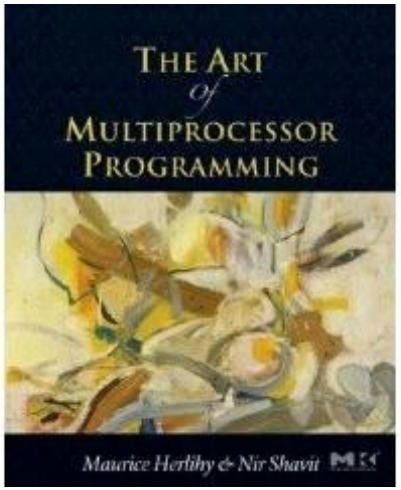
Linked Lists: Locking, Lock-Free, and Beyond …



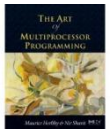
Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

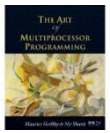
Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by scalable locks



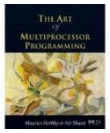
Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by scalable locks
- Should increase throughput
  - Not possible if inherently sequential
  - Surprising things are parallelizable



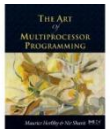
Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using scalable locks



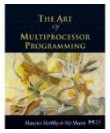
Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using scalable locks
  - Easy to reason about
    - In simple cases



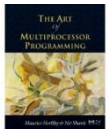
Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using scalable locks
  - Easy to reason about
    - In simple cases
- So, are we done?



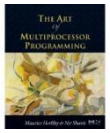
Coarse-Grained Synchronization

- Sequential bottleneck
  - Threads “stand in line”



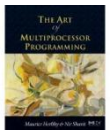
Coarse-Grained Synchronization

- Sequential bottleneck
  - Threads “stand in line”
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse



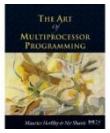
Coarse-Grained Synchronization

- Sequential bottleneck
  - Threads “stand in line”
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse
- So why even use a multiprocessor?
  - Well, some apps inherently parallel …



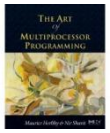
This Lecture

- Introduce four “patterns”
  - Bag of tricks …
  - Methods that work more than once …



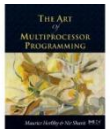
This Lecture

- Introduce four “patterns”
  - Bag of tricks …
  - Methods that work more than once …
- For highly-concurrent objects
  - Concurrent access
  - More threads, more throughput



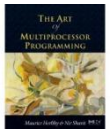
First: Fine-Grained Synchronization

- Instead of using a single lock …
- Split object into
  - Independently-synchronized components
- Methods conflict when they access
  - The same component …
  - At the same time



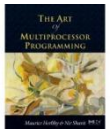
Second: Optimistic Synchronization

- Search without locking …



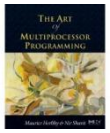
Second: Optimistic Synchronization

- Search without locking …
- If you find it, lock and check …
  - OK: we are done
  - Oops: start over



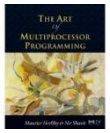
Second: Optimistic Synchronization

- Search without locking …
- If you find it, lock and check …
  - OK: we are done
  - Oops: start over
- Evaluation
  - Usually cheaper than locking, but
  - Mistakes are expensive



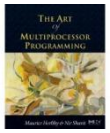
Third: Lazy Synchronization

- Postpone hard work
- Removing components is tricky
  - Logical removal
    - Mark component to be deleted
  - Physical removal
    - Do what needs to be done



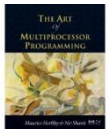
Fourth: Lock-Free Synchronization

- Don’t use locks at all
  - Use compareAndSet() & relatives …



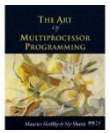
Fourth: Lock-Free Synchronization

- Don’t use locks at all
  - Use compareAndSet() & relatives …
- Advantages
  - No Scheduler Assumptions/Support



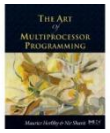
Fourth: Lock-Free Synchronization

- Don't use locks at all
  - Use compareAndSet() & relatives …
- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead



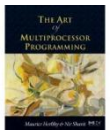
Linked List

- Illustrate these patterns …
- Using a list-based Set
  - Common application
  - Building block for other apps



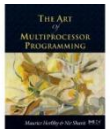
Set Interface

- Unordered collection of items



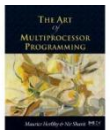
Set Interface

- Unordered collection of items
- No duplicates



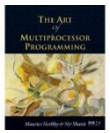
Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - `add(x)` put `x` in set
  - `remove(x)` take `x` out of set
  - `contains(x)` tests if `x` in set



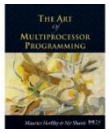
List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```



List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

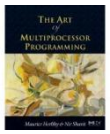


Add item to set

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

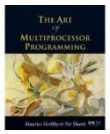
Remove item from set



List-Based Sets

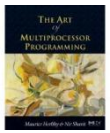
```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

Is item in set?



List Node

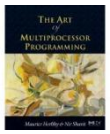
```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```



List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

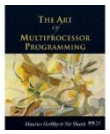
item of interest



List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

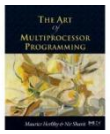
Usually hash code



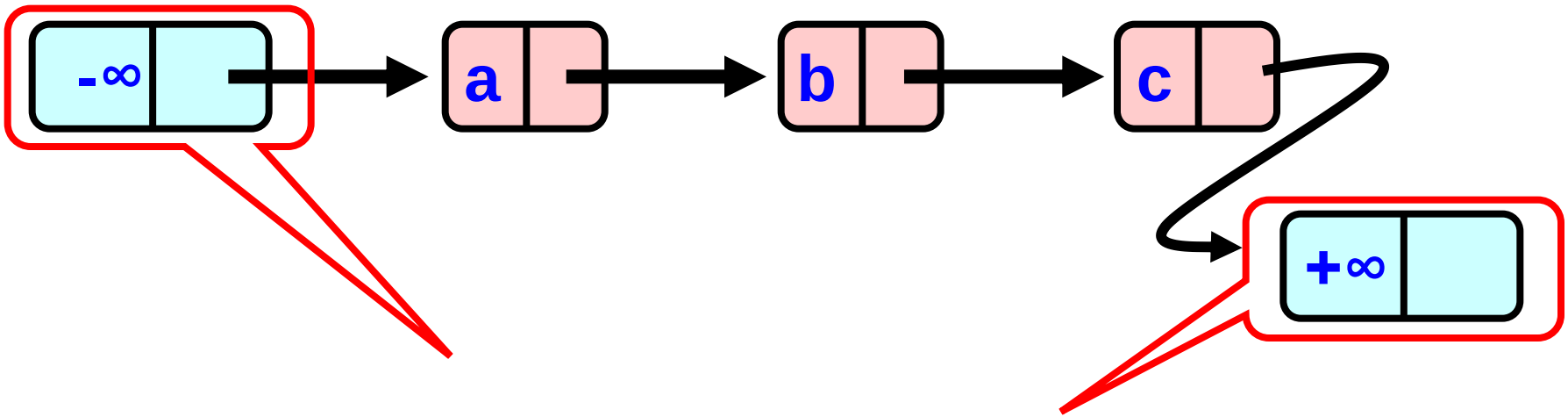
List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

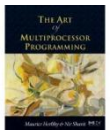
Reference to next node



The List-Based Set

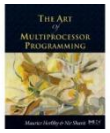


Sorted with Sentinel nodes
(min & max possible keys)



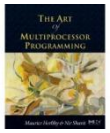
Reasoning about Concurrent Objects

- Invariant
  - Property that always holds



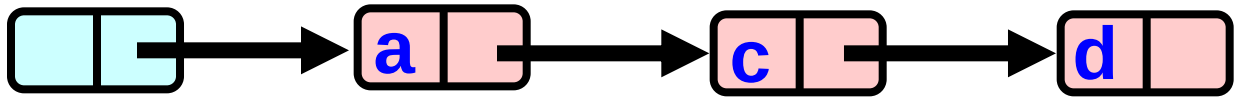
Reasoning about Concurrent Objects

- Invariant
  - Property that always holds
- Established because
  - True when object is **created**
  - Truth **preserved** by each method
    - Each **step** of each method

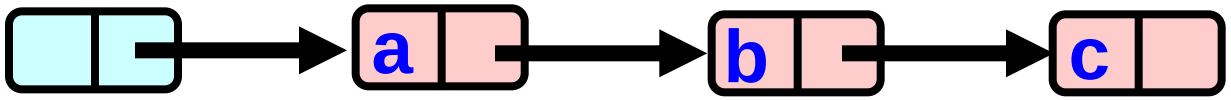


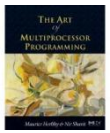
Sequential List Based Set

add()



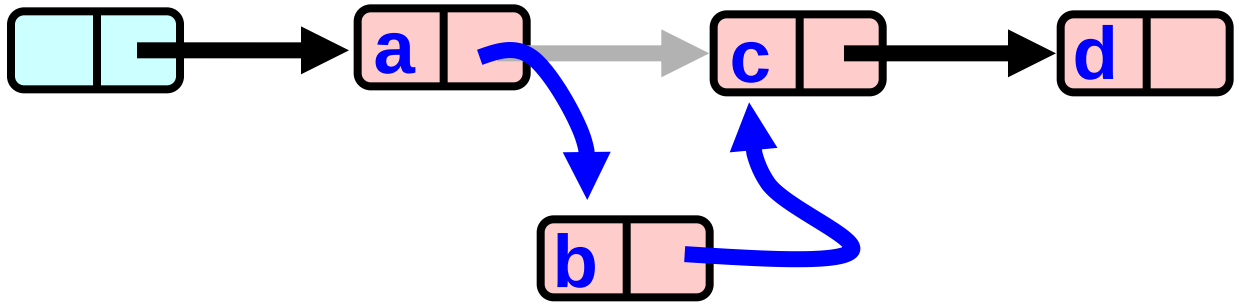
remove()



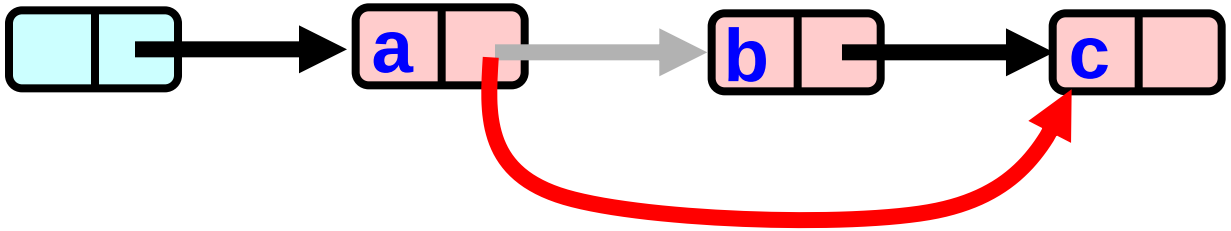


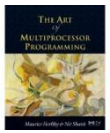
Sequential List Based Set

add()

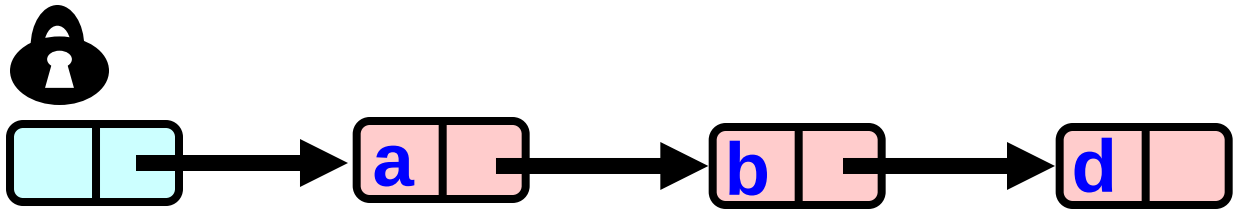


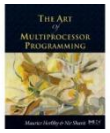
remove()



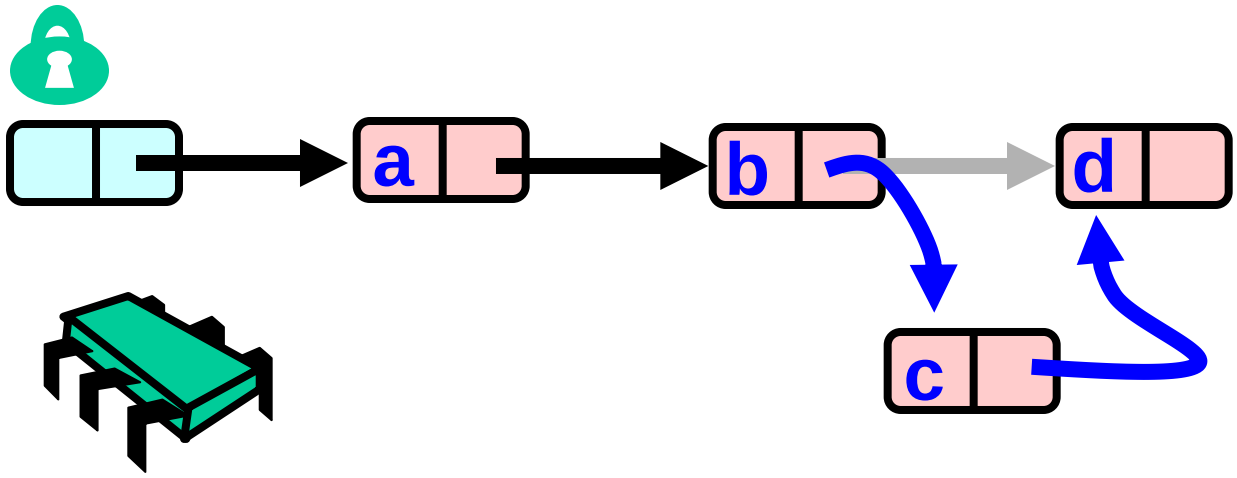


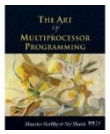
Coarse-Grained Locking



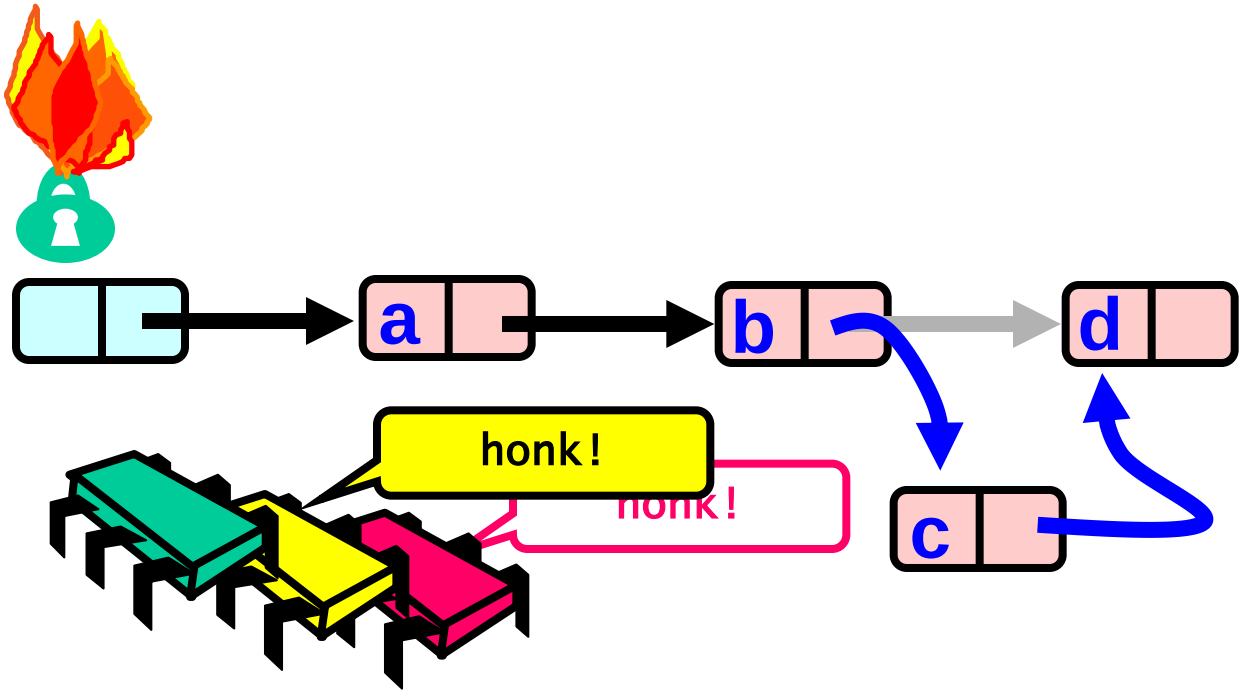


Coarse-Grained Locking

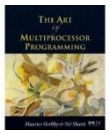




Coarse-Grained Locking

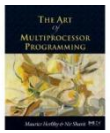


Simple but hotspot + bottleneck



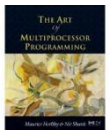
Coarse-Grained Locking

- Easy, same as synchronized methods
  - "One lock to rule them all …"



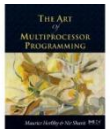
Coarse-Grained Locking

- Easy, same as synchronized methods
  - "One lock to rule them all …"
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention
  - Queue locks help
  - But bottleneck still an issue



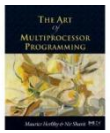
Fine-grained Locking

- Requires **careful** thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"

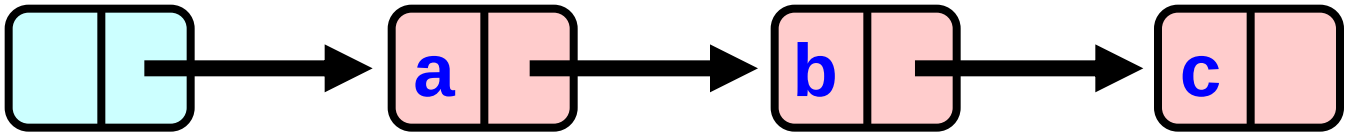


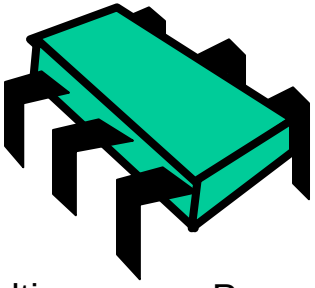
Fine-grained Locking

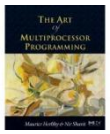
- Requires **careful** thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other



