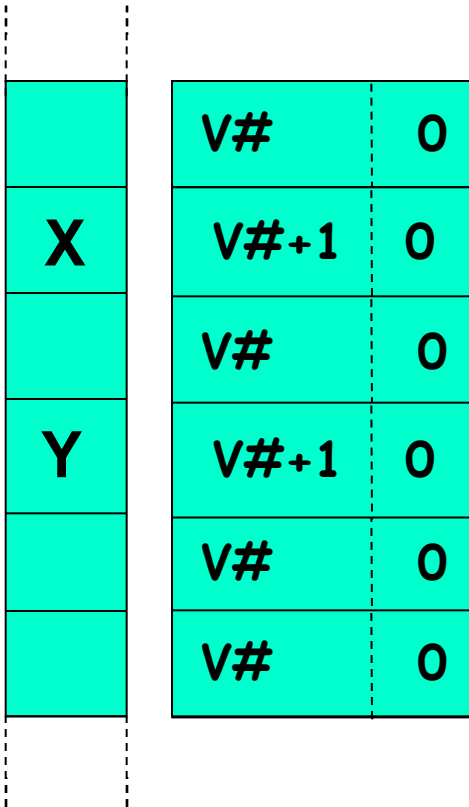
Increment version numbers

Unlock.



Encounter Order Locking (Undo Log)

Mem Locks

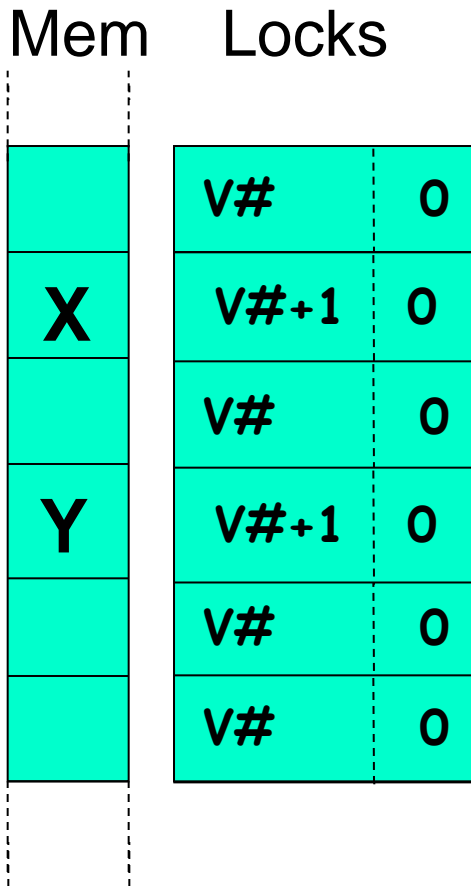


1. To Read: load lock + location
2. Check unlocked add to Read-Set
3. To Write: lock location, store value
4. Add old value to undo-set
5. Validate read-set v#'s unchanged
6. Release each lock with v#+1

Quick read of values freshly written by the reading transaction



Commit Time Locking (Write Buff)



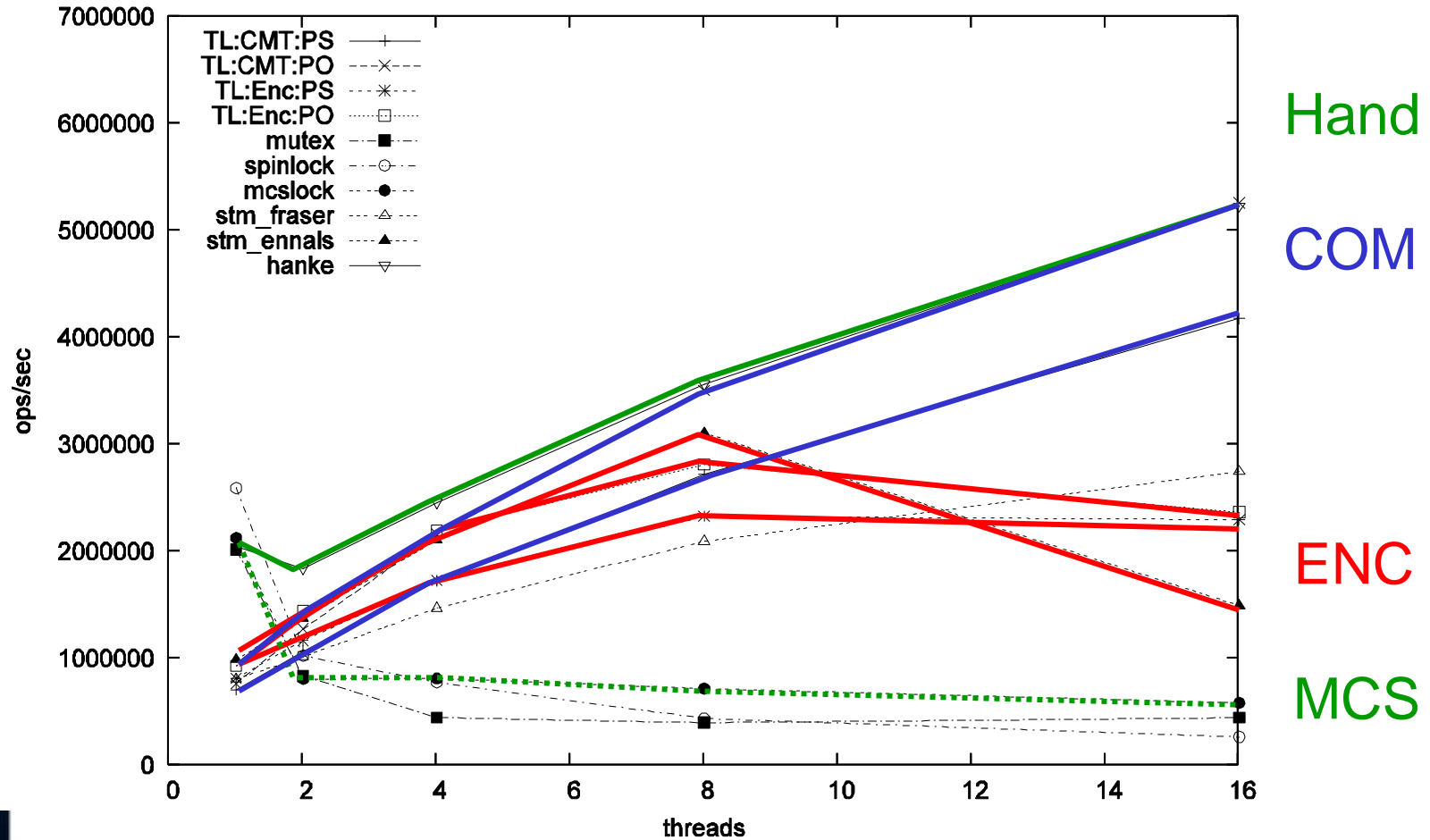
1. To Read: load lock + location
2. Location in write-set? (Bloom Filter)
3. Check unlocked add to Read-Set
4. To Write: add value to write set
5. **Acquire Locks**
6. Validate read/write v#' s unchanged
7. **Release each lock with v#+1**

