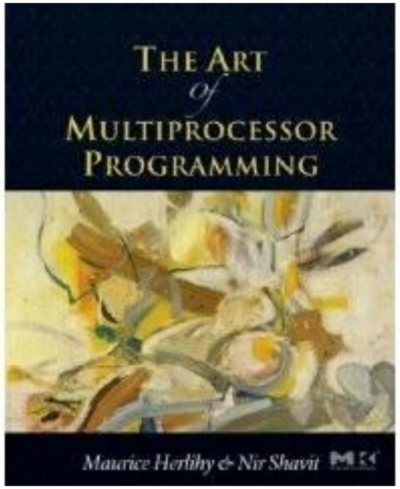
# NON-BLOCKING DATA STRUCTURES AND TRANSACTIONAL MEMORY

Tim Harris, 25 November 2016

# Lecture 8

- 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

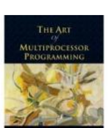
# Our Vision for the Future

In this course, we covered ….

Best practices …

New and clever ideas …

And common-sense observations.



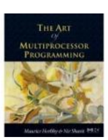
# Our Vision for the Future

Nevertheless …

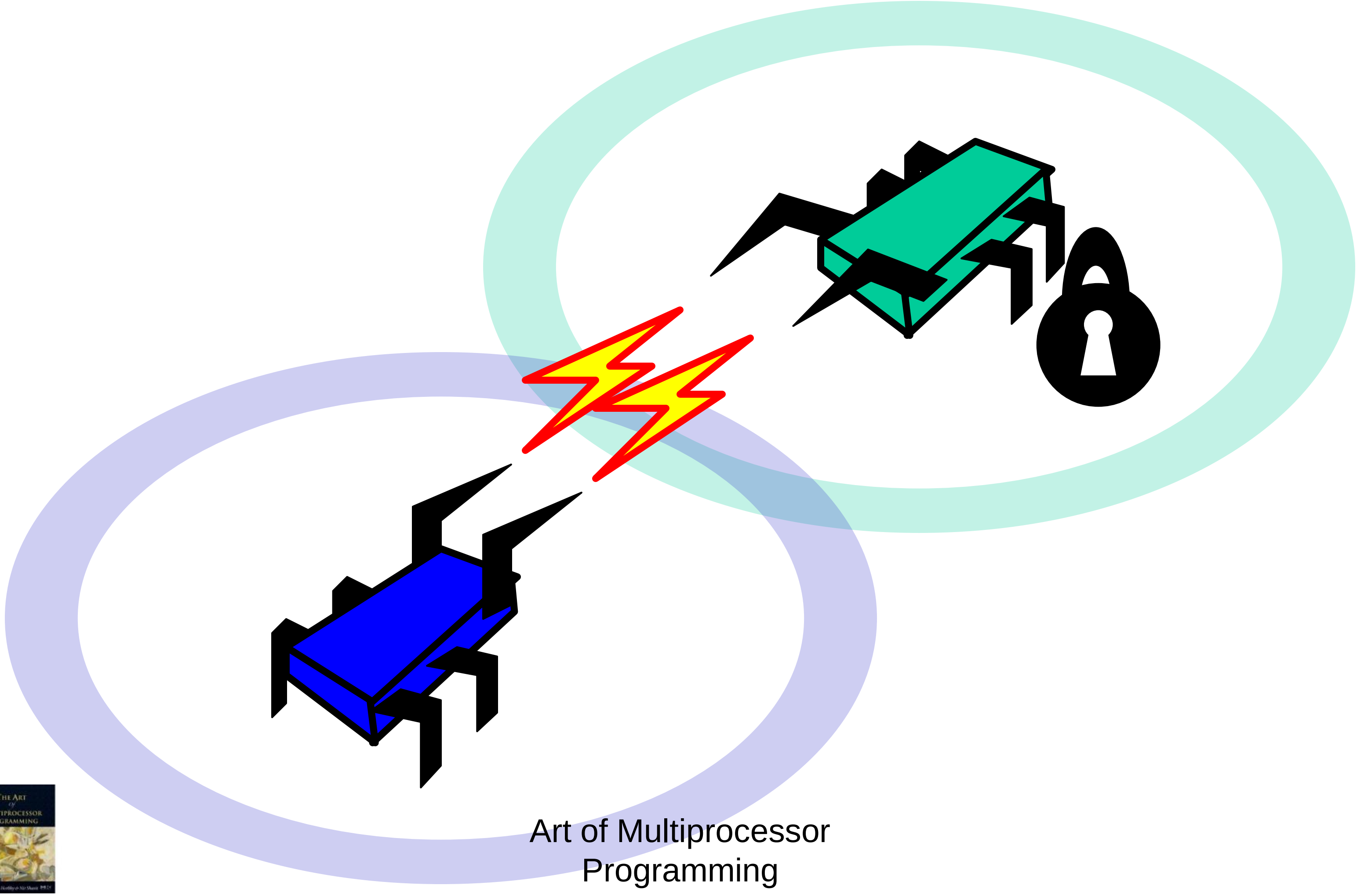
Concurrent programming is still too hard …

Here we explore why this is ….

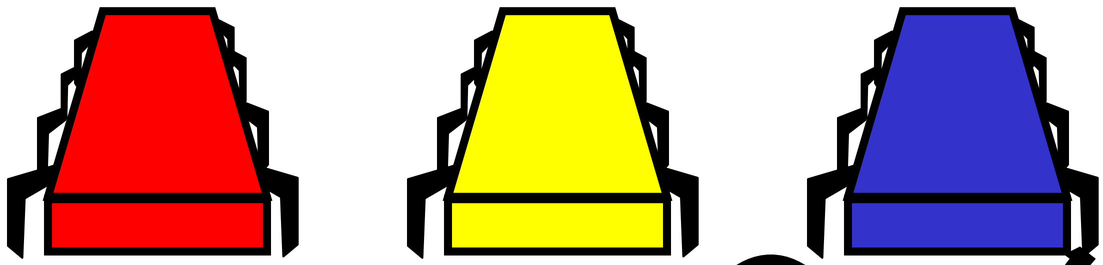
And what we can do about it.



# Locking

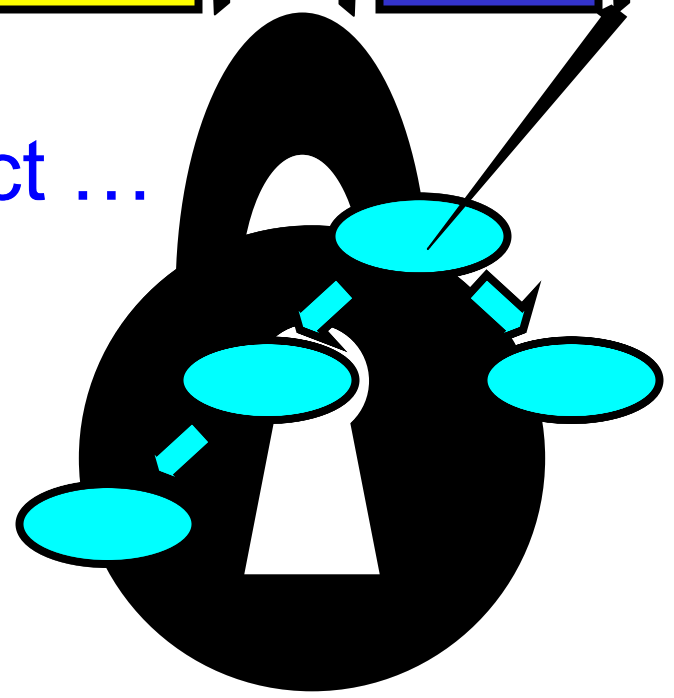


# Coarse-Grained Locking

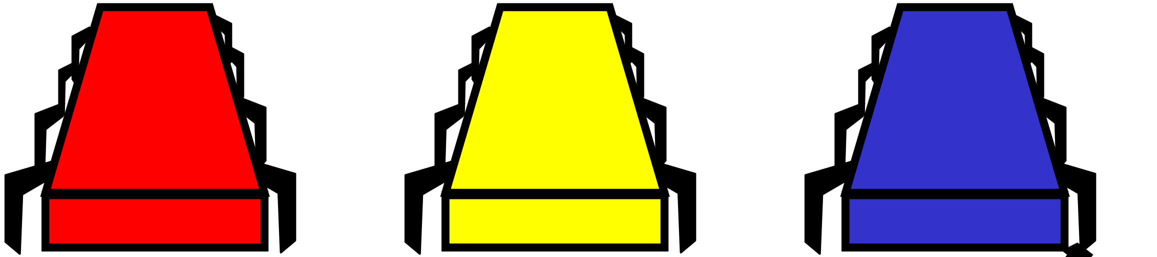


Easily made correct …

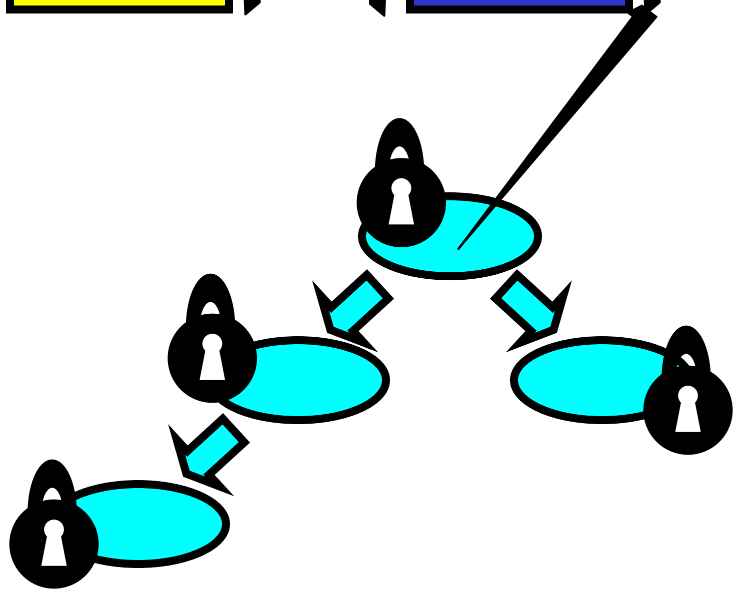
But not scalable.



# Fine-Grained Locking

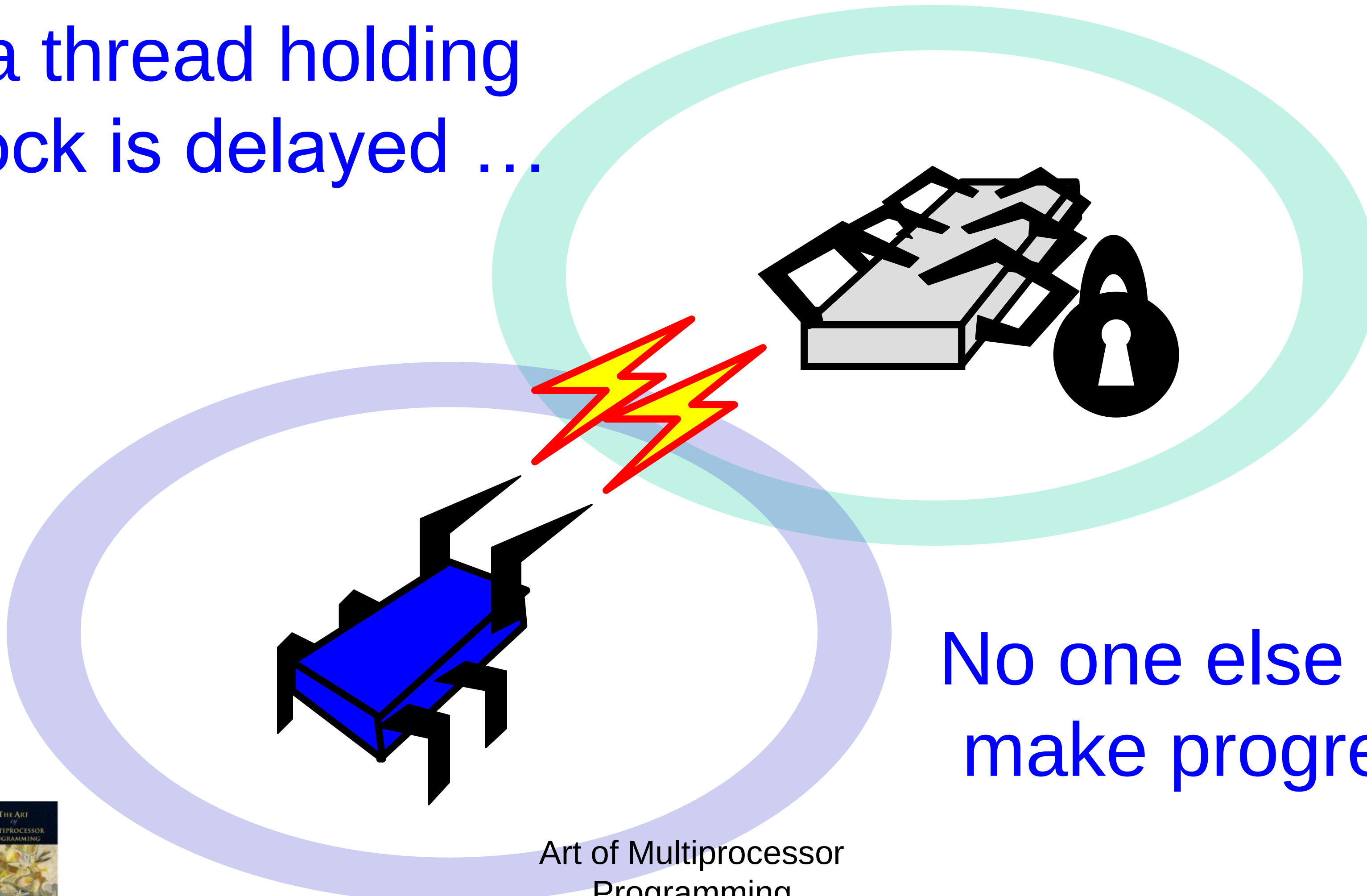


Can be tricky …



# Locks are not Robust

If a thread holding
a lock is delayed …



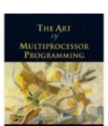
No one else can
make progress

# Locking Relies on Conventions

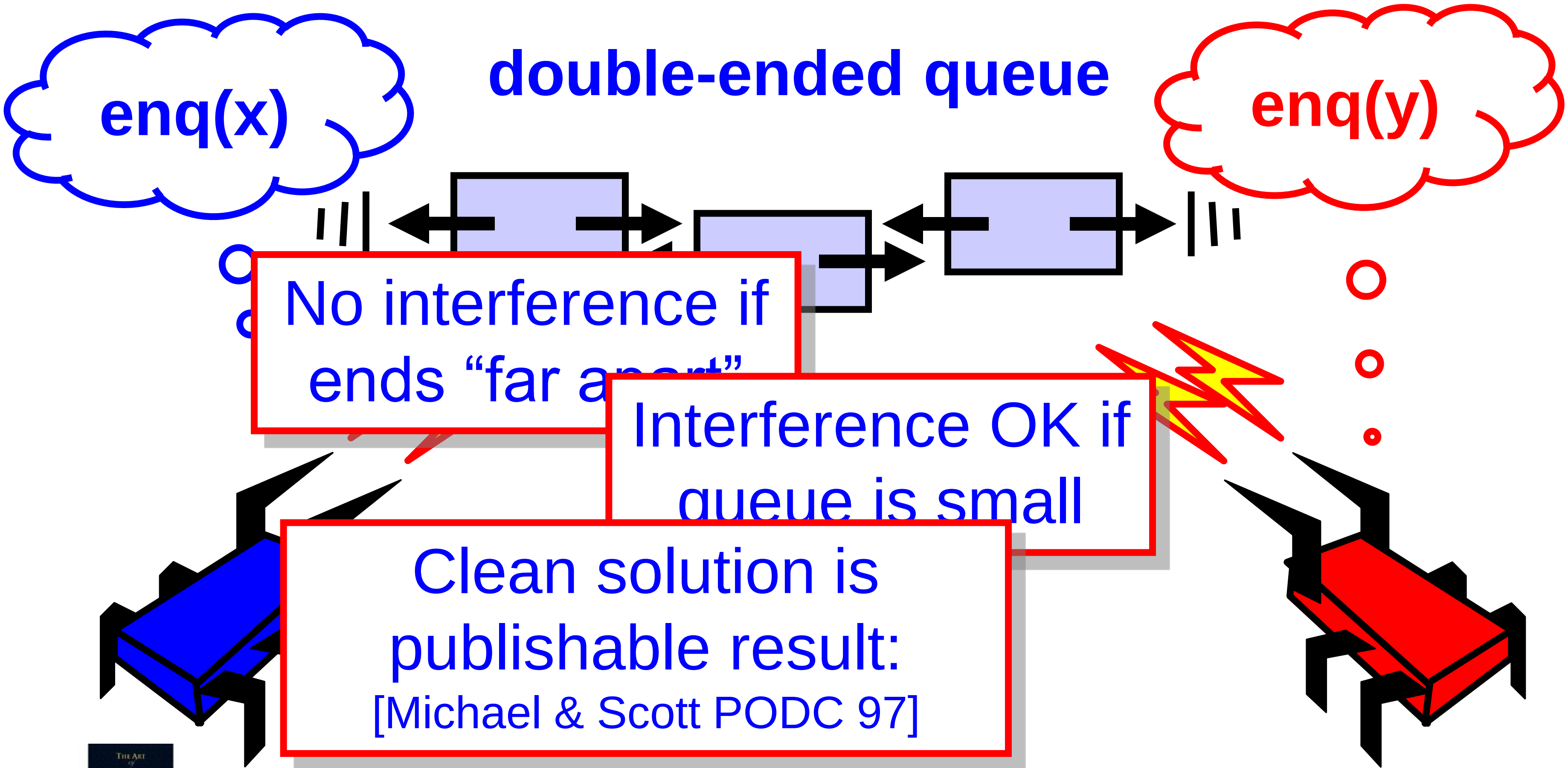
- Relation between
  - Locks and objects
  - Exists only in programmer's

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

```
/*
 * When a locked buffer is visible to the I/O layer
 * BH_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.
 */
```



