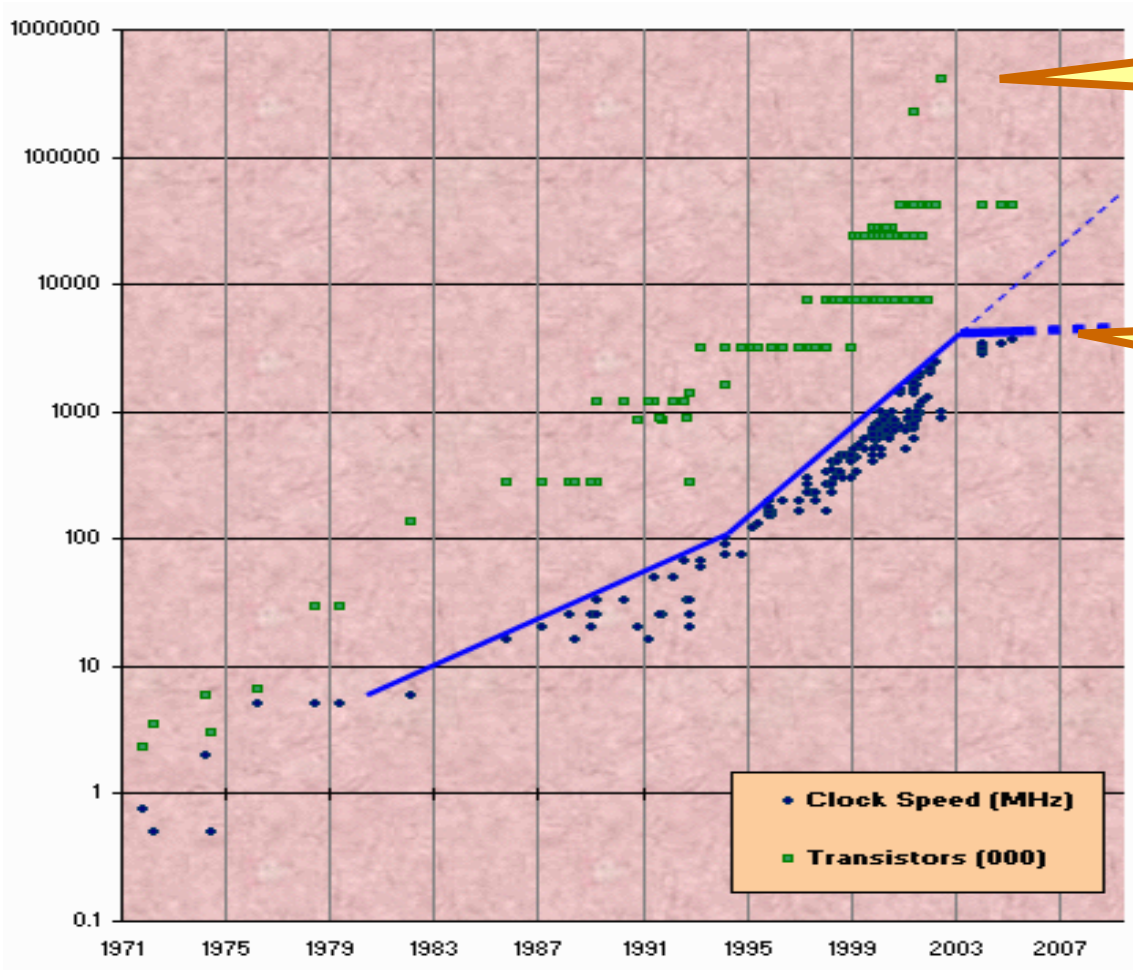
Taking Concurrency Seriously: New Directions in Multiprocessor Synchronization

Maurice Herlihy
Brown University

Moore's Law



Transistor count still rising

Clock speed flattening sharply

(hat tip: Herb Sutter)

Multicore Architetures

- "Learn how the multi-core processor architecture plays a central role in Intel's platform approach. …."
- "AMD is leading the industry to multi-core technology for the x86 based computing market …"
- "**Sun's** multicore strategy centers around multi-threaded software. ... "

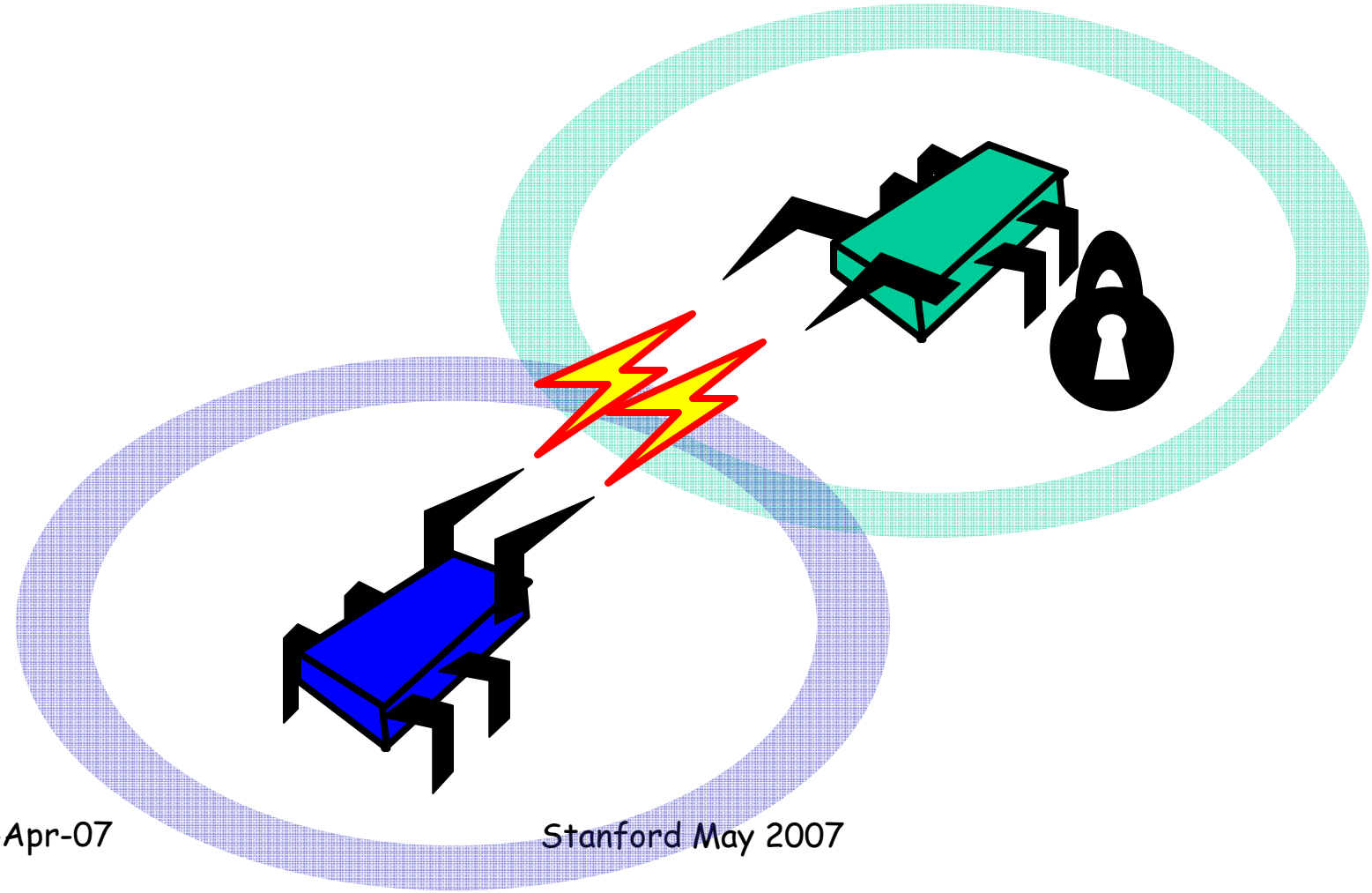
Why do we care?

- Time no longer cures software bloat
- When you double your path length
  - You can't just wait 6 months
  - Your software must somehow exploit twice as much concurrency

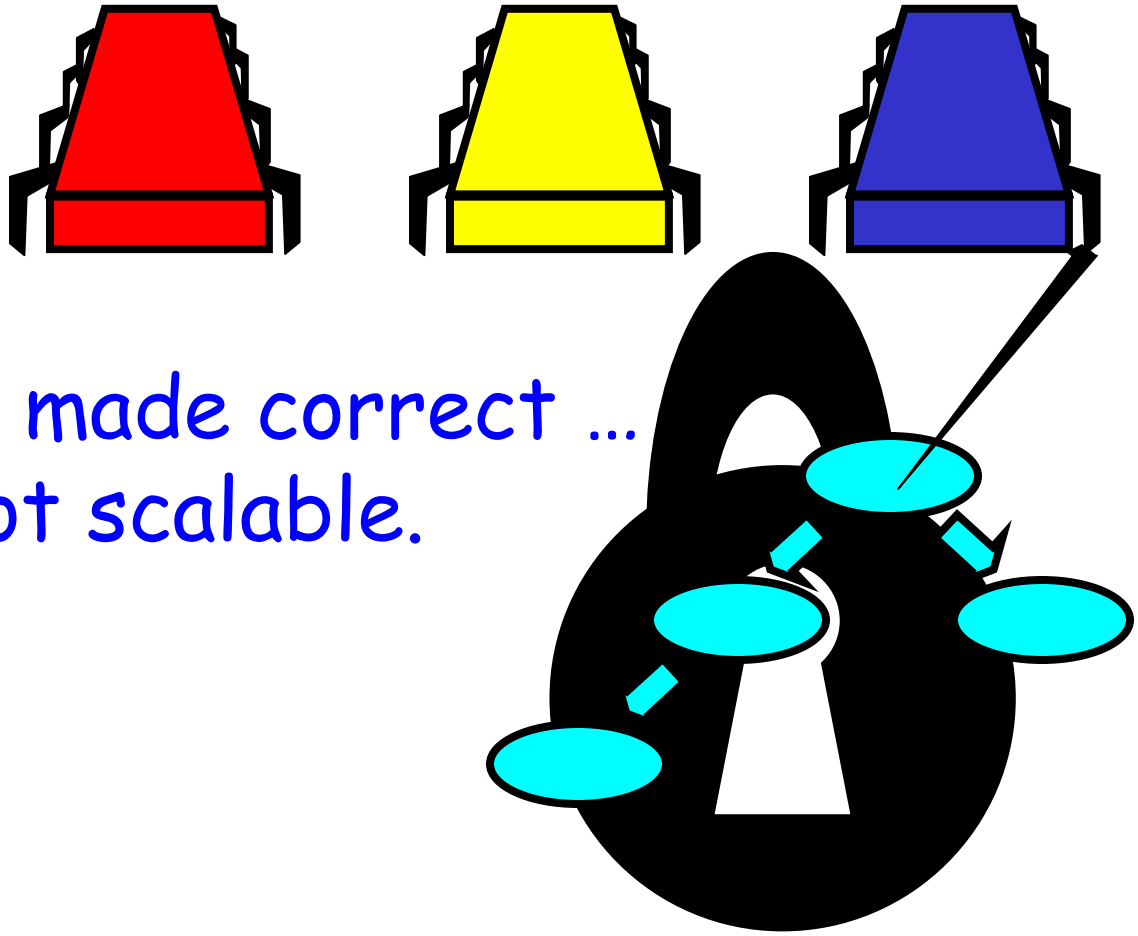
The Problem

- Cannot exploit cheap threads
- Today's Software
  - Non-scalable methodologies
- Today's Hardware
  - Poor support for scalable synchronization