Hold locks for very short duration



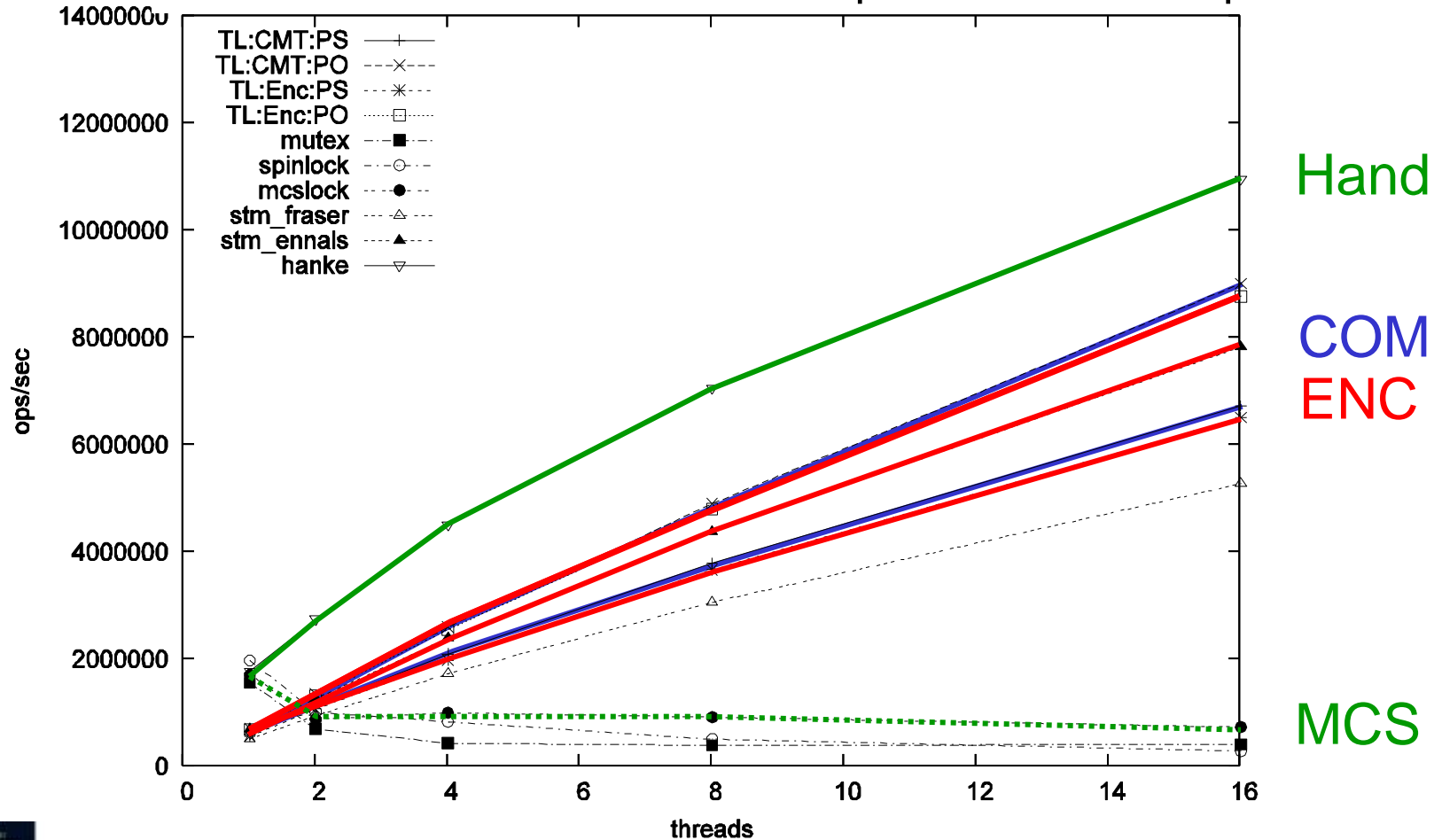
COM vs. ENC High Load

Red-Black Tree 20% Delete 20% Update 60% Lookup



COM vs. ENC Low Load

Red-Black Tree 5% Delete 5% Update 90% Lookup





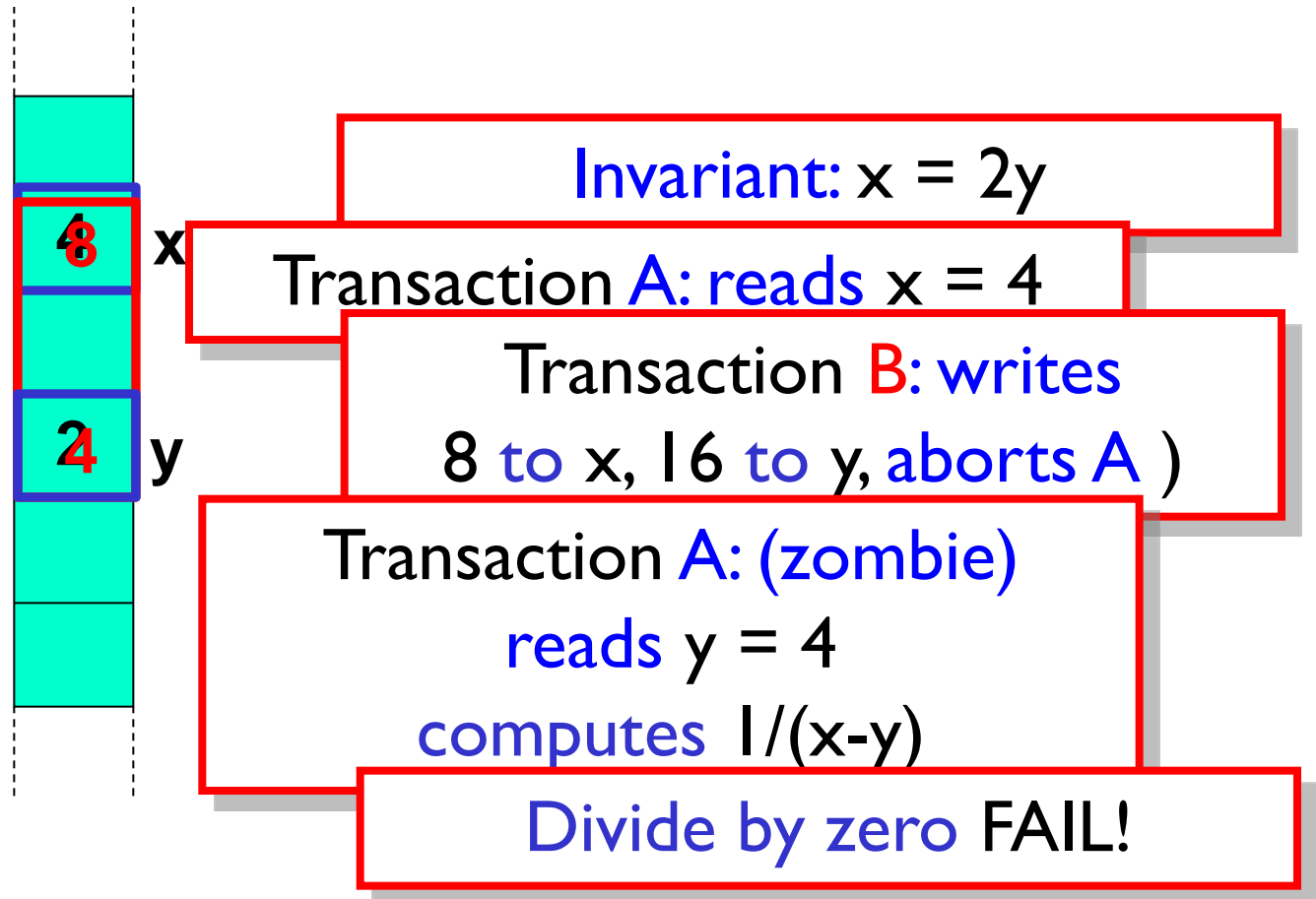
Problem: Internal Inconsistency



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



Internal Consistency



Solution: The Global Clock (The TL2 Algorithm)

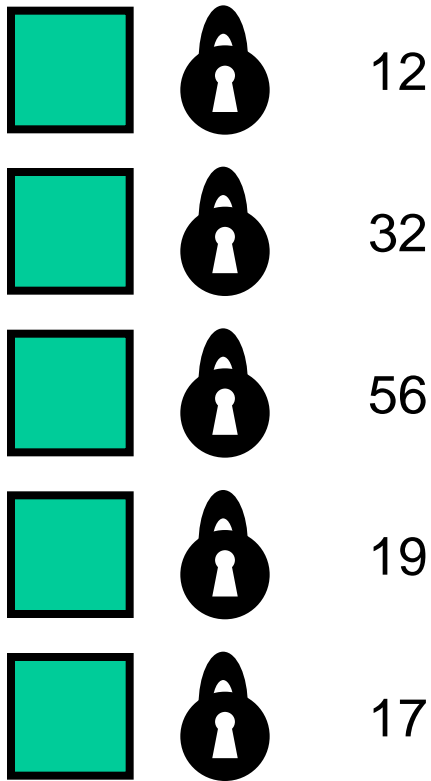
- Have one shared global clock
- Incremented by (small subset of) writing transactions