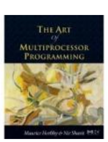
# Simple Problems are hard



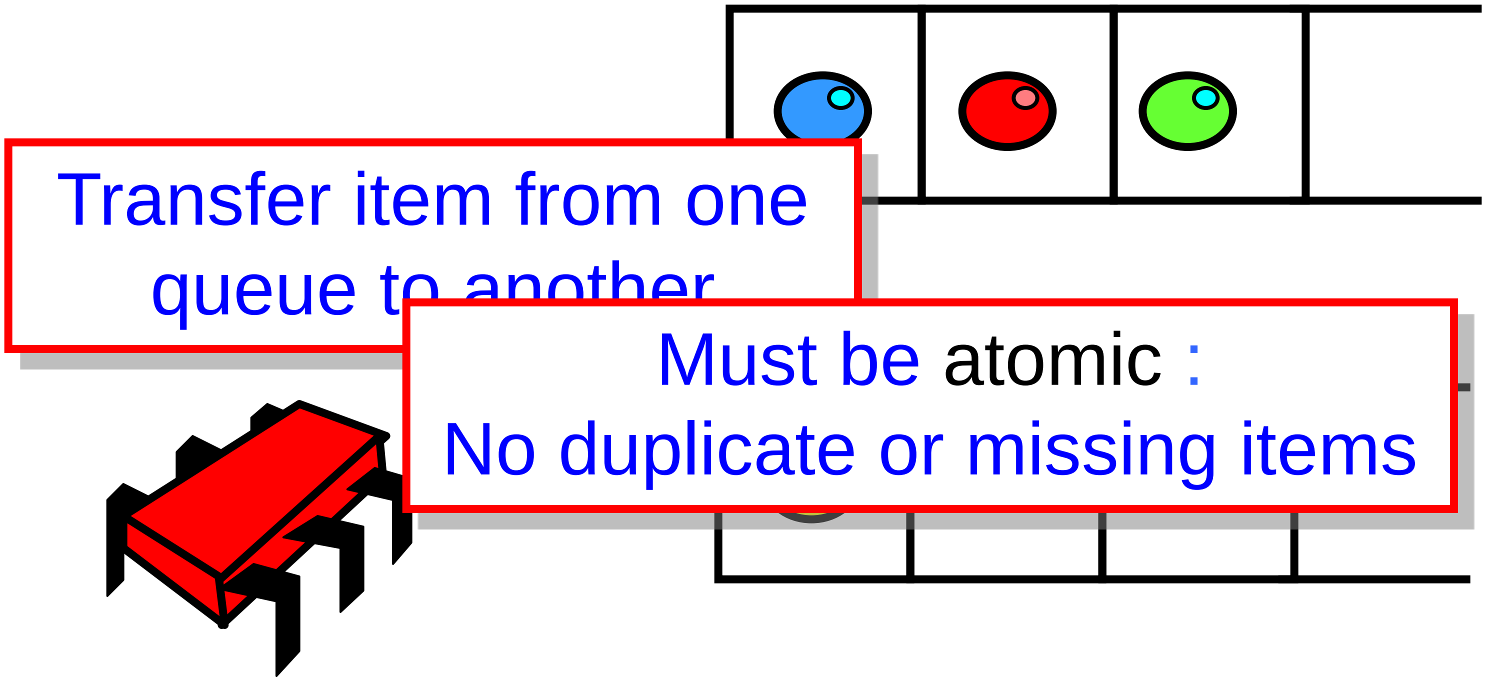

No interference if ends “far apart”

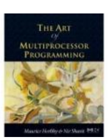
Interference OK if queue is small

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

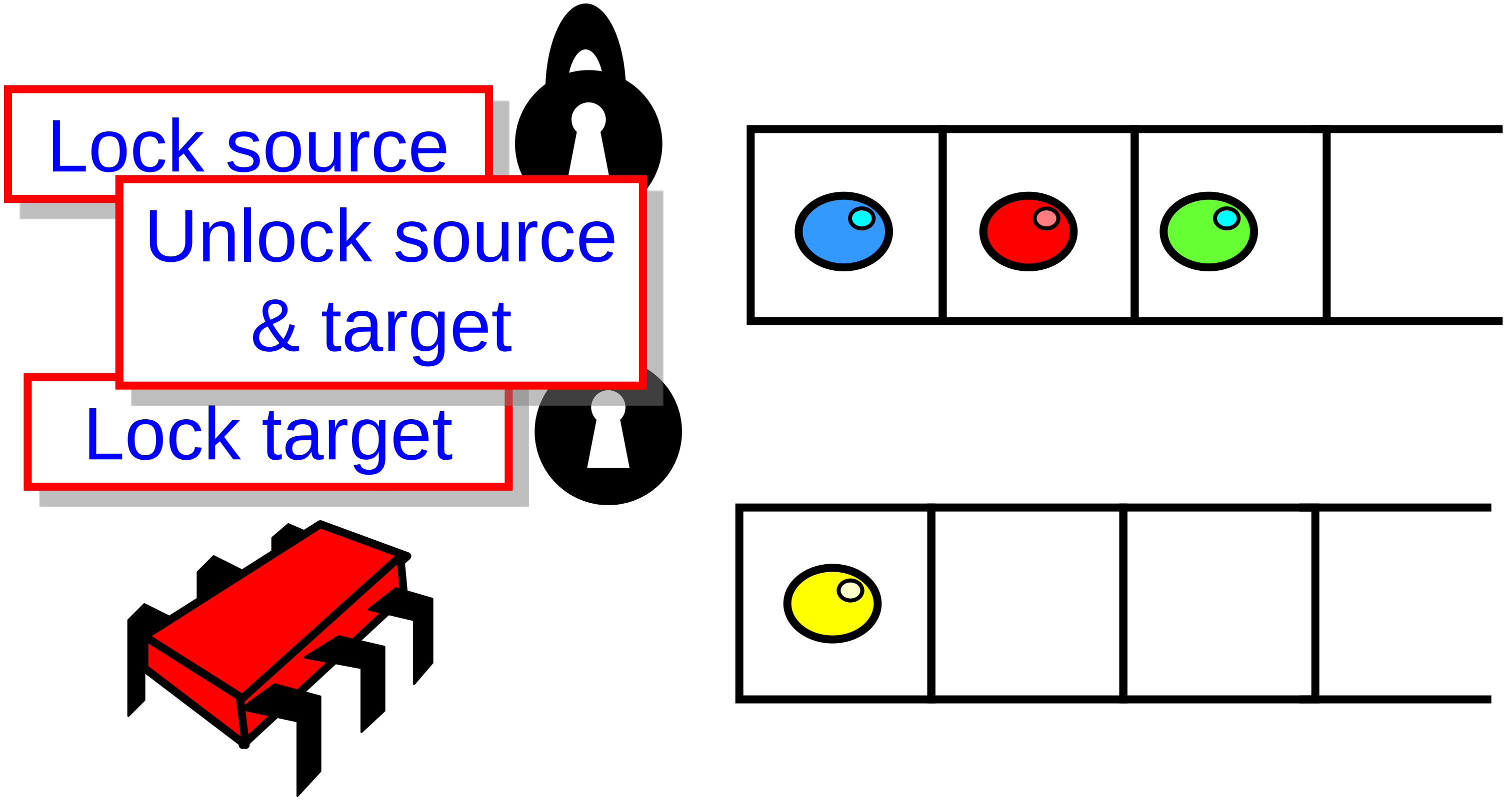


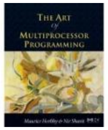
# Locks Not Composable



# Locks Not Composable



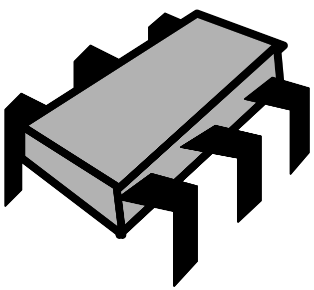
# Locks Not Composable

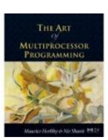
Methods cannot provide internal synchronization

Objects must expose locking protocols to clients

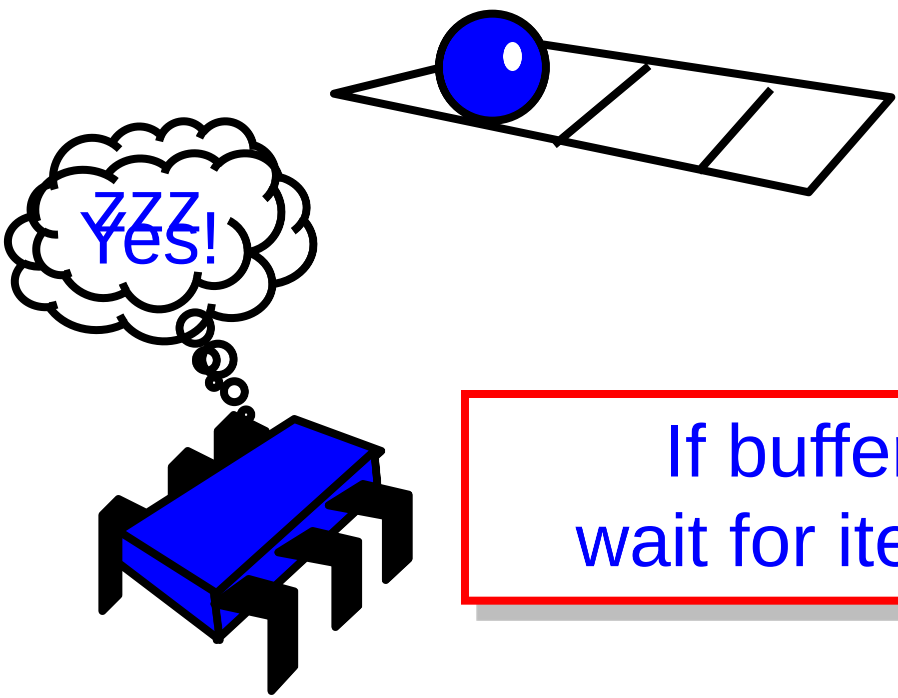
Clients must devise and follow protocols

Abstraction broken!



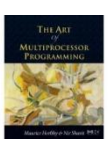


# Monitor Wait and Signal

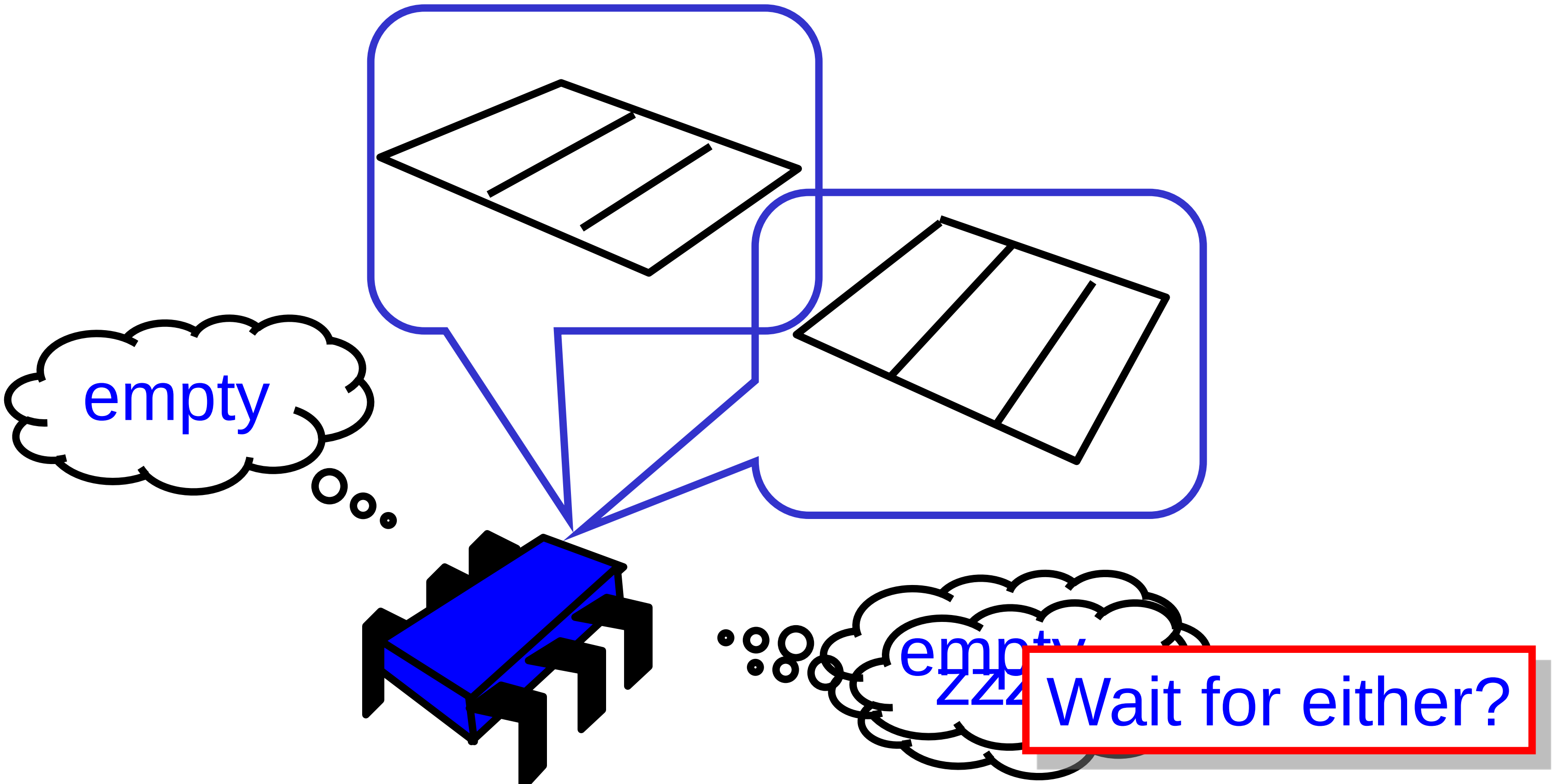



Empty buffer

If buffer is empty,
wait for item to show up

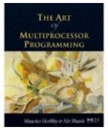


# Wait and Signal do not Compose



# The Transactional Manifesto

- Current practice inadequate
  – to meet the multicore challenge
- 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

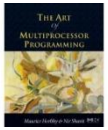
Commit: takes effect (atomically)

Abort: has no effect (typically restarted)

# Atomic Blocks

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

atomic {
 y = null;
}
```

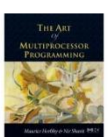



# Atomic Blocks

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

atomic {
 y = null;
}
```

**No data race**



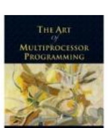
# A Double-Ended Queue

```
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;
 }
}
```



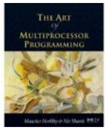
# A Double-Ended Queue

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

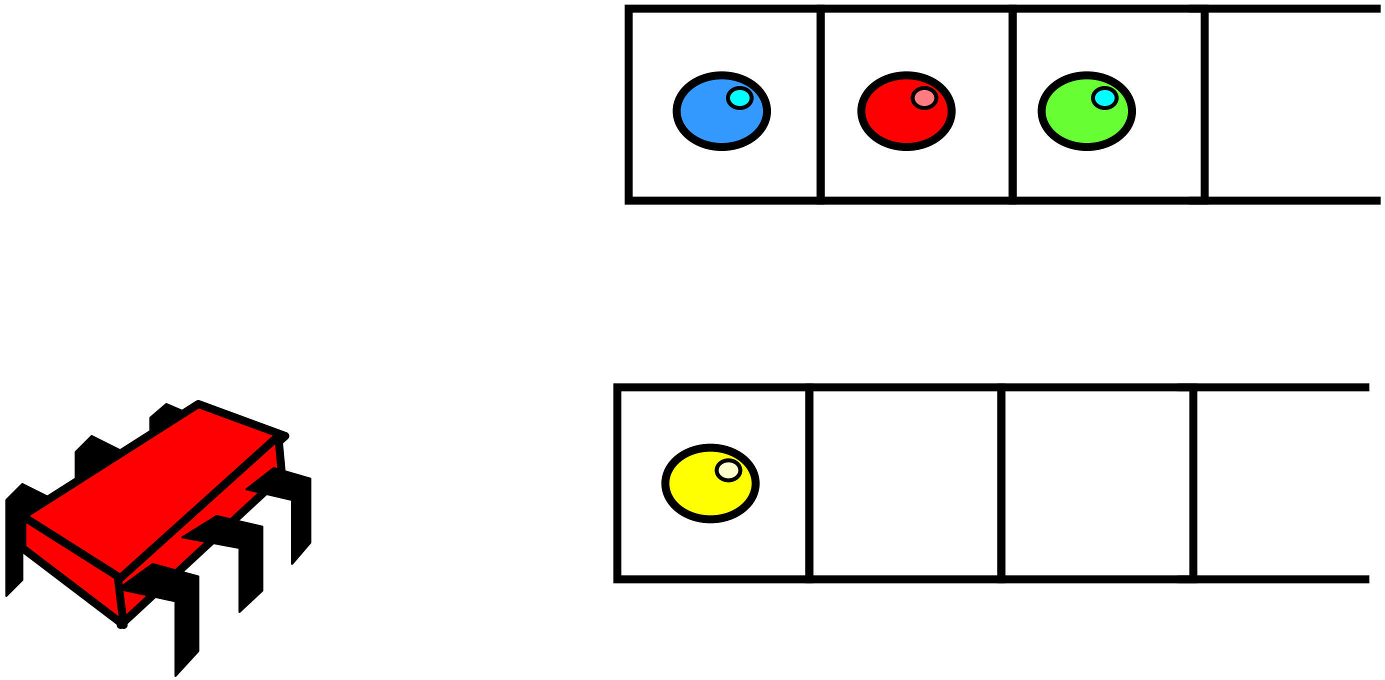
**Enclose in atomic block**

# Warning

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

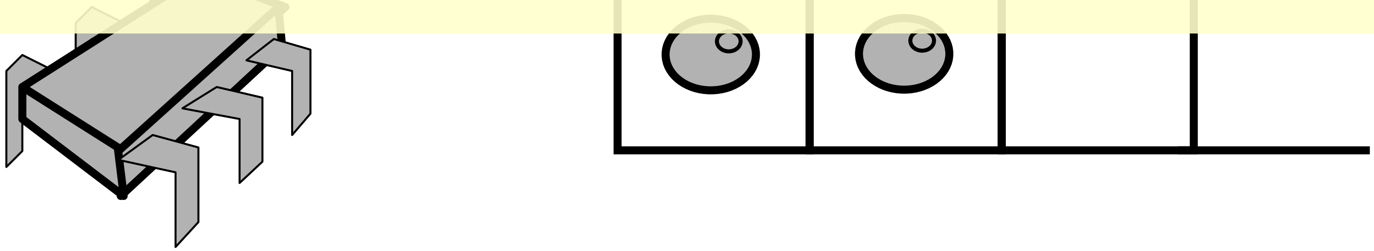
# Composition?

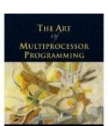


# Composition?

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

Trivial or what?