Threads and Locking

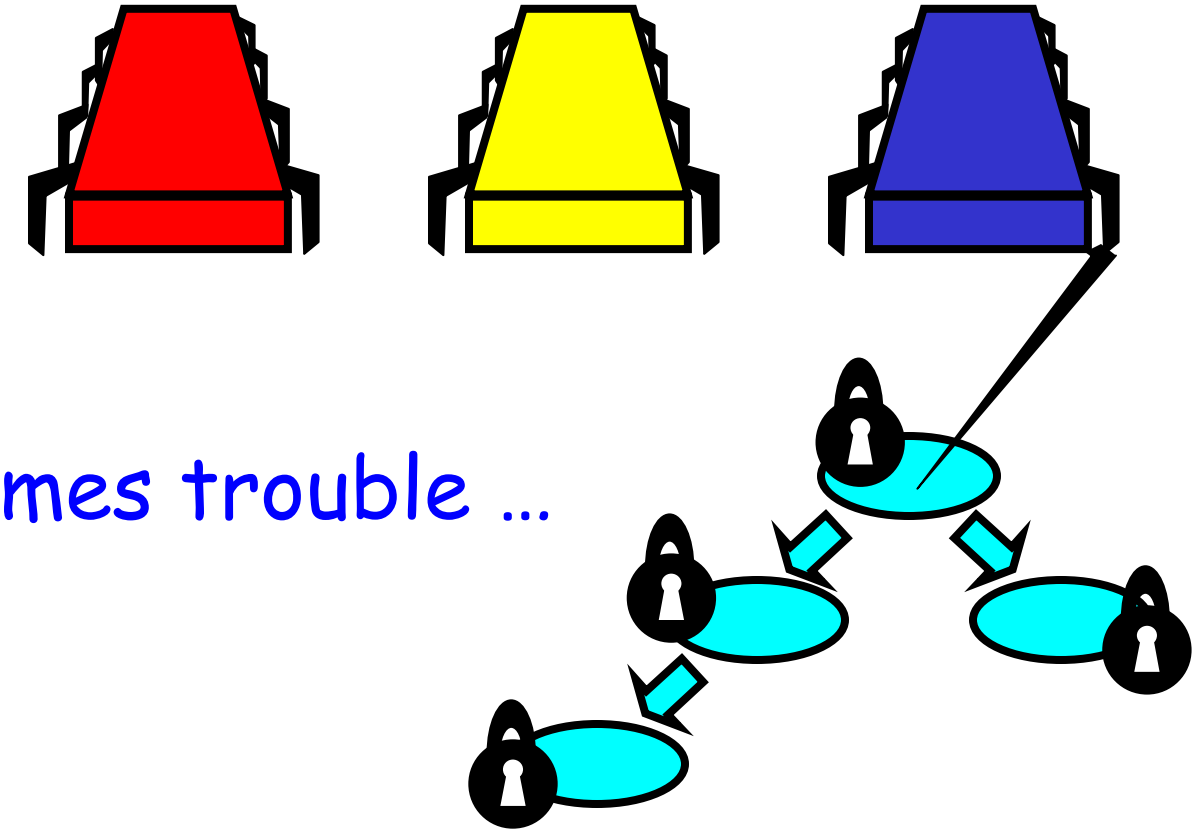


Coarse-Grained Locking



Easily made correct …
But not scalable.

Fine-Grained Locking



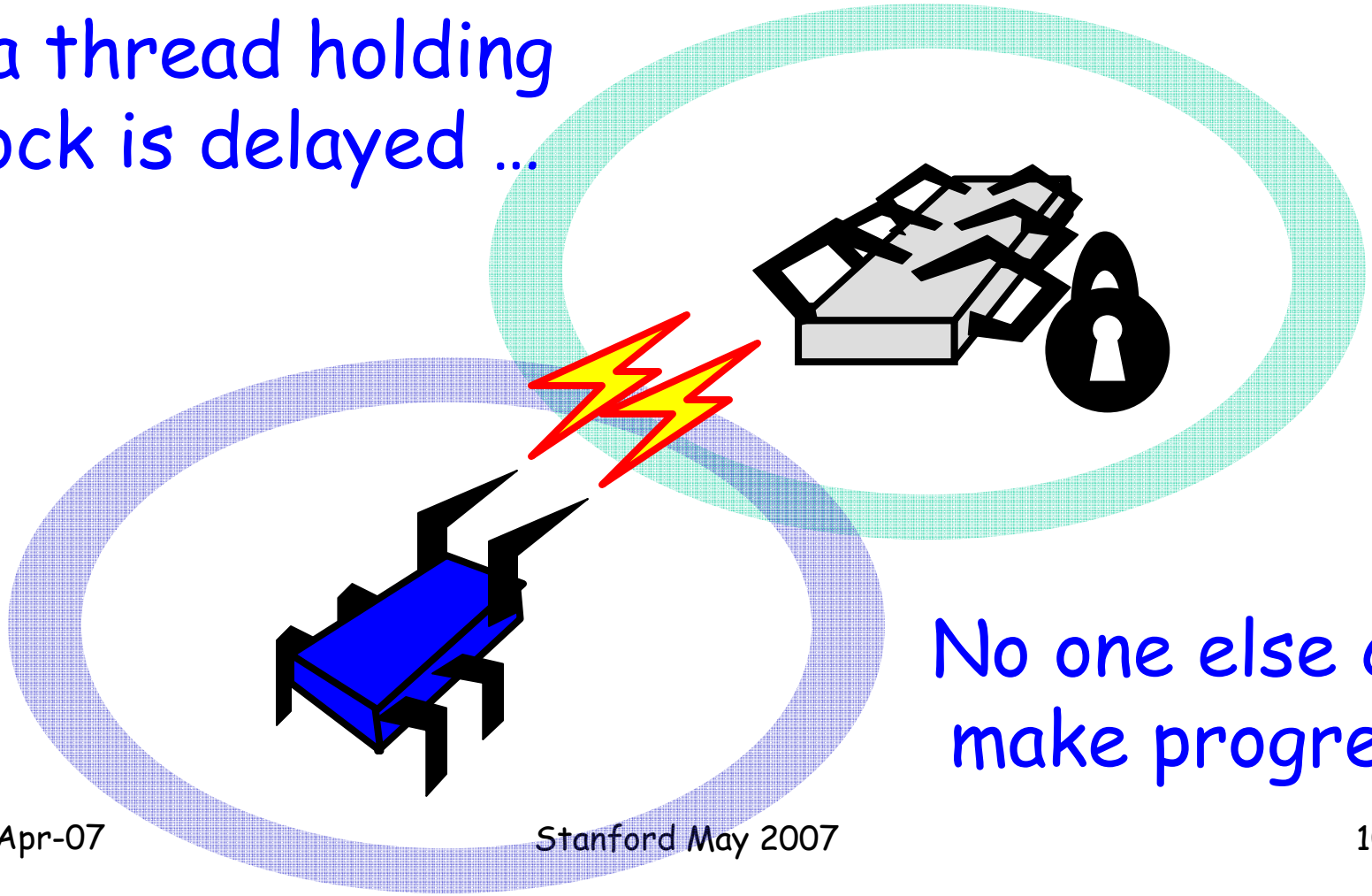
Here comes trouble …

Why Locking Doesn't Scale

- Not Robust
- Relies on conventions
- Hard to Use
  - Conservative
  - Deadlocks
  - Lost wake-ups
- Not Composable

