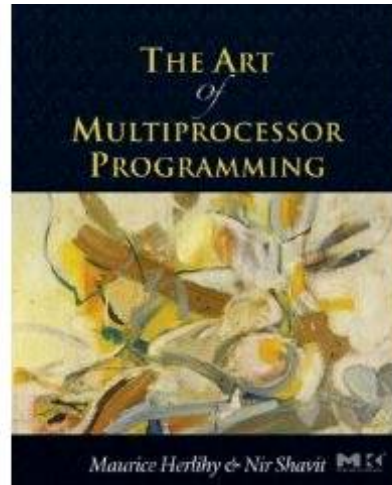
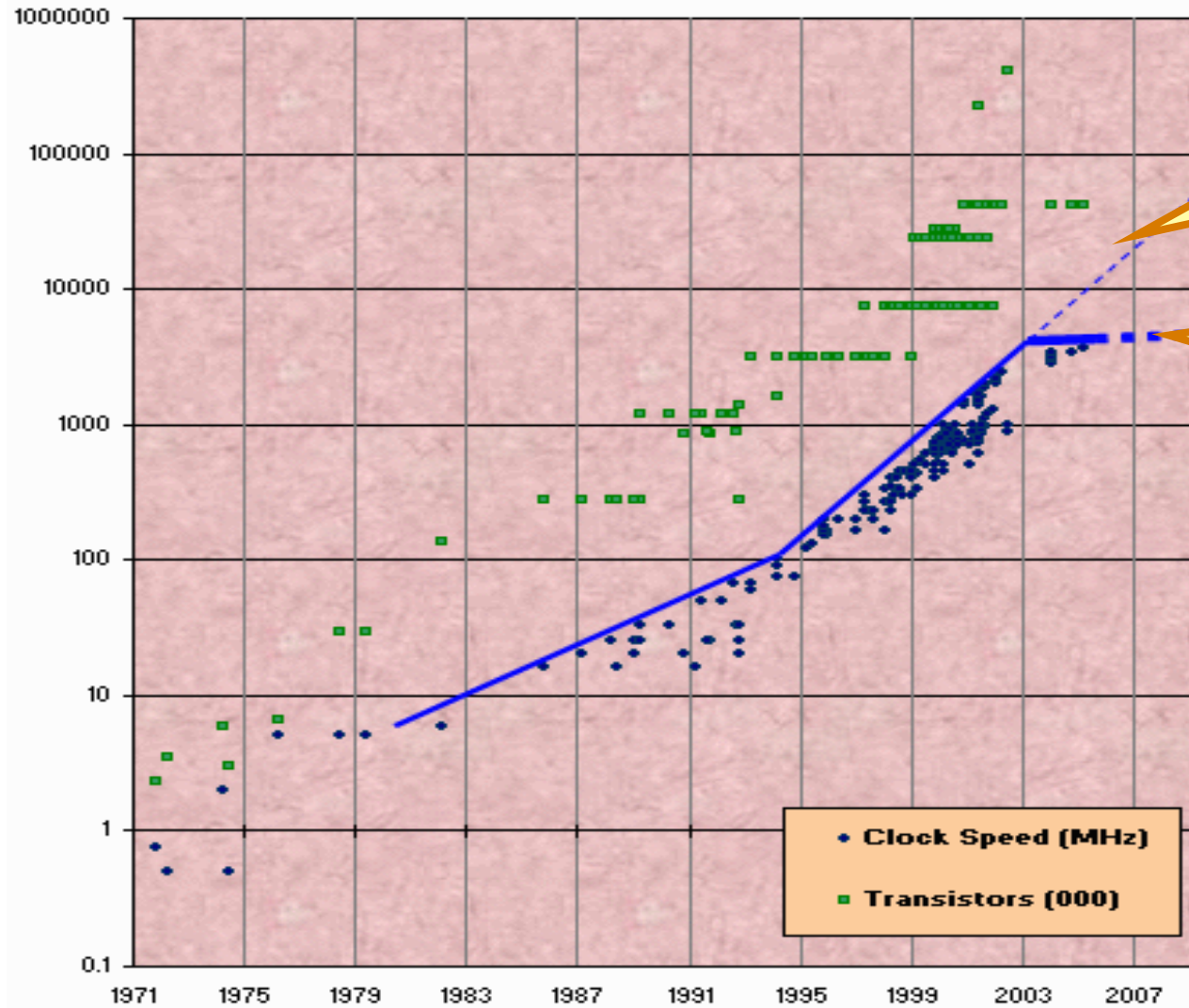


Transactional Memory



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

Moore's Law



Transistor
count still
rising

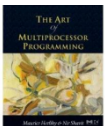
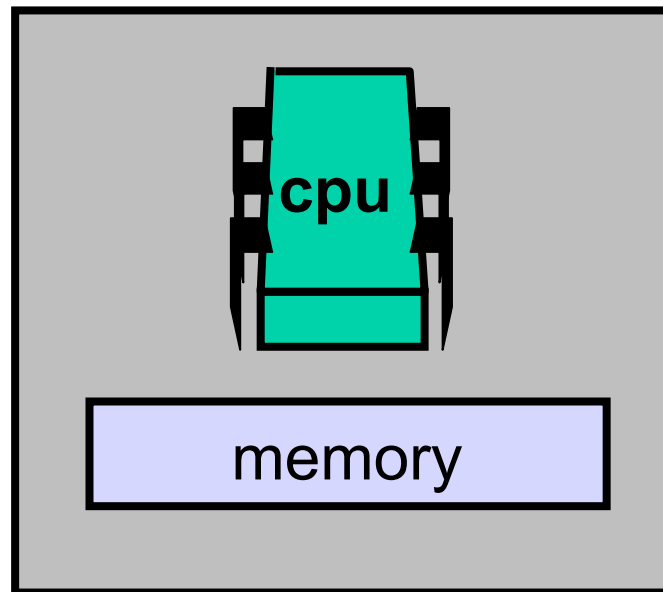
Clock speed
flattening
sharply



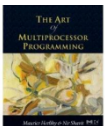
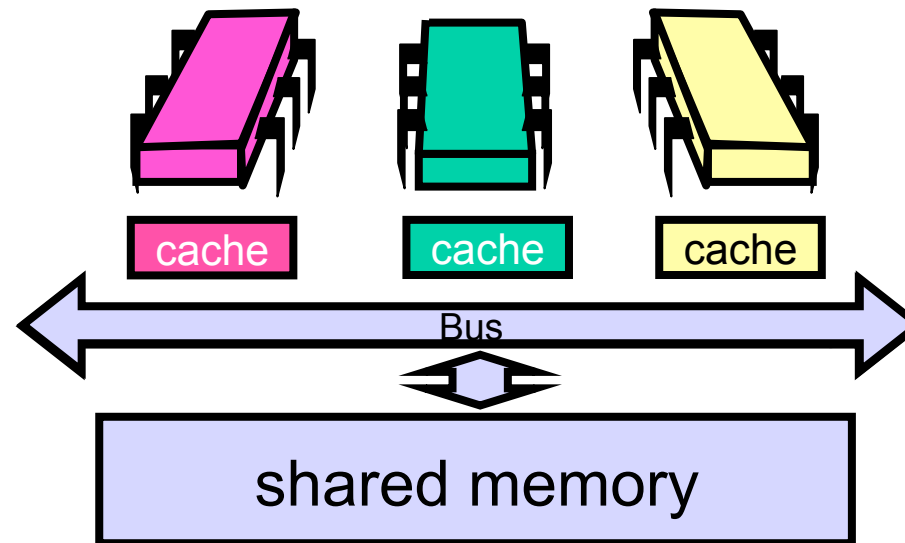
Moore's Law (in practice)



Nearly Extinct: the Uniprocessor

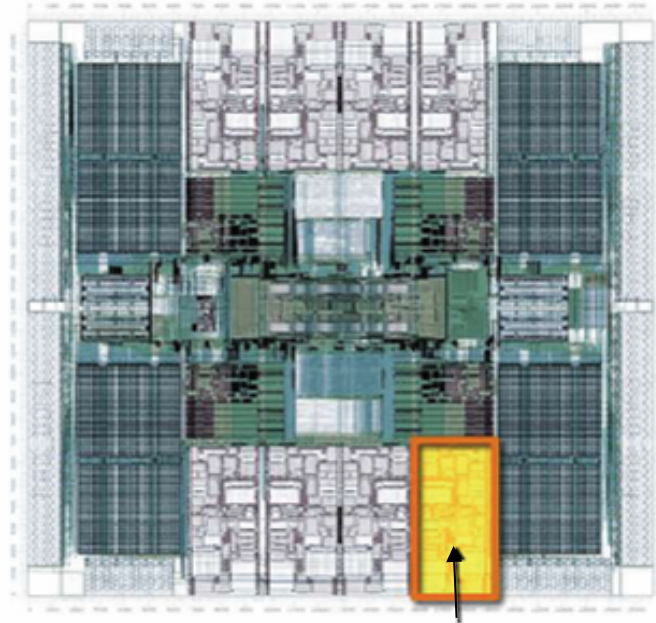


Endangered: The Shared Memory Multiprocessor (SMP)

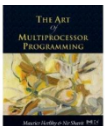


The New Boss: The Multicore Processor (CMP)

**All on the
same chip**

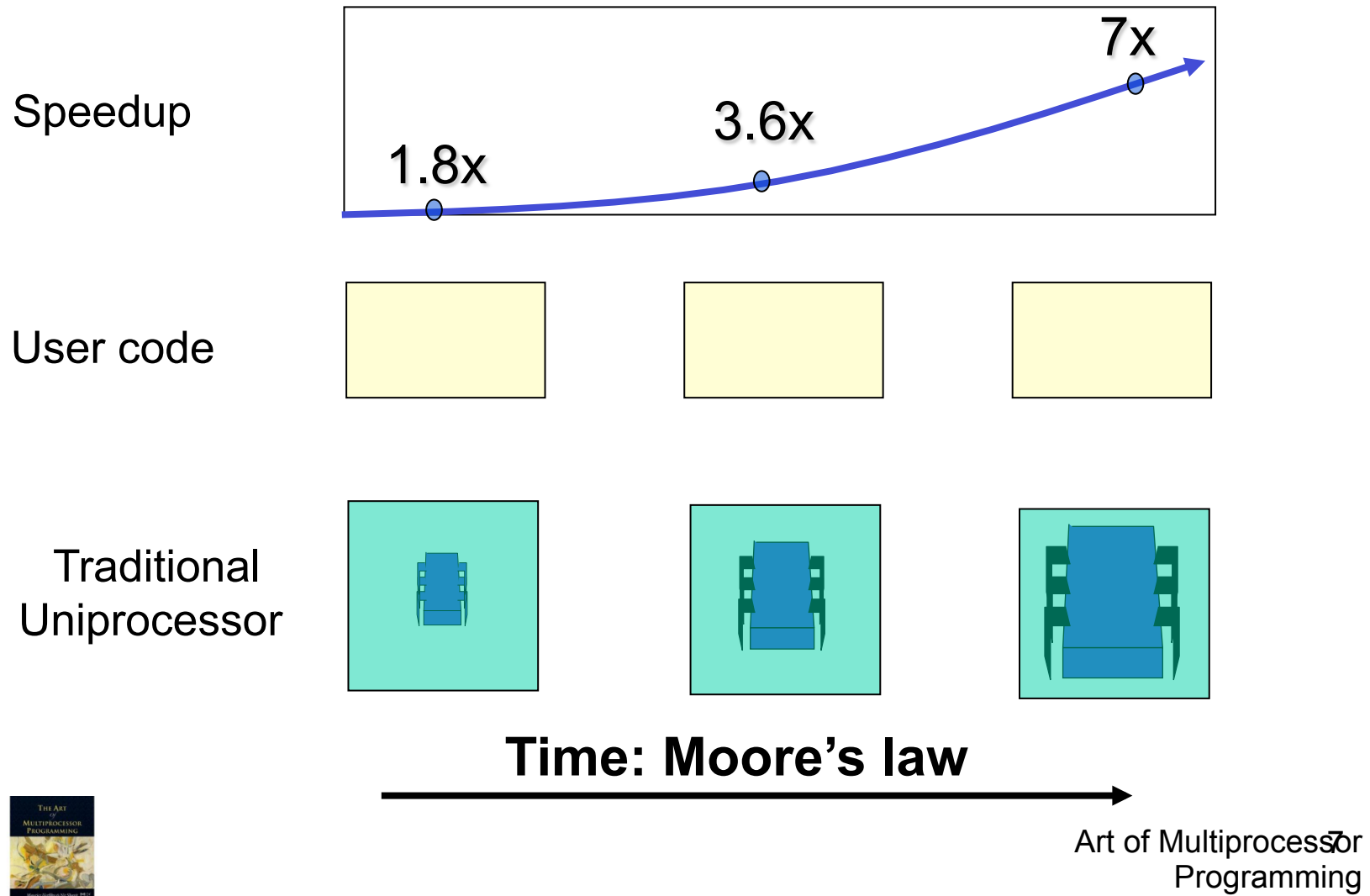


**Sun
T2000
Niagara**

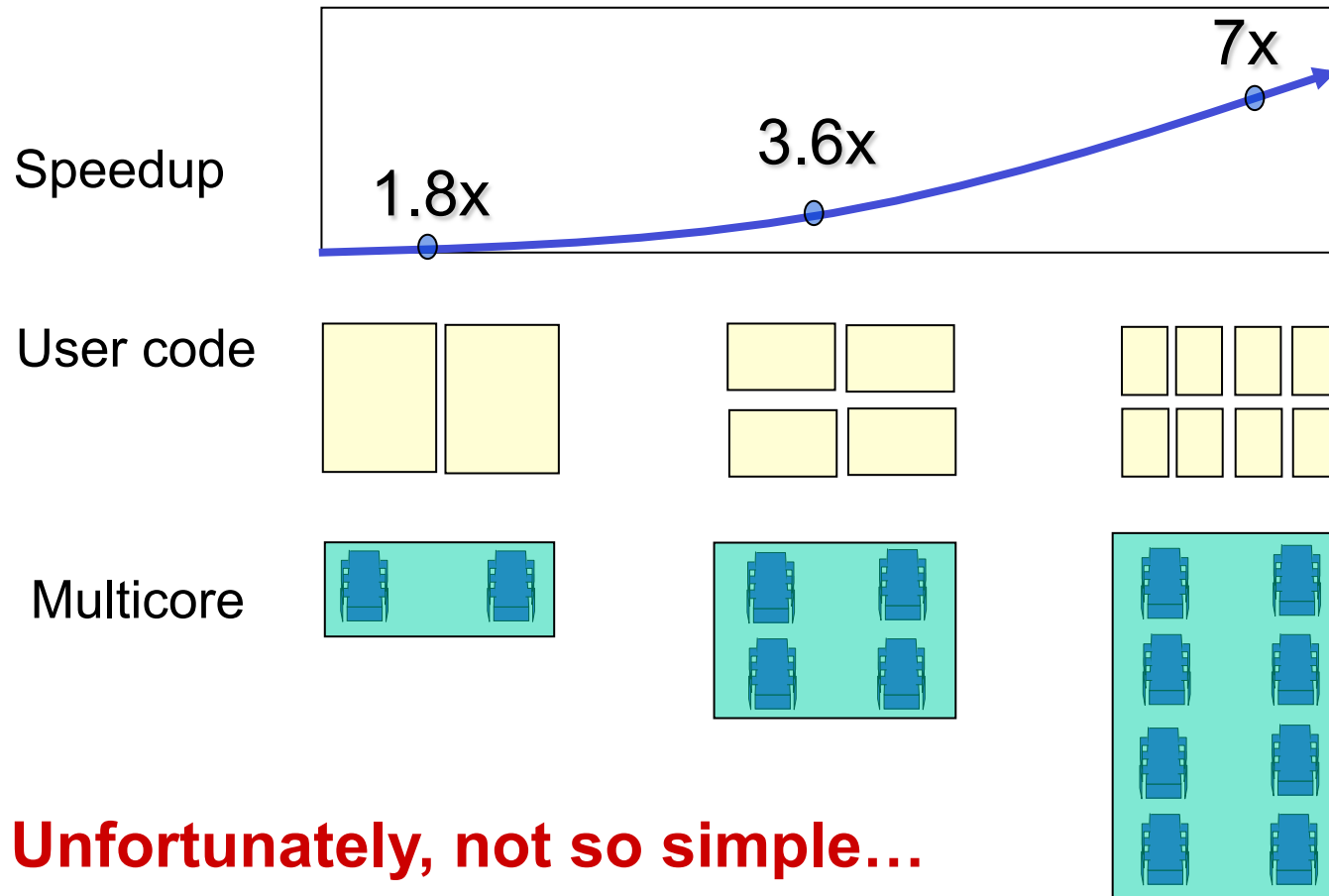


Art of Multiprocessor
Programming

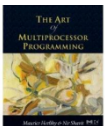
Traditional Scaling Process



Ideal Scaling Process

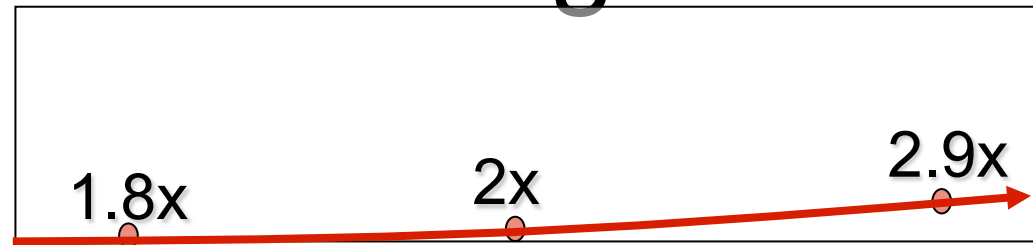


Unfortunately, not so simple...

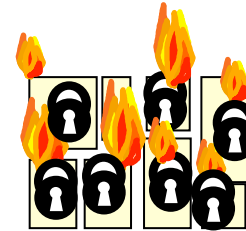
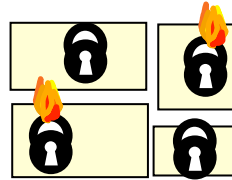
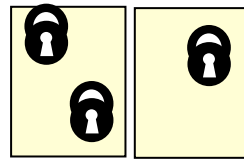


Actual Scaling Process

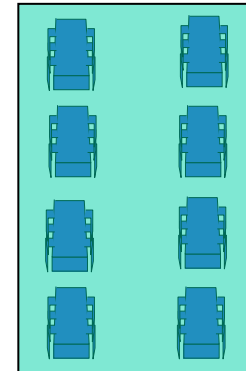
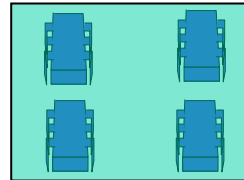
Speedup



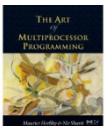
User code



Multicore



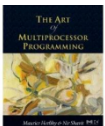
**Parallelization and Synchronization
require great care...**



Art of Multiprocessor
Programming

Amdahl's Law

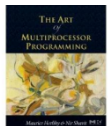
$$\text{Speedup} = \frac{\text{1-thread execution time}}{n\text{-thread execution time}}$$



Amdahl's Law

$$1 / (1 + p + p/n)$$

Speedup=

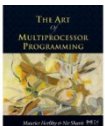
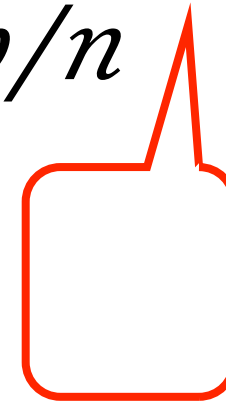


Amdahl's Law

Speedup=

$$1 / (1 + p + \frac{p}{n})$$

parallel fraction



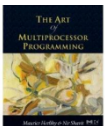
Amdahl's Law

Speedup=

$$1 / (1 + p + p/n)$$

**parallel
fraction**

**Number of
threads**



Amdahl's Law

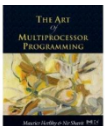
sequential
fraction

parallel
fraction

$$1 / (1 + p + p/n)$$

Speedup =

Number of
threads



Amdal's Law

Bad synchronization ruins everything



Example

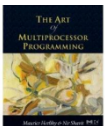
You buy a 10-core machine ...

Your application is:

60% concurrent

40% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine ...

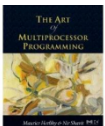
Your application is:

60% concurrent

40% sequential

$$1 / (1 - 0.6 - 0.6 / 10) = 2$$

How close to a 10-fold speedup?



Example

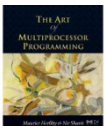
You buy a 10-core machine ...

Your application is:

80% concurrent

20% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine ...

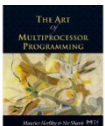
Your application is:

80% concurrent

20% sequential

$$1 / (1 - 0.8 - 0.8/10) = 3$$

How close to a 10-fold speedup?



Example

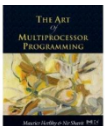
You buy a 10-core machine ...

Your application is:

90% concurrent

10% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine ...

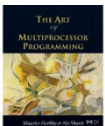
Your application is:

80% concurrent

20% sequential

$$1 / (1 - 0.9 - 0.9/10) = 5$$

How close to a 10-fold speedup?



Example

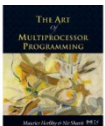
You buy a 10-core machine ...

Your application is:

99% concurrent

01% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine ...

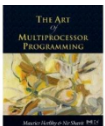
Your application is:

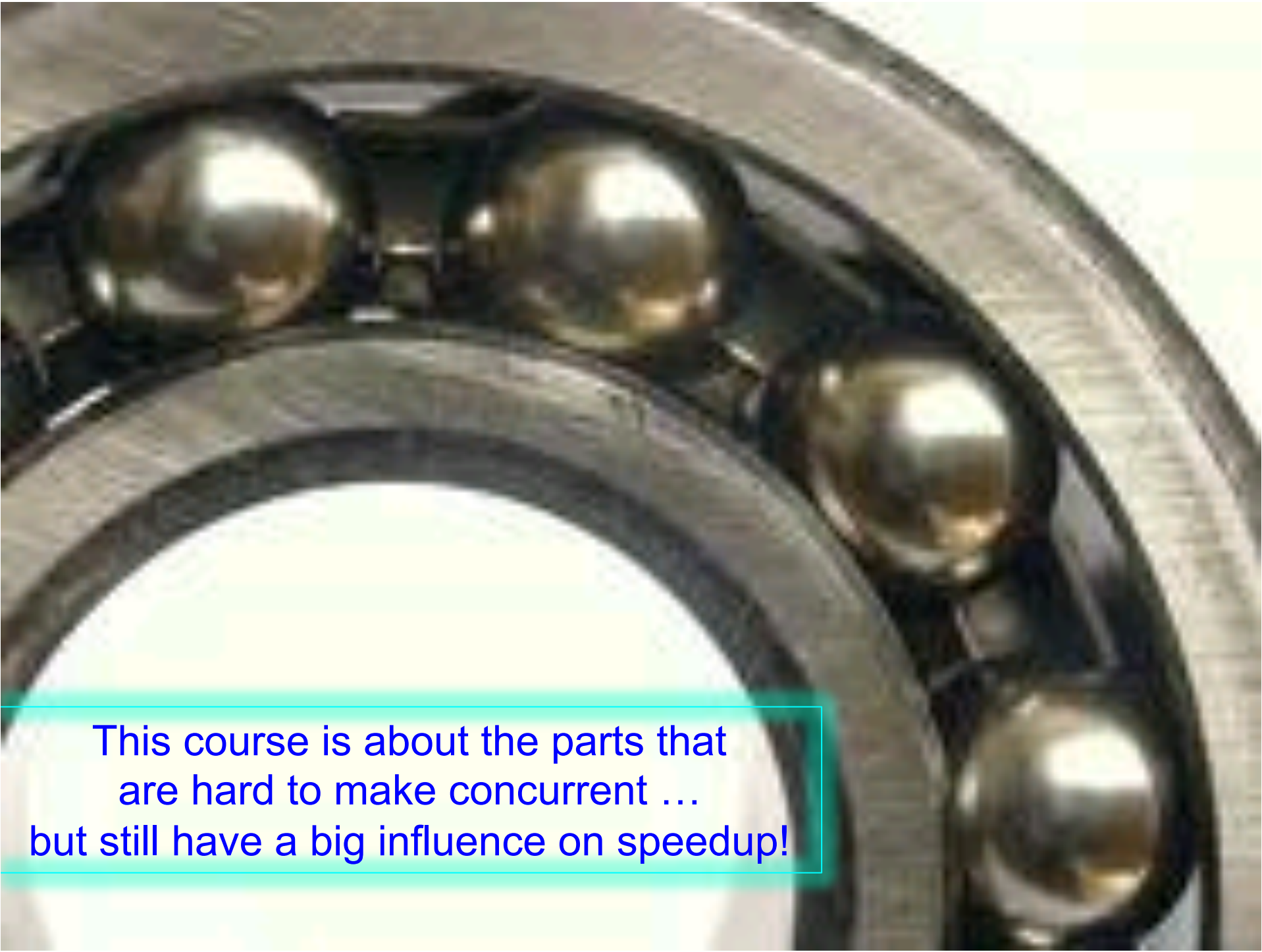
80% concurrent

20% sequential

$$1 / (1 - 0.99 - 0.99 / 10) =$$

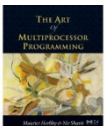
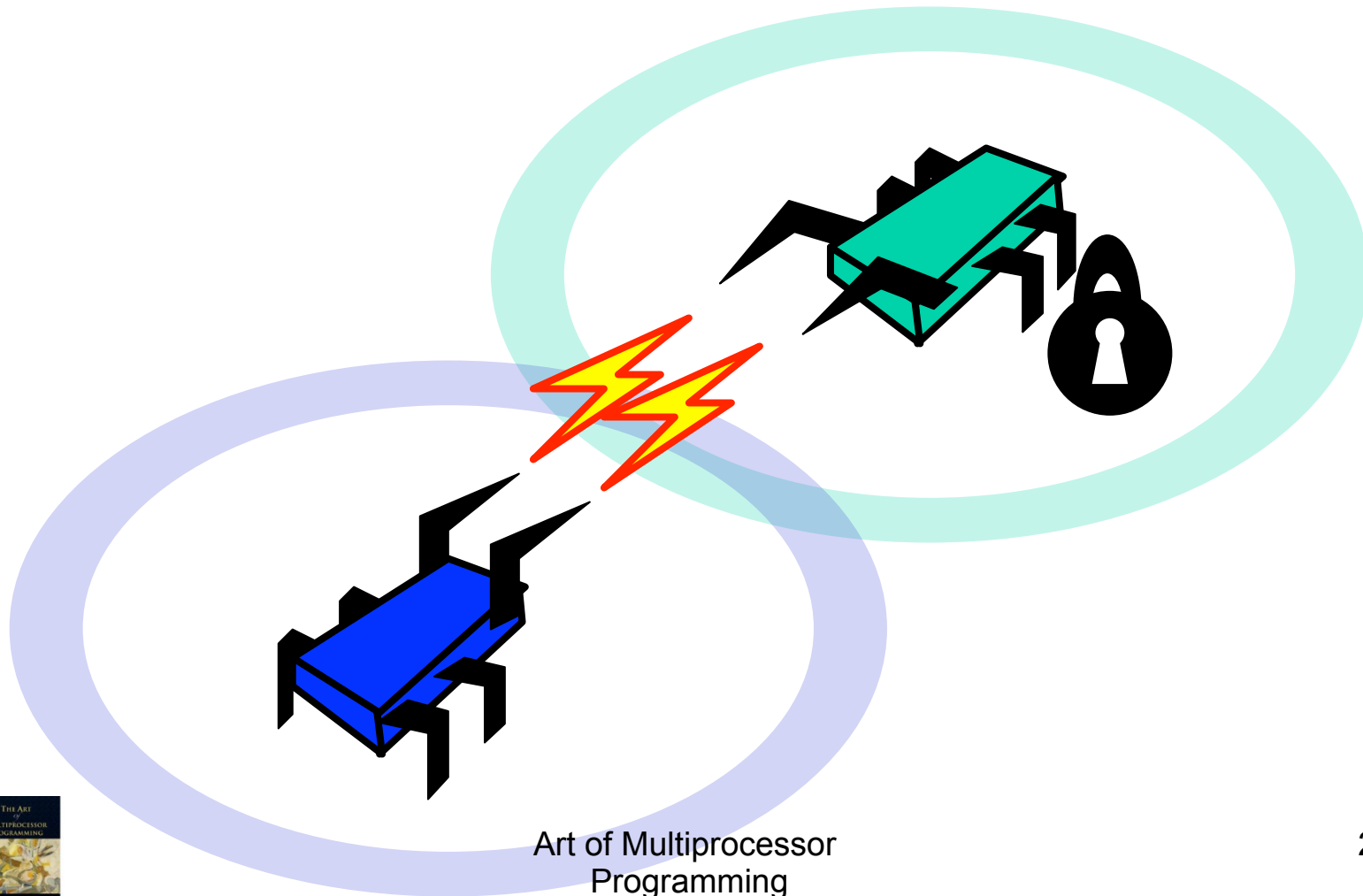
How close to a 10-fold speedup?



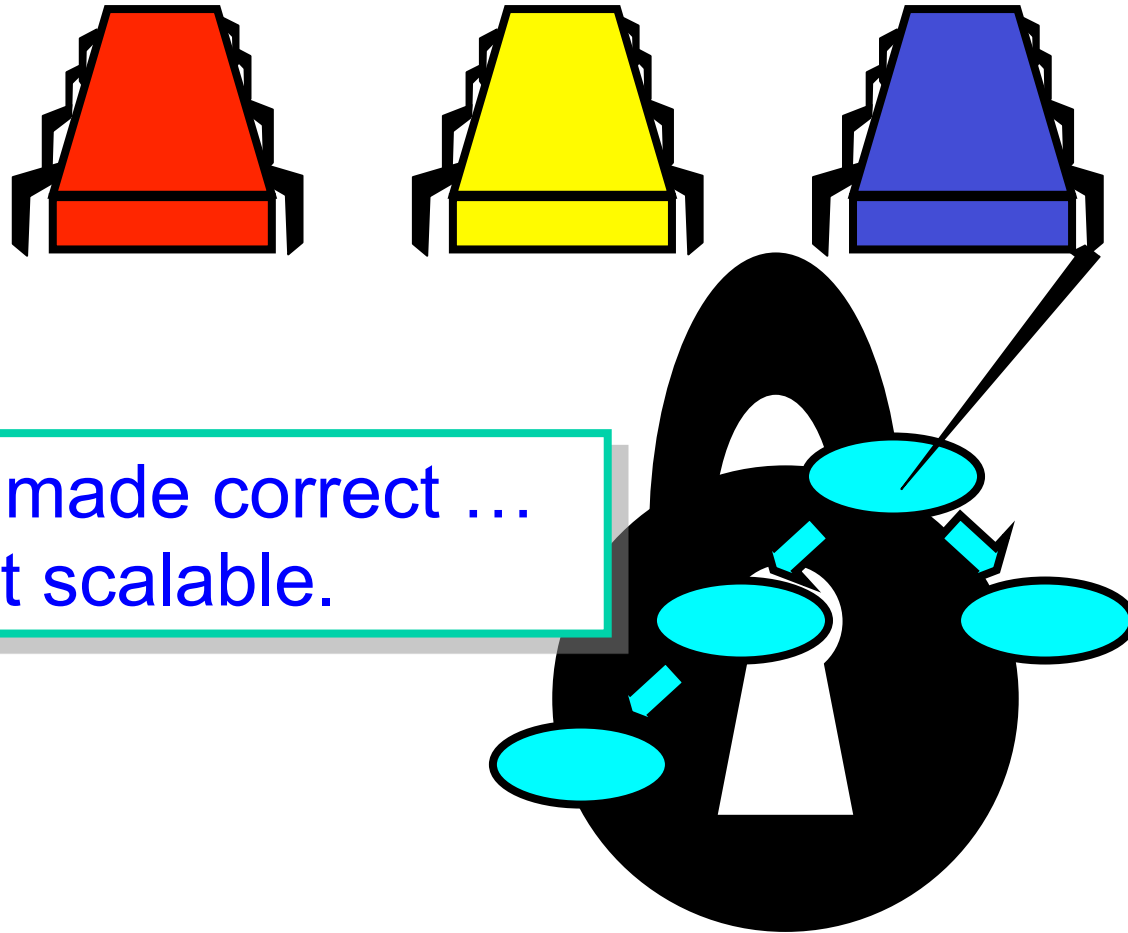
A close-up photograph of a ball bearing, showing several polished steel balls held in place by a dark metal cage. The balls are arranged in a circular pattern, and the image is slightly out of focus, emphasizing the texture and shape of the components.

This course is about the parts that
are hard to make concurrent ...
but still have a big influence on speedup!