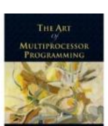


# Conditional Waiting

```
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();
}
```

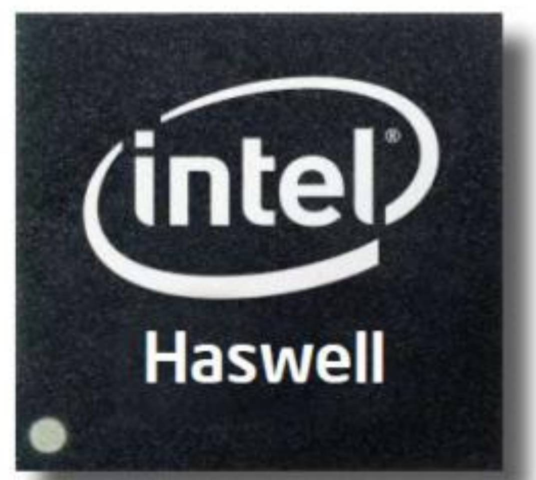
Run 1st method. If it retries

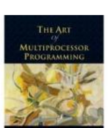
Run 2nd method. If it retries …

Entire statement retries

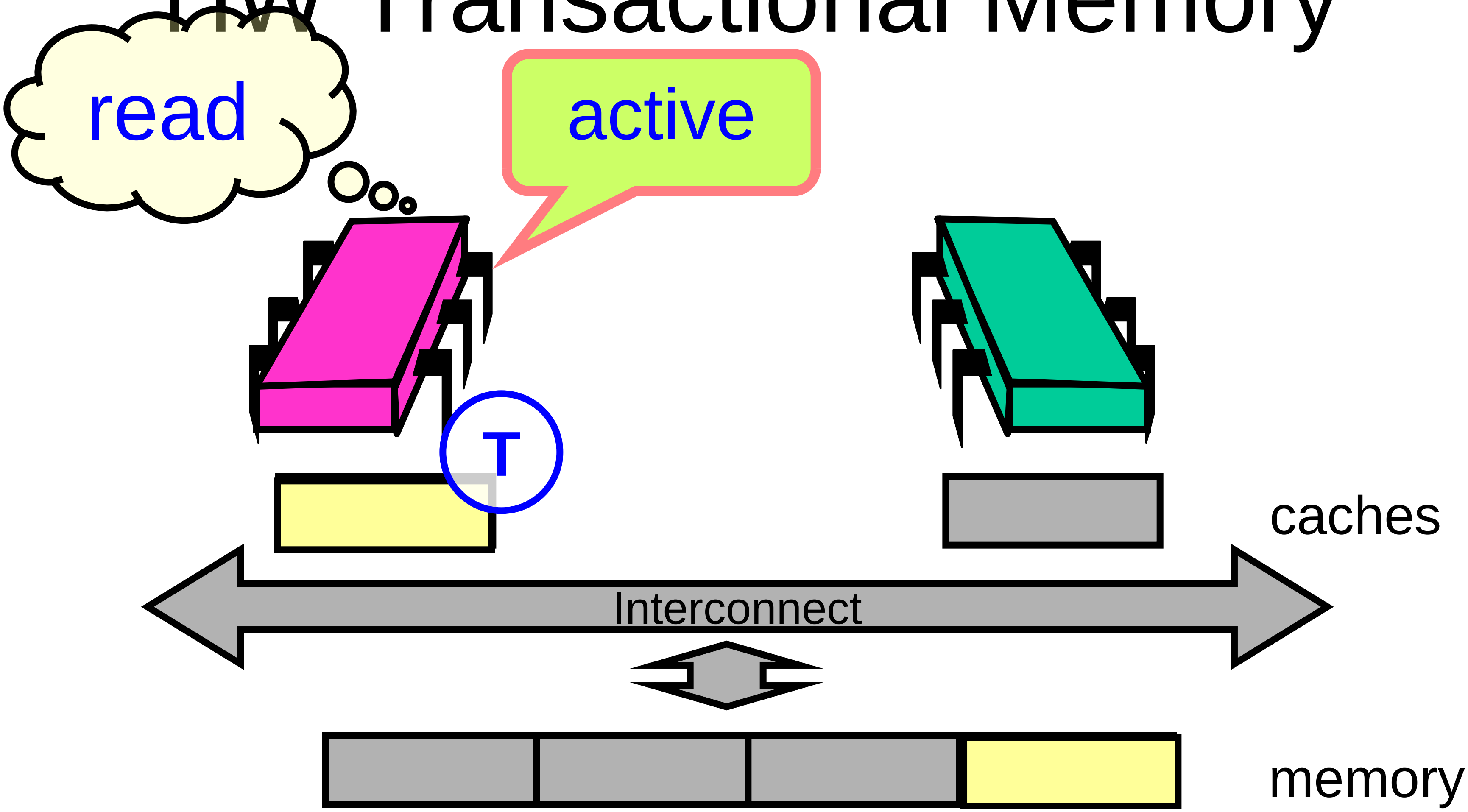
# Hardware Transactional Memory

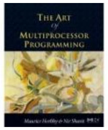
- Exploit Cache coherence
- Already almost does it
  - Invalidation
  - Consistency checking
- Speculative execution
  - Branch prediction = optimistic synch!

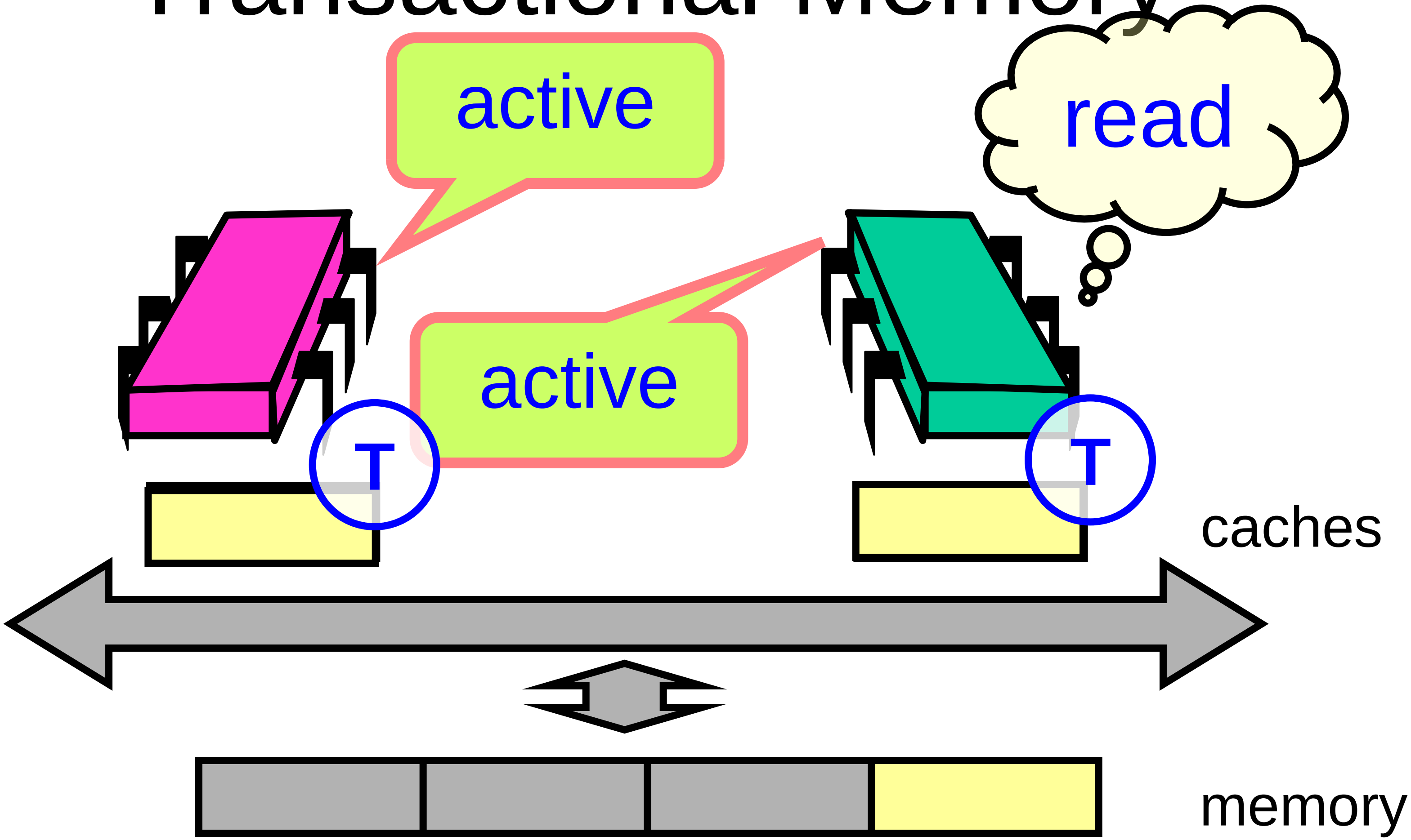



# HW Transactional Memory

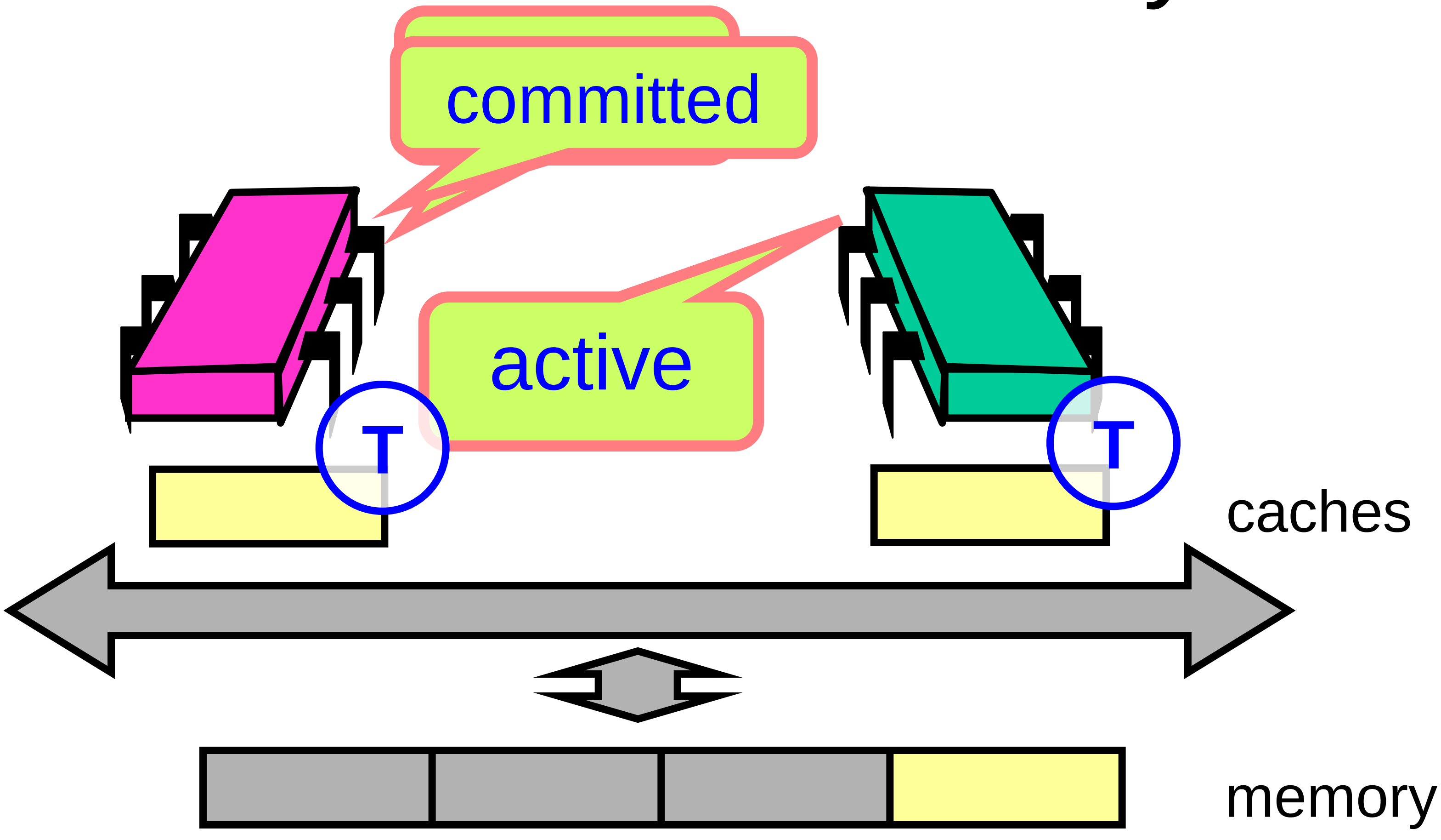


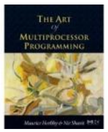



# Transactional Memory

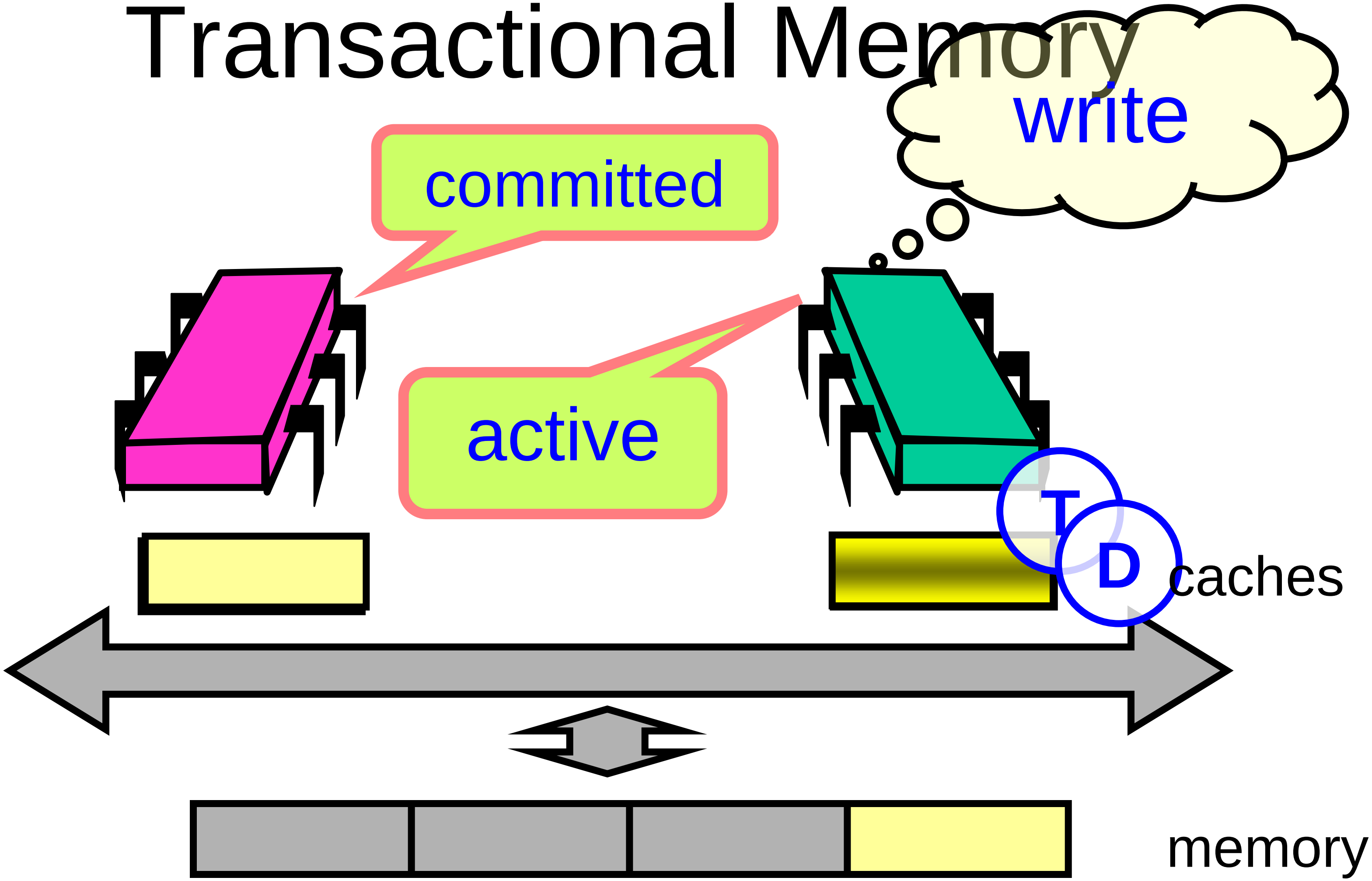


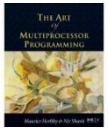
# Transactional Memory



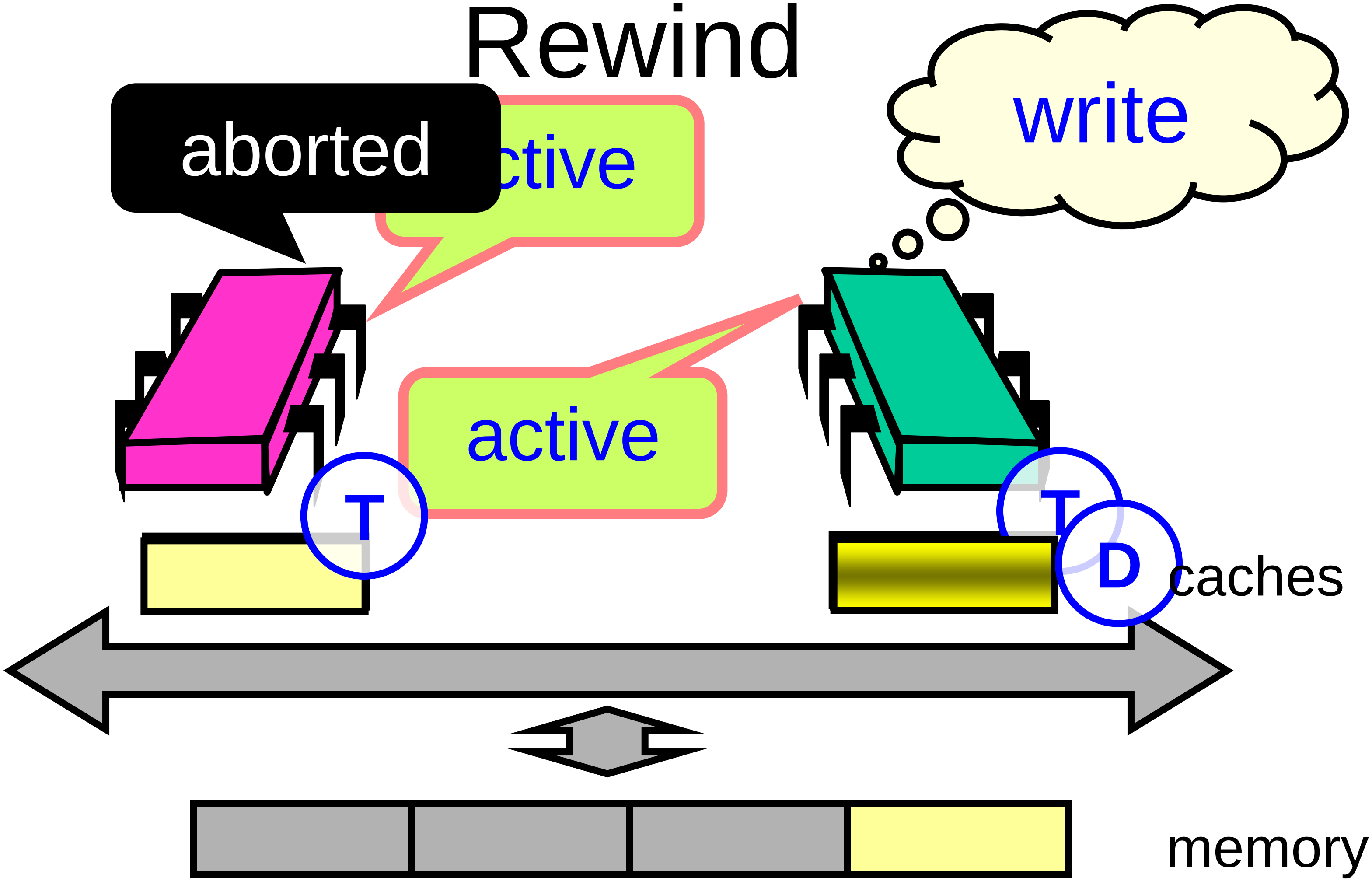



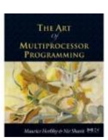
# Transactional Memory



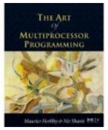
# Rewind



# Transaction Commit

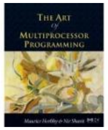
- At commit point
  - If no cache conflicts, we win.
- Mark transactional entries
  - Read-only: valid
  - Modified: dirty (eventually written back)
- That’s all, folks!
  - Except for a few details …



# 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

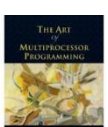
- Ideal for lock-free data structures

# 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
  - Try again in software?



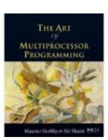
# Composi

Locks don't compose, transactions do.