- Read by all transactions
- Used to validate that state worked on is always consistent



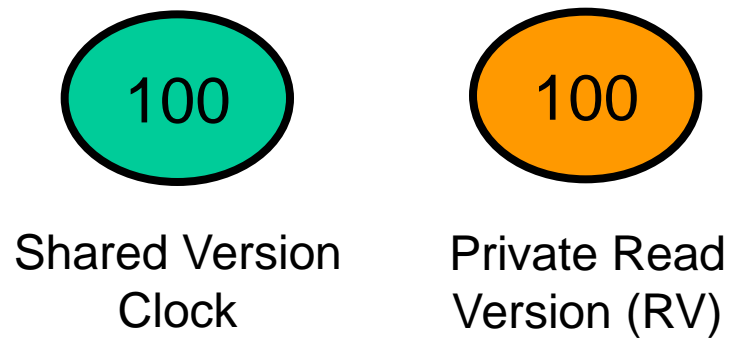
Read-Only Transactions

Mem

Locks



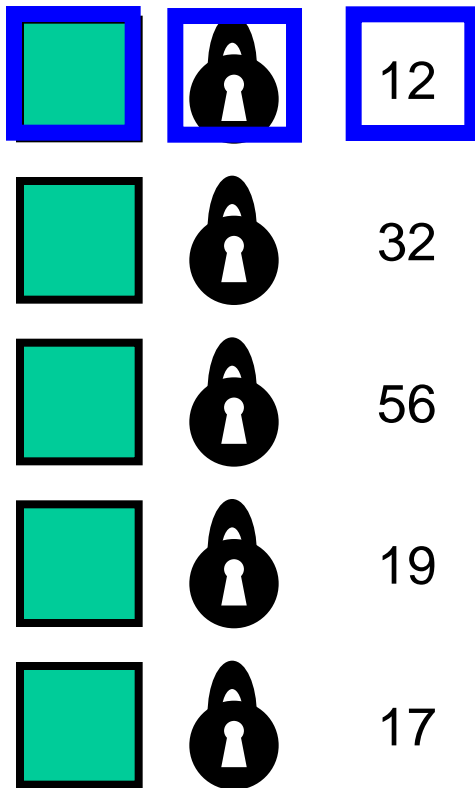
Copy version clock to local read version clock



Read-Only Transactions

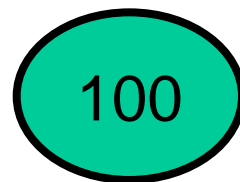
Mem

Locks

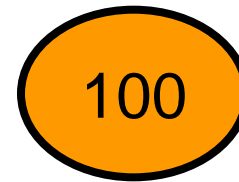


Copy version clock to local

Read lock, version #, and memory



Shared Version Clock



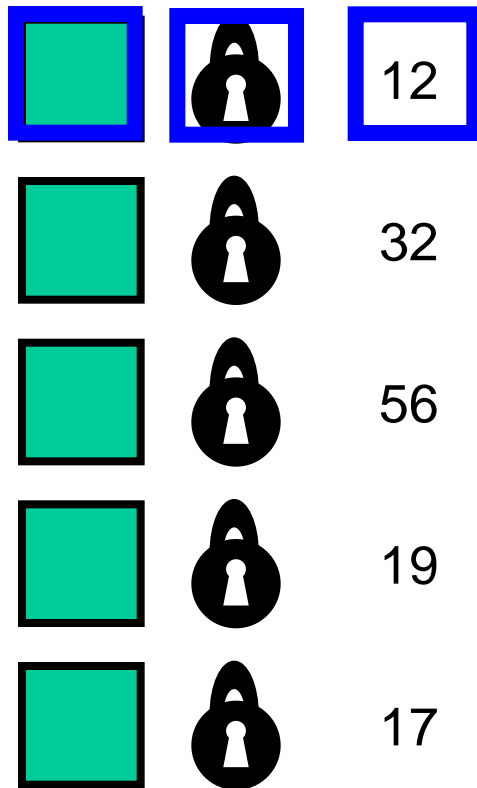
Private Read Version (RV)