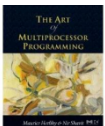
Locking



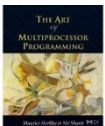
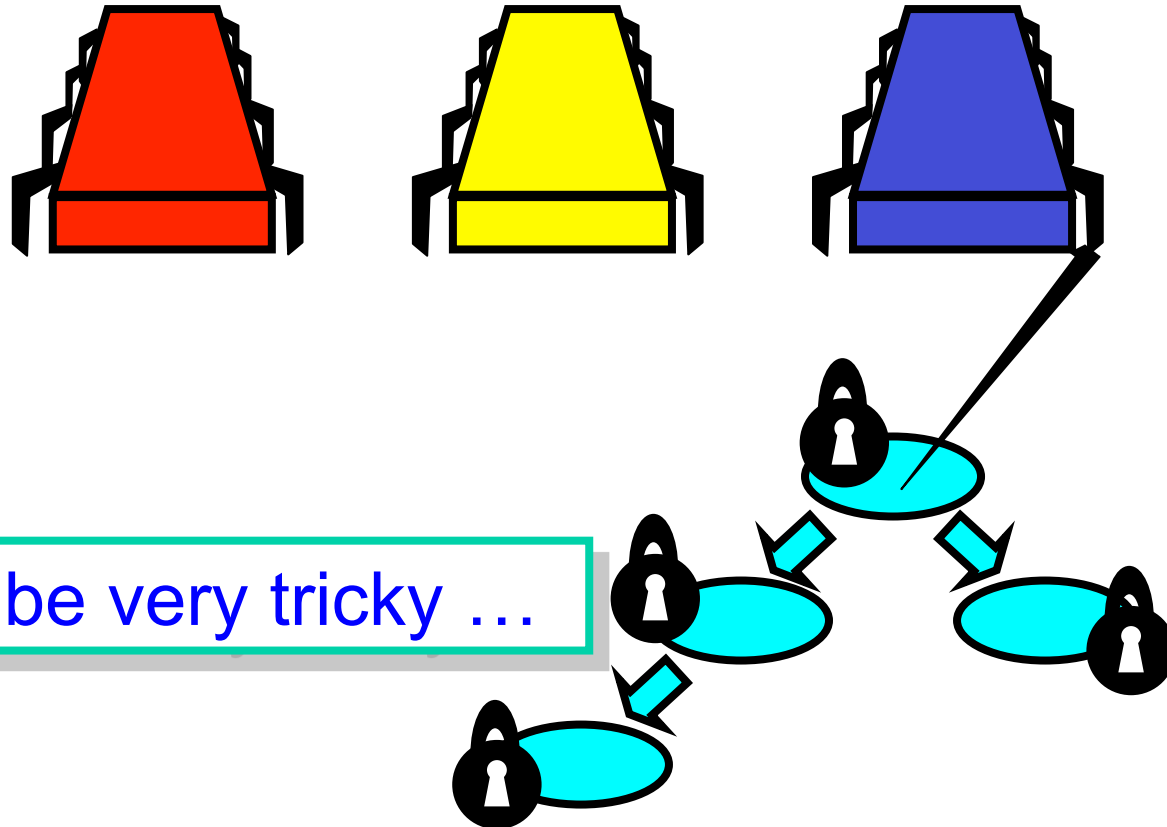
Coarse-Grained Locking



Easily made correct ...
But not scalable.

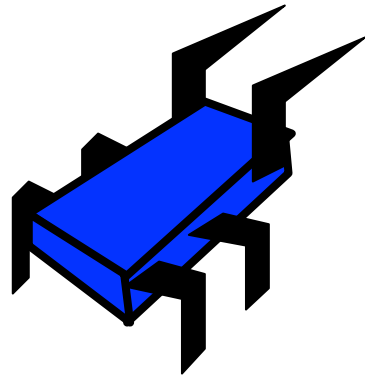
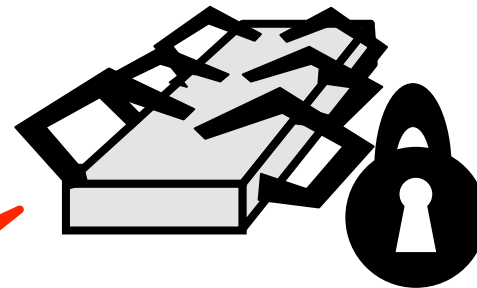


Fine-Grained Locking

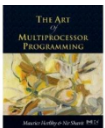


Locks are not Robust

If a thread holding
a lock is delayed ...



No one else can make
progress



Locking Relies on Conventions

Relation between ...

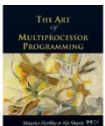
Lock data and object data

Exists only in programmer's

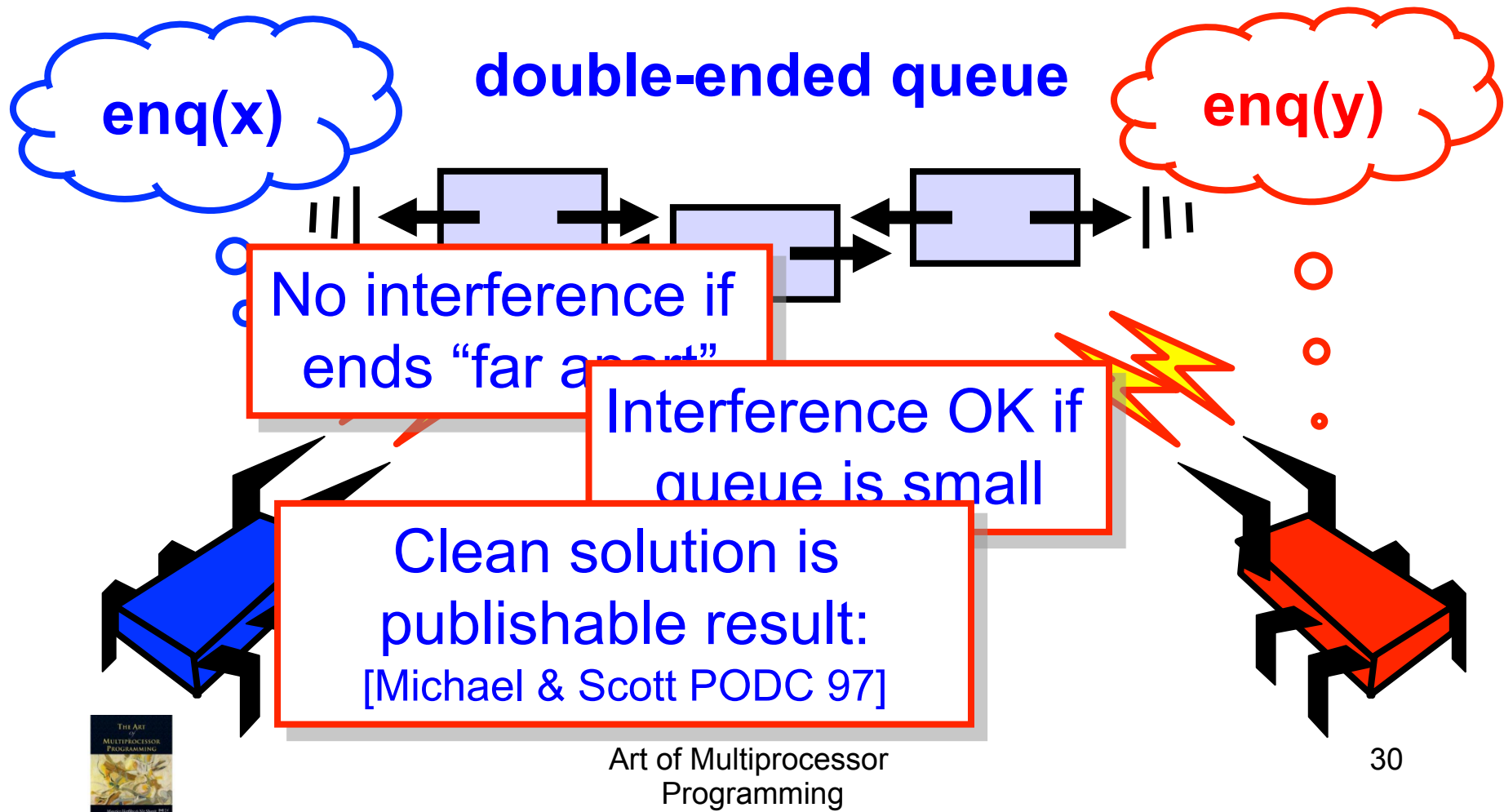
Actual comment
from Linux Kernel

(hat tip: Bradley Kuszmaul)

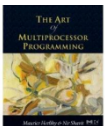
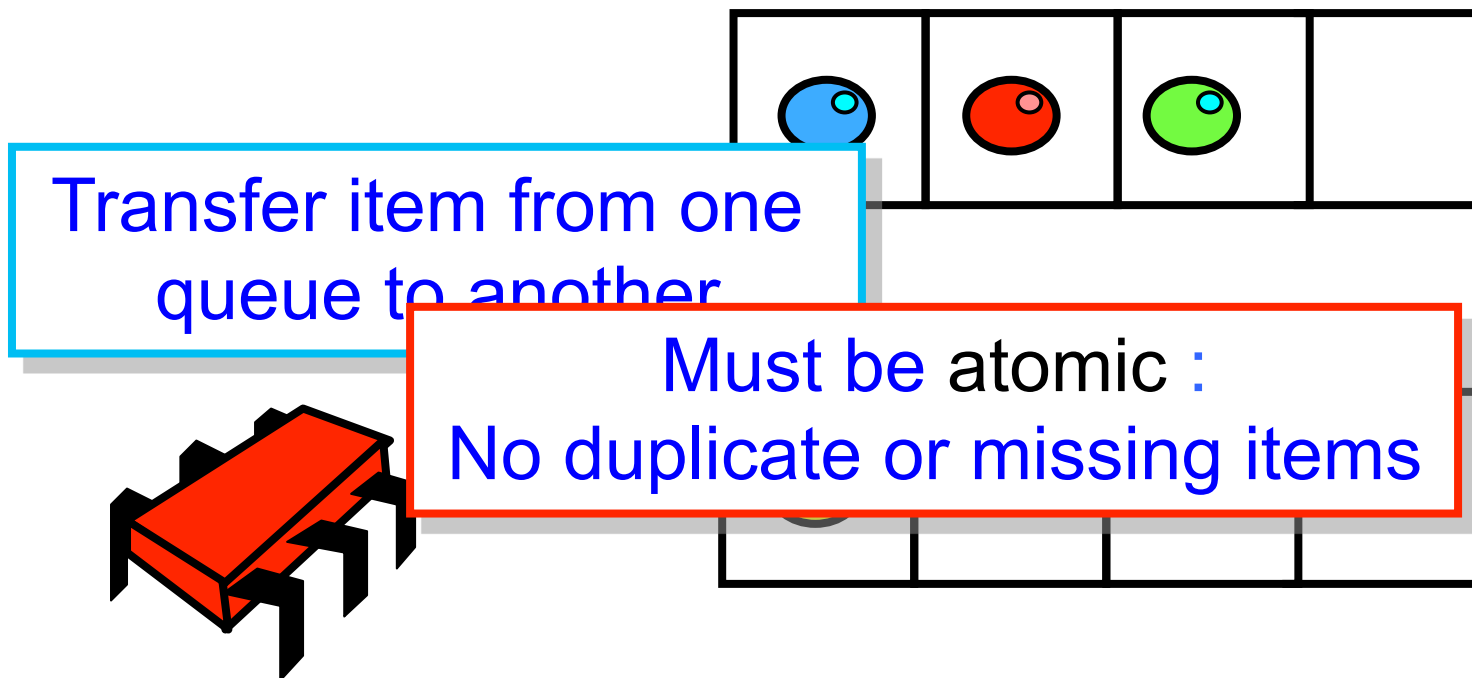
```
/*  
 * When a locked buffer is visible to the I/O layer  
 * BH_Launder is set. This means before unlocking  
 * we must clear BH_Launder,mb() on alpha and then  
 * clear BH_Lock, so no reader can see BH_Launder set  
 * on an unlocked buffer and then risk to deadlock.  
 */
```



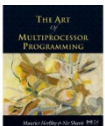
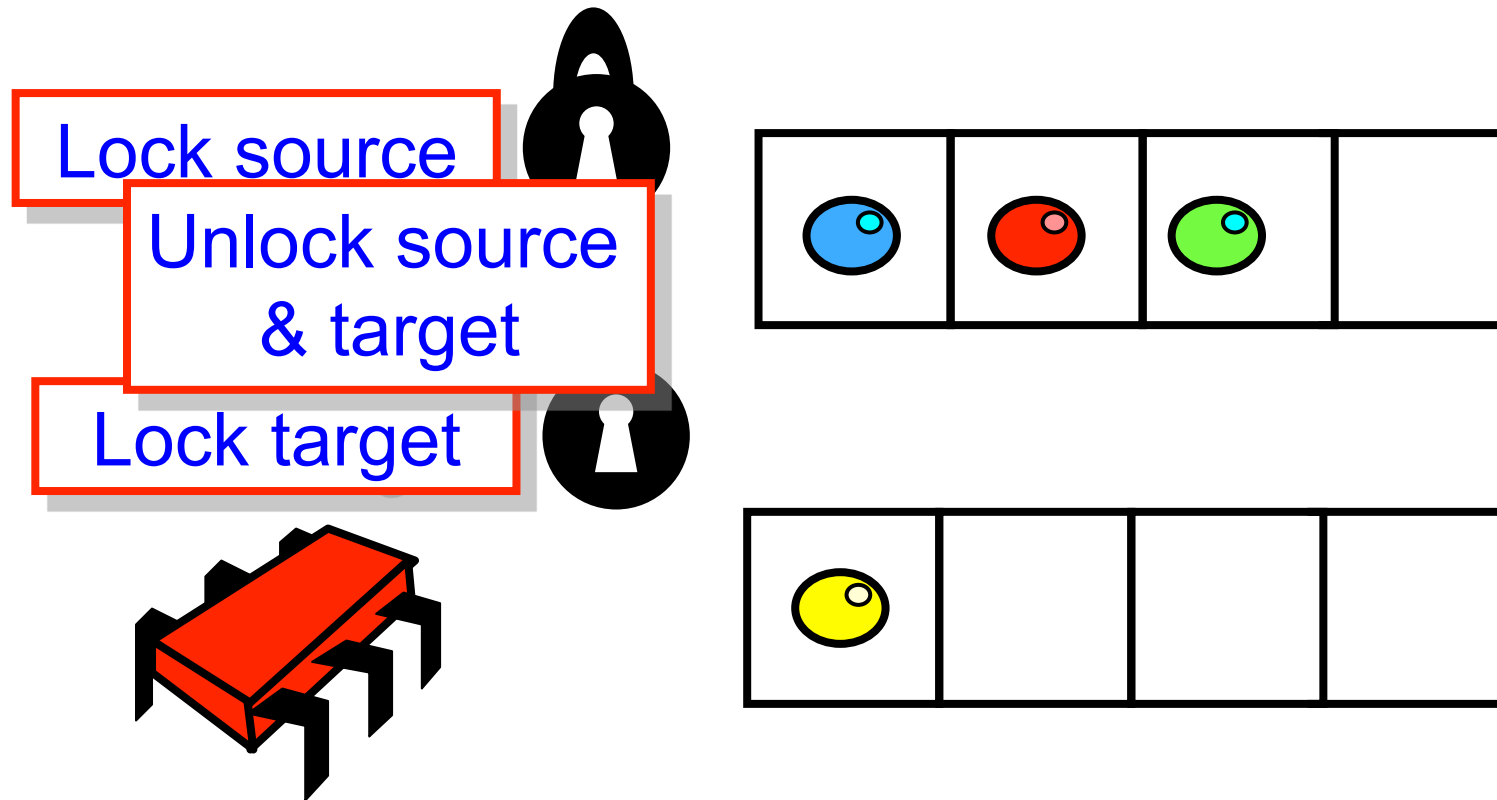
Simple Problems are hard



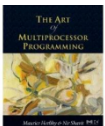
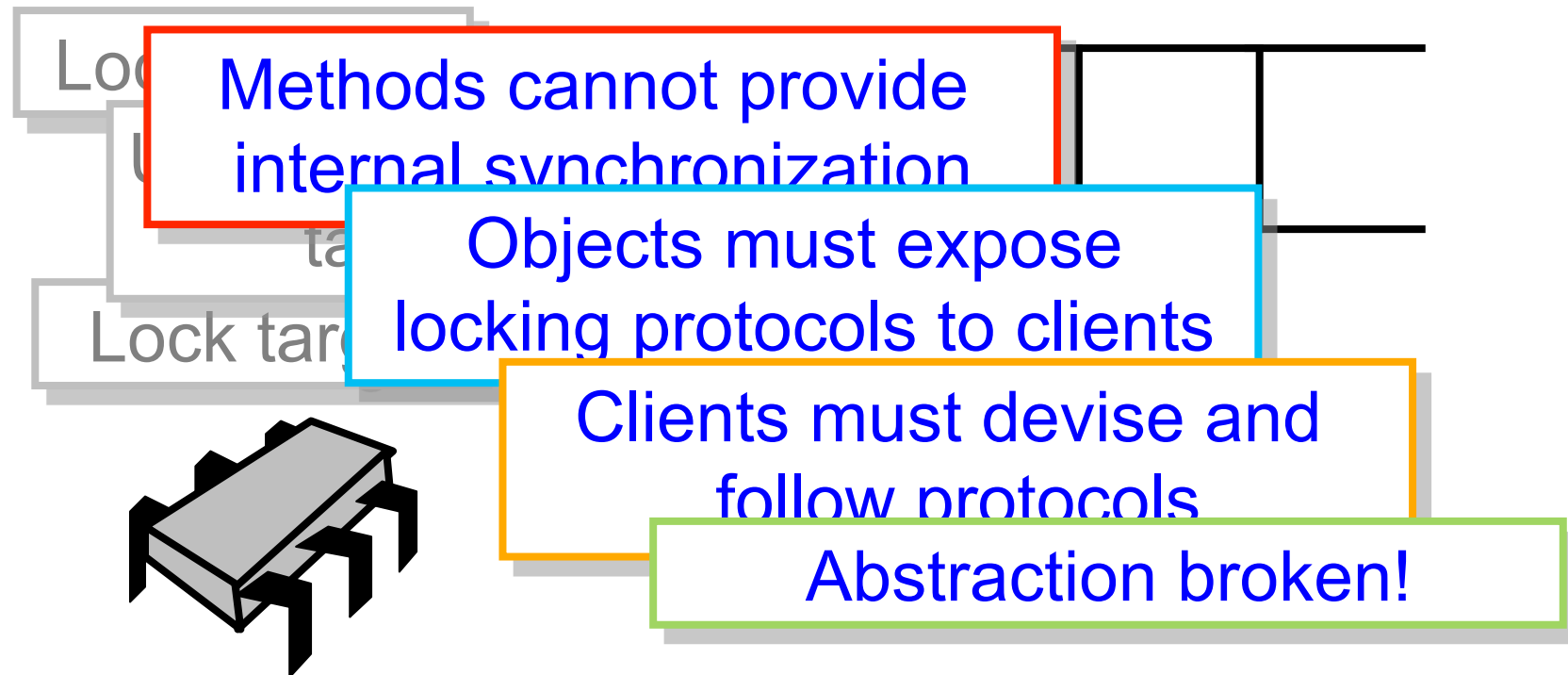
Locks Not Composable



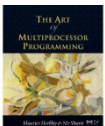
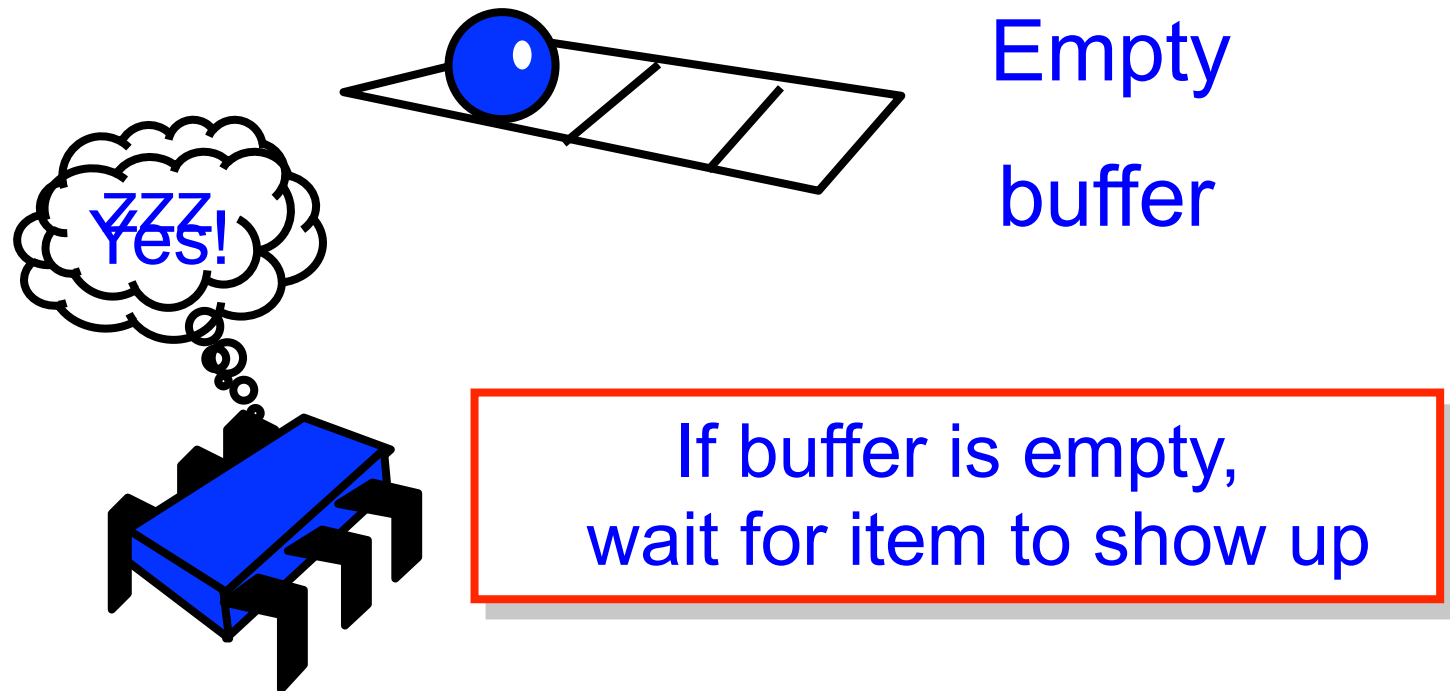
Locks Not Composable



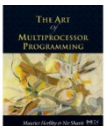
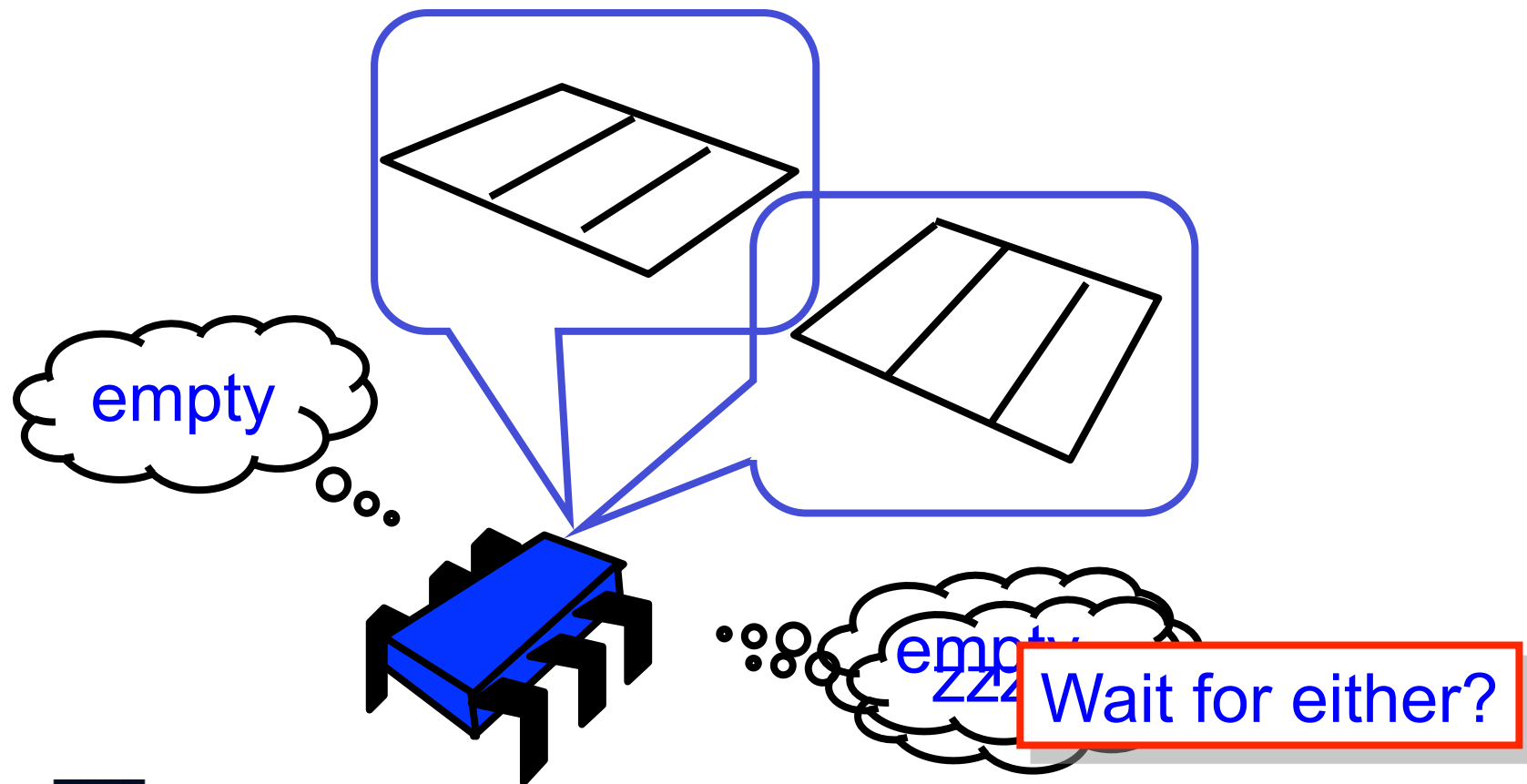
Locks Not Composable



Monitor Wait and Signal

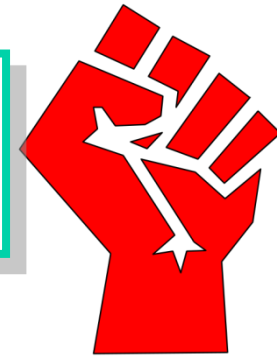


Wait and Signal do not Compose



The Transactional Manifesto

Much modern programming practice
inadequate for multicore world

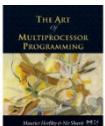


Agenda

Replace **locking** with a **transactional API**

Design **languages** and **libraries**

Implement **efficient run-times**



Road Map

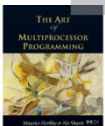
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

Research Questions



Road Map

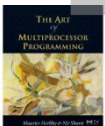
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

Research Questions



Transactions

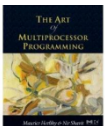
Block of code

Atomic: appears to happen
instantaneously

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

Commit: takes effect
(atomically)

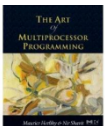
Abort: has no effect
(typically restarted)



Atomic Blocks

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

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

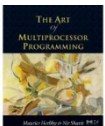


Atomic Blocks

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

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

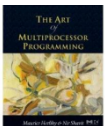
No data race



A Double-Ended Queue

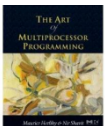
```
public void LeftEnq(item x) {  
    Qnode q = new Qnode(x);  
    q.left = left;  
    left.right = q;  
    left = q;  
}
```

Write sequential Code



A Double-Ended Queue

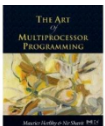
```
public void LeftEnq(item x)
atomic {
    Qnode q = new Qnode(x);
    q.left = left;
    left.right = q;
    left = q;
}
```



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

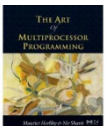
Not always this simple!

Conditional waits?

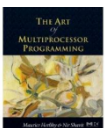
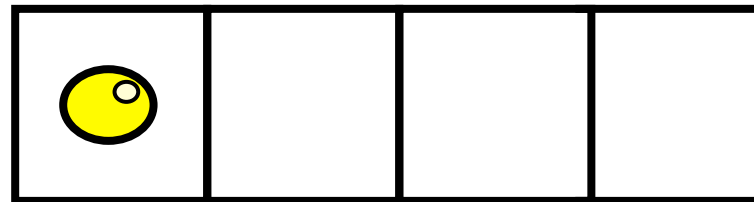
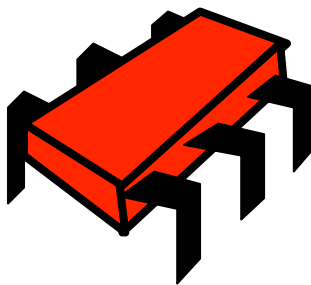
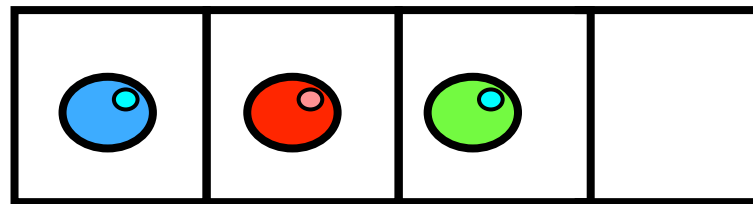
False conflicts?

Resource limits?

Better problems to have ...



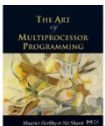
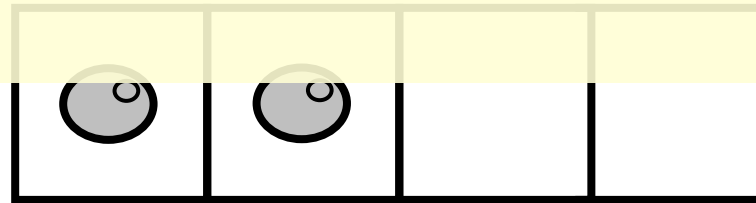
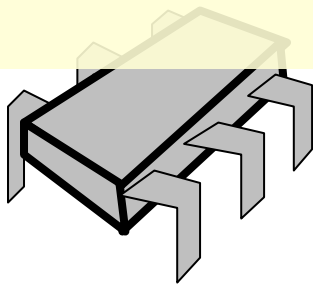
Composition?



Composition?

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

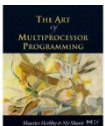
Trivial or what?



Conditional Waiting

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

**Roll back transaction
and restart when
something changes**



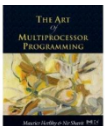
Composable Conditional Waiting

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

Run 1st method. If it retries

Run 2nd method. If it retries ...