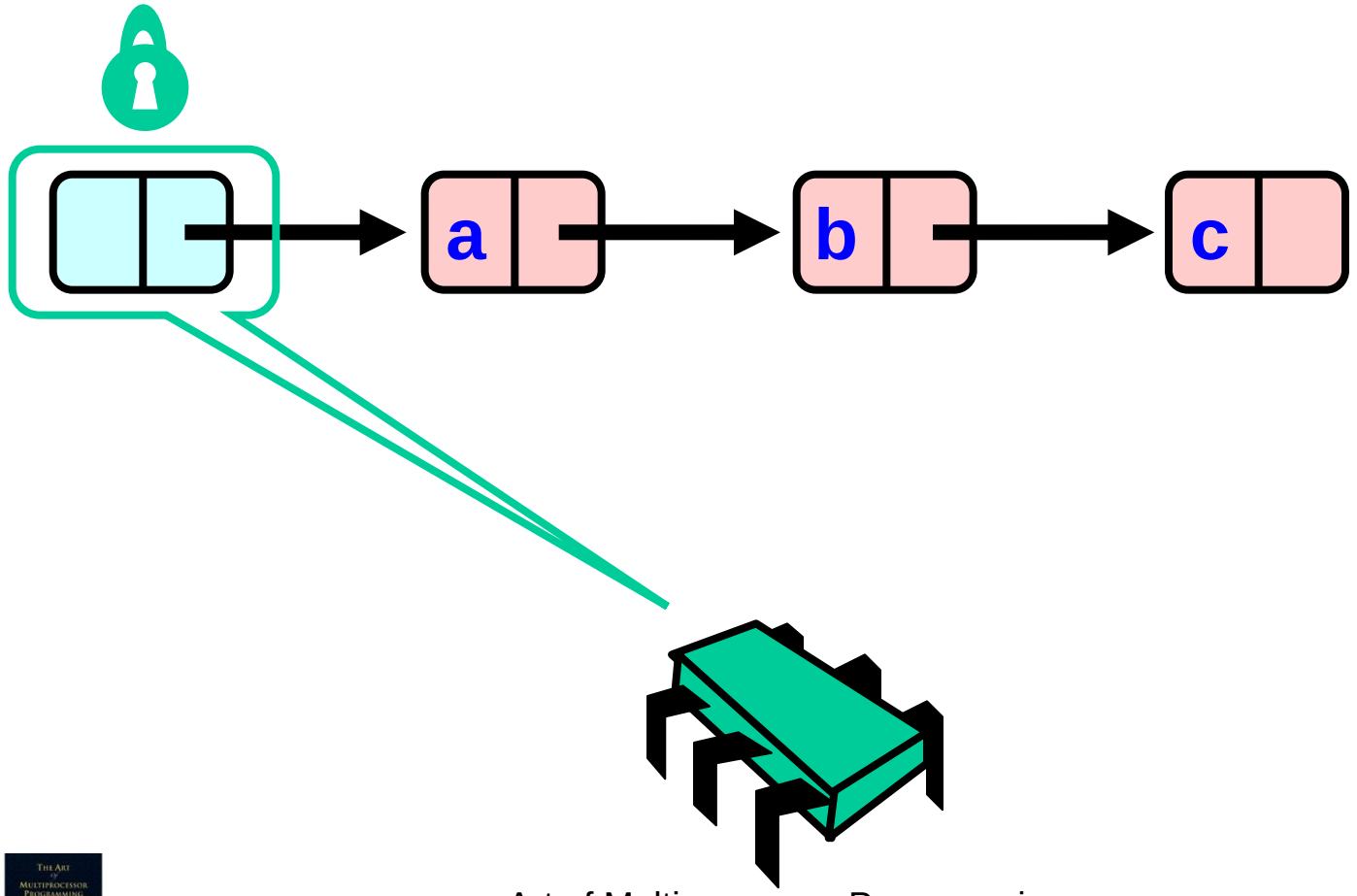
Hand-over-Hand locking

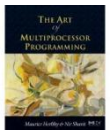




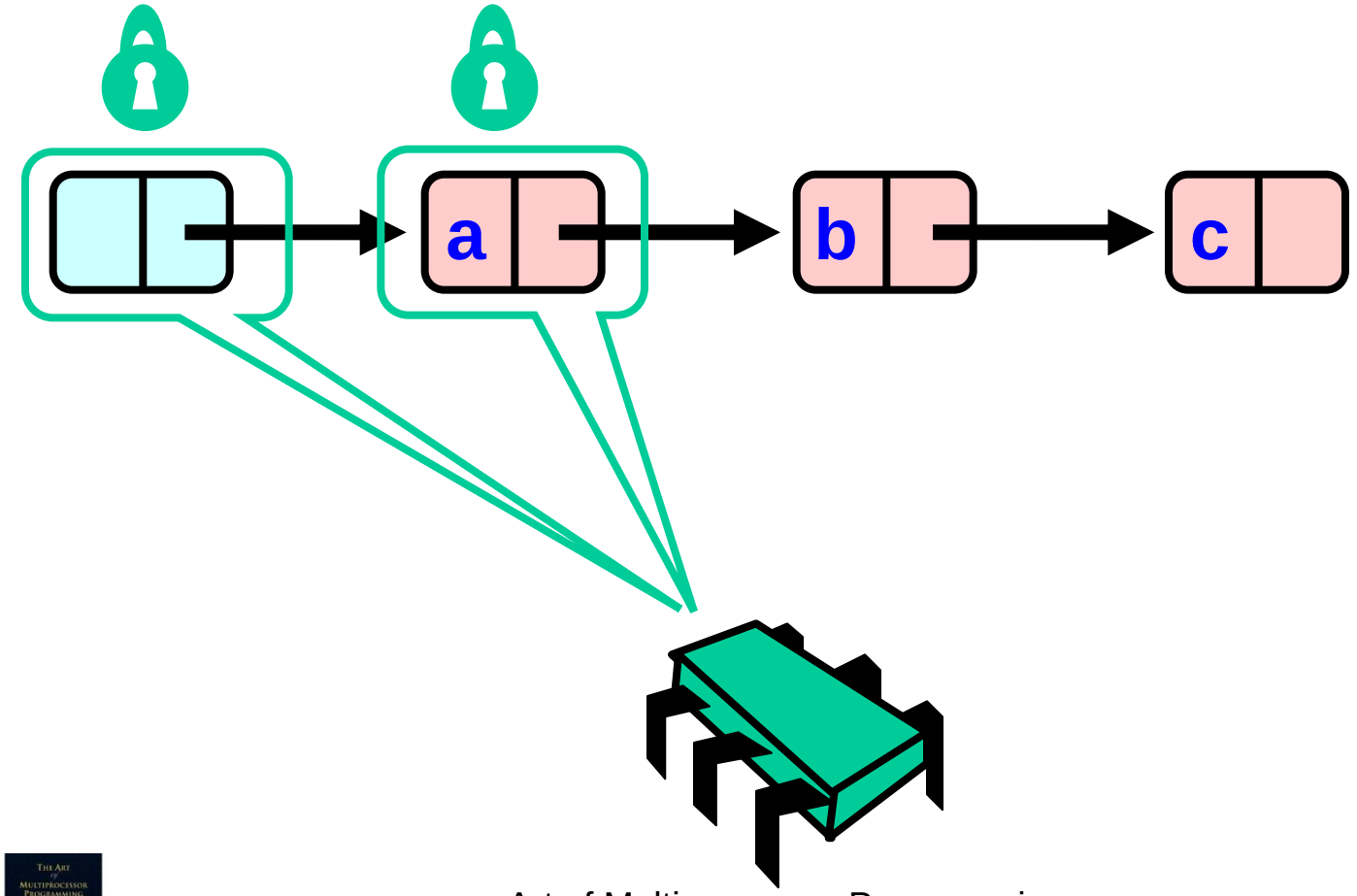


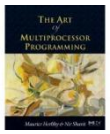
Hand-over-Hand locking



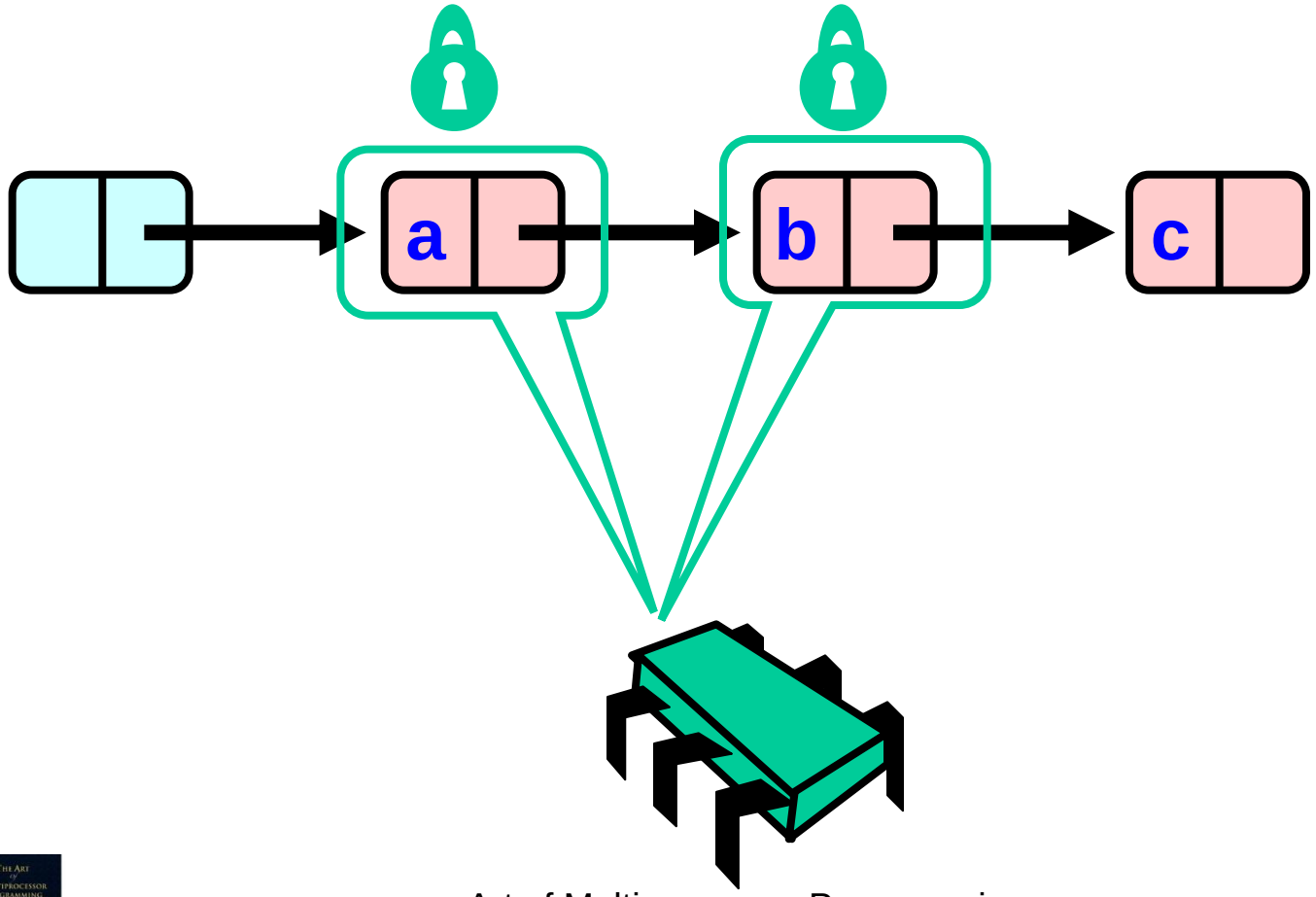


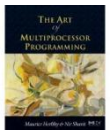
Hand-over-Hand locking



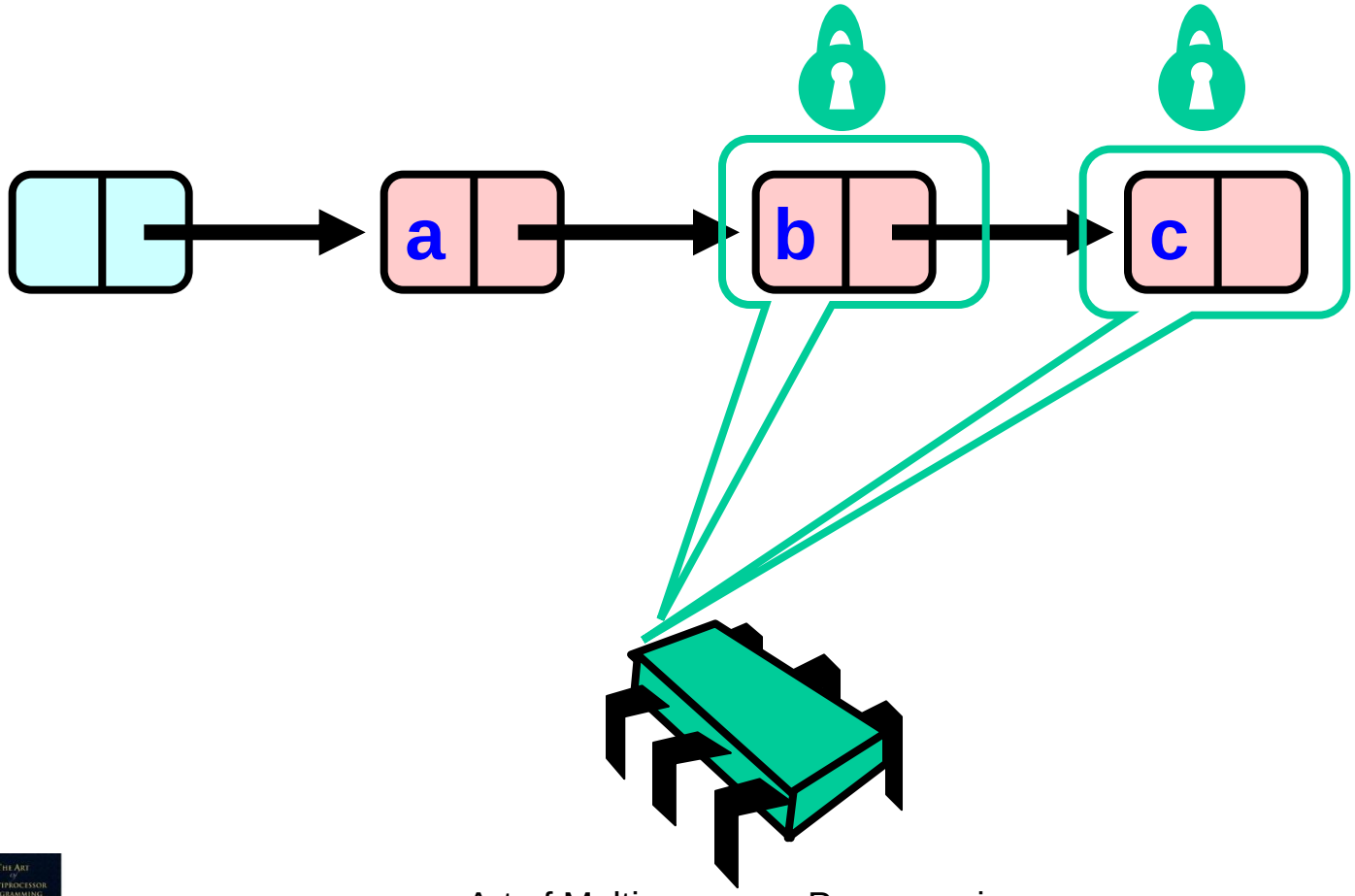


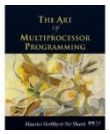
Hand-over-Hand locking



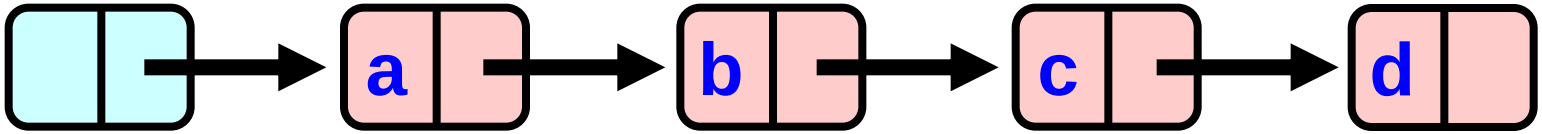


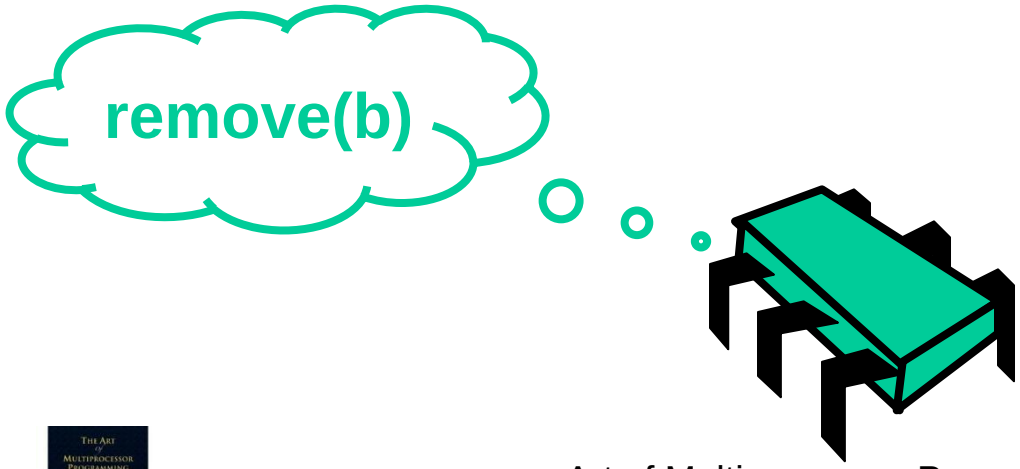
Hand-over-Hand locking

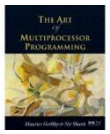




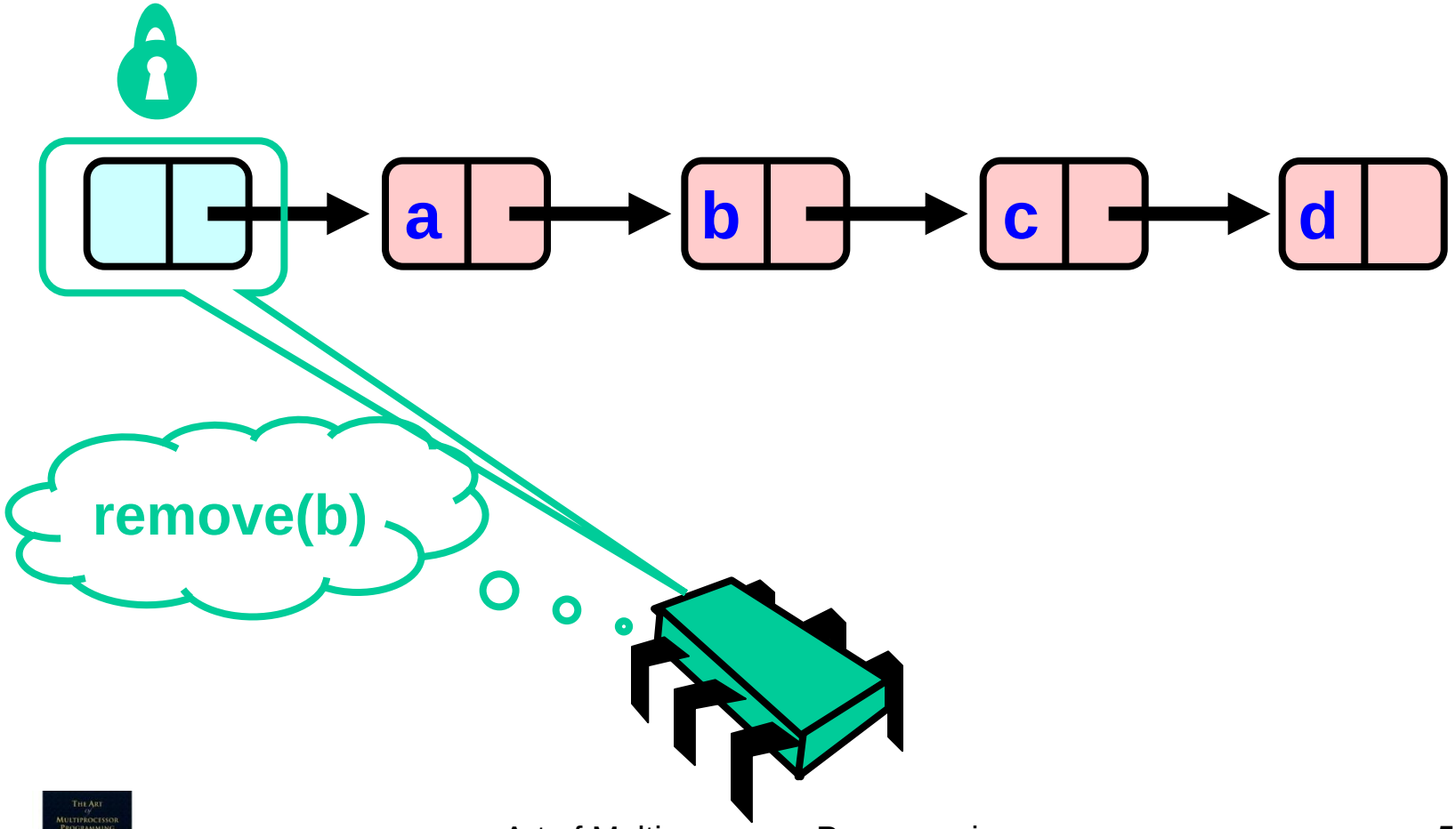
Removing a Node

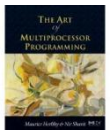




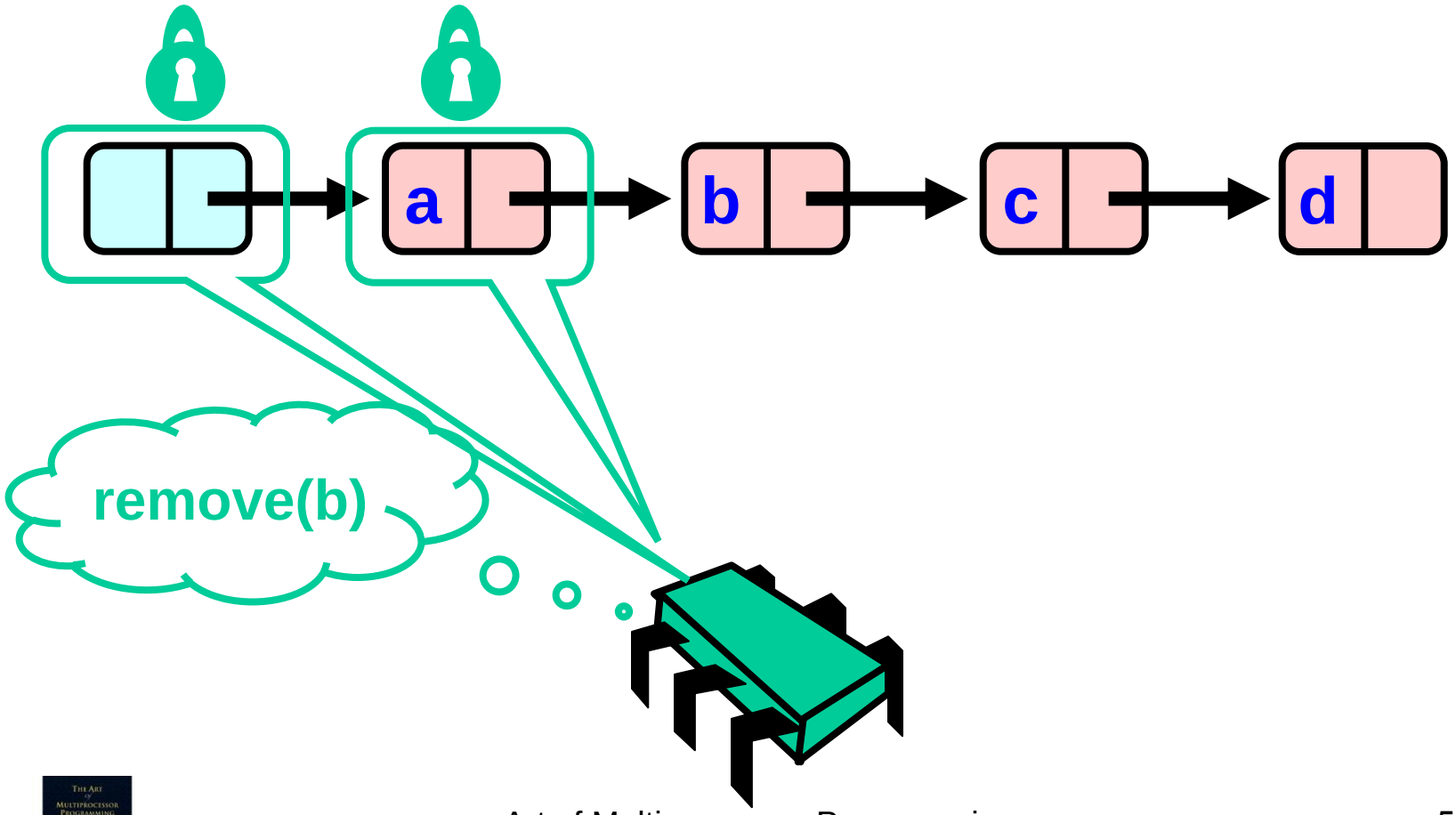


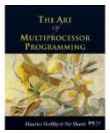
Removing a Node



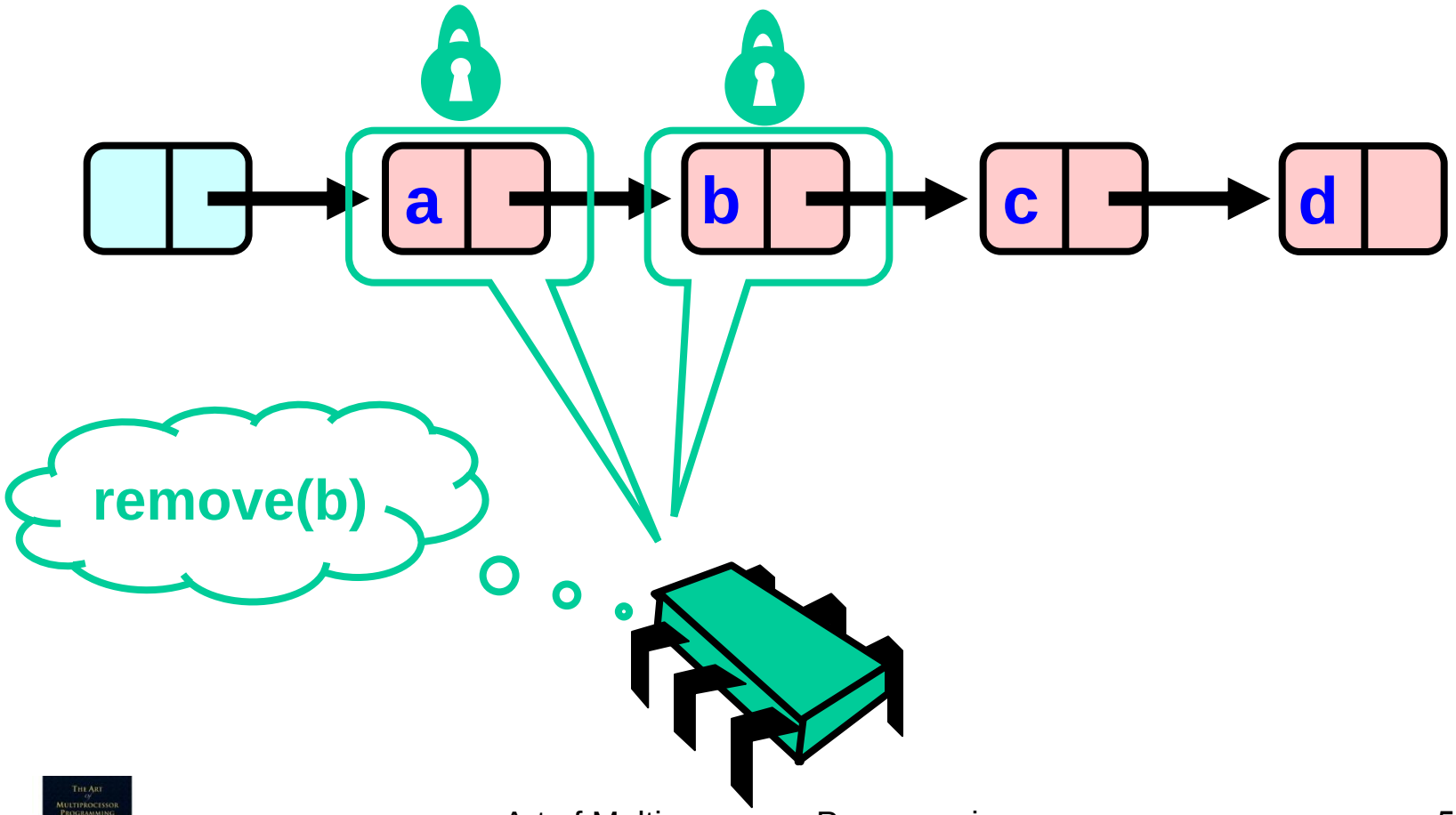


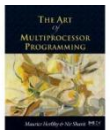
Removing a Node



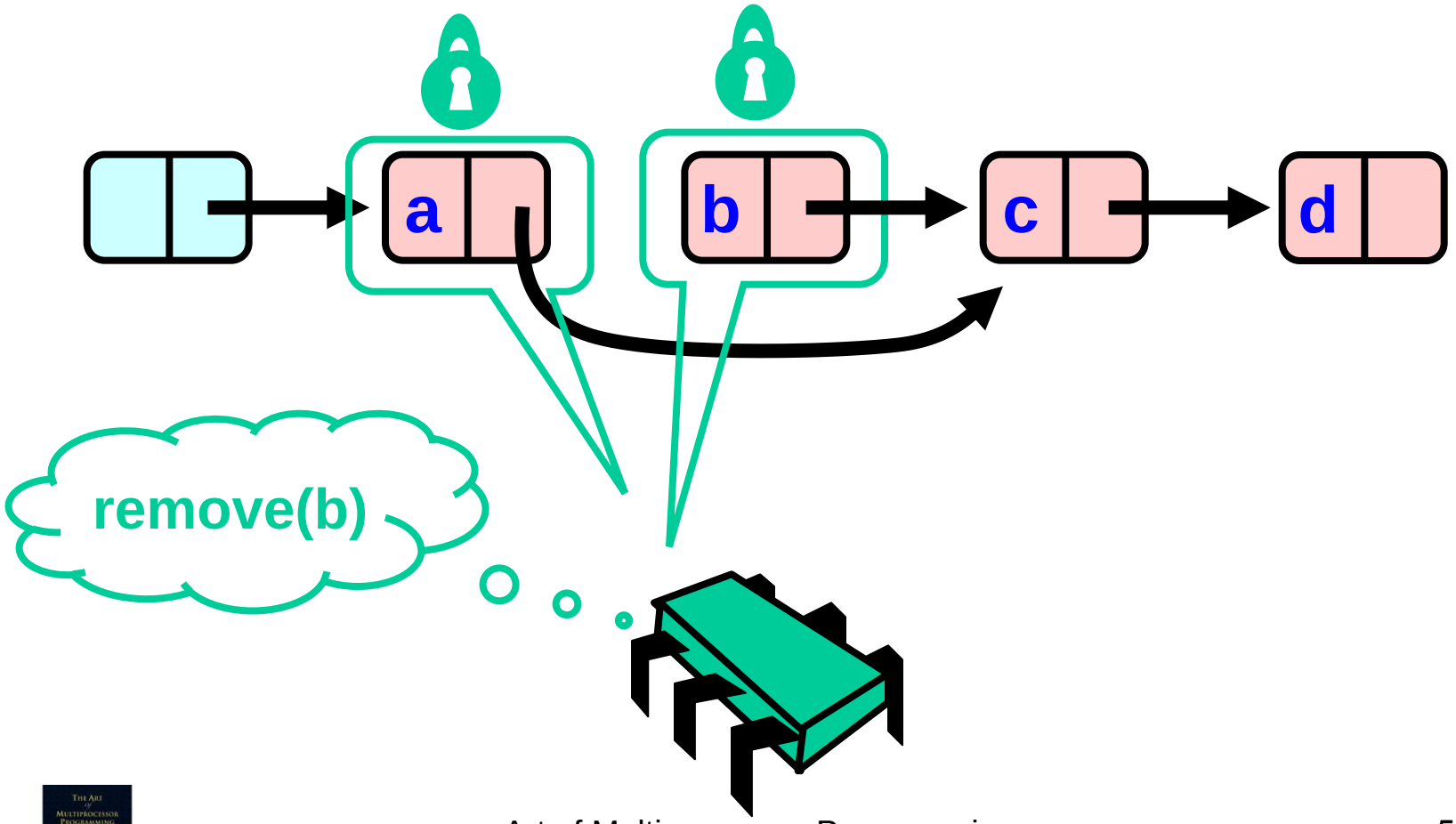


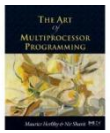
Removing a Node



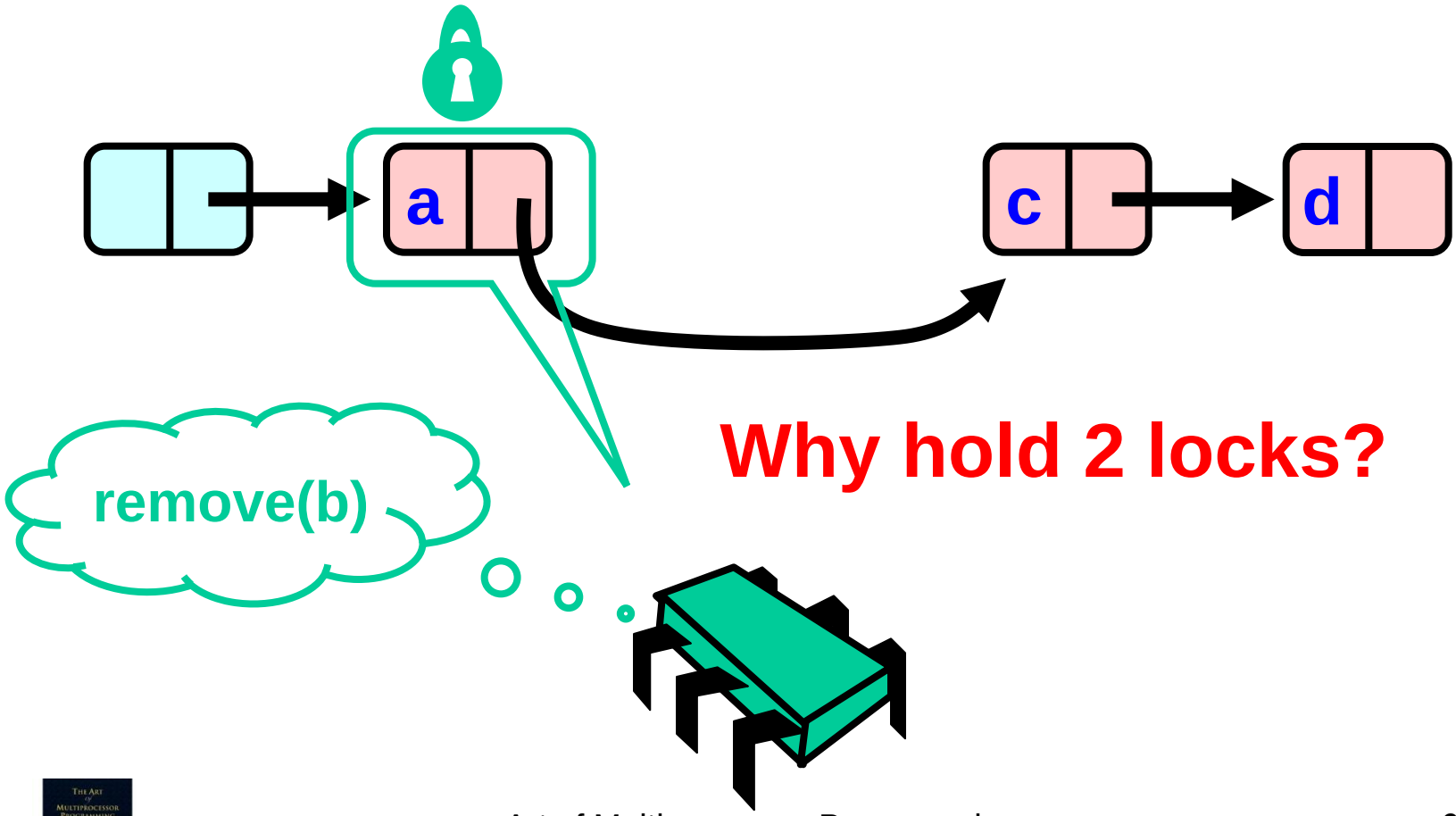


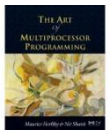
Removing a Node



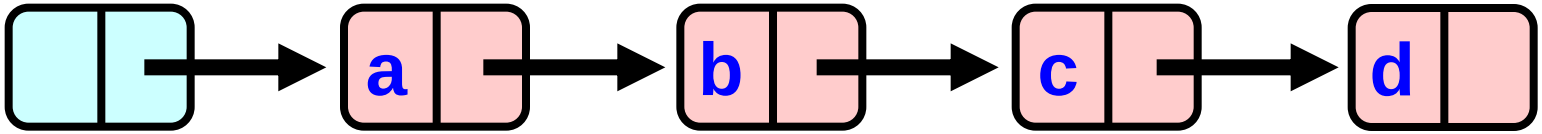


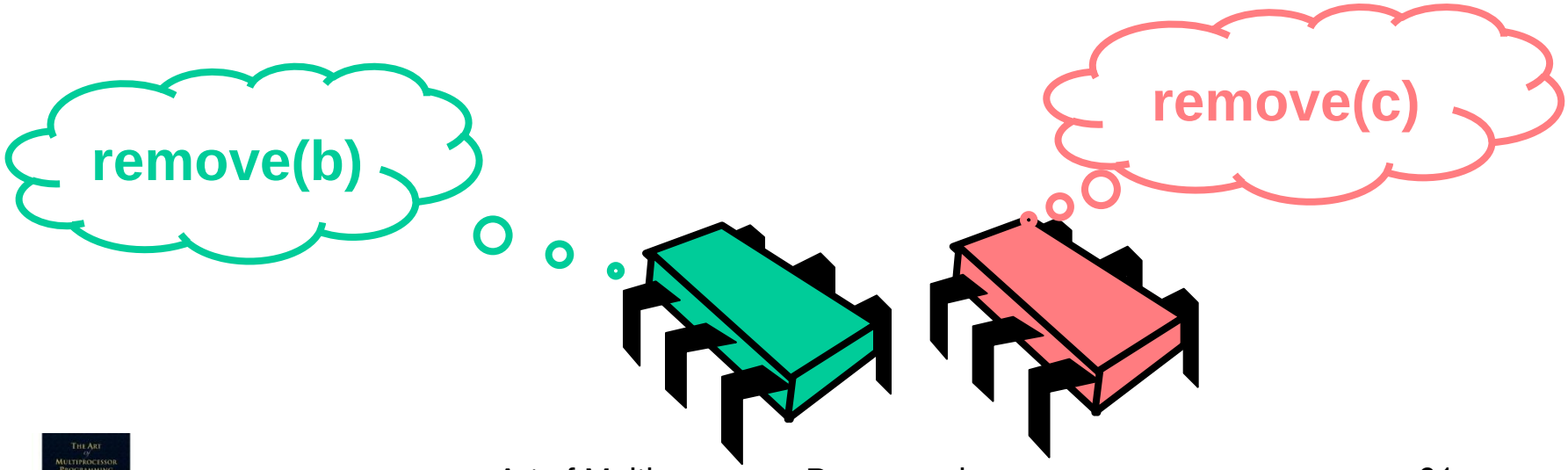
Removing a Node

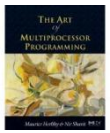




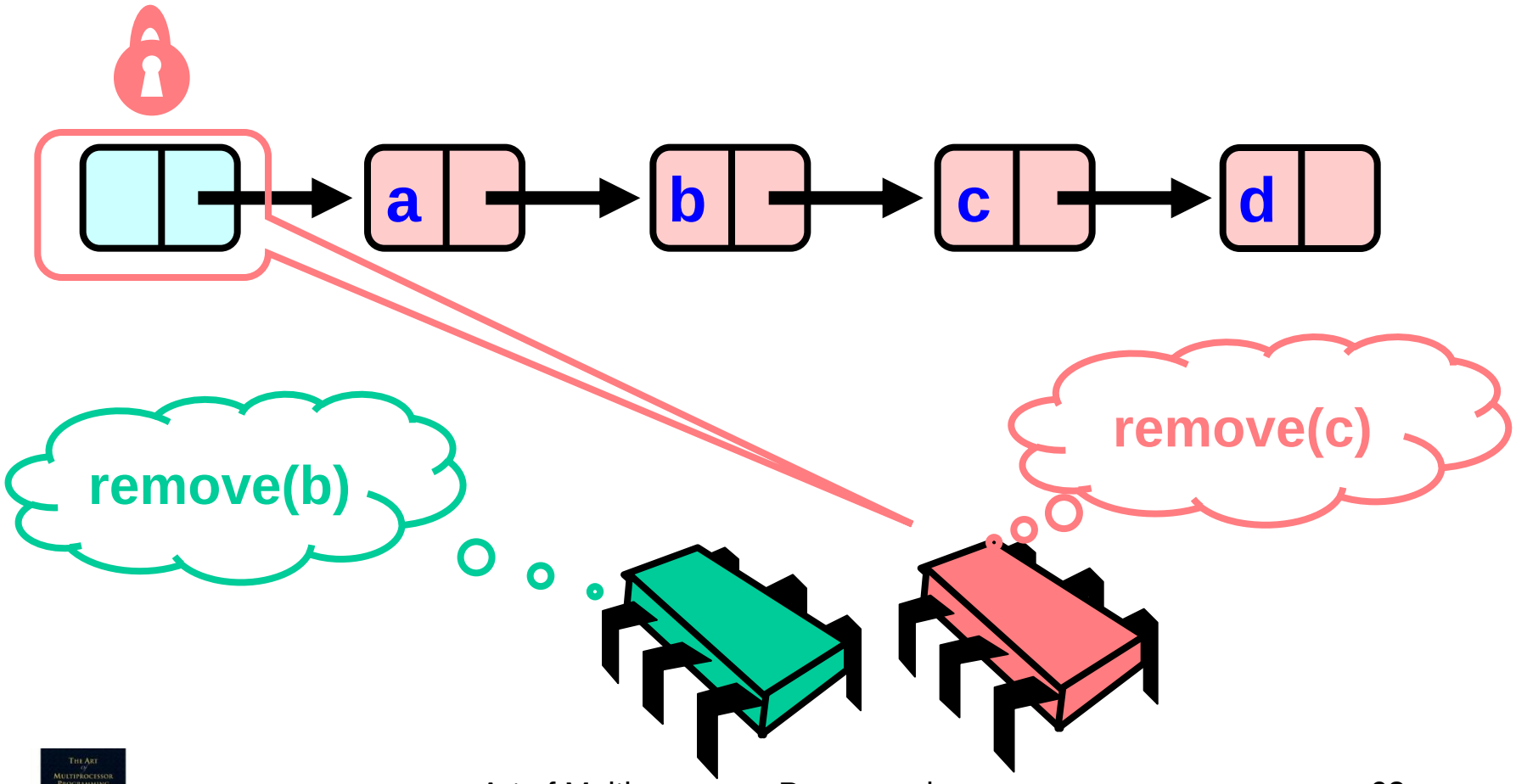
Concurrent Removes

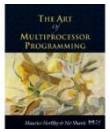




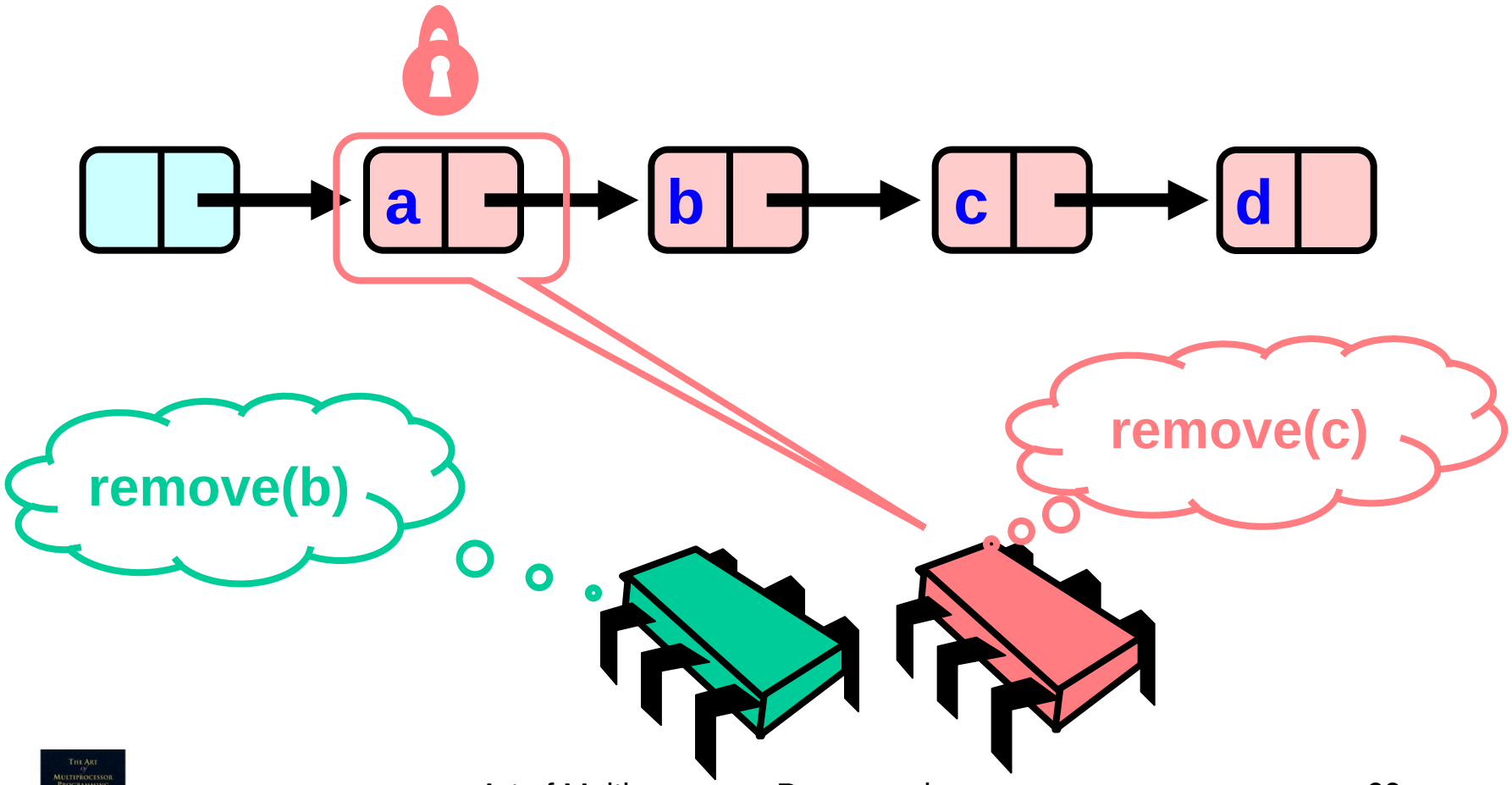


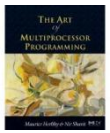
Concurrent Removes



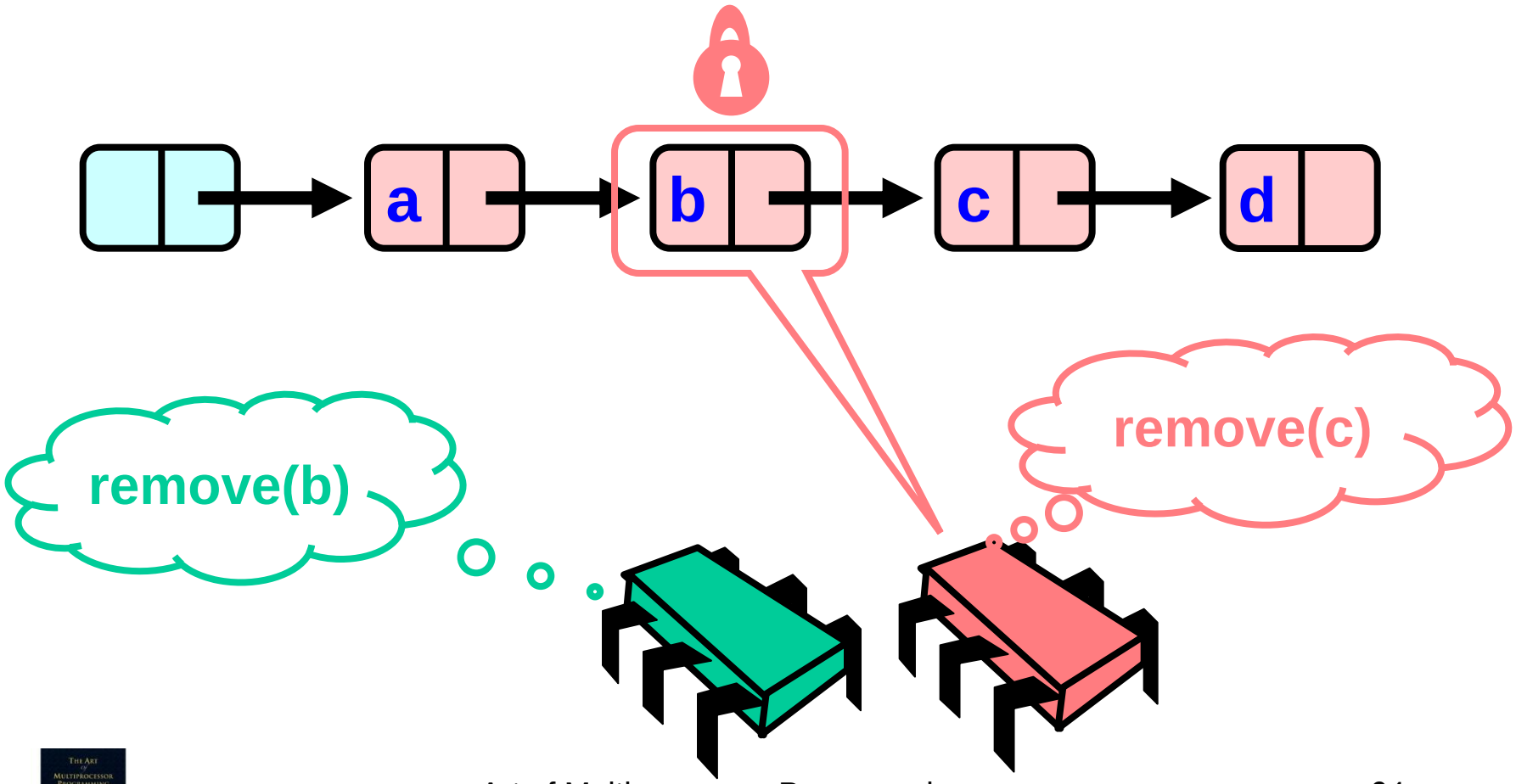


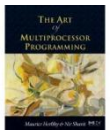