Locks are not Robust

If a thread holding a lock is delayed …



No one else can make progress

Why Locking Doesn't Scale

- Not Robust
- Relies on conventions
- Hard to Use
- Not Composable

Locking Relies on Conventions

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

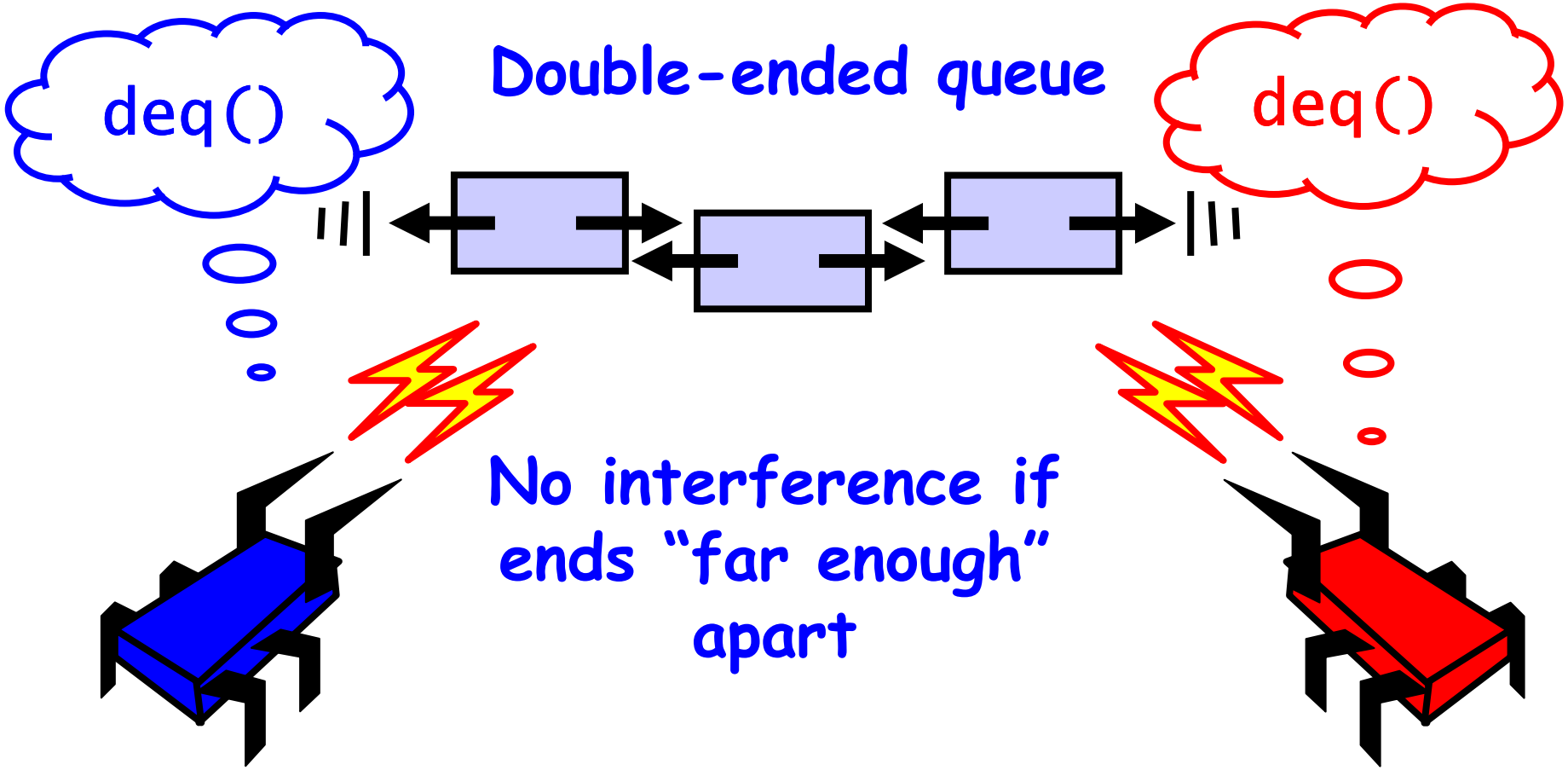
Actual comment from Linux Kernel
(hat tip: Bradley Kuszmaul)

```
/*
 * 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.
 */
```

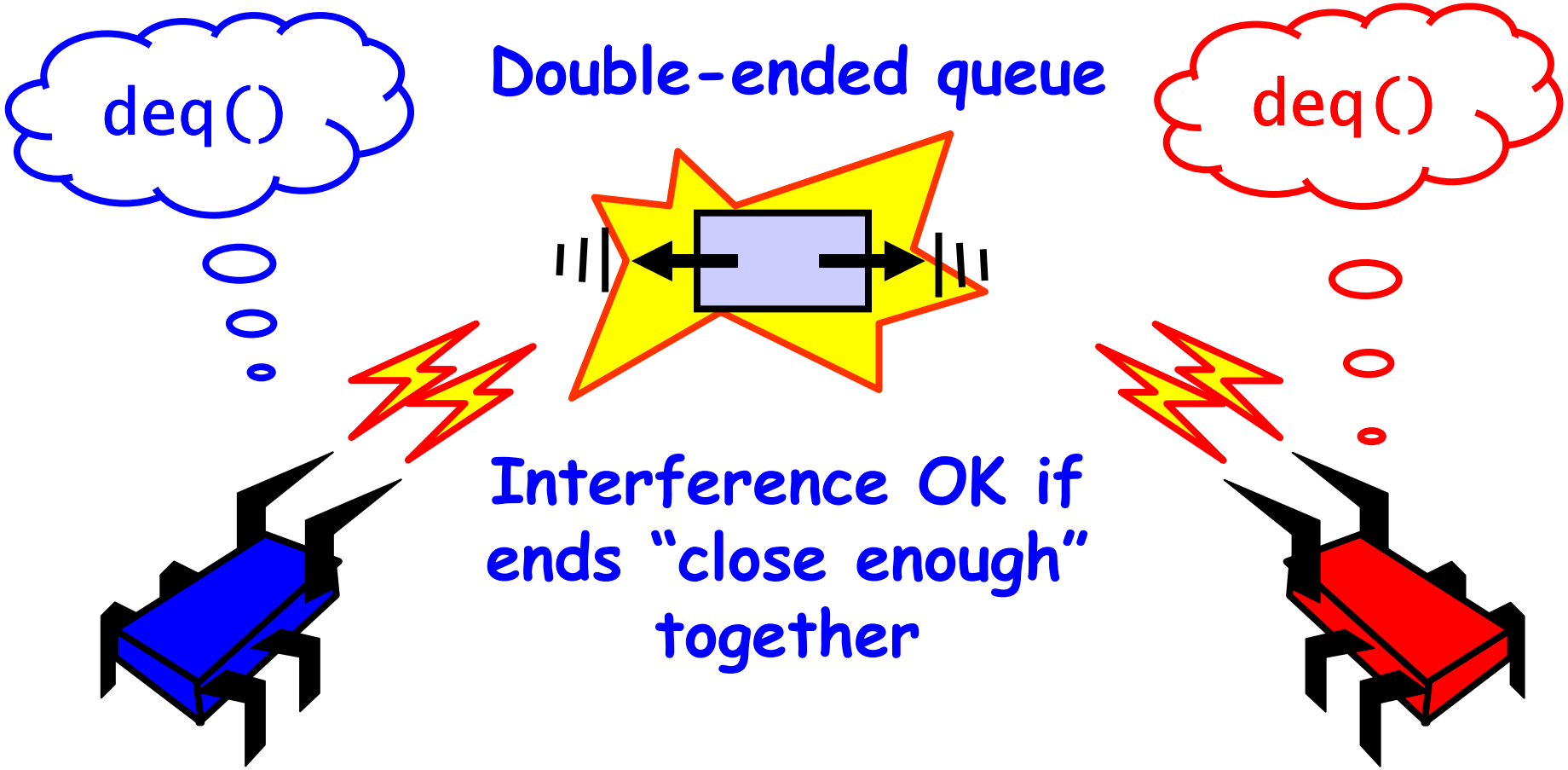
Why Locking Doesn't Scale

- Not Robust
- Relies on conventions
- Hard to Use
- Not Composable

Sadistic Homework



Sadistic Homework



You Try It …

- One lock?
  - Too Conservative
- Locks at each end?
  - Deadlock, too complicated, etc
- Waking blocked dequeuers?
  - Harder that it looks

Actual Solution

- Clean solution would be a publishable result
- [Michael & Scott, PODC 96]
- We are not doing Number Theory: solutions to simply-stated problems should not be publishable.

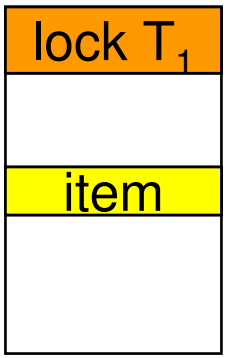
Why Locking Doesn't Scale

- Not Robust
- Relies on conventions
- Hard to Use
- Not Composable

Locks do not compose

Hash Table

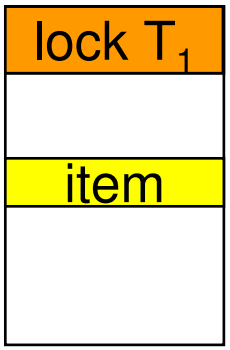
add(T_1, item)

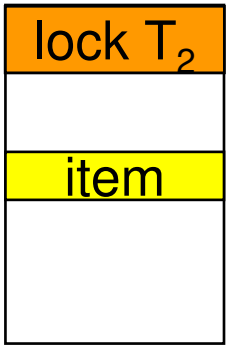


Must lock T_1 before adding item

Move from T_1 to T_2

delete(T_1, item)
add(T_2, item)





Must lock T_2 before deleting from T_1

Exposing lock internals breaks abstraction

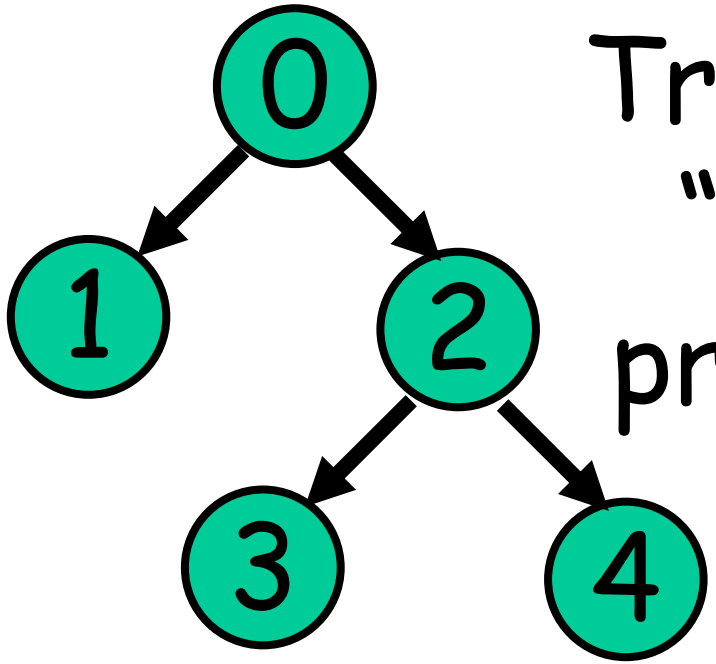
The Transactional Manifesto

- Threads + locking don't scale
- Replace locking with a transactional API
- Host of research issues …
- Let's talk about one.