Read-Only Transactions

Mem

Locks



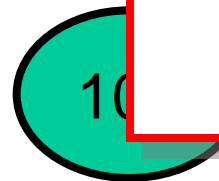
Copy version clock to local

Read lock, version #, and

memory address to local version #s

On Commit:

check unlocked &
version #s less than
local read clock



Shared Version
Clock

Private Read
Version (RV)



Read-Only Transactions

Mem

Locks



Copy version clock to local

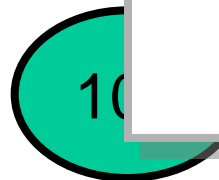


Read lock, version #, and

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



version #s less than local lock

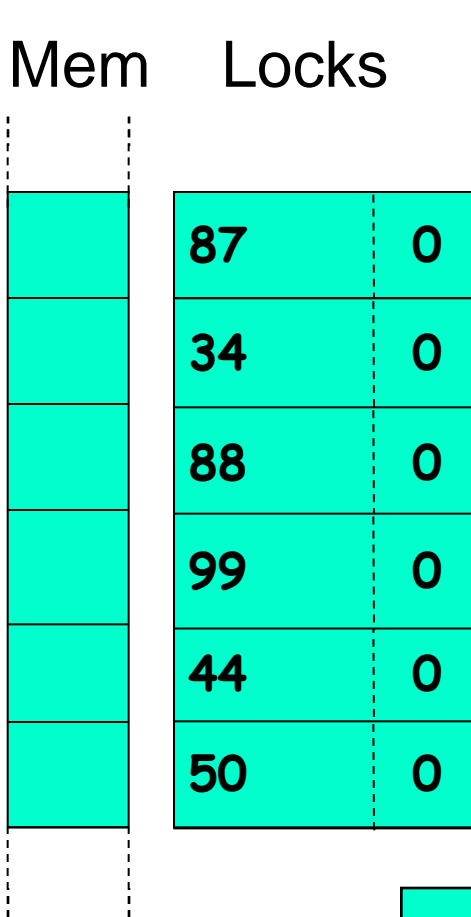


Shared Version Clock

Private Read Version (RV)



Example Execution: Read Only Trans



100

Shared Version Clock

1. $RV \leftarrow$ Shared Version Clock
2. On Read: read lock, read mem, read lock: check unlocked, unchanged, and $v\# \leq RV$
3. Commit.

Reads form a snapshot of memory.
No read set!

100

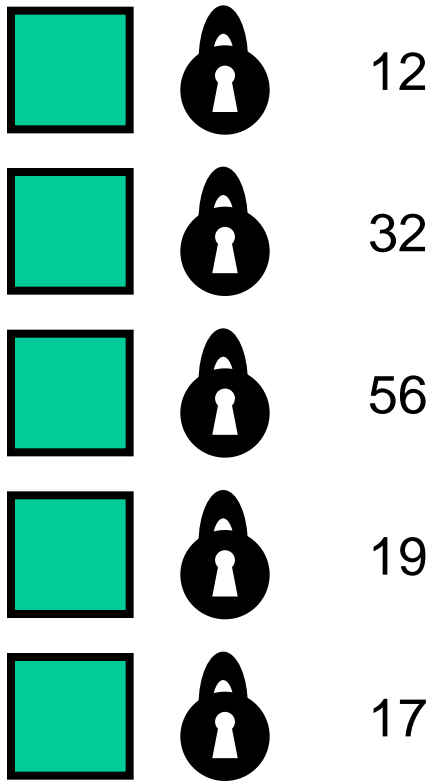
RV



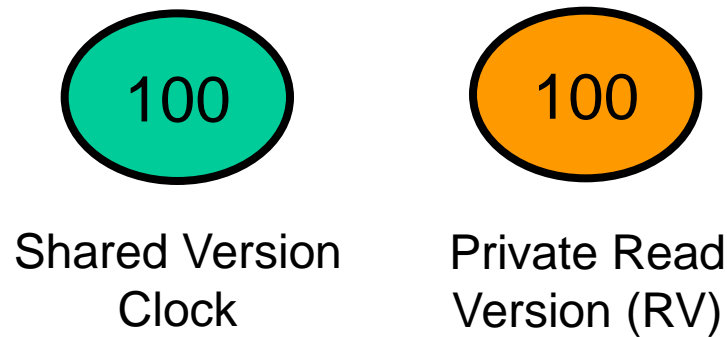
Ordinary (Writing) Transactions

Mem

Locks



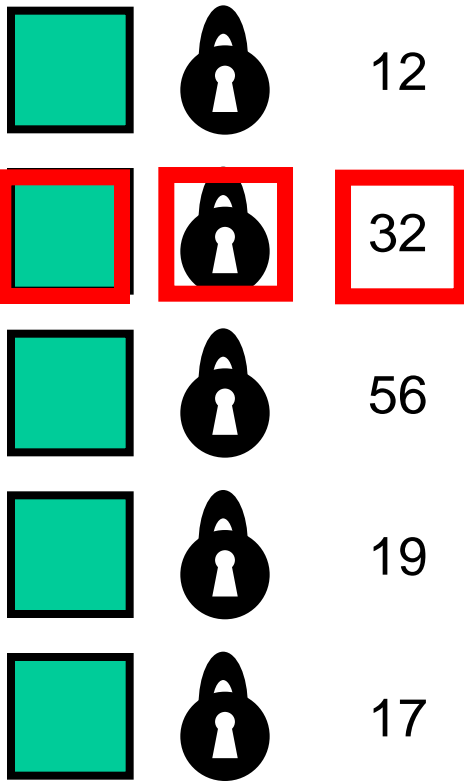
Copy version clock to local read version clock