Concurrent Removes



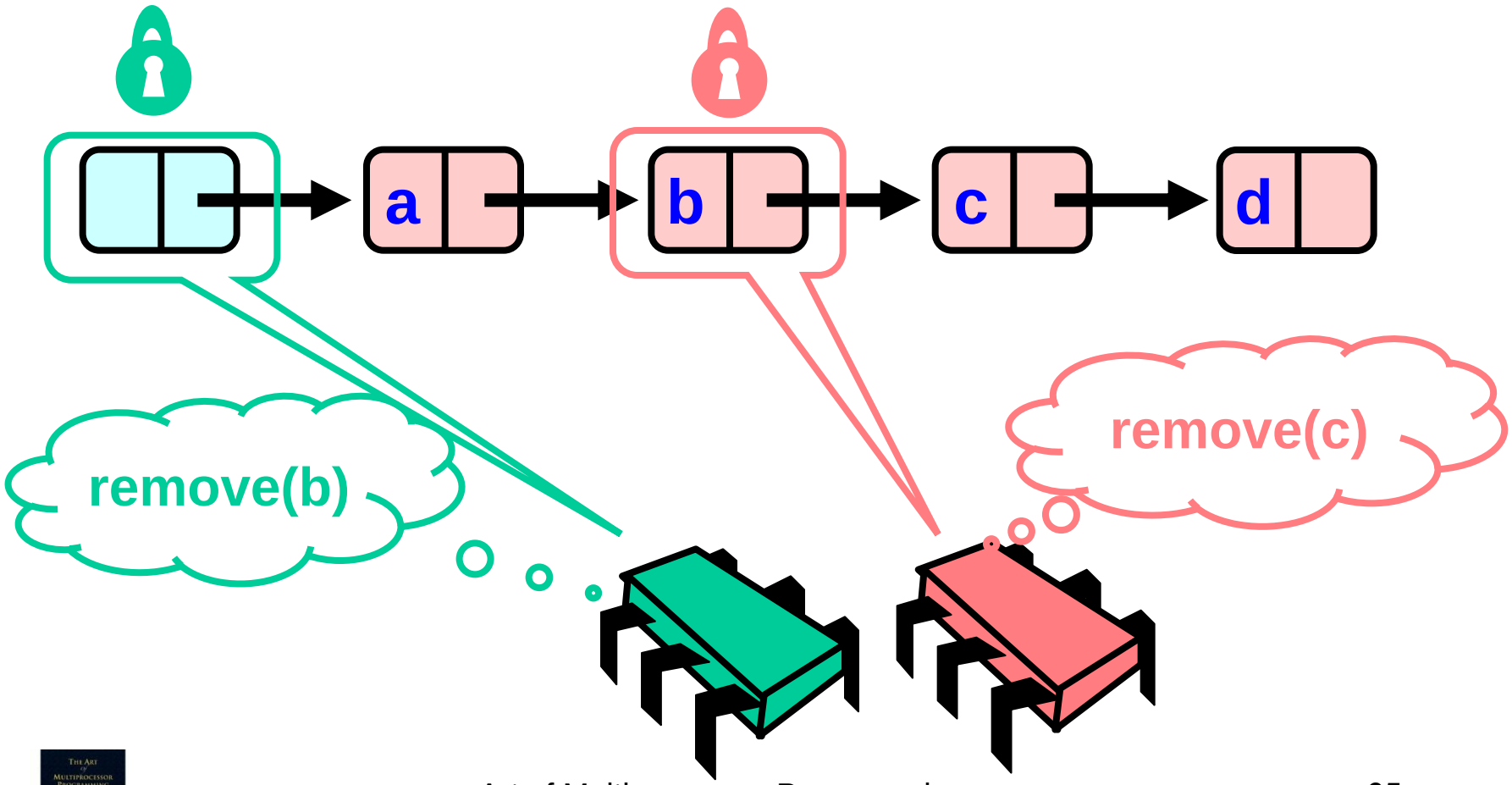


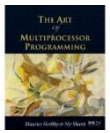
Concurrent Removes



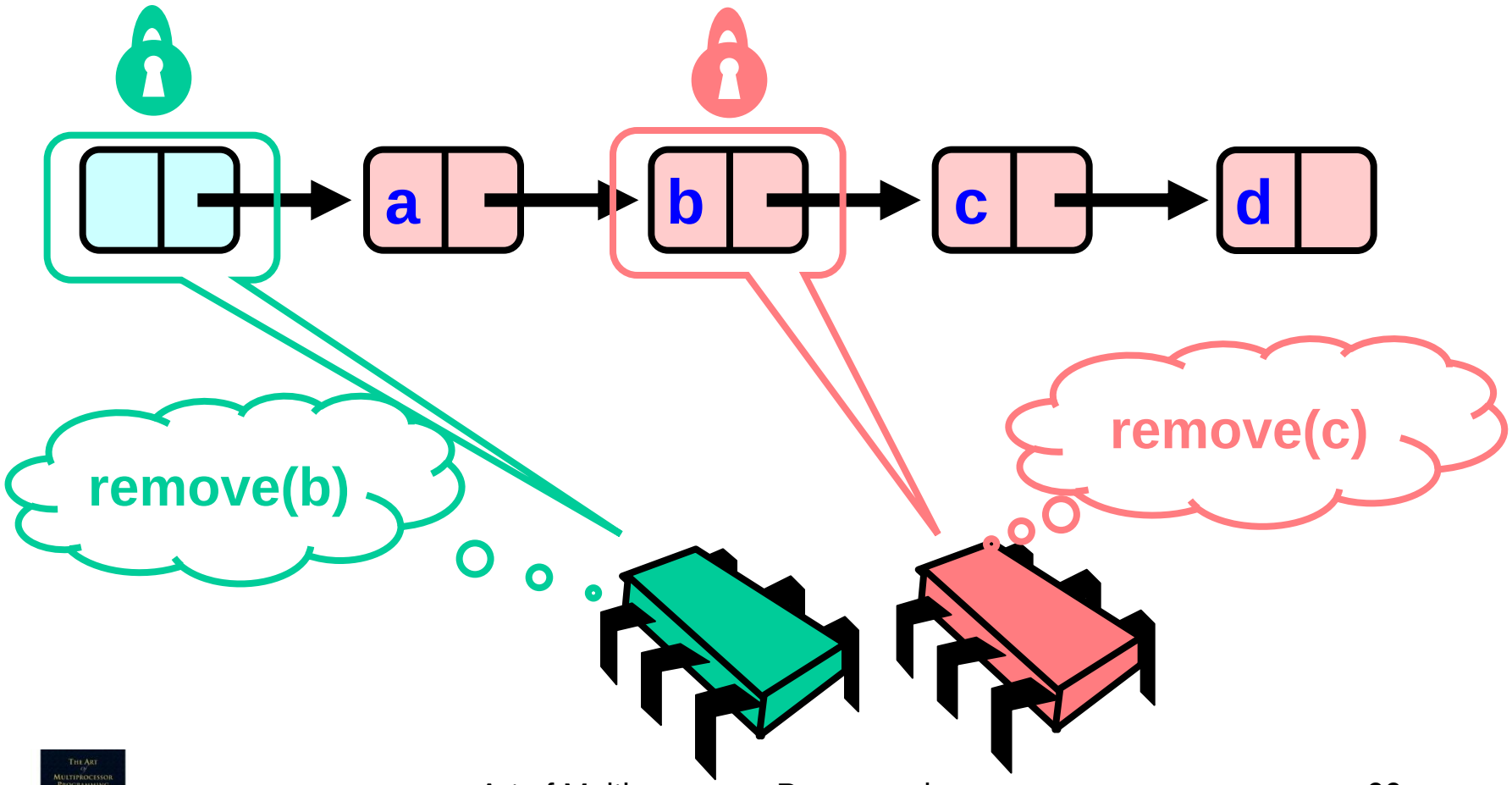


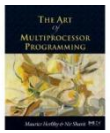
Concurrent Removes



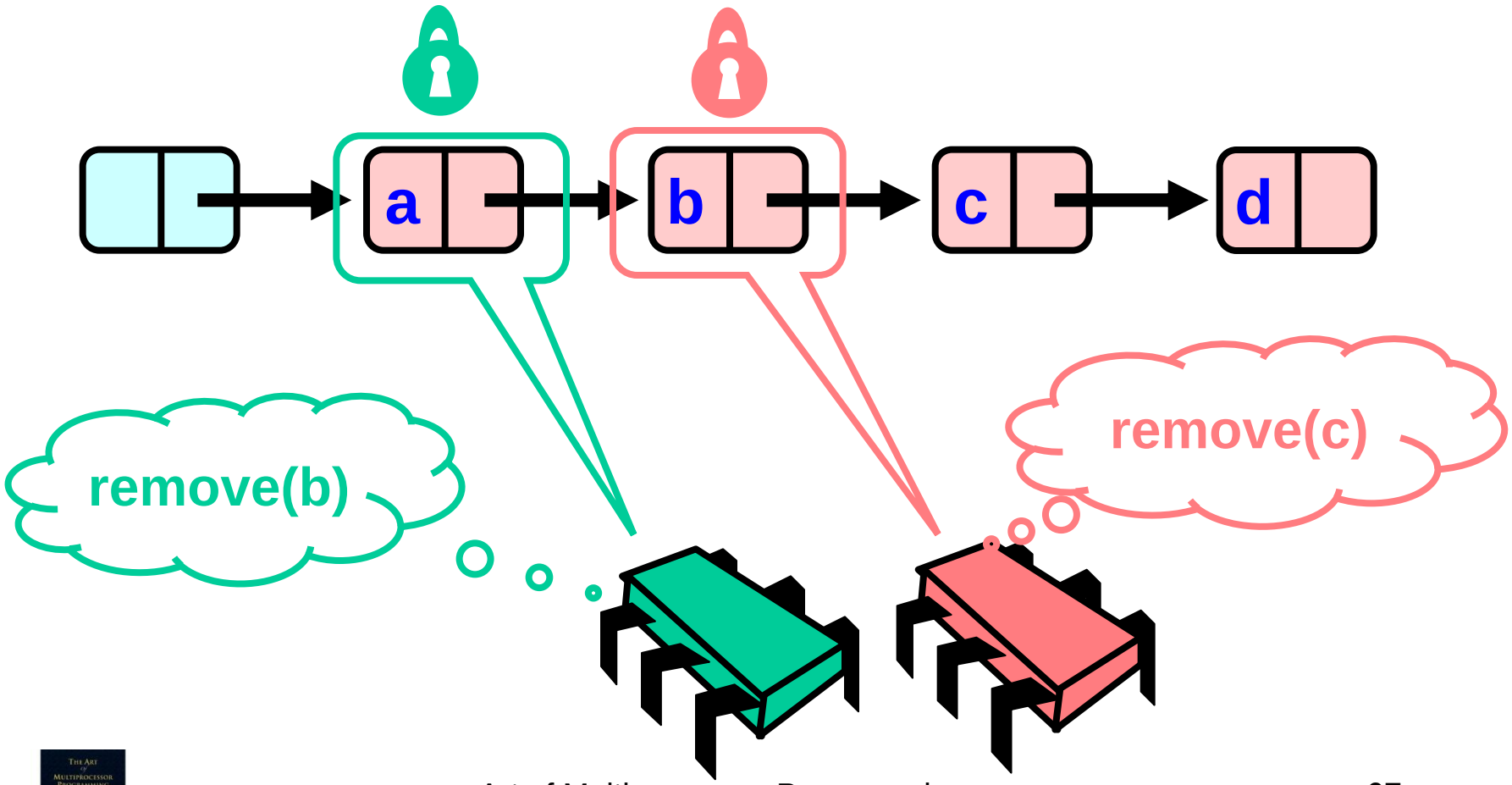


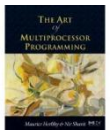
Concurrent Removes



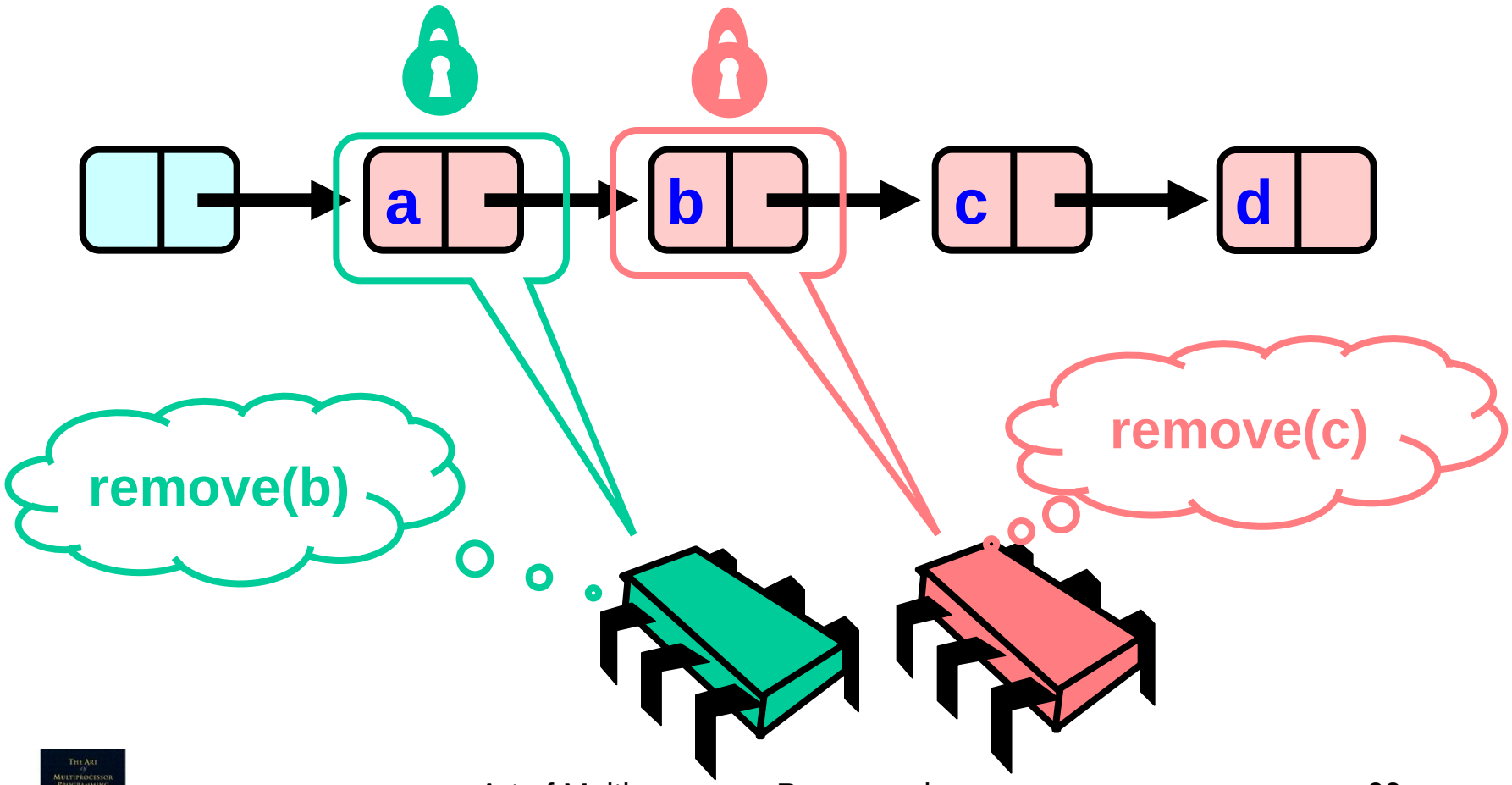


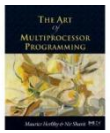
Concurrent Removes



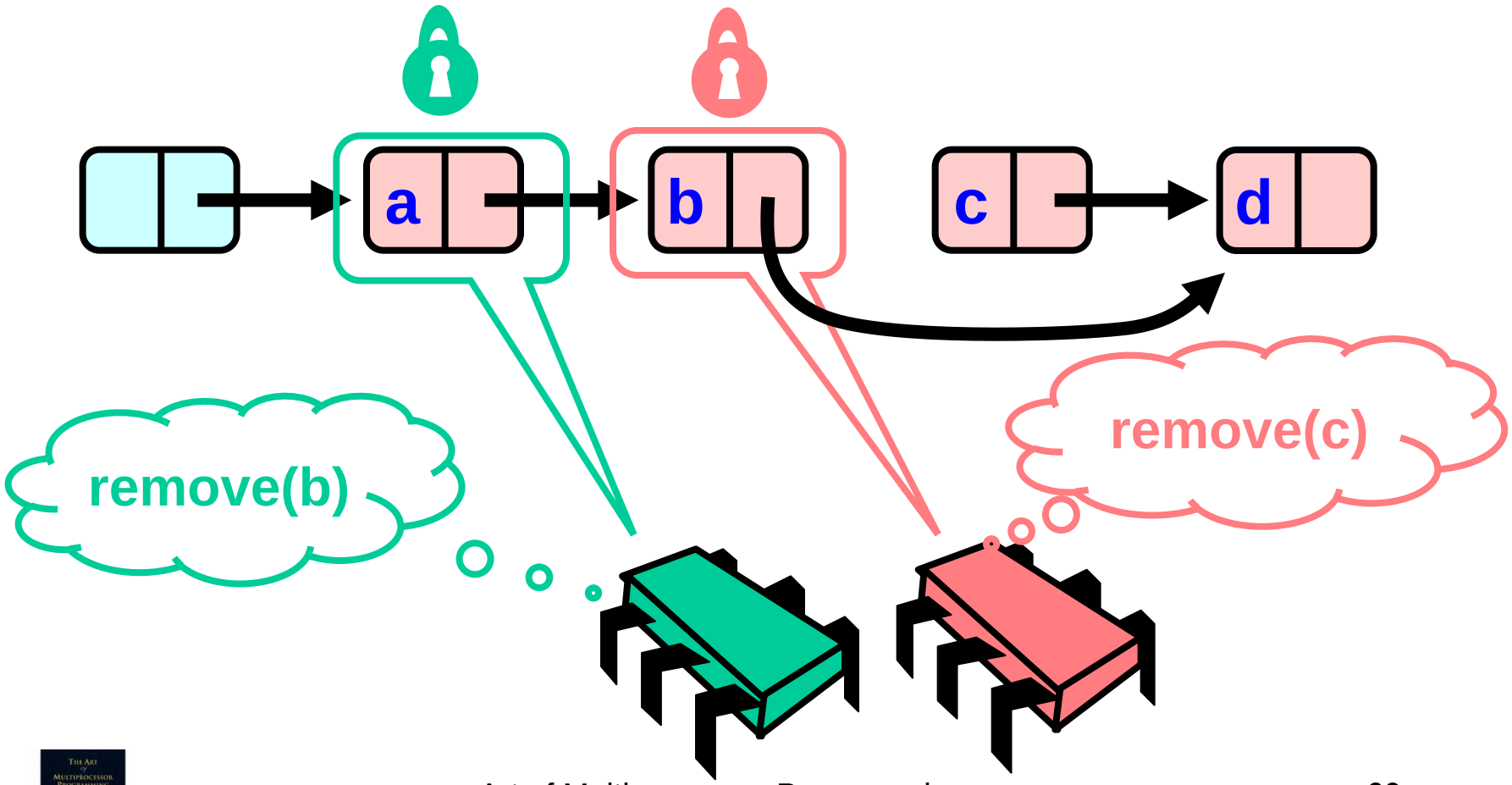


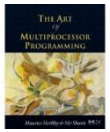
Concurrent Removes



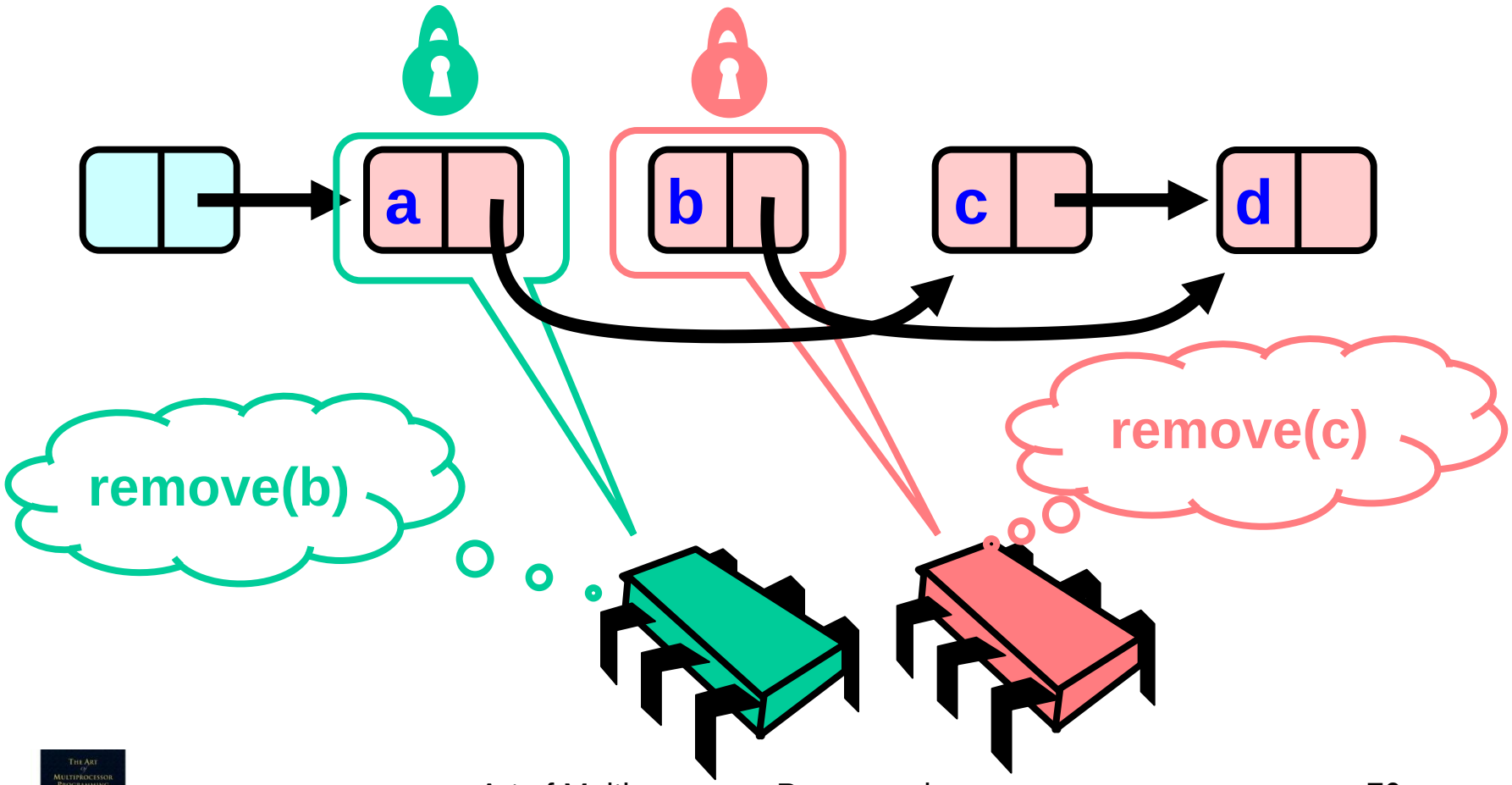


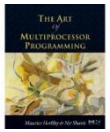
Concurrent Removes



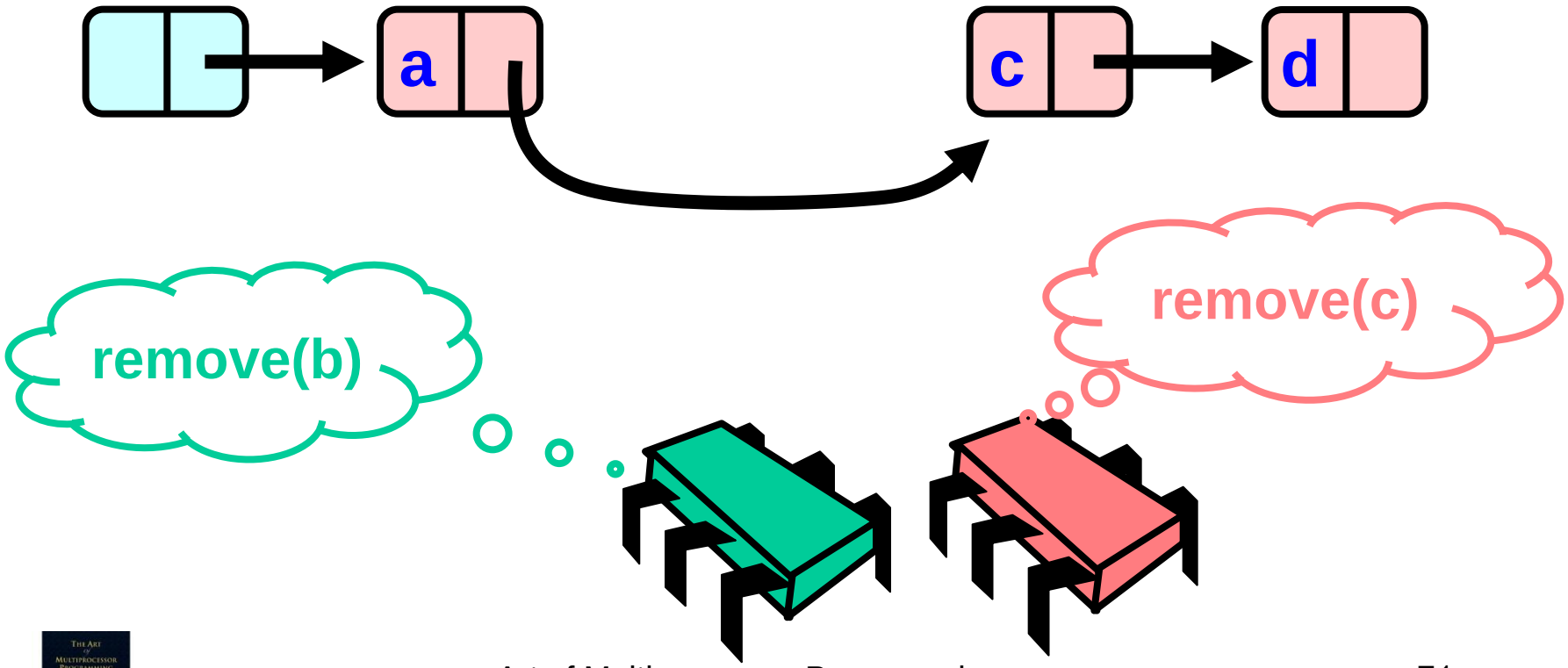


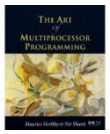
Concurrent Removes





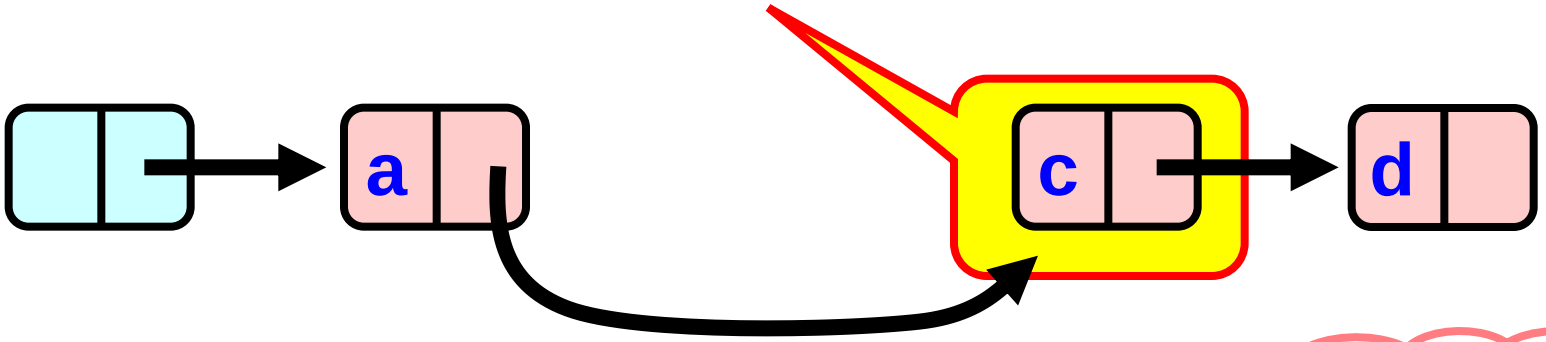
Uh, Oh

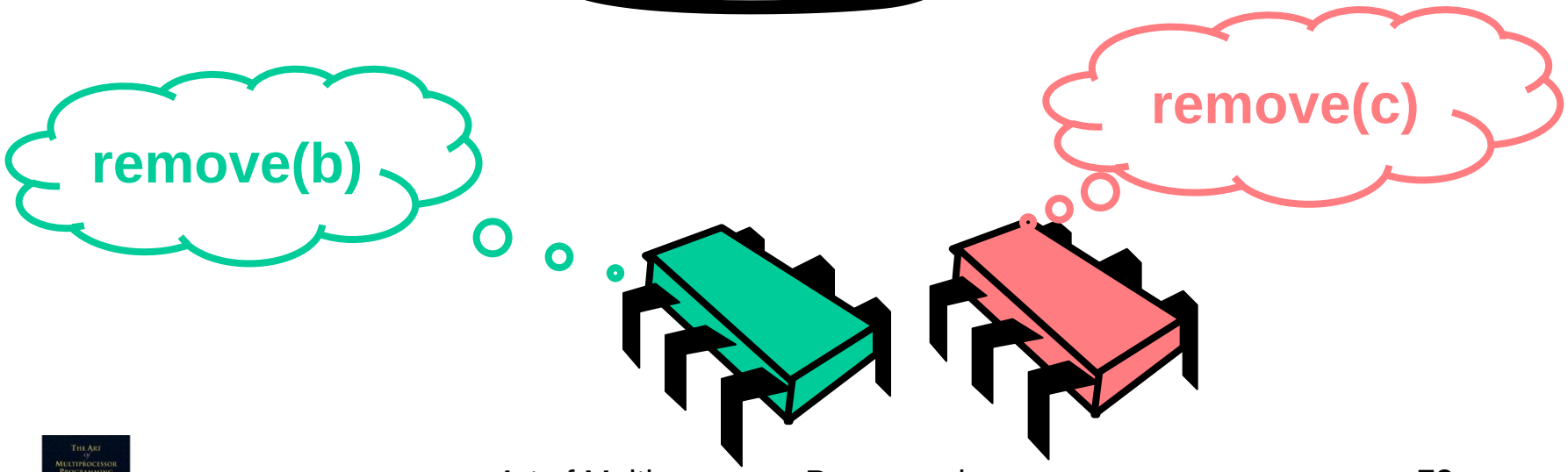


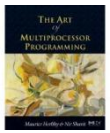


Uh, Oh

Bad news, c not removed

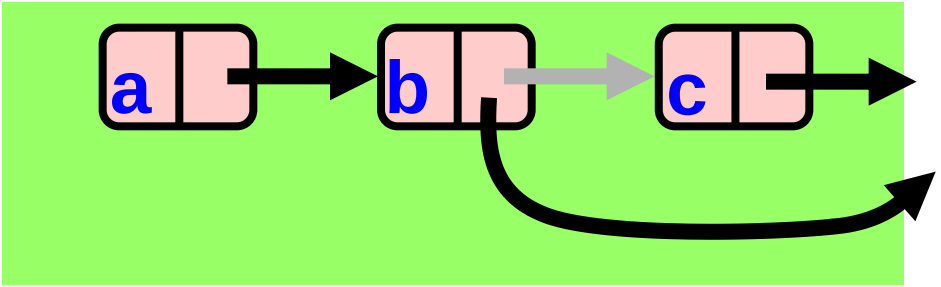




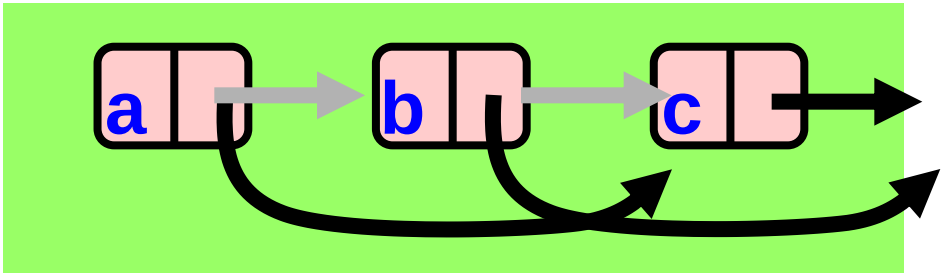


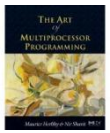
Problem

- To delete node c
  - Swing node b's next field to d



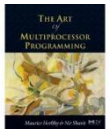
- Problem is,
  - Someone deleting b concurrently could direct a pointer to c



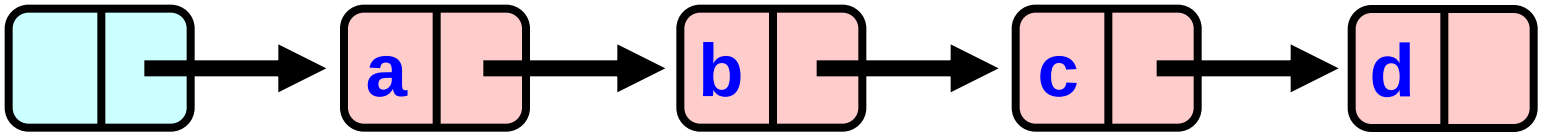


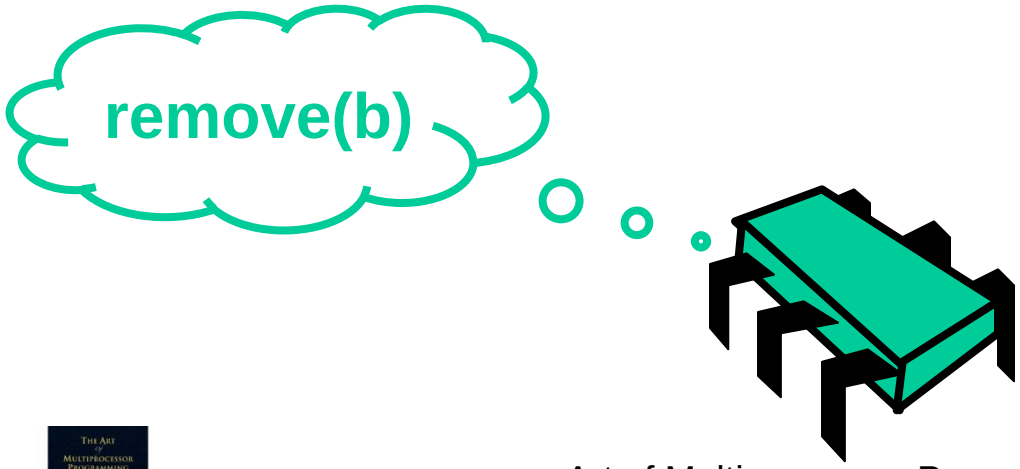
Insight

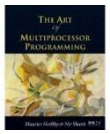
- If a node is locked
  - No one can delete node's *successor*
- If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works