Entire statement retries



Road Map

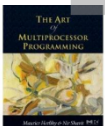
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

Research Questions

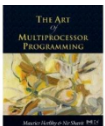
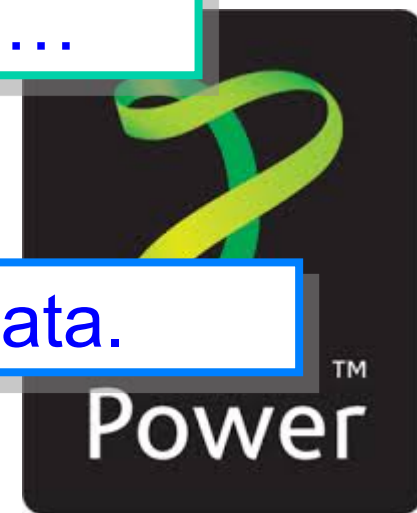
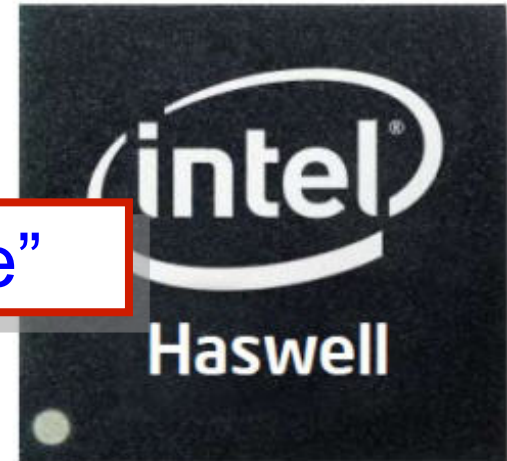


Hardware Transactional Memory

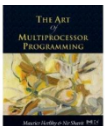
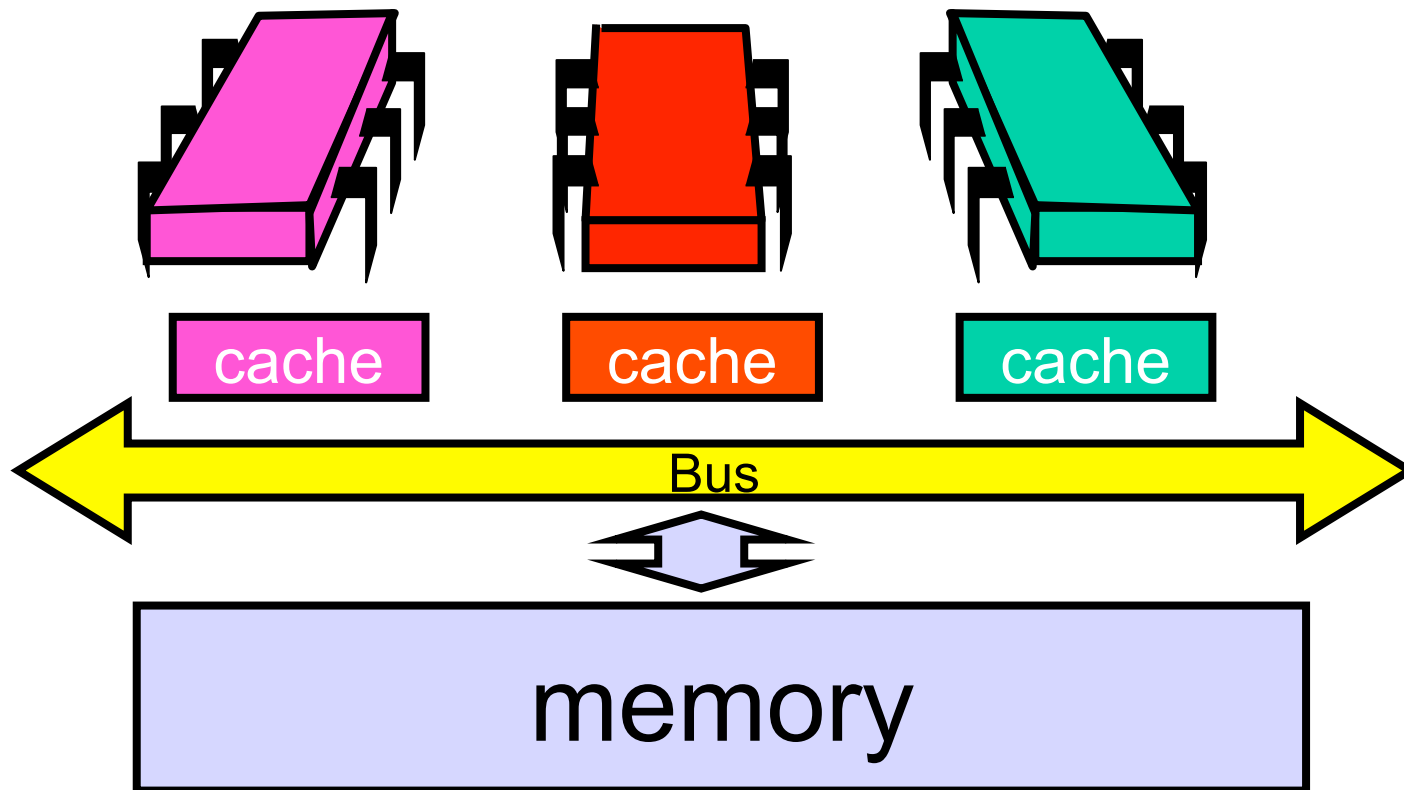
Exploit standard “cache coherence”

Detect synchronization conflicts ...

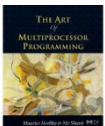
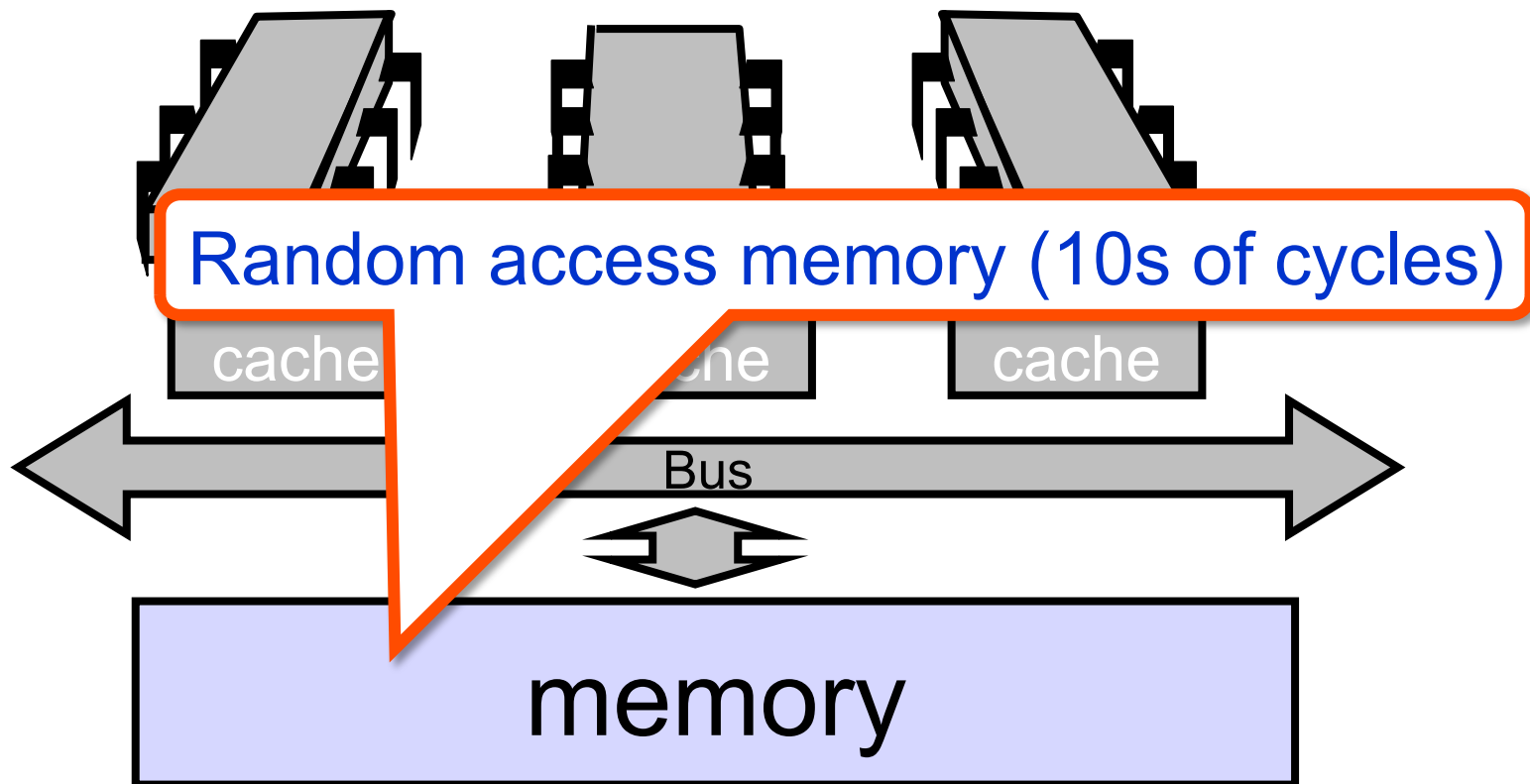
Invalidate cached copies of data.



Standard Cache Coherence



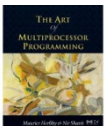
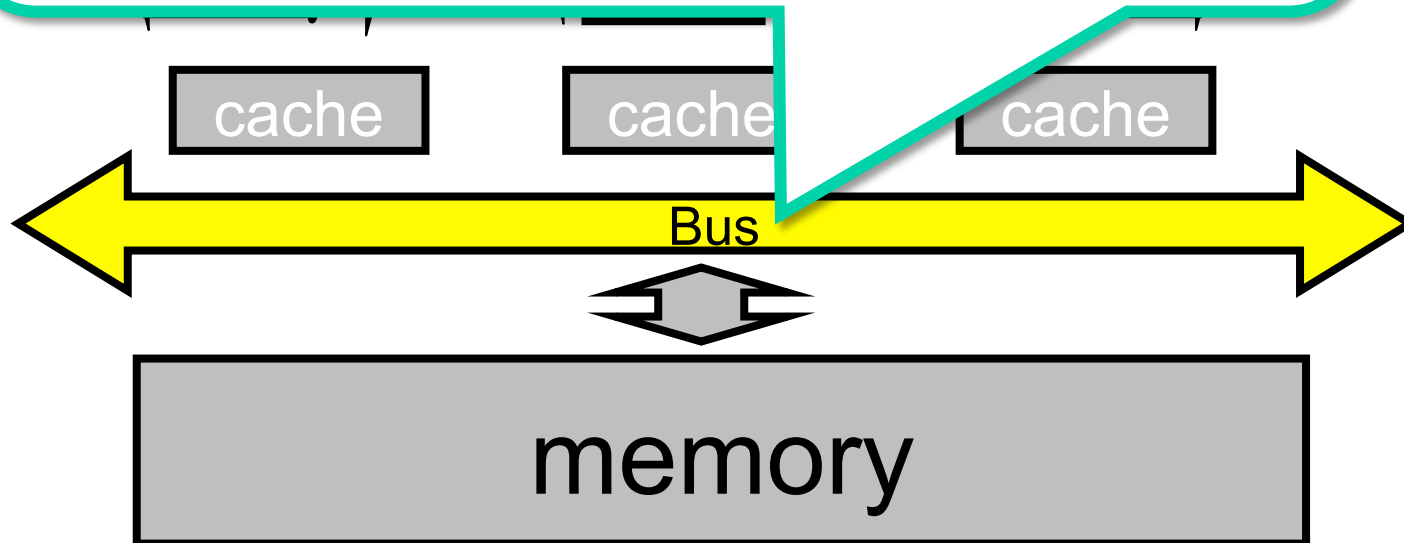
Standard Cache Coherence



Standard Cache Coherence

Shared Bus

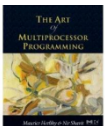
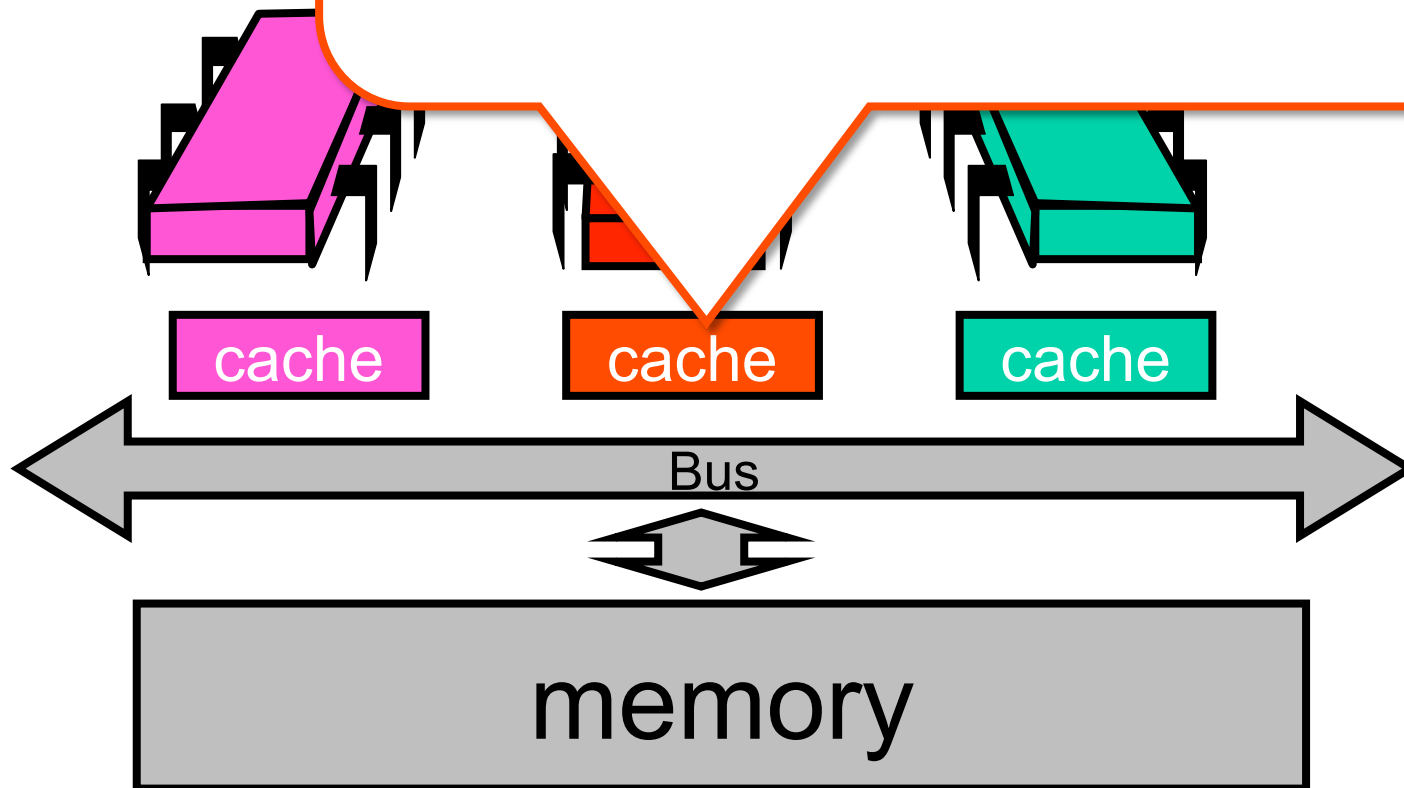
- Broadcast medium
- One broadcaster at a time
- Processors and memory all “snoop”



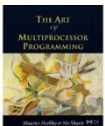
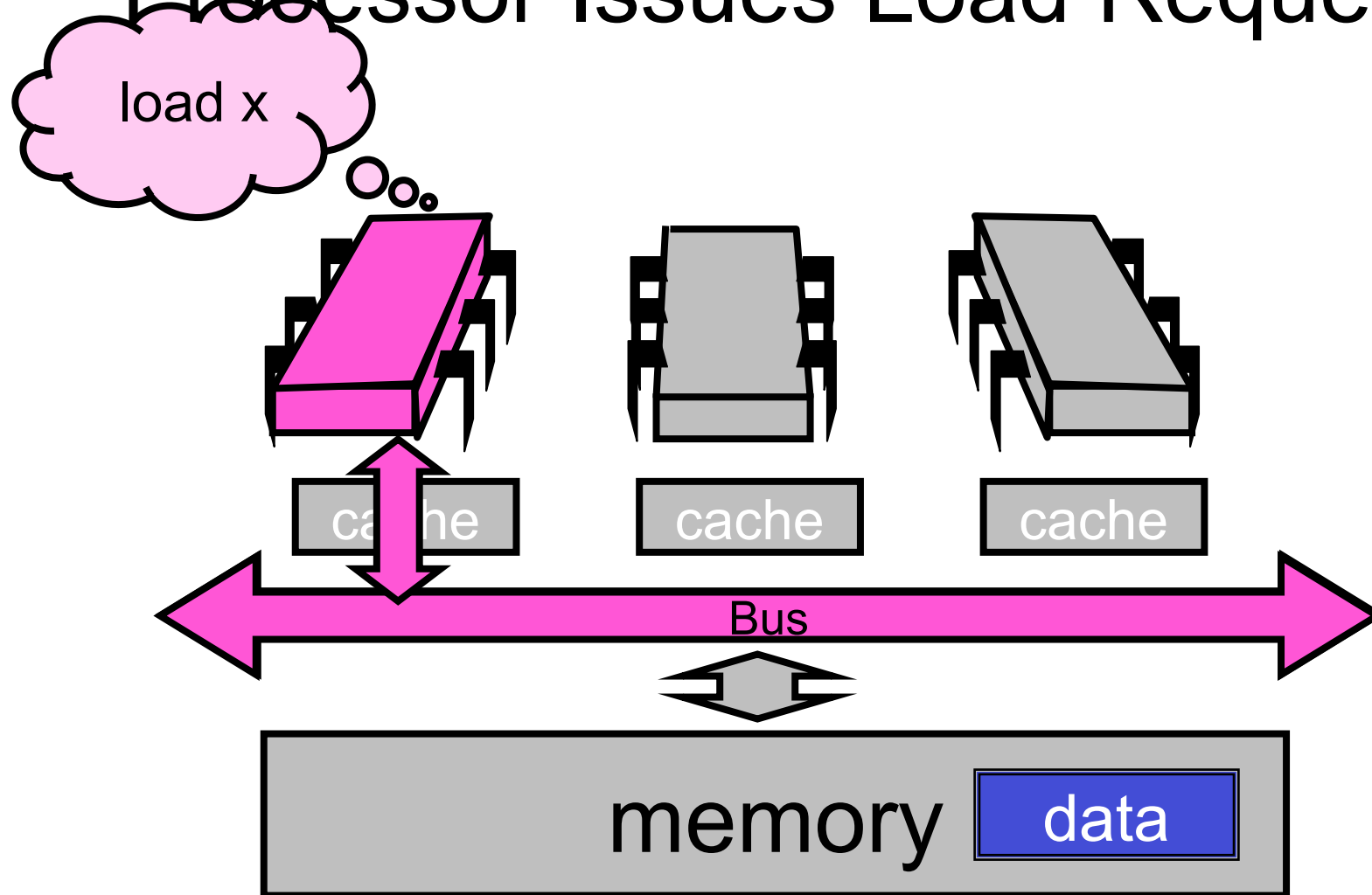
Stand

Per-Processor Caches

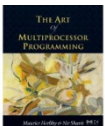
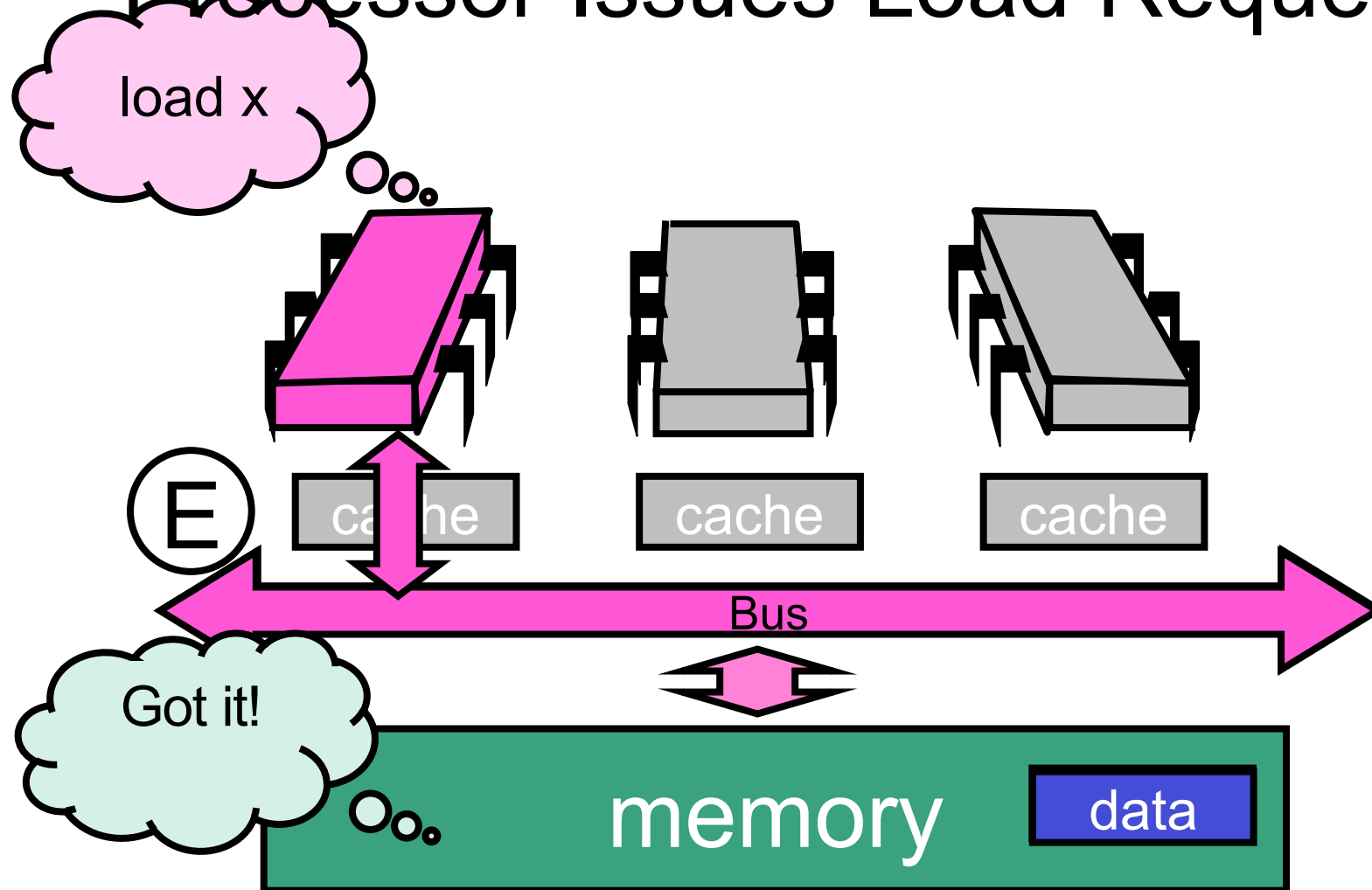
- Small
- Fast: 1 or 2 cycles
- Address & state information



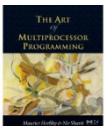
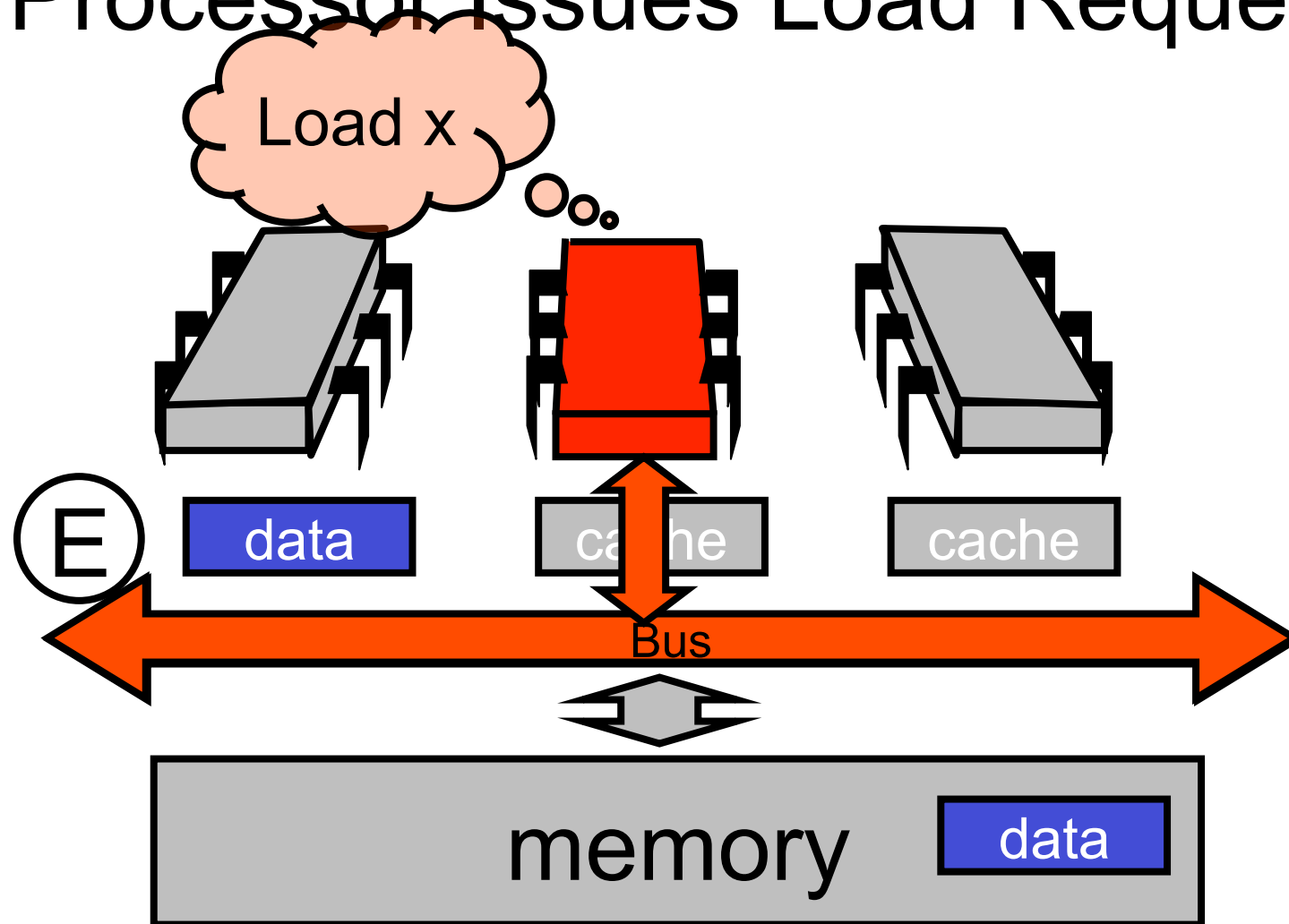
Processor Issues Load Request



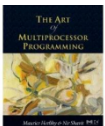
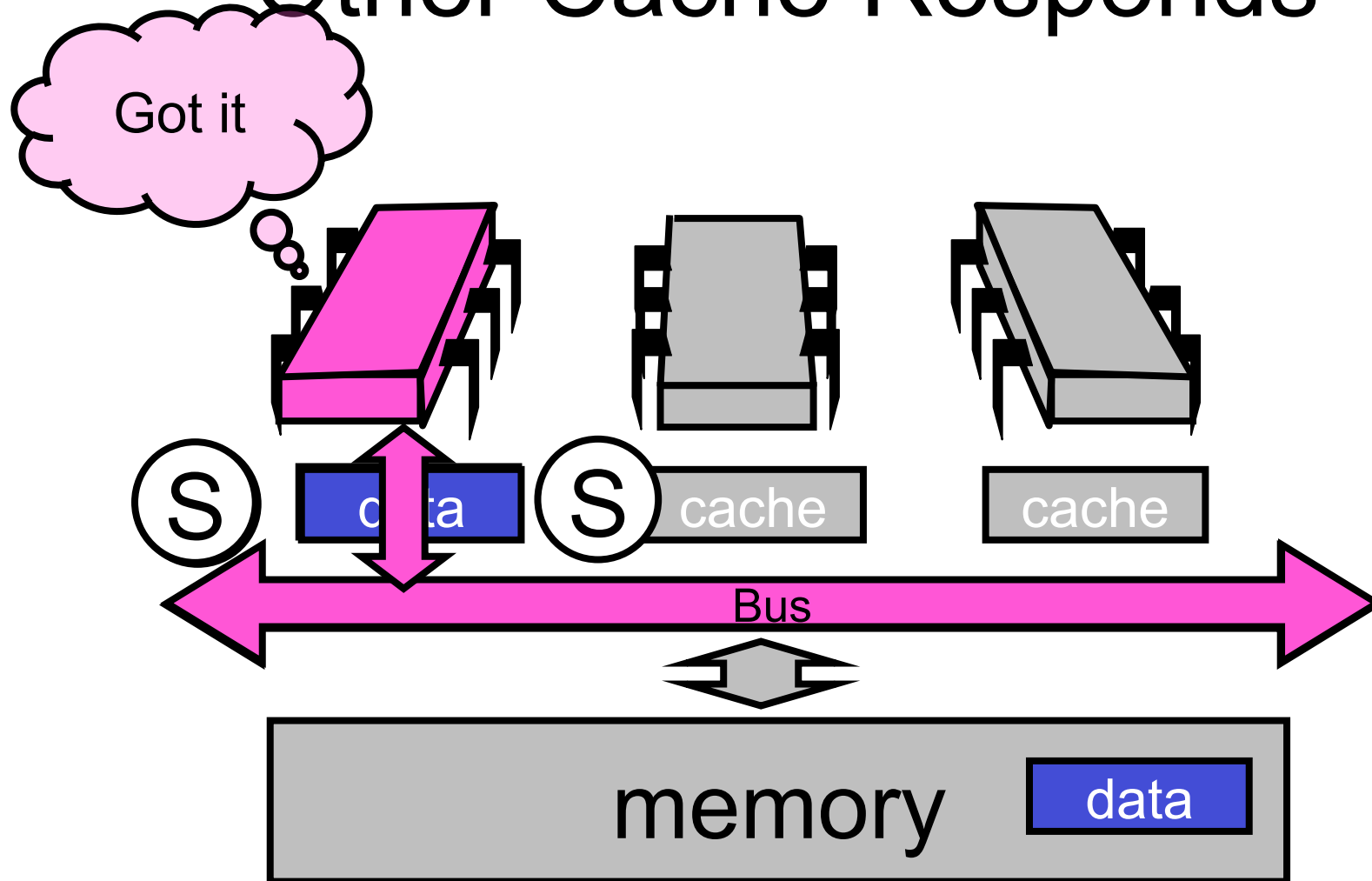
Processor Issues Load Request



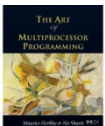
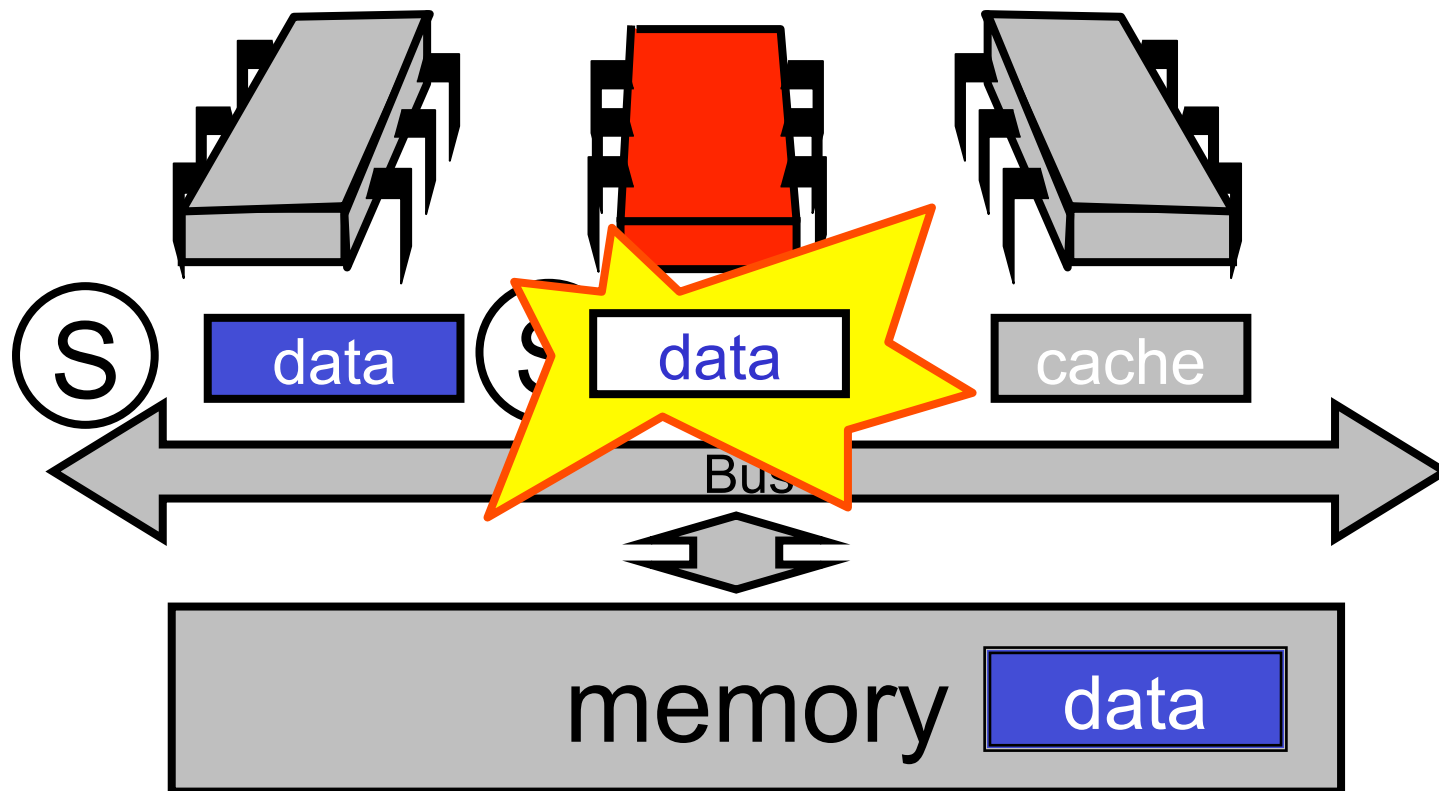
Processor Issues Load Request