Composition necessary for Software Engineering.

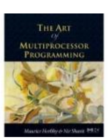
But practical HTM doesn't ***really*** support composition!

Why we need STM

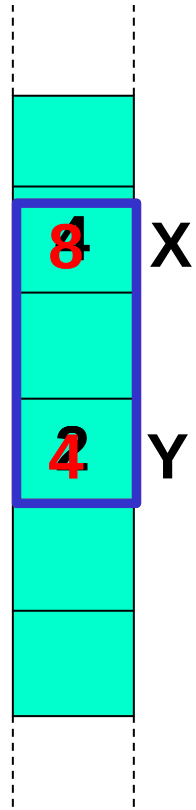


# Transactional Consistency

- 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

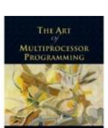



Application
Memory

*Invariant x = 2y*

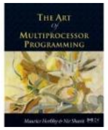
*Transaction A:*
*Write x*
*Write y*

*Transaction B:*
*Read x*
*Read y*
*Compute z = 1/(x-y) = 1/2*



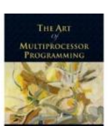
# A Simple Lock-Based STM

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

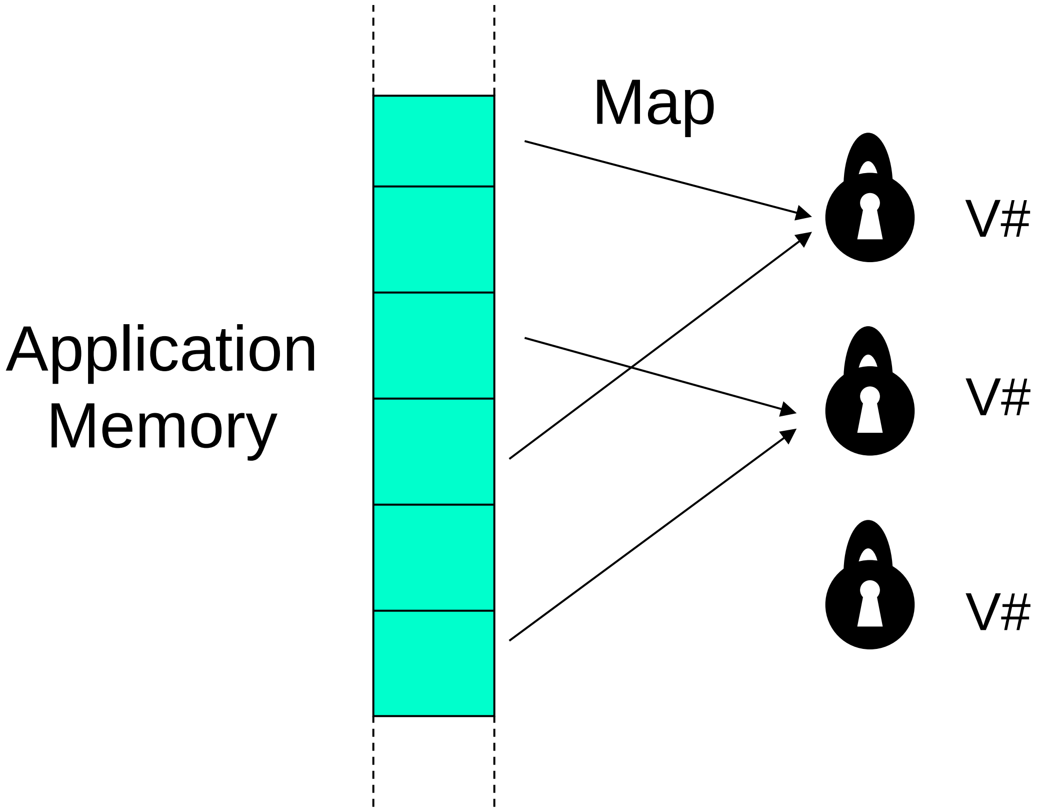


# Synchronization

- 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

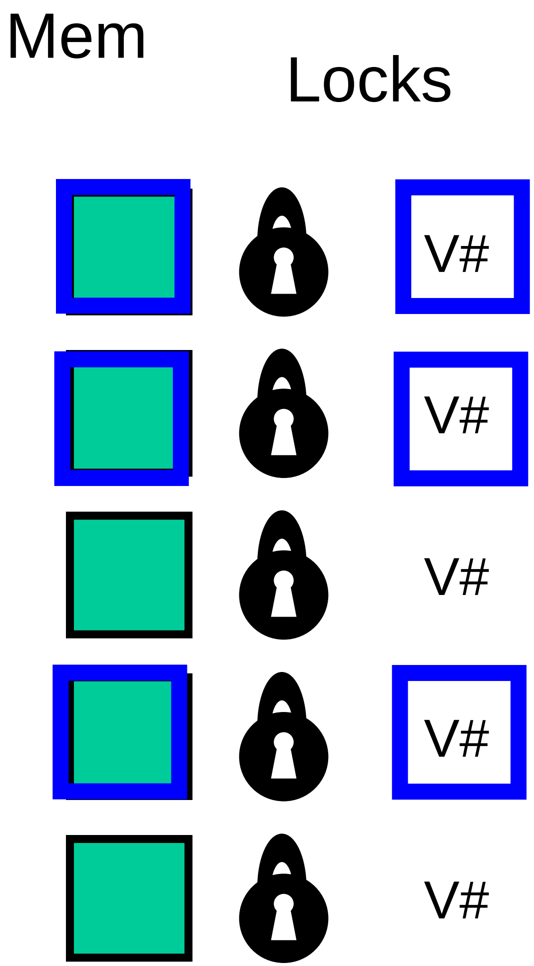


# STM: Transactional Locking

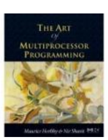



Array of version #s & locks

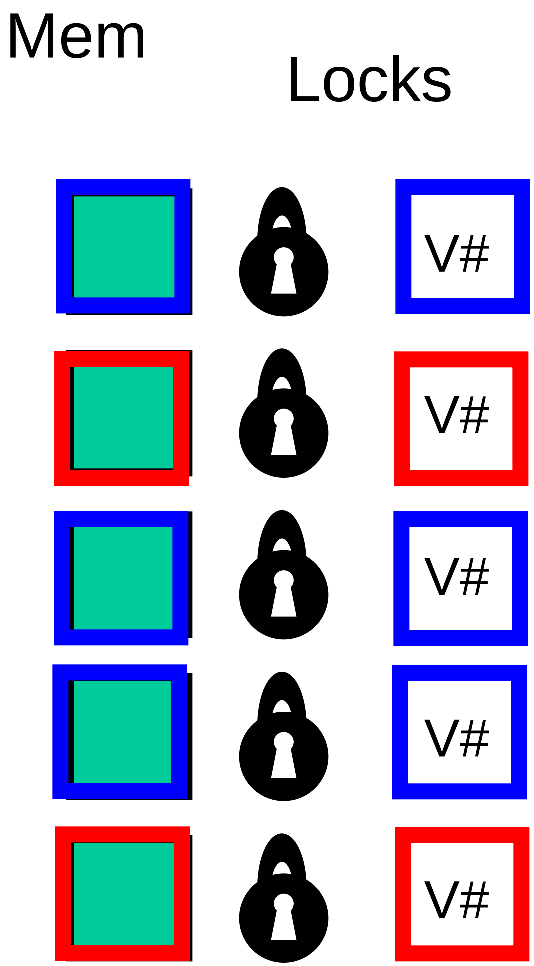
# Reading an Object



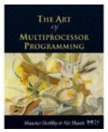

Add version numbers & values to read set



# To Write an Object



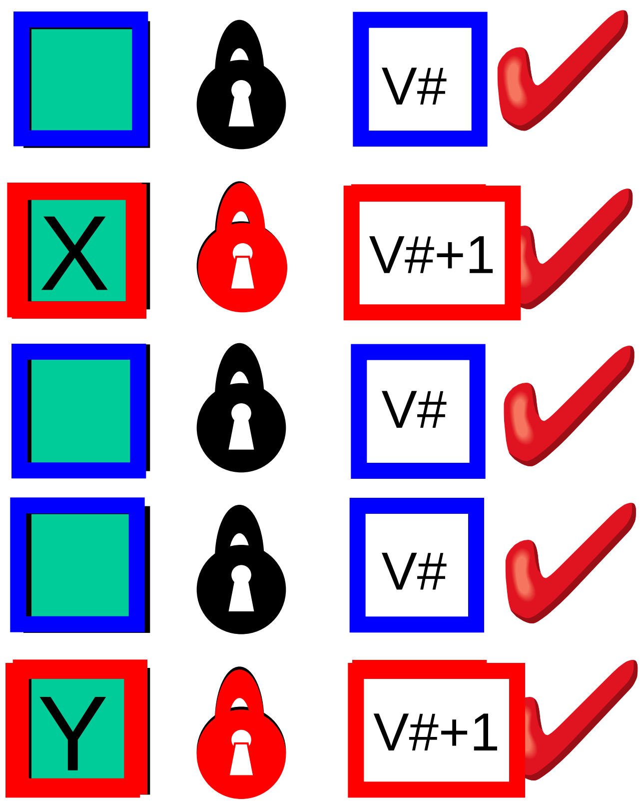

Add version numbers & new values to write set



# To Commit

Mem

Locks




Acquire write locks

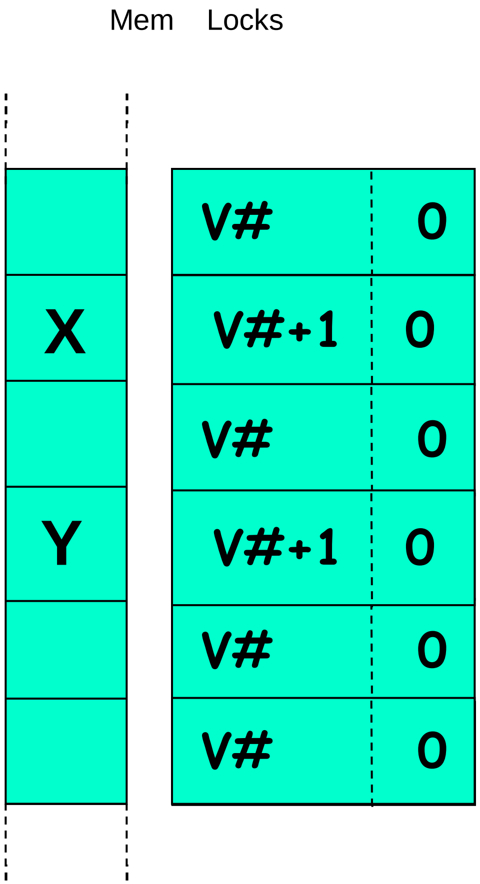
Check version numbers unchanged

Install new values

Increment version numbers

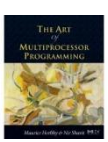
Unlock.

# Encounter Order Locking (Undo Log)

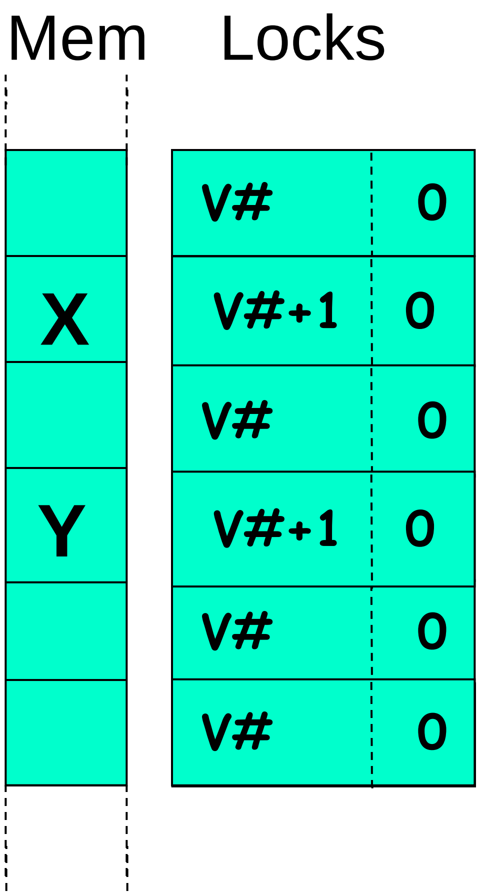



1. To Read: load lock + location
2. Check unlocked add to Read-Set
3. To Write: lock location, store value
4. Add old value to undo-set
5. Validate read-set v#'s unchanged
6. Release each lock with v#+1

Quick read of values freshly written by the reading transaction

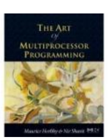


# Commit Time Locking (Write Buff)

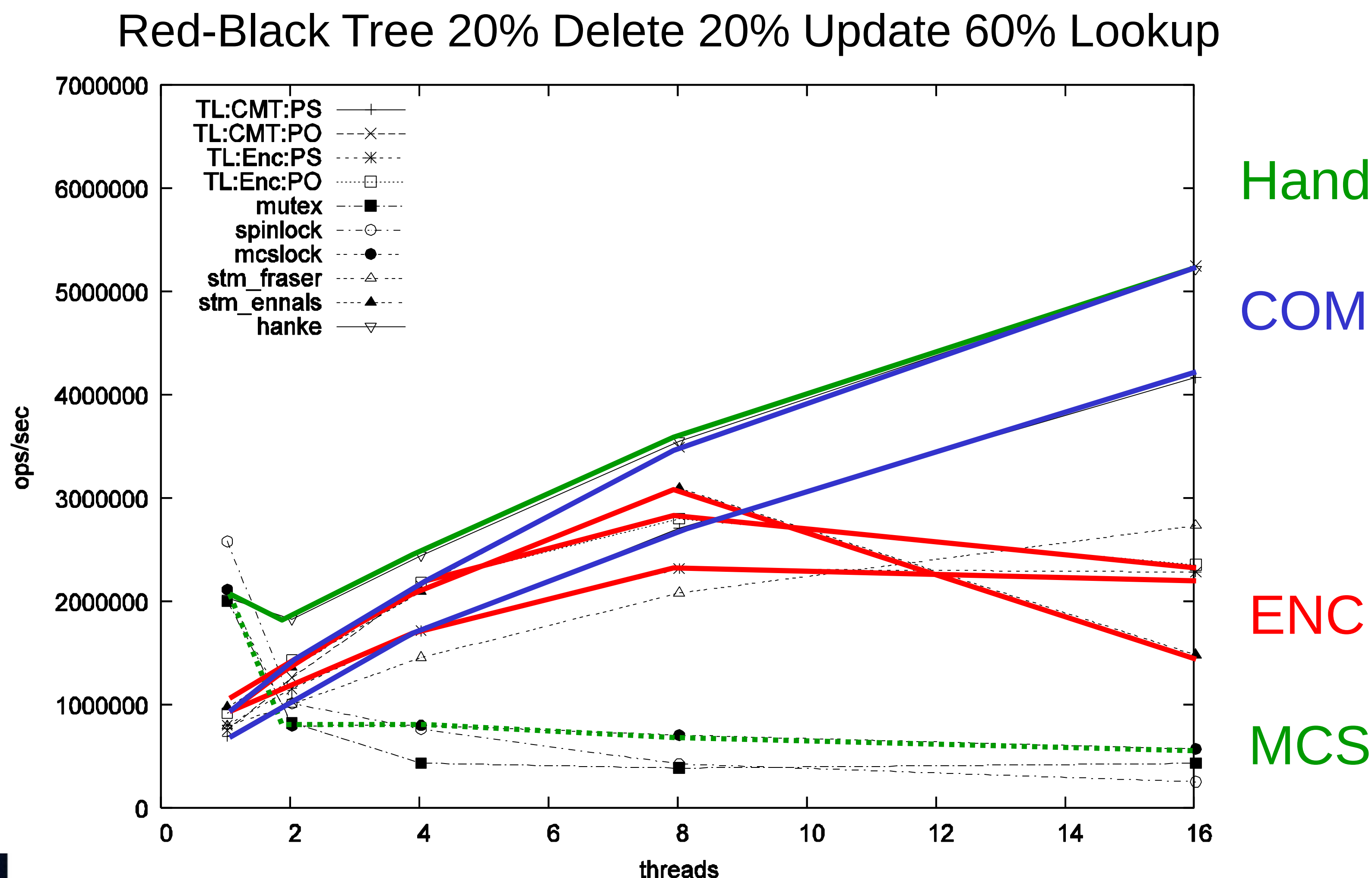



1. To Read: load lock + location
2. Location in write-set? (Bloom Filter)
3. Check unlocked add to Read-Set
4. To Write: add value to write set
5. Acquire Locks
6. Validate read/write v#' s unchanged
7. Release each lock with v#+1