One Challenge

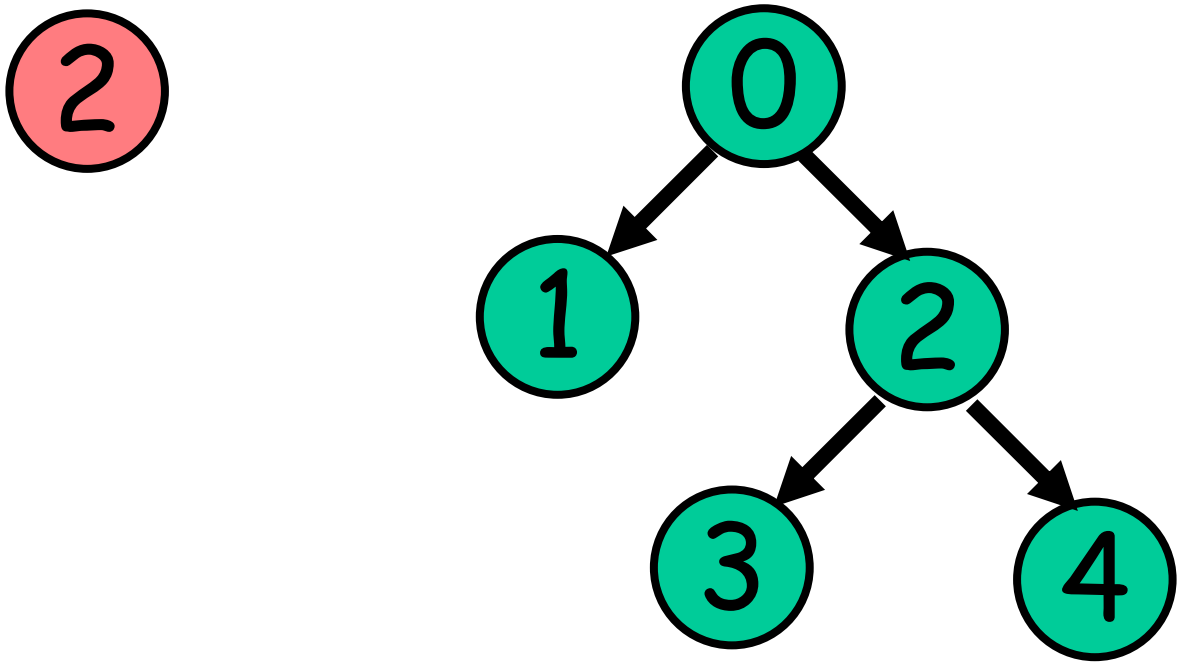
- What are efficient algorithms and data structures for this model?
- Naïve approach (RW locks) provides too little concurrency

Example: Skew Heaps

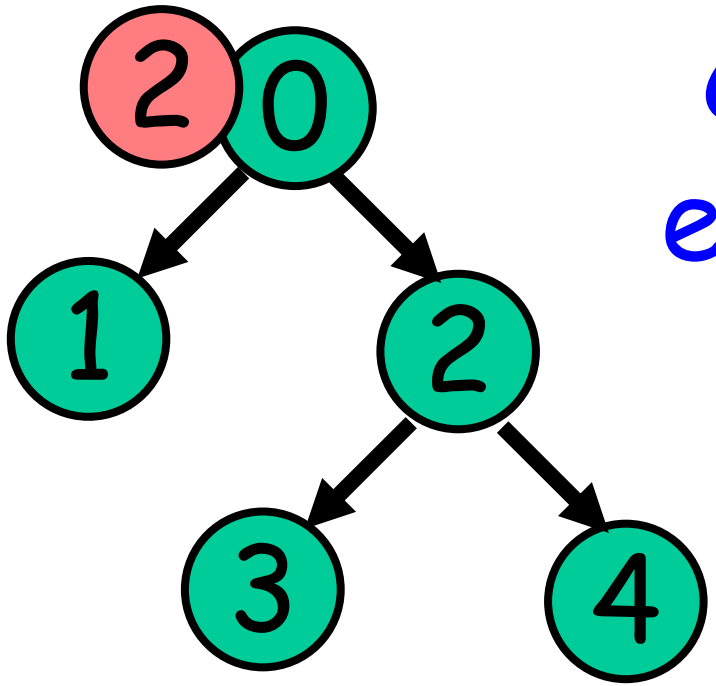


Tree with "heap" property

Skew Heaps



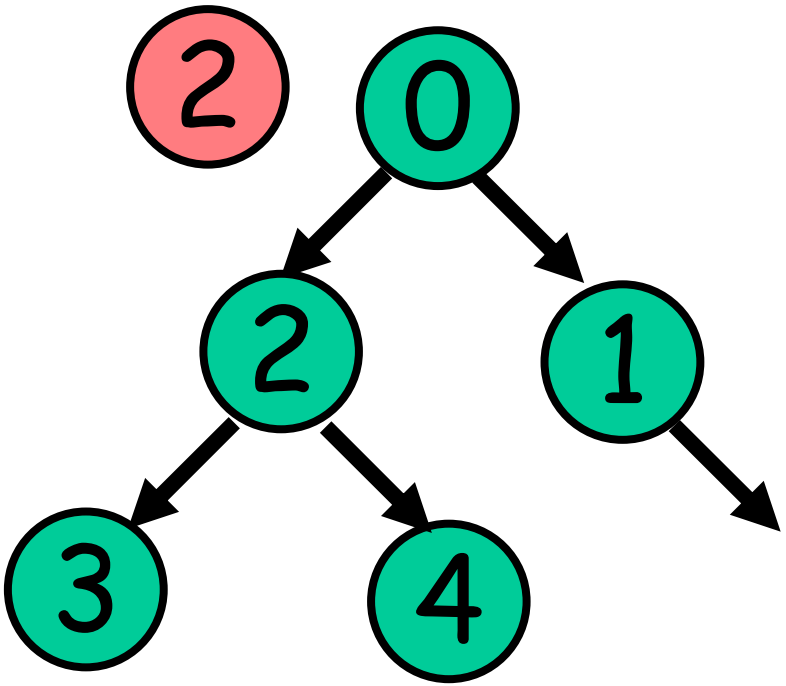
Skew Heaps



Rotate children each visit

Skew Heaps

Go right



Skew Heaps

- No global rebalancing
- Good amortized performance
- Works with fine-grained locking
  - Lock parent
  - Rotate
  - Lock child
  - Release parent …

Transactional Model

- Ill-suited for transactions
  - Rotating children is a write
- Every visit modifies root
  - No concurrency
- Algorithm that works well for locks works poorly for transactions

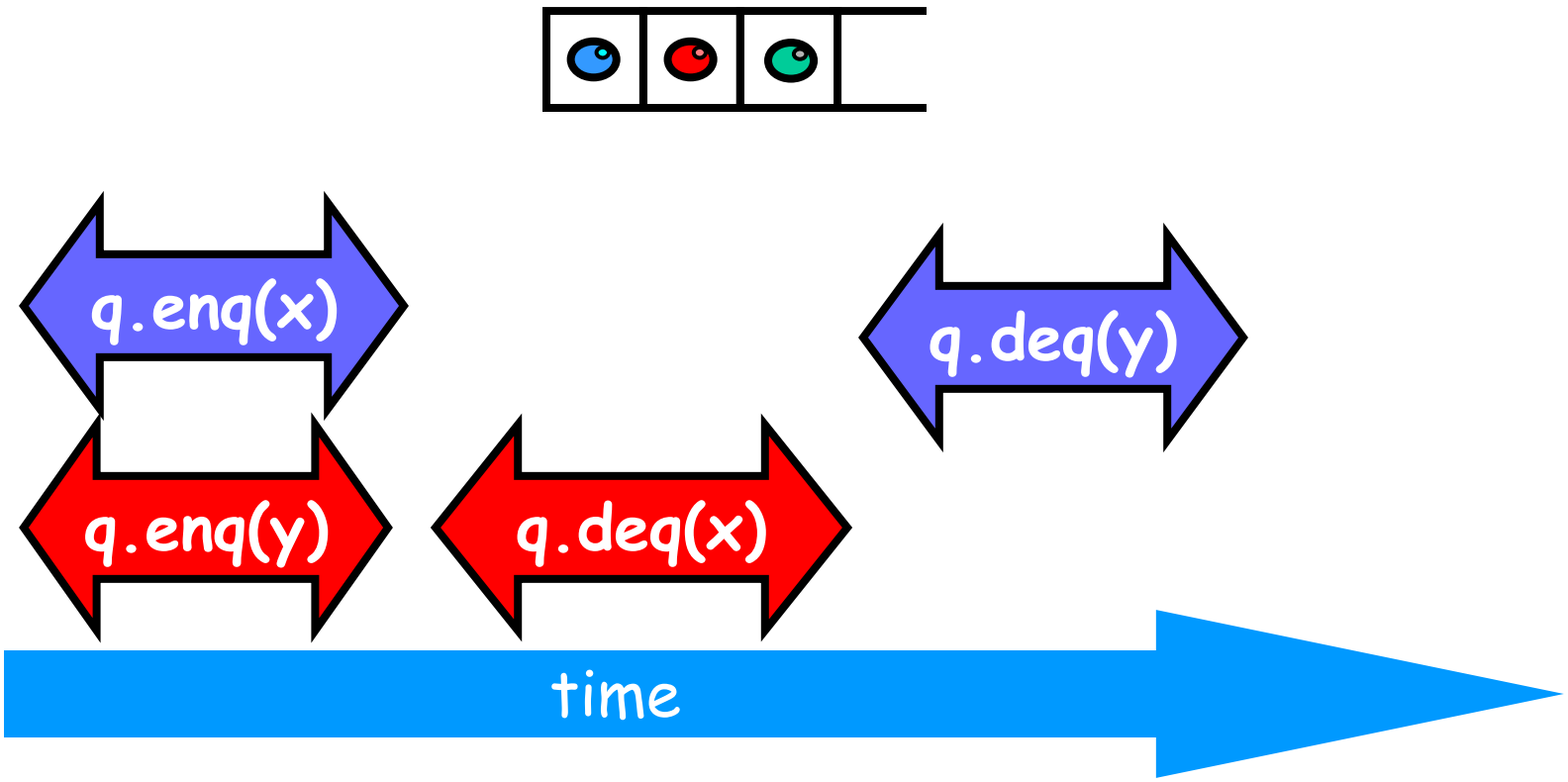
Premise

- Existing TM systems don't distinguish between thread-level and transaction-level synchronization.
- Treating thread-level synchronization as if it were transactional destroys concurrency.