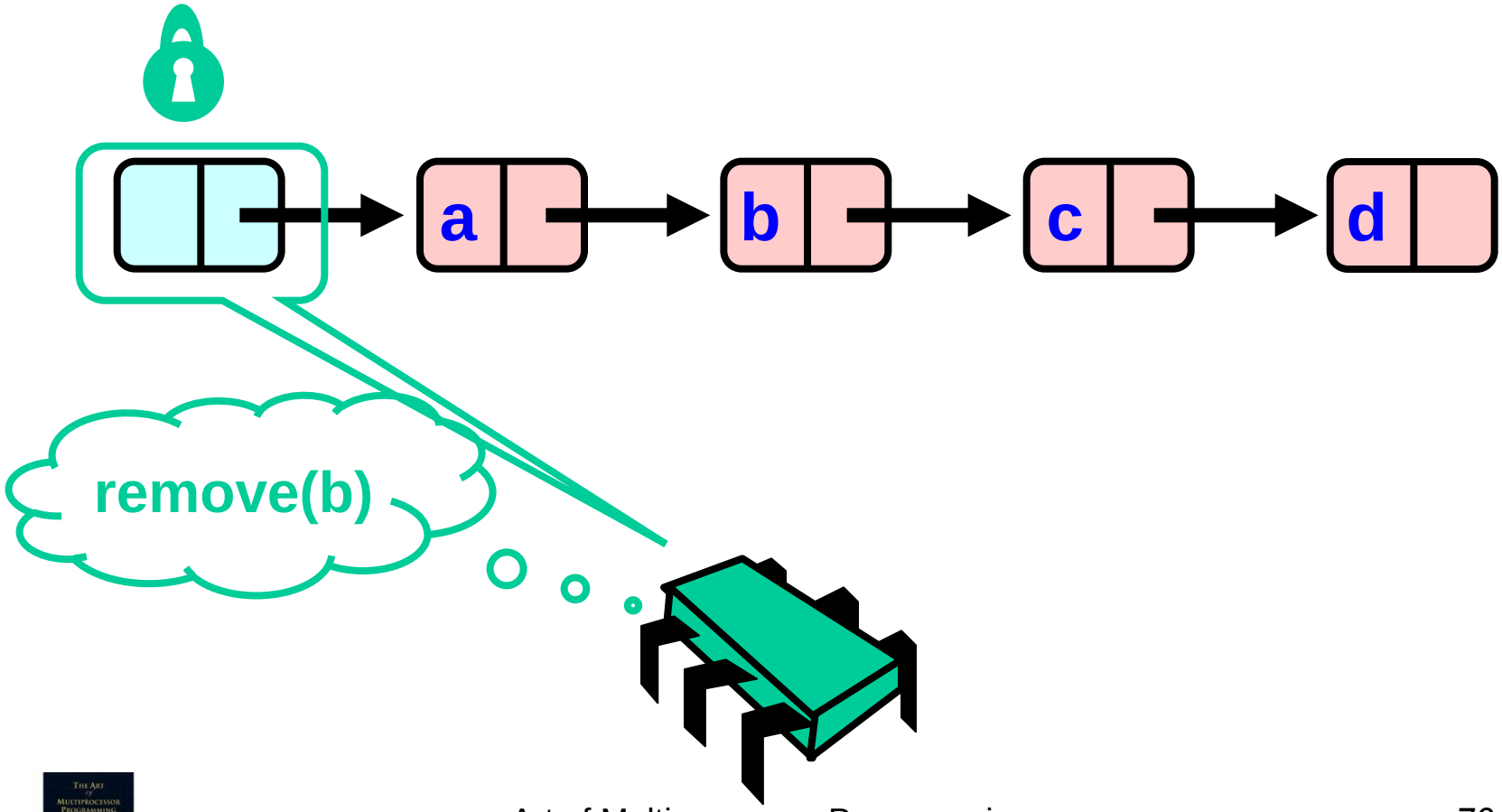
Hand-Over-Hand Again

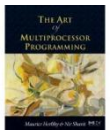




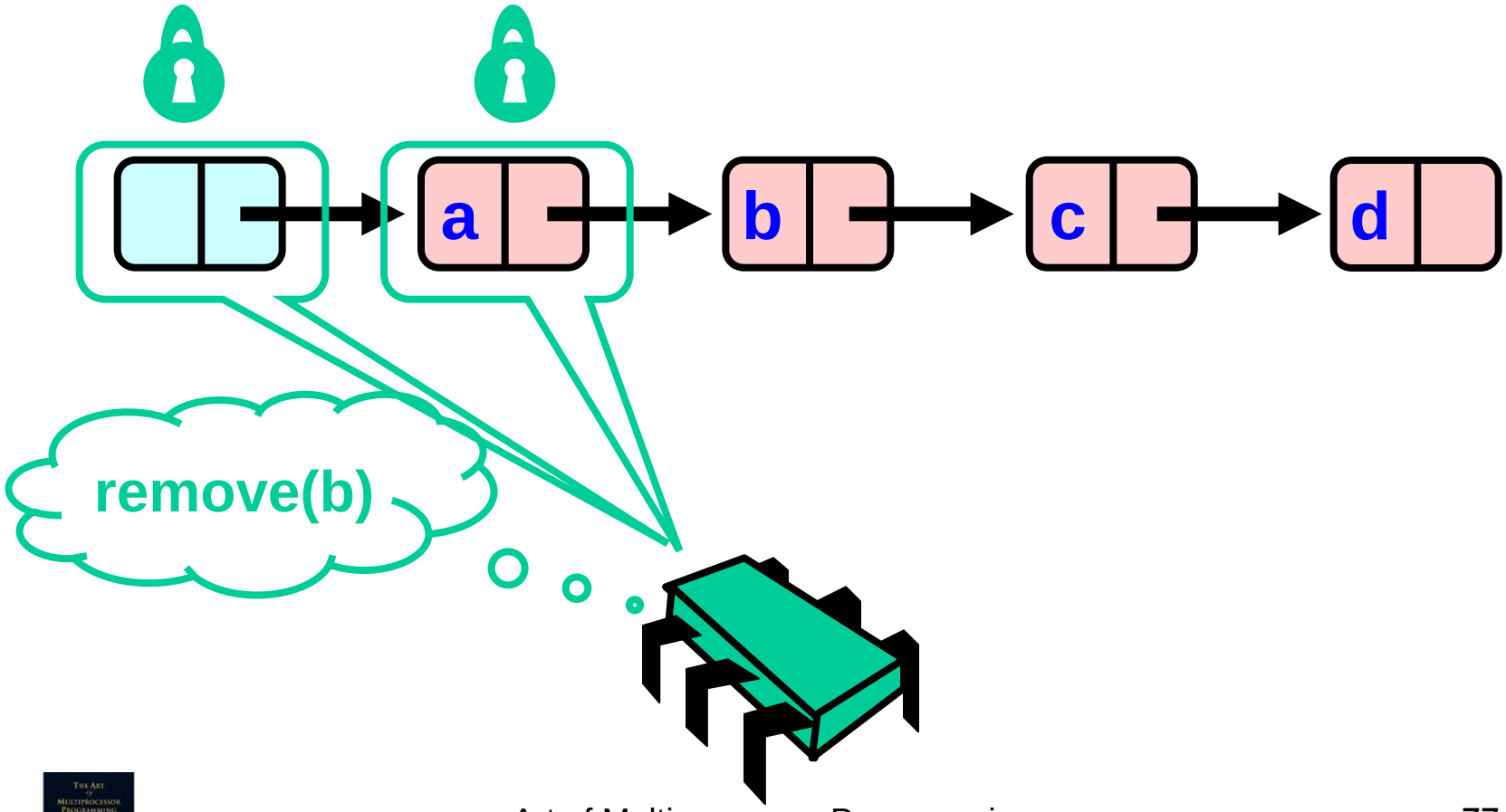


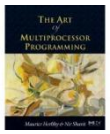
Hand-Over-Hand Again



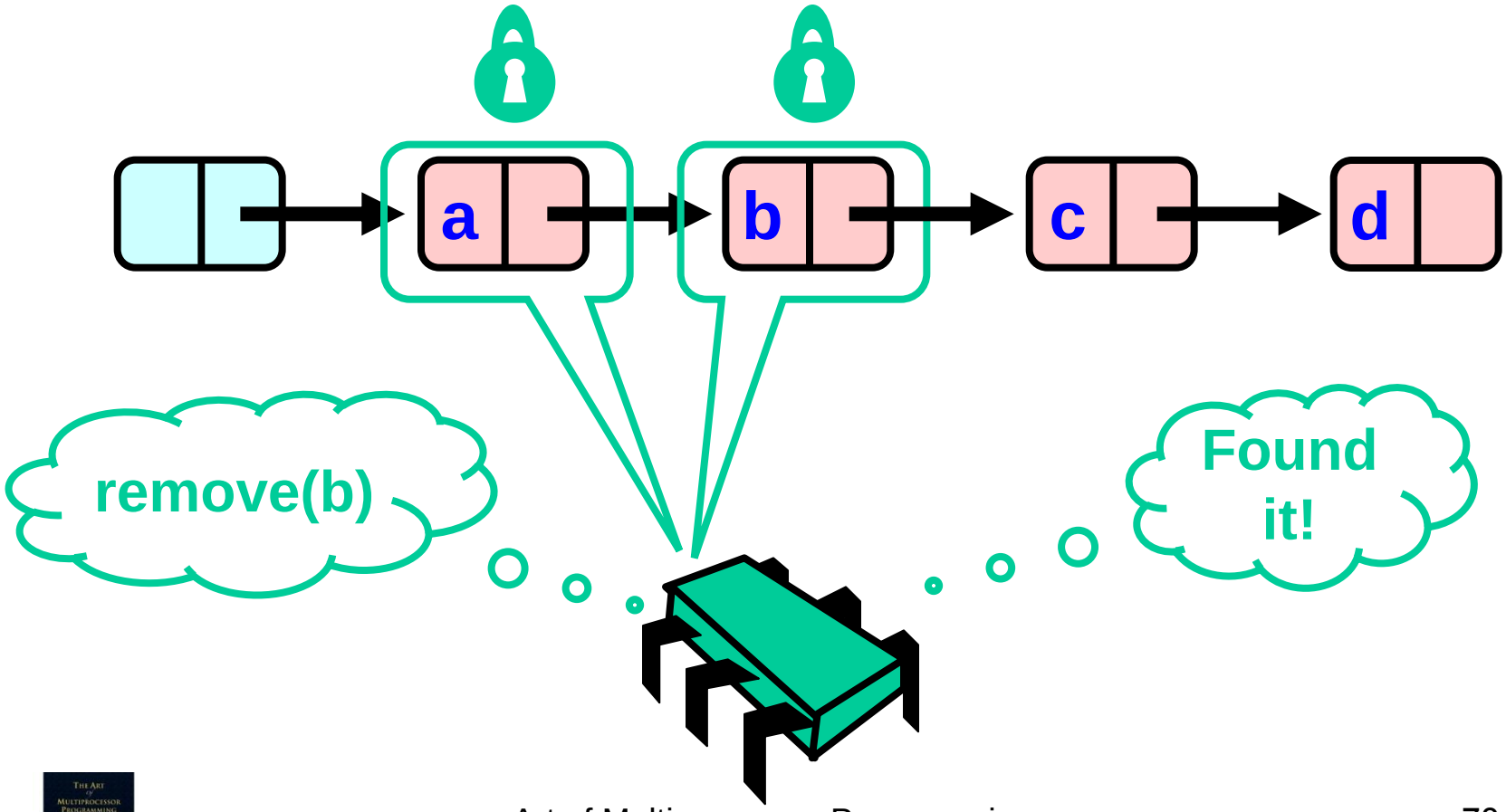


Hand-Over-Hand Again

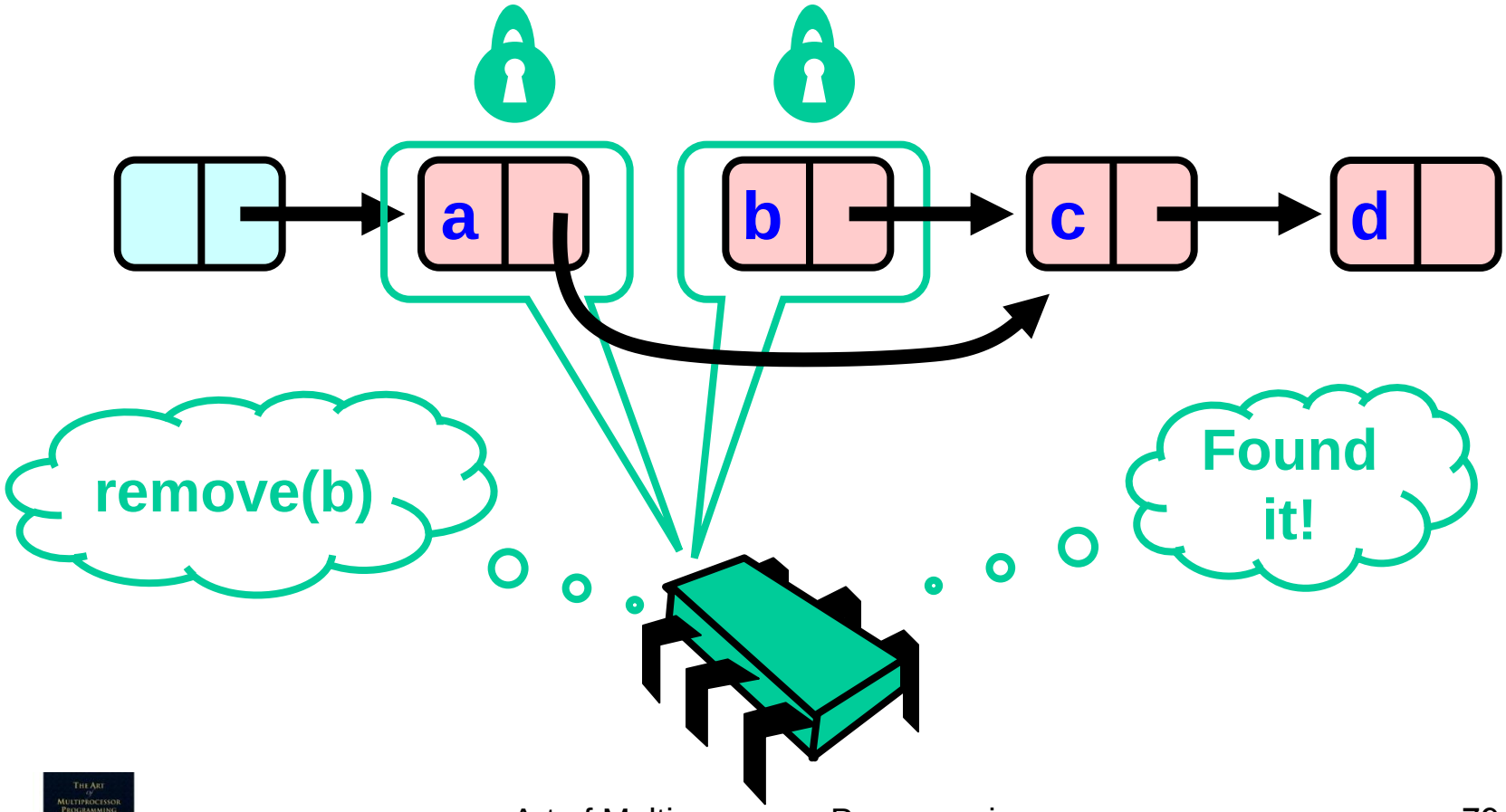




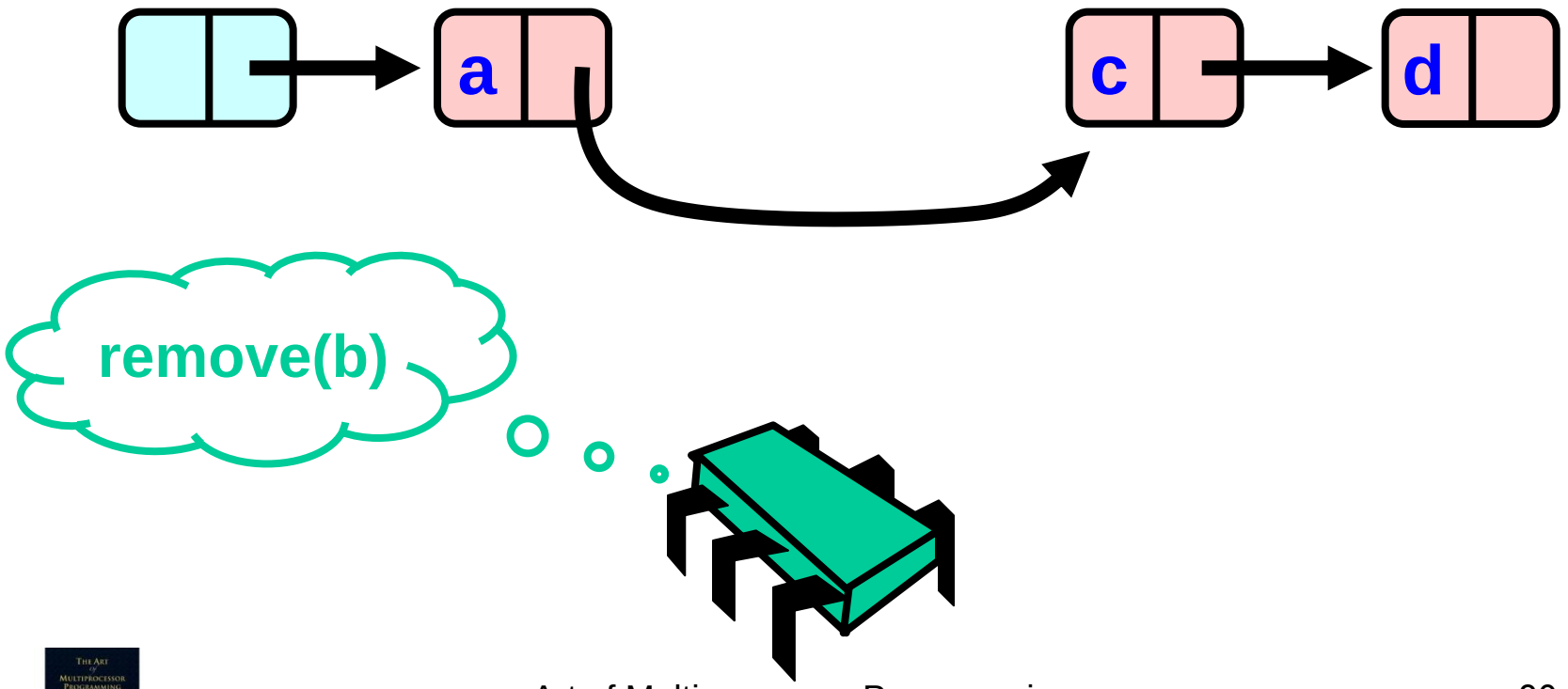
Hand-Over-Hand Again

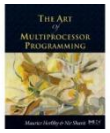


Hand-Over-Hand Again

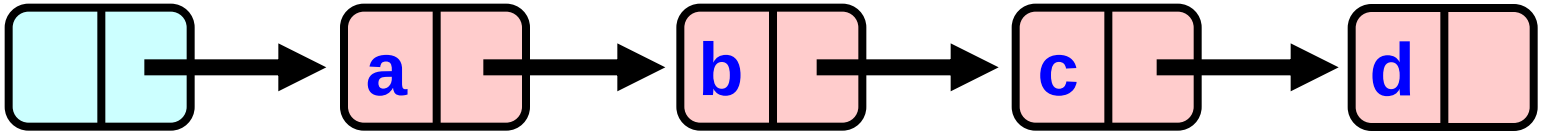


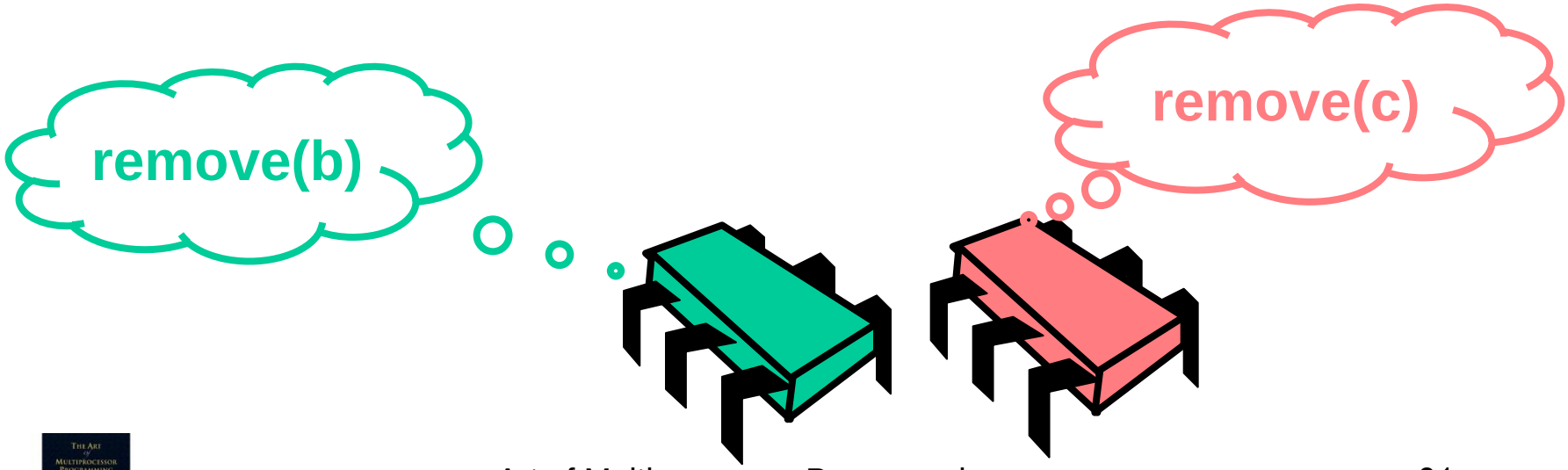
Hand-Over-Hand Again

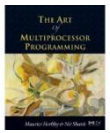




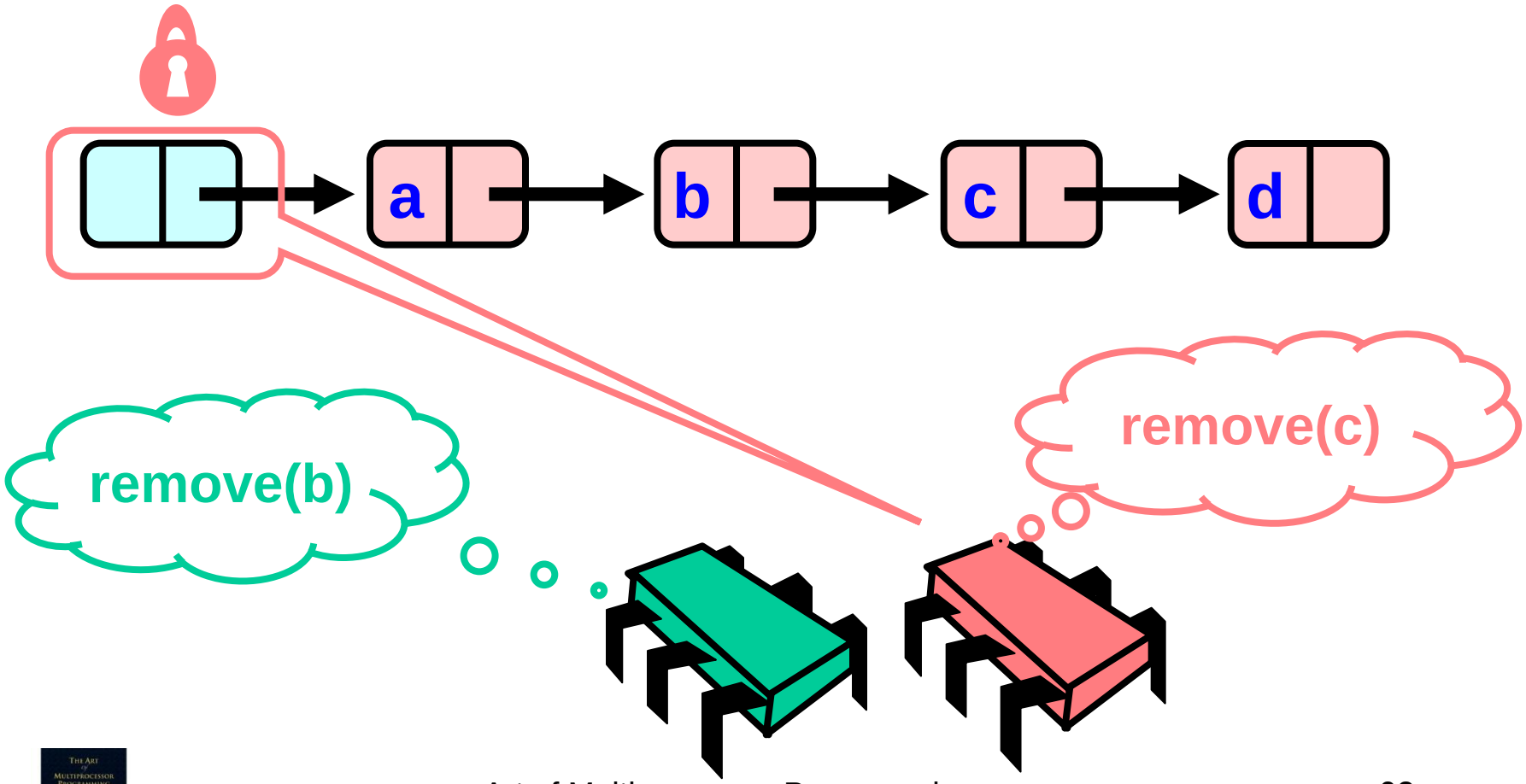
Removing a Node

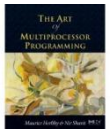




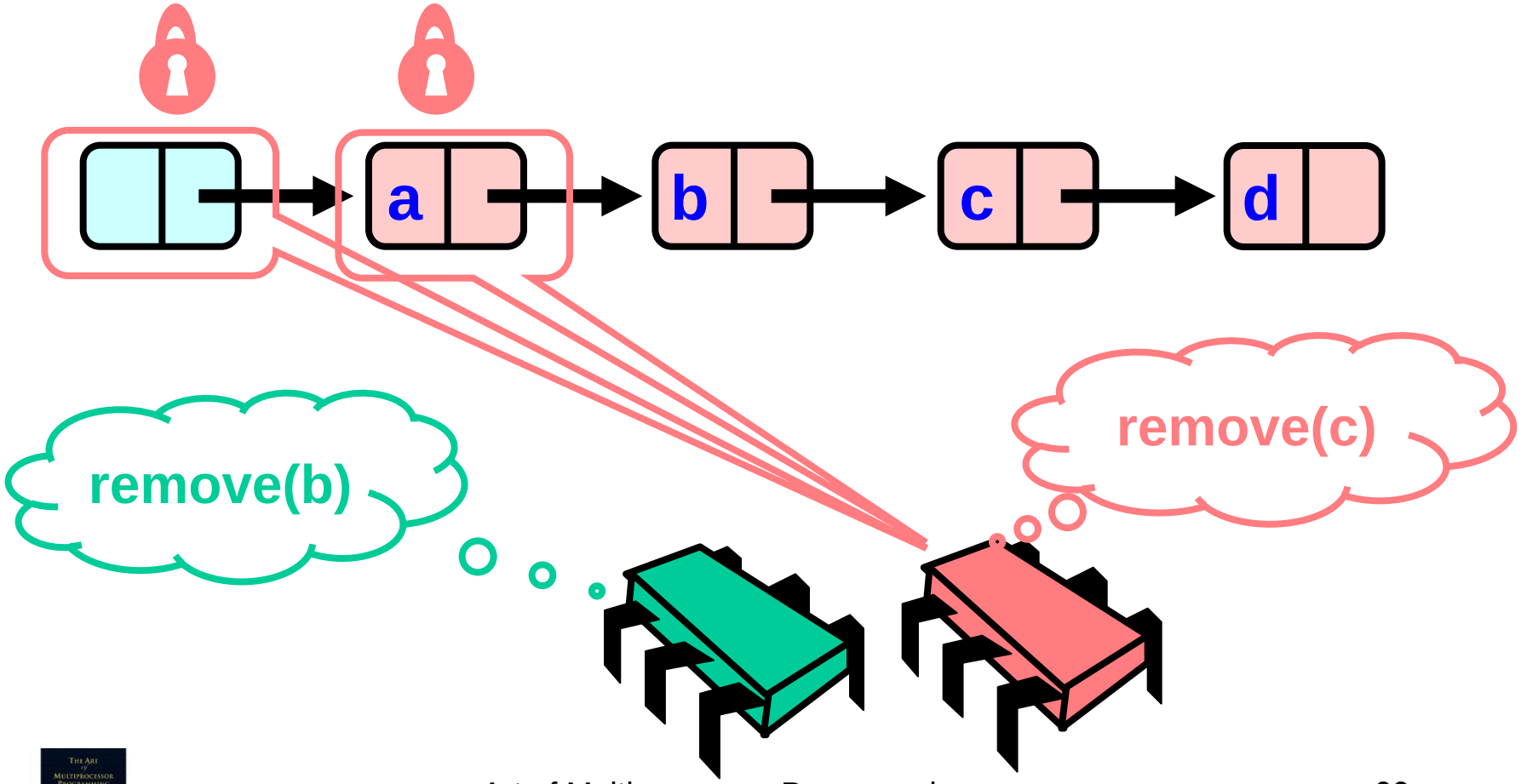


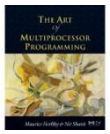
Removing a Node



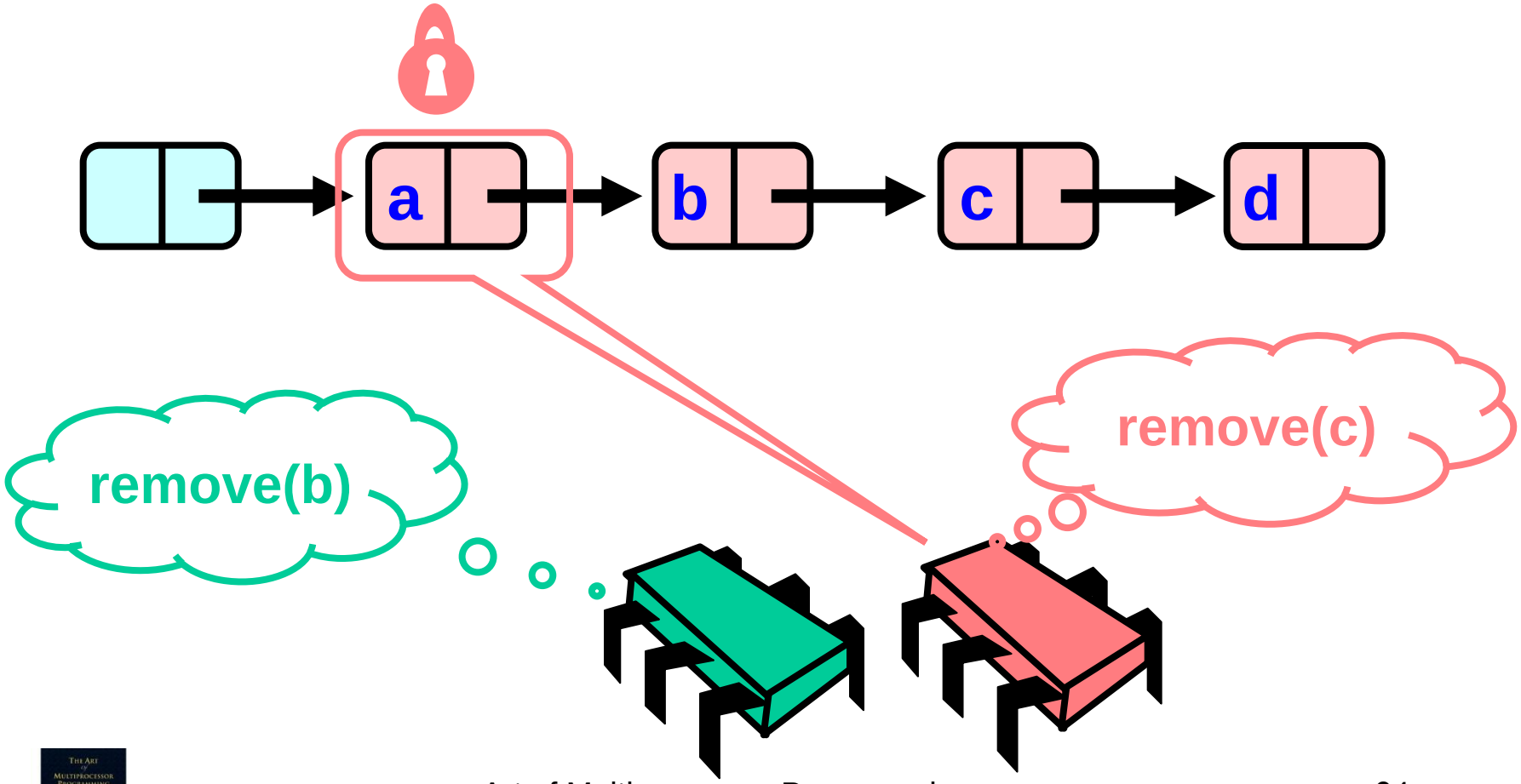


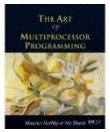
Removing a Node



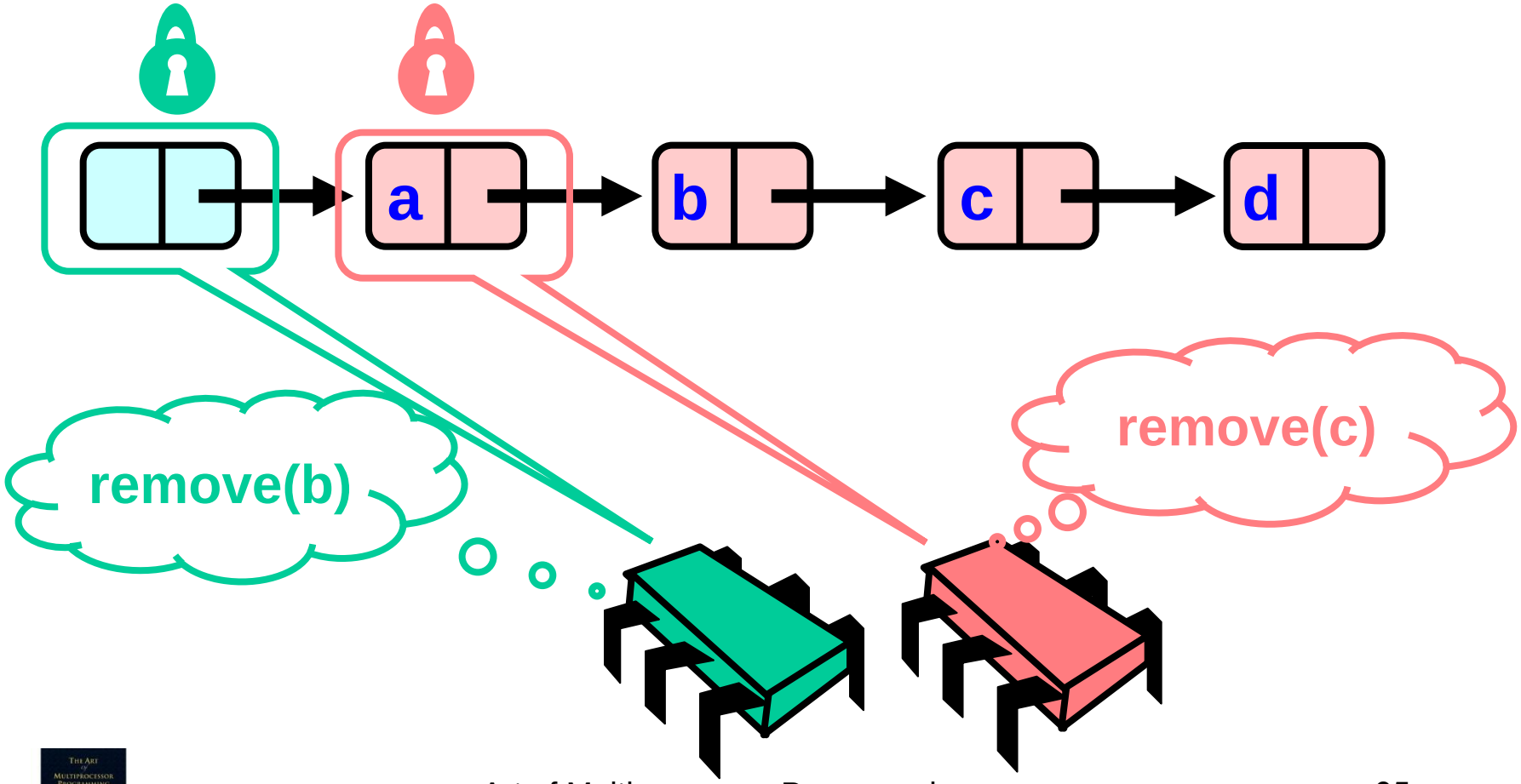


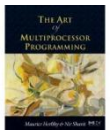
Removing a Node



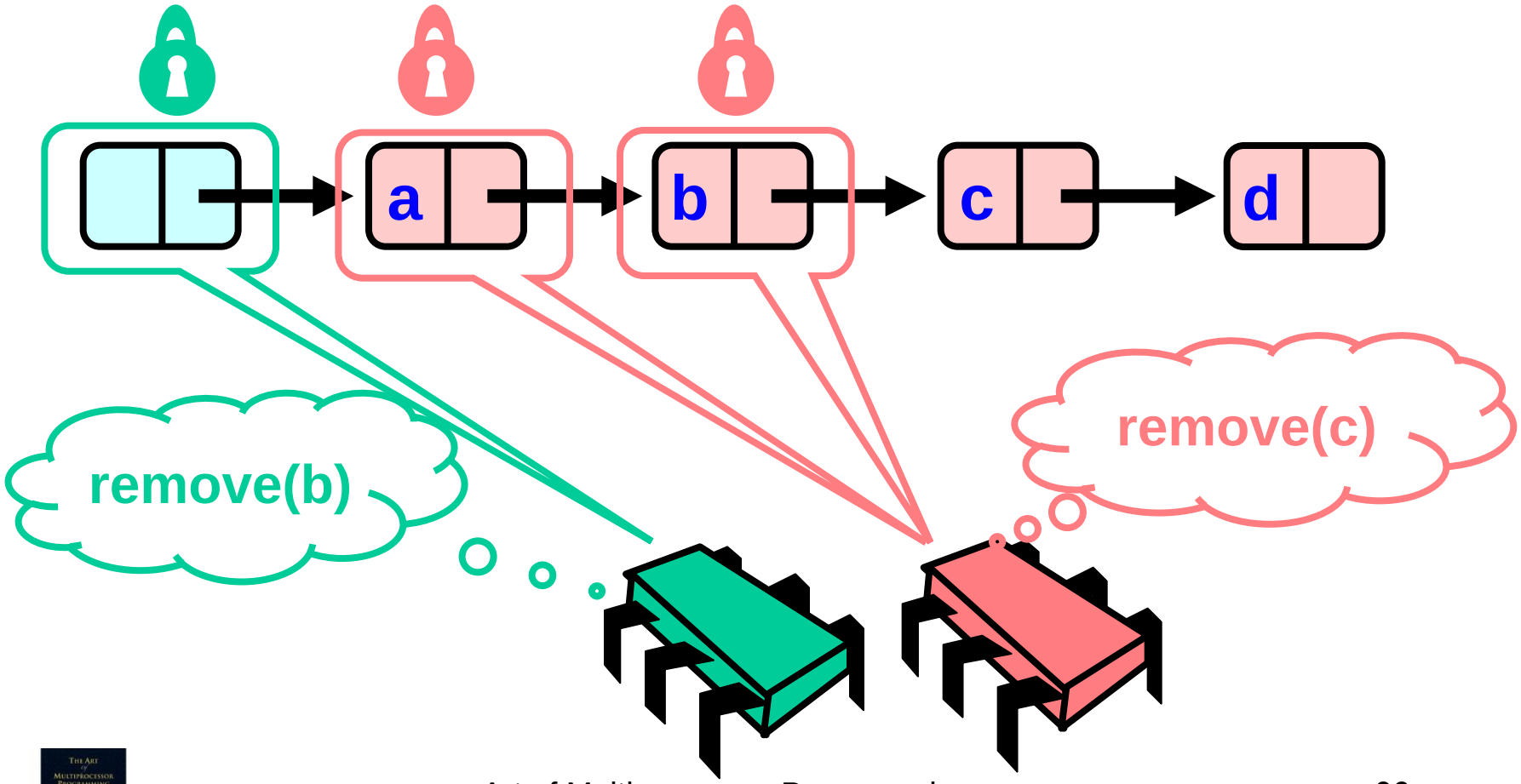


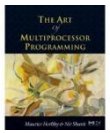
Removing a Node



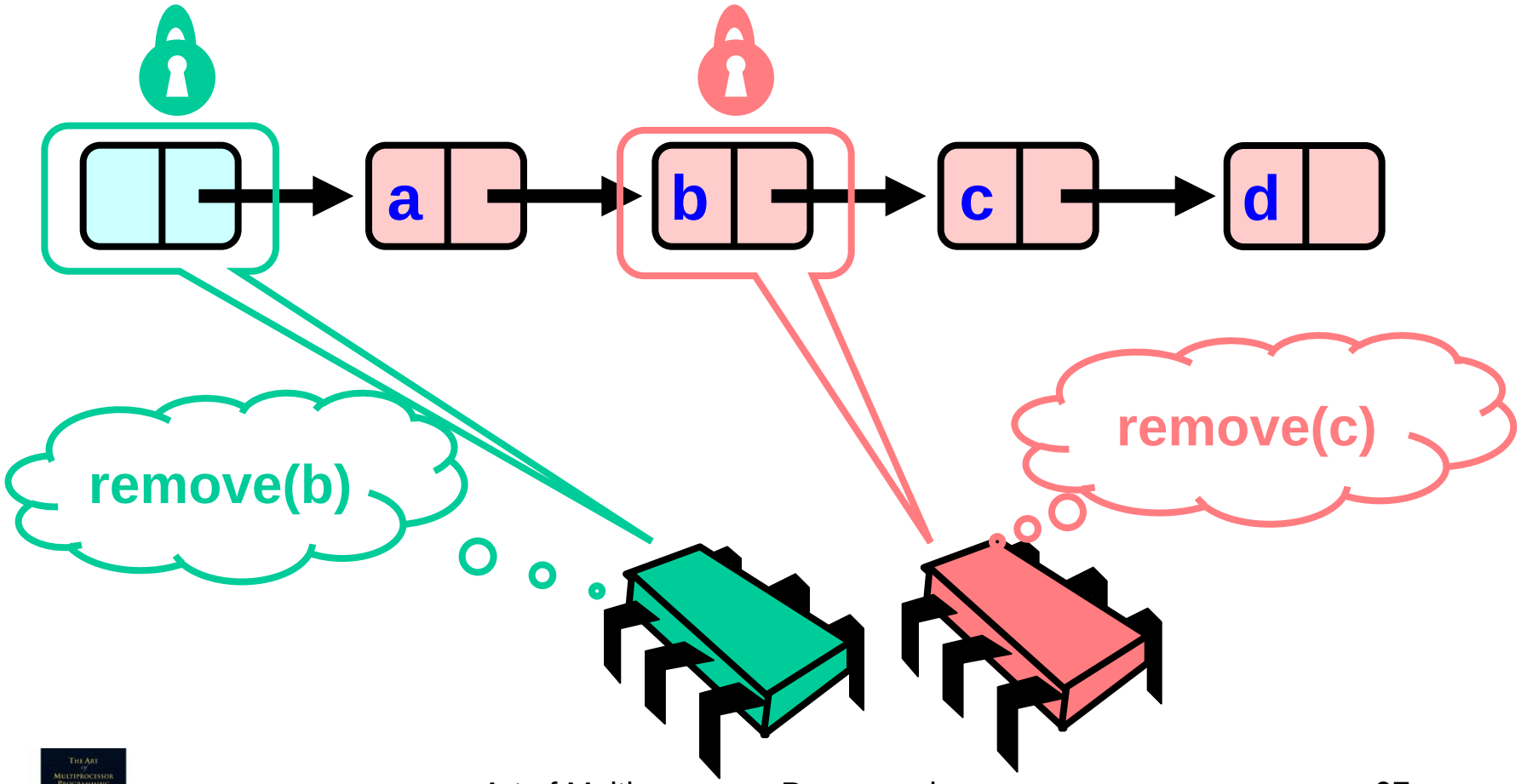


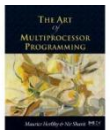
Removing a Node



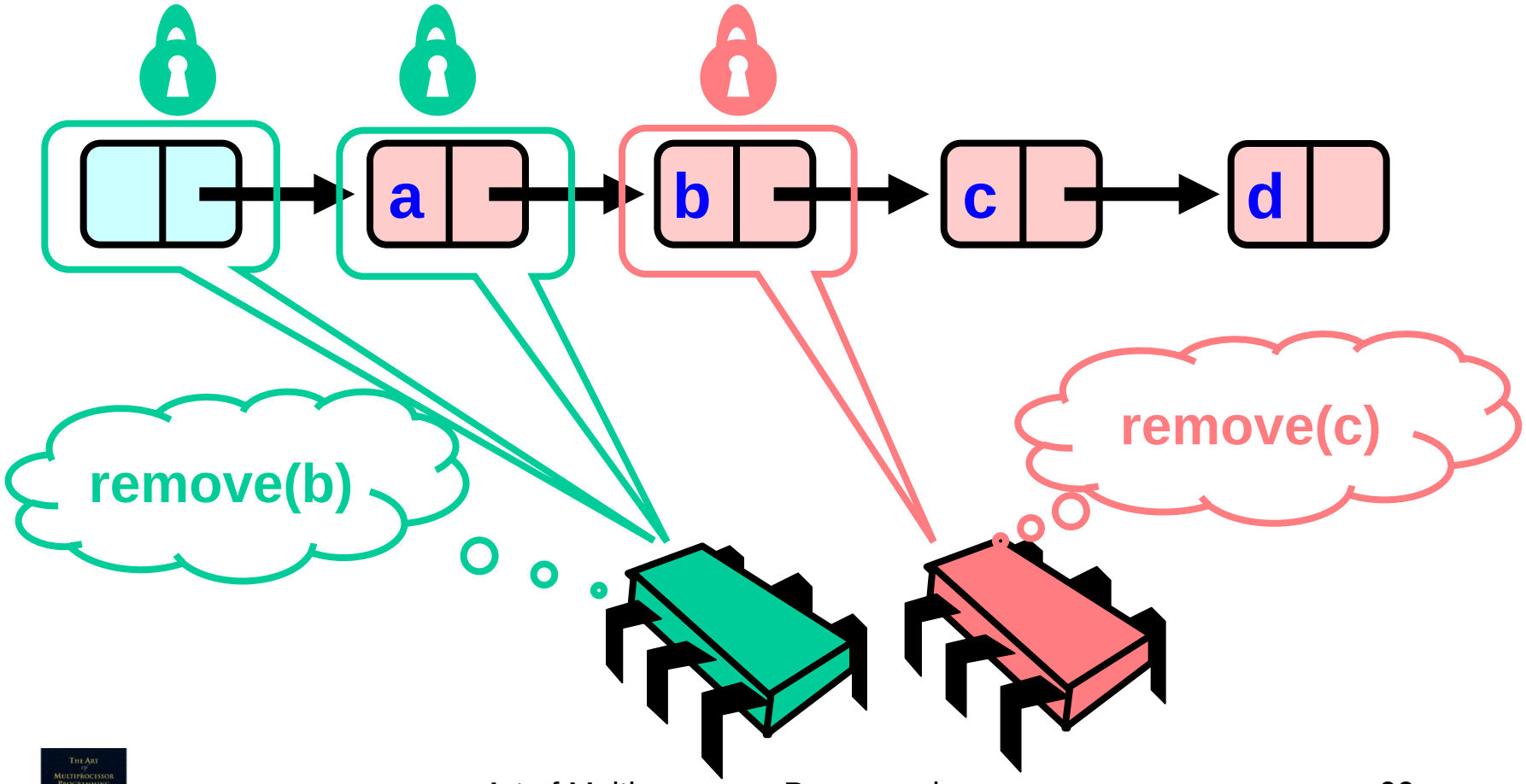


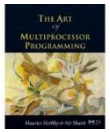
Removing a Node



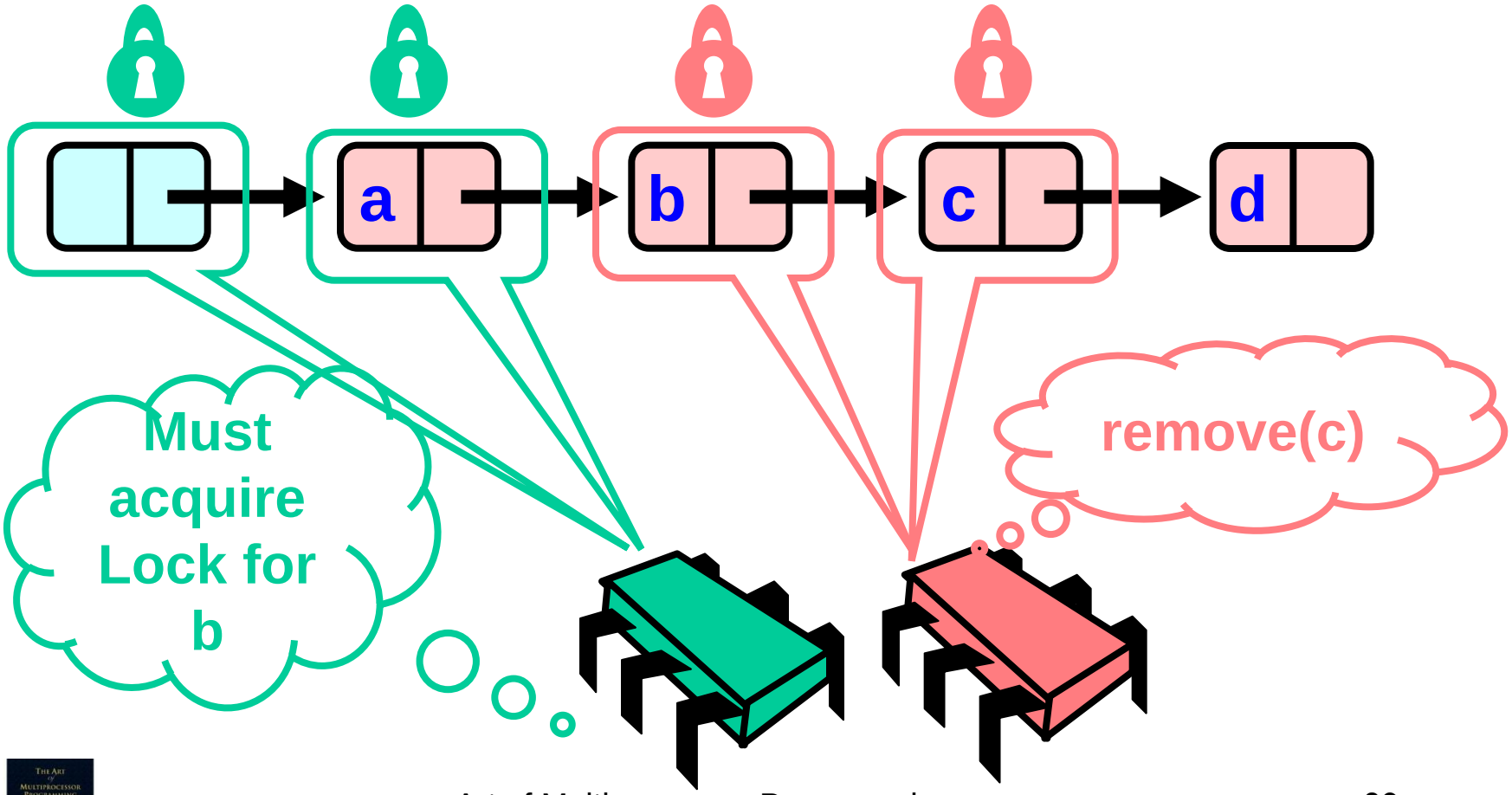


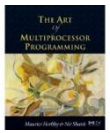
Removing a Node



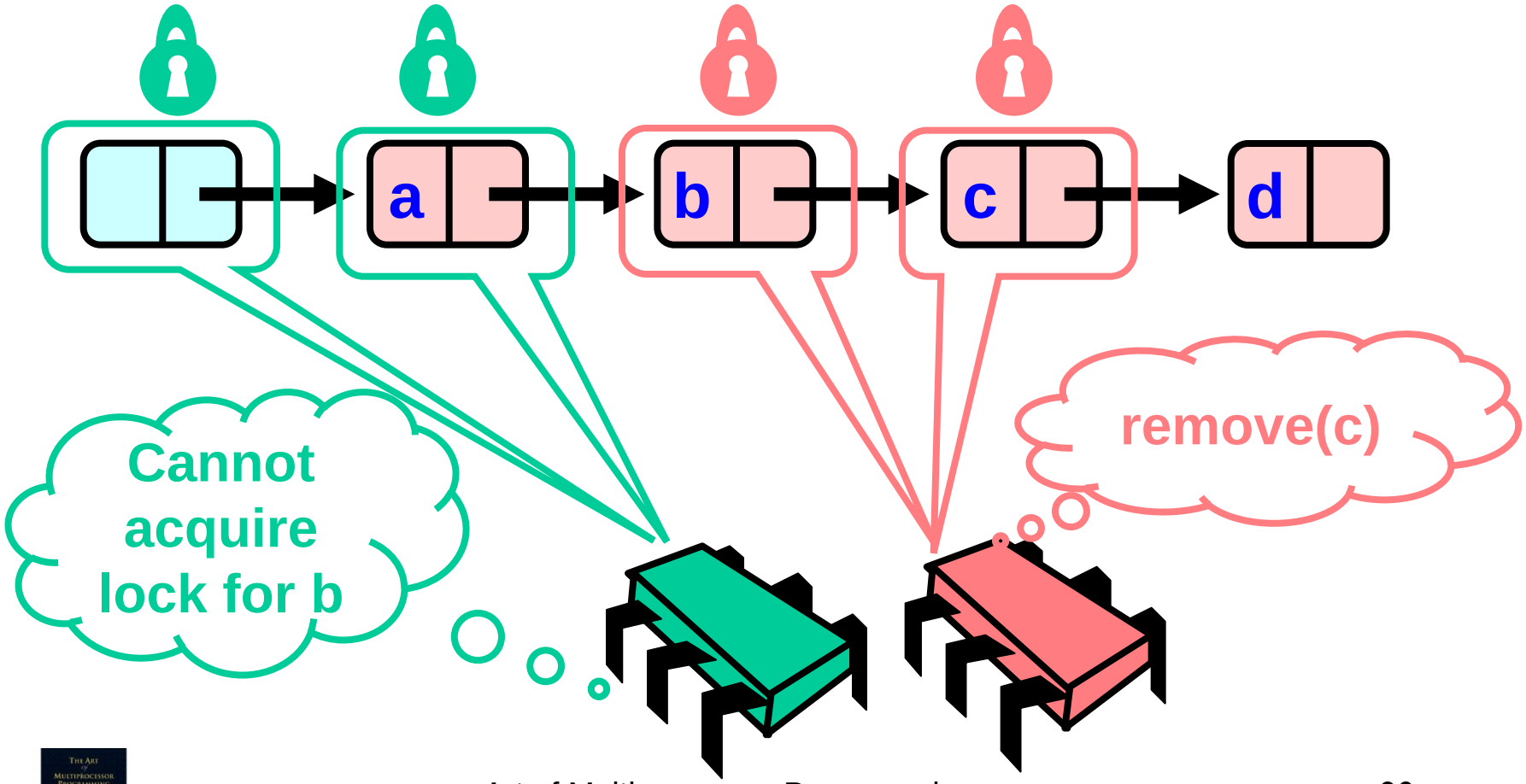


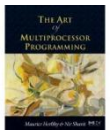
Removing a Node



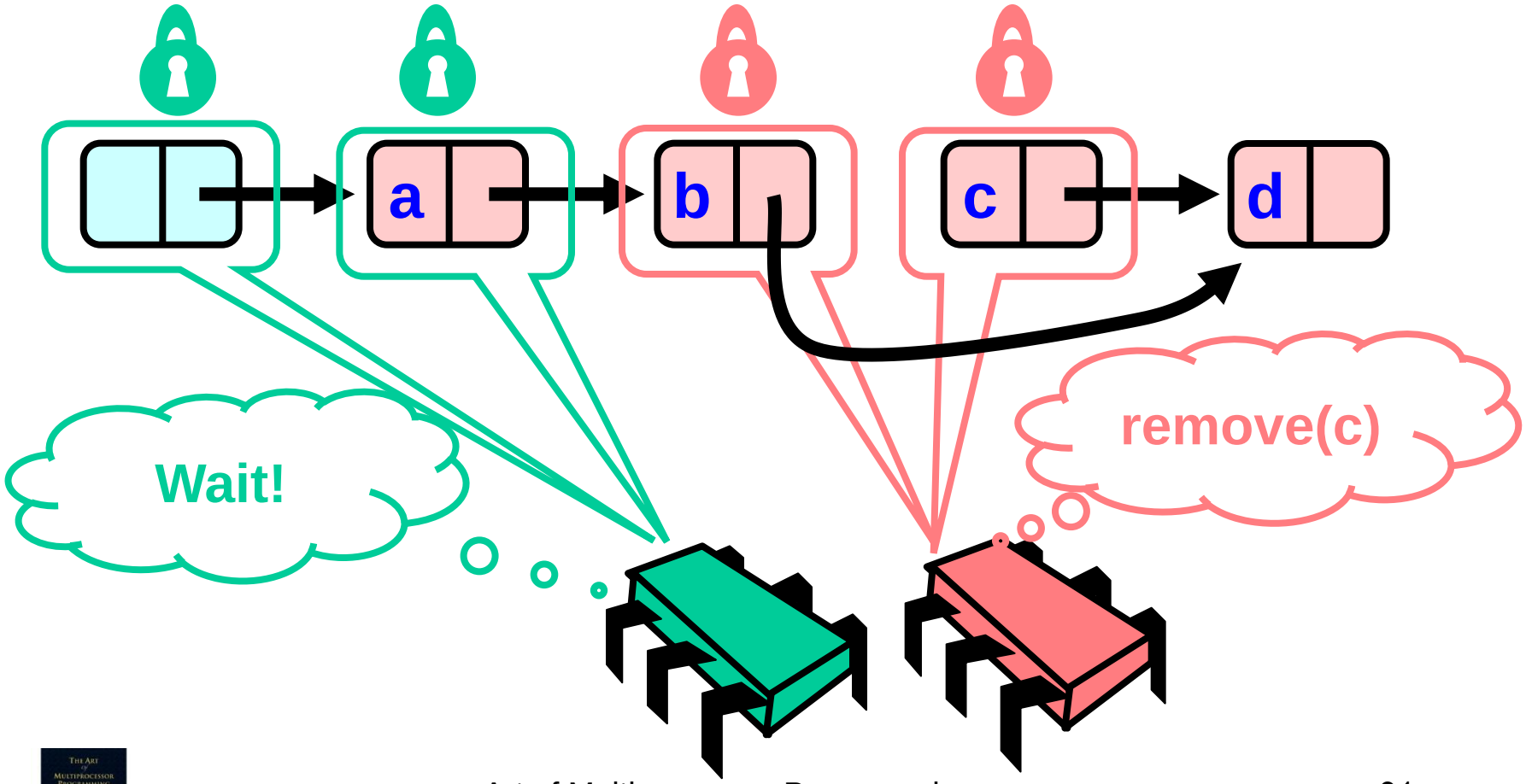


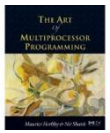