Ordinary Transactions

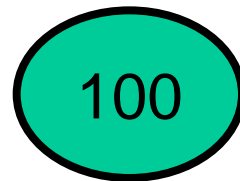
Mem

Locks

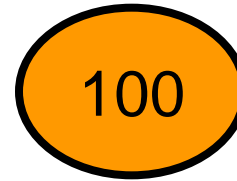


Copy version clock to local
read version clock

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



Shared Version
Clock



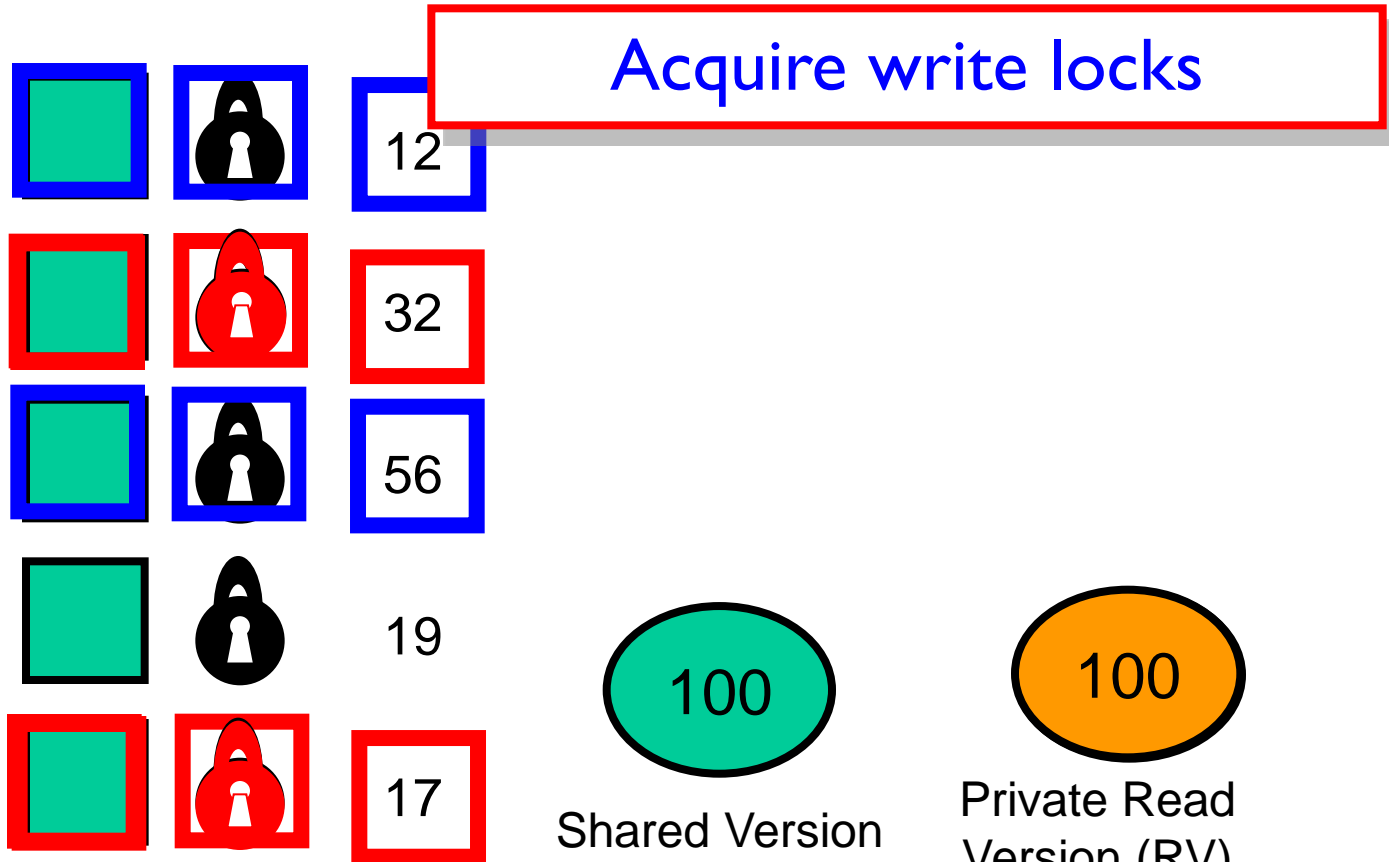
Private Read
Version (RV)