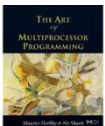
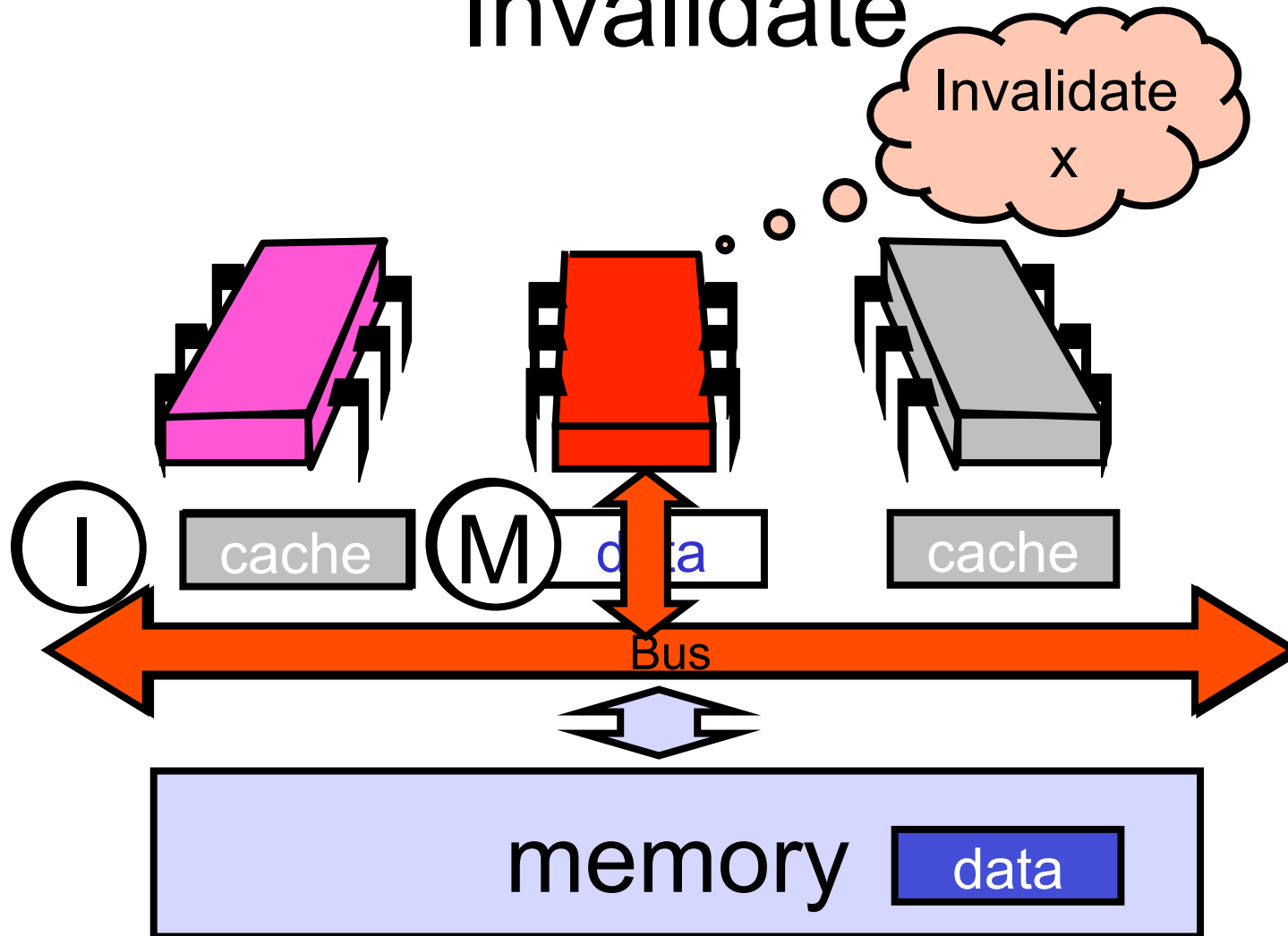
Other Cache Responds



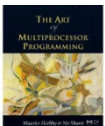
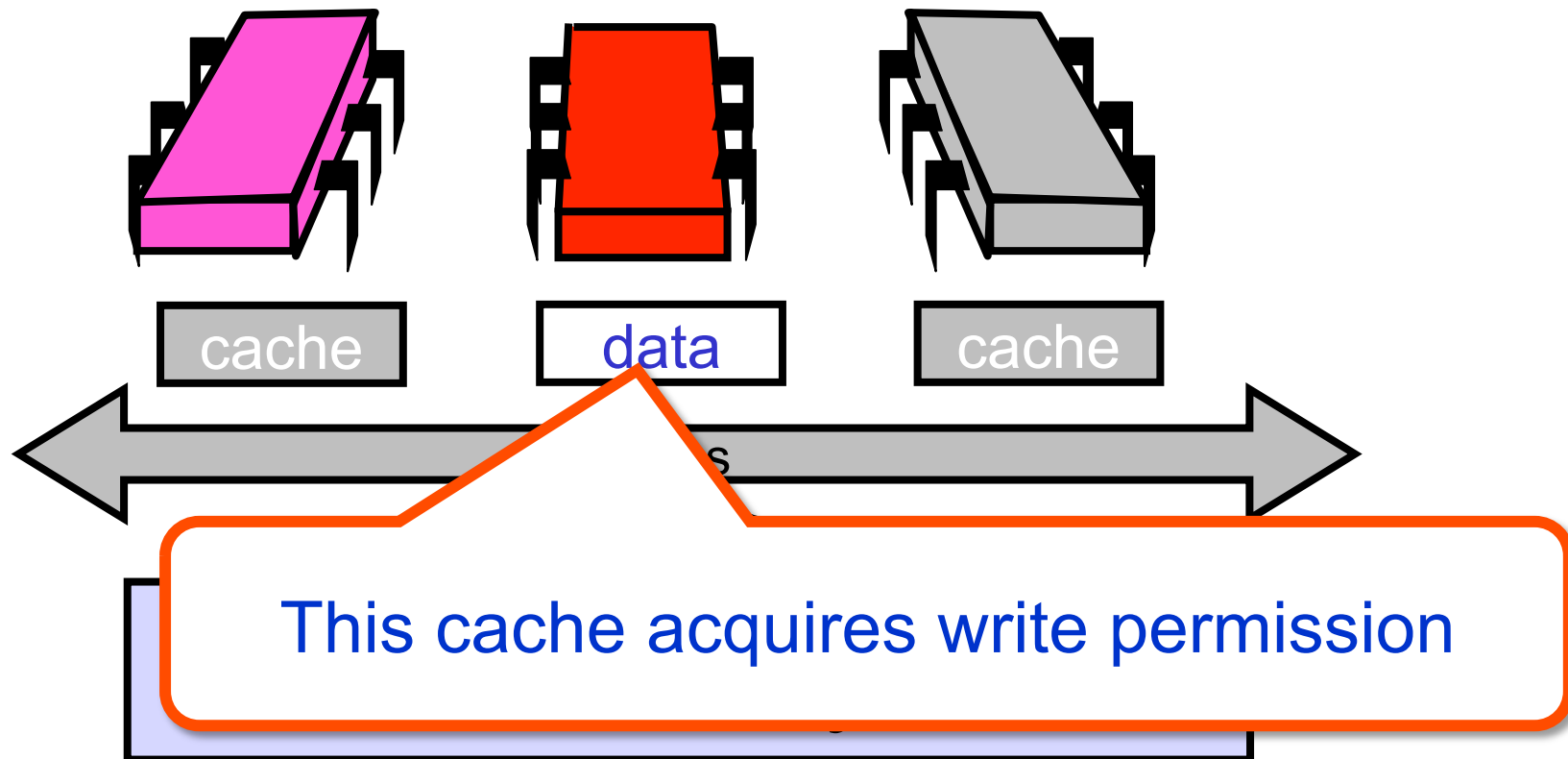
Modify Cached Data



Invalidate

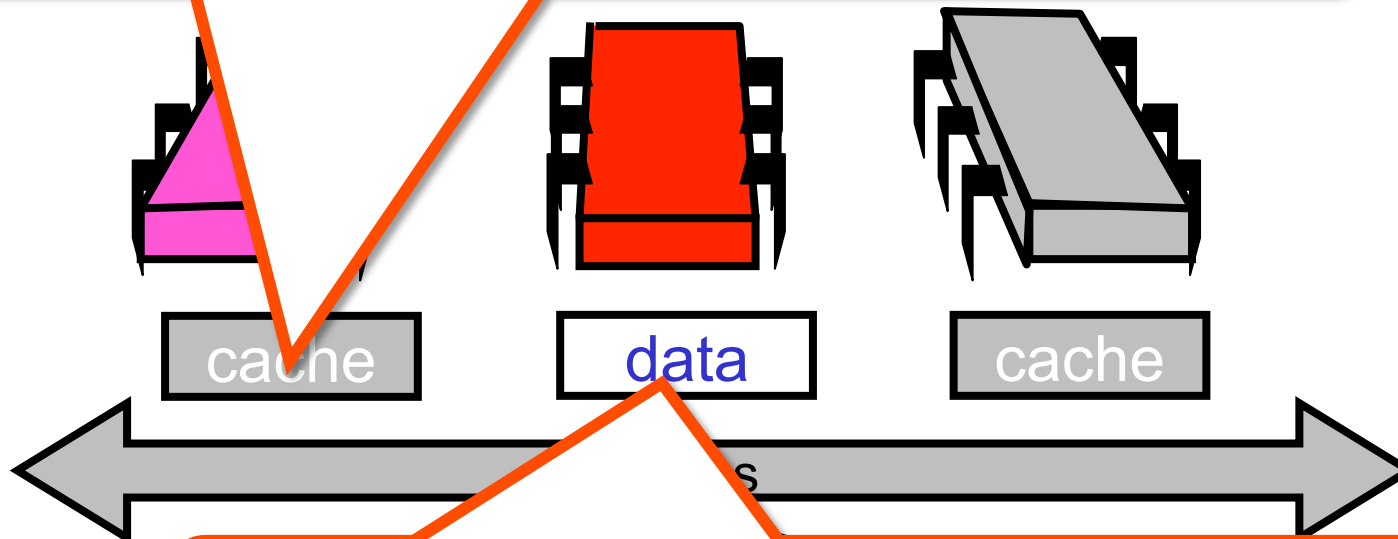


Invalidate

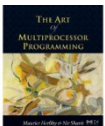


Invalidate

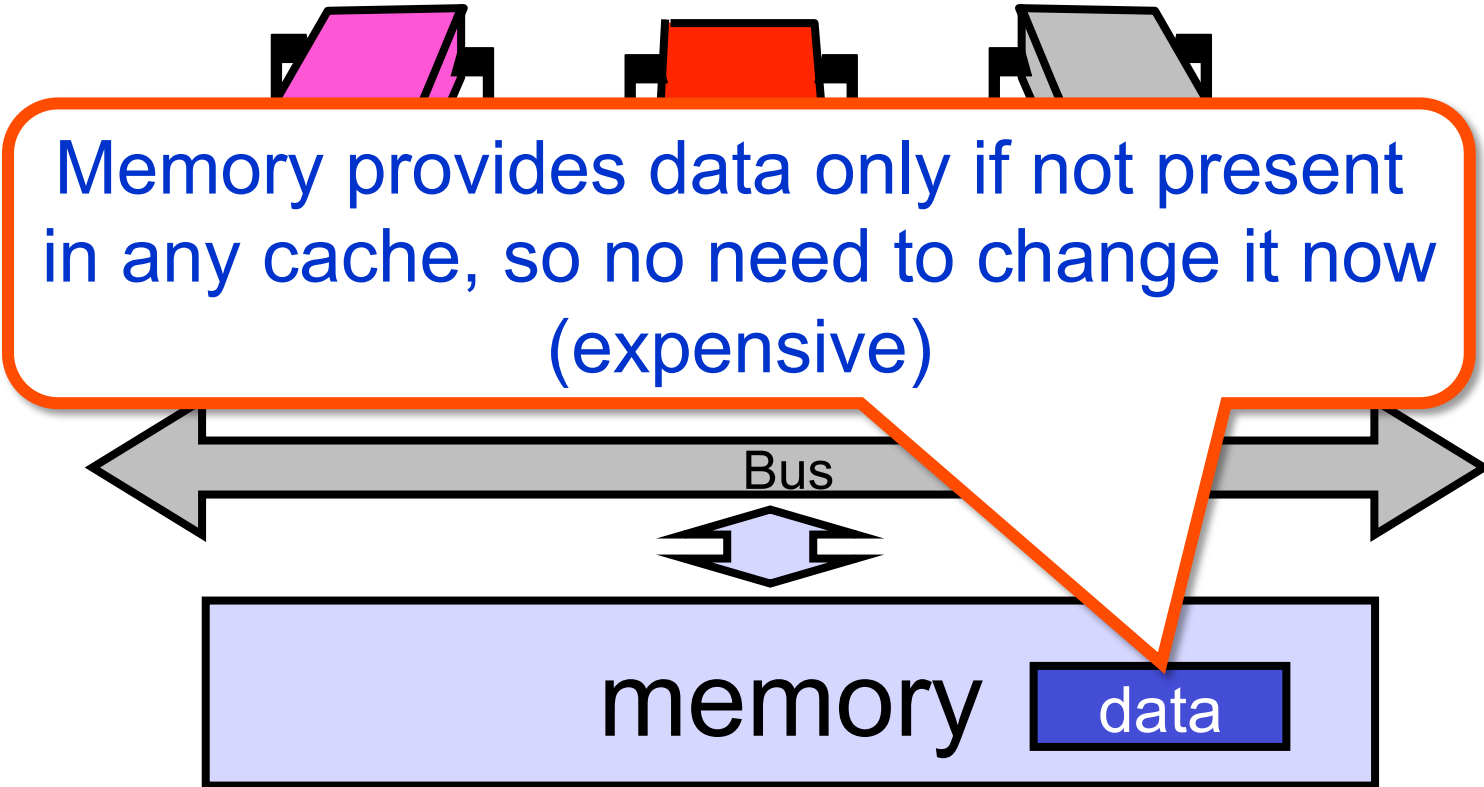
Other caches lose read permission



This cache acquires write permission

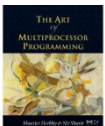


Invalidate

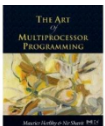
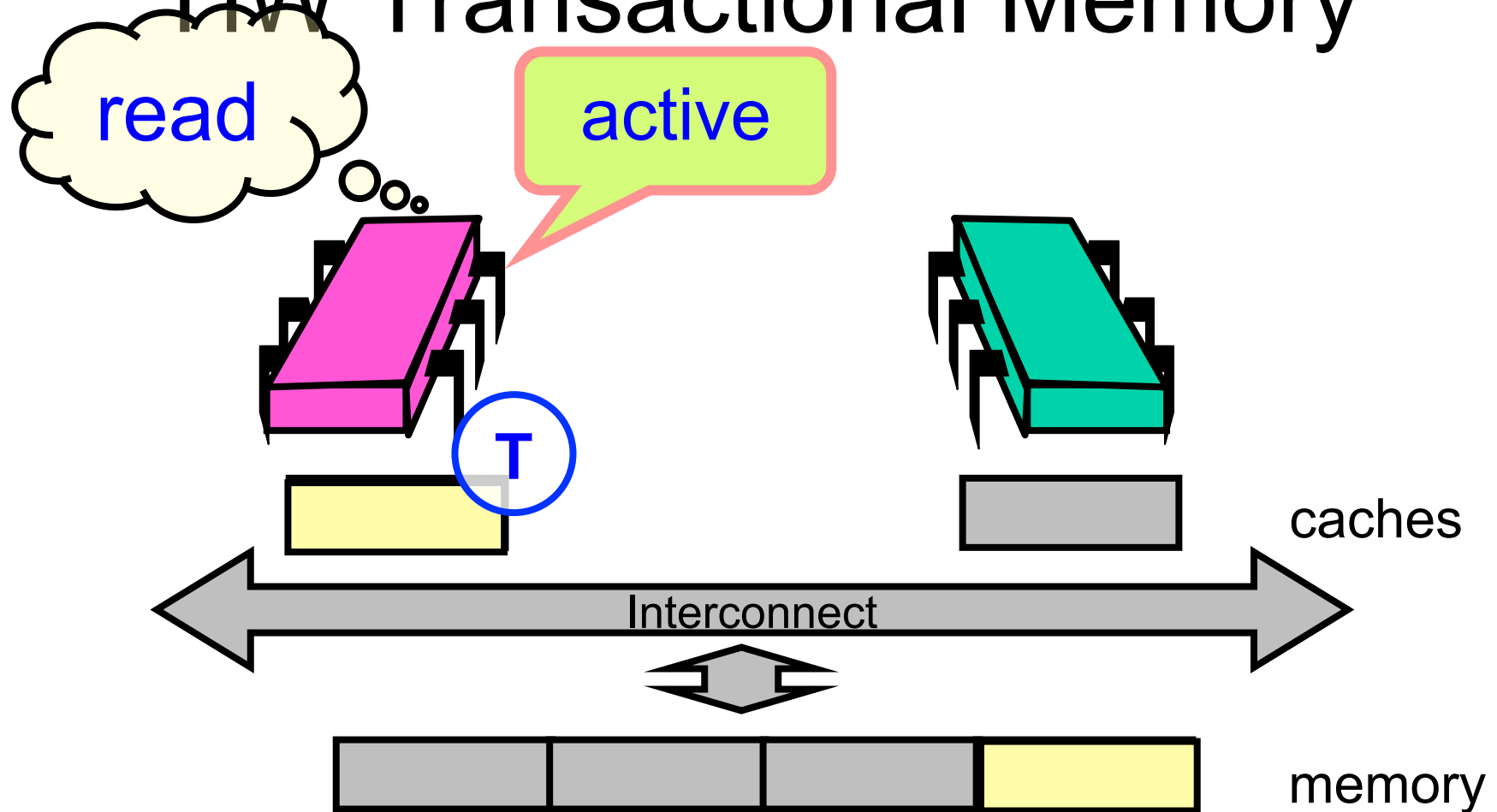


Memory provides data only if not present in any cache, so no need to change it now (expensive)

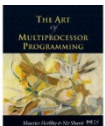
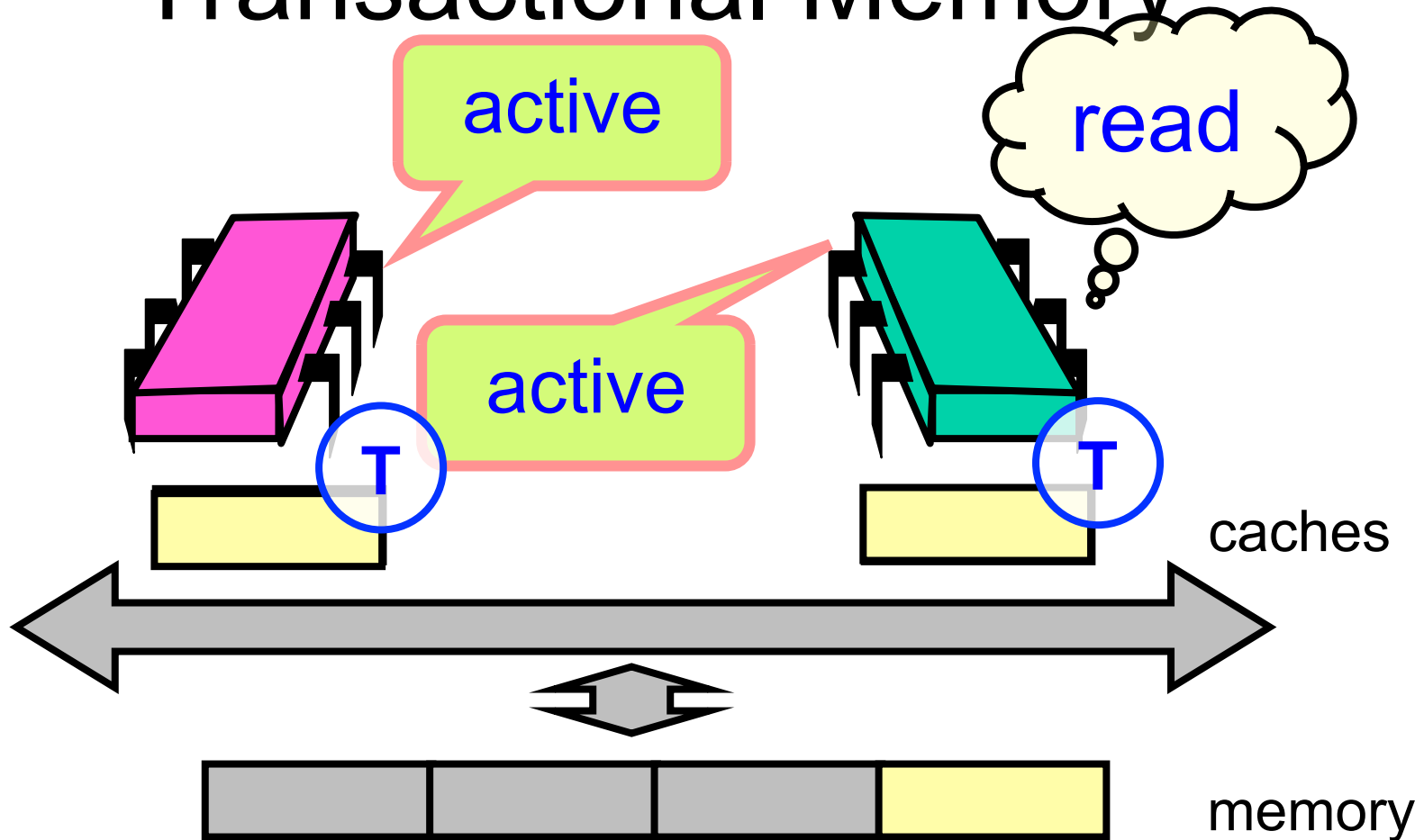
The diagram illustrates a cache invalidation scenario. At the top, three cache units are shown with pink, red, and grey fronts. Below them is a large orange-bordered box containing the text 'Memory provides data only if not present in any cache, so no need to change it now (expensive)'. An orange arrow points from this box to a 'data' block within a 'memory' block. The 'memory' block is a light blue rectangle at the bottom. Above it is a grey double-headed arrow labeled 'Bus'. Between the bus and the memory block is a blue double-headed arrow. The 'data' block is a smaller dark blue rectangle inside the 'memory' block.



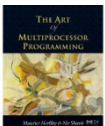
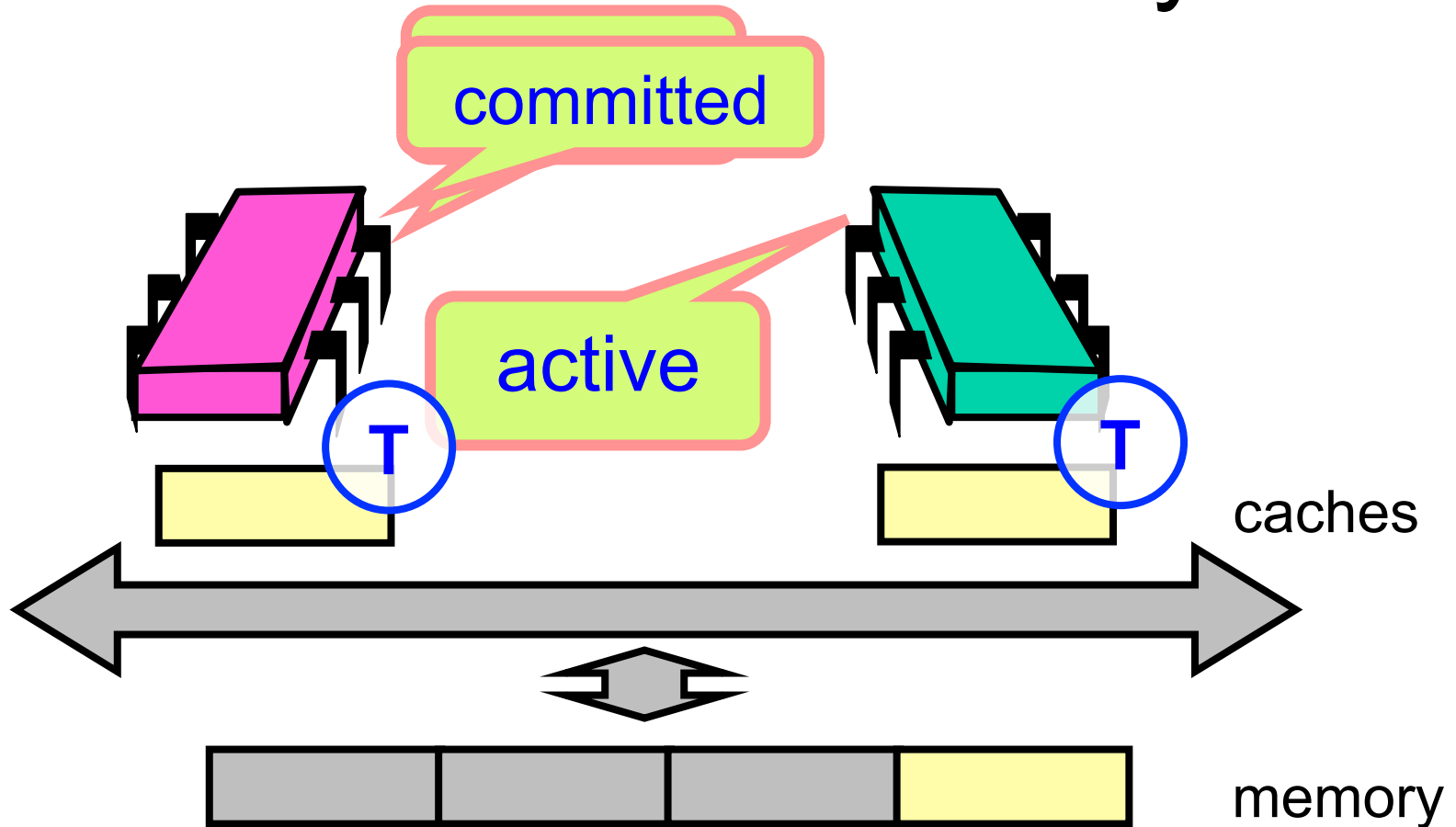
HW Transactional Memory



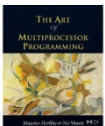
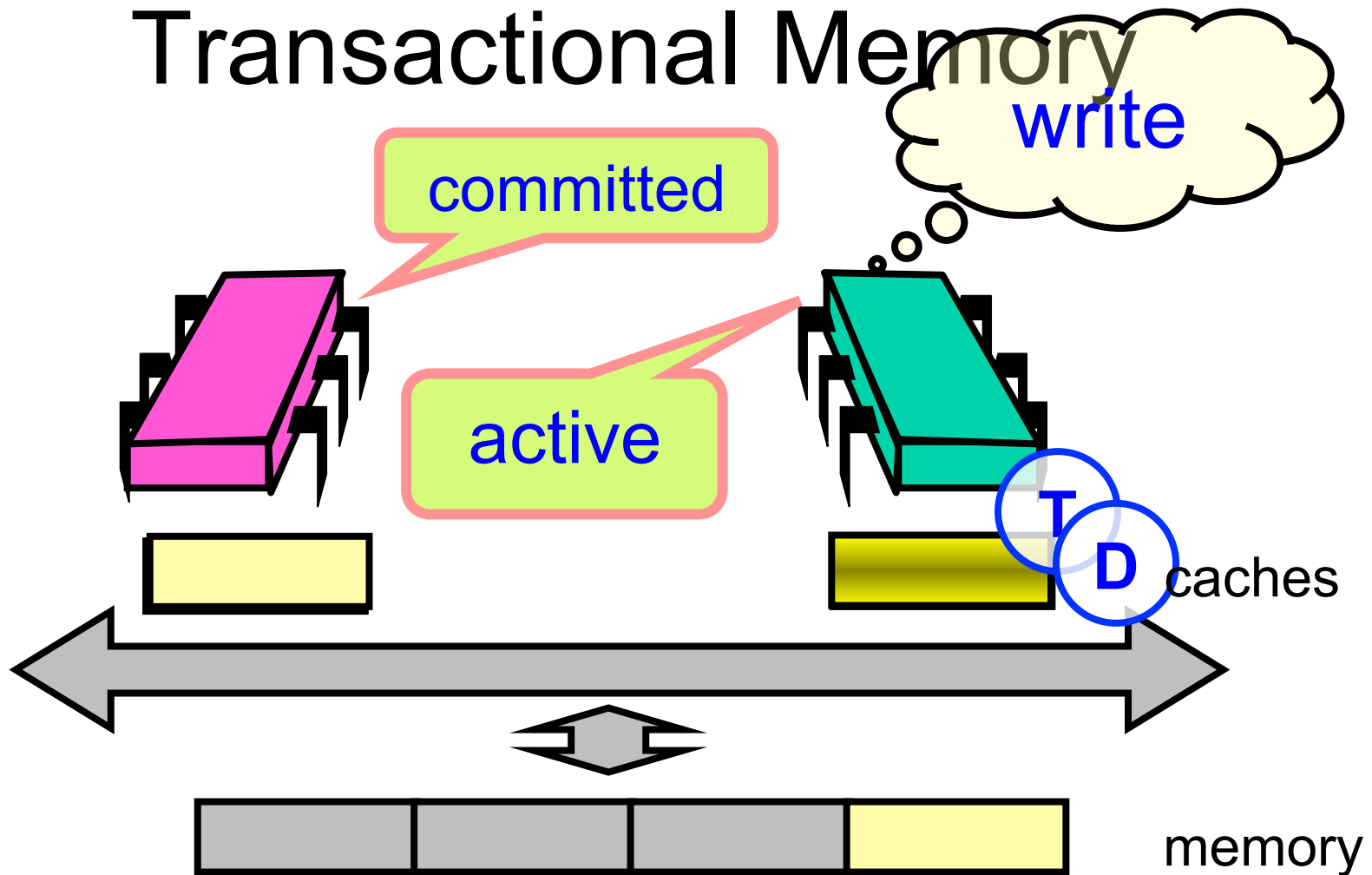
Transactional Memory

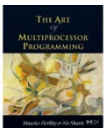
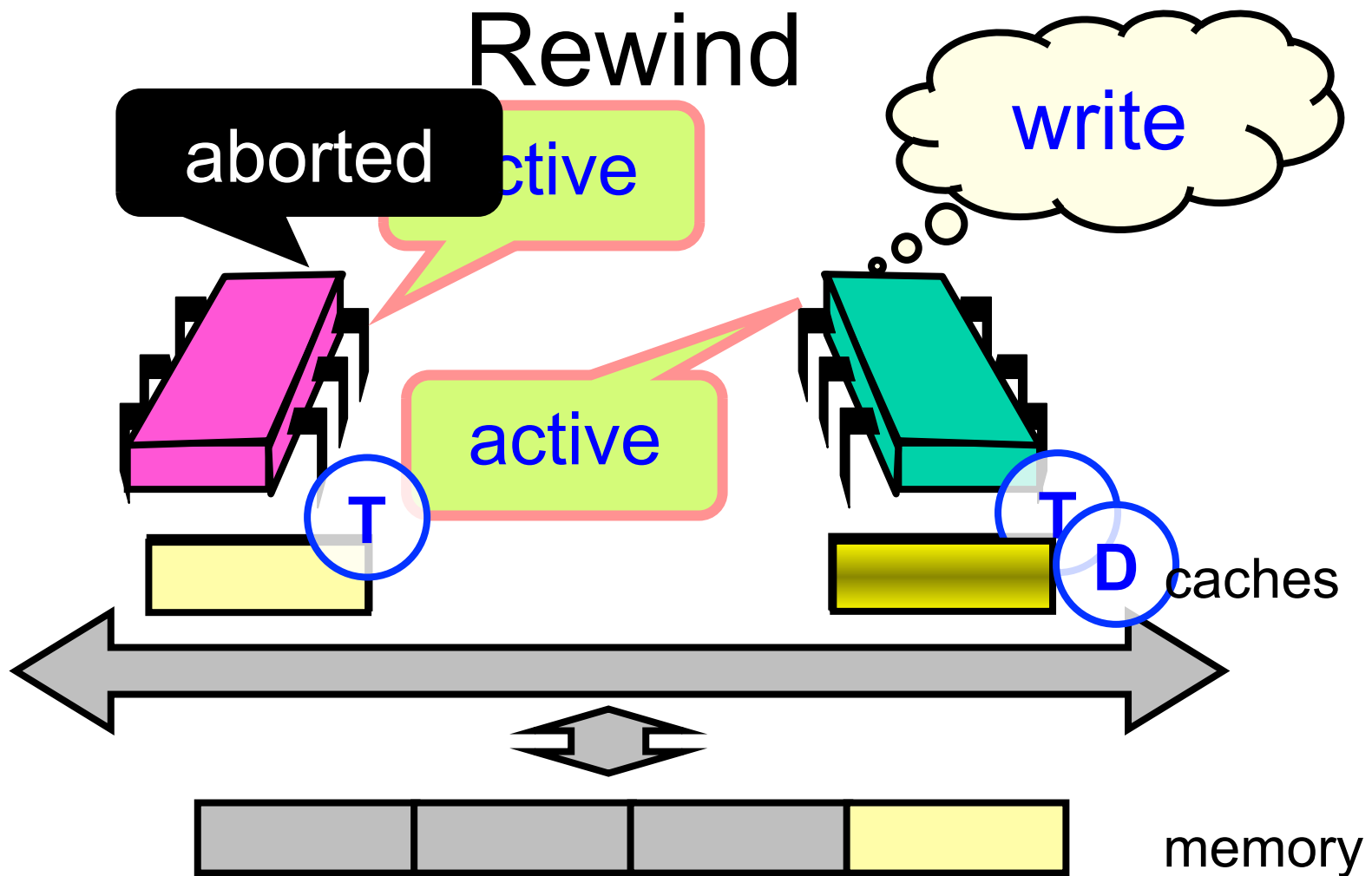


Transactional Memory



Transactional Memory





Transaction Commit

At Commit point ...