Hold locks for very short duration

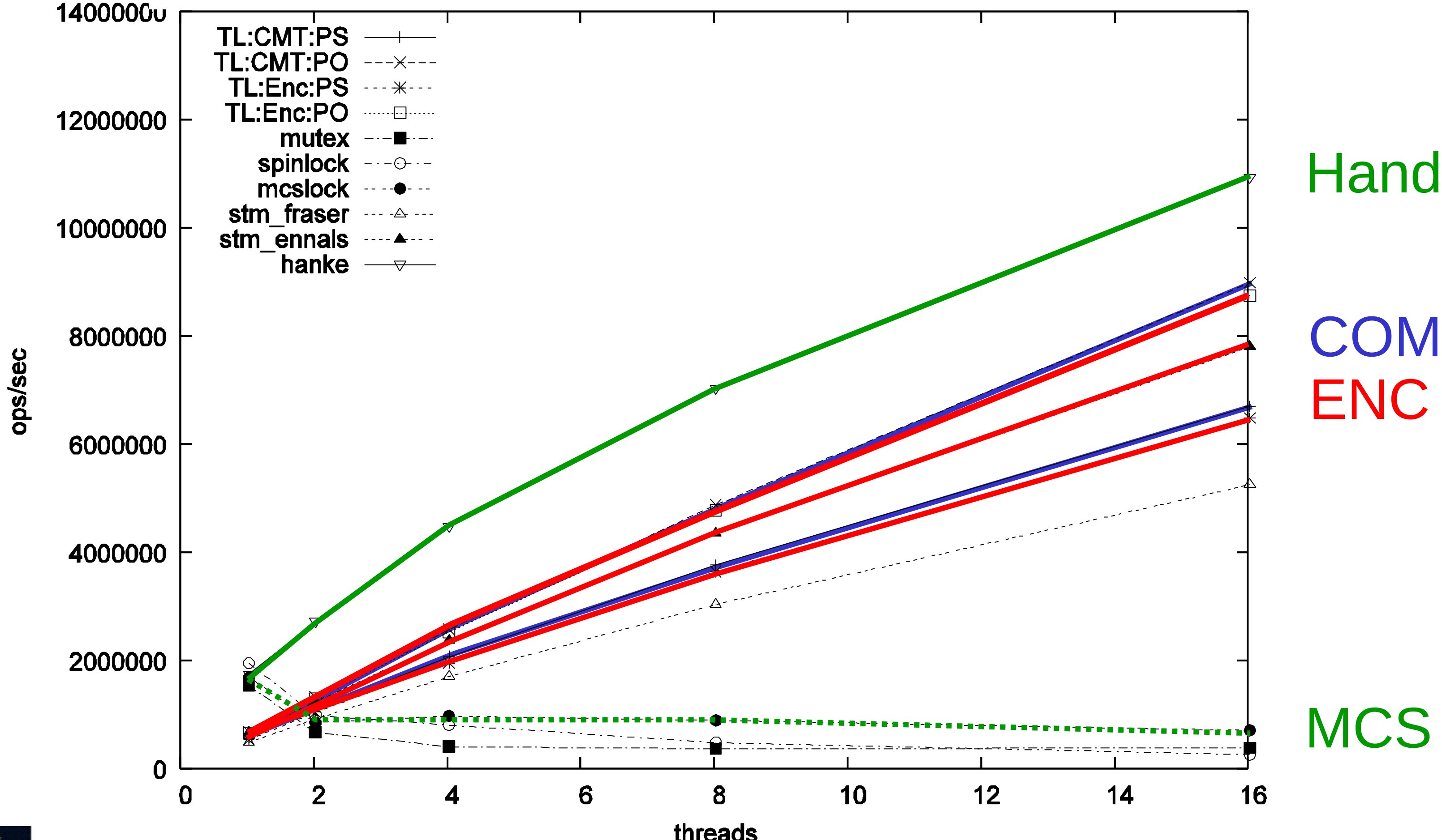


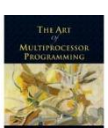
# COM vs. ENC High Load



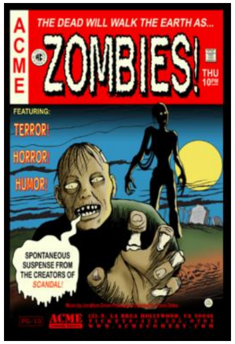
# COM vs. ENC Low Load

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

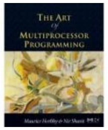


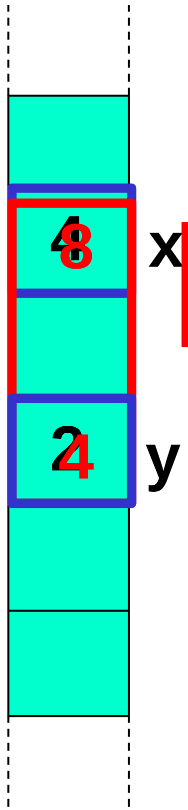

# Problem: Internal Inconsistency




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



# Internal Consistency



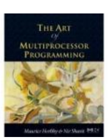

Invariant: x = 2y

Transaction A: reads x = 4

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

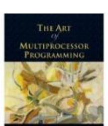
Transaction A: (zombie)
reads y = 4
computes 1/(x-y)

Divide by zero FAIL!

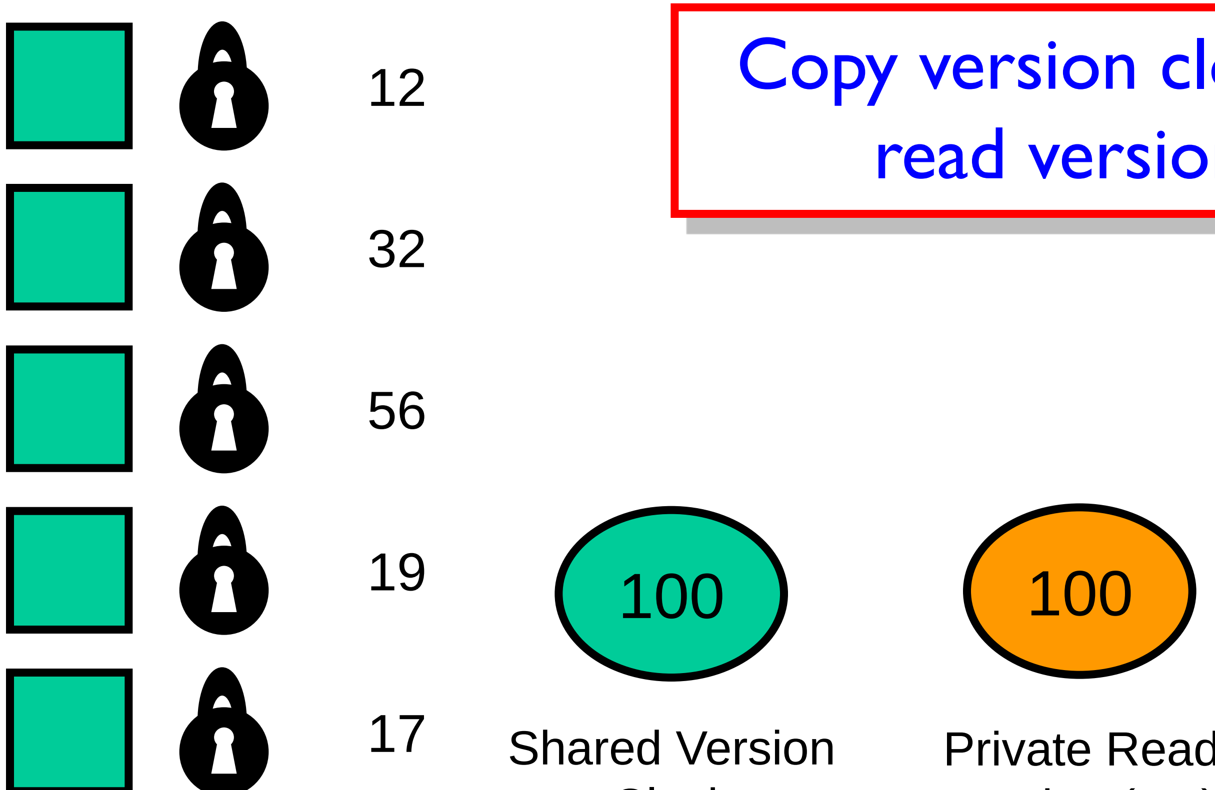


# Solution: The Global Clock (The TL2 Algorithm)

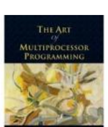
- Have one shared global clock
- Incremented by (small subset of) writing transactions
- Read by all transactions
- Used to validate that state worked on is always consistent