On Commit

Mem

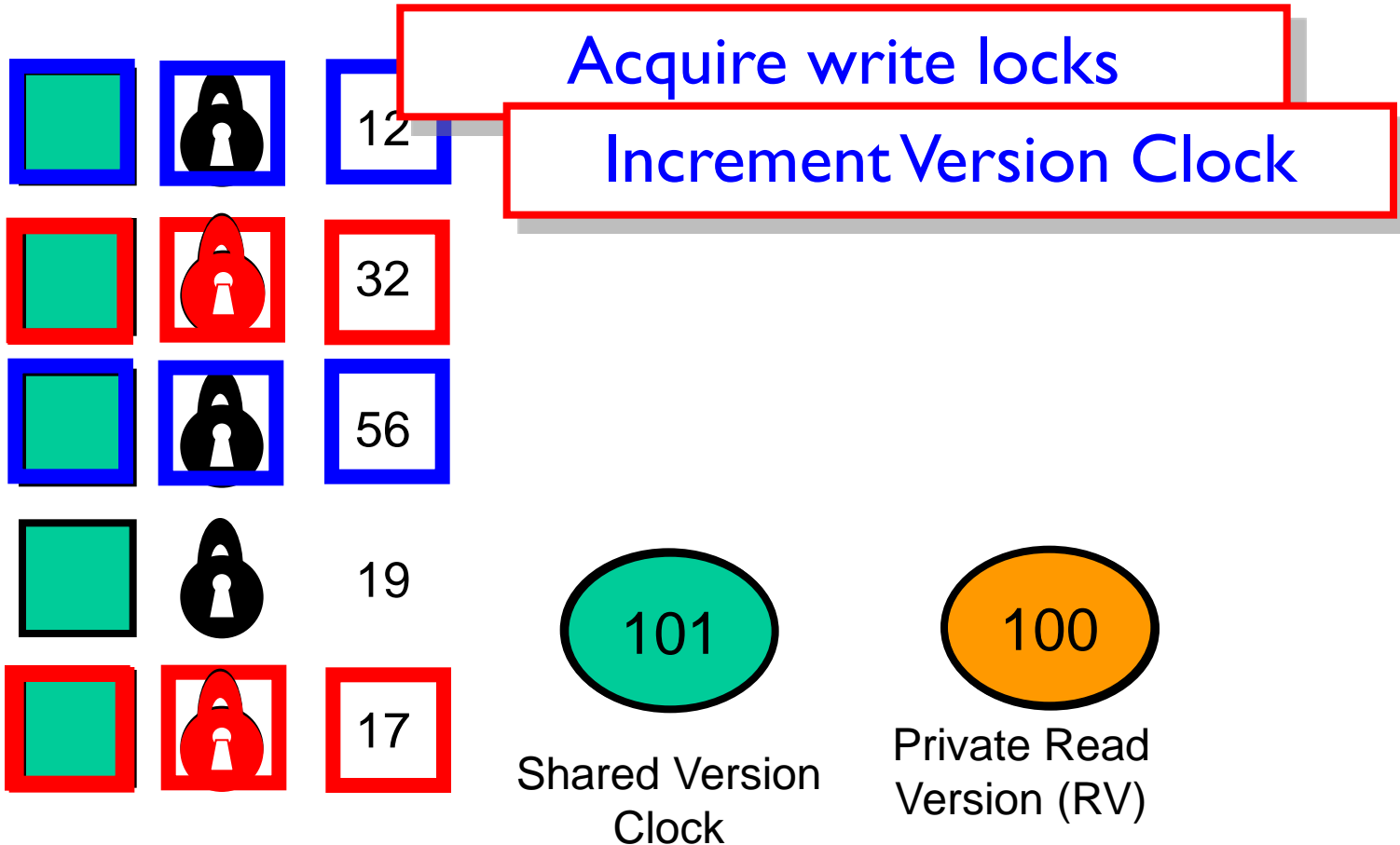
Locks



On Commit

Mem

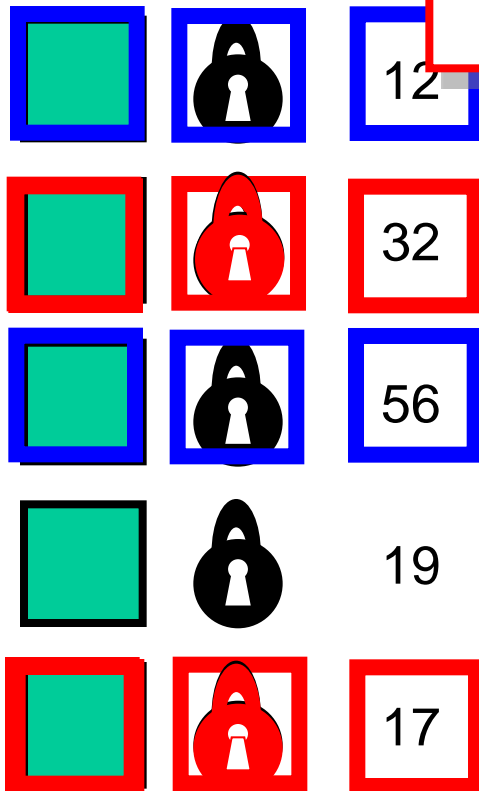
Locks



On Commit

Mem

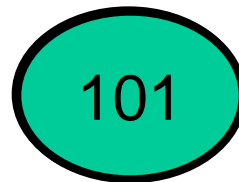
Locks



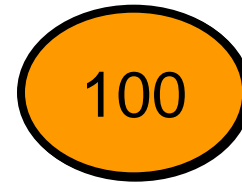
Acquire write locks

Increment Version Clock

Check version numbers $\leq RV$



Shared Version
Clock



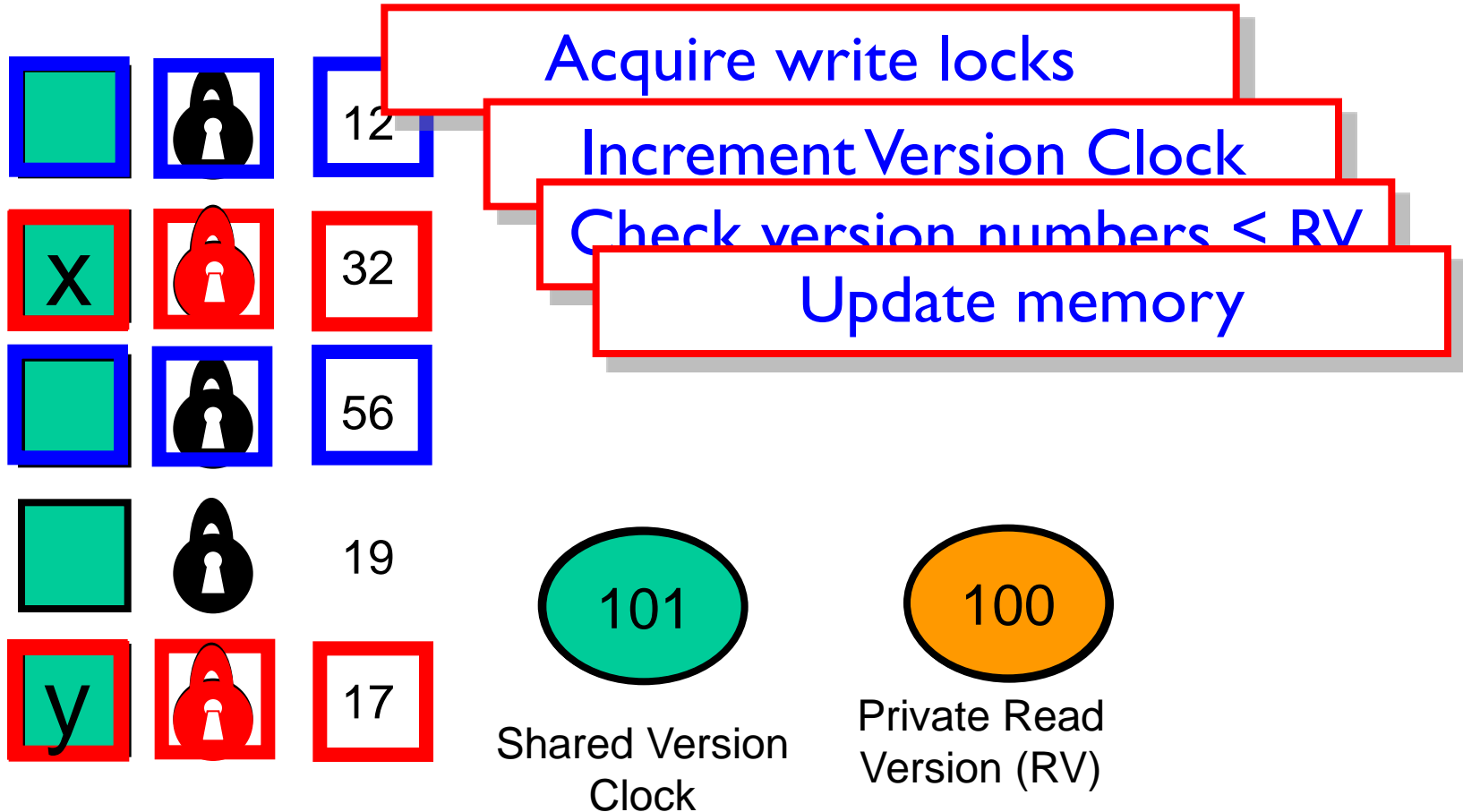
Private Read
Version (RV)