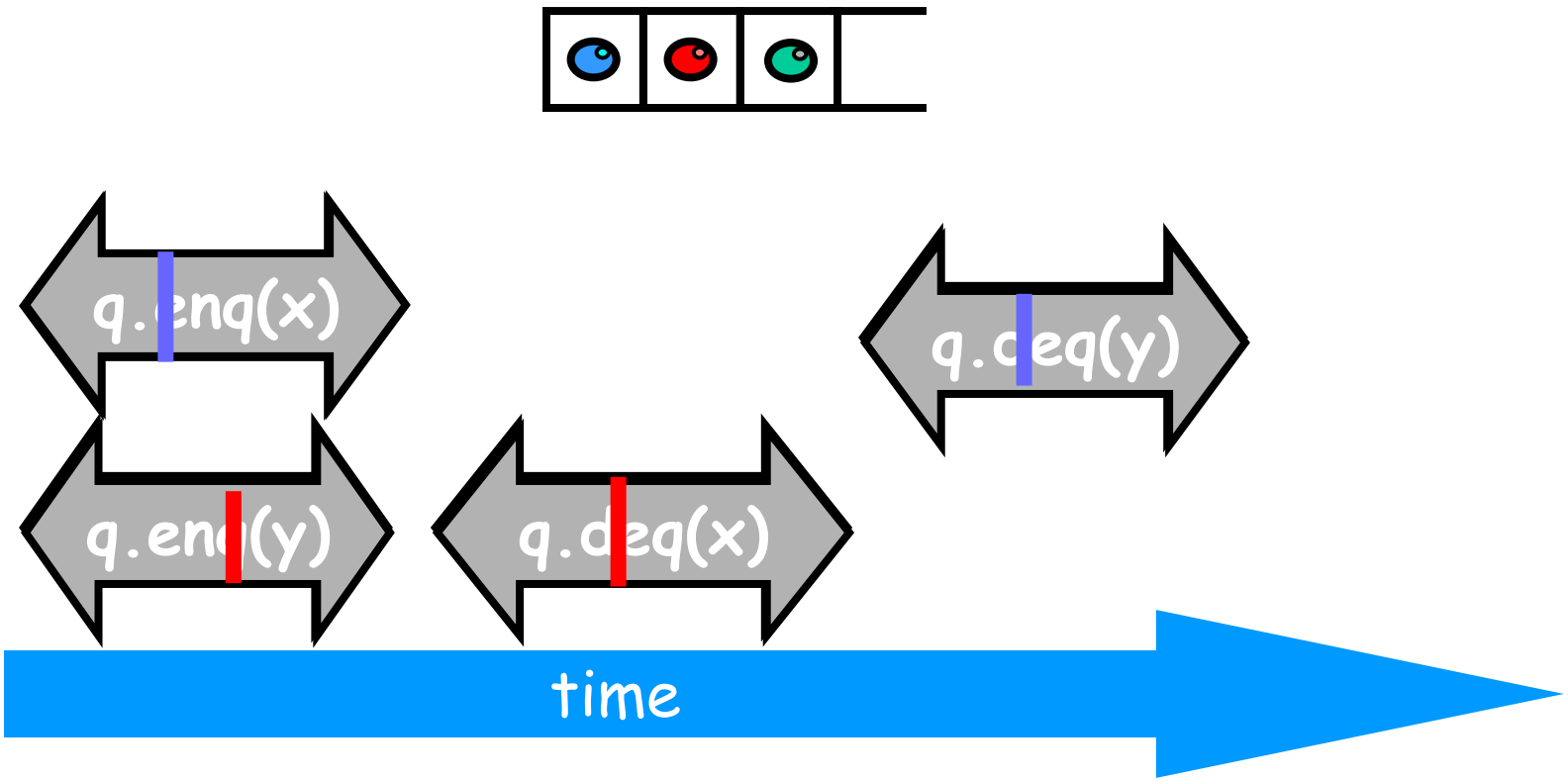
Transactional Boosting

- Methodology for transforming
  - Highly-Concurrent **linearizable** objects
- Into
  - Highly-concurrent **transactional** objects
- Where
  - Non-interfering transactions have same thread-level synchronization granularity

Concurrent Objects



Linearizability



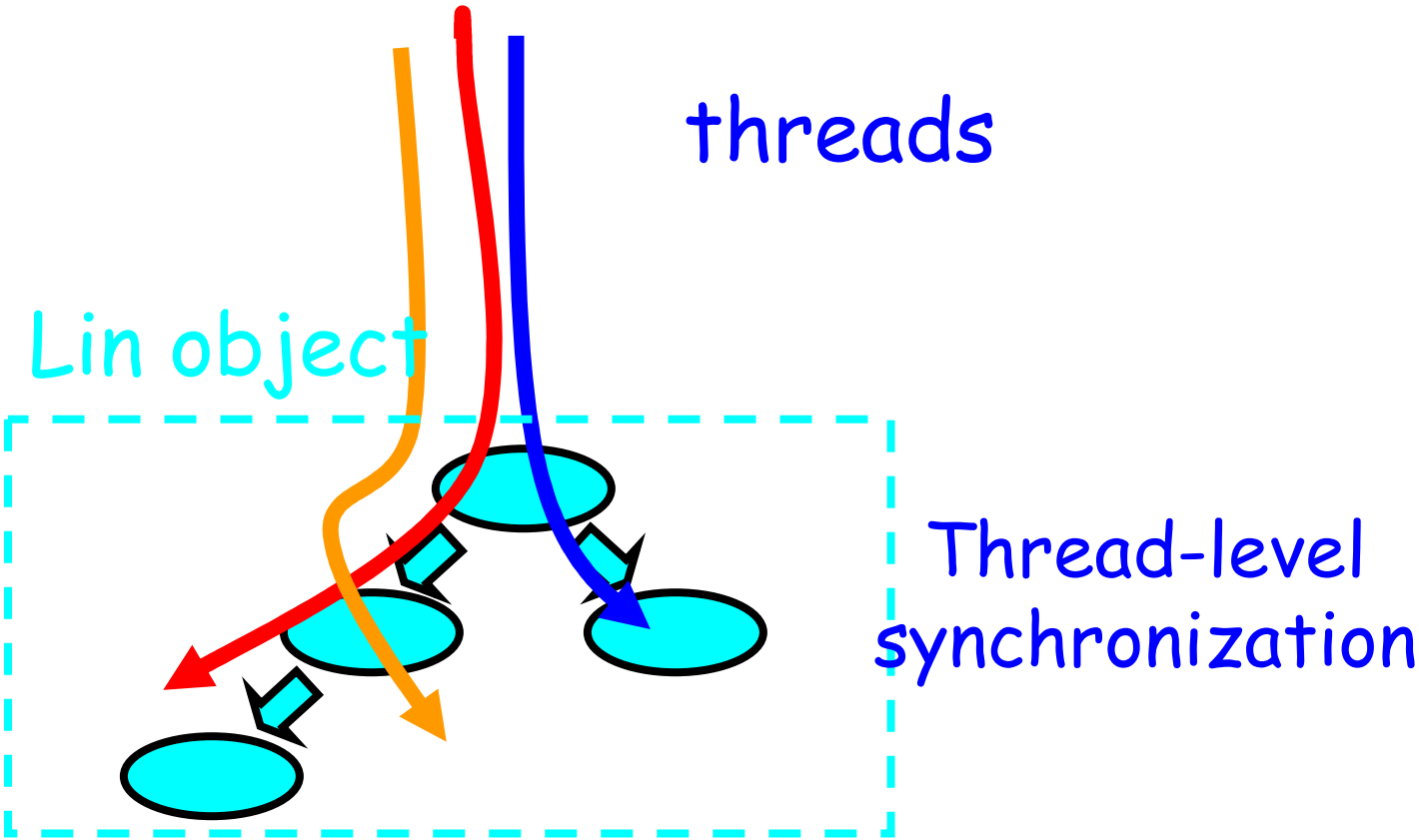
Highly-Concurrent Linearizable Objects

- Active area
  - See `java.util.concurrent`
- Concurrent
  - Sets, hash tables, heaps, queues, stacks, etc.
- Styles
  - Non-blocking (lock-free …)
  - Blocking (fine-grained locks)

Transactional Boosting

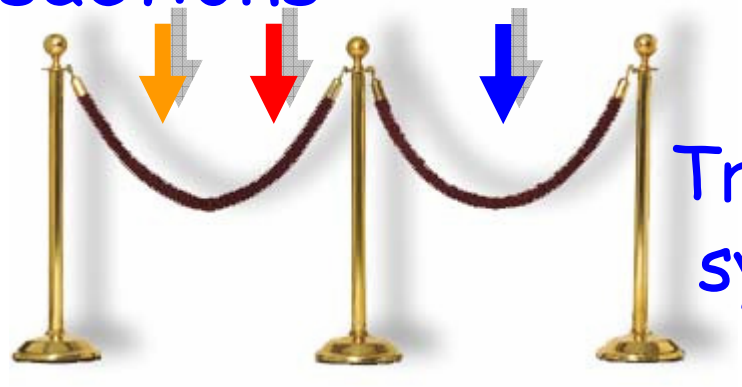
- Linearizable base object
  - Thread-level synchronization
  - Not transaction-aware
- Transactional wrapper
  - Delays or admits transactions' calls
  - Undoes effects on abort
  - Treats base object as **black box**