Removing a Node



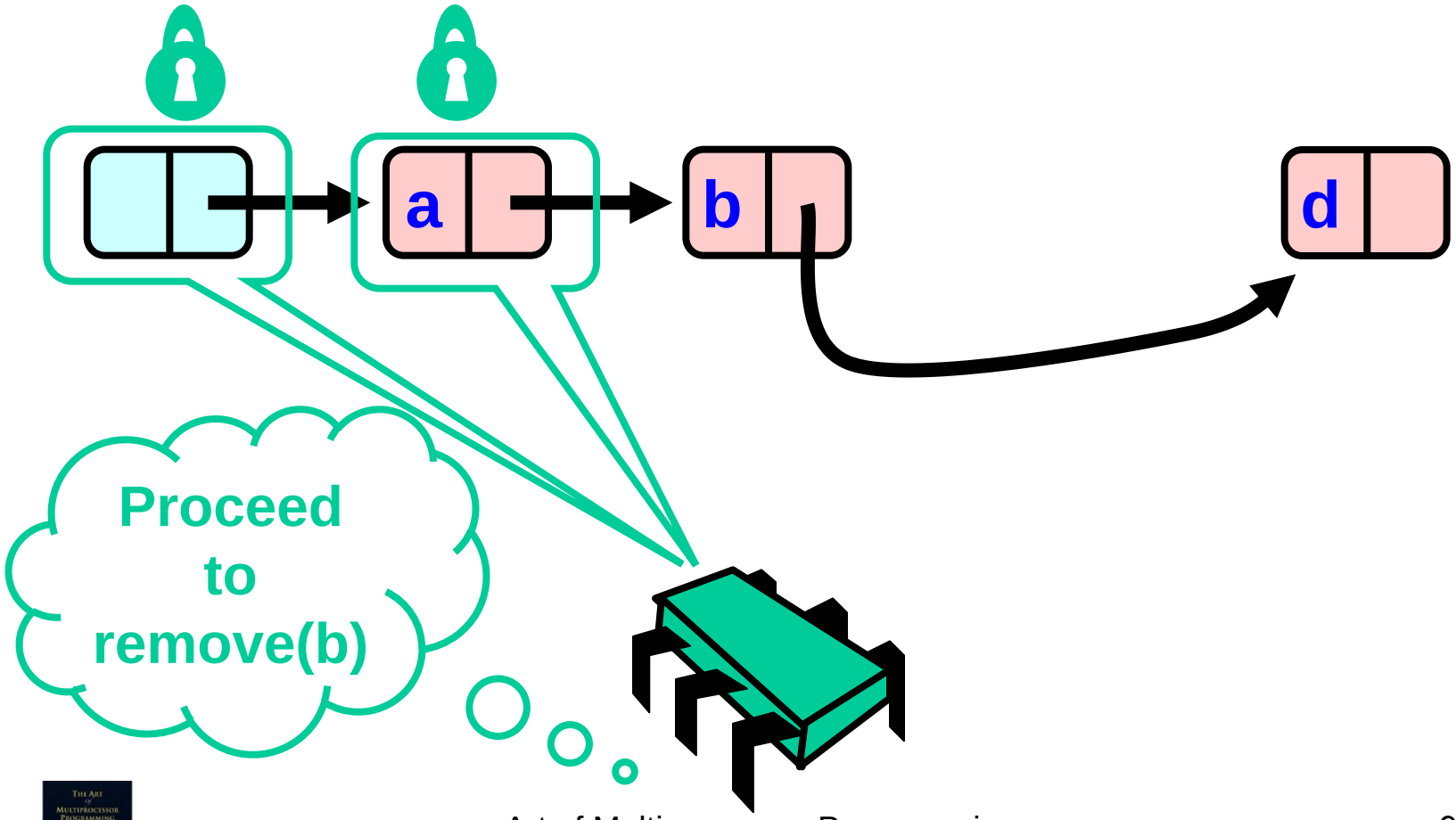


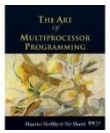
Removing a Node



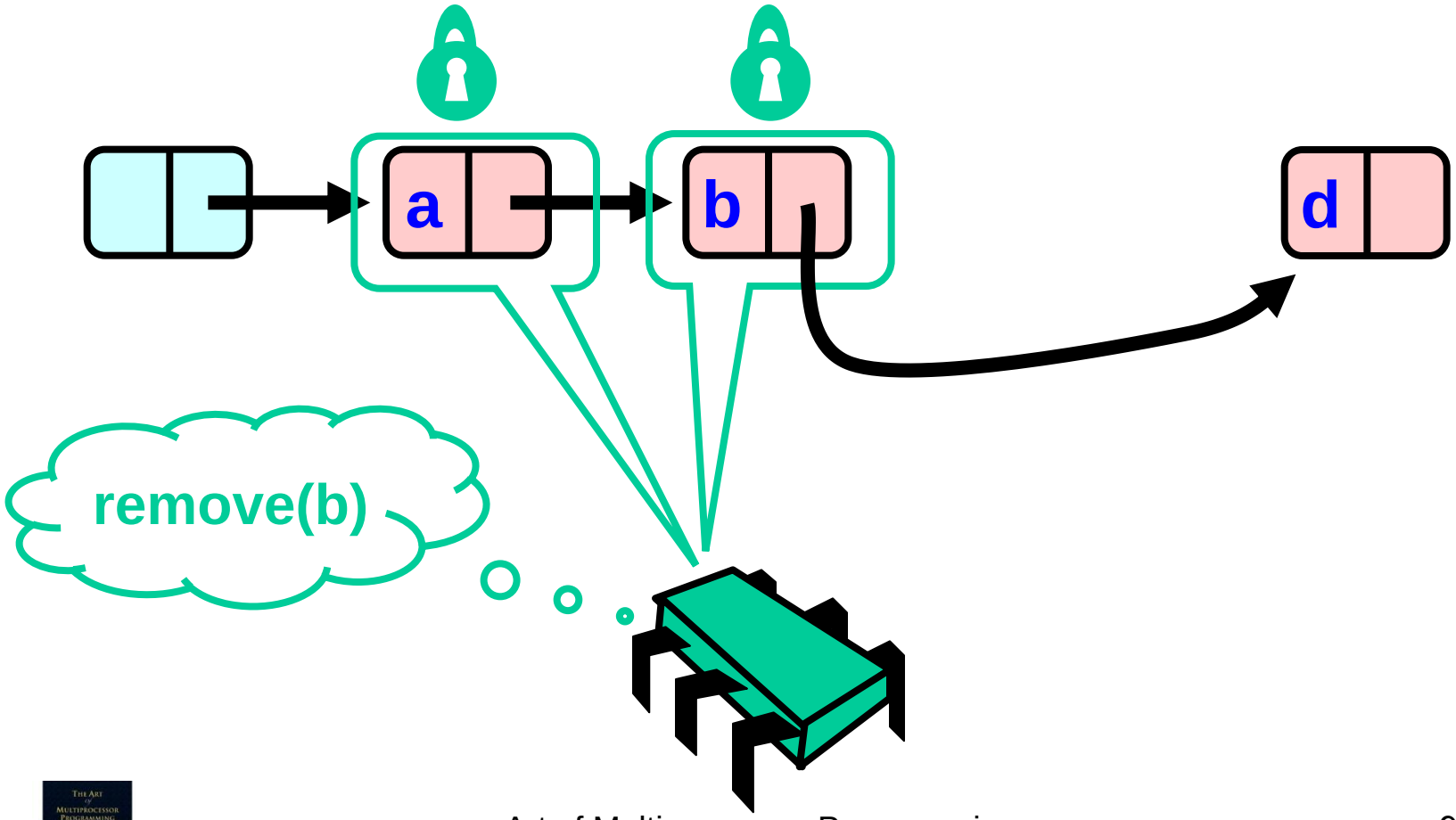


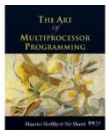
Removing a Node



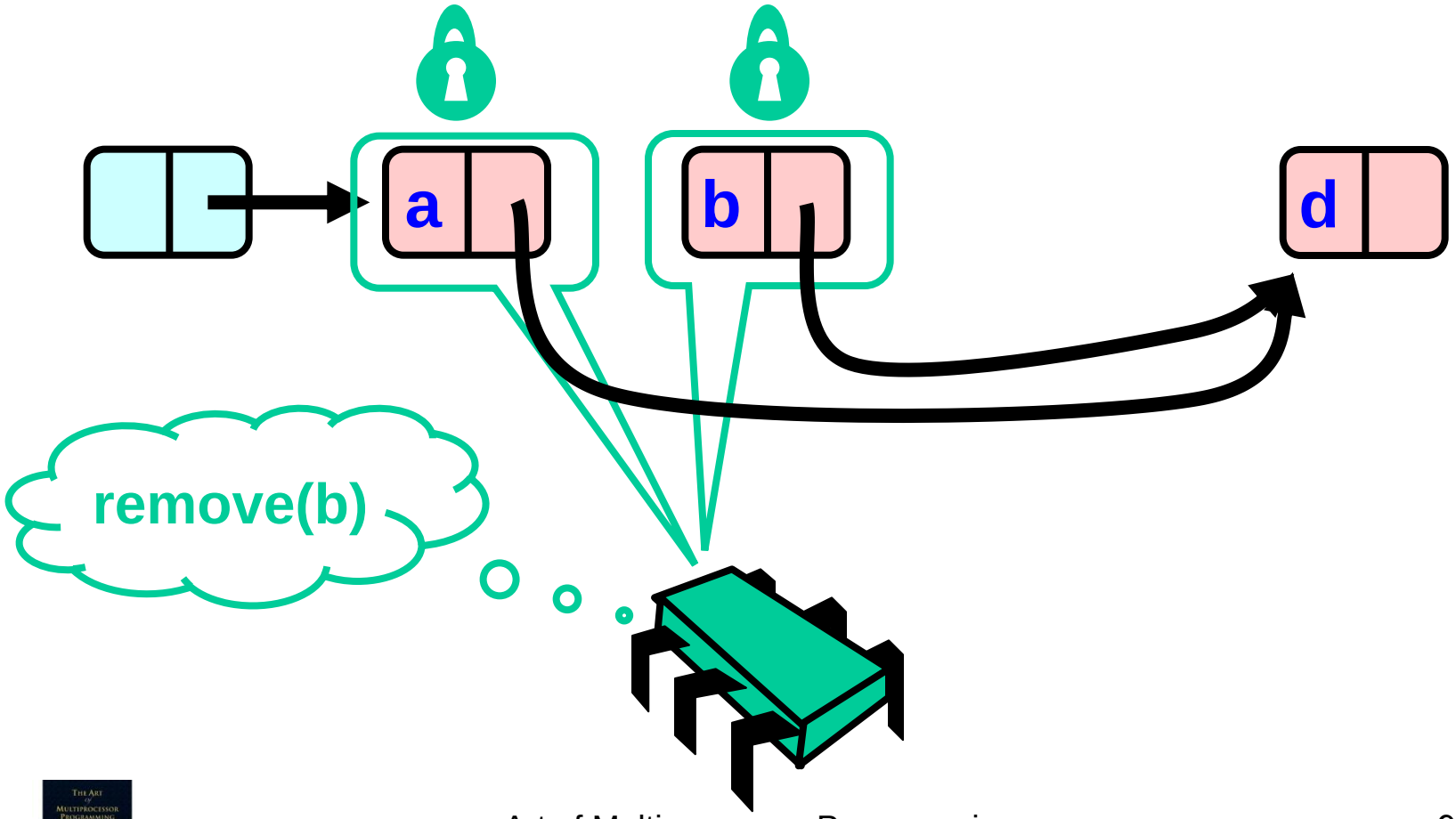


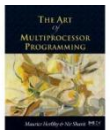
Removing a Node



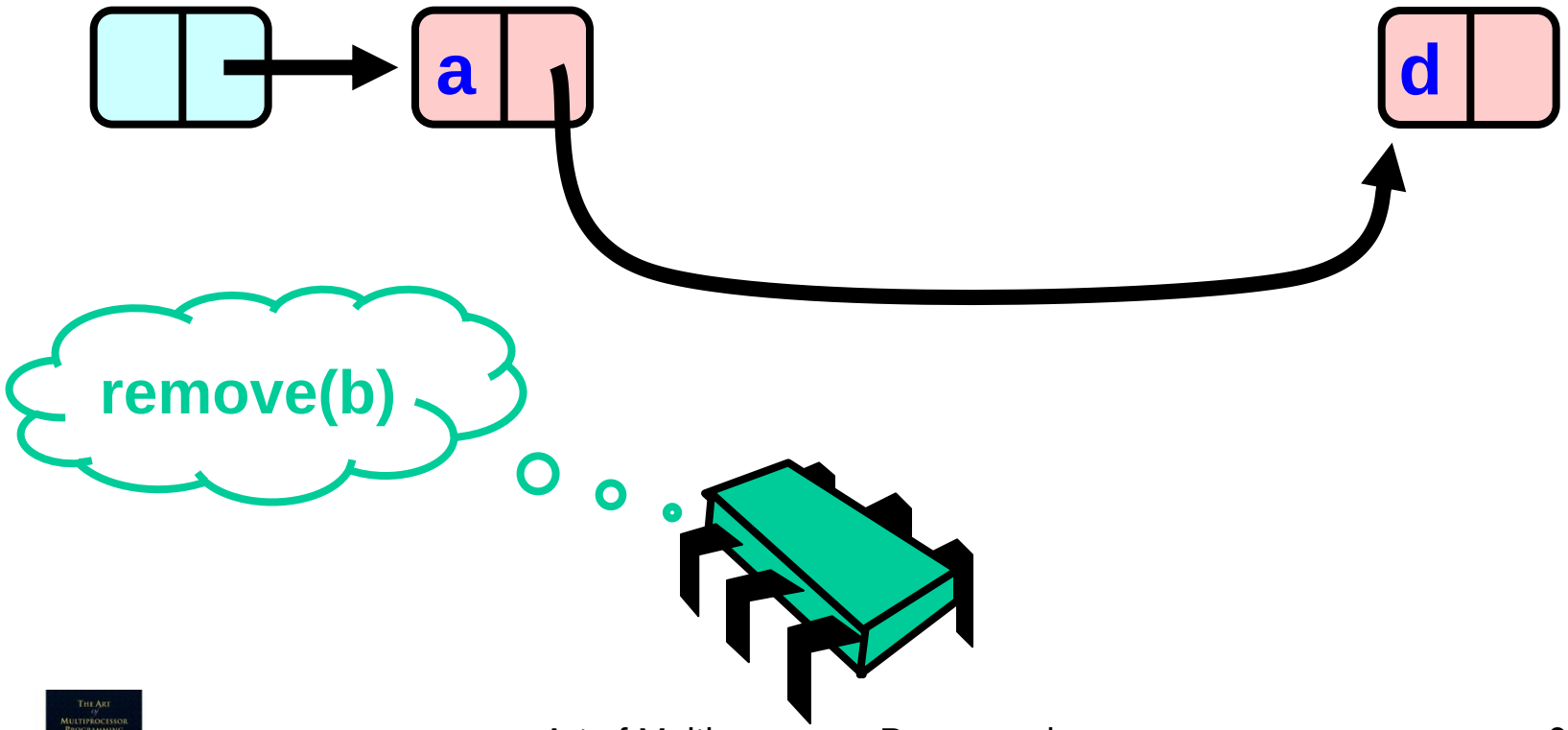


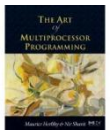
Removing a Node



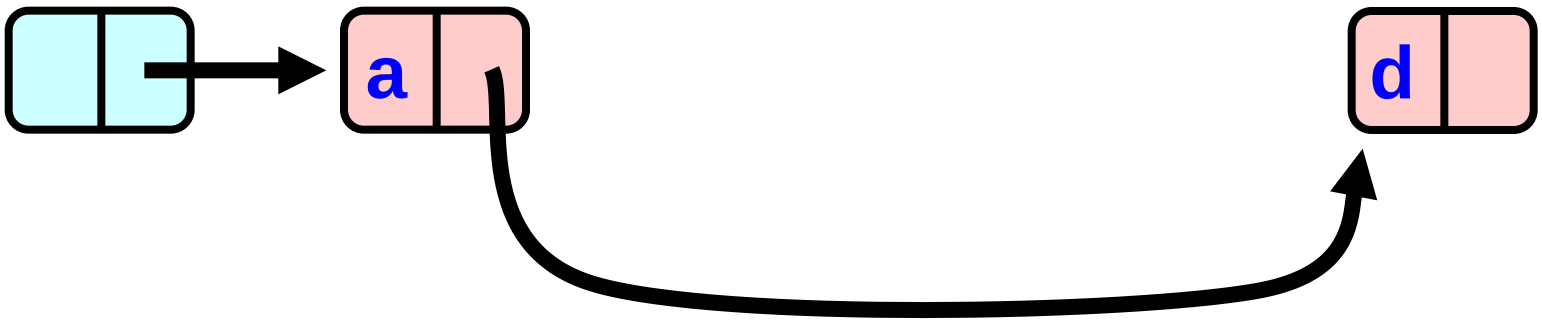


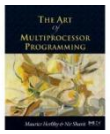
Removing a Node





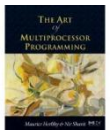
Removing a Node





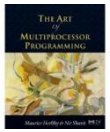
Adding Nodes

- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted
  - (Is successor lock actually required?)



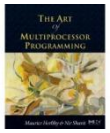
Drawbacks

- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient

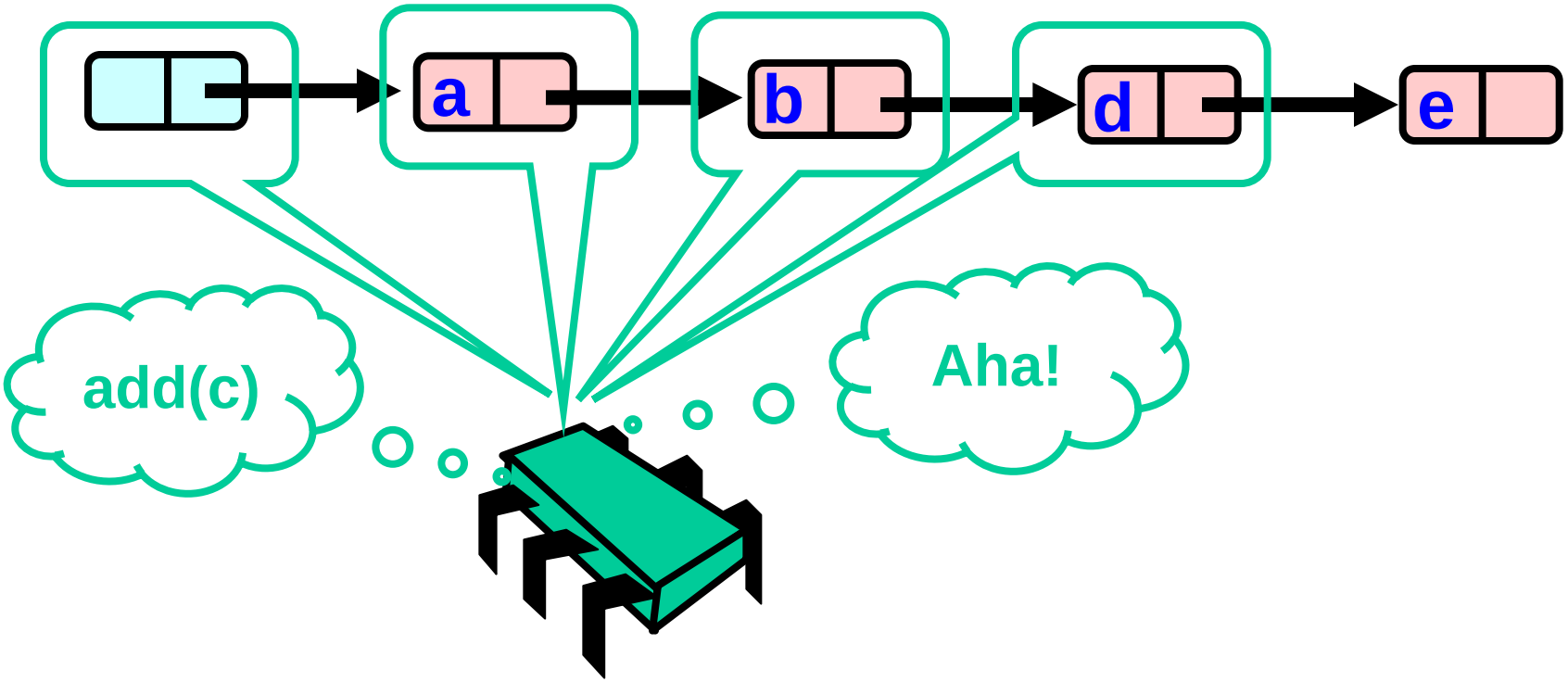


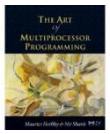
Optimistic Synchronization

- Find nodes without locking
- Lock nodes
- Check that everything is OK

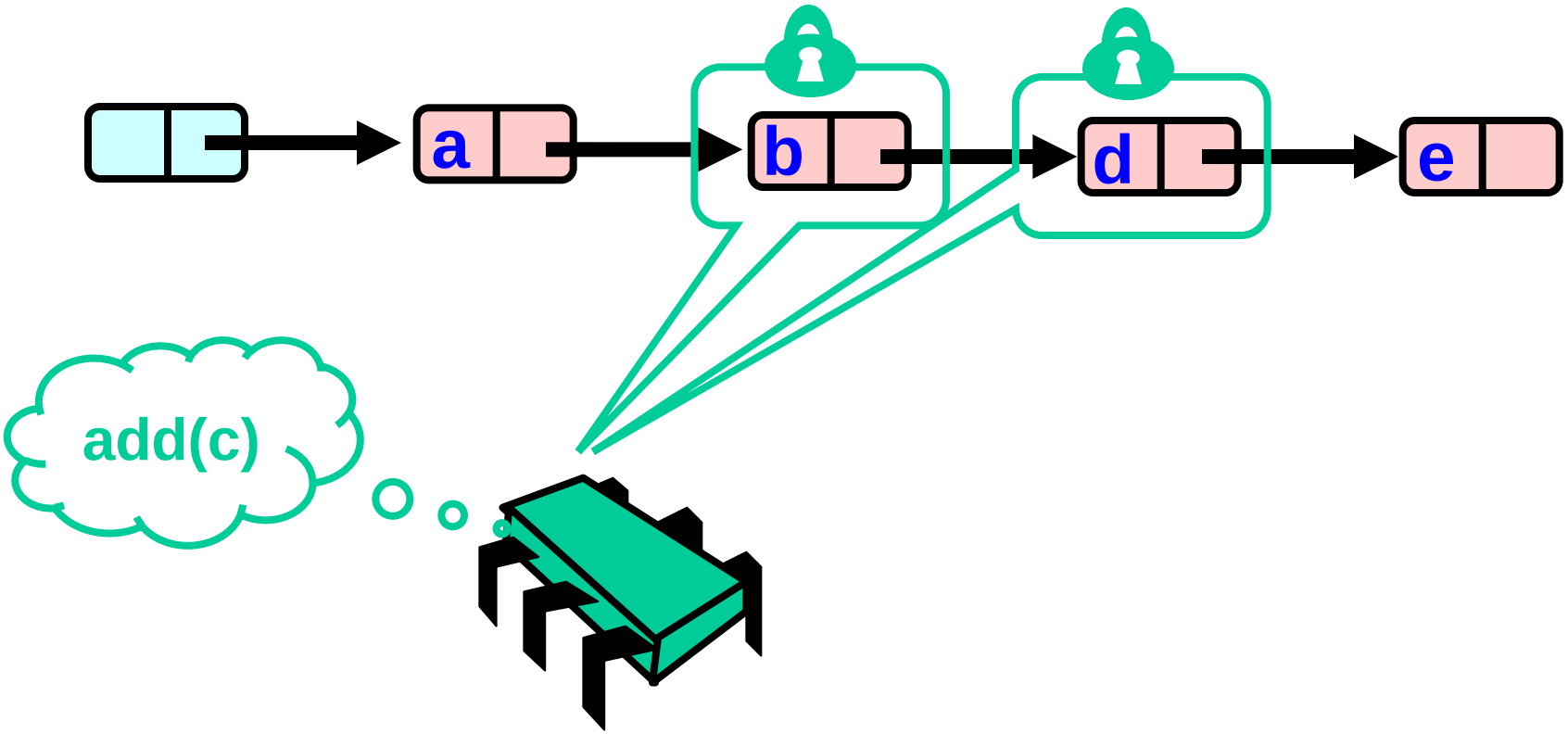


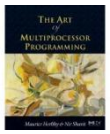
Optimistic: Traverse without Locking



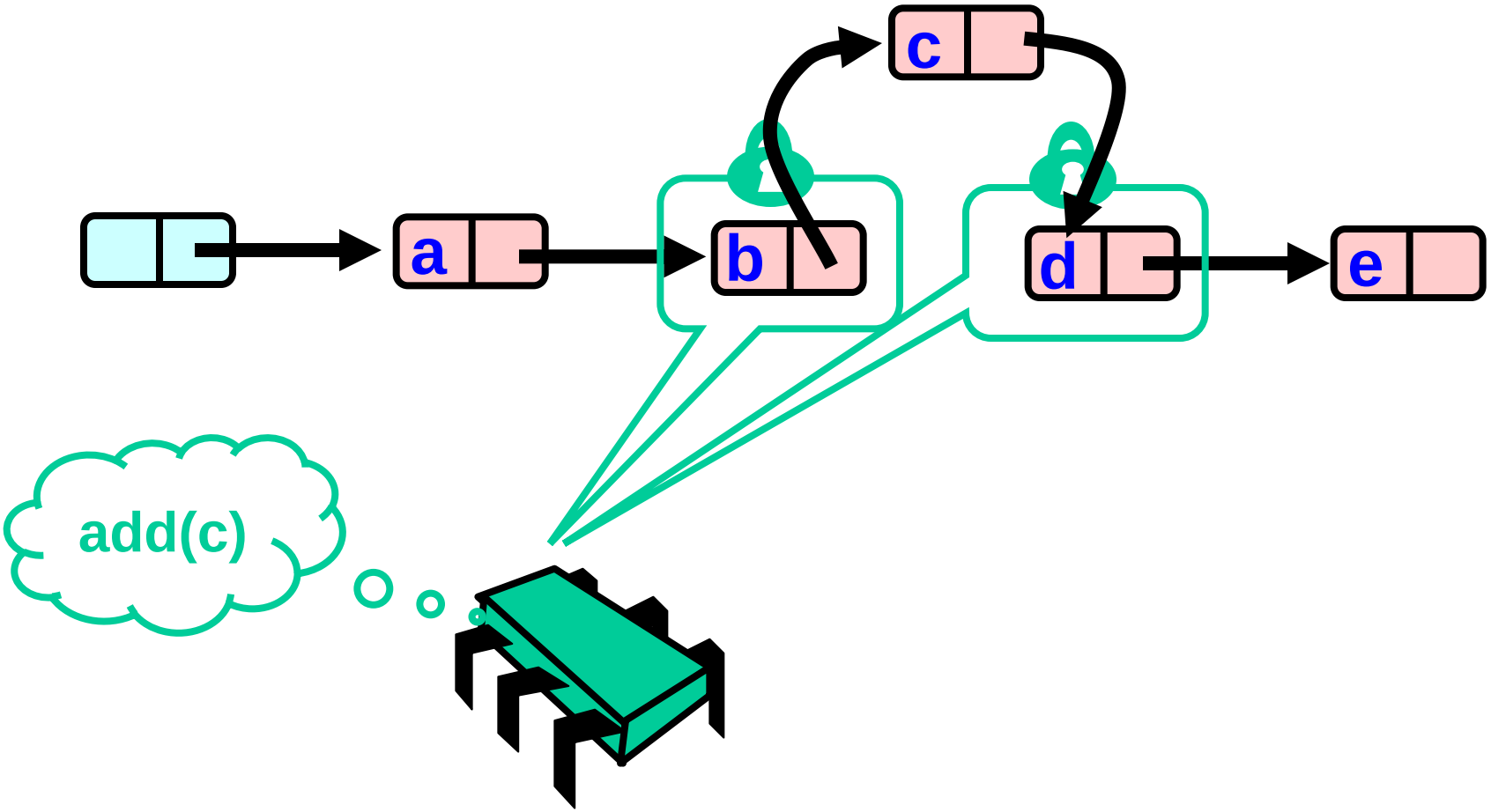


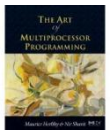
Optimistic: Lock and Load



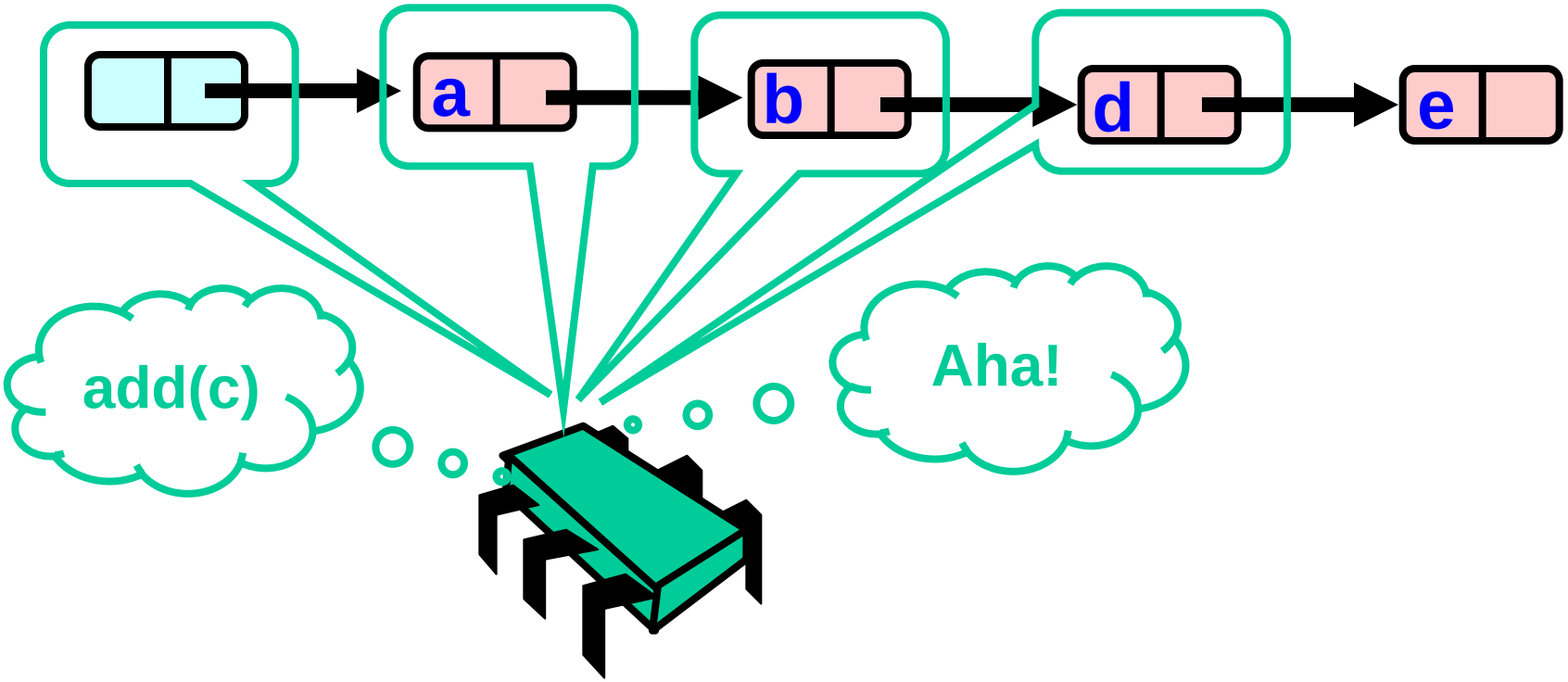


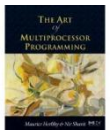
Optimistic: Lock and Load



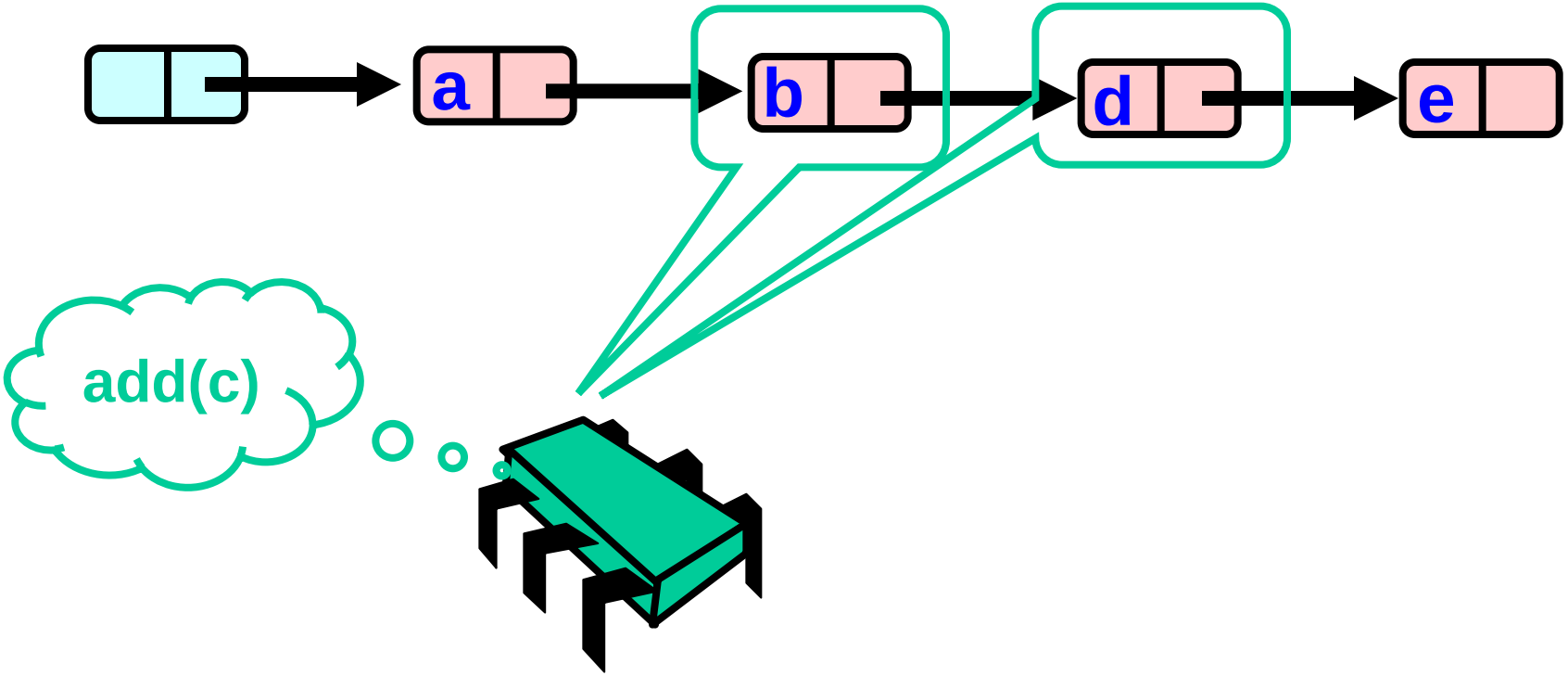


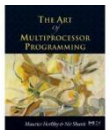
What could go wrong?



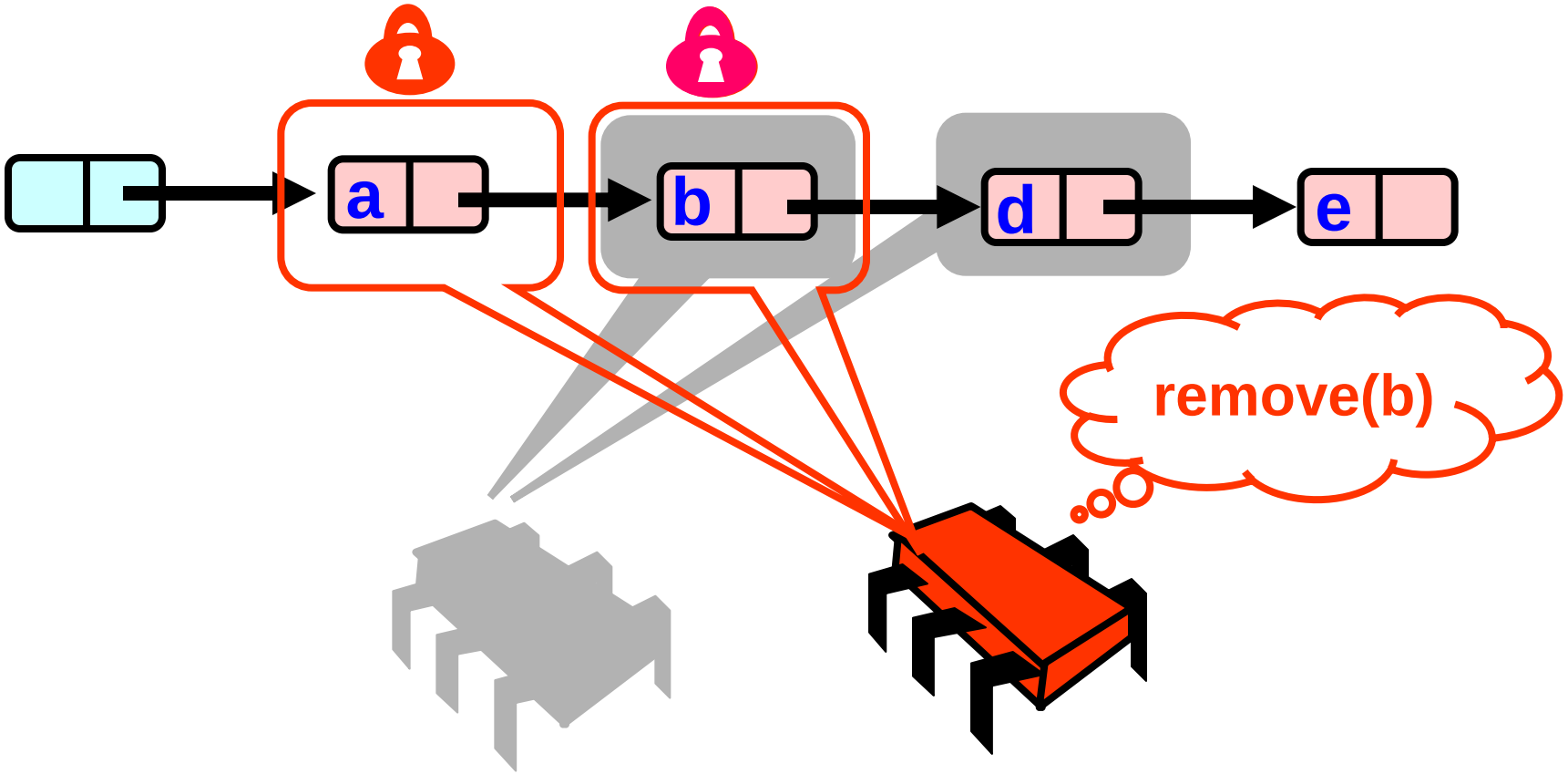


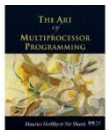
What could go wrong?