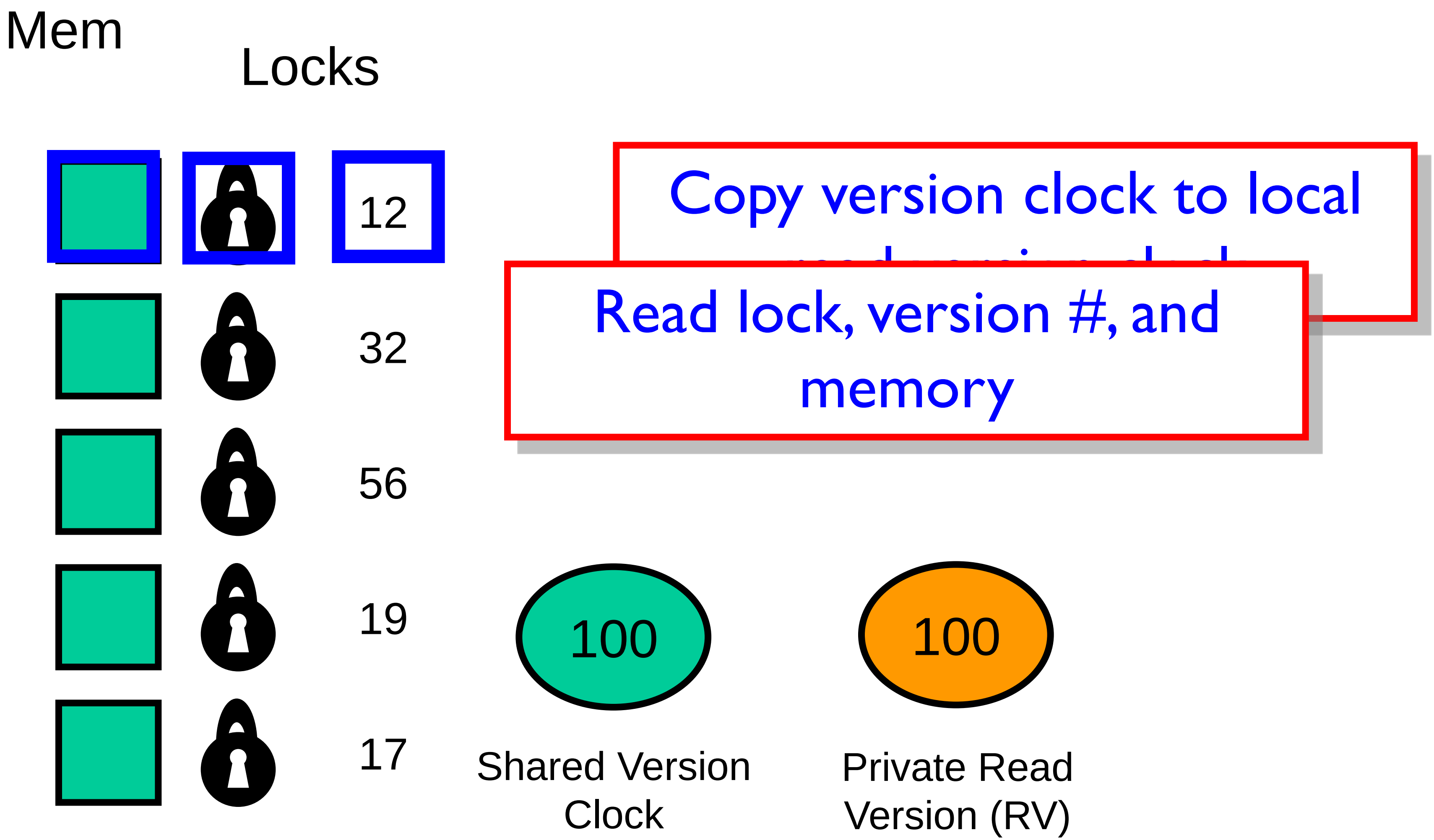
# Read-Only Transactions

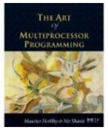



Copy version clock to local read version clock

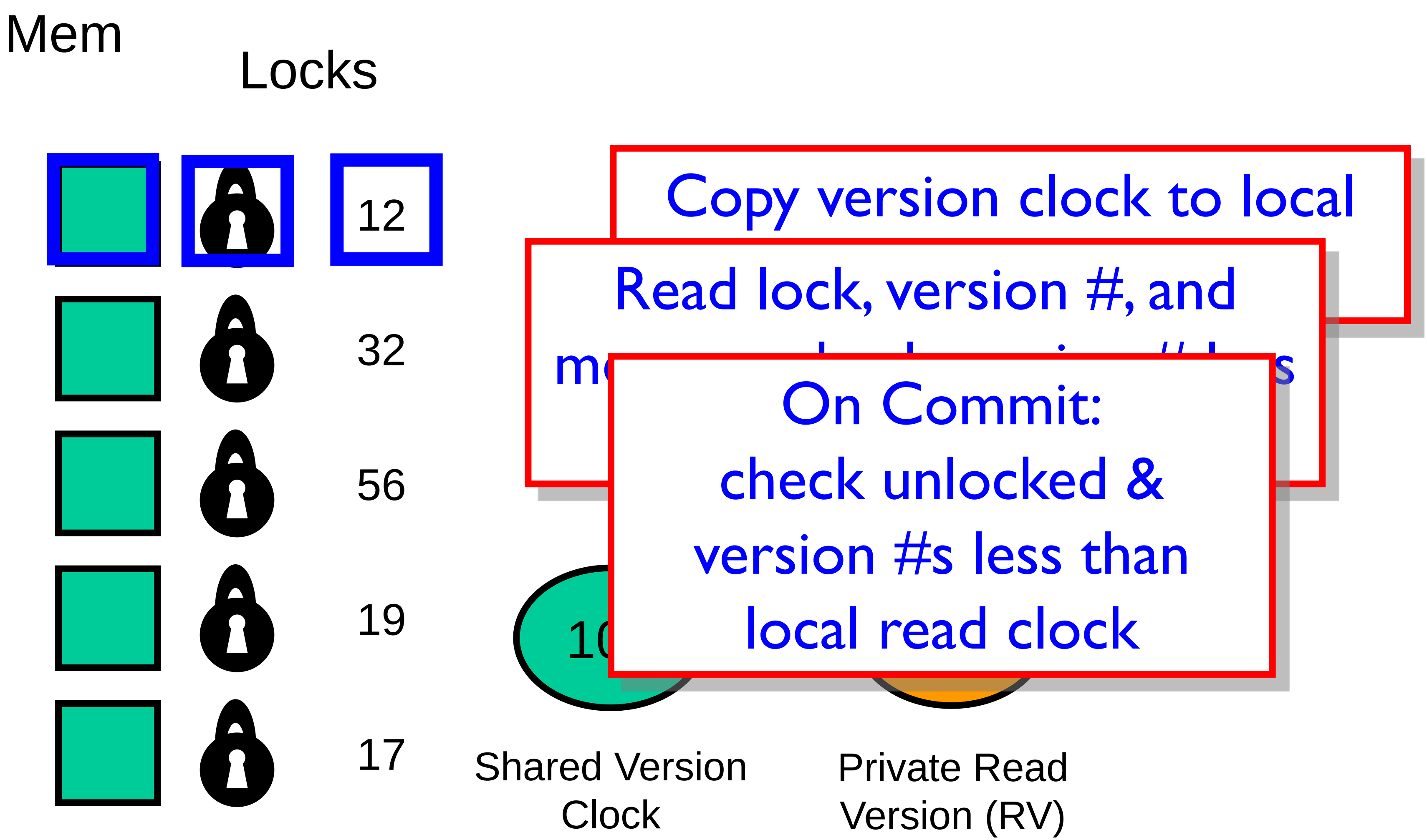


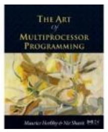
# Read-Only Transactions



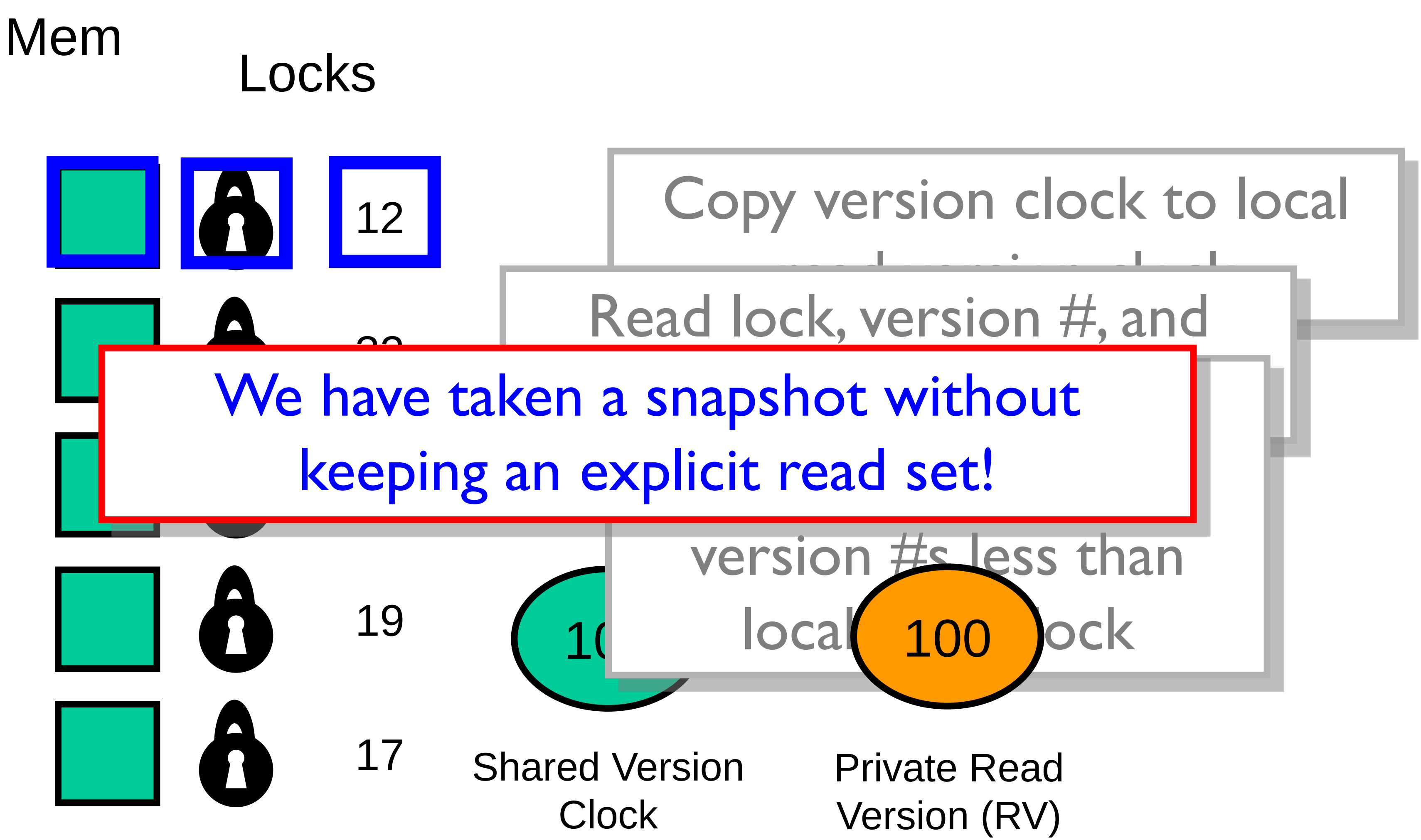



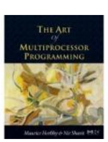
# Read-Only Transactions



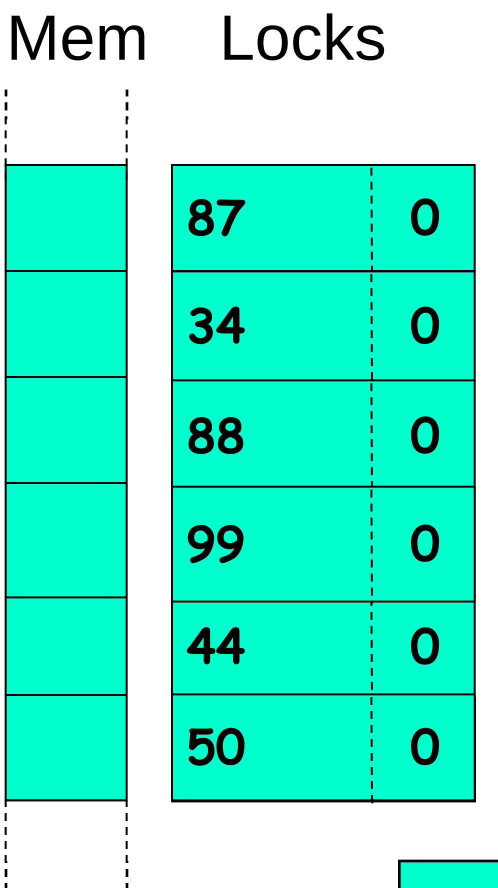



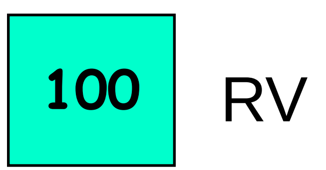
# Read-Only Transactions



# Example Execution: Read Only Trans

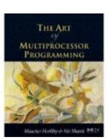


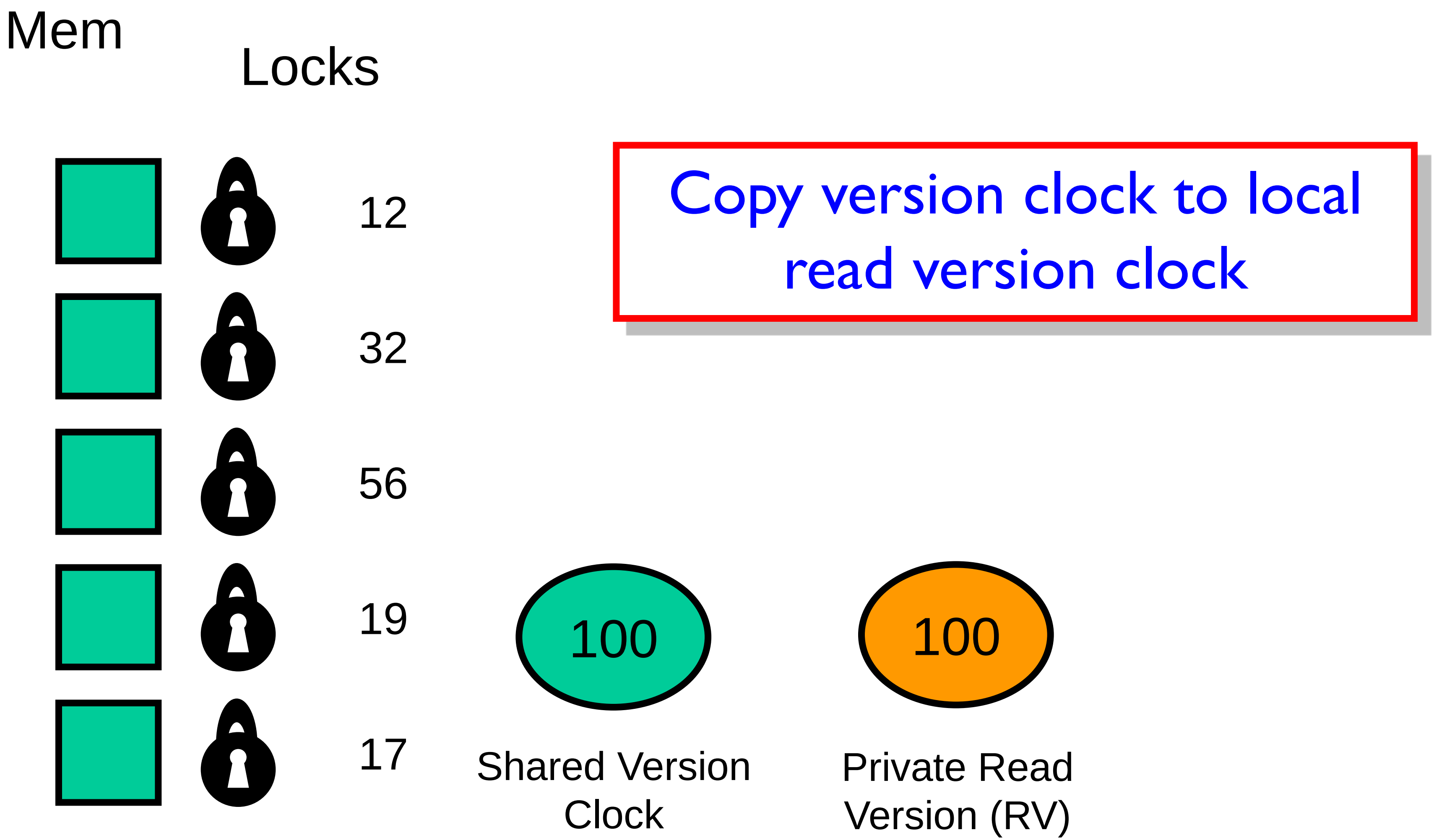

100 Shared Version Clock

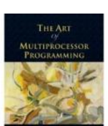
1. RV ← Shared Version Clock
2. On Read: read lock, read mem, read lock: check unlocked, unchanged, and v# <= RV
3. Commit.

Reads form a snapshot of memory. No read set!

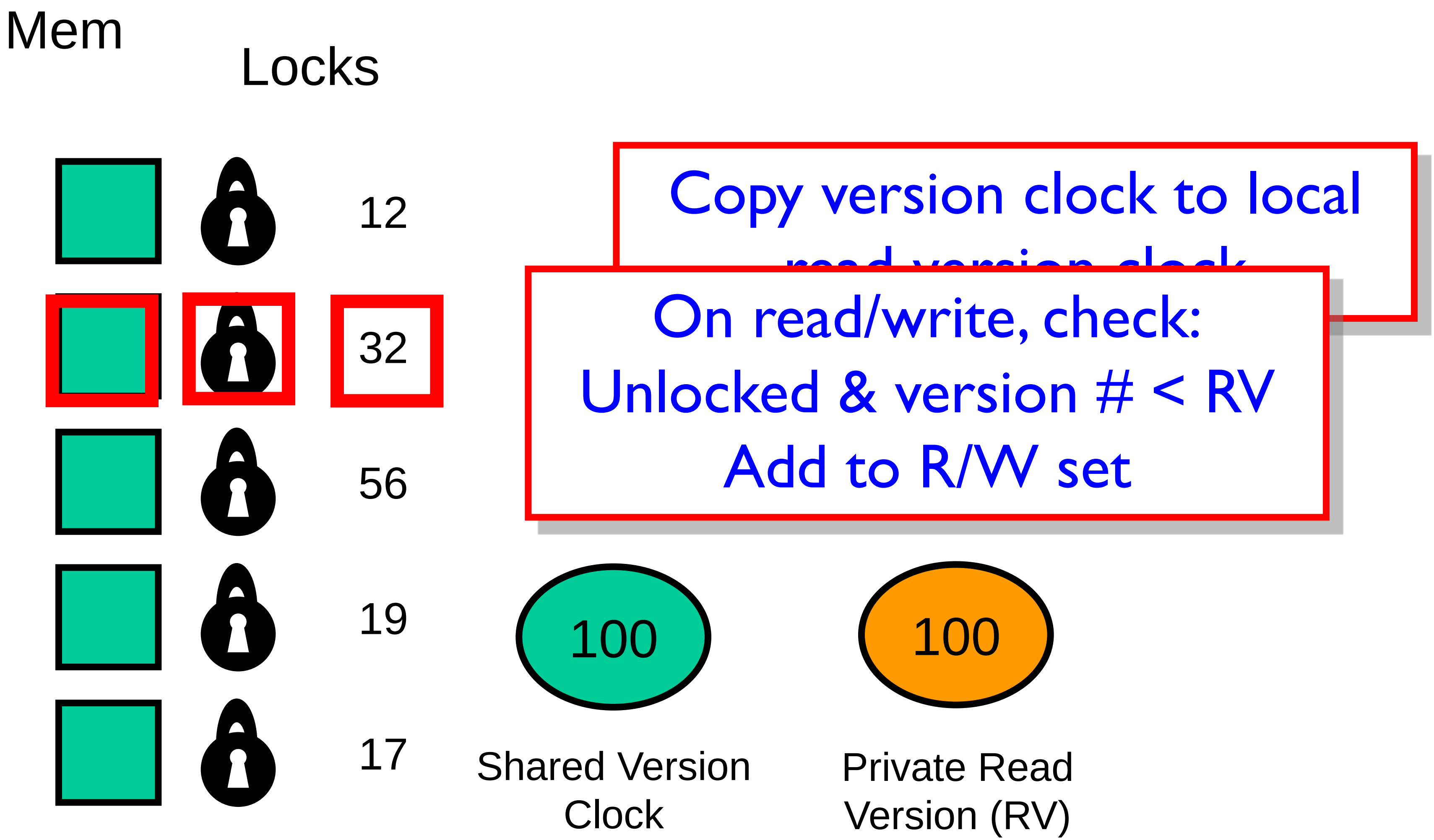


# Ordinary (Writing) Transactions

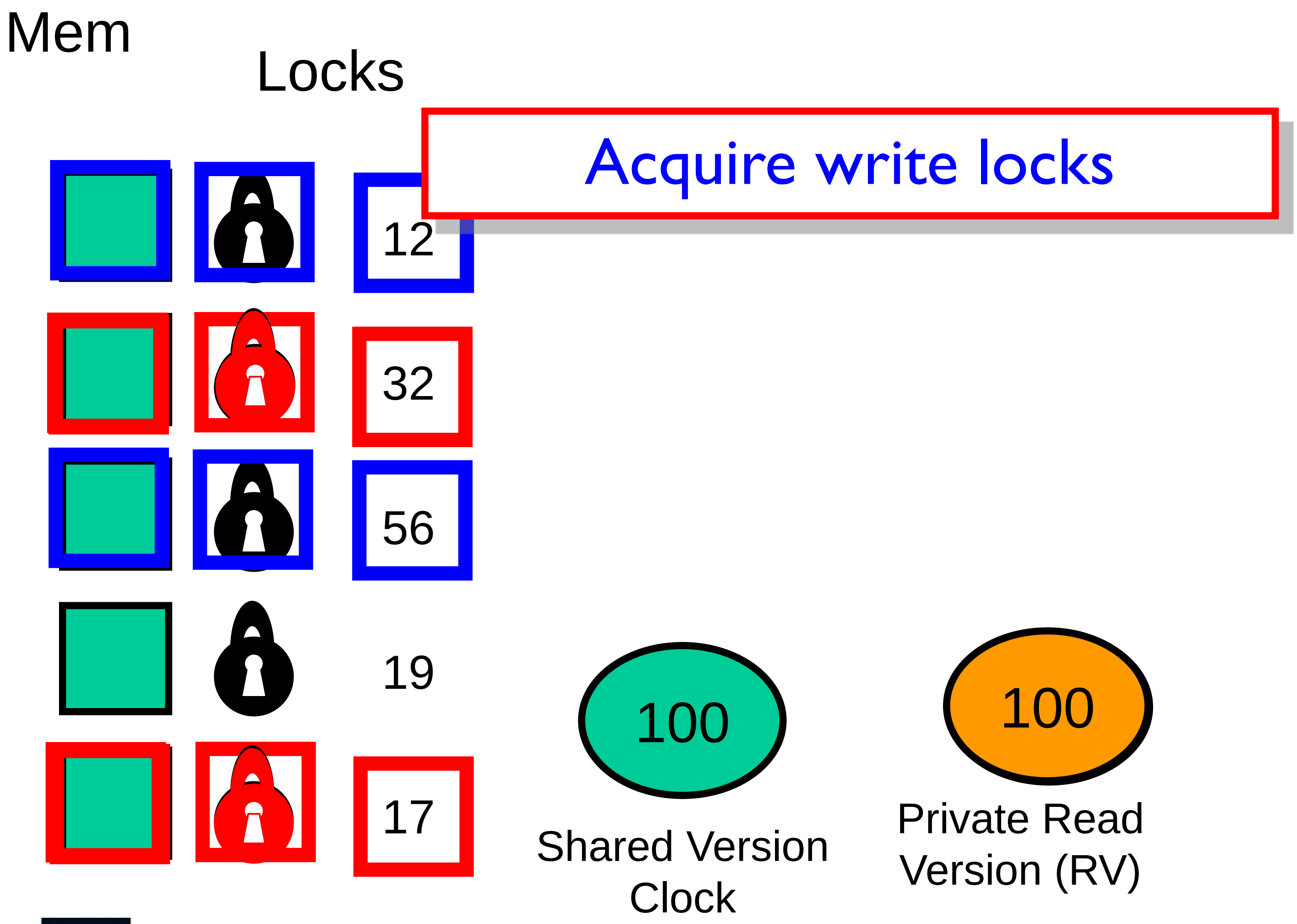


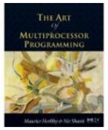
# Ordinary Transactions



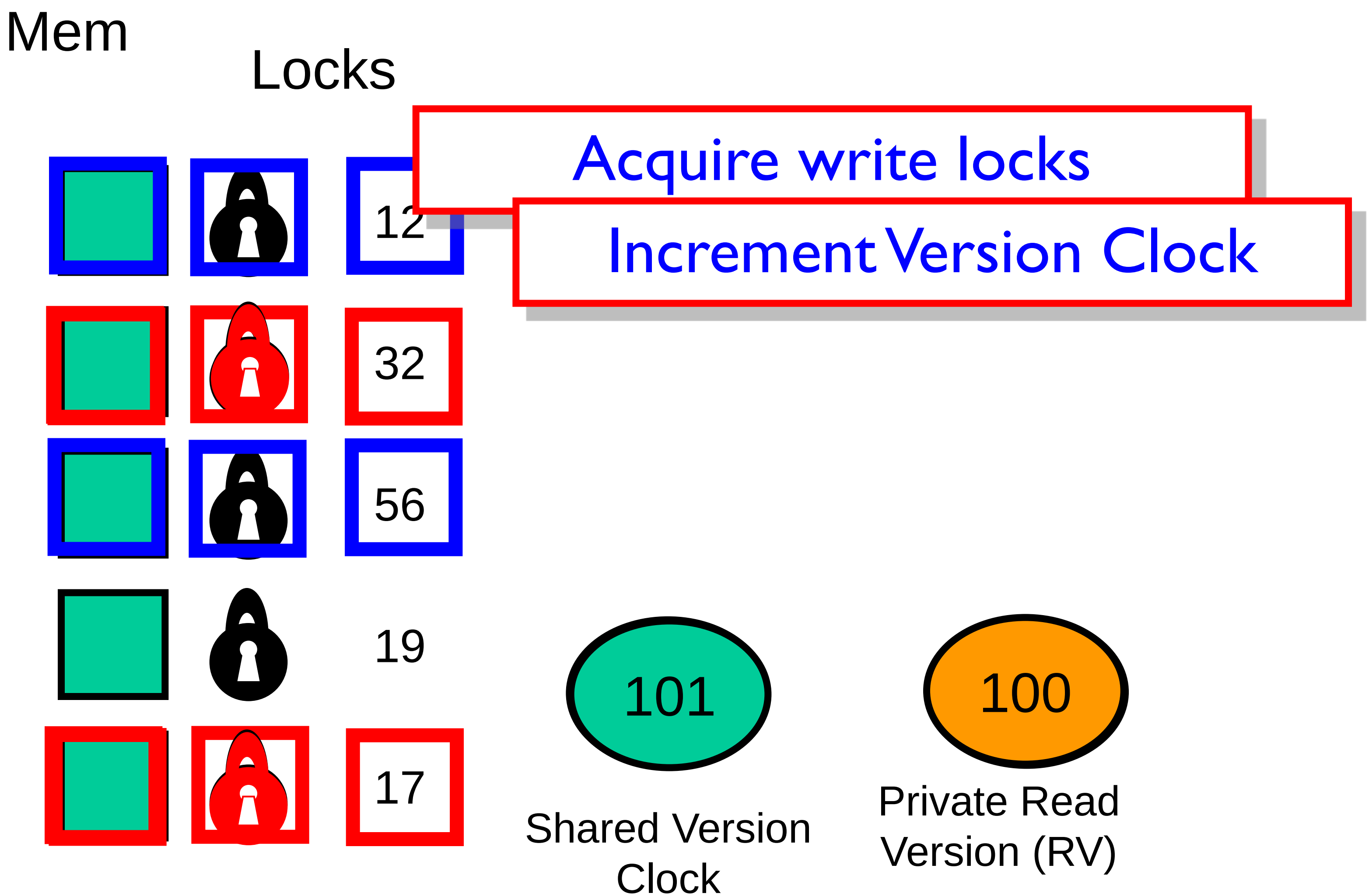
# On Commit



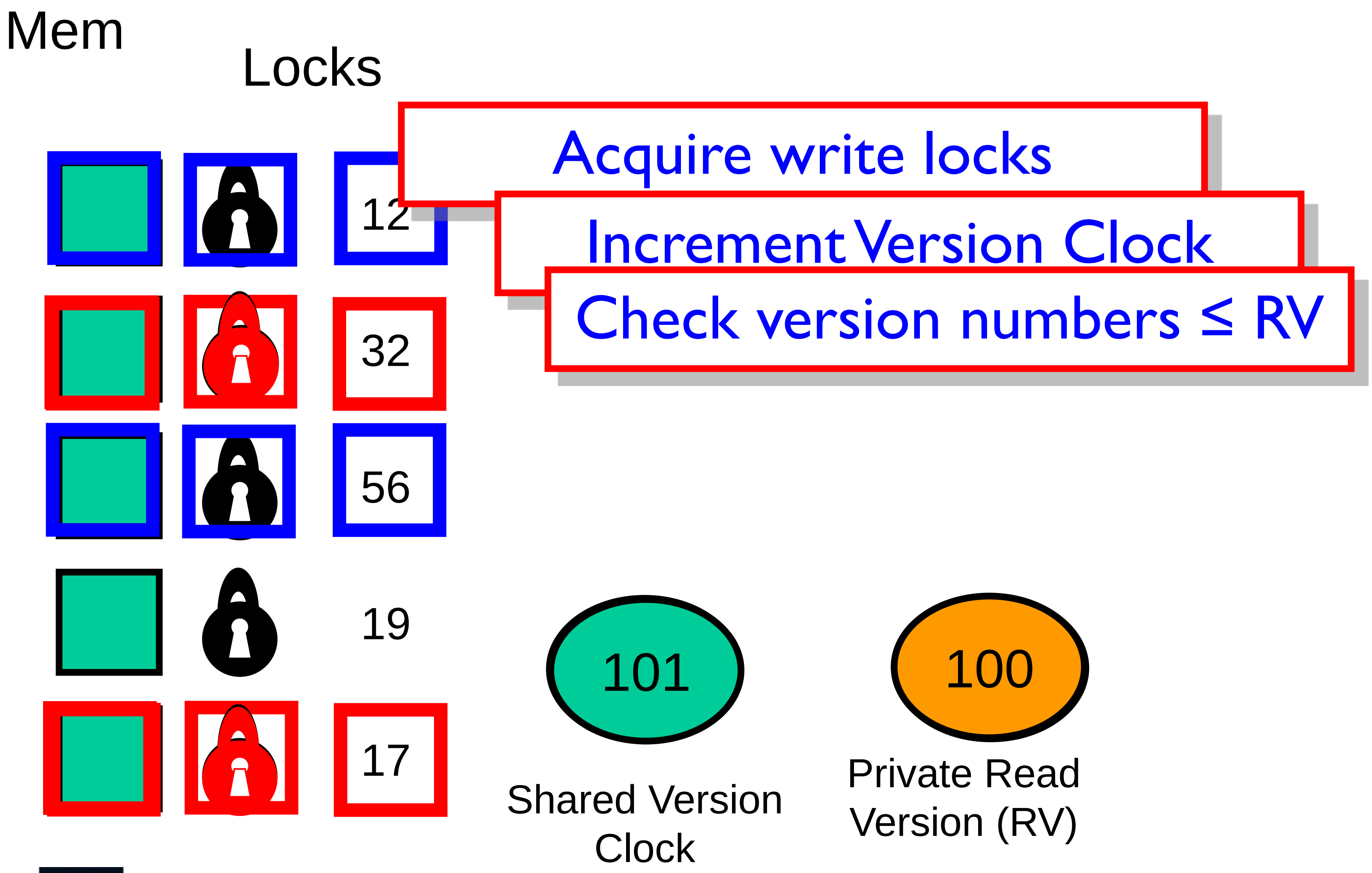
Mem
Locks
Acquire write locks
12
32
56
19
17
100
Shared Version Clock
100
Private Read Version (RV)