No cache conflicts? We win.

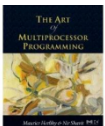
Mark transactional cache entries

Was: read-only, Now: valid

Was: modified, Now: dirty

(will be written back)

That's (almost) everything!



Road Map

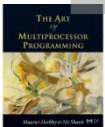
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

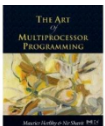
Research Questions



Hardware Transactional Memory (HTM)

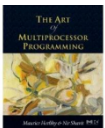
IBM's Blue Gene/Q & System Z & Power8

Intel's Haswell TSX extensions



Intel RTM

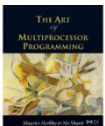
```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
    _xend()  
} else {  
    abort handler  
}
```



Intel RTM

```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
    _xend()  
} else {  
    abort handler  
}
```

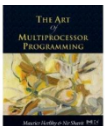
**start a speculative
transaction**



Intel RTM

```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
    _xend()  
} else {  
    abort handler  
}
```

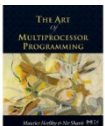
**If you see this, you are
inside a transaction**



Intel RTM

```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
    xend()  
} else {  
    abort handler  
}
```

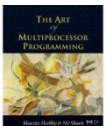
**If you see anything else,
your transaction aborted**



Intel RTM

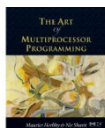
```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
    _xend()  
} else {  
    abort handler  
}
```

**you could retry the
transaction, or take an
alternative path**



Abort codes

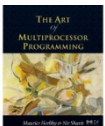
```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
} else if (status & _XABORT_EXPLICIT) {  
    aborted by user code  
} else if (status & _XABORT_CONFLICT) {  
    read-write conflict  
} else if (status & _XABORT_CAPACITY) {  
    cache overflow  
} else {  
    ...  
}
```



Abort codes

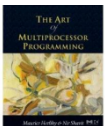
```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
} else if (status & _XABORT_EXPLICIT) {  
    aborted by user code  
} else if (status & _XABORT_CONFLICT) {  
    read-write conflict  
} else if (status & _XABORT_CAPACITY) {  
    cache overflow  
} else {  
    ...  
}
```

**speculative code can call
_xabort()**



Abort codes

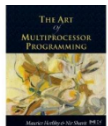
```
if (status & _XABORT_STARTED) {  
    synchronization conflict  
    occurred (maybe retry)  
}  
else if (status & _XABORT_EXPLICIT) {  
    aborted by user code  
}  
else if (status & _XABORT_CONFLICT) {  
    read-write conflict  
}  
else if (status & _XABORT_CAPACITY) {  
    cache overflow  
}  
else {  
    ...  
}
```



Abort codes

```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
} else if (status == _XABORT_EXPLICIT) {  
    abort  
} else if (status == _XABORT_CONFLICT) {  
    read-write conflict  
} else if (status & _XABORT_CAPACITY) {  
    cache overflow  
} else {  
    ...  
}
```

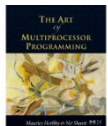
**read/write set too big
(maybe don't retry)**



Abort codes

```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
}  
else if (status & _XABORT_EXPLICIT) {  
    aborted by user code  
}  
else if (status & _XABORT_CONFLICT) {  
    read-write conflict  
}  
else if (status & _XABORT_CAPACITY) {  
    cache overflow  
}  
else {  
    ...  
}
```

other abort codes ...



Too Big



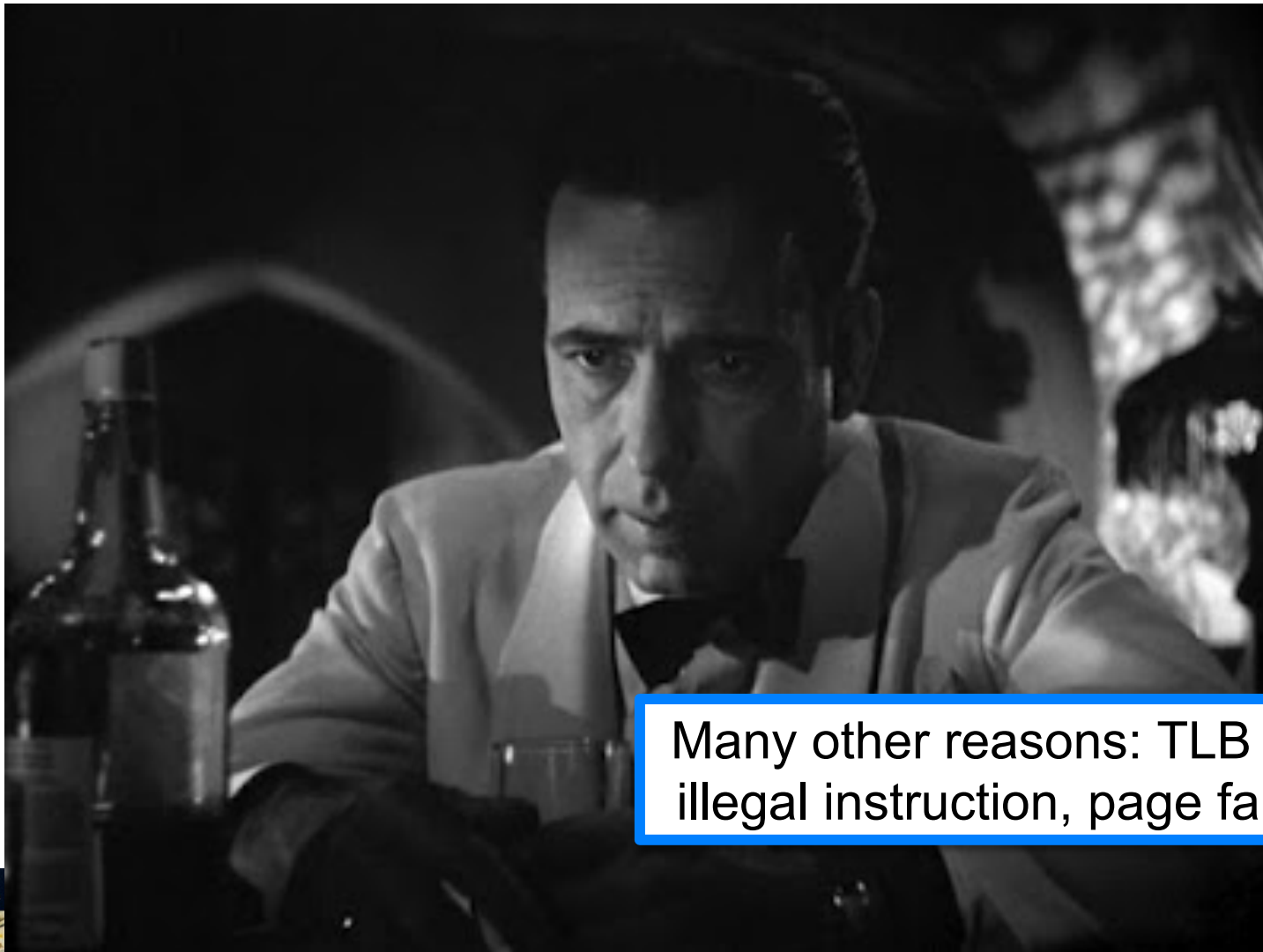
Transaction aborts if data set overflows caches, internal buffers

Too Slow



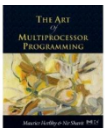
Transaction aborts on timer interrupt

Just Not in the Mood



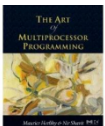
Many other reasons: TLB miss, illegal instruction, page fault ...

Hybrid Transactional Memory



Non-Speculative Fallback

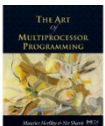
```
if (_xbegin() == _XBEGIN_STARTED) {  
    read lock state  
    if (lock taken) _xabort();  
    work;  
    _xend()  
} else {  
    lock->lock();  
    work;  
    lock->unlock();  
}
```



Non-Speculative Fallback

```
if (_xbegin() == _XBEGIN_STARTED) {  
    read lock state  
    if (lock taken) _xabort();  
    work;  
    xend();  
}
```

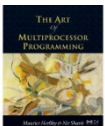
**reading lock ensures that
transaction will abort if another
thread acquires lock**



Non-Speculative Fallback

```
if (_xbegin() == _XBEGIN_STARTED) {  
    read lock state  
    if (lock taken) _xabort();  
    work;  
    _xend()  
} else {  
    lock ...  
    work ...  
    lock ...  
}
```

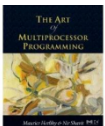
**abort if another thread has
acquired lock**



Non-Ordered Locks Follow

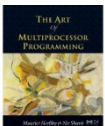
**on abort, acquire lock & do work
(aborting concurrent speculative
transactions)**

```
if  
re  
if (lock taken) _xabort();  
work;  
xend()  
} else {  
lock->lock();  
work;  
lock->unlock();  
}
```



Lock Elision

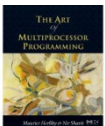
```
<HLE acquire prefix> lock();  
do work;  
<HLE release prefix> unlock();
```



Lock Elision

```
<HLE acquire prefix> lock();  
do work;  
<HLE release prefix> unlock();
```

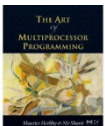
**first time around,
read lock and
execute speculatively**



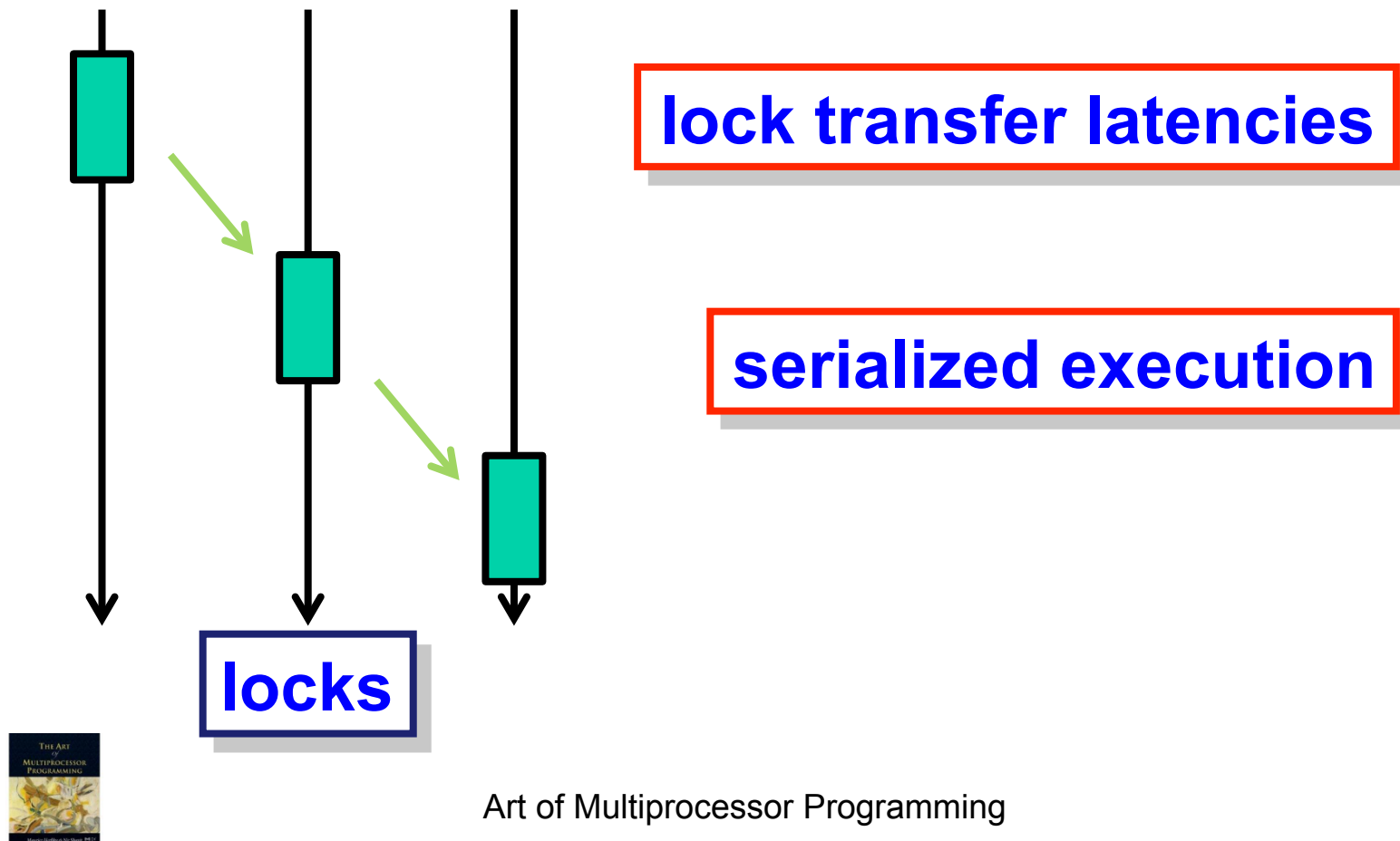
Lock Elision

```
<HLE acquire prefix> lock();  
do work;  
<HLE release prefix> unlock();
```

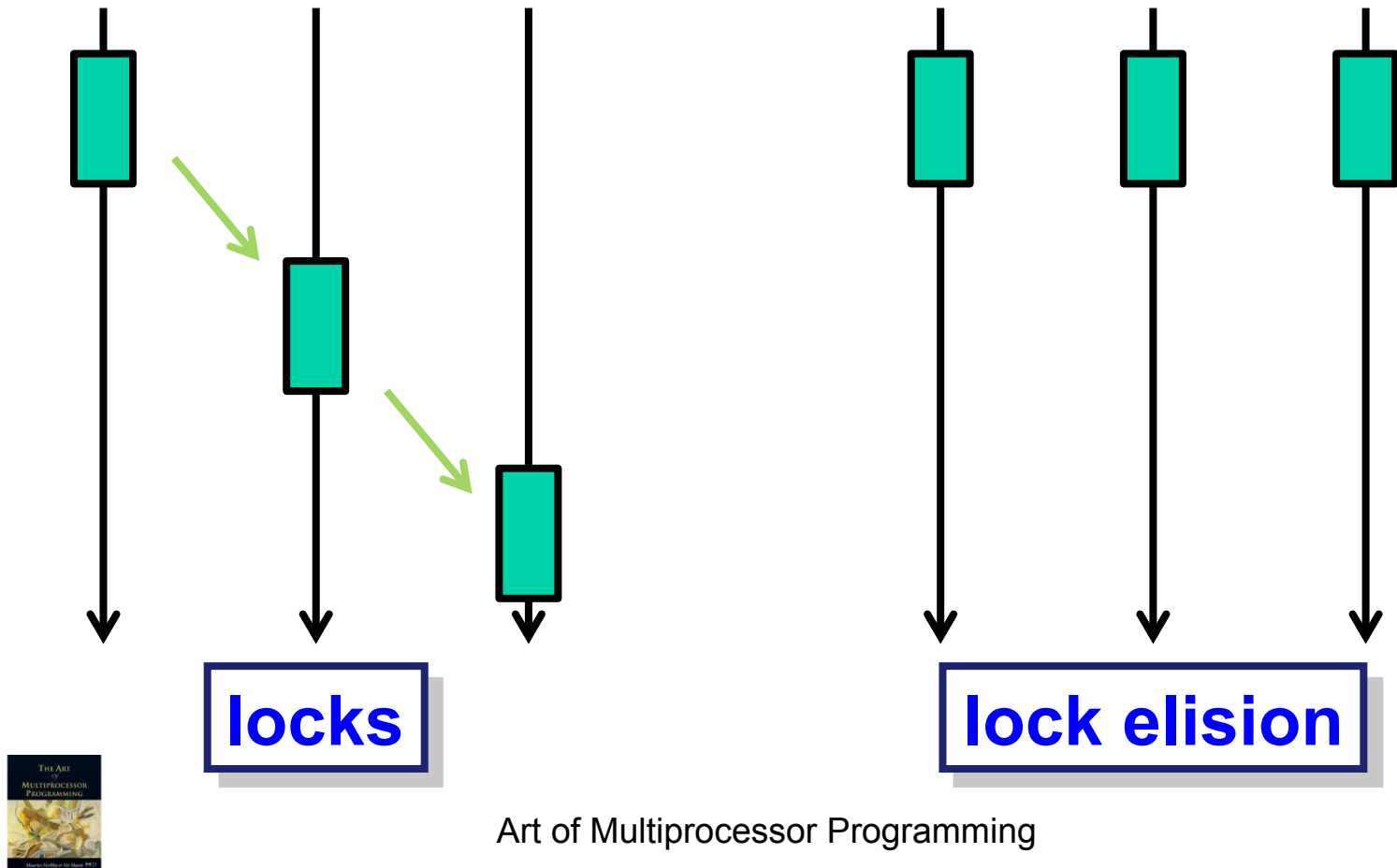
**if speculation fails,
no more Mr. Nice Guy,
acquire the lock**



Conventional Locks



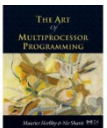
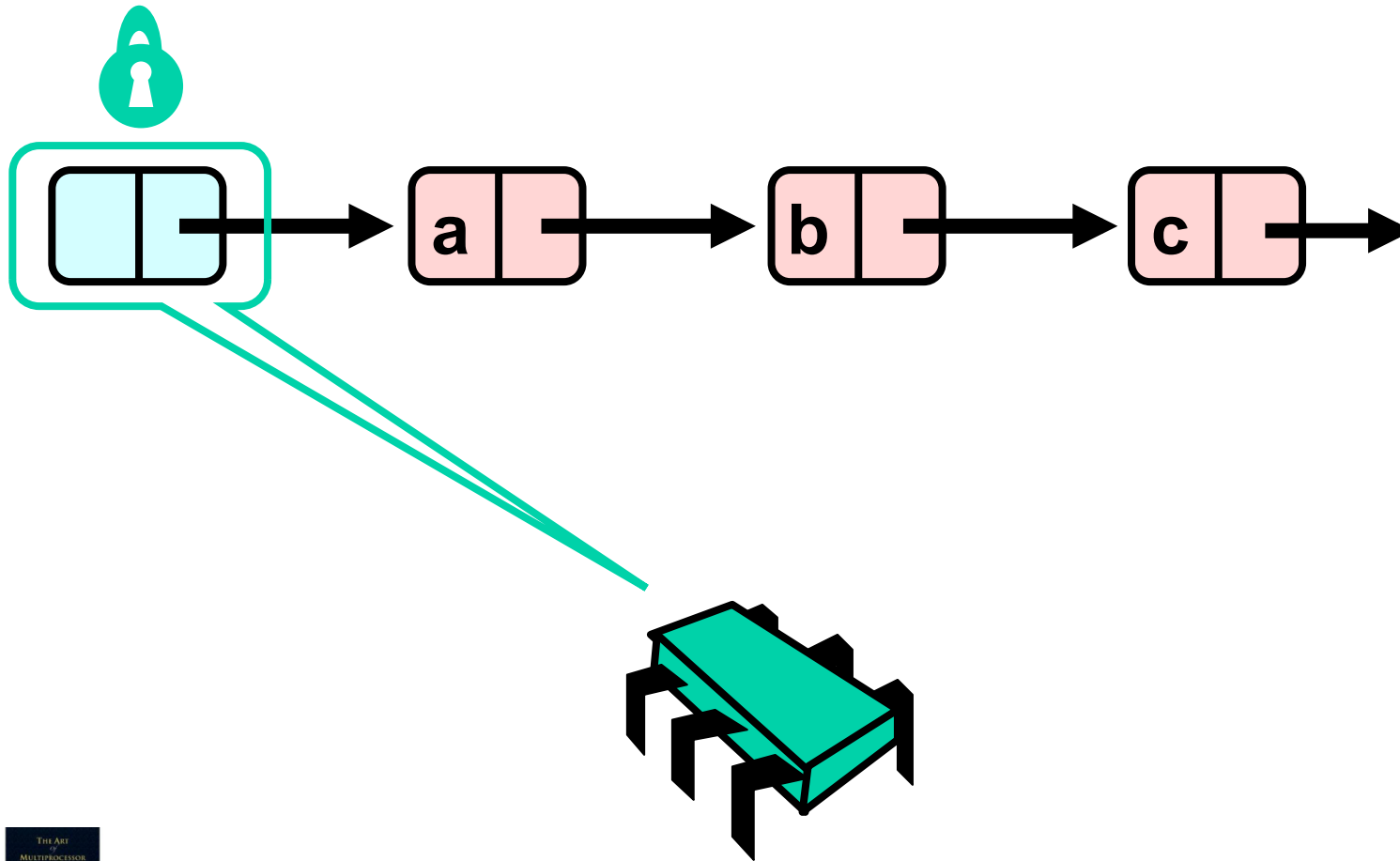
Lock Elision



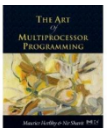
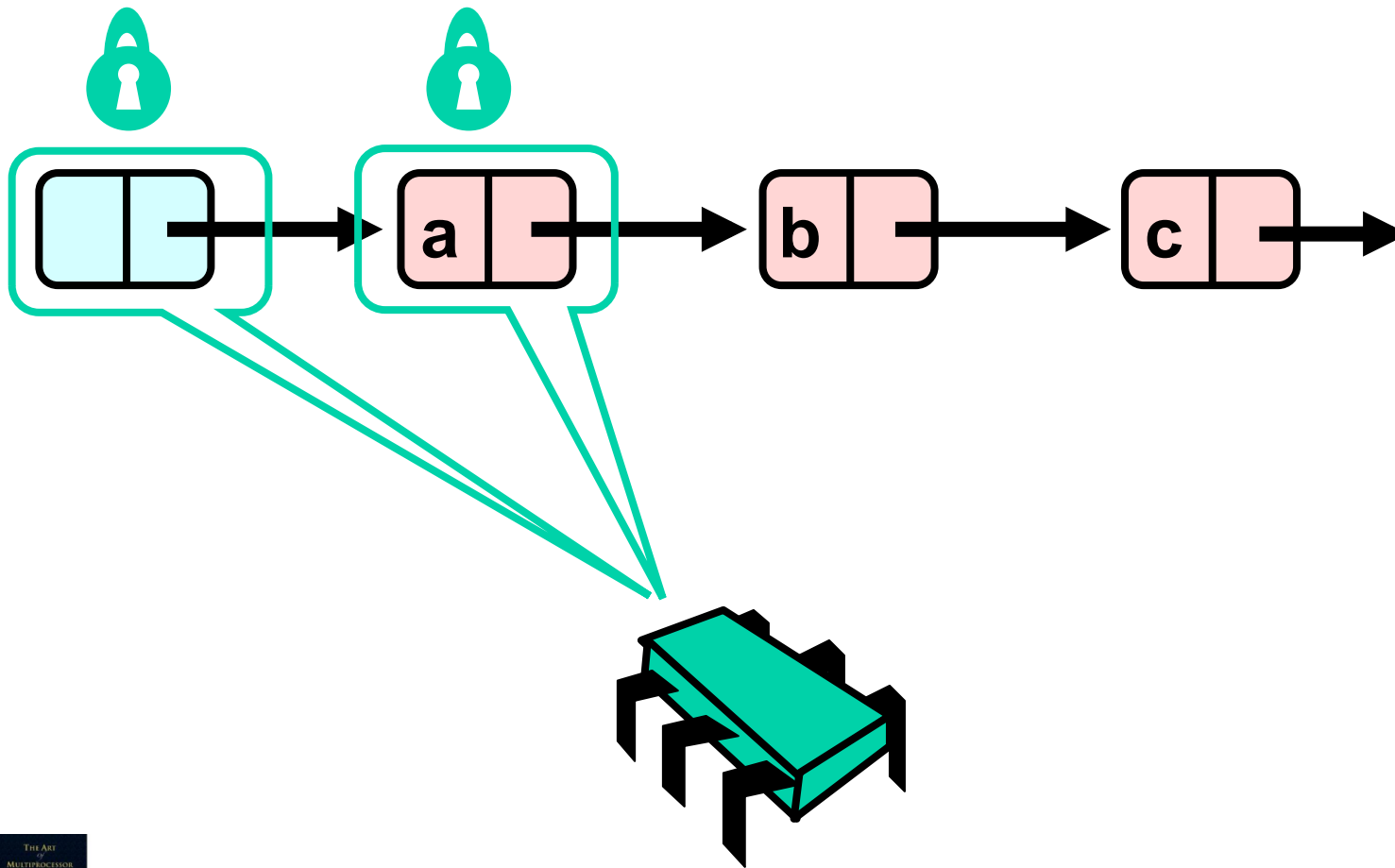
Lock Teleportation



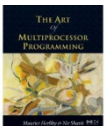
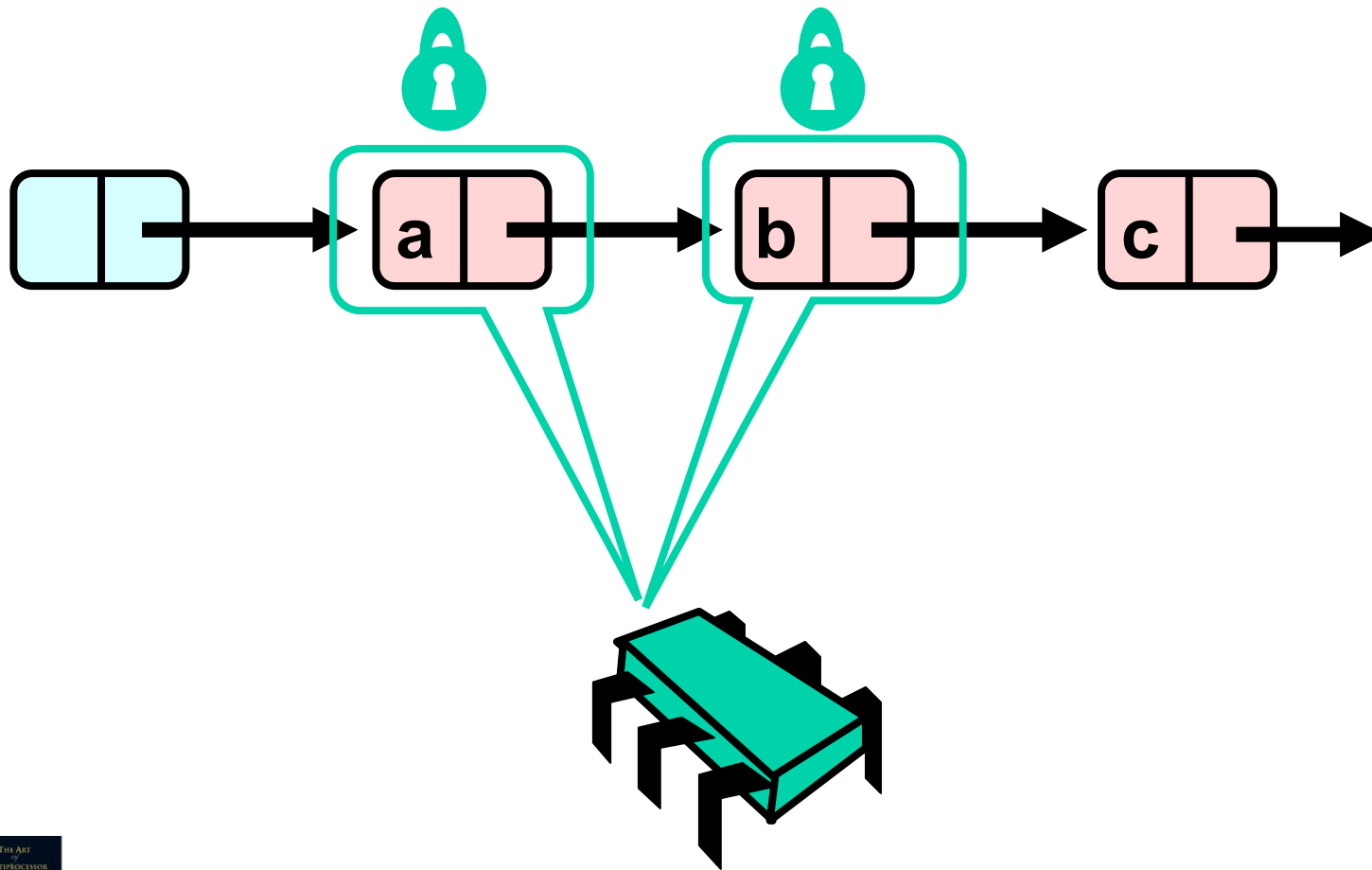
Hand-over-Hand locking



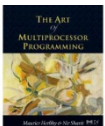
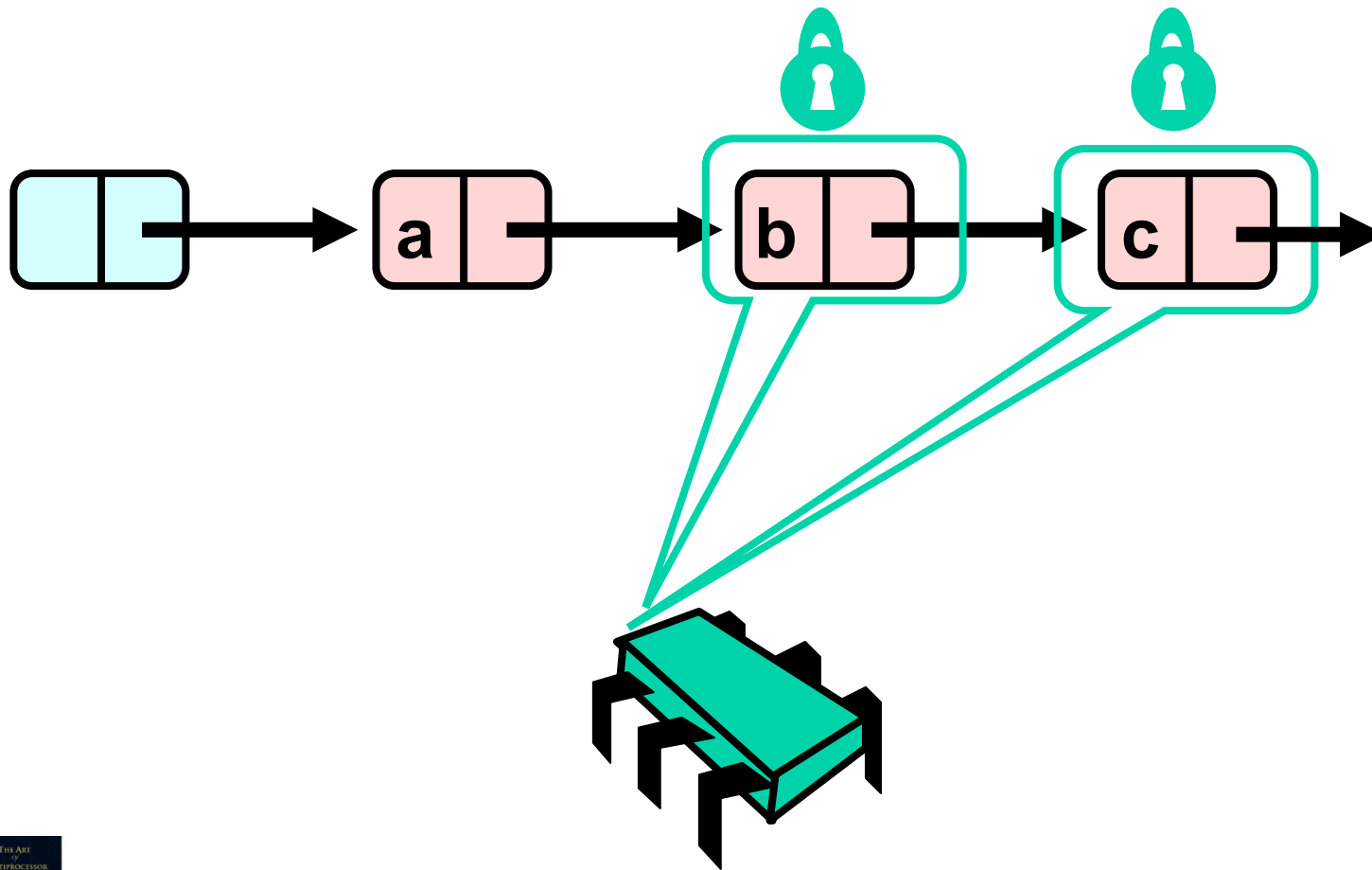
Hand-over-Hand locking