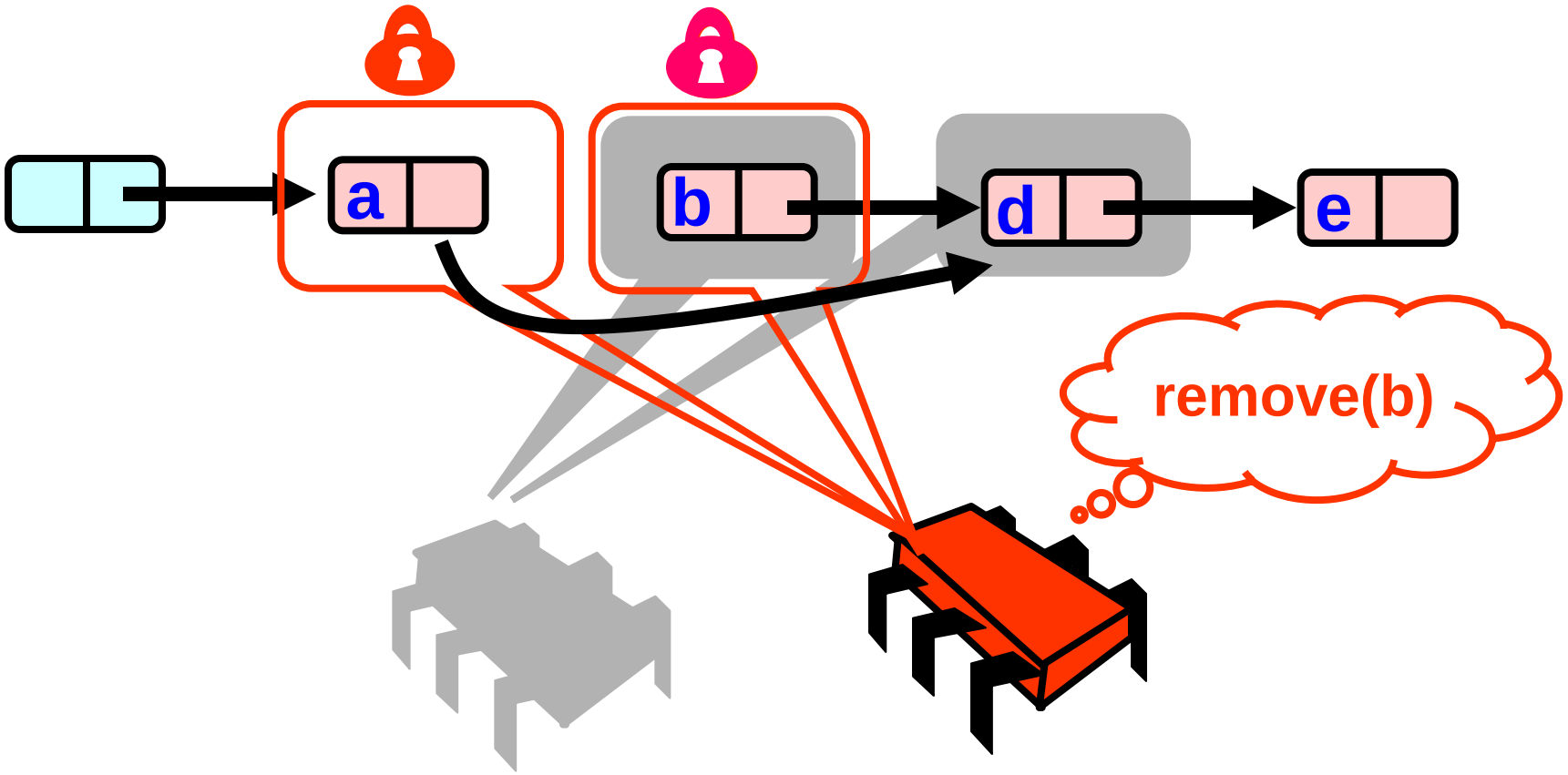


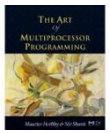
What could go wrong?



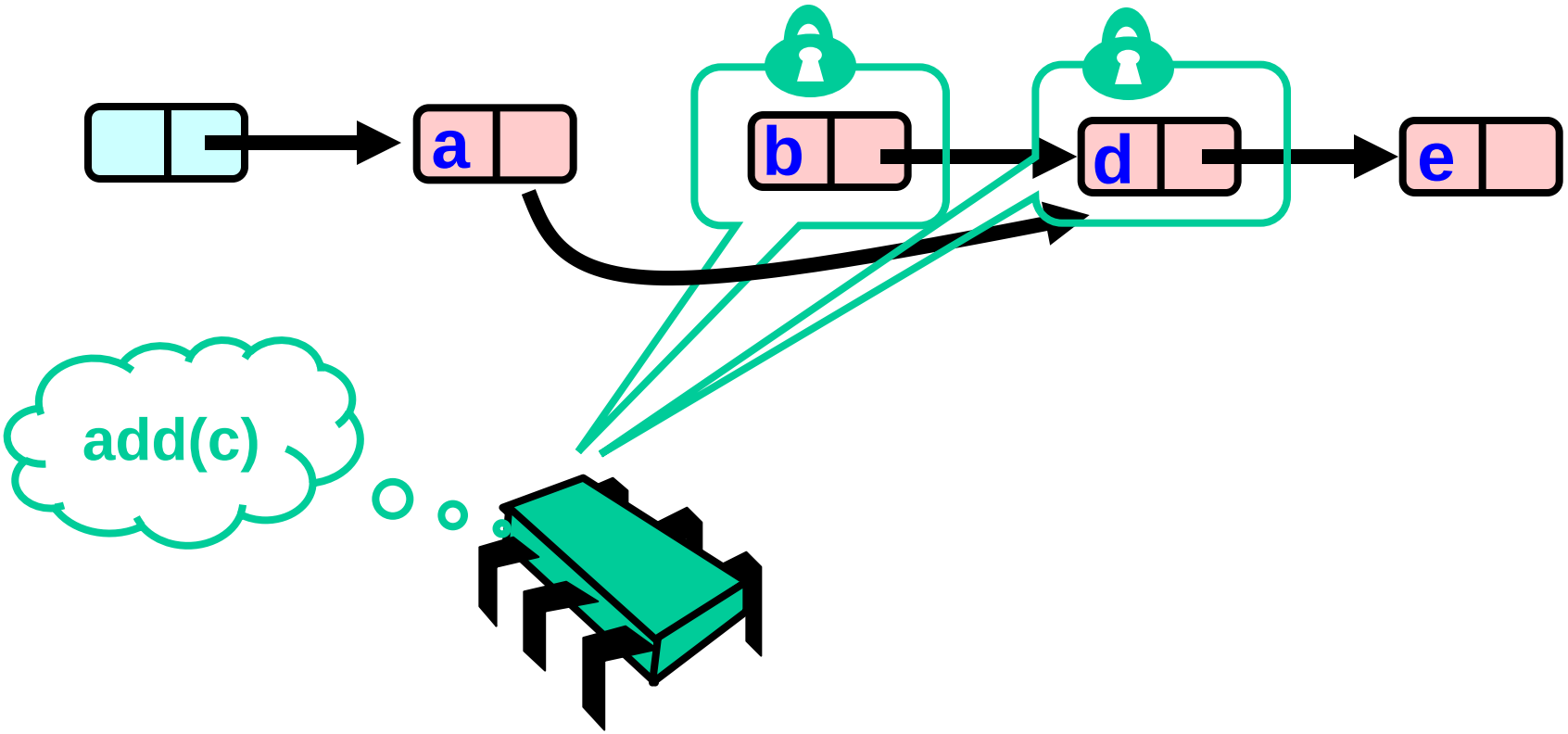


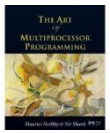
What could go wrong?



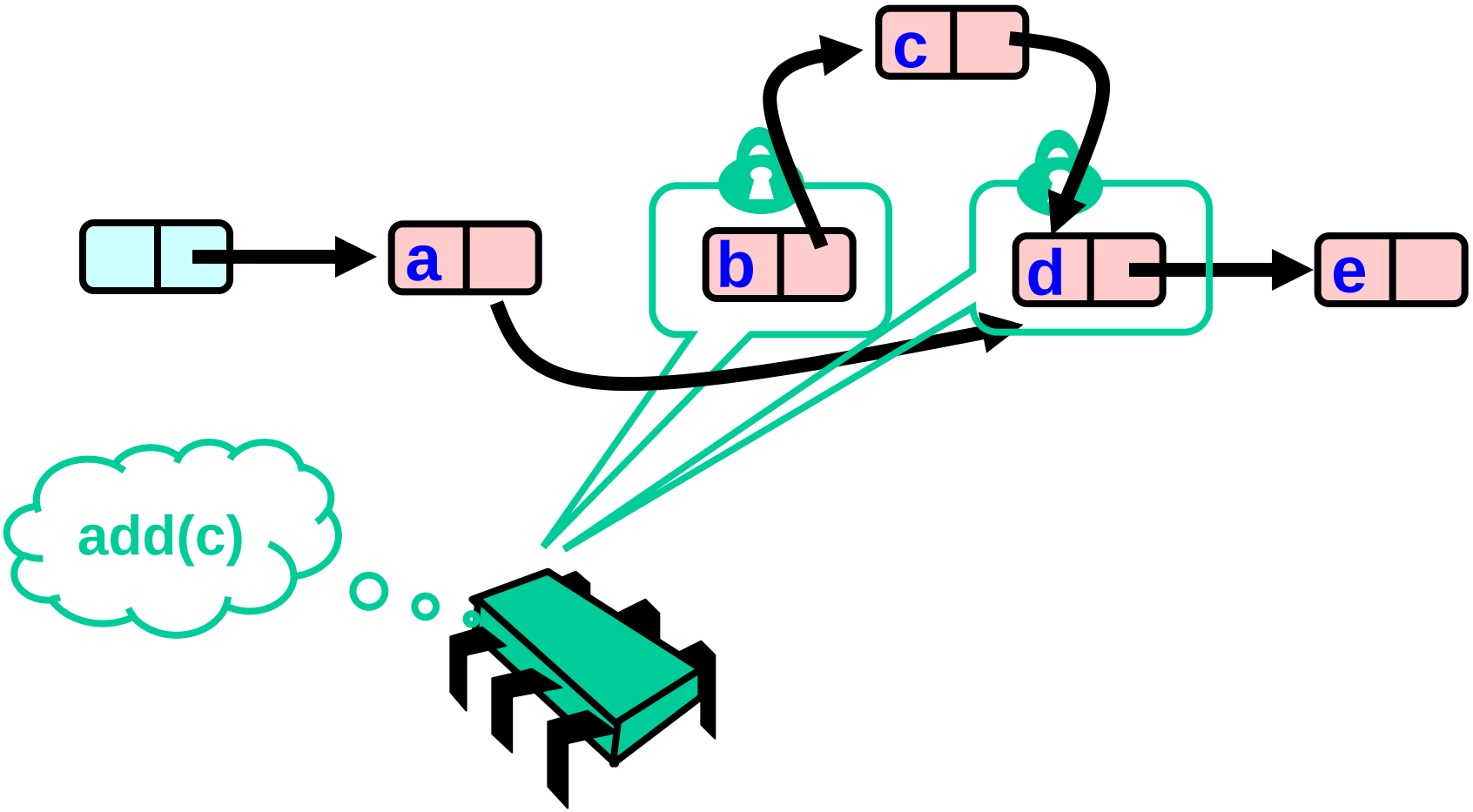


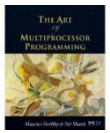
What could go wrong?



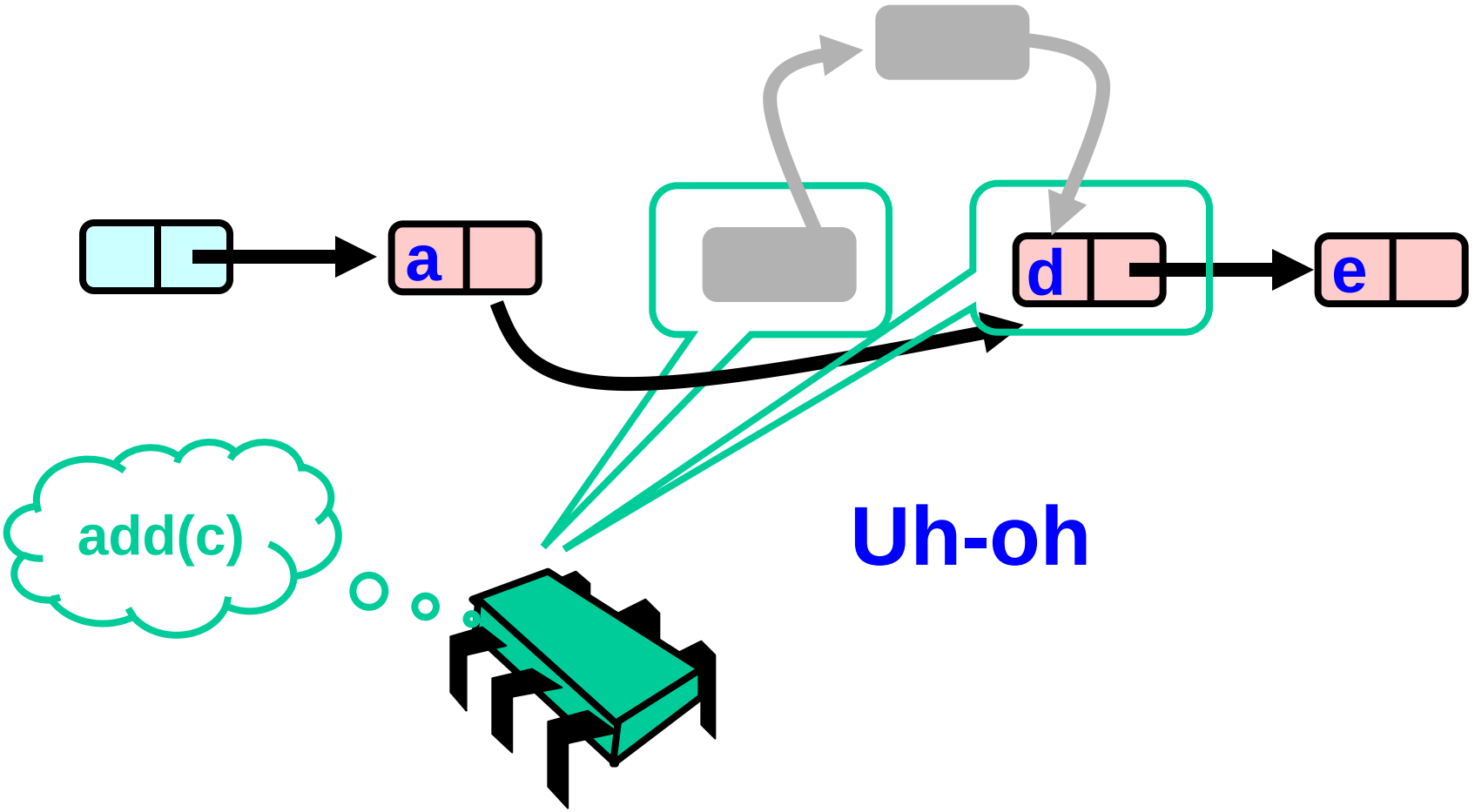


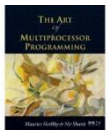
What could go wrong?



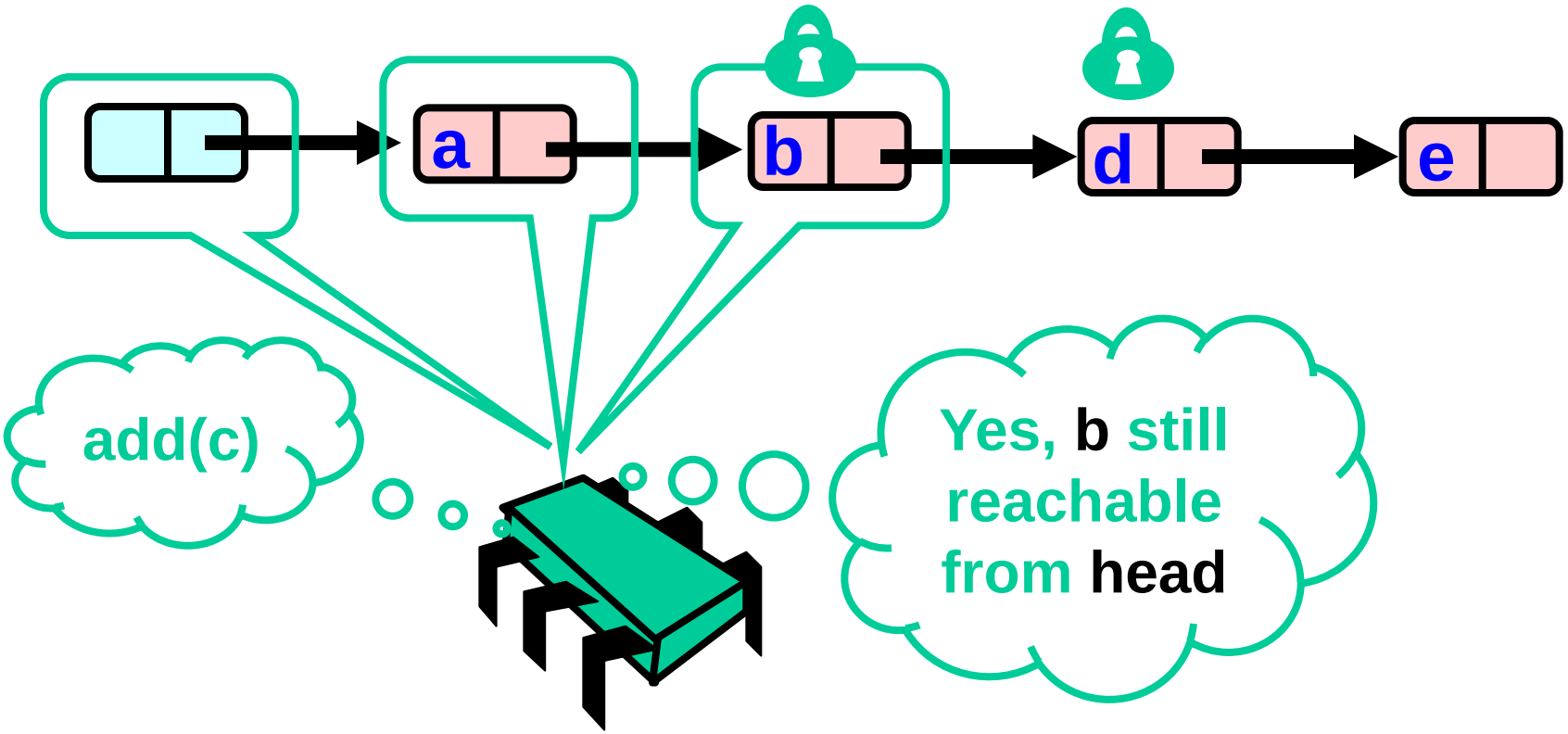


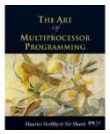
What could go wrong?



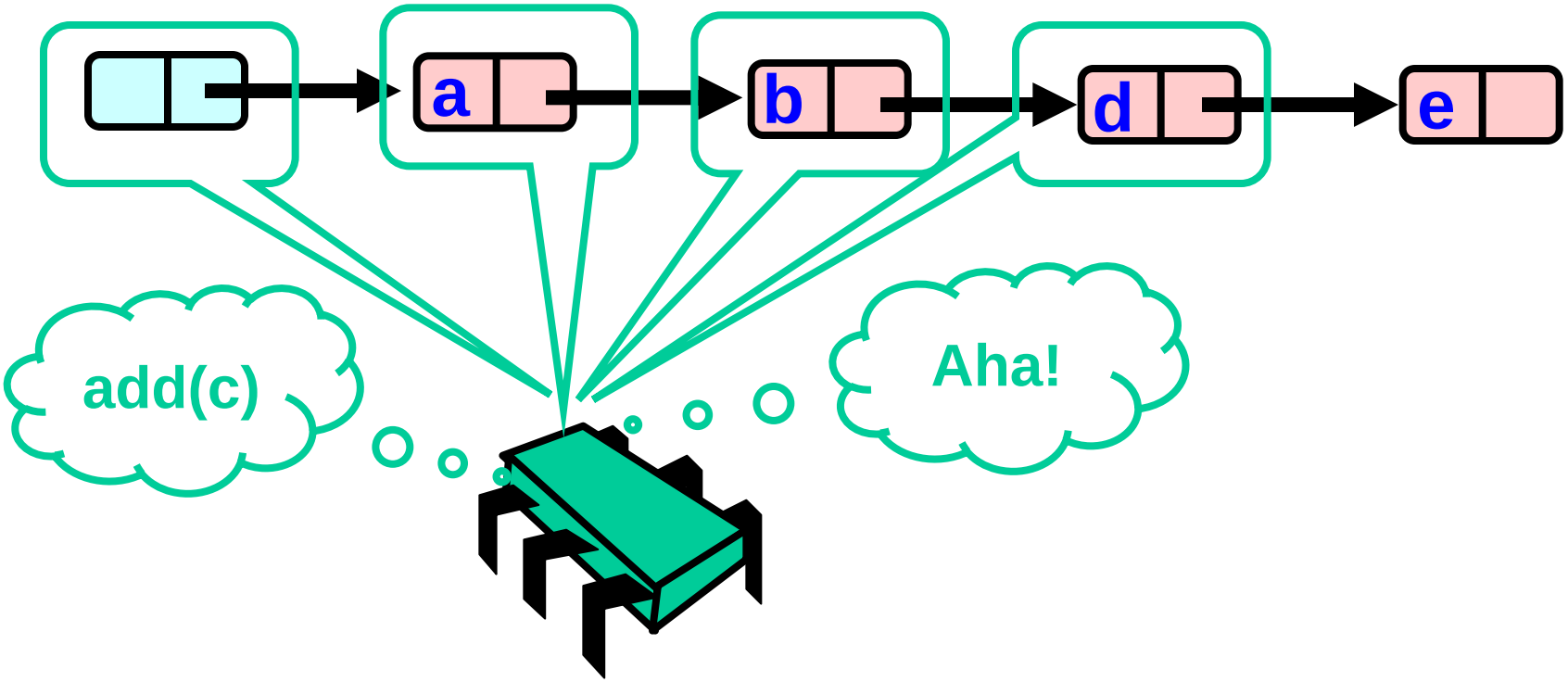


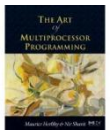
Validate – Part 1



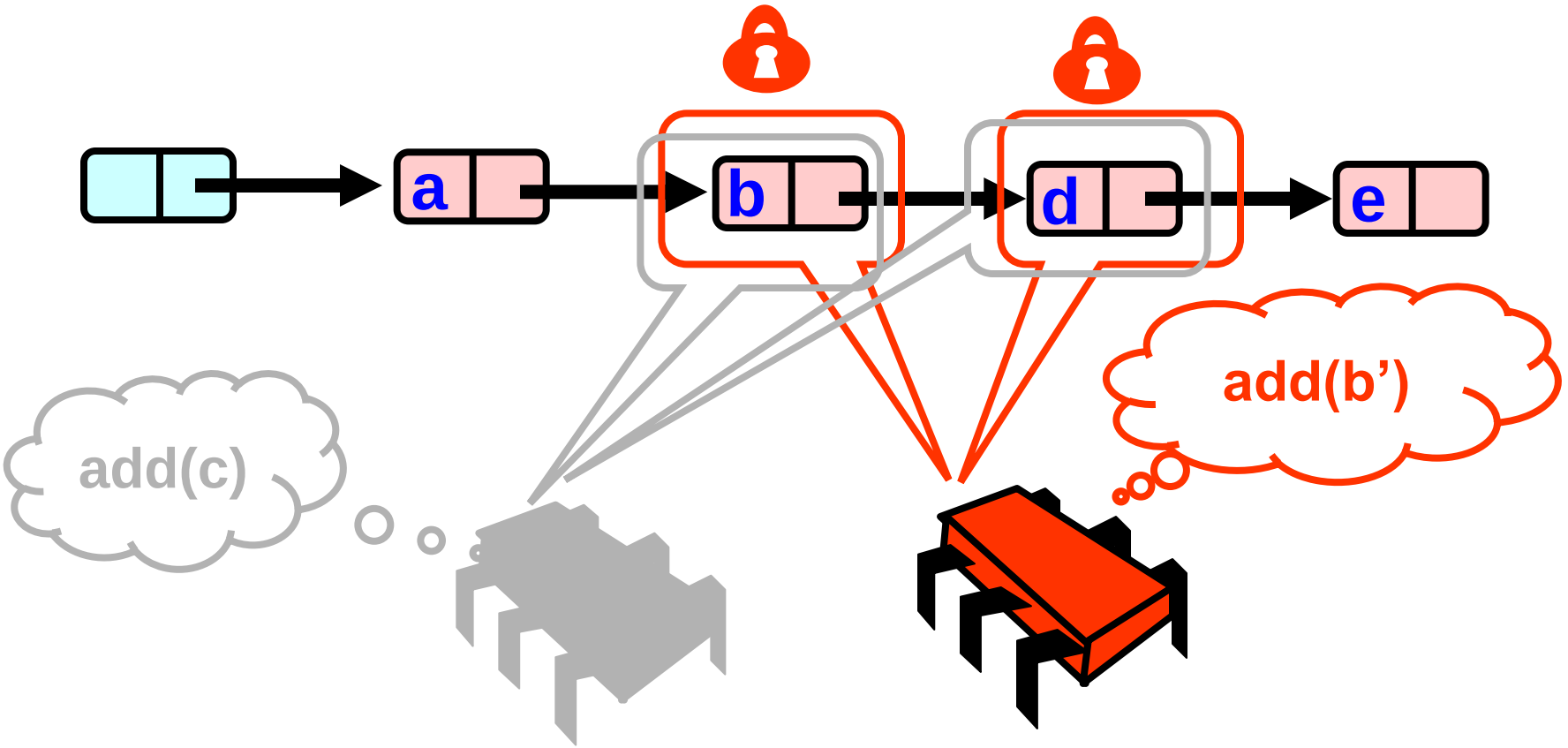


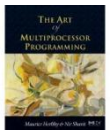
What Else Could Go Wrong?



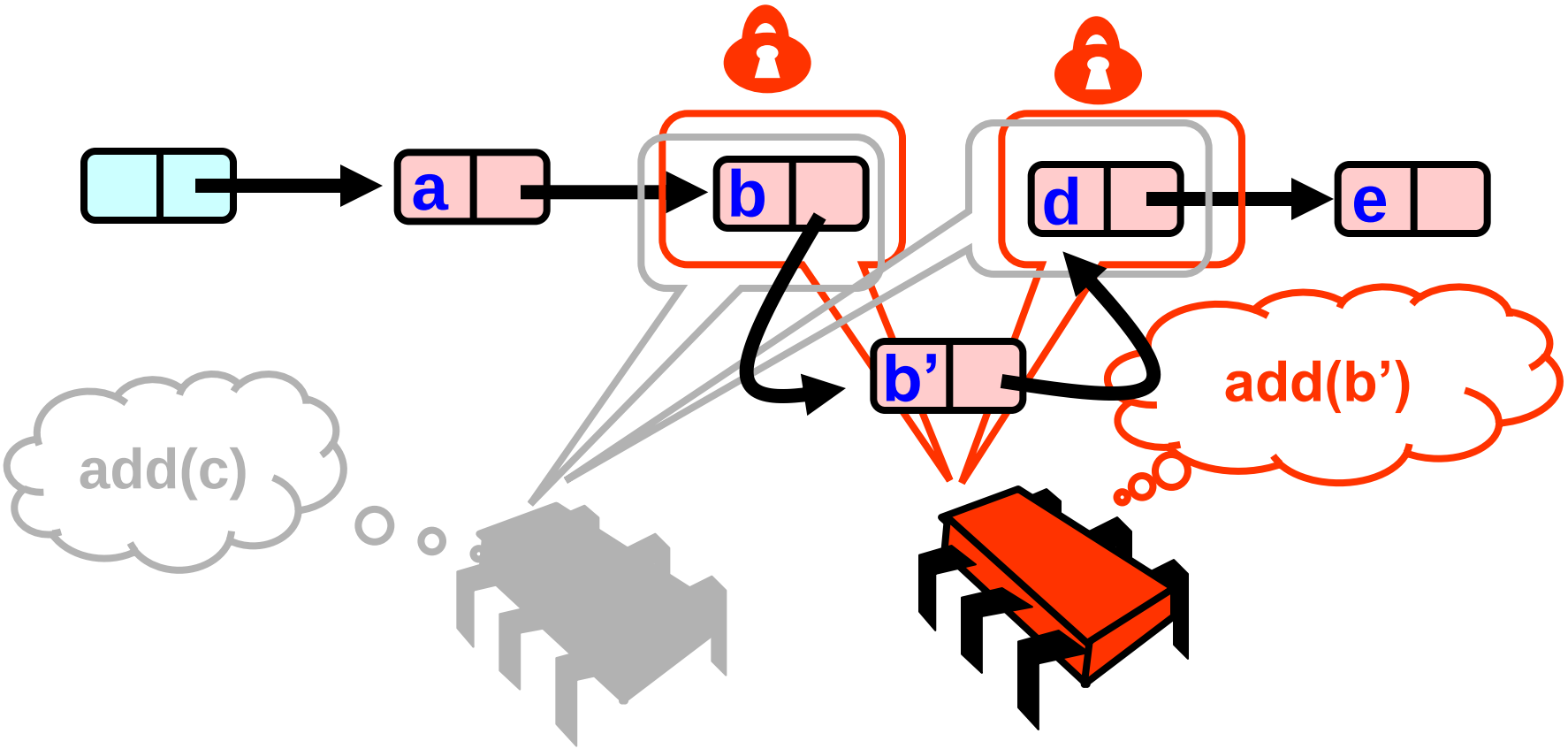


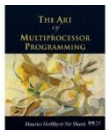
What Else Coould Go Wrong?



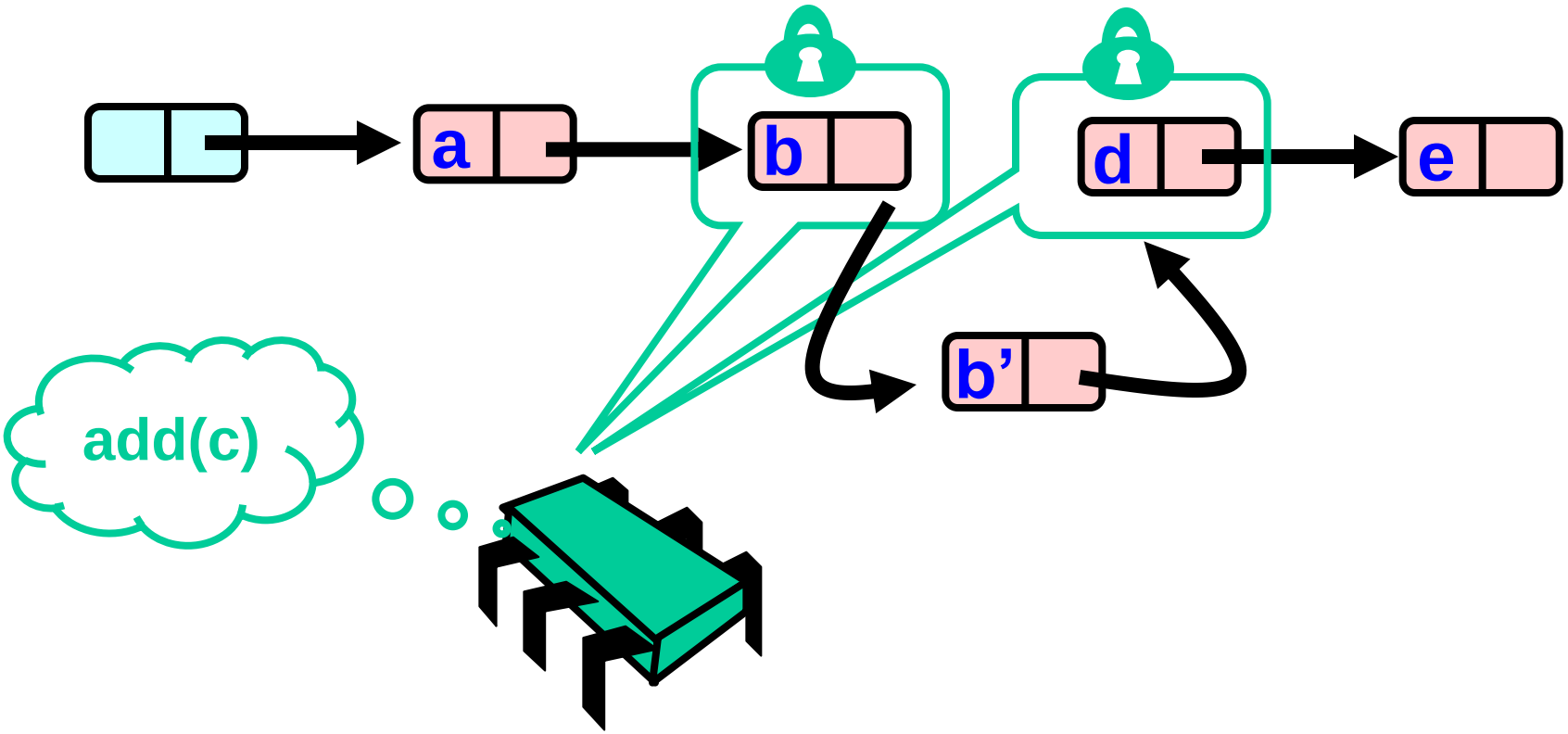


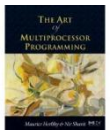
What Else Coould Go Wrong?



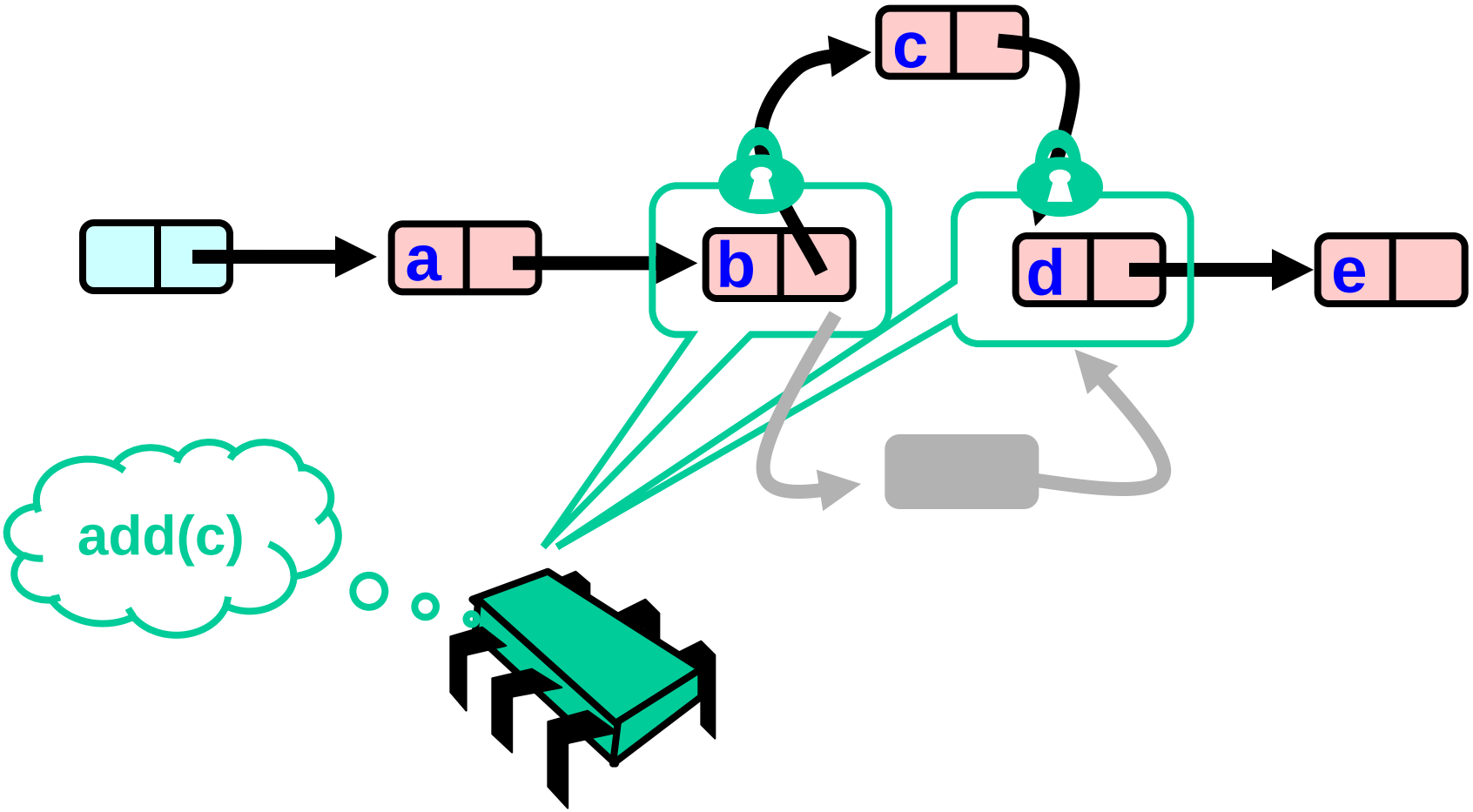


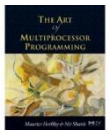
What Else Could Go Wrong?



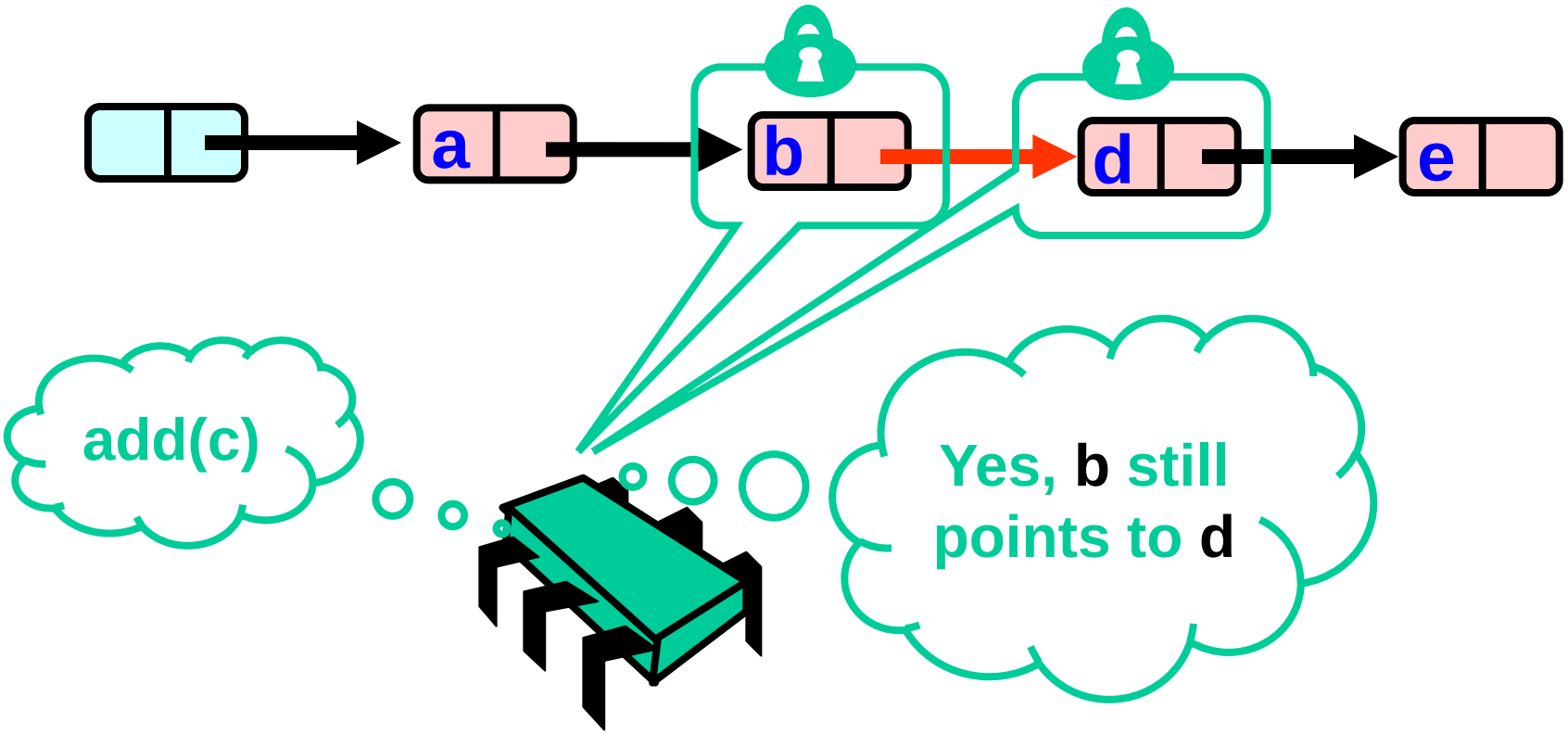


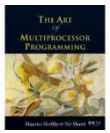
What Else Could Go Wrong?