# On Commit

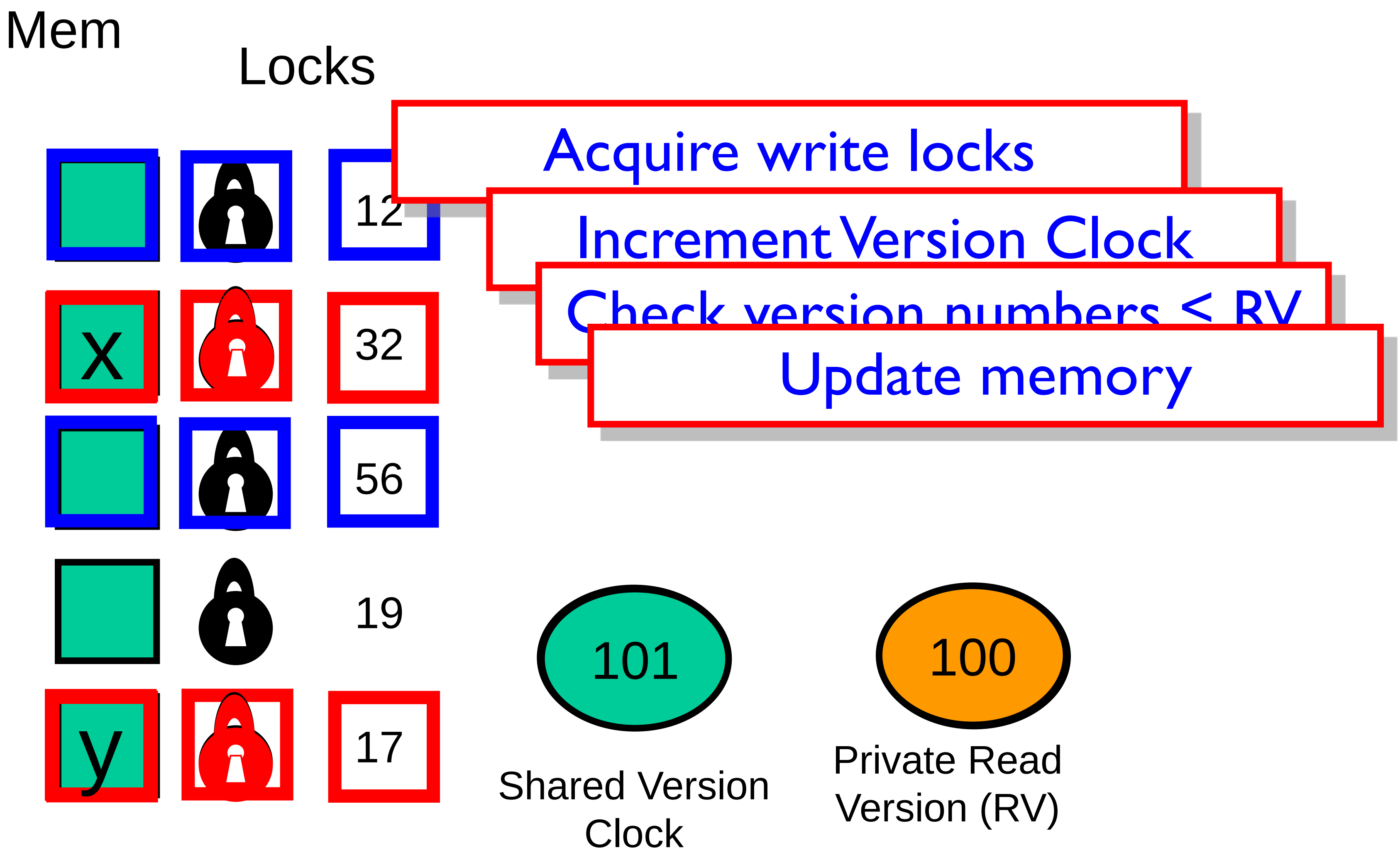

Mem
Locks
Acquire write locks
Increment Version Clock
12
32
56
19
17
101
100
Shared Version Clock
Private Read Version (RV)

# On Commit



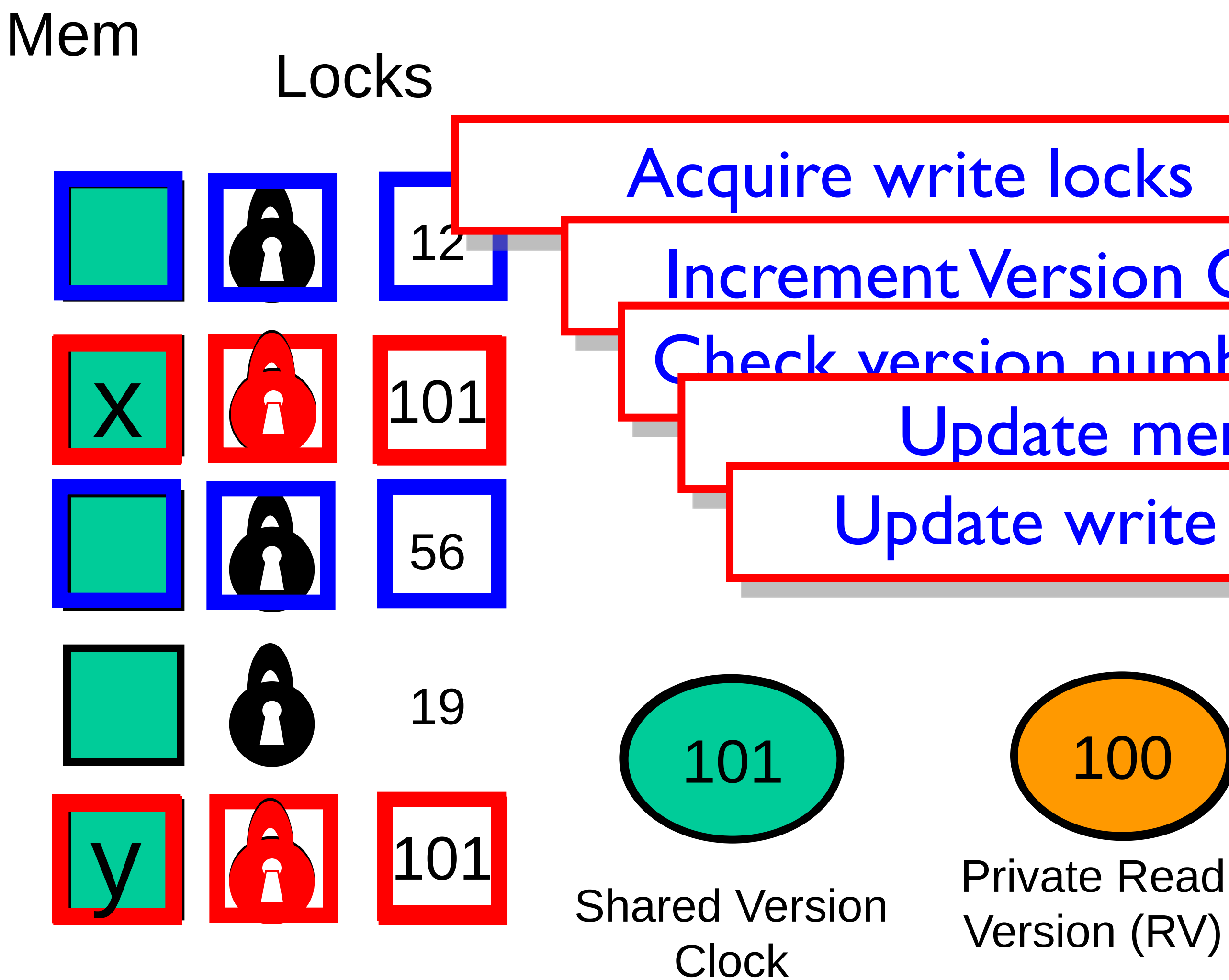
Mem
Locks
12
32
56
19
17
Acquire write locks
Increment Version Clock
Check version numbers ≤ RV
101
100
Shared Version Clock
Private Read Version (RV)

# On Commit

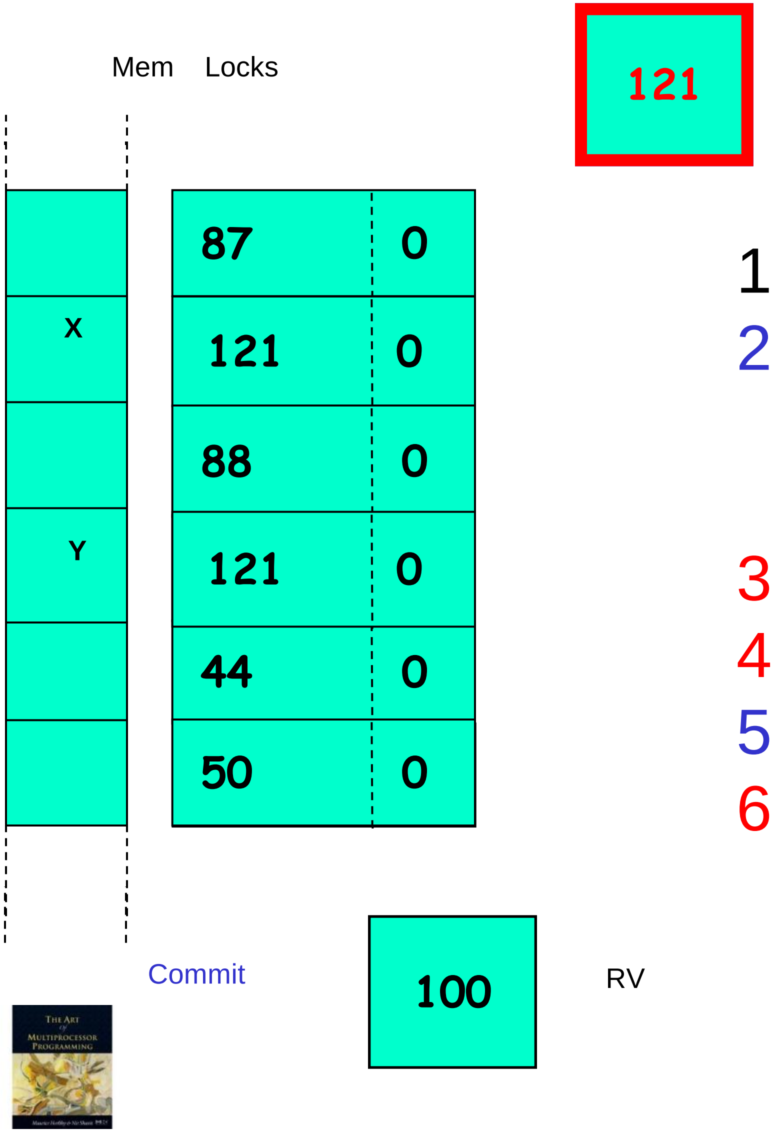

Mem
Locks
12
x
32
56
19
y
17
Acquire write locks
Increment Version Clock
Check version numbers < RV
Update memory
101
100
Shared Version Clock
Private Read Version (RV)

# On Commit

- Acquire write locks
- Increment Version Clock
- Check version numbers < RV
- Update memory
- Update write version #s



# Example: Writing Trans



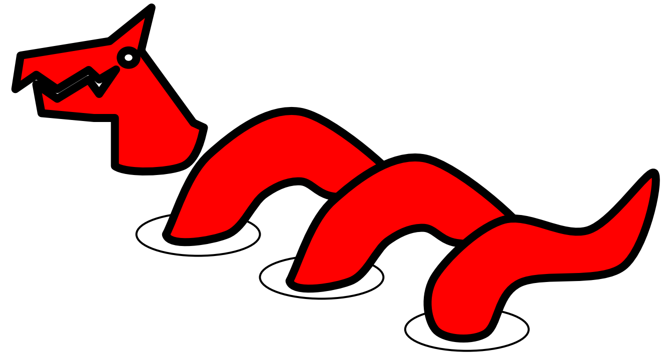

Shared Version Clock

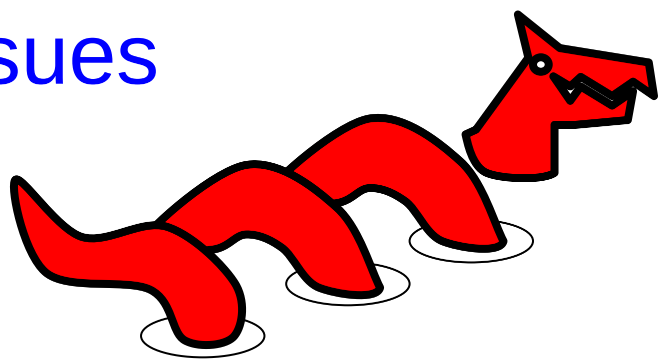
1. RV ← Shared Version Clock
2. On Read/Write: check unlocked and v# <= RV then add to Read/Write-Set
3. Acquire Locks
4. WV = F&I(VClock)
5. Validate each v# <= RV
6. Release locks with v# ← WV

Reads+Inc+Writes =serializable

# TM Design Issues

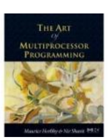
- Implementation choices
- Language design issues
- Semantic issues





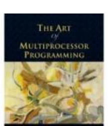
# Granularity

- 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



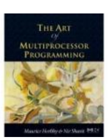
# Direct/Deferred Update

- *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 …

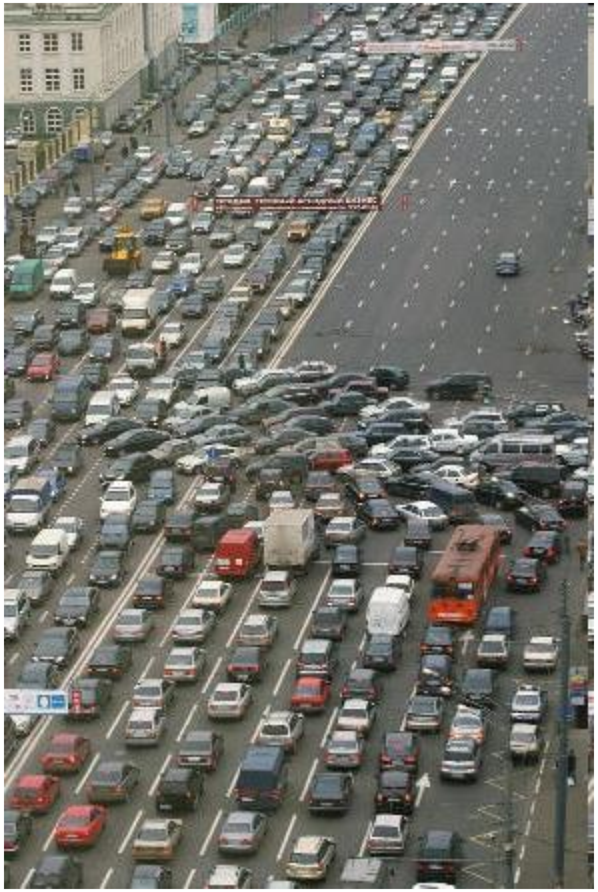


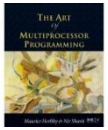
# Conflict Detection

- 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





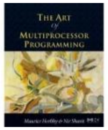
# Contention Manager Strategies

- 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

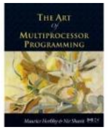


# I/O & System Calls

- 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

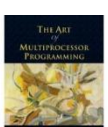
```
int i = 0;
try {
  atomic {
    i++;
    node = new Node();
  }
} catch (Exception e) {
  print(i);
}
```



# Exceptions

Throws OutOfMemoryException!

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

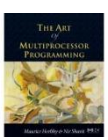


# Exceptions

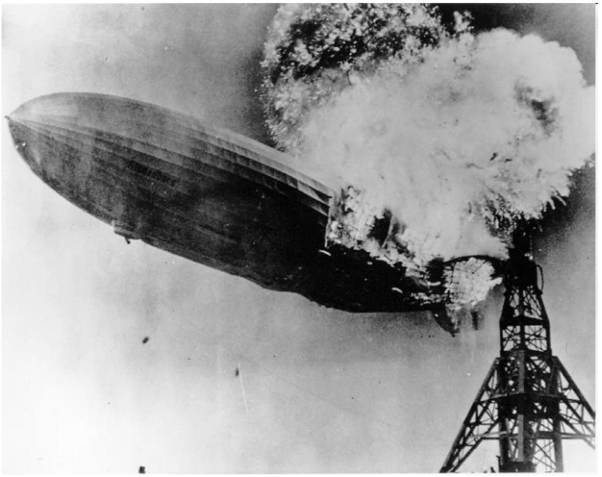
Throws OutOfMemoryException!

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

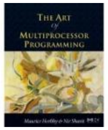
What is printed?



# Unhandled Exceptions



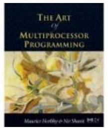
- Aborts transaction
  - Preserves invariants
  - Safer
- Commits transaction
  - Like locking semantics
  - What if exception object refers to values modified in transaction?