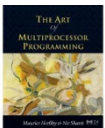
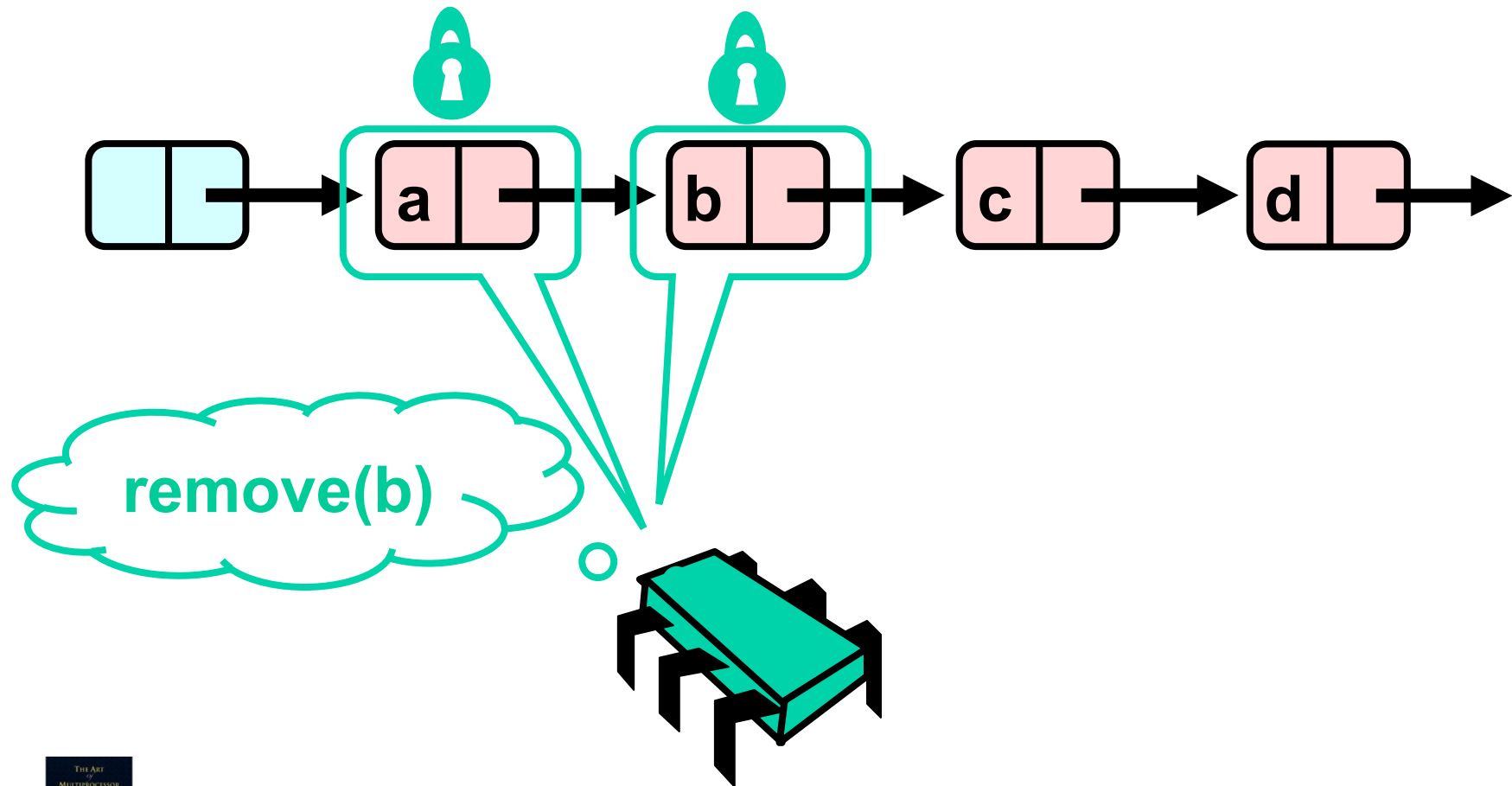
Hand-over-Hand locking



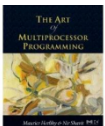
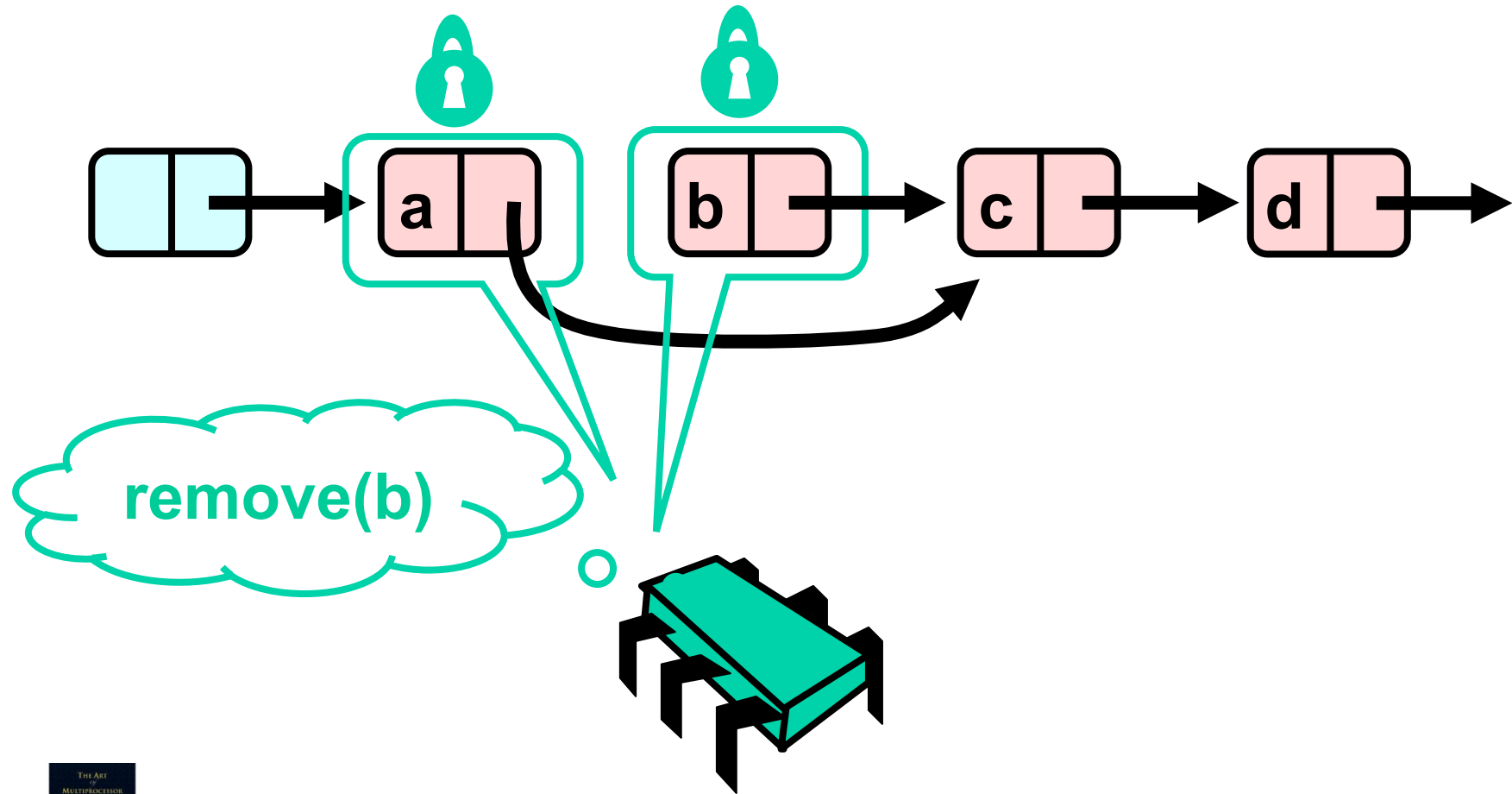
Hand-over-Hand locking



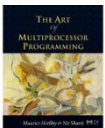
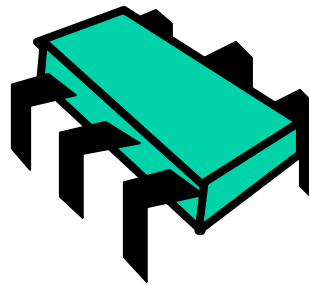
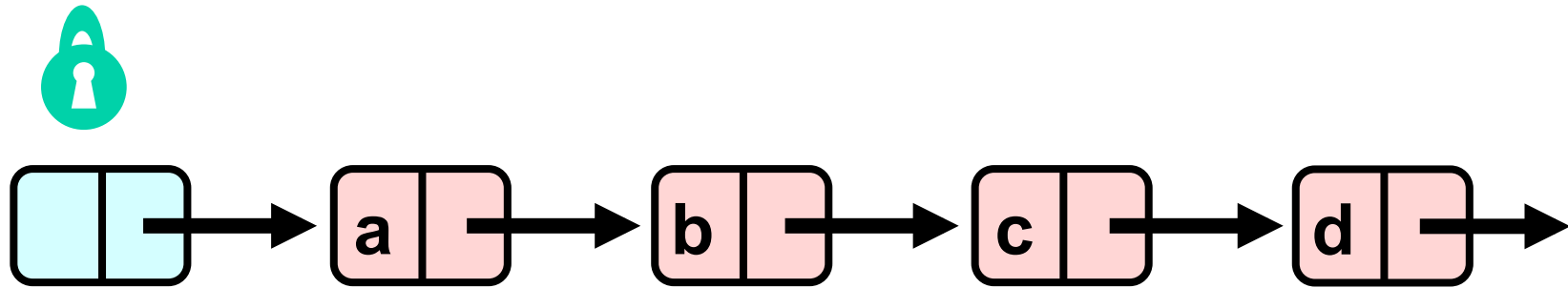
Removing a Node



Removing a Node

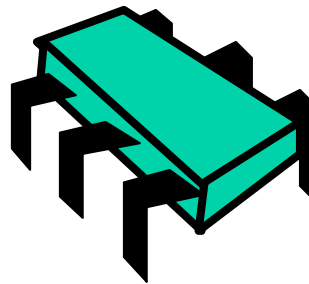
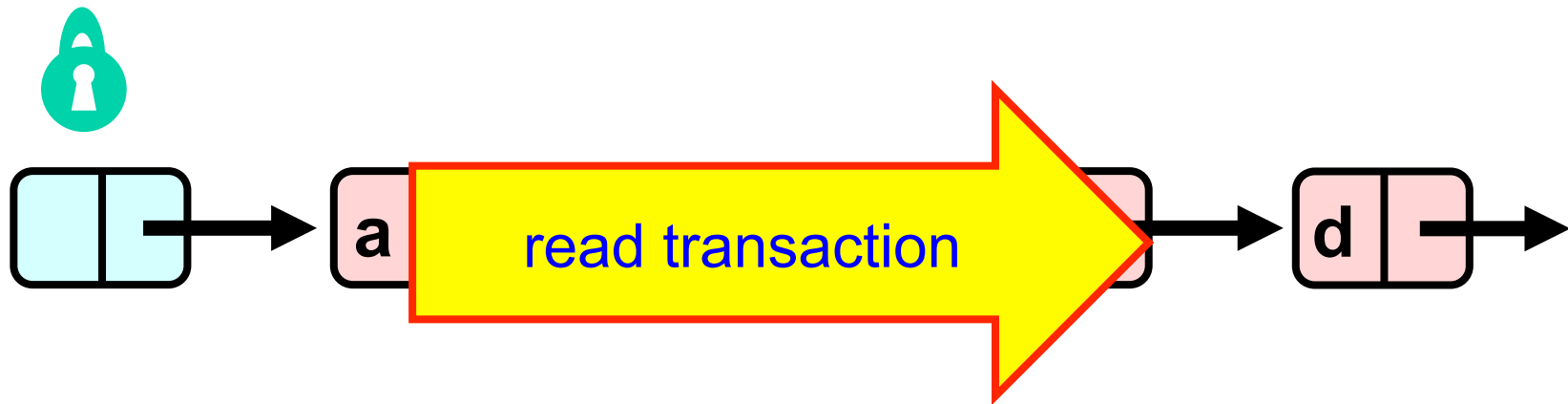


Lock Teleportation

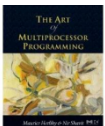


Art of Multiprocessor Programming

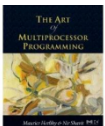
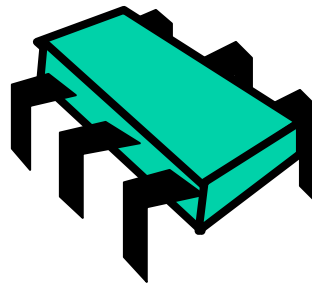
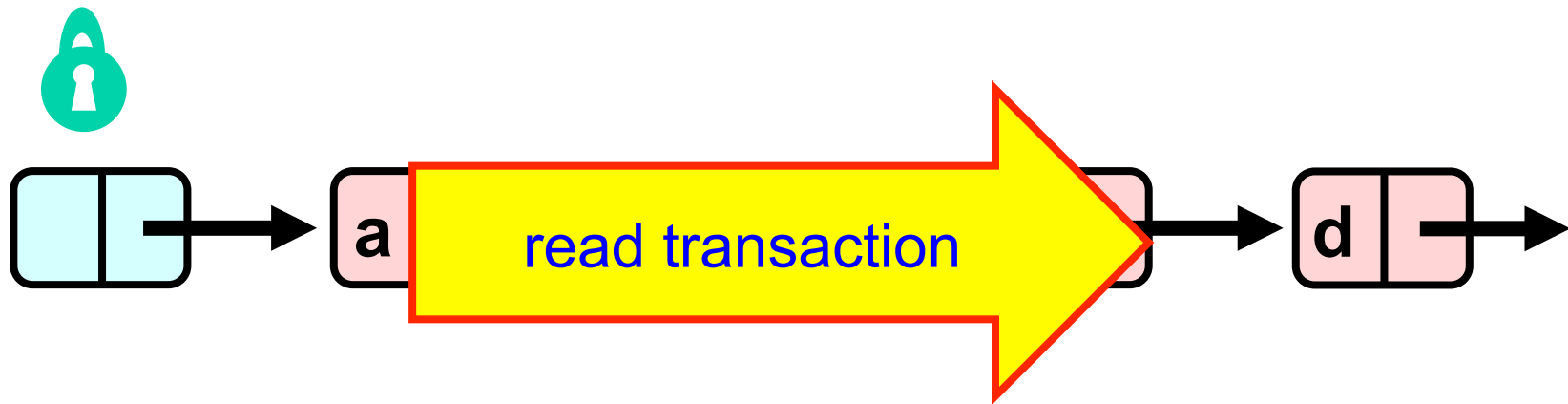
Lock Teleportation



Art of Multiprocessor Programming



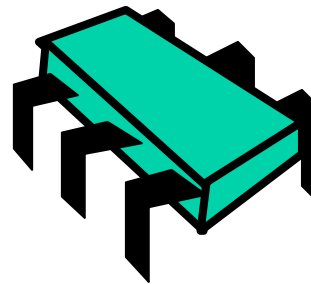
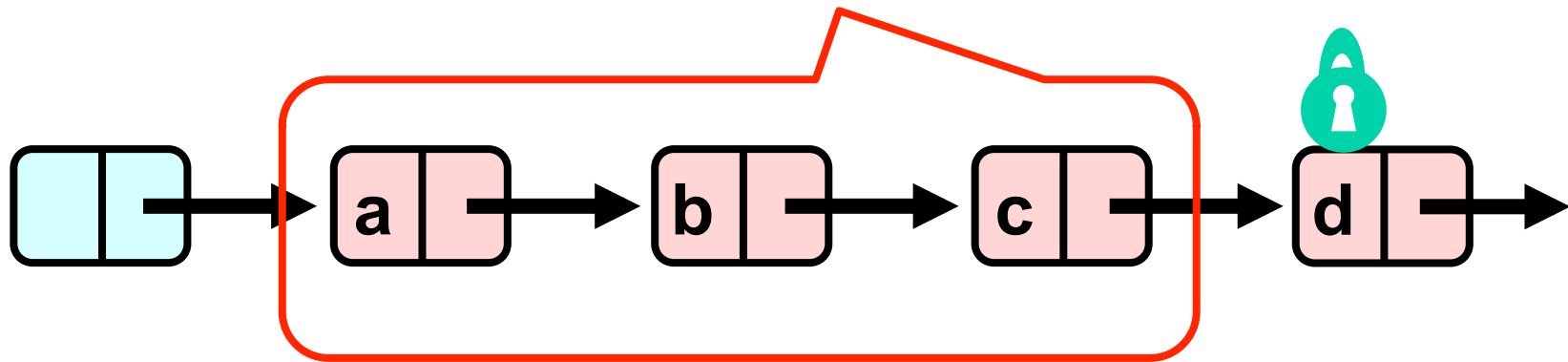
Lock Teleportation



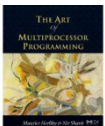
Art of Multiprocessor Programming

Lock Teleportation

no locks acquired



Art of Multiprocessor Programming



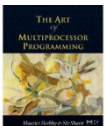
How Far to Teleport?

Too short?

Missed opportunity

Too far?

Transaction aborts, work lost

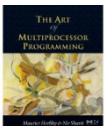


Adaptive Teleportation

On Success:

$$\text{limit} = \text{limit} + 1$$

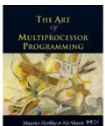
On Failure:

$$\text{limit} = \text{limit} / 2$$


```

Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            move lock
            _xend();
            teleportLimit++;
            return pred;
        } else {
            teleportLimit = teleportLimit/2
        }
    }
};

```

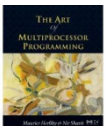



```

Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        i
        i
        locked node  
In sorted list
        BEGIN
        ele
        s
        move lock
        _xend();
        teleportLimit++;
        return pred;
    } else {
        teleportLimit = teleportLimit/2
    }
}

```

Value to search for

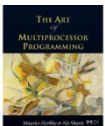


```

Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int pred;
        if (start->value > v) {
            move lock
            _xend();
            teleportLimit++;
            return pred;
        } else {
            teleportLimit = teleportLimit/2;
        }
    }
}

```

Returns locked node with value less than or equal to v

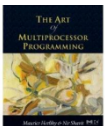


```

Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            m
            _
            teleportLimit++;
            return pred;
        } else {
            teleportLimit = teleportLimit/2
        }
    }
};

```

Try for a fixed number of times

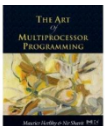


```

Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            move lock
            _xend();
            teleportLimit++;
            return pred;
        } else {
            telepor
        }
    }
};

```

Executed as read-only transaction

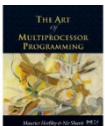


```

Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            move lock
            _xend();
            telep
            retur
        } else
            telep
    }
}

```

Thread-local variable that controls how far to traverse the list

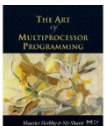


```

Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            move lock
            _xend();
            telep
            retur
        } else
            telep
    }
}

```

**Stop if either (1) we find
value **v**, or (2) we traverse
teleportLimit nodes**

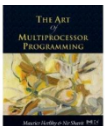


```

Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            move lock
            _xend();
            teleportLimit++;
        }
    }
}

```

Unlock starting node, lock final node



```

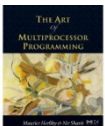
Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            move lock
            _xend();
            teleportLimit++;
            return pred;
        }
    }
}

```

Try to commit transaction

hit/2

}}};



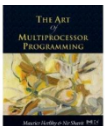
```

Node* teleport(Node* start, T v) {
    int retries
    while (--retries) {
        int distance
        if (xbegin
            traverse list up to threshold
            move lock
            xend();
            teleportLimit++;
            return last node;
        } else {
            teleportLimit = teleportLimit/2
        }
    }
}

```

On commit, advance
teleportLimit by 1,
and return locked node

teleportLimit++;
return last node;

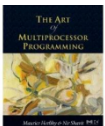


```

Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == XBEGIN_STARTED) {
            traverse
            move lock
            _xend();
            teleportLimit++;
            return last node;
        } else {
            teleportLimit = teleportLimit/2
        }
    }
};

```

**On abort, cut
teleportLimit in half**

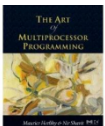


Lock-Based STMs

STMs come in different forms:

Lock-Free

Lock-based

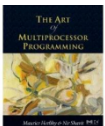


Lock-Based STM

But, didn't you just say that locks are **evil**?

For applications, **yes!**

For run-time systems
written by experts,
maybe not



Lock-Based STMs

Each transaction keeps

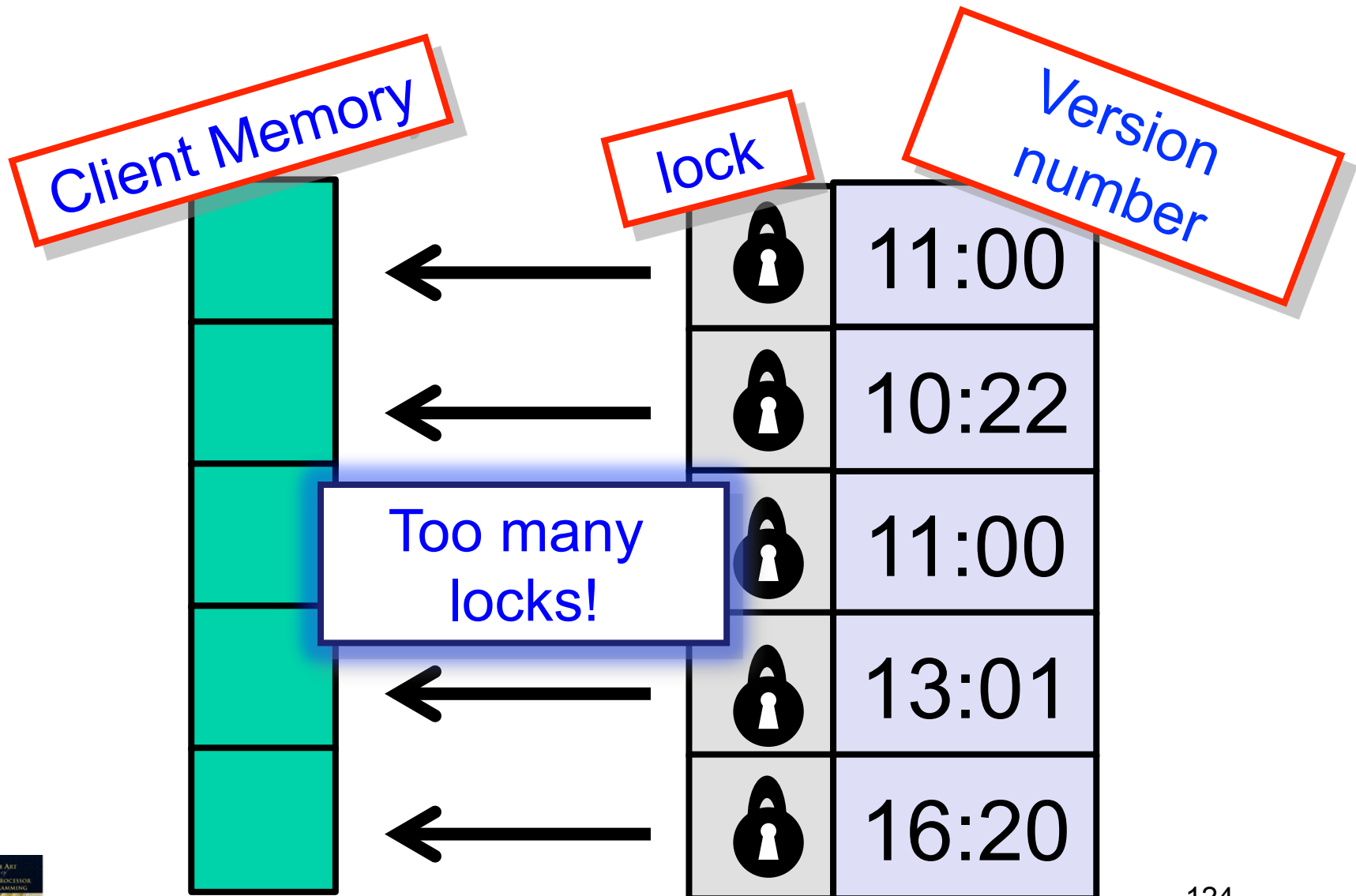
Read Set: locations and values read

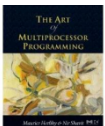
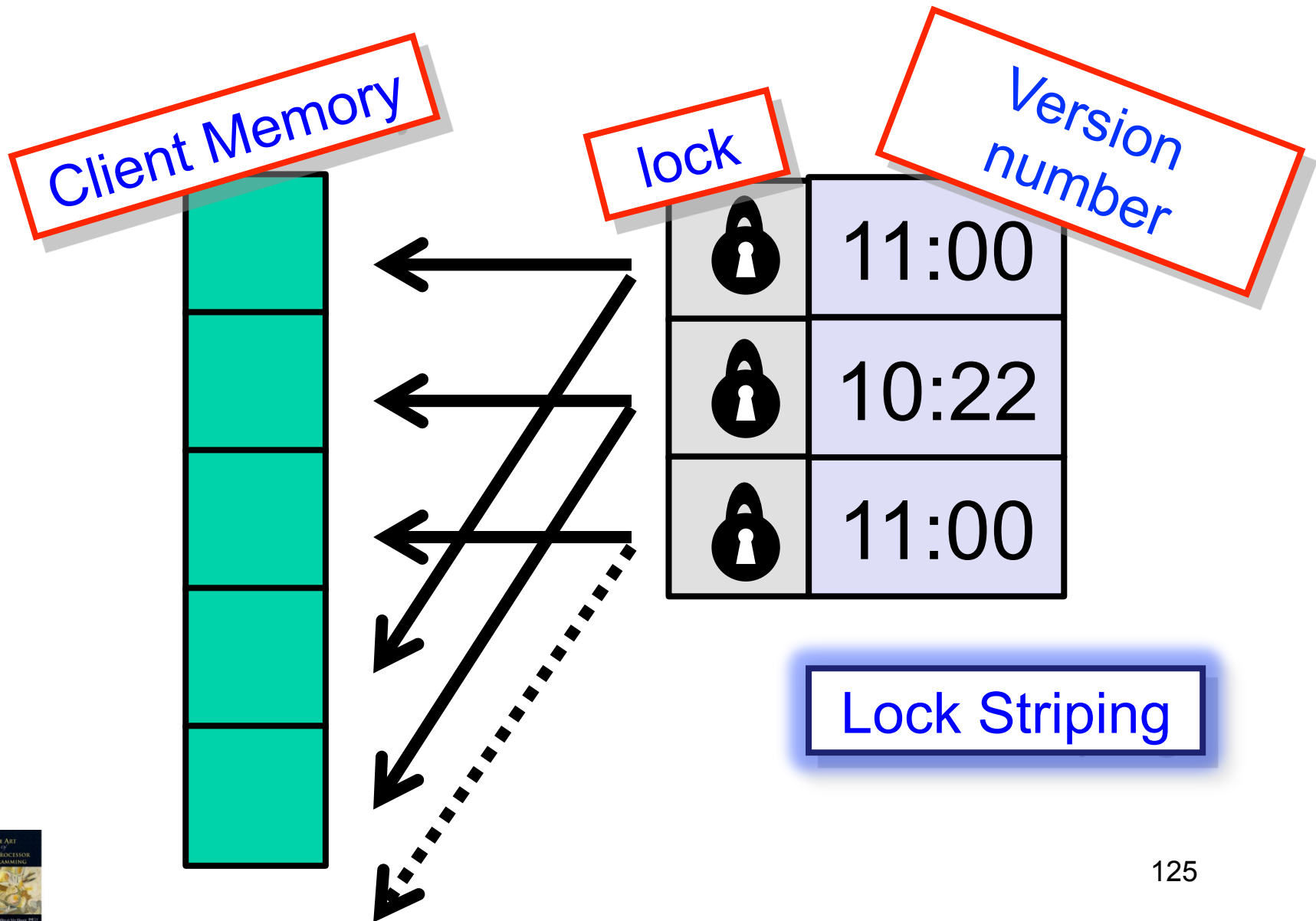
Write Set: locations and values written

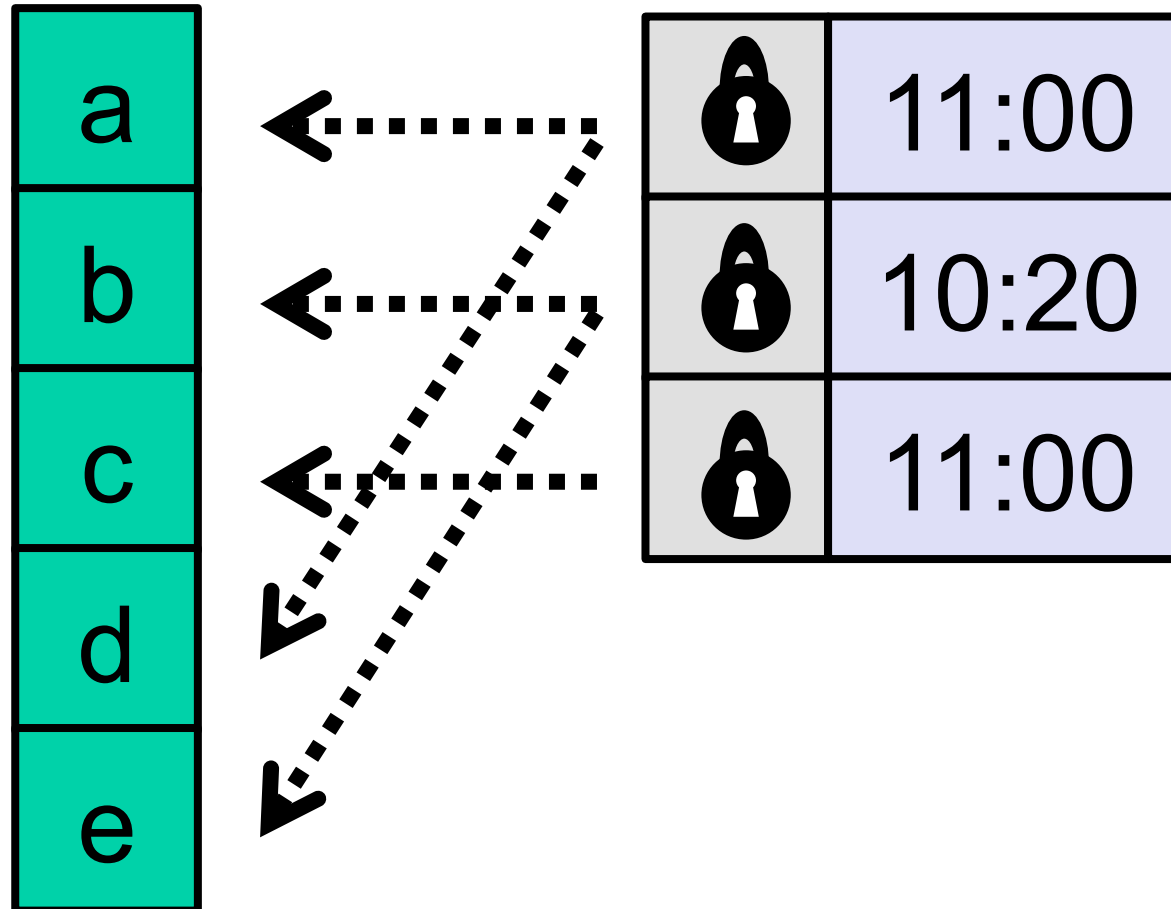
Changes installed at commit

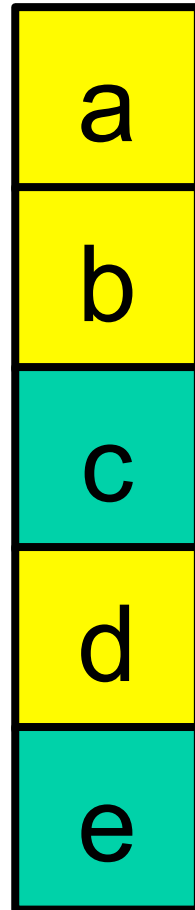
Conflicts detected at comit



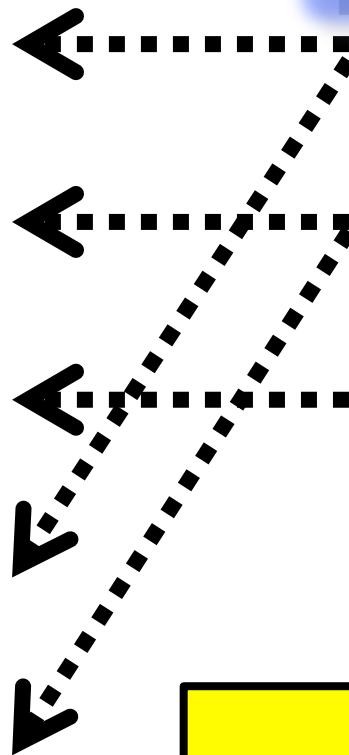








Check unlocked	
	11:00
	10:22
	11:00

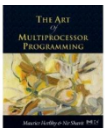


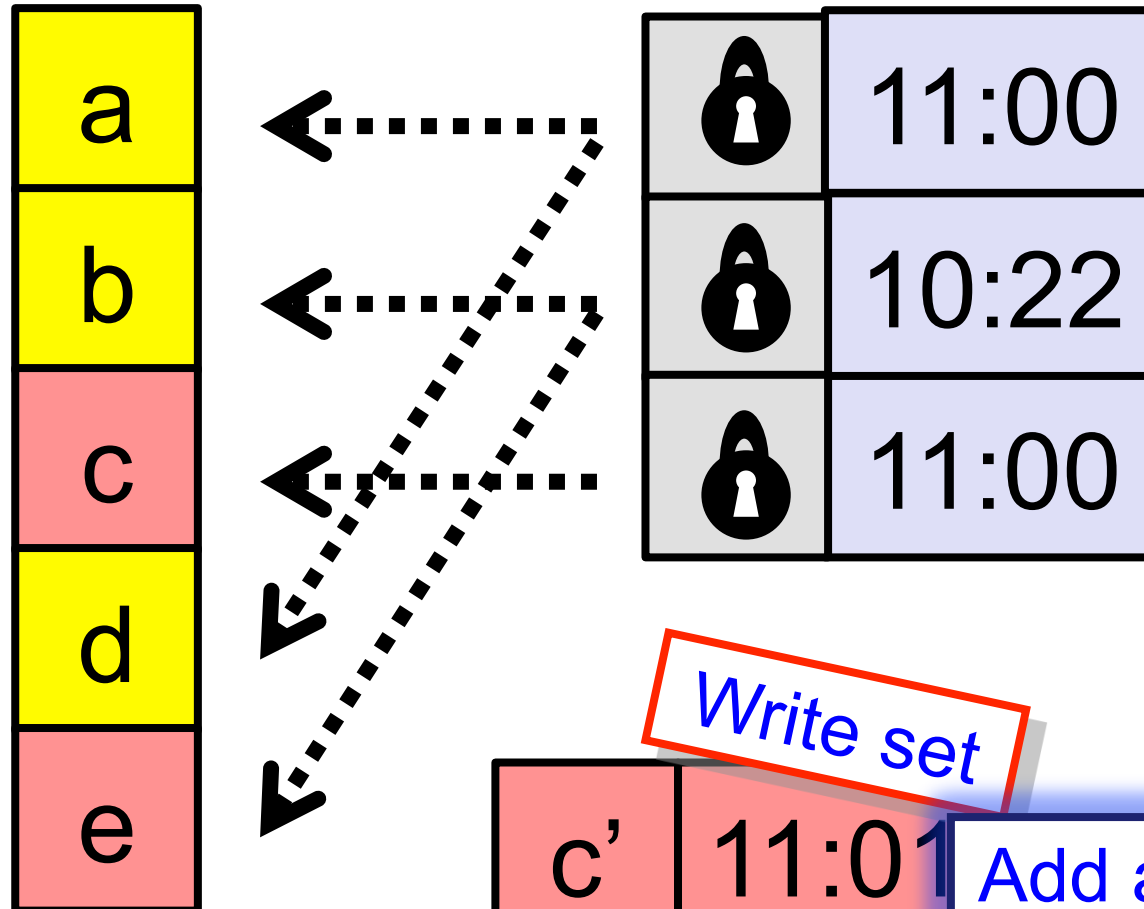
Read set

a	11:00
b	10:22
c	11:00

To read memory ...