Linearizable Objects

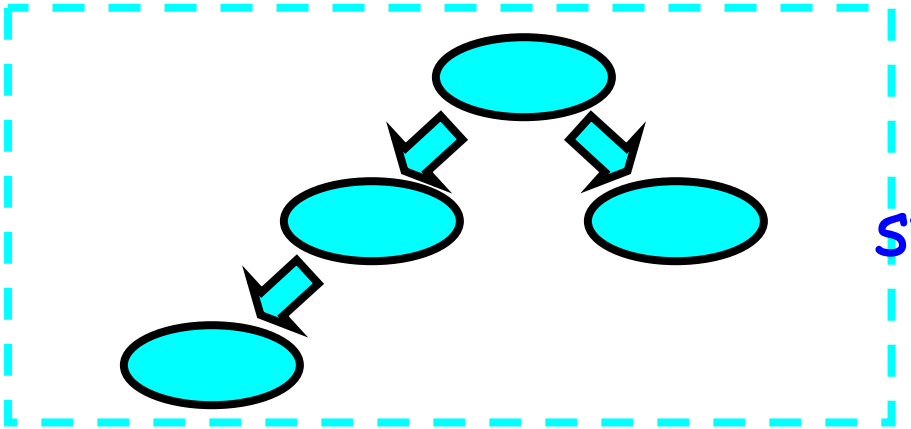


Transactional Boosting

transactions

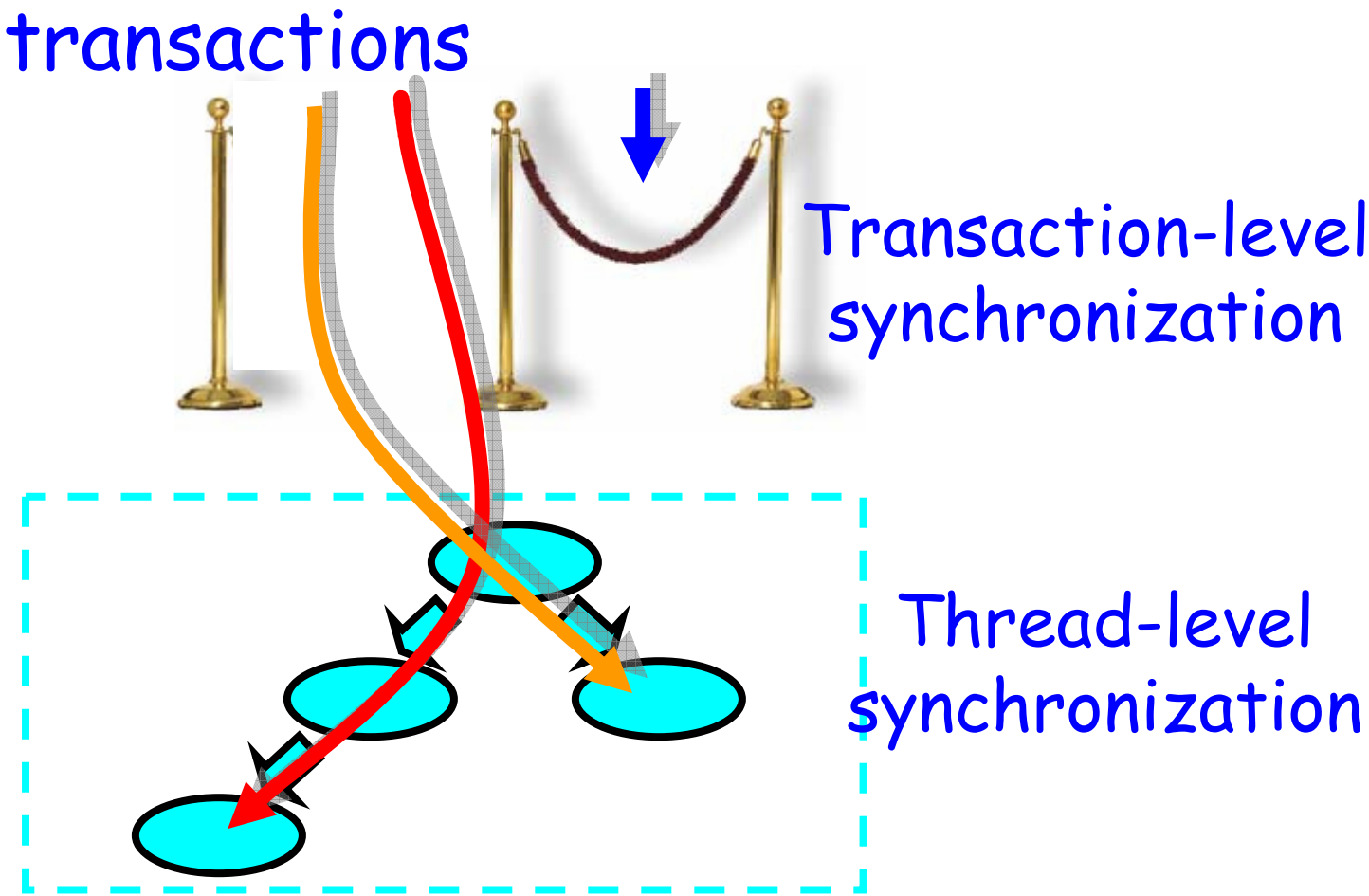


Transaction-level synchronization

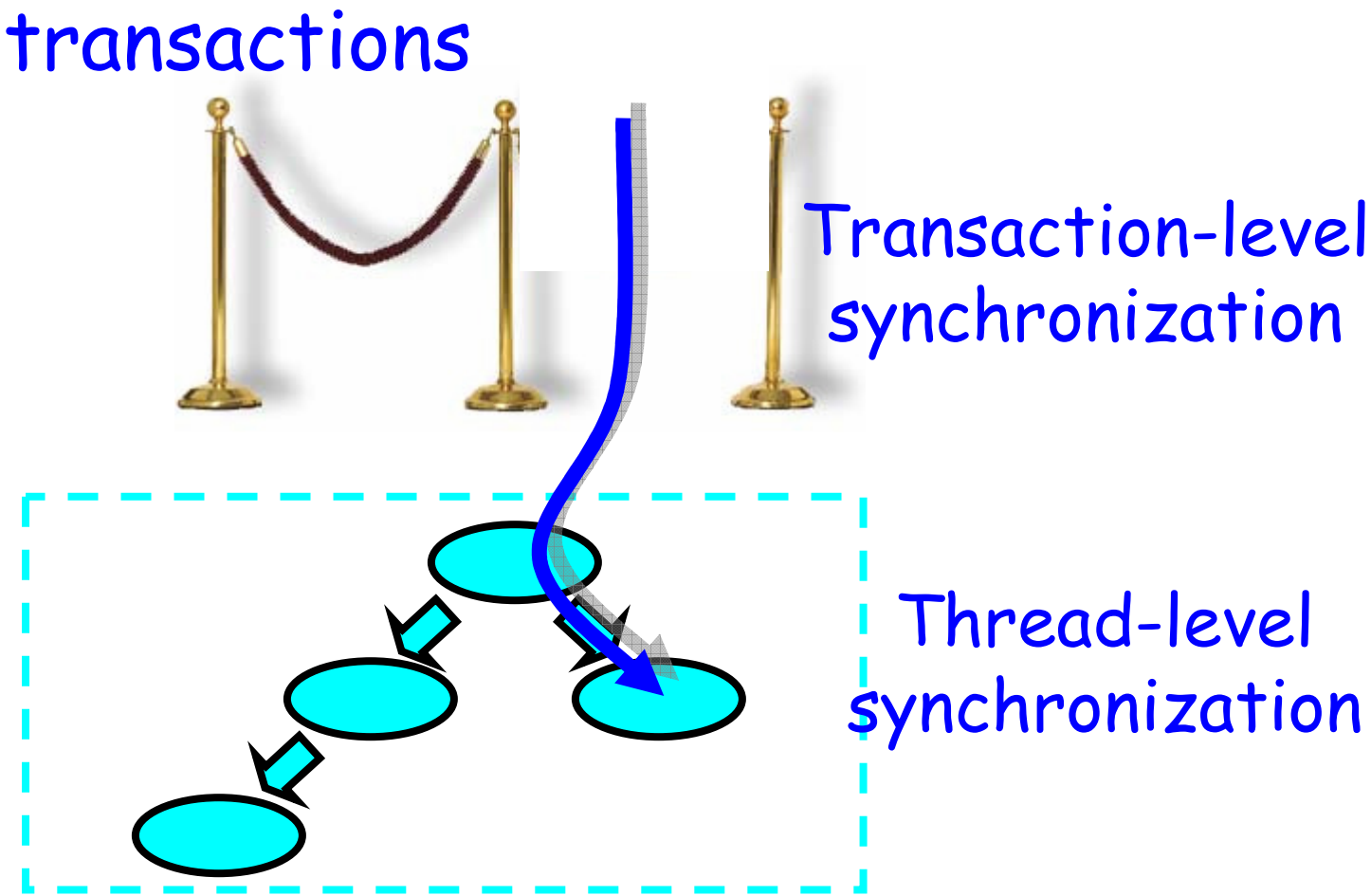


Thread-level synchronization

Transactional Boosting



Transactional Boosting



What's The Catch?

- Concurrent calls must commute
  - Any order yields same results, state
  - There are a few subtleties …
- Each call must have an inverse
  - Applying inverse immediately after restores prior state

Advantages

- Can exploit existing algorithms & data structures
- Synchronization at abstract, not physical level
  - More concurrency, less overhead
- Clear how to use correctly

Runtime Structure

- Block non-commuting calls
  - Application-specific techniques best
- Log calls & results (thread-local)
  - Results needed to know inverse
- On abort:
  - apply inverses

Example: Set

```
public boolean add(int x);
```

- Adds argument to set
- Returns true iff set changed

```
public boolean remove(int x);
```

- Removes argument from set
- Returns true iff set changed

```
public boolean contains(int x);
```

- Returns true iff set contains argument

Inverses

- Depend on result values:
  - add(100)/true^{-1} is remove(100)
  - Add(100)/false^{-1} is no-op
  - remove(100)/true^{-1} is add(100)
  - remove(100)/false^{-1} is no-op

Commutativity

- Calls with different args commute
  - `add(100)` commutes with `add(200)`
  - not with `remove(100)`
- Here, determined by
  - Methods
  - Arguments

Example: SkipList Set

- Linearizable Base Object
  - `ConcurrentSkipListSet`
  - From `java.util.concurrent`
  - lock-free
- Black Box
- Provides inverses

DSTM2

- Java-based Software Transactional Memory
- Can register commit/abort/validate handlers
- www.cs.brown.edu/~mph

SkipList Set

```
public boolean insert(final int v) {
    keyLock.lock(v);
    boolean result = list.add(v);
    if (result) {
      Log.Entry e = new Log.Entry() {
        public void undo() {
          list.remove(v);
        }};
        Log.add(e);
    }
    return result;
  }
```

SkipList Set

```
public boolean insert(final int v) {
    keyLock.lock(v);
    boolean result = list.add(v);
    if (result) {
      Log.Entry e = new Log.Entry() {
        public void undo() {
          list.remove(v);
        }};
        Log.add(e);
    }
    return result;
  }
```