On Commit

Mem

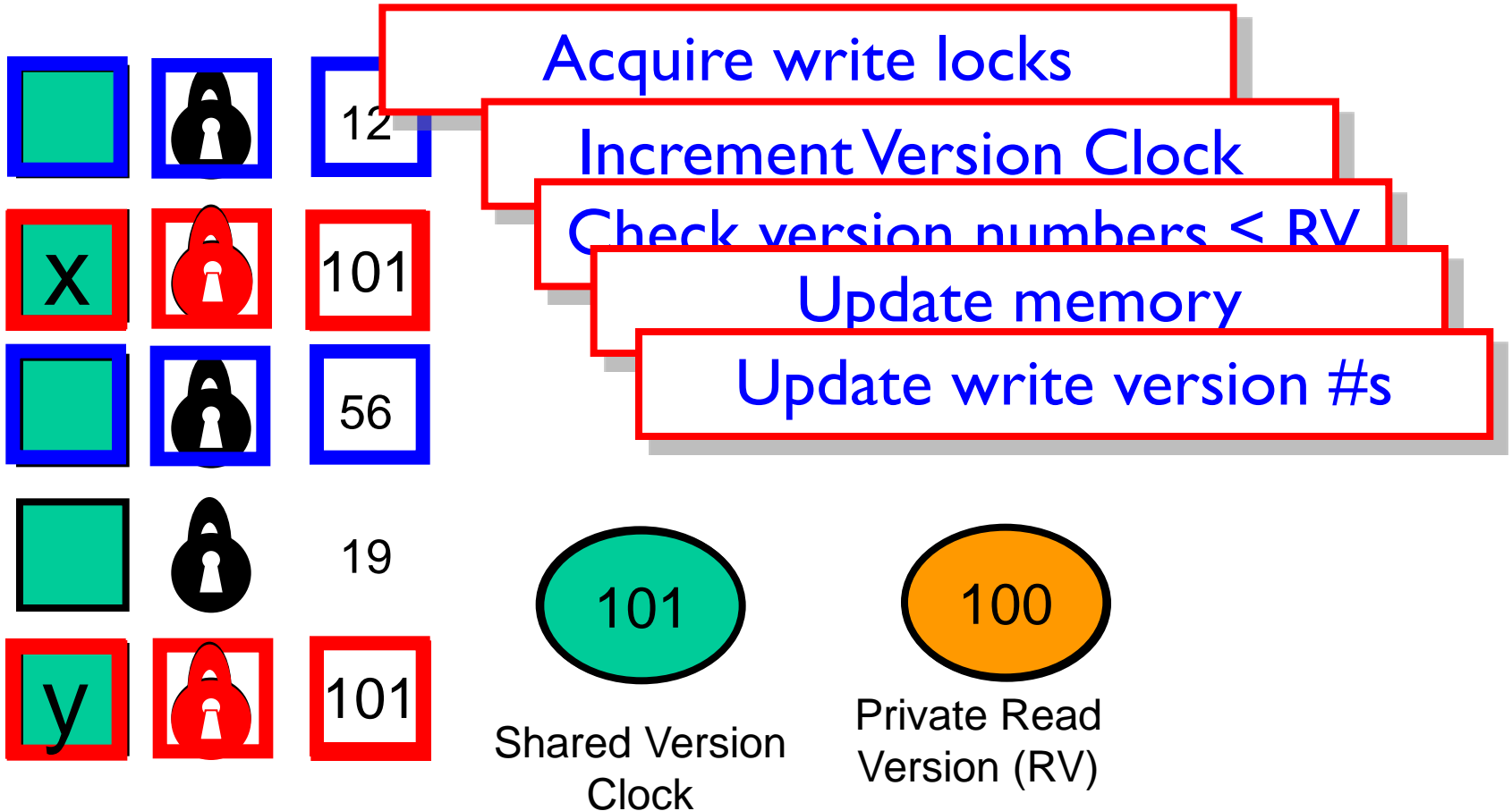
Locks



On Commit

Mem

Locks



Example: Writing Trans

Mem	Locks
	87 0
x	121 0
	88 0
Y	121 0
	44 0
	50 0

121

Shared Version Clock

1. $RV \leftarrow$ Shared Version Clock
2. On Read/Write: check unlocked and $v\# \leq RV$ then add to Read/Write-Set
3. Acquire Locks
4. $WV = F\&I(VClock)$
5. Validate each $v\# \leq RV$
6. Release locks with $v\# \leftarrow WV$

Commit

100

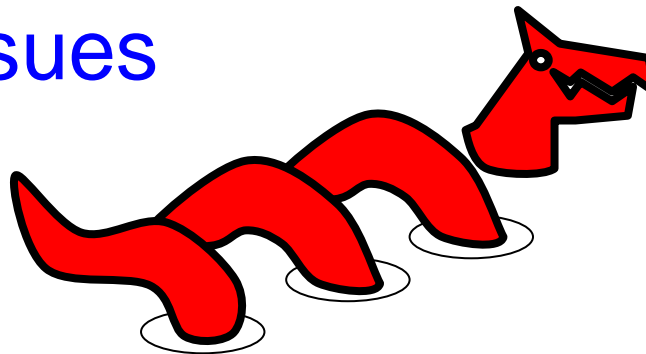
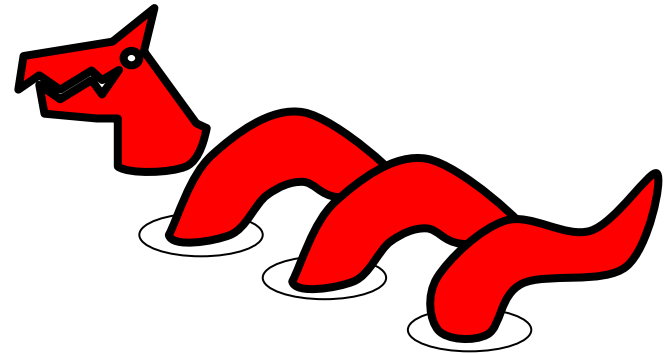
RV

Reads+Inc+Writes
=serializable



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!