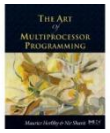
Validate Part 2 (while holding locks)





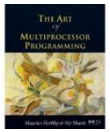
Invariants

- Careful: we may traverse deleted nodes
- But we establish properties by
  - Validation
  - After we lock target nodes

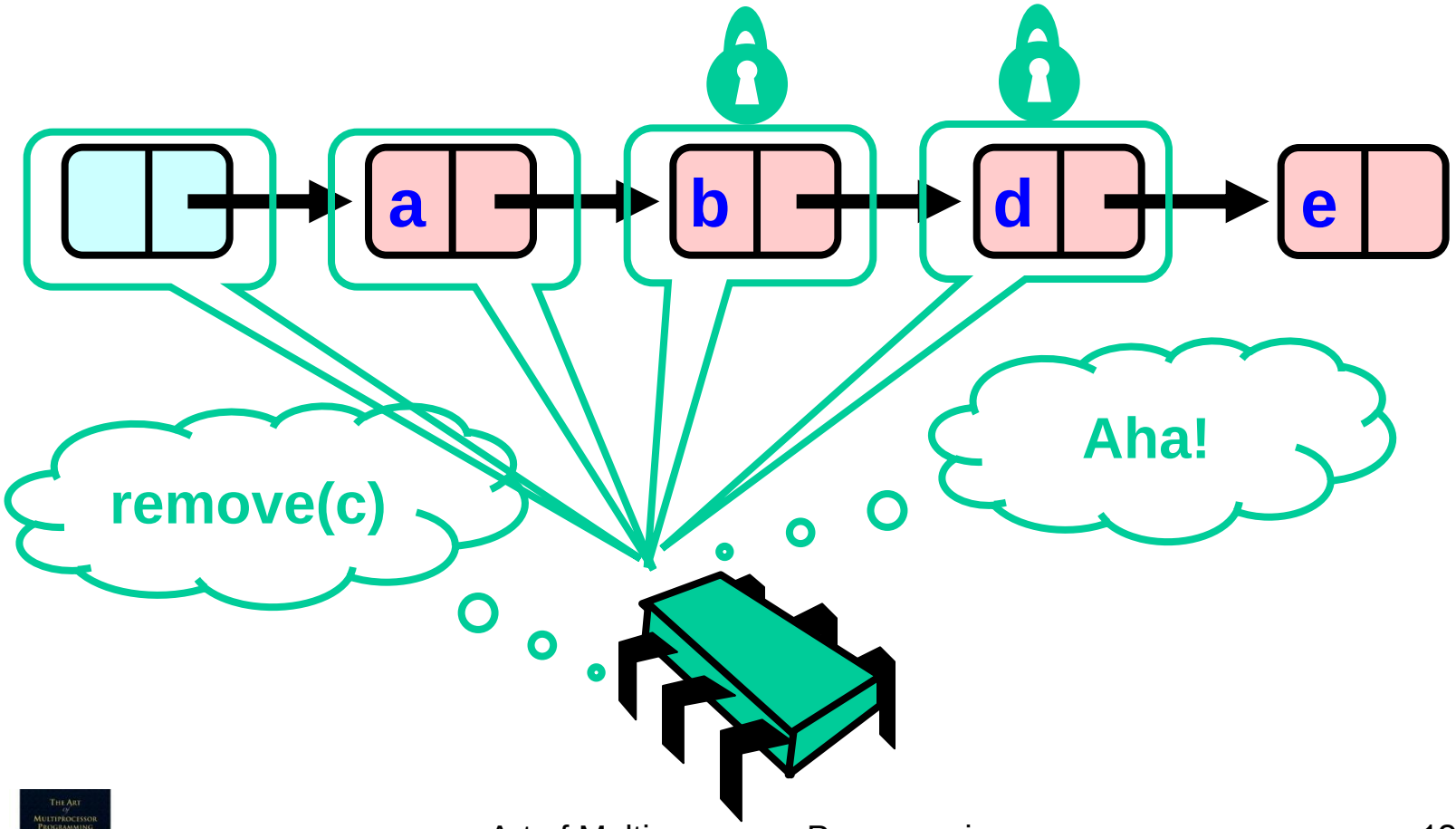


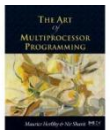
Correctness

- If
  - Nodes b and c both locked
  - Node b still accessible
  - Node c still successor to b
- Then
  - Neither will be deleted
  - OK to delete and return true

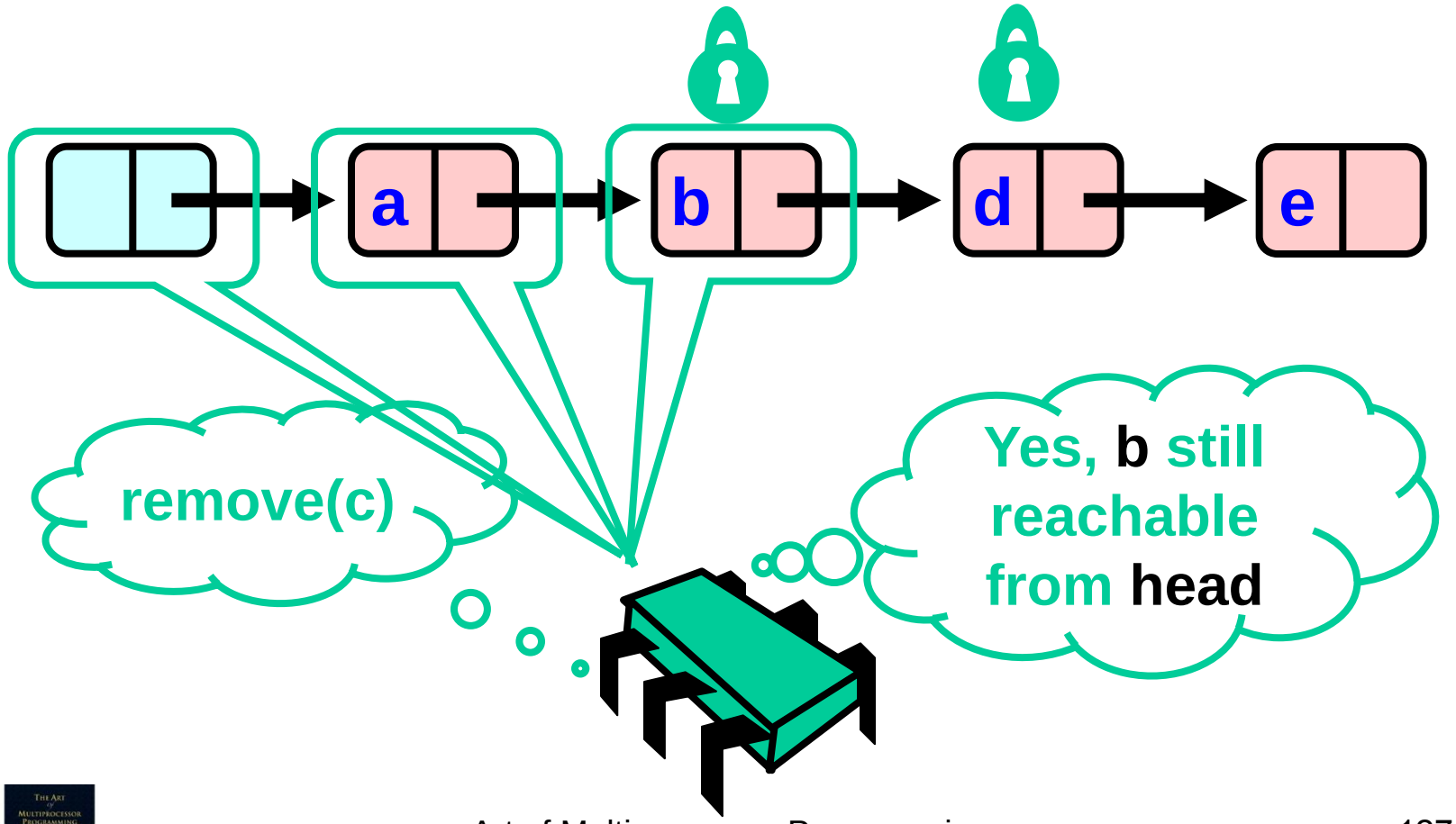


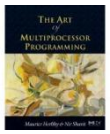
Unsuccessful Remove



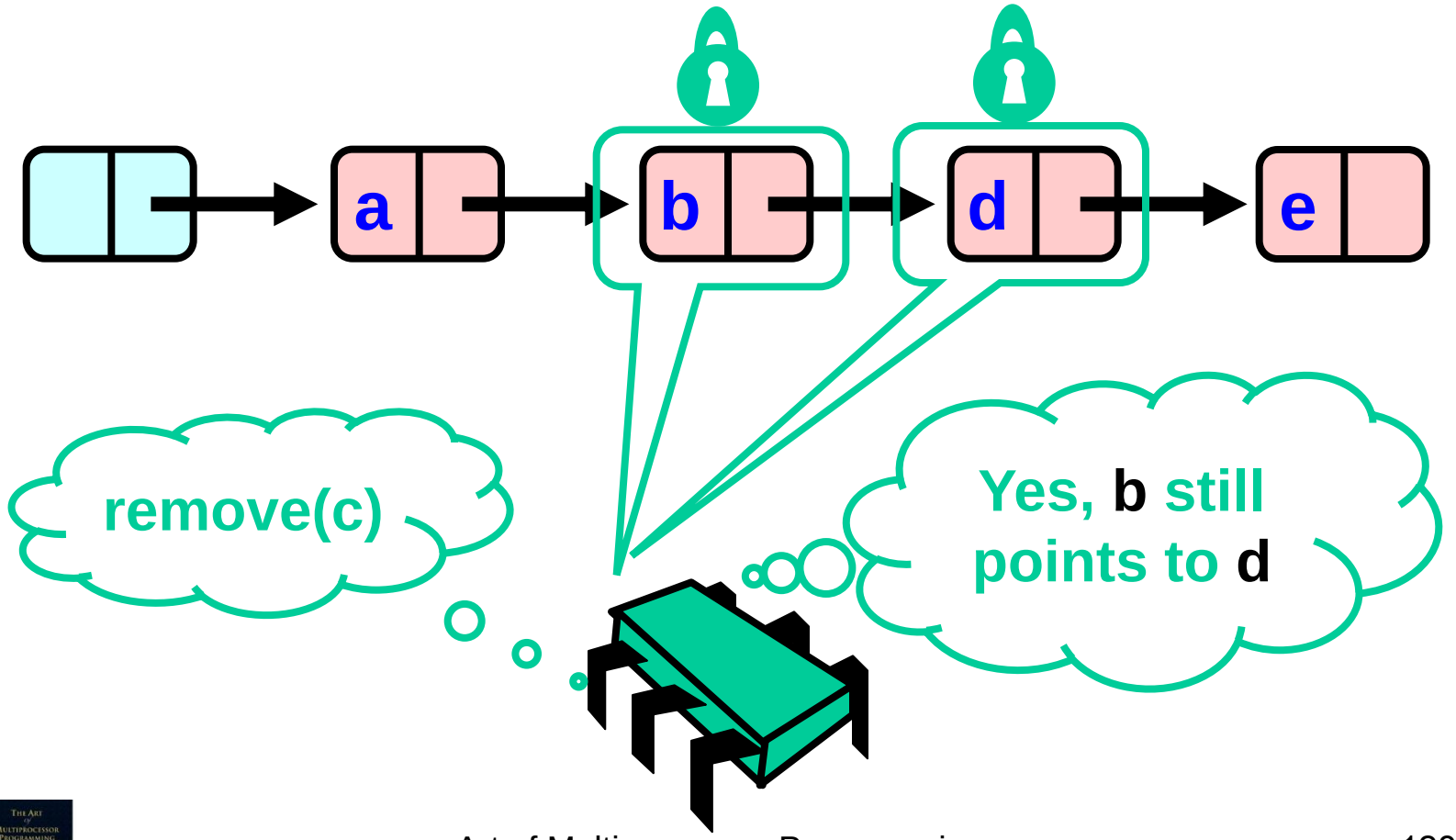


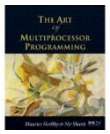
Validate (1)



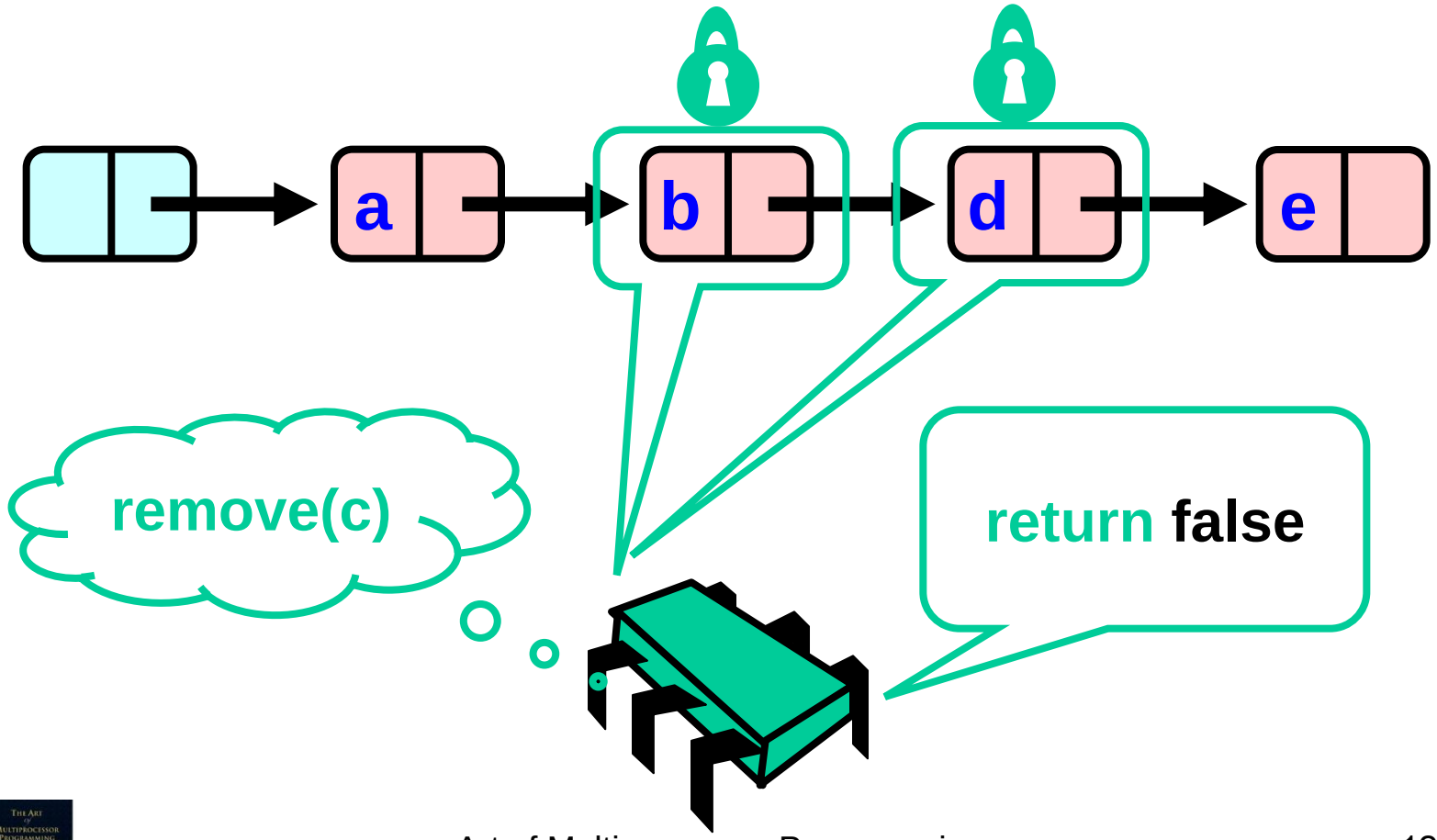


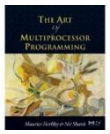
Validate (2)





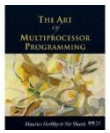
OK Computer





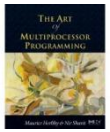
Correctness

- If
  - Nodes b and d both locked
  - Node b still accessible
  - Node d still successor to b
- Then
  - Neither will be deleted
  - No thread can add c after b
  - OK to return false



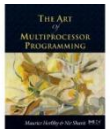
Optimistic List

- Limited hot-spots
  - Targets of add(), remove(), contains()
  - No contention on traversals
- Moreover
  - Traversals are wait-free
  - Food for thought …



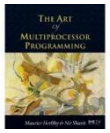
So Far, So Good

- Much less lock acquisition/release
  - Performance
  - Concurrency
- Problems
  - Need to traverse list twice
  - contains() method acquires locks



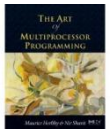
Evaluation

- Optimistic is effective if
  - cost of scanning twice without locks
    is less than
  - cost of scanning once with locks
- Drawback
  - contains() acquires locks
  - 90% of calls in many apps



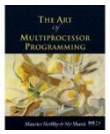
Lazy List

- Like optimistic, except
  - Scan once
  - **`contains(x)`** never locks …
- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”

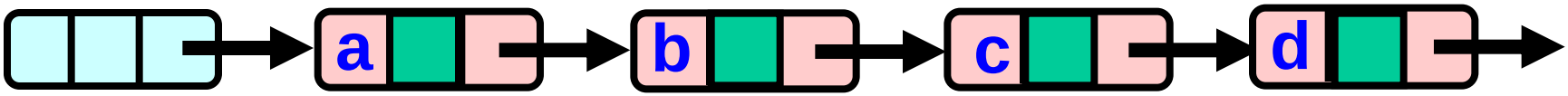


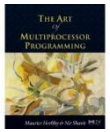
Lazy List

- `remove()`
  - Scans list (as before)
  - Locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

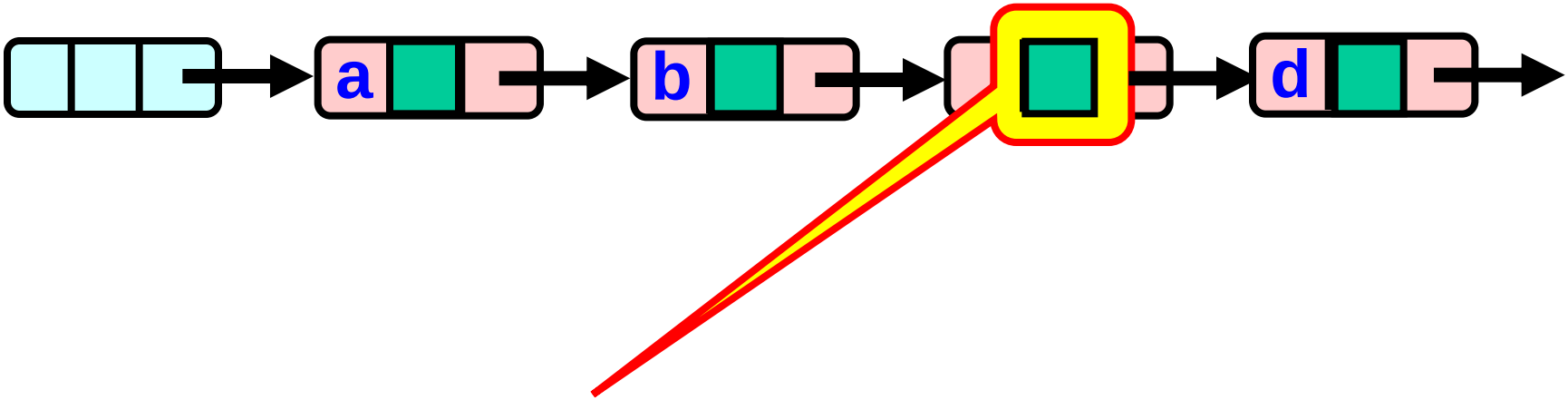


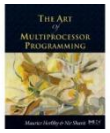
Lazy Removal



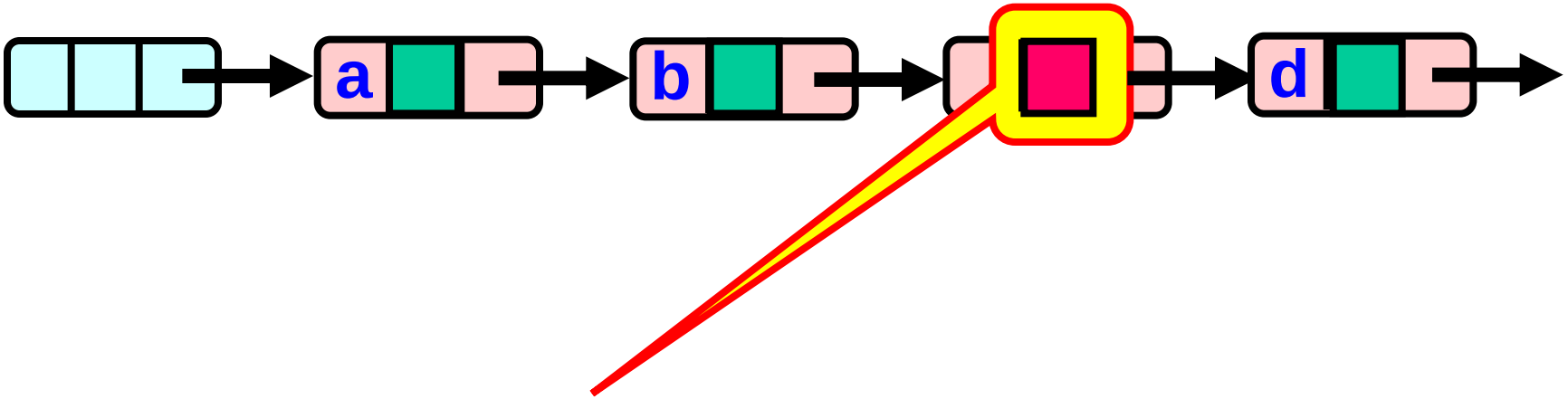


Lazy Removal

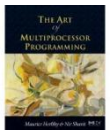




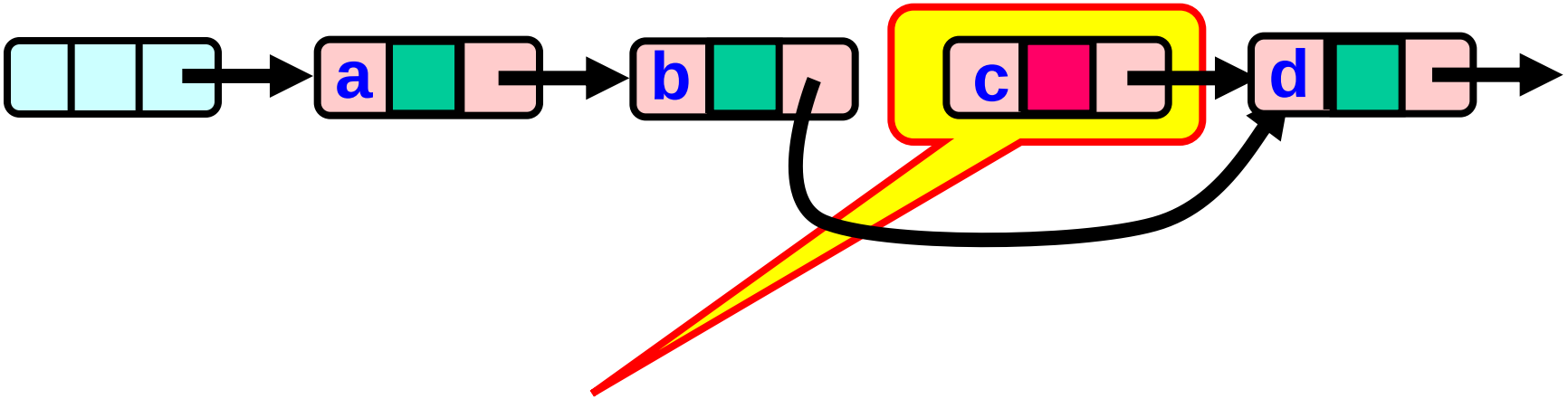
Lazy Removal



Logically deleted

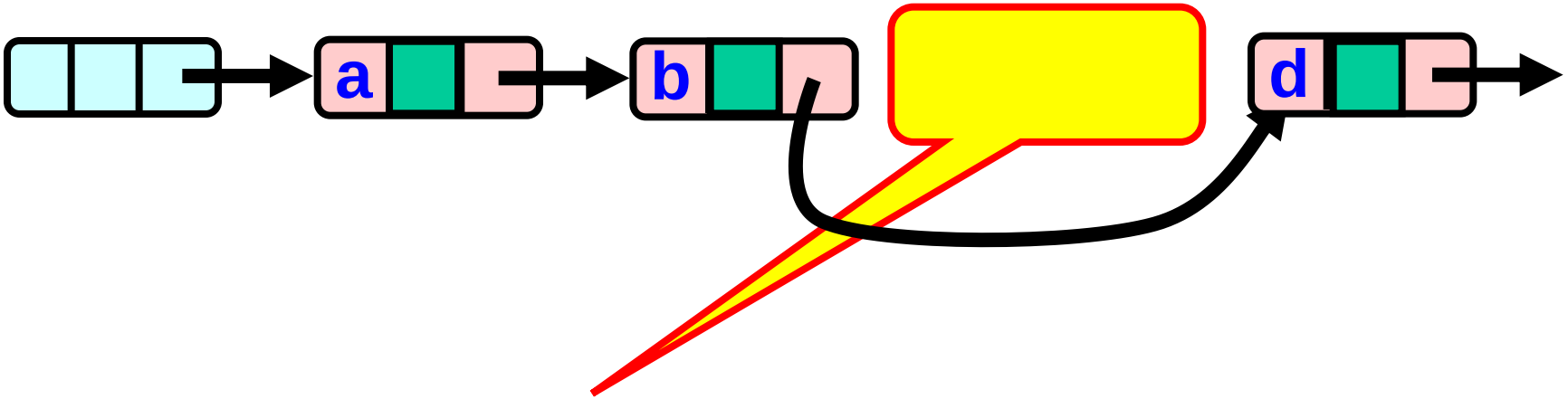


Lazy Removal



Physically deleted

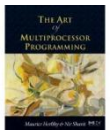
Lazy Removal



Physically deleted

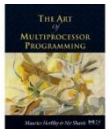
Lazy List

- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn't slow down other method calls …
- Must still lock pred and curr nodes.

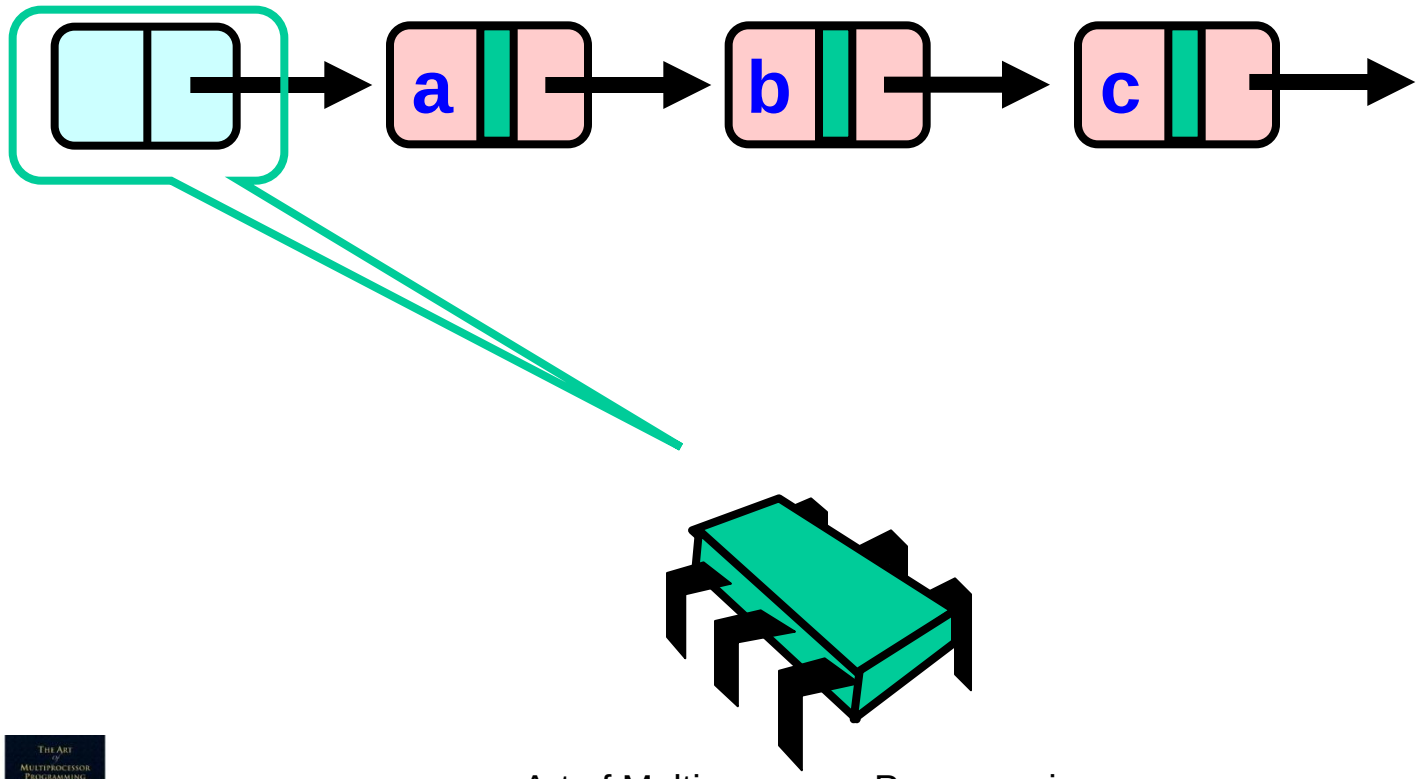


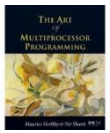
Validation

- No need to rescan list!
- Check that pred is not marked
- Check that curr is not marked
- Check that pred points to curr