Lock element (actually just hash into an array of locks)

SkipList Set

```
public boolean insert(final int v) {
    keyLock.lock(v);
    boolean result = list.add(v);
    if (result) {
      Log.Entry e = new Log.Entry() {
        public void undo() {
          list.remove(v);
        }};
        Log.add(e);
    }
    return result;
  }
```

On return, no concurrent calls to same element …

SkipList Set

```
public boolean insert(final int v) {
    keyLock.lock(v);
    boolean result = list.add(v);
    if (result) {
      Log.Entry e = new Log.Entry() {
        public void undo() {
          list.remove(v);
        }};
        Log.add(e);
    }
    return result;
  }
```

Call linearizable object's method

SkipList Set

If set was modified, log call

```
public boolean add(final int v) {
  keyLock.lock(v);
  boolean result = list.add(v);
  if (result) {
    Log.Entry e = new Log.Entry() {
      public void undo() {
        list.remove(v);
      }};
      Log.add(e);
  }
  return result;
}
```

SkipList Set

```
public boolean insert(final int v) {
    keyLock.lock(v);
    boolean result = list.add(v);
    if (result) {
      Log.Entry e = new Log.Entry() {
        public void undo() {
          list.remove(v);
        }};
        Log.add(e);
    }
    return result;
  }
```

Register inverse

Example: Concurrent Heap

- Linearizable Base Object
  - Concurrent Heap [HMPS 94]
  - Uses fine-grained locks
- Must add inverses
- Can use call results for commutativity

Heap

```
public void add(int v);
```

- Adds value to heap
- Duplicates OK

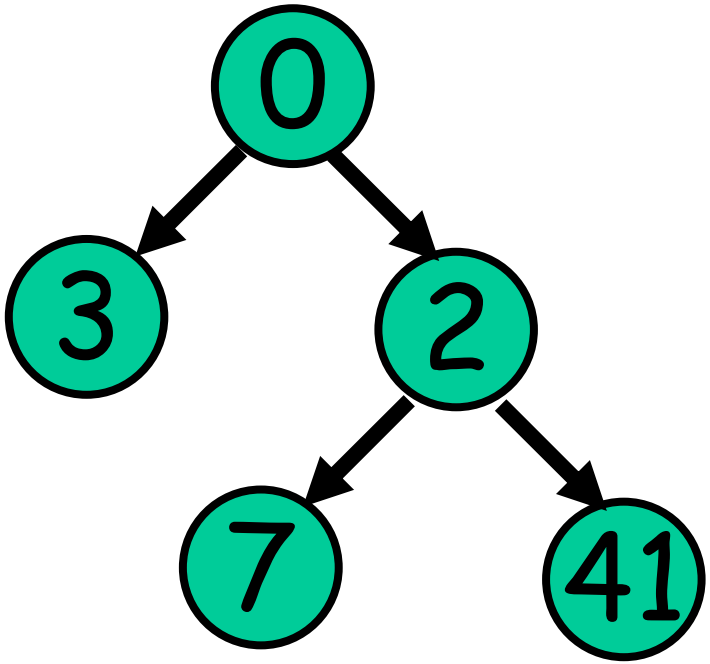
```
public int removeMin();
```

- Removes & returns least value in heap

Problem

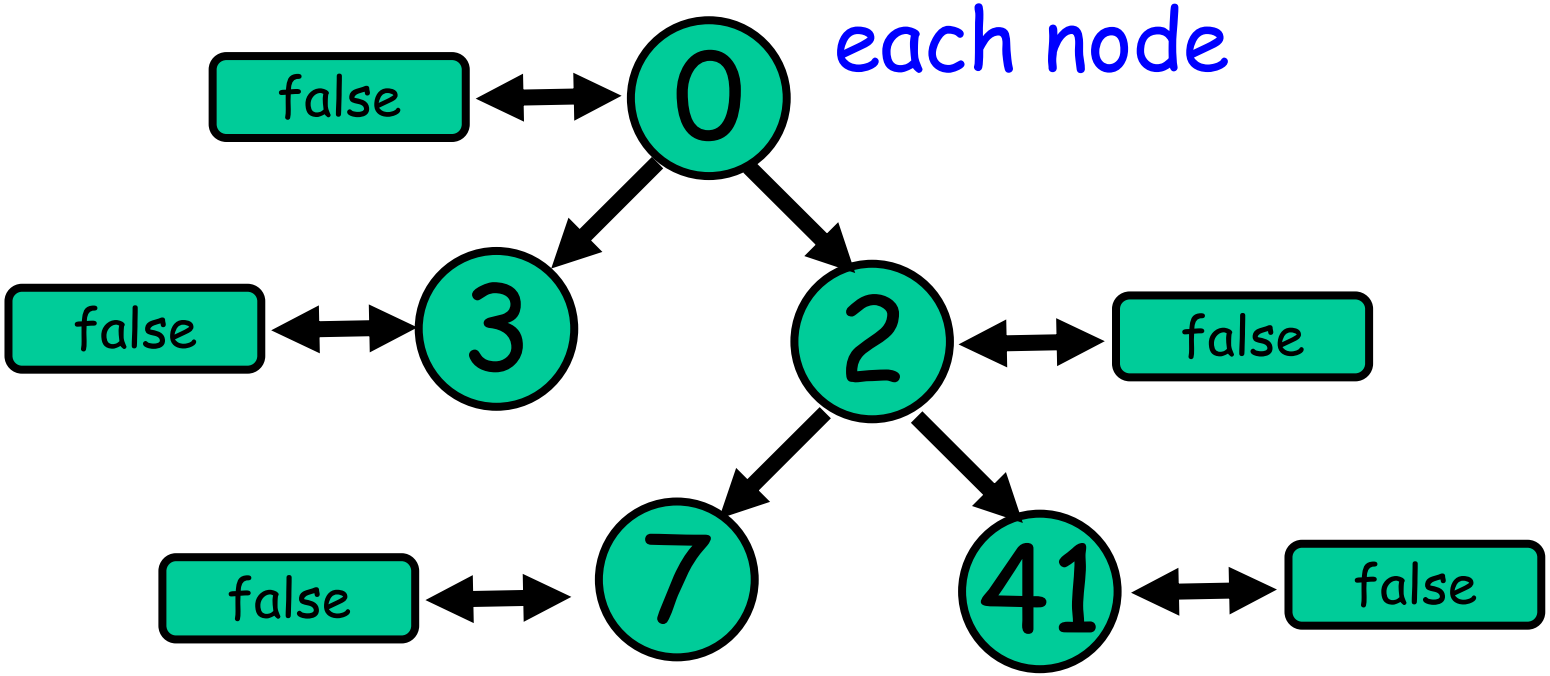
- In the original algorithm,
  - `add(x)` has no inverse
- Not at all clear
  - how to implement `remove(x)` efficiently
- Fortunately,
  - That's the **wrong question**