# Nested Transactions

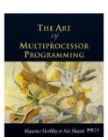
```
atomic void foo() {
  bar();
}

atomic void bar() {
 …
}
```

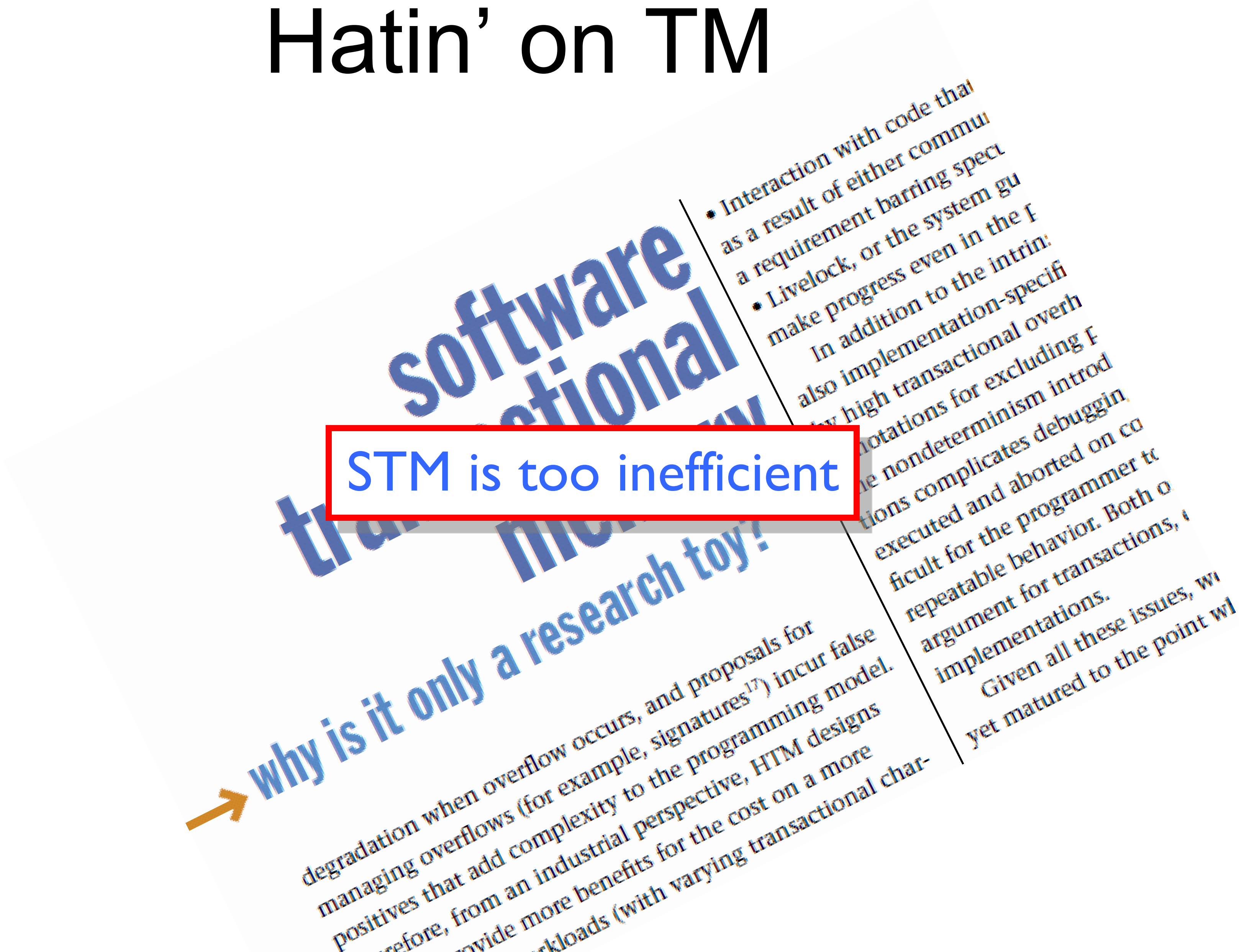


# Nested Transactions

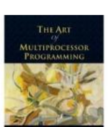
- Needed for modularity
  - Who knew that cosine() contained a transaction?
- Flat nesting
  - If child aborts, so does parent
- First-class nesting
  - If child aborts, partial rollback of child only



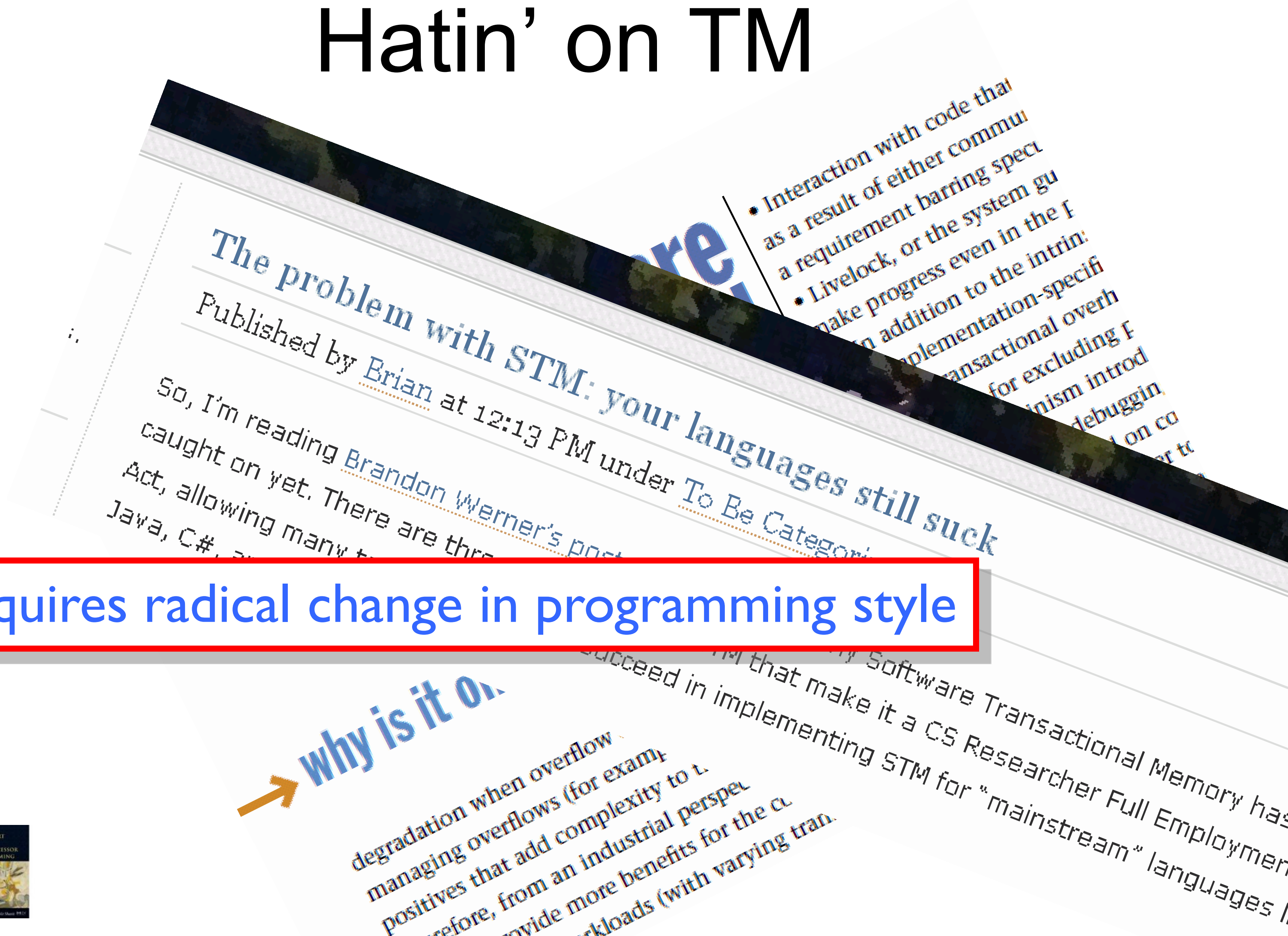
# Hatin' on TM



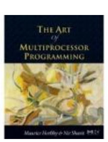

STM is too inefficient



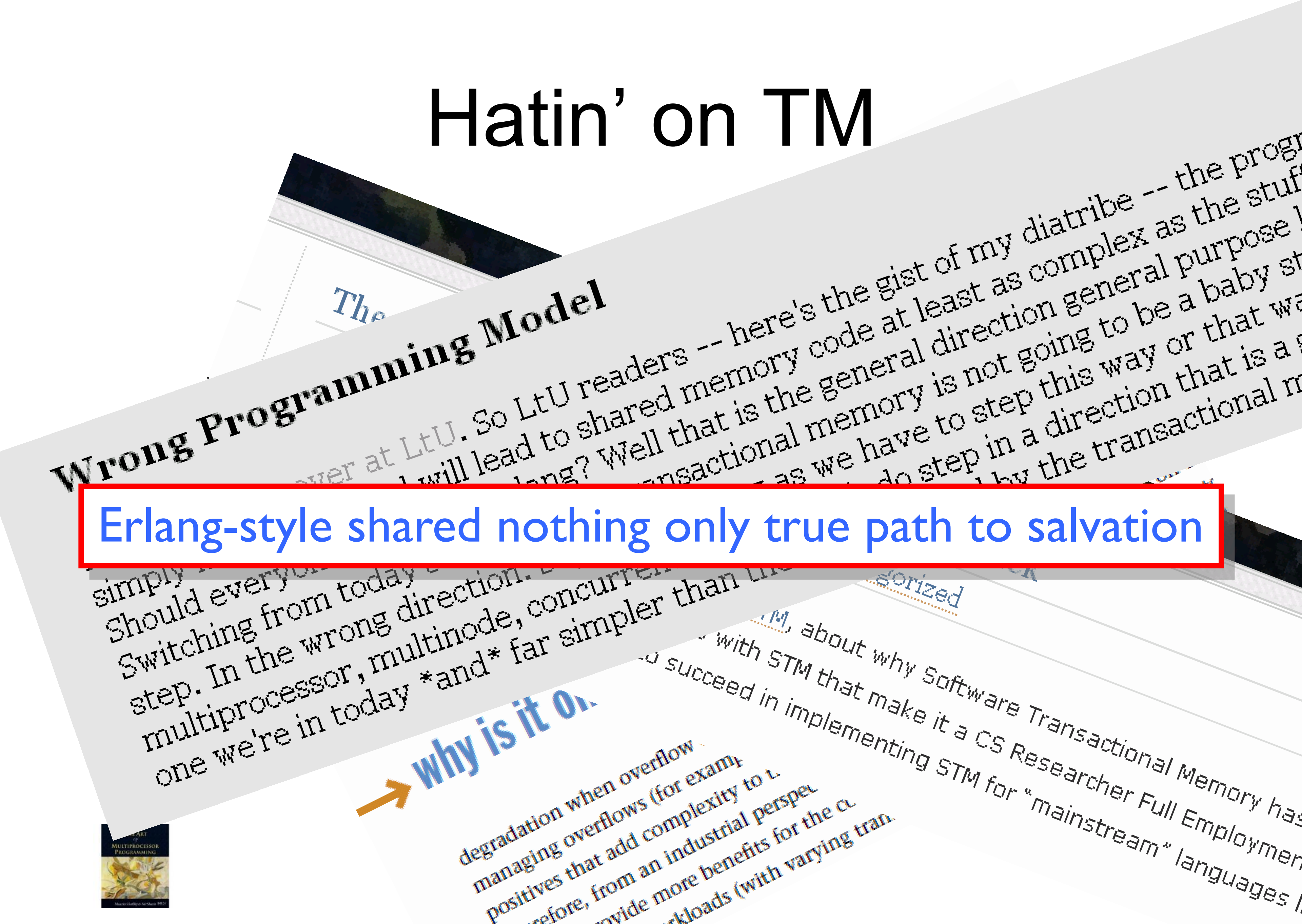
# Hatin' on TM




Requires radical change in programming style

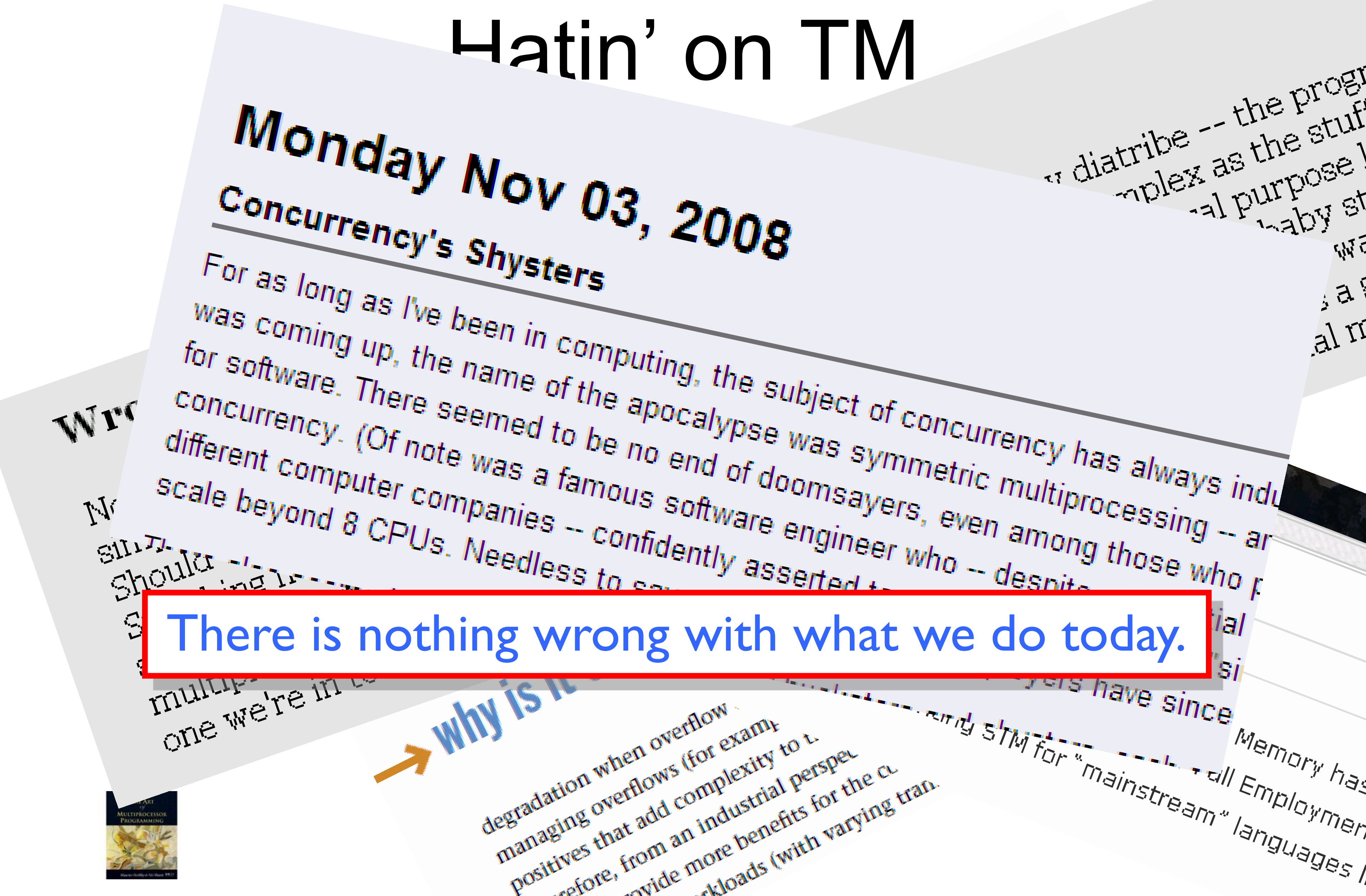


# Hatin' on TM



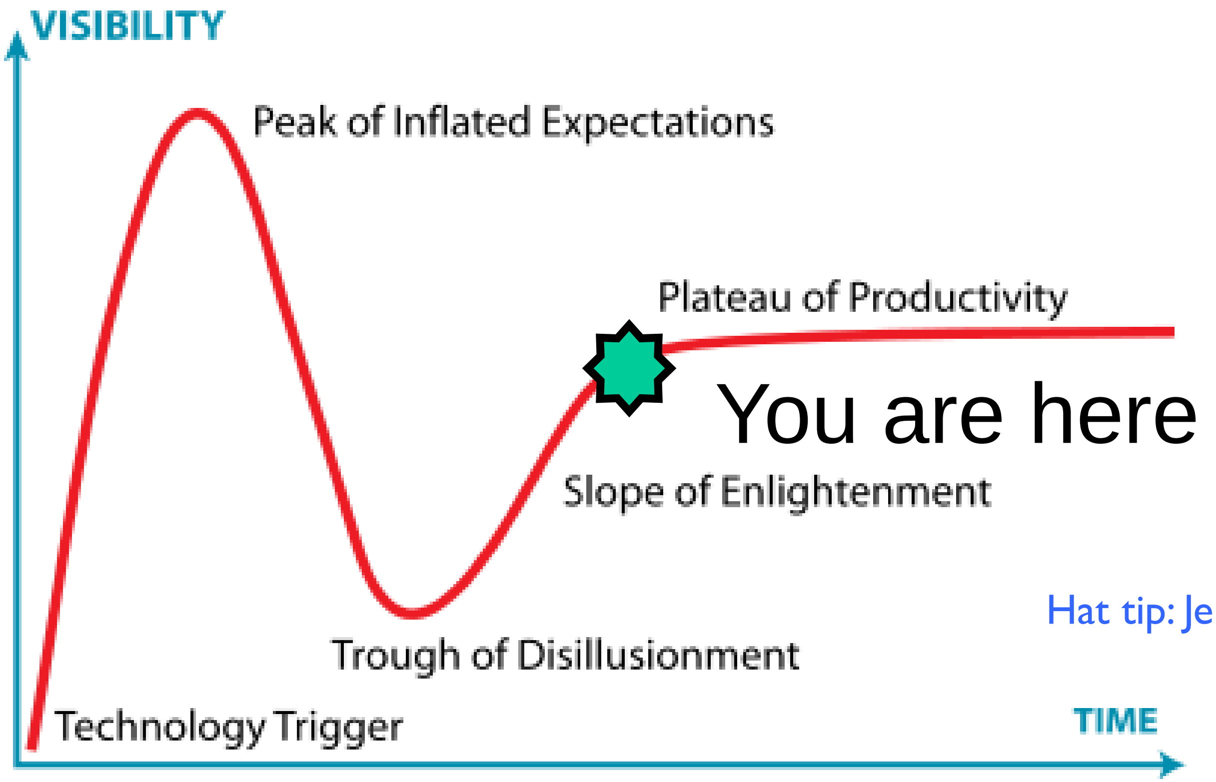

Erlang-style shared nothing only true path to salvation

# Hatin' on TM

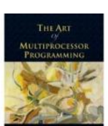



There is nothing wrong with what we do today.

# Gartner Hype Cycle




Hat tip: Jeremy Kemp



# Thanks ! תודה