Add address, values, and versions to read set



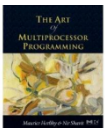


To write memory ...

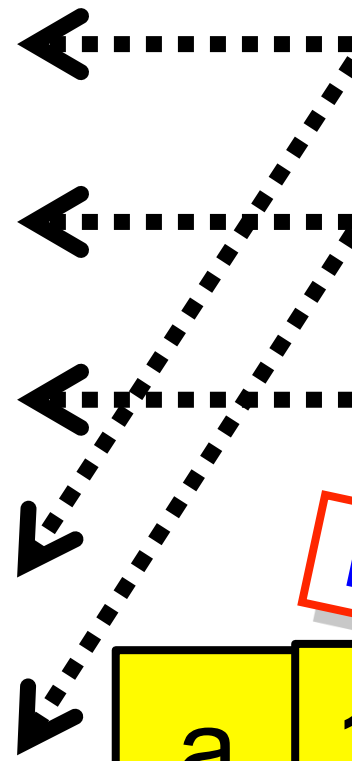
Write set

c'	11:01
e'	11:01

Add address,
new values
and versions
to **write set**



a
b
c
d
e



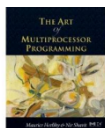
	11:00
	10:22
	11:00

Read set

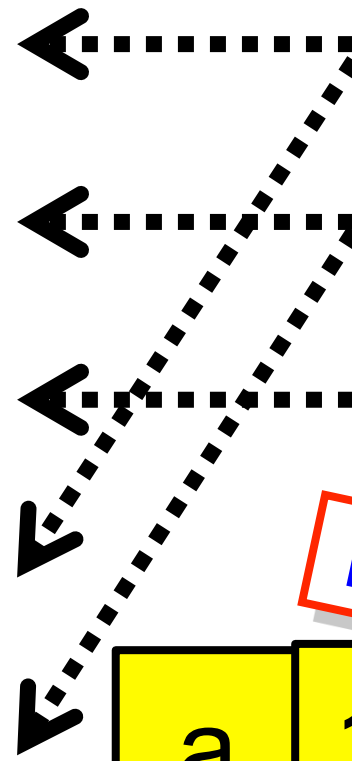
a	11:00
b	10:22
c	11:00

Write set

c'	11:01
e'	11:01



a
b
c
d
e



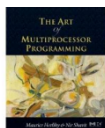
	11:00
	10:22
	11:00

Read set

a	11:00
b	10:22
c	11:00

Write set

c'	11:01
e'	11:01



To commit ...

a
b
c
d
e

	11:00 ✓
	10:22 ✓
	11:00 ✓

Read set

a	11:00
b	10:22
c	11:00

Write set

c'	11:01
e'	11:01

Acquire write locks

Compare version #s

Install new values

To commit ...

a
b
c'
d
e'

	11:00
	11:01
	11:01

Read set

Write set

Acquire write locks

Compare version #s

Install new values

Increment version #s

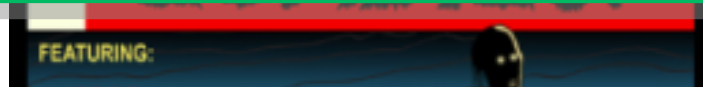
Release locks

a	11:00		
b	11:00		
c	11:00		
d	11:00		
e	11:00		

Zombie Transactions



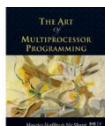
A **zombie human** is dead but act like it is alive ...

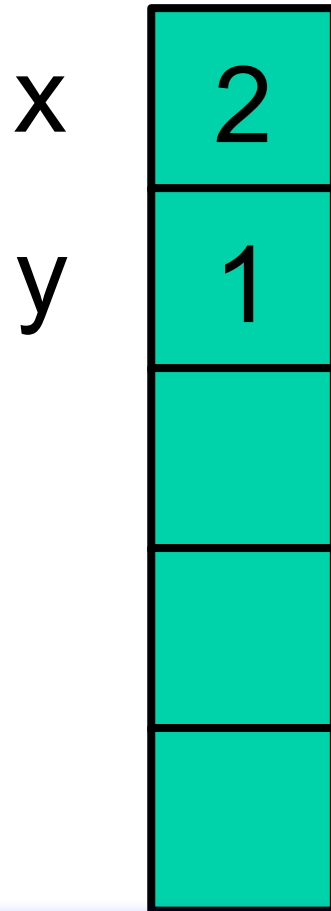


A **zombie transaction** is one that will certainly abort, but continues to run ...

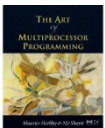


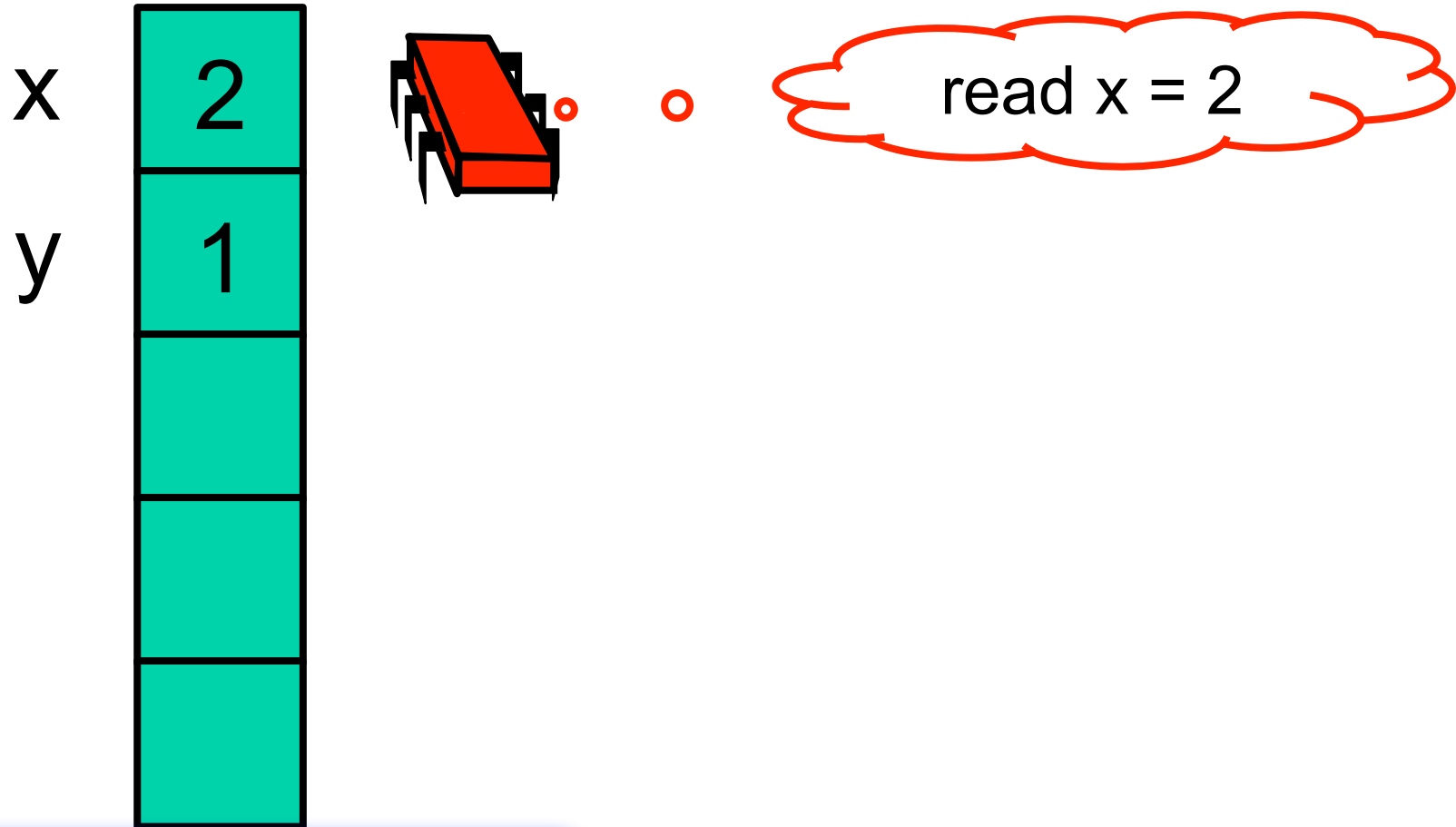
Why do we care?



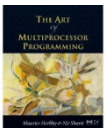


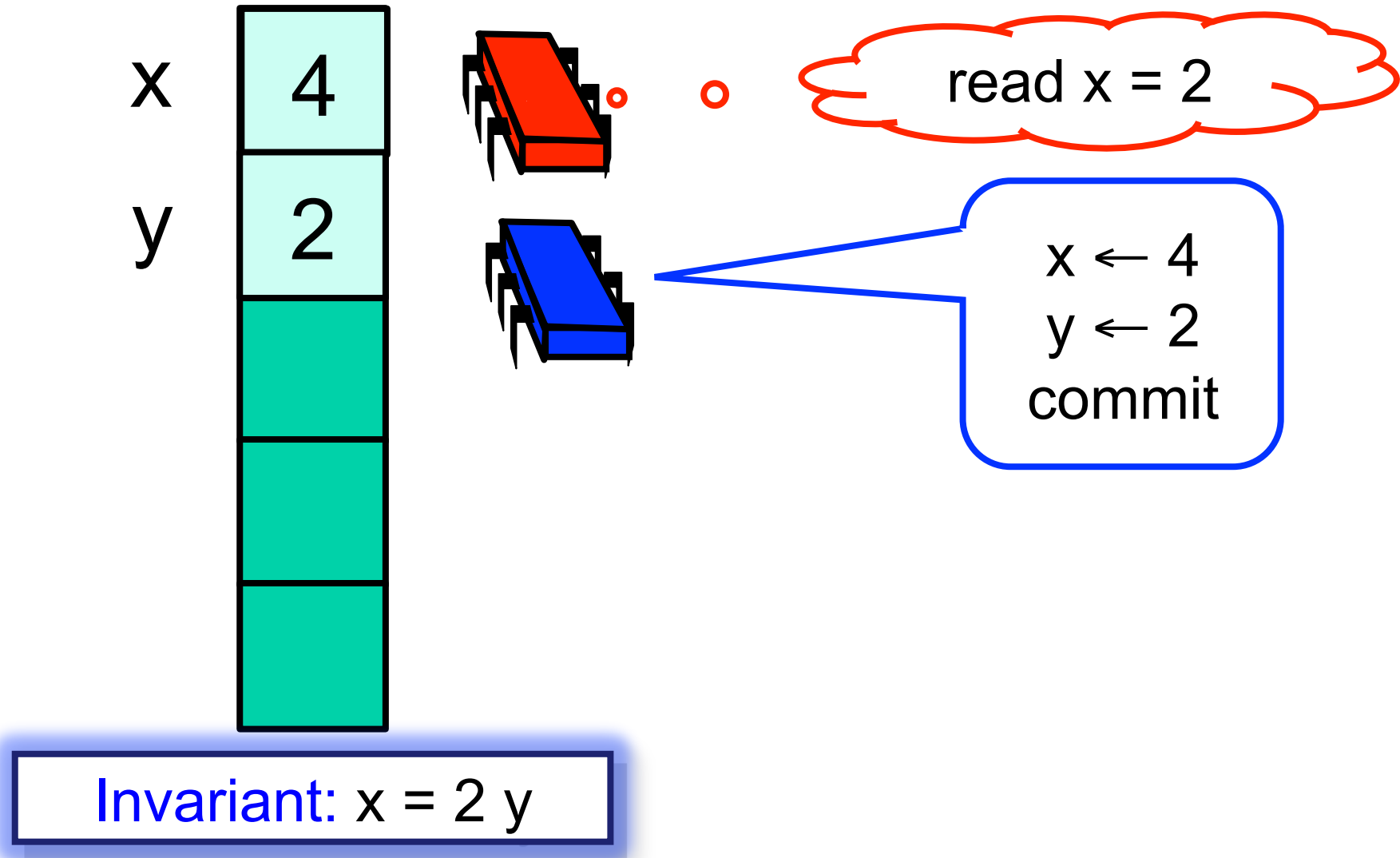
Invariant: $x = 2y$

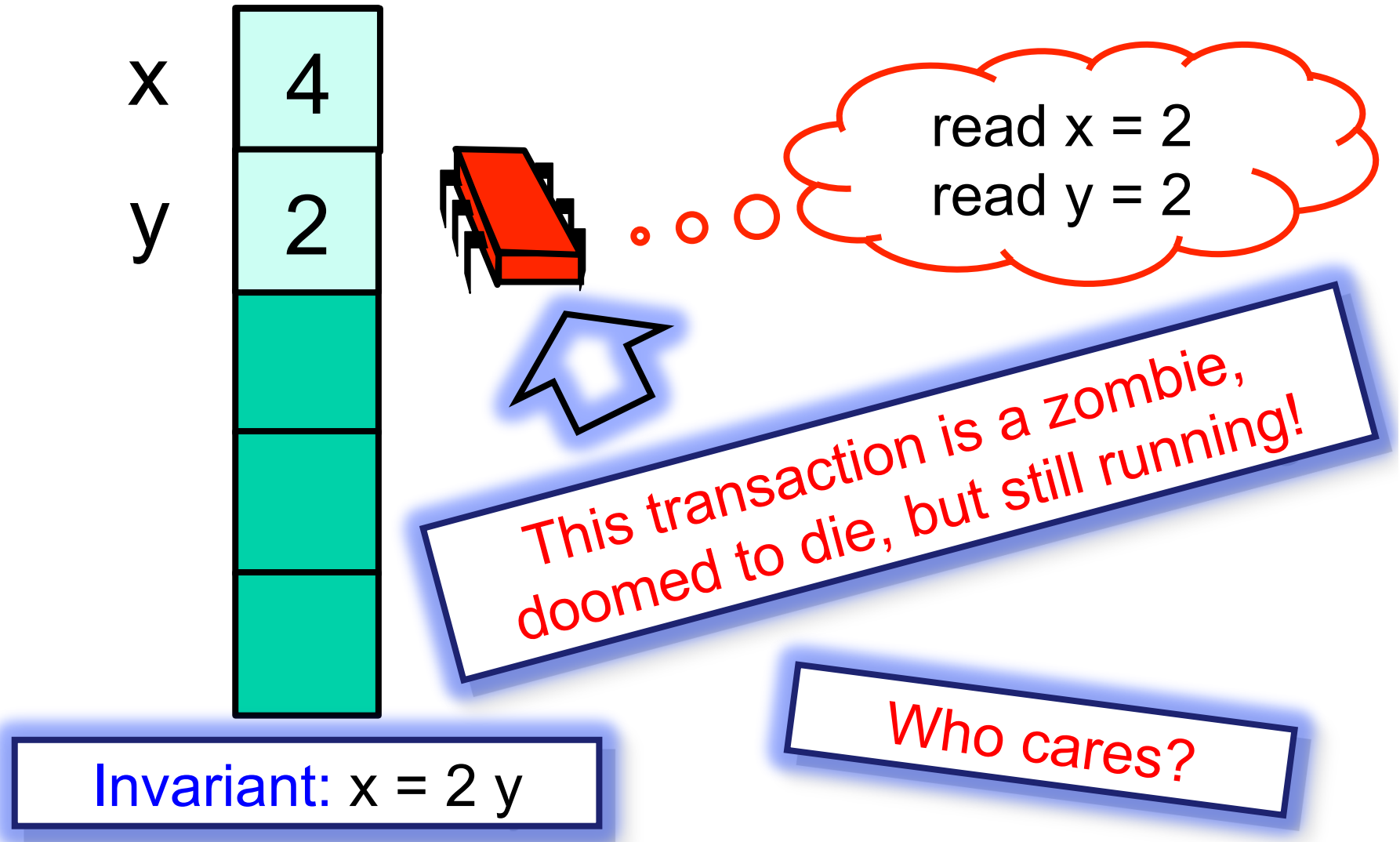


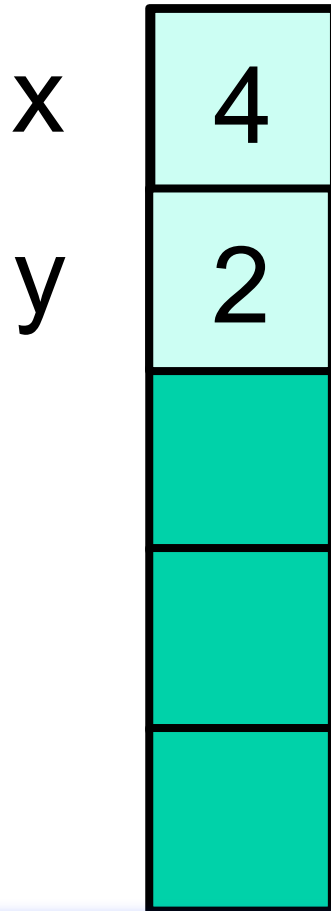


Invariant: $x = 2y$







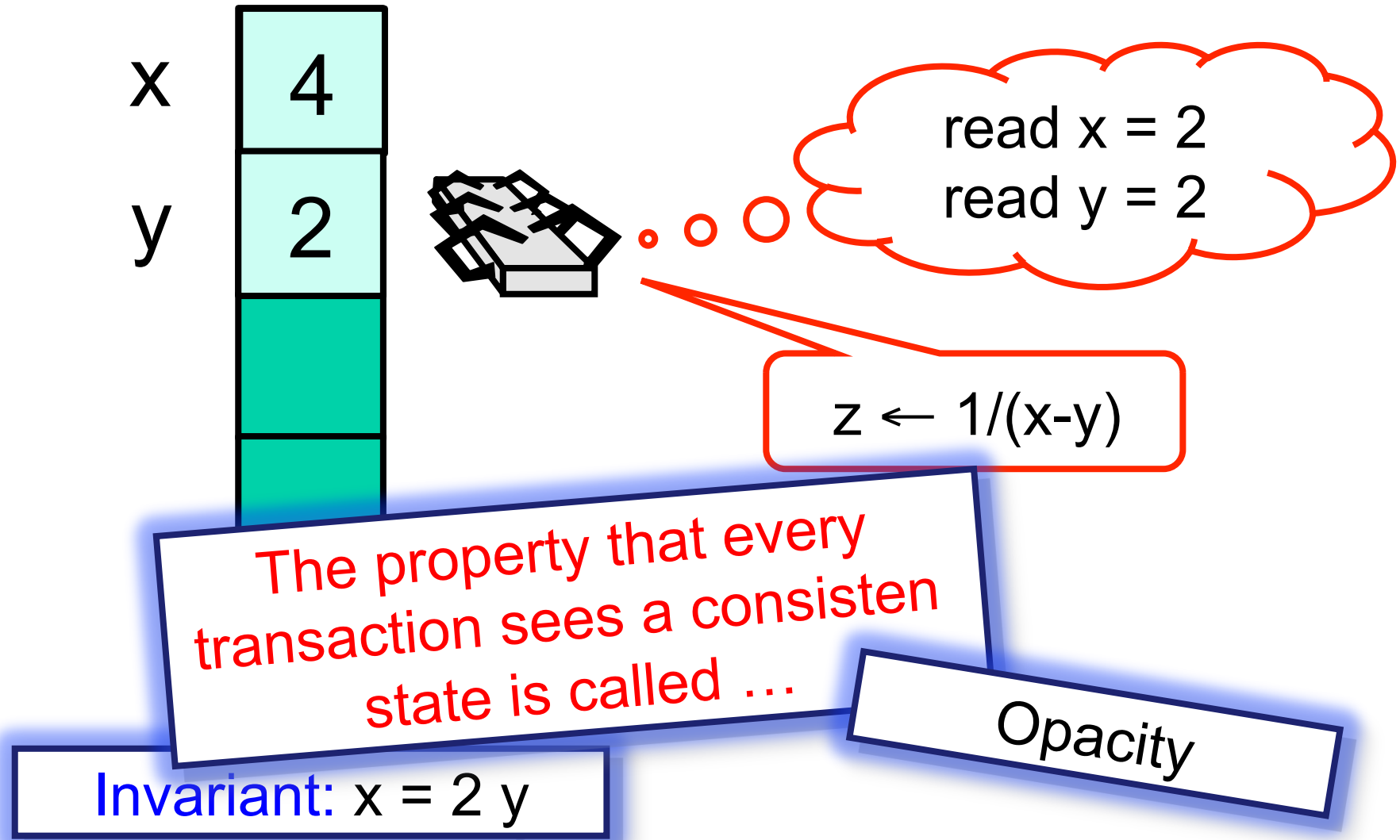


read $x = 2$
read $y = 2$

$$z \leftarrow 1/(x-y)$$

Oh, no! It divides by zero and
crashes the system!

Invariant: $x = 2y$



Version Clock

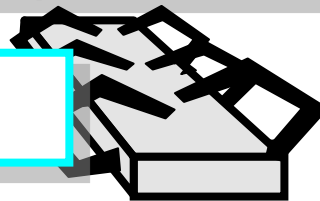
11:00

Introduce **version clock**

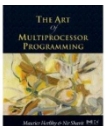
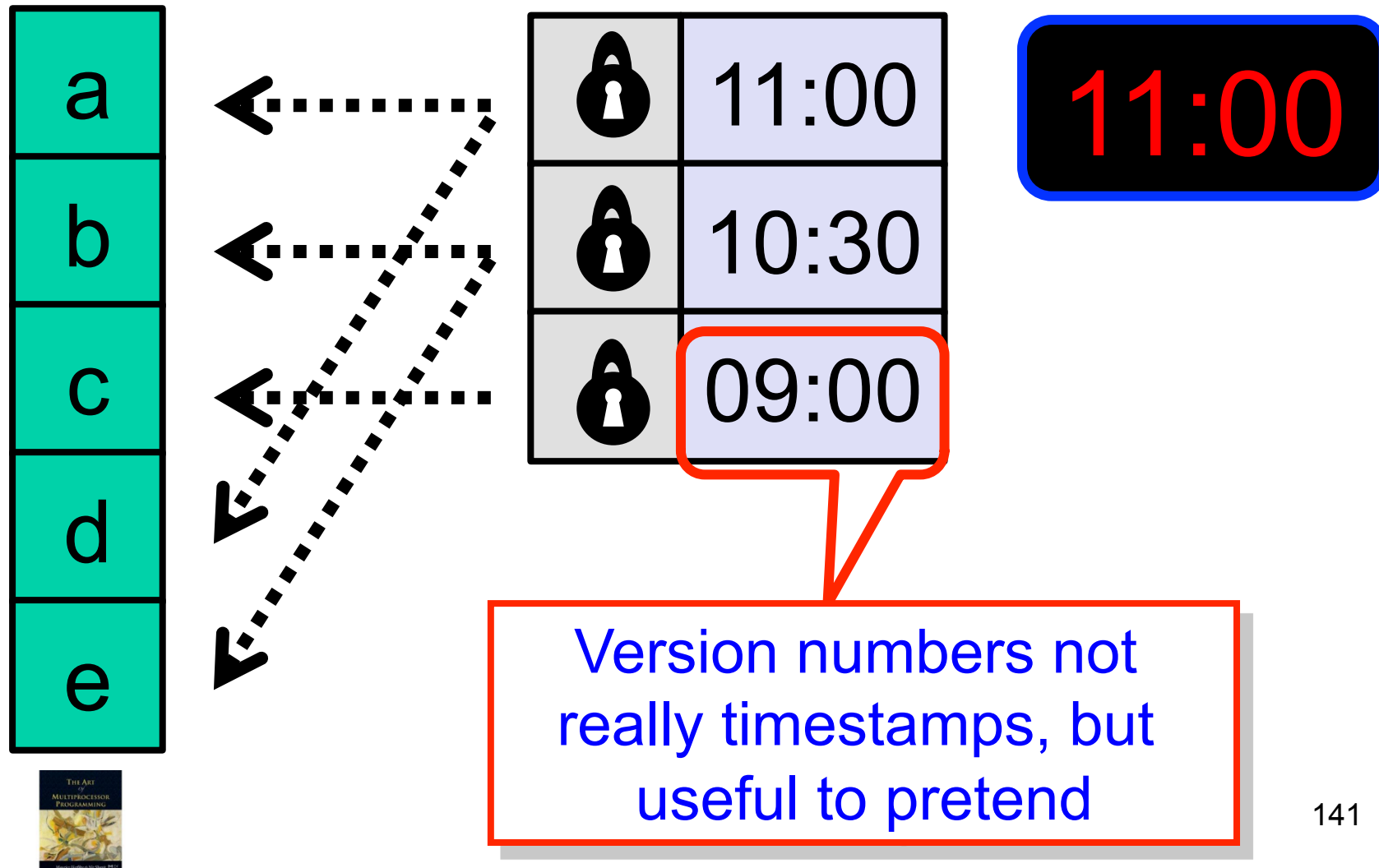
Incremented by (some) writers

Read by everyone

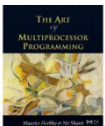
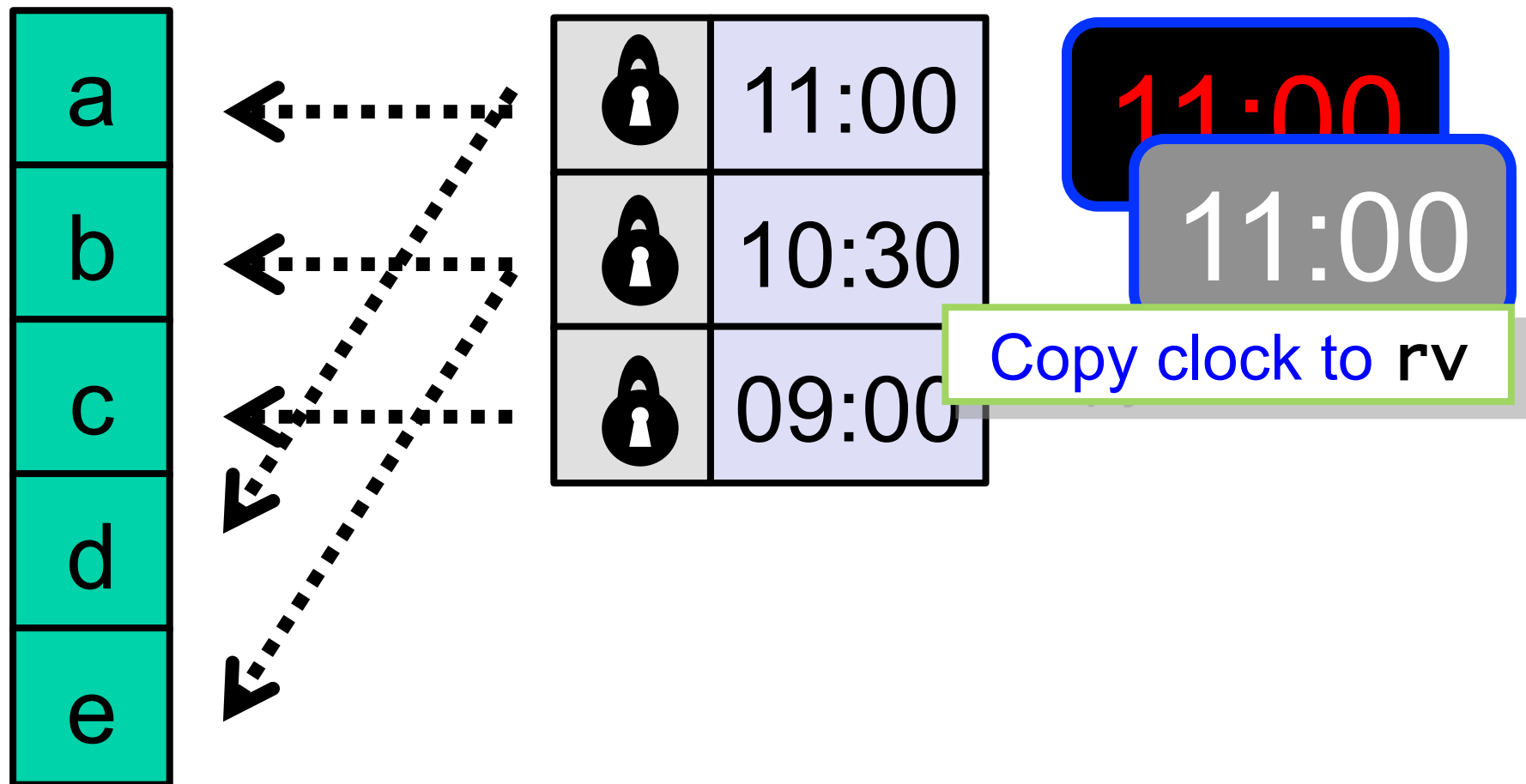
Guarantees **opacity**

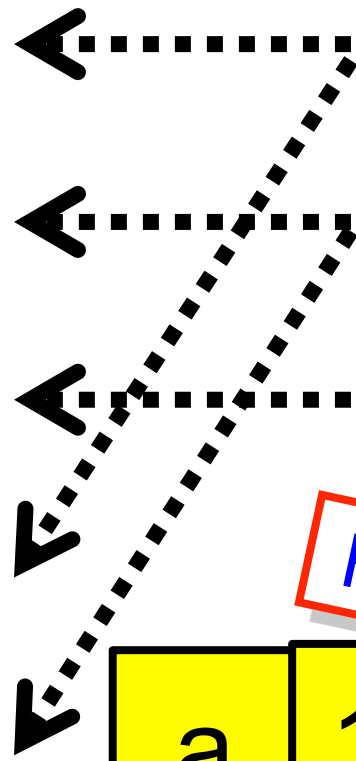
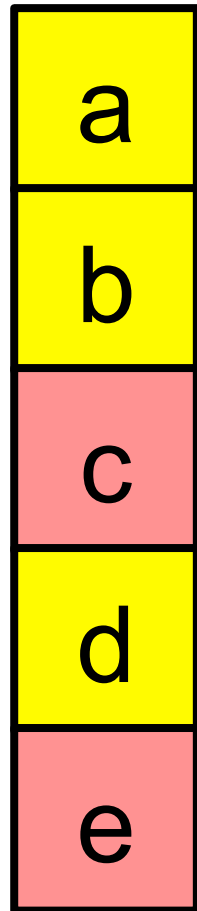


Transactin



Transactions





	11:00
	10:22
	11:00



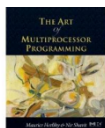
Read set

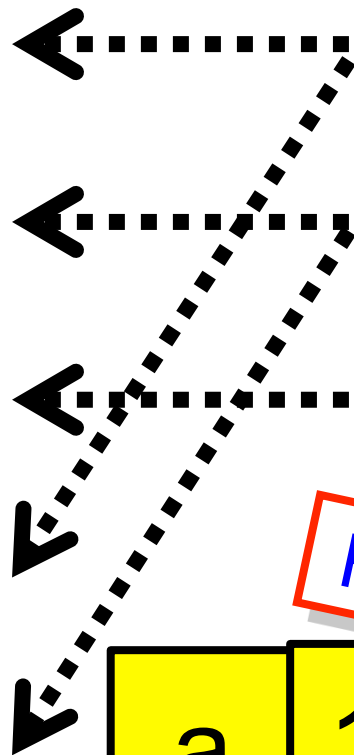
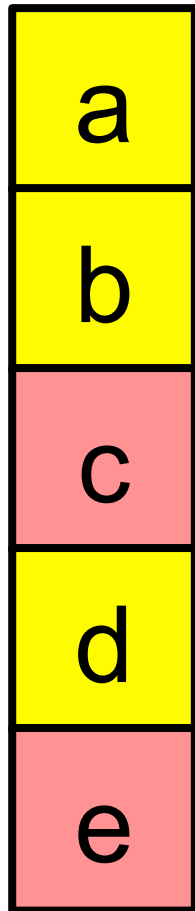
a	11:00
b	10:22

Write set

c	11:01
e	11:01

Run speculative transaction
as before ...