Typical Heap



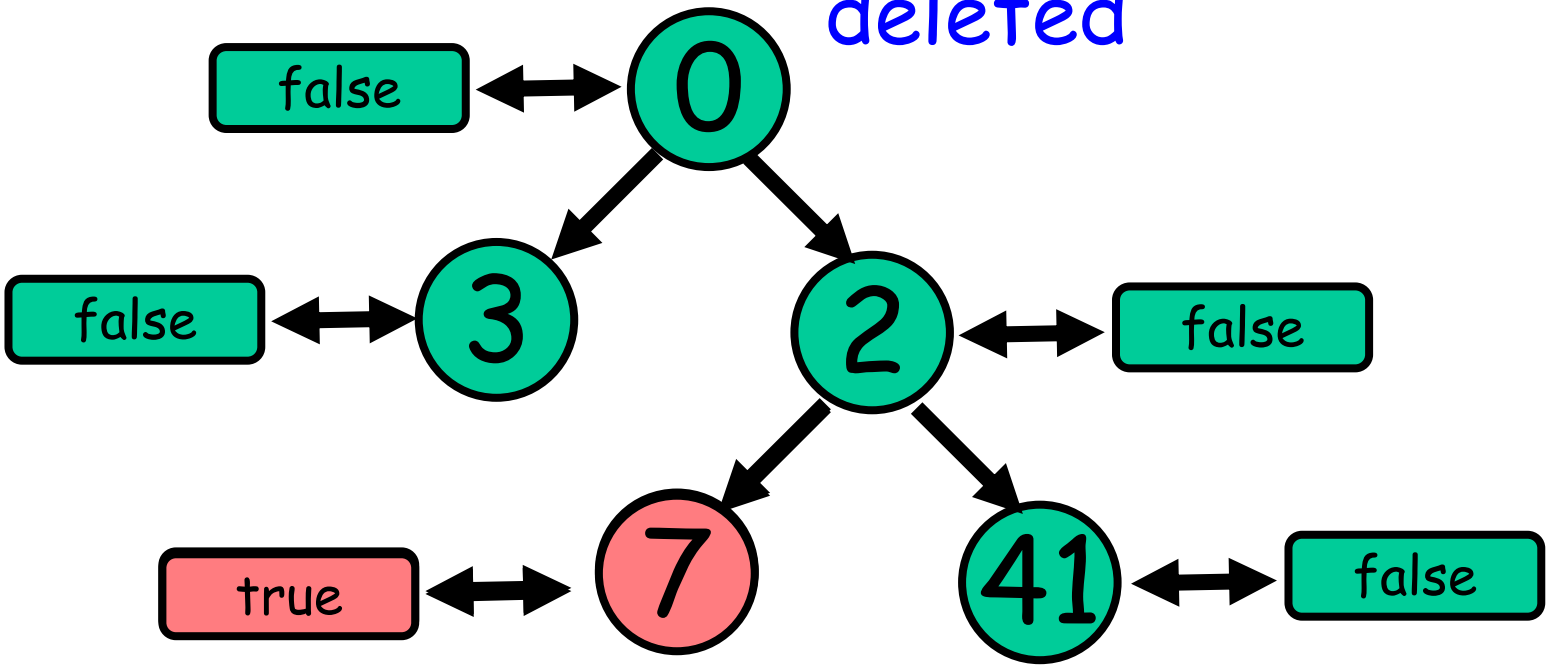
Typical Heap

Add *deleted* bit to each node

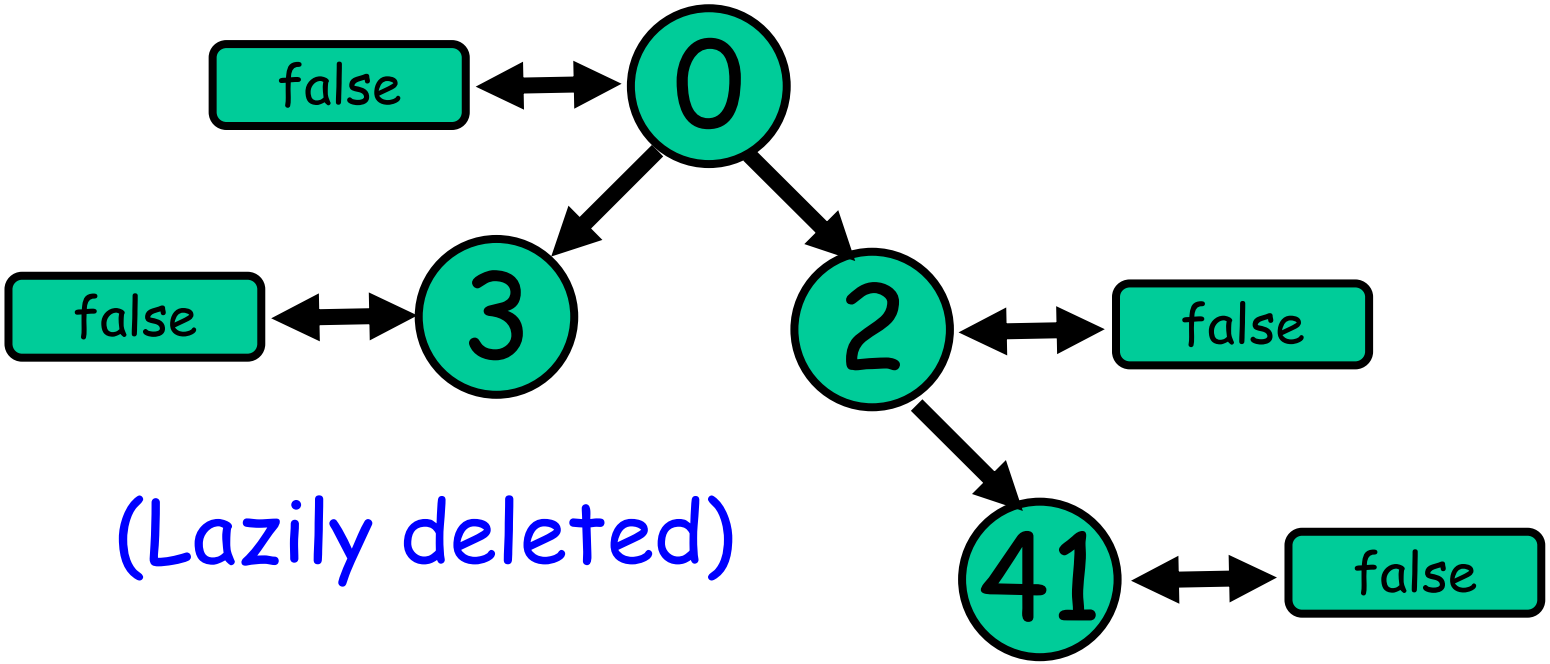


Typical Heap

True means logically deleted



Typical Heap



(Lazily deleted)

An Inverse for Add(x)

- `add()` returns reference to heap node
- `add`$^{-1}$`(node)` sets *deleted* bit
  - Constant time
- `removeMin()`
  - Discards deleted entries …
  - And tries again
- Logarithmic amortized time

Commutativity

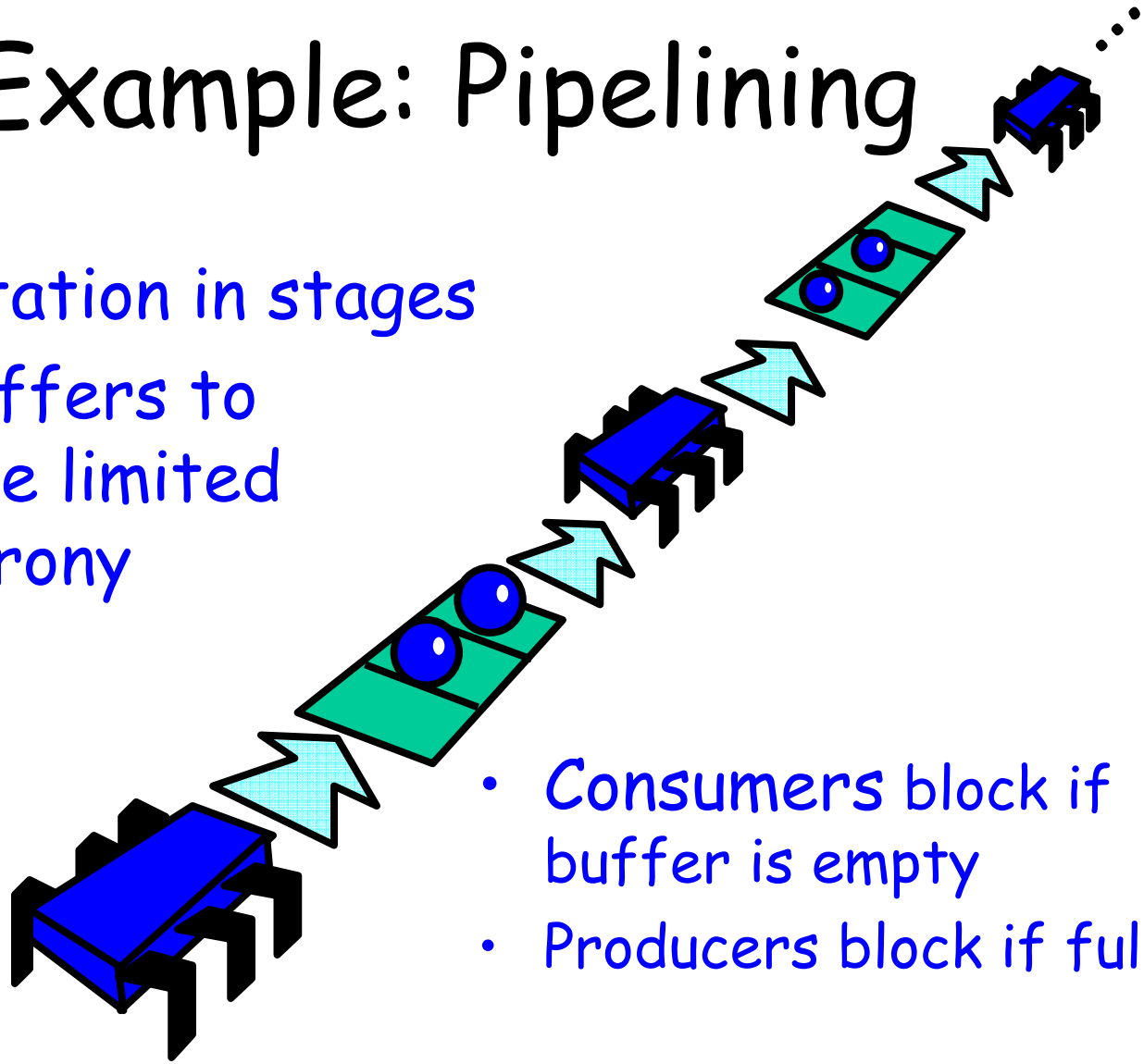
- `add(…)` calls commute with each other
- `removeMin()/x` commutes with `add(y)` calls if `x ≤ y` …
- Commutativity depends on
  - Methods
  - Arguments
  - And now, **results**

Heap Locks

- Read/Write locks
- Where write lock has target value
- `removeMin()` call acquires
  - Write lock with target $+\infty$
  - On return, **downgrades** to result value
- `add(x)` acquires read lock
  - Conflicts if write lock target is > x

Example: Pipelining

- Computation in stages
- Use buffers to tolerate limited asynchrony



- Consumers block if buffer is empty
- Producers block if full

Transactional Blocking Queue

```
public void offer(int v);
```

- Puts value at end of queue
- Blocks while queue is full

```
public int take();
```

- Remove & return value from front of queue
- Blocks while queue is empty

Inverses

```
public void offerLast(int v);
public int  takeLast();       // inverse
```

```
public int  takeFirst();
public void offerFirst();     // inverse
```

- Use a double-ended queue (DEQueue)
  - `java.util.concurrent.BlockingDeque`

Commutativity

- `offer(x)` commutes with `take()` if the buffer is non-empty
- Commutativity depends on
  - Method
  - Arguments
  - Results
  - And now, **object state**

Transactional Semaphores

- Increment
  - Applied on commit
- Decrement
  - Blocks if zero
  - Undone on abort
- Low-rent, easy to implement with DSTM2 commit/abort handlers

Experimental Validation

- Not ready in time for this talk.
- Sorry.
- But …
  - Invocation-based locking seem best for short transactions
  - Result-based locking seem best for long transactions

But What About Open Nested Transactions?

- Nested transactions
  - Commit: release effects to parent
  - Abort: discard effects, parent OK
- Open nested transactions
  - Commit: release effects to **everyone**
  - Parent abort: run "compensating" action

Open Nested Transactions (humble opinion)

toenail trimmer

Limitations of Open Nested Transactions

- No user manual
  - How can you tell if what you did is **correct**?
  - No proof rules
  - Recent proposals reintroduce **deadlocks**
  - Troubling **semantic** questions
    - All the ways parent and child read and write sets can intersect …

Limitations of Open Nested Transactions

- Lack of expressive power
  - Lock coupling algorithms?
    - (Locking intervals not nested)
  - Fine-grained thread interleavings?
    - Each memory op is open transaction?
    - With its own registered compensation?
  - Low-level object not black box

Limitations of Transactional Boosting

- Must recognize **commutativity**
- Must have **inverses**
- Seems to work for **collections**
  - Sets, maps, stacks, queues, etc….
  - Lots of calls commute
  - Inverses often exist (or can be added)

I, for one, Welcome our new Multicore Overlords …

- Multicore architectures force us to rethink how we do synchronization
- Standard locking model won't work
- Transactional model might
  - Software
  - Hardware
- A TV-community full-employment act!