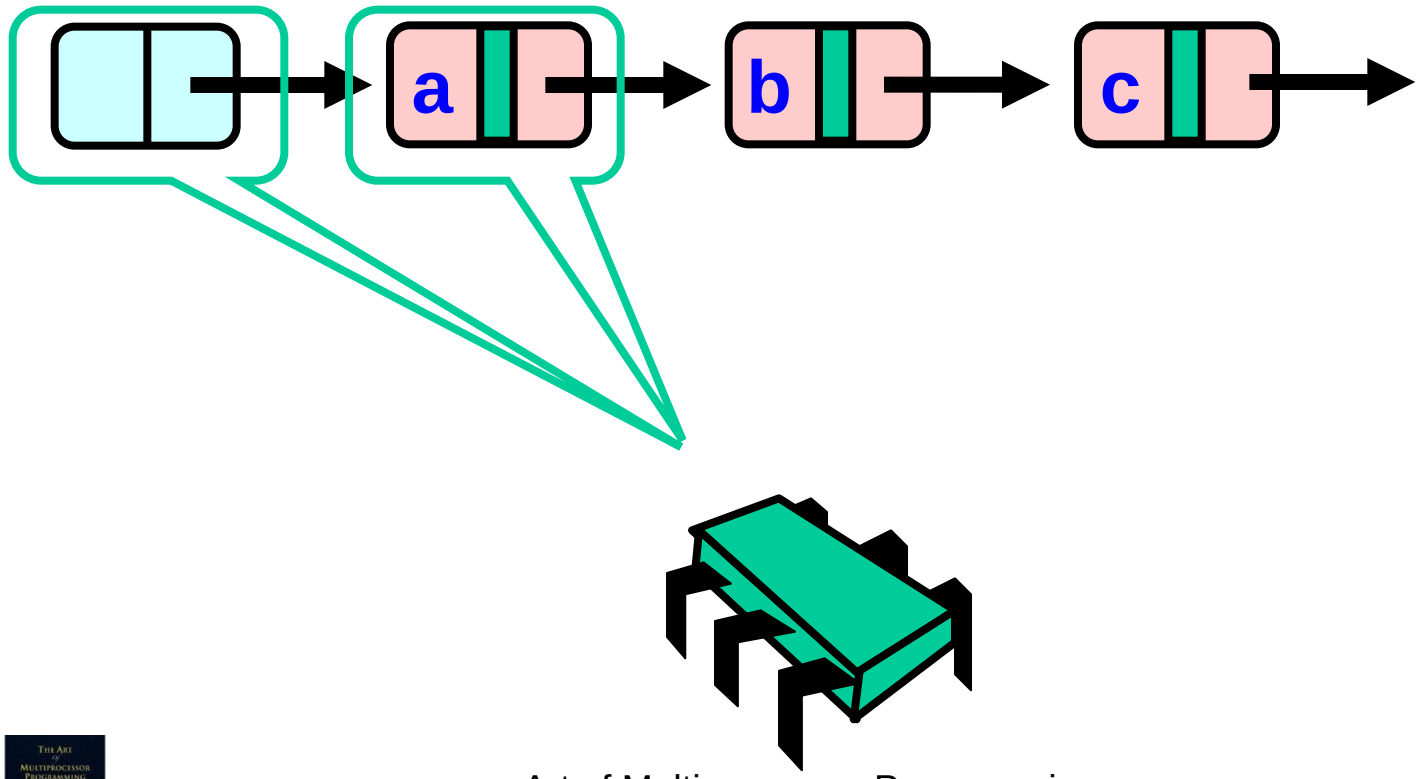


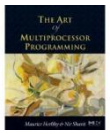
Business as Usual



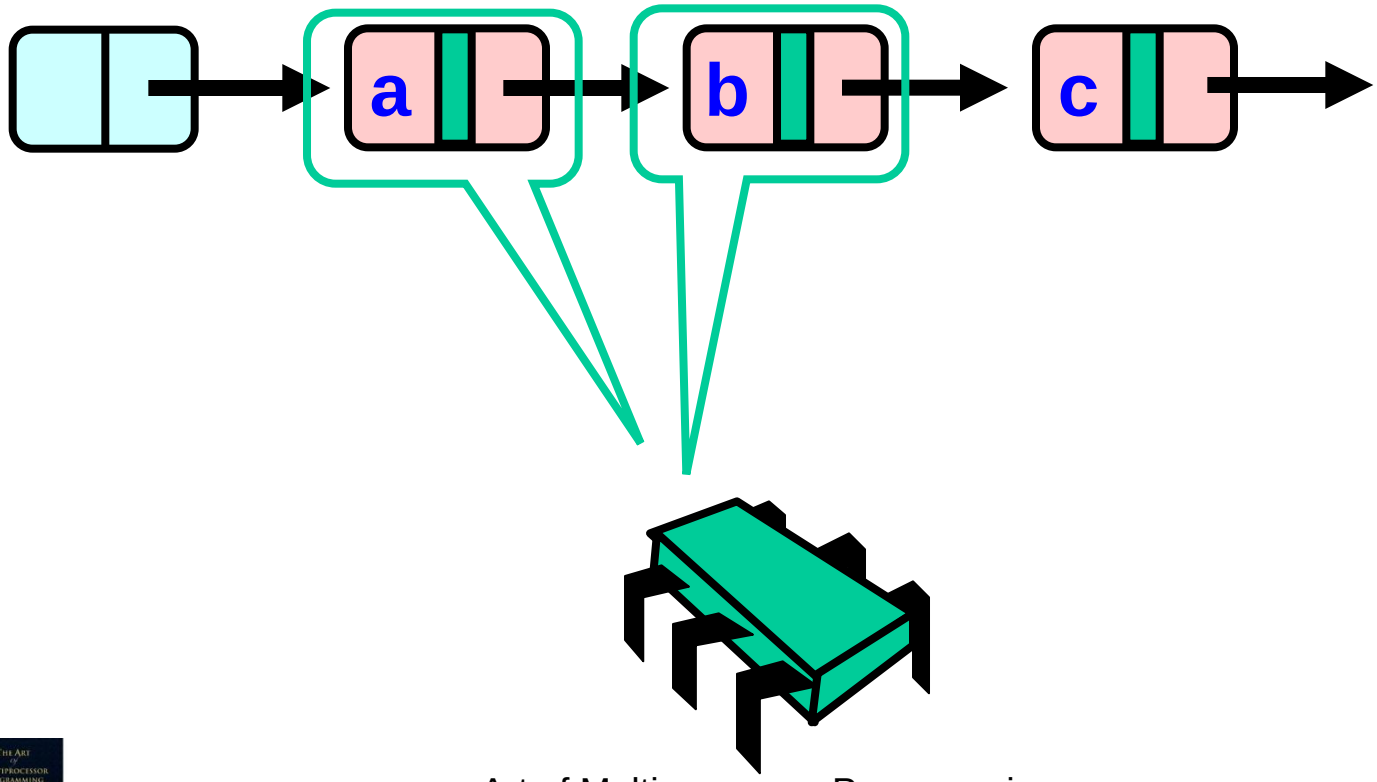


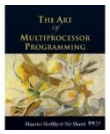
Business as Usual



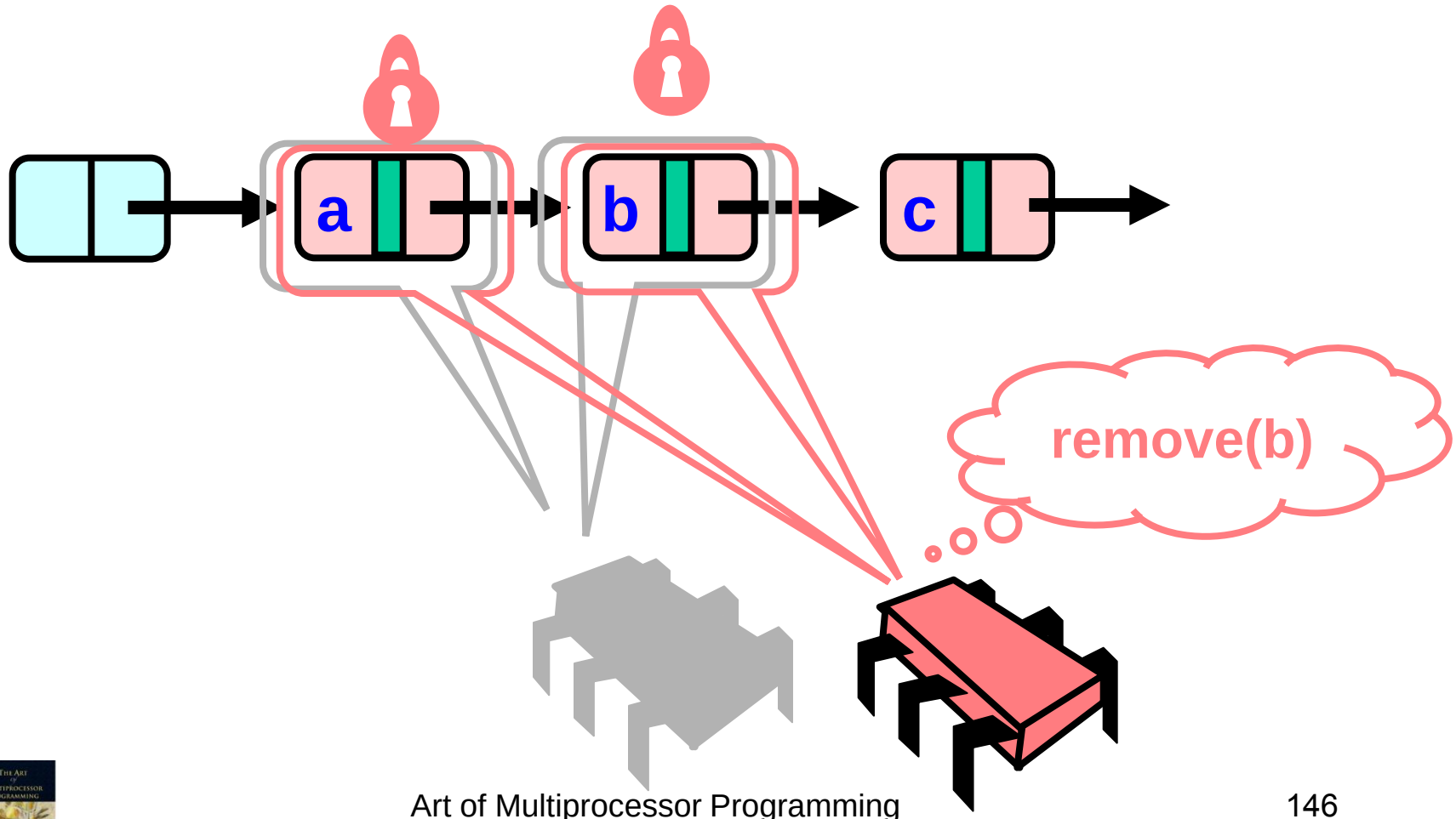


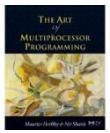
Business as Usual



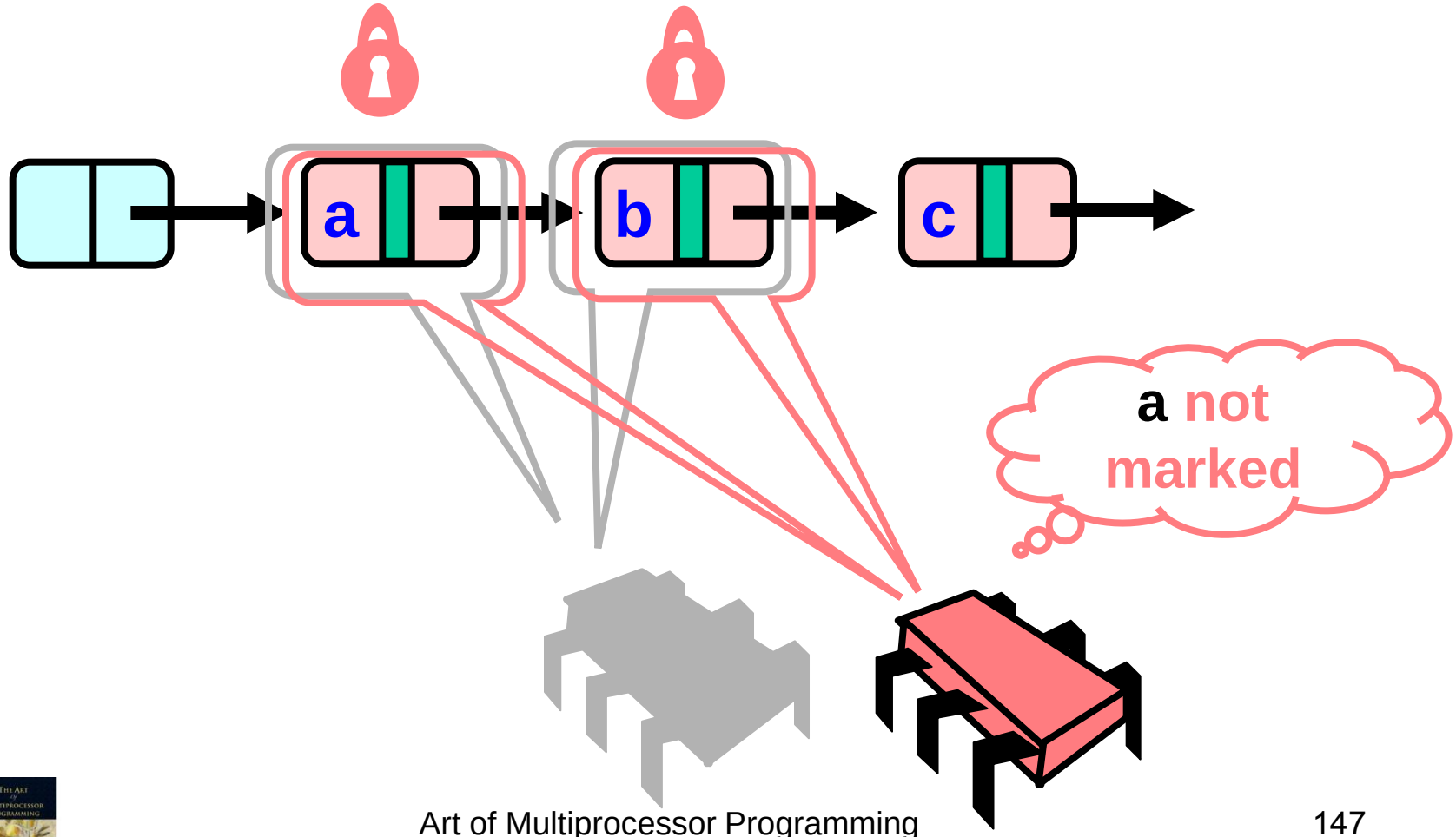


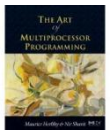
Business as Usual



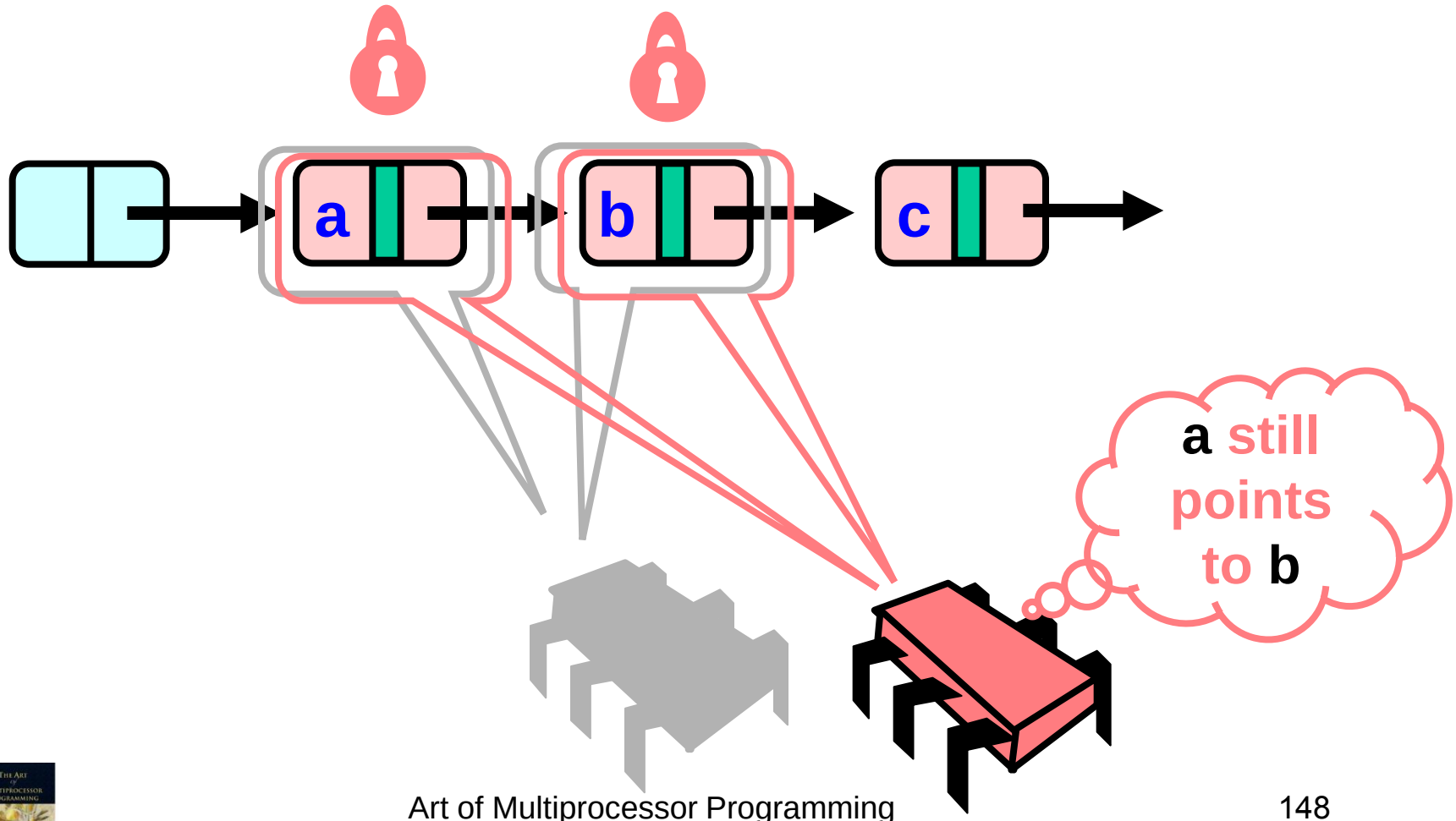


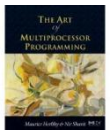
Business as Usual



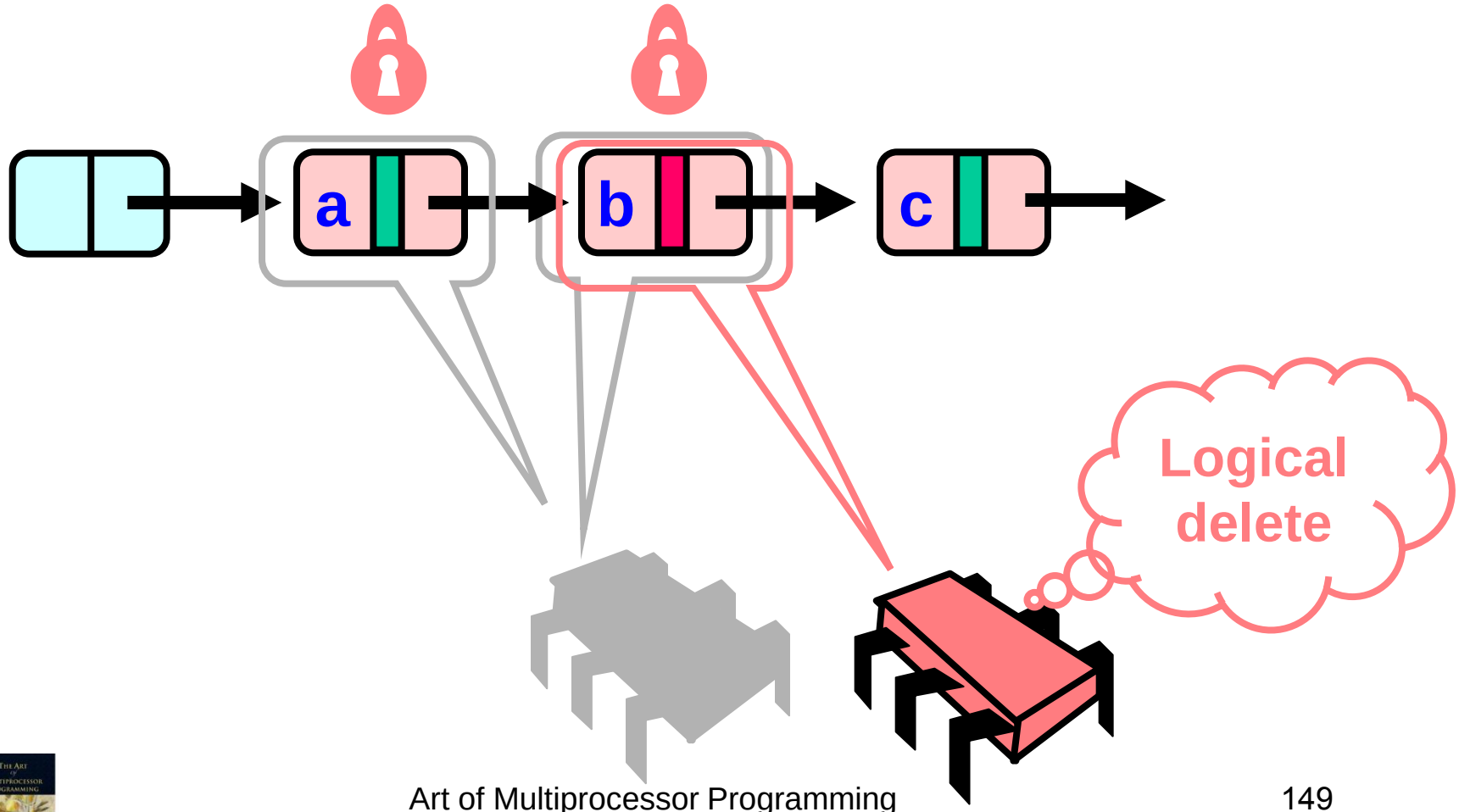


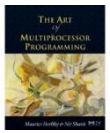
Business as Usual



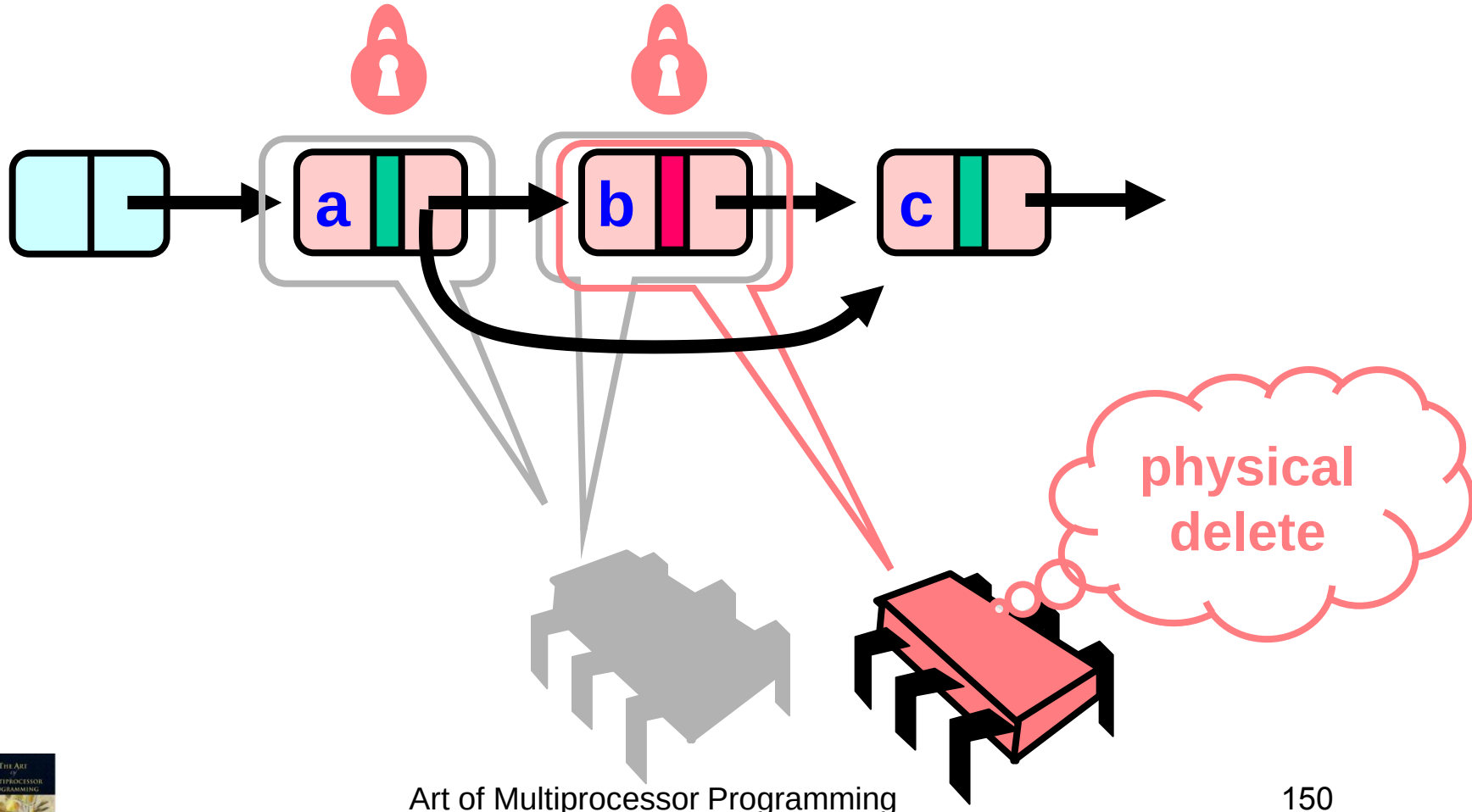


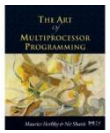
Business as Usual



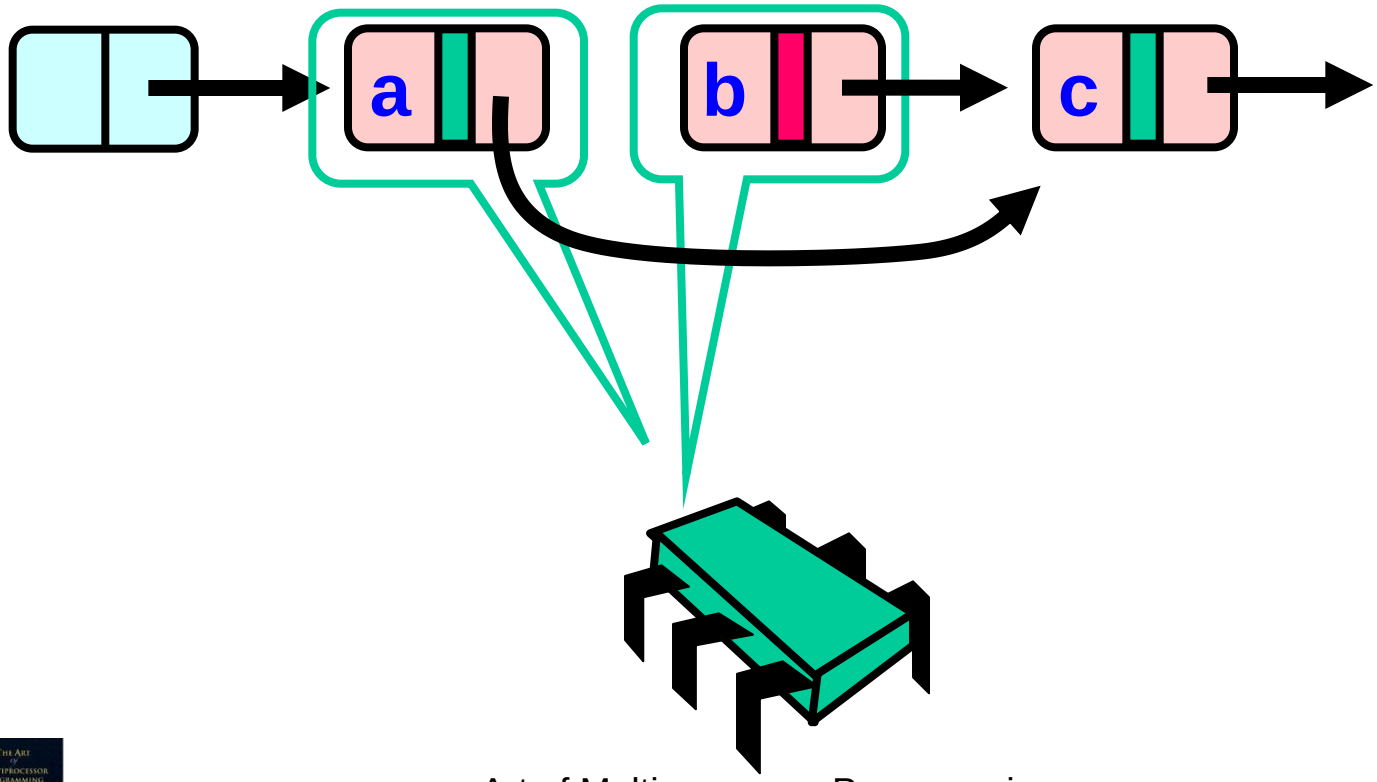


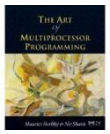
Business as Usual





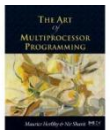
Business as Usual





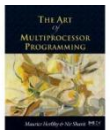
Invariant

- An item is in the set if
  - not marked
  - reachable from head
  - and if not yet traversed it is reachable from pred



Validation

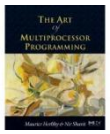
```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```



List Validate Method

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```

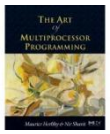
Predecessor not Logically removed



List Validate Method

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```

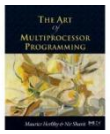
Current not Logically removed



List Validate Method

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```

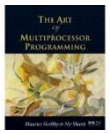
Predecessor still
Points to current



Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

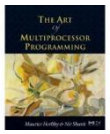
Start at the head



Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

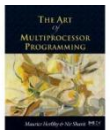
Search key range



Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

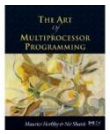
Traverse without locking
(nodes may have been removed)



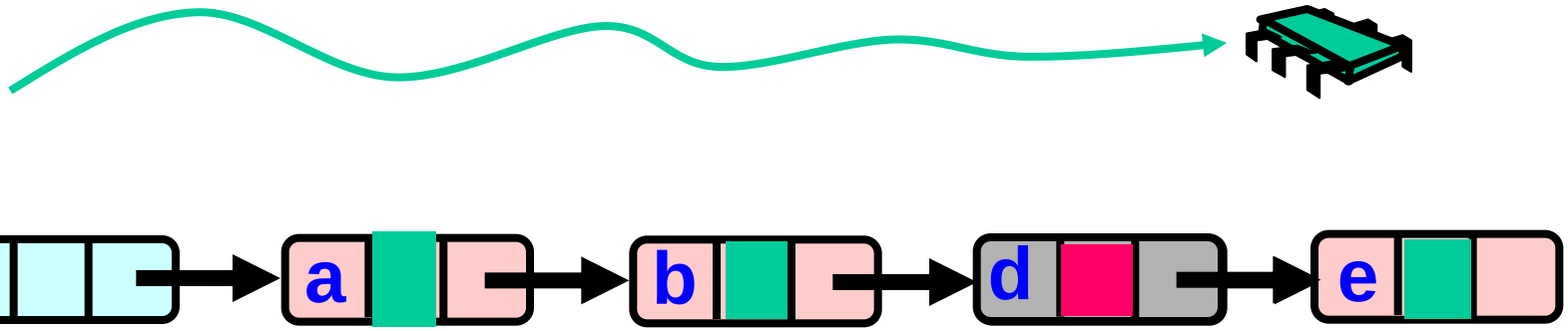
Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

Present and undeleted?

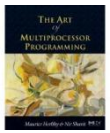


Summary: Wait-free Contains

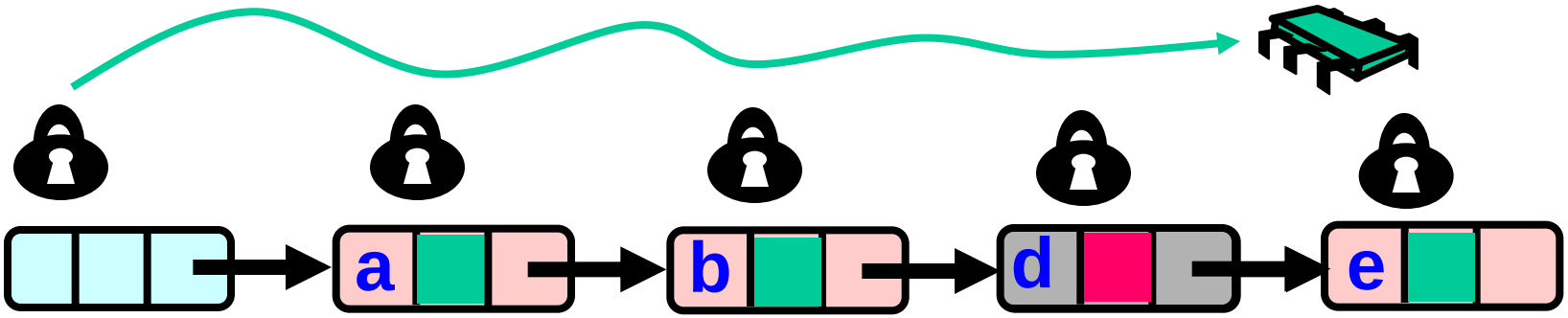


Use Mark bit + list ordering

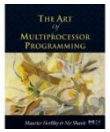
1. Not marked → in the set
2. Marked or missing → not in the set



Lazy List

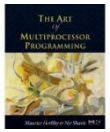


Lazy add() and remove() + Wait-free contains()



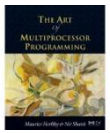
Evaluation

- Good:
  - contains() doesn't lock
  - In fact, its wait-free!
  - Good because typically high % contains()
  - Uncontended calls don't re-traverse
- Bad
  - Contended add() and remove() calls do re-traverse
  - Traffic jam if one thread delays



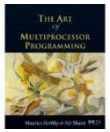
Traffic Jam

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And “eats the big muffin”
    - Cache miss, page fault, descheduled …
  - Everyone else using that lock is stuck!
  - Need to trust the scheduler….



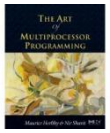
Reminder: Lock-Free Data Structures

- No matter what …
  - Guarantees minimal progress in any execution
  - i.e. Some thread will always complete a method call
  - Even if others halt at malicious times
  - Implies that implementation can't use locks



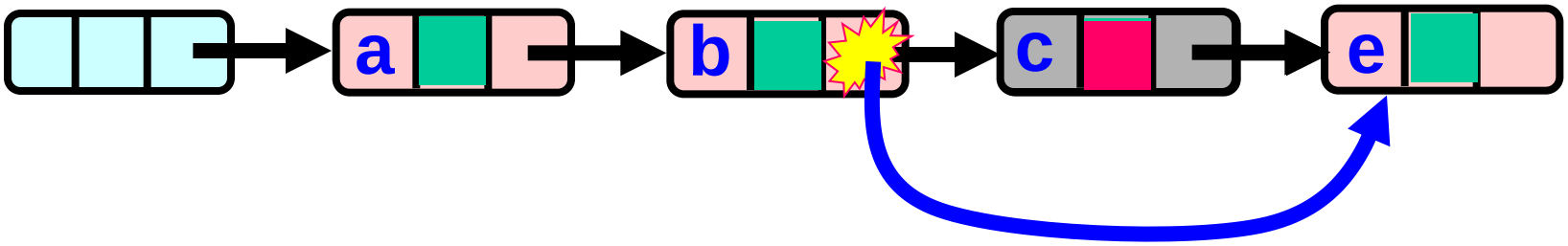
Lock-free Lists

- Next logical step
  - Wait-free contains()
  - lock-free add() and remove()
- Use only compareAndSet()
  - What could go wrong?



Lock-free Lists

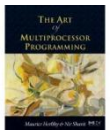
Logical Removal



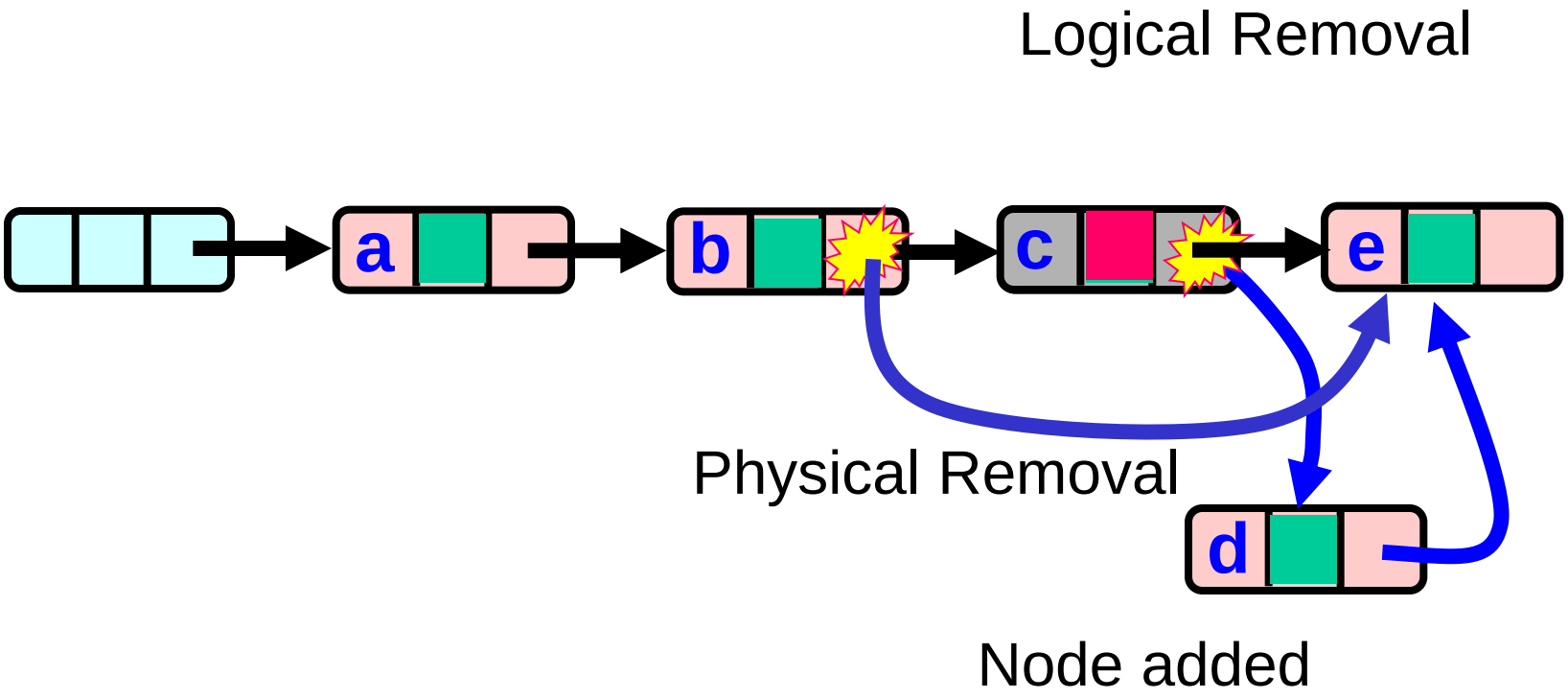
Physical Removal

Use CAS to verify pointer is correct

Not enough!

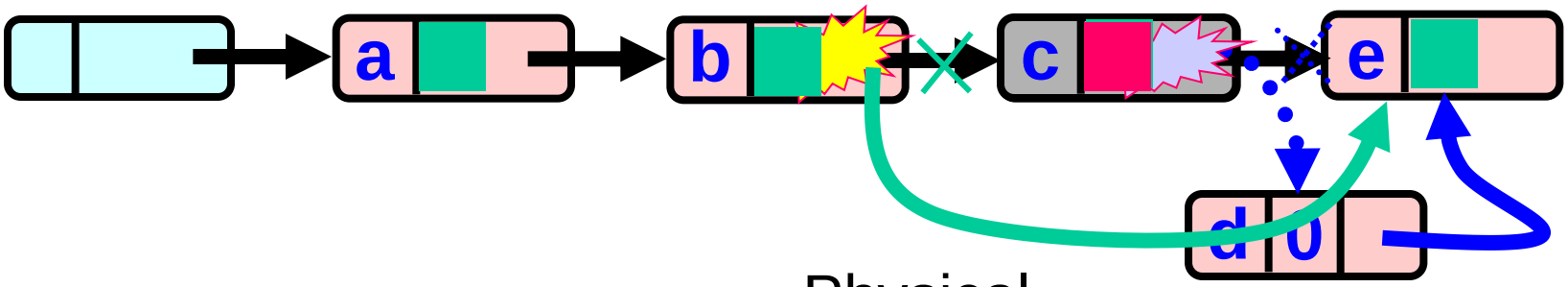


Problem…



The Solution: Combine Bit and Pointer

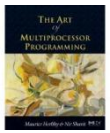
Logical Removal =
Set Mark Bit



Physical Removal CAS

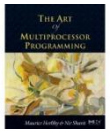
Failed CAS: Node not added after logical Removal

Mark-Bit and Pointer are CASed as one (`AtomicMarkableReference`)

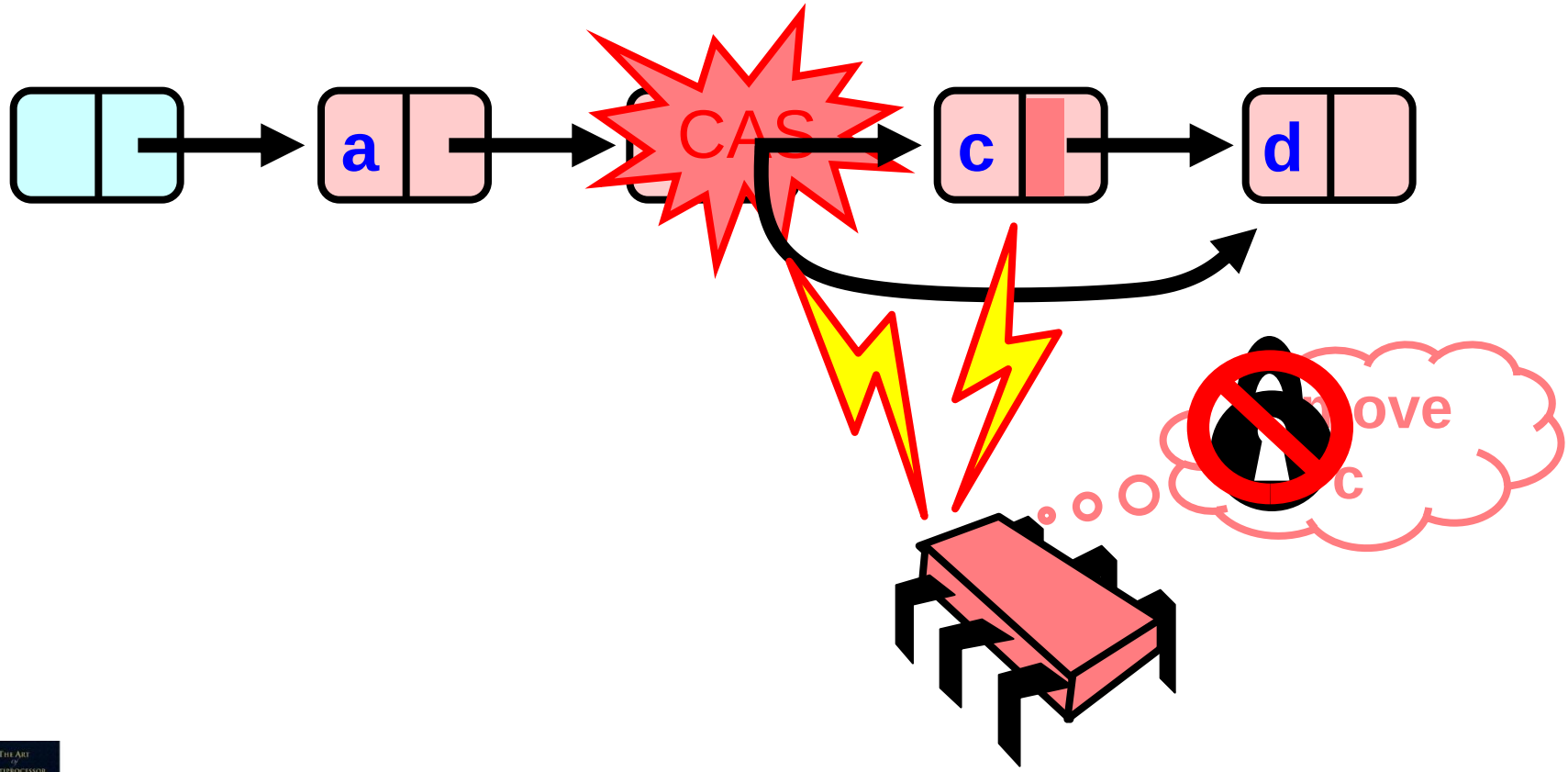


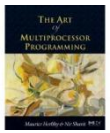
Solution

- Use AtomicMarkableReference
- Atomically
  - Swing reference and
  - Update flag
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor's pointer

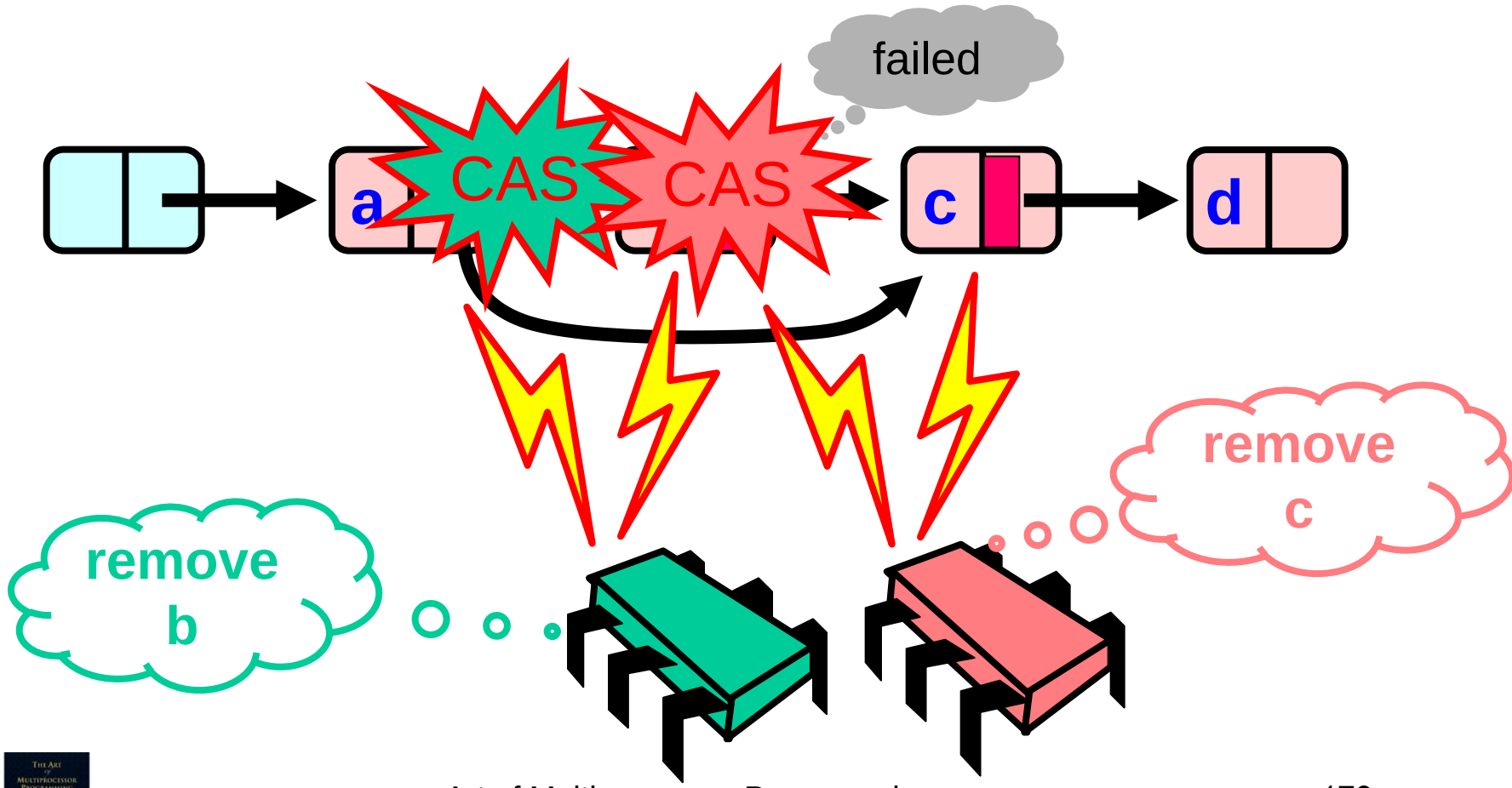


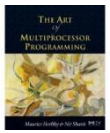
Removing a Node



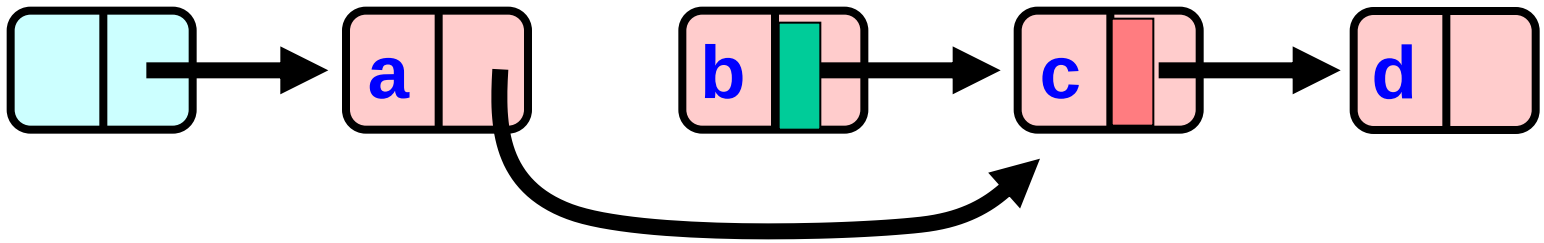


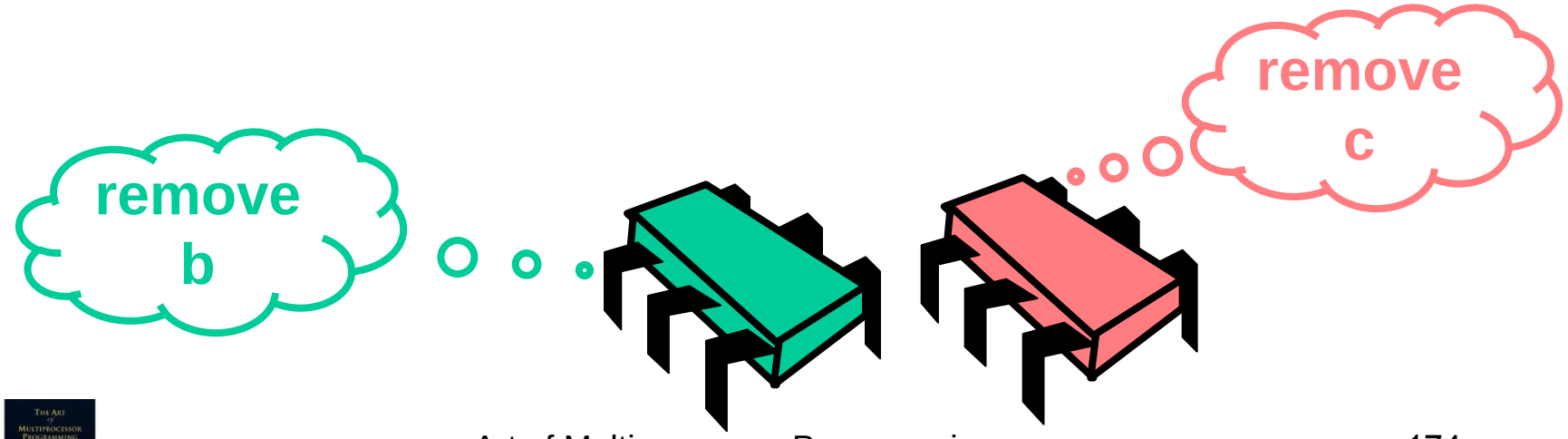
Removing a Node

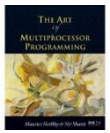