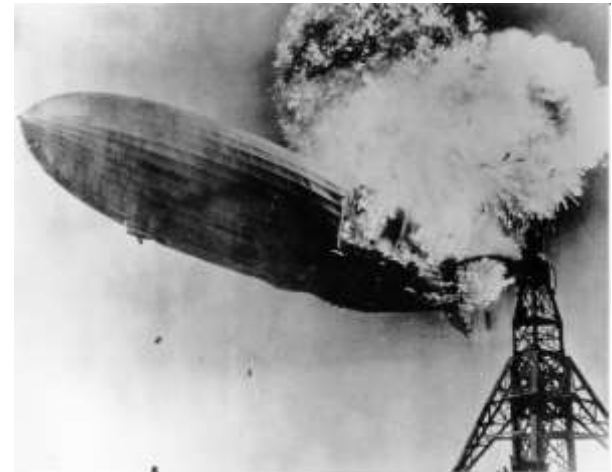
```
int i = 0;
try {
  atomic {
    i++;
    node = new Node();
  }
} catch (Exception e) {
  print(i);
}
```

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



Hatin' on TM

STM is too inefficient

software
transactional

→ why is it only a research toy?

- Interaction with code that as a result of either commu
- Livelock, or the system gu

In addition to the intrin:
also implementation-specifi
by high transactional overh
nondeterminism introd
tions complicates debuggin
executed and aborted on co
repeatable behavior. Both o
argument for the programmer to
implementations. Both o
Given all these issues, wi
yet matured to the point wi

degradation when overflow occurs, and proposals for
managing overflows (for example, signatures¹⁷) incur false
positives that add complexity to the programming model.
Therefore, from an industrial perspective, HTM designs
provide more benefits for the cost on a more
workloads (with varying transactional char-



Hatin' on TM

The problem with STM: your languages still suck

Published by [Brian](#) at 12:13 PM under [To Be Categori](#)

So, I'm reading [Brandon Werner's post](#) caught on yet. There are three Act, allowing many to Java, C#,

- Interaction with code that as a result of either commu a requirement barring spec
- Livelock, or the system gu make progress even in the f
- In addition to the intrin: implementation-specifi
- Transactional overh for excluding F
- Debuggin
- on co
- t

Requires radical change in programming style

→ why is it o

degradation when overflow managing overflows (for exam positives that add complexity to L. Therefore, from an industrial persp provide more benefits for the a. workloads (with varying tran.

Software Transactional Memory has succeed in implementing STM for "mainstream" languages l that make it a CS Researcher Full Employment



Hatin' on TM

Wrong Programming Model

Erlang-style shared nothing only true path to salvation

→ why is it ok

degradation when overflow
managing overflows (for exam
positives that add complexity to L
Therefore, from an industrial persp
provide more benefits for the c
workloads (with varying tran

TM, about why Software Transactional Memory has
with STM that make it a CS Researcher Full Employment
succeed in implementing STM for "mainstream" languages

Hatin' on TM

Monday Nov 03, 2008

Concurrency's Shysters

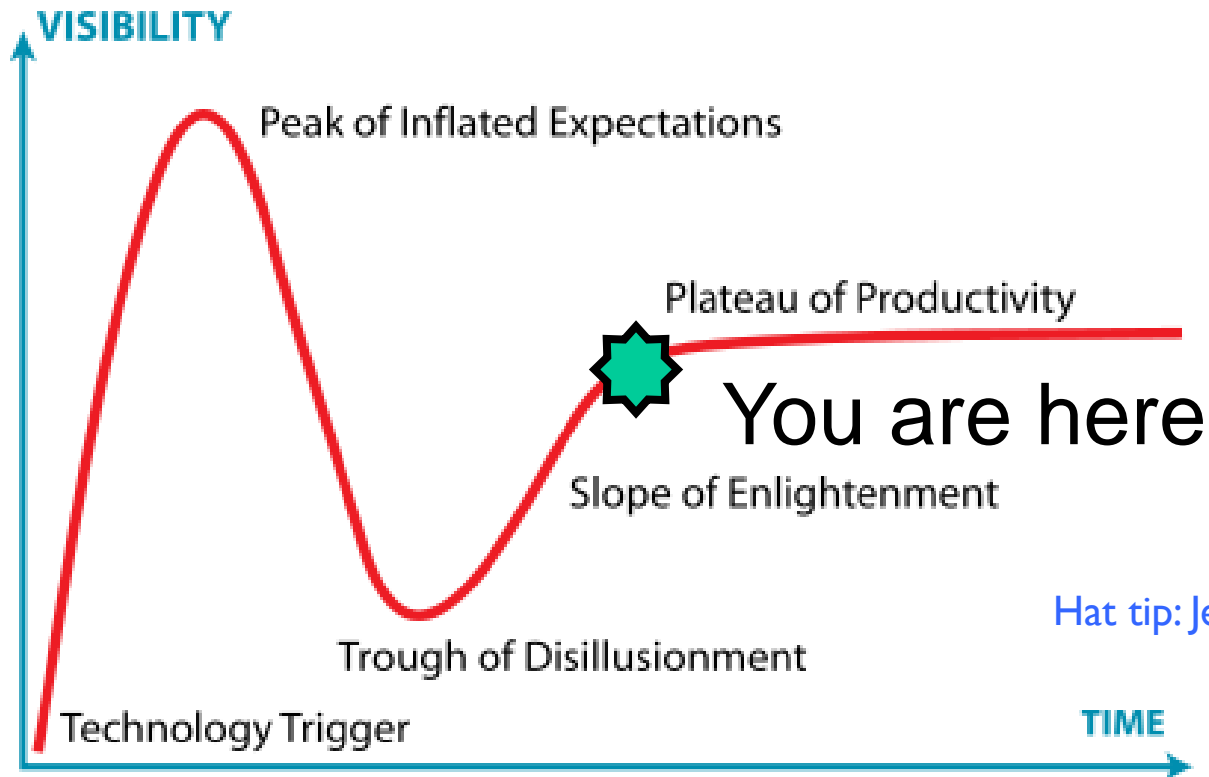
For as long as I've been in computing, the subject of concurrency has always induced a diatribe -- the program was coming up, the name of the apocalypse was symmetric multiprocessing -- and as complex as the stuff was, it was for a purpose -- a baby step towards a more complex purpose. I was a software engineer who -- despite the fact that I was a software engineer -- confidently asserted that I could handle a scale beyond 8 CPUs. Needless to say, I was wrong.

There is nothing wrong with what we do today.

→ why is it
degradation when overflow
managing overflows (for exam
positives that add complexity to
therefore, from an industrial persp
provide more benefits for the c
workloads (with varying tran



Gartner Hype Cycle



Hat tip: Jeremy Kemp



Thanks ! תודה



Overview

- Building shared memory data structures
 - Lists, queues, hashtables, ...
- Why?
 - Used directly by applications (e.g., in C/C++, Java, C#, ...)
 - Used in the language runtime system (e.g., management of work, implementations of message passing, ...)
 - Used in traditional operating systems (e.g., synchronization between top/bottom-half code)
- Why not?
 - Don't think of "threads + shared data structures" as a default/good/complete/desirable programming model
 - It's better to have shared memory and not need it...

Different techniques for different problems