	11:00
	10:22
	11:00



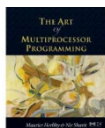
Read set

a	11:00
b	10:22
c	11:00

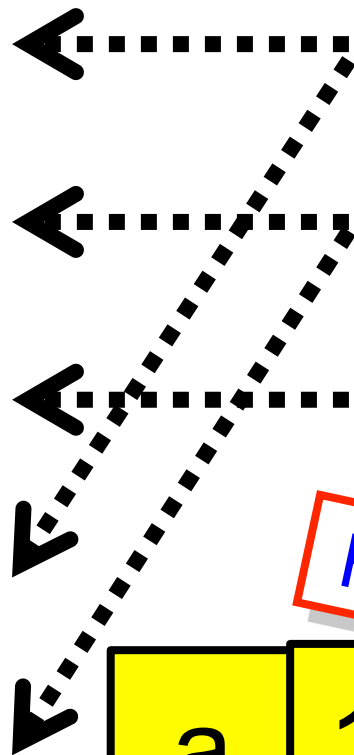
Write set

c'	11:01
e'	11:01

Lock Write Set ...



a
b
c
d
e



	11:00
	10:22
	11:00

11:01

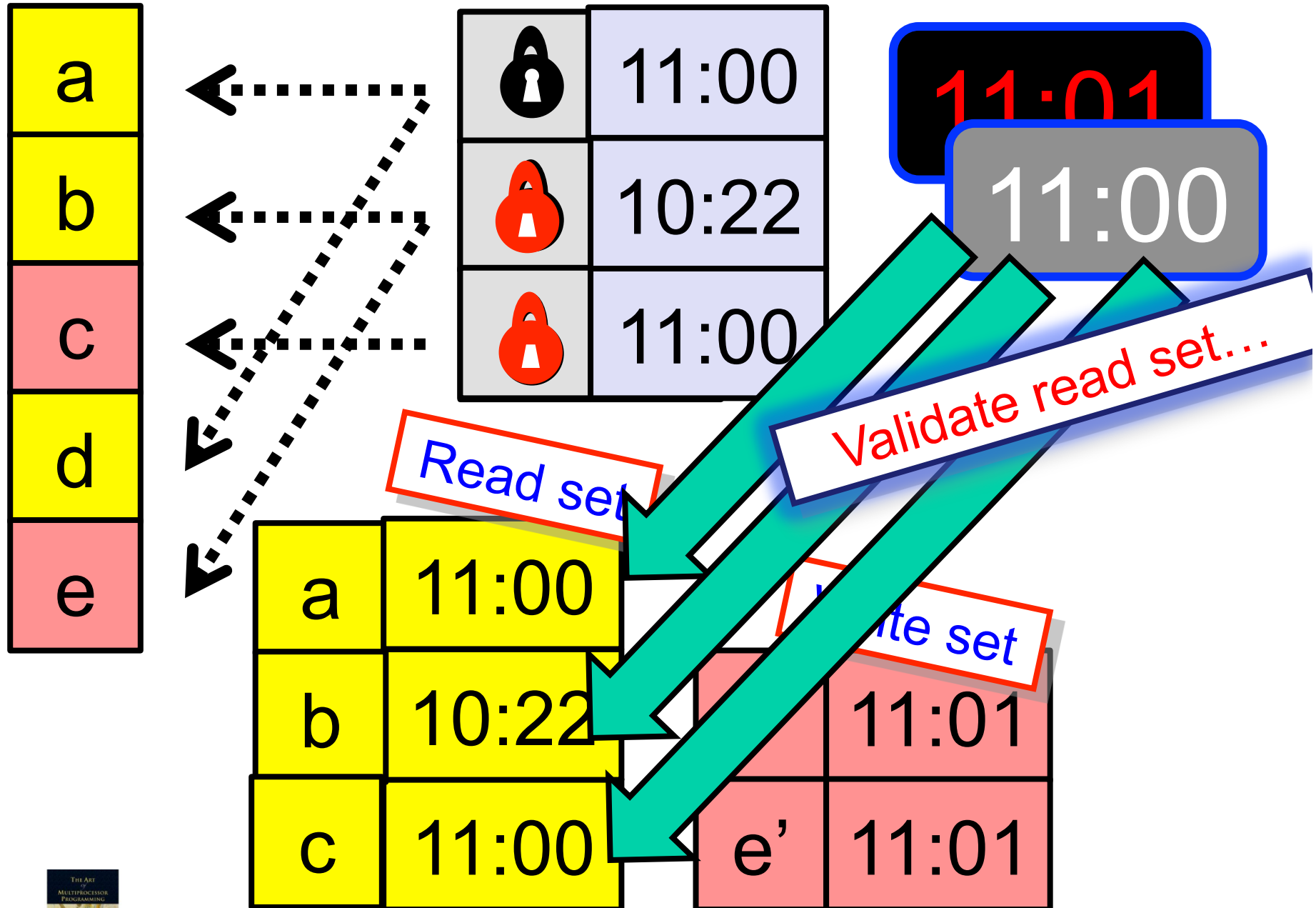
Increment global clock

Read set

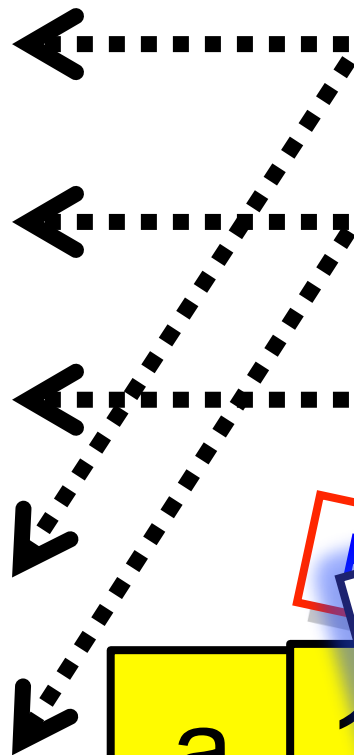
a	11:00
b	10:22
c	11:00

Write set

c'	11:01
e'	11:01



a
b
c
d
e



	11:00
	10:22
	11:00

11:01

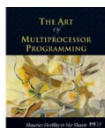
11:00

Commit & release locks

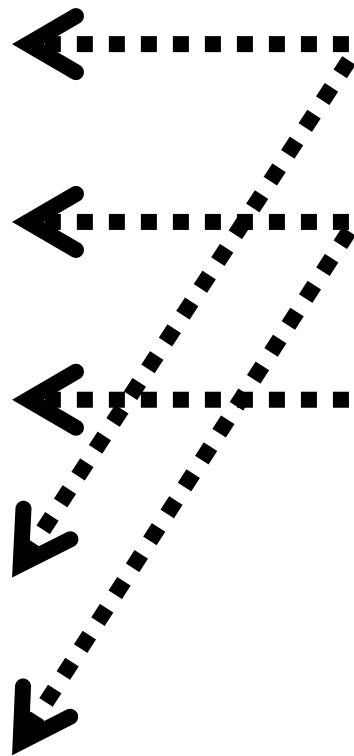
a	11:00
b	10:22
c	11:00

c'	11:01
e'	11:01

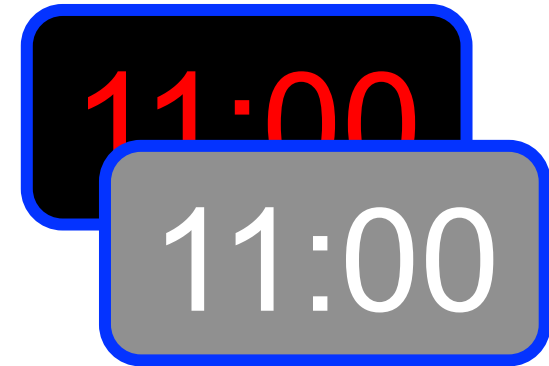
Write set



a
b
c
d
e



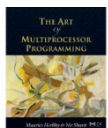
	11:00
	10:22
	11:00



Read set

Read-only transactions?

a	11:00
b	10:22
c	11:00





🔒	11:00
🔒	1
🔒	11:00

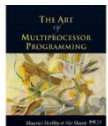
11:00

11:00

Check that variables read are unlocked

Check that version numbers less than or equal to cached clock

a	11:00
b	10:22
c	11:00



Road Map

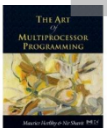
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

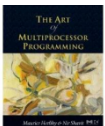
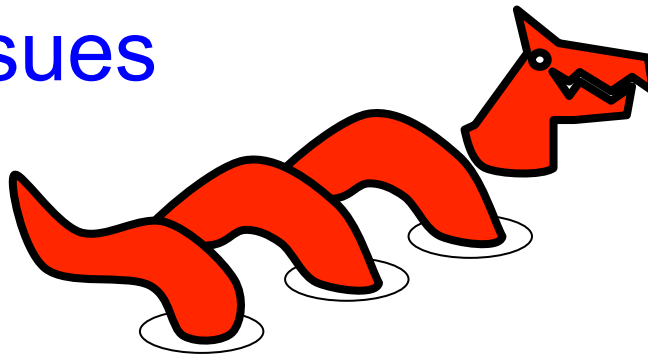
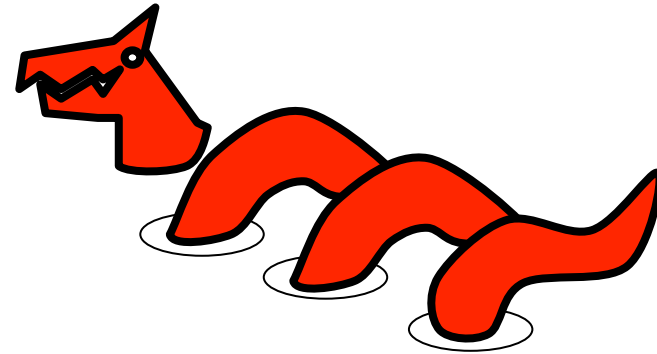
Software Transactional Memory

Research Questions



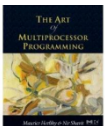
TM Design Issues

- Implementation choices
- Language design issues
- Semantic issues



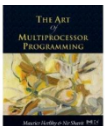
Granularity

- 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



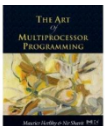
Direct/Deferred Update

- *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



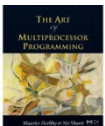
Conflict Detection

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



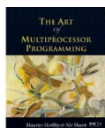
Conflict Detection

- 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

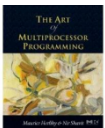


Contention Manager Strategies

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

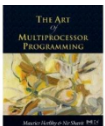


Judgment of Solomon



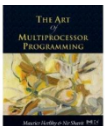
I/O & System Calls?

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

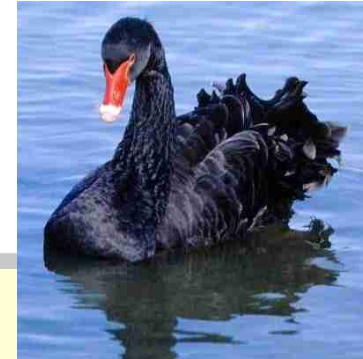


I/O & System Calls

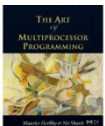
- One solution: make transaction irrevocable
 - If transaction tries I/O, switch to irrevocable mode.
- There can be only one ...
 - Requires serial execution
- No explicit aborts
 - In irrevocable transactions



Exceptions



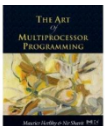
```
int i = 0;  
try {  
    atomic {  
        i++;  
        node = new Node();  
    }  
} catch (Exception e) {  
    print(i);  
}
```



Exceptions

Throws OutOfMemoryException!

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

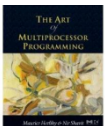


Exceptions

Throws OutOfMemoryException!

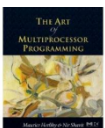
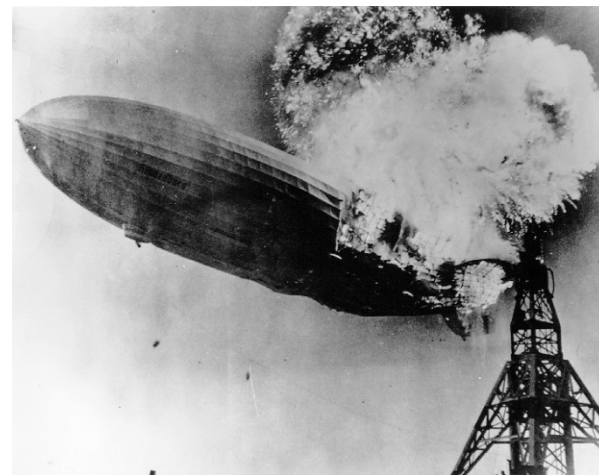
```
int i = 0;  
try {  
    atomic {  
        i++;  
        node = new Node();  
    }  
} catch (Exception e) {  
    print(i);  
}
```

What is
printed?



Unhandled Exceptions

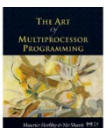
- Aborts transaction
 - Preserves invariants
 - Safer
- Commits transaction
 - Like locking semantics
 - What if exception object refers to values modified in transaction?



Nested Transactions

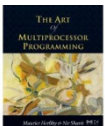
```
atomic void foo() {  
    bar();  
}
```

```
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



Locks

Hardware Transactions and Locks, Together at Last*

Elias Wald
Computer Science Department
Brown University
elias_wald@brown.edu

Maurice Herlihy
Computer Science Department
Brown University
mph@cs.brown.edu

2014

**Using Hardware Transactional Memory to Correct
and Simplify a Readers-Writer Lock Algorithm**

Dave Dice
Victor Luchangen

Yujie Xia
Loughborough University
y.j.xia@loughborough.ac.uk

locks and hardware transactions. Many
 - apparently synchronized nodes, each pro-
 - trying to access the same node at the
 - er by lock coupling: a thread holds

Locks and transactions complement on another

Abstract

[illegible]

signed differently in the three phases to avoid structural similarity of the variables, we decided to retain the difference vector of the WTM for the SGA algorithm in order to maintain even if the use of WTM has to impose and simplify an algorithm even if the use of Transactional Locks are not anticipated in the original design.

The fact is that the **Transaction (TLE)** whereby a bank and a customer execute a transaction, appears a lock, and therefore cannot be interrupted. To preserve correctness, the lock cannot be transaction nonpersistently. To preserve correctness, the lock cannot be transaction nonpersistently. To preserve correctness, the lock cannot be transaction nonpersistently.

...and another, ...

inherent list based on lock
algorithm, and slightly outper-

Memory Management

StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation

Dan Alistarh*
MSR, Cambridge
alistarh@microsoft.com

Patrick Eugster†
Purdue University
p@cs.purdue.edu

Maurice Herlihy†
Brown University
mhp@cs.brown.edu

Alexander Marteev
MIT
amarteev@csail.mit.edu

Nir Shavit‡
MIT and TAU
shavit@csail.mit.edu

On the Impact of Dynamic Memory Management on Software Transactional Memory Performance

Alexander Koblenz
UNICAMP - Institute of Computing
alex@ic.unicamp.br

Edson Rosta
UNICAMP - Institute of Computing
edson@ic.unicamp.br

Guilherme Araújo
UNICAMP - Institute of Computing
guilherme@ic.unicamp.br

Chihuahua: A Concurrent, Moving, Garbage Collector using Transactional Memory

Todd A. Anderson
Intel Labs
todd.a.anderson@intel.com

Melissa O'Neill
Harvey Mudd College
omel@hmc.edu

John Serrano
University of California San Diego
jserrano@cs.ucsd.edu

Exploring Garbage Collection with Haswell Hardware Transactional Memory

Carl G. Gjorgiev
University of Kansas
cgjorgiev@ku.edu

Richard B. Jones
University of Kent
R.B.Jones@kent.ac.uk

Tamara Uspenskiy
Kavli Institute for Theoretical Physics
uspenskiy@theory.kitp.ucsb.edu

TM can improve memory management, both automatic and explicit.

Power and Energy

Energy-Aware Microprocessor Synchronization
Transaction Memory vs. Locks

Maurice Herlihy[†]
Providence, RI 02912
Providence, RI 02912

Tali Moresht
*Brown Univ
Providence, RI 02912

Playing with Fire: Transactional
Error-Resilient and Energy-Efficient

Dimitra Papagiannopoulou

New research in energy-efficient synchronization

Marong
Zurich
amis.ee.ethz

F. Klein, A. ...

Efficiency of Software Transactional
Memory

GPUs, etc.

Hardware Transactional Memory for GPU Architectures

Yunlong Xu*, Rui Wang†, Nilanjan Goswami‡, Tao Li‡, Lan (Xiao) Lan‡, Xi'an Jiaotong University
*School of Electronic Engineering, Beihang University
†School of Computer Science, University of Florida, Gainesville, FL
‡ECE Department, University of Florida, Gainesville, FL
xjtu.ylxu@stu.xjtu.edu.cn, ruiwang@ufl.edu, nilanjan.goswami@ufl.edu, tao.li@ufl.edu, lan.gao@ufl.edu, lan.xiao@ufl.edu

ABSTRACT

Graphics processors

Improvements in Hardware Transactional Memory for GPU Architectures

Alejandro Villegas, Angeles Navarro, Rafael Asenjo, and Oscar Plata
Dept. Computer Architecture
University of Malaga, Andalucia Tech, 29071 Malaga
{avillegas, angeles, asenjo, oscar}@cc.uma.es

Hardware Transactional Memory for GPU Architectures

Wilson W. L. Fung†, Inderpreet Singh†, Andrew Brownsword†, Tor M. Aamodt†
Department of Computer and Electrical Engineering
University of British Columbia
wulfung@ece.ubc.ca, isingh@ece.ubc.ca, aabrowsw@ece.ubc.ca, aamodt@ece.ubc.ca

ABSTRACT

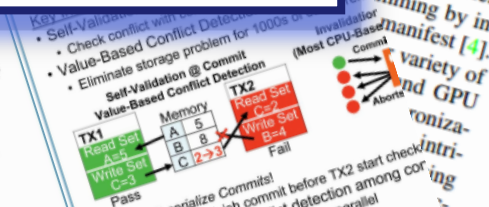
Modern GPUs have shown promise in accelerating computation intensive applications with limited dynamic data sharing among concurrently executing threads. However, sharing requires mutual exclusion, which is not supported by GPUs. GPUs provide atomic primitives to construct fine-grained synchronization. However, this requires significant overhead in terms of transactional correctness and execution parallelism. To make the most of GPU acceleration, transactional memory systems are needed. The major challenge is preventing livelocks and deadlocks of GPUs. To this end, transactional memory systems have emerged as an alternative to CPUs. GPUs offer two main challenges: 1) Control Flow Divergence: Transaction aborts may cause a warp to diverge. 2) Scalable Conflict Detection: 1000s of concurrent transactions require a scalable conflict detection protocol on GPU.

GPUs and accelerators need synchronization

Challenges

Control Flow Divergence:

- Transaction aborts may cause a warp to diverge
- Scalable Conflict Detection: 1000s of concurrent transactions require a scalable conflict detection protocol on GPU
- 1000 x 1000 parallel address-set comparison – too expensive?
- Cache coherence protocol on GPU
- 1000s of threads is not cheap



OS

ing Address-Space Operations on Linux with TSX

by
Christopher Ryan Johnson

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2014, in partial fulfillment of the
requirements for the
Master of Science

Abstract
Modern programming is becoming increasingly complex and error-prone. Address-Space Operations (ASO) is a new programming model that simplifies the execution of ASO operations on multi-processor systems. This thesis describes the design and implementation of ASO on Linux, and how it can be used to simplify the execution of ASO operations on multi-processor systems.

This thesis describes a new system, which currently serializes ASO operations, and how it can be used to simplify the execution of ASO operations on multi-processor systems. This thesis also describes how the ASO operations can be used to simplify the execution of ASO operations on multi-processor systems.

Thesis Supervisor:
Prof. Professor C

Thesis Support:
Prof. Assistant

Categories and Subject Descriptors
C.2.2 [Operating Systems]: Distributed Systems; D.1.2 [Programming Techniques]: Languages; D.1.3 [Programming Techniques]: Languages

General Terms
Design, Performance

Keywords
Transactional Memory, Hardware Transactional Memory

1. INTRODUCTION

Modern multi-processor systems are becoming increasingly complex and error-prone. Address-Space Operations (ASO) is a new programming model that simplifies the execution of ASO operations on multi-processor systems. This thesis describes the design and implementation of ASO on Linux, and how it can be used to simplify the execution of ASO operations on multi-processor systems.

Permission to make digital or physical copies of this work for personal or internal use, or the internal or external use of specific clients, is granted by MIT for users registered with the Copyright Clearance Center (CCC) Transactional Reporting Service, provided that the fee of \$12.00 per copy is paid directly to CCC. For those organizations that have been granted a photocopy licence by CCC, a separate system of payment has been arranged. The fee code for users of the Transactional Reporting Service is 0895-1939/2014 \$12.00. Copyright 2014 ACM 978-1-4558-7655-5/14/0000...\$12.00.

TxLinux: Using and Managing Hardware Transactional Memory in an Operating System

Christopher J. Rossbach, Owen S. Hofmann, Donald E. Long, Harry E. Rasmussen,
Aditya Ravi, and Emma M. Witcher
Department of Computer Science, University of Texas at Austin
{rossbach, long, rasmussen, ravi, witcher}@cs.utexas.edu

Appears in SOSP 2017

ABSTRACT

Transactional memory (TM) is a new programming model that simplifies the execution of ASO operations on multi-processor systems. This thesis describes the design and implementation of TM on Linux, and how it can be used to simplify the execution of ASO operations on multi-processor systems.

Modern multi-processor systems are becoming increasingly complex and error-prone. Address-Space Operations (ASO) is a new programming model that simplifies the execution of ASO operations on multi-processor systems. This thesis describes the design and implementation of ASO on Linux, and how it can be used to simplify the execution of ASO operations on multi-processor systems.

This thesis describes a new system, which currently serializes ASO operations, and how it can be used to simplify the execution of ASO operations on multi-processor systems. This thesis also describes how the ASO operations can be used to simplify the execution of ASO operations on multi-processor systems.

Hardware support for Local Memory Transactions on GPU Architectures

Aditya Ravi, Owen S. Hofmann, Donald E. Long, Harry E. Rasmussen,
Aditya Ravi, and Emma M. Witcher
Department of Computer Science, University of Texas at Austin
{ravi, long, rasmussen, ravi, witcher}@cs.utexas.edu

Appears in SOSP 2017

Modern multi-processor systems are becoming increasingly complex and error-prone. Address-Space Operations (ASO) is a new programming model that simplifies the execution of ASO operations on multi-processor systems. This thesis describes the design and implementation of ASO on Linux, and how it can be used to simplify the execution of ASO operations on multi-processor systems.

OS Support for Virtualizing Hardware Transactional Memory

Michael M. Swift, Peter W. O'Neil, Nathan G. Leung, Yoon, Abhishek, David A.
University of Wisconsin-Madison
{swift, o'neil, leung, yoon, abhishek, david}@cs.wisc.edu

Abstract

Transactional memory (TM) is a new programming model that simplifies the execution of ASO operations on multi-processor systems. This thesis describes the design and implementation of TM on Linux, and how it can be used to simplify the execution of ASO operations on multi-processor systems.

1. Introduction

Modern multi-processor systems are becoming increasingly complex and error-prone. Address-Space Operations (ASO) is a new programming model that simplifies the execution of ASO operations on multi-processor systems. This thesis describes the design and implementation of ASO on Linux, and how it can be used to simplify the execution of ASO operations on multi-processor systems.

TM can simplify operating system kernels, device drivers, security ...

Theory

The Theory of Transactional Memory

Guerraoui, Rachid; Kapalka, Michal

Published in: Bulletin of the EATCS, num. 97

Publication date: 2009

Transactional memory (TM) is a promising paradigm for parallel programming. This paper is an overview of our work on the theory of TM. We first present a correctness proof for existing TM implementations. Then we characterize the two main classes of TM: lock-based TMs and lock-free TMs. We then present power-law models for TM.

8th Workshop on the Theory of Transactional Memory

WTTM 2016

July 25, 2016, Chicago

International Conference on Computer Aided Verification
CAV 2009: [Computer Aided Verification](#) pp 1-15

Transactional Memory: Glimmer of a Theory

(Extended Paper)

Authors and affiliations

Architecture



Journal of Systems Architecture

Volume 73, February 2017, Pages 42–52



Hardware transactional memory architecture with adaptive
version management for multi-processor FPGA platforms

van Sirkunan^a, Chia Yee Ooi^b, N. Shaikh-Husin^a, Yuan W.

[View more](#)

<https://doi.org/10.1016/j.sysarc.2017.02.001>

**Transactional Memory Architecture and Implementation for
IBM System Z**

38
Paper
Citations

26
Patent
Citations

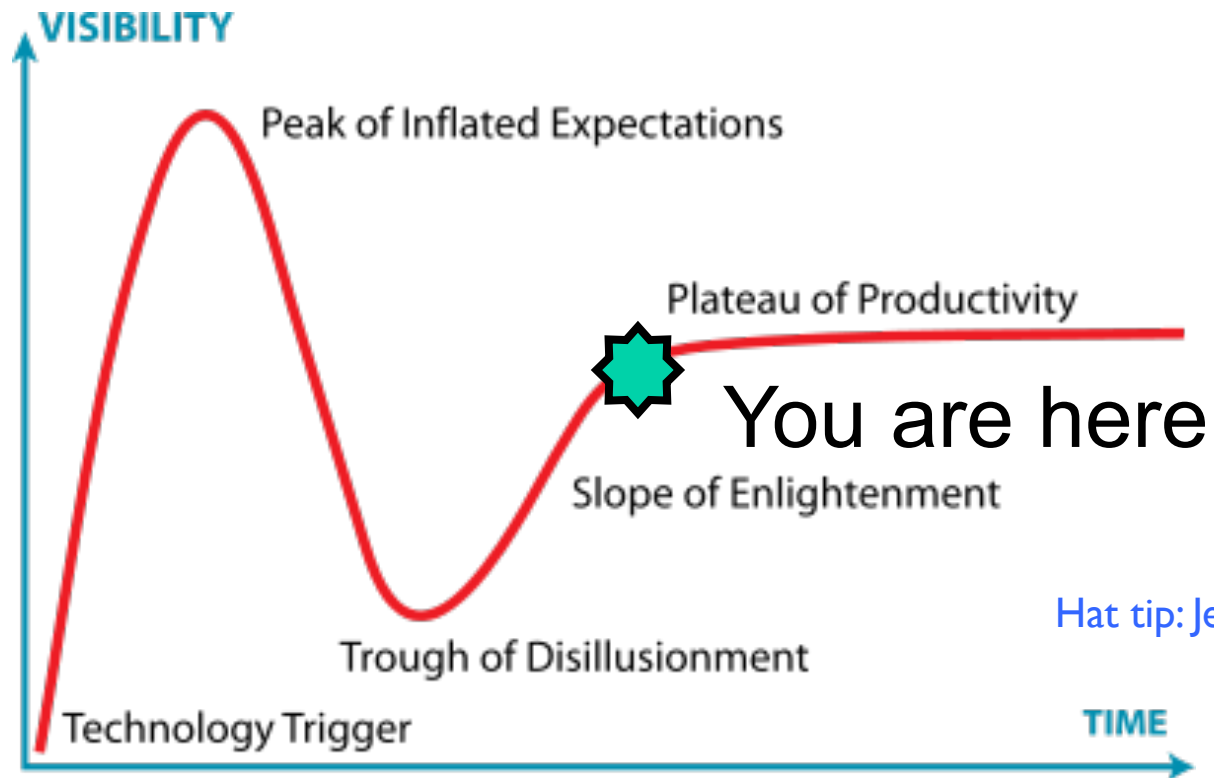
322
Full
Text Views

**Hardware Acceleration of Transactional
Memory on Commodity Systems**

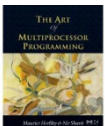
act

rocessor en
high perfor
er, lock-bas
ent system

Gartner Hype Cycle



Hat tip: Jeremy Kemp



Transactions are Here to Stay

Transactional Language Constructs for C++

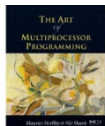
Authors: Hans Boehm, HP, hans.boehm@hp.com
Justin Gottschlich, Intel, justin.e.gottschlich@intel.com
Victor Luchangco, Oracle, victor.luchangco@oracle.com
Maged Michael, IBM, maged.michael@acm.org
Mark Moir, Oracle, mark.moir@oracle.com
Clark Nelson, Intel, clark.nelson@intel.com
Torvald Riegel, Red Hat, triegel@redhat.com
Tatiana Shpeisman, Intel, tatiana.shpeisman@intel.com
Michael Wong, IBM, michaelw@ca.ibm.com

Document number: N3341-12-0031
Date: 2012-01-11
Project: Programming Language C++, Evolution Working Group
Reply-to: Michael Wong, IBM, michaelw@ca.ibm.com
Revision: 1

Introduction



Intel® Architecture Instruction Set Extensions Programming Reference



Спасибо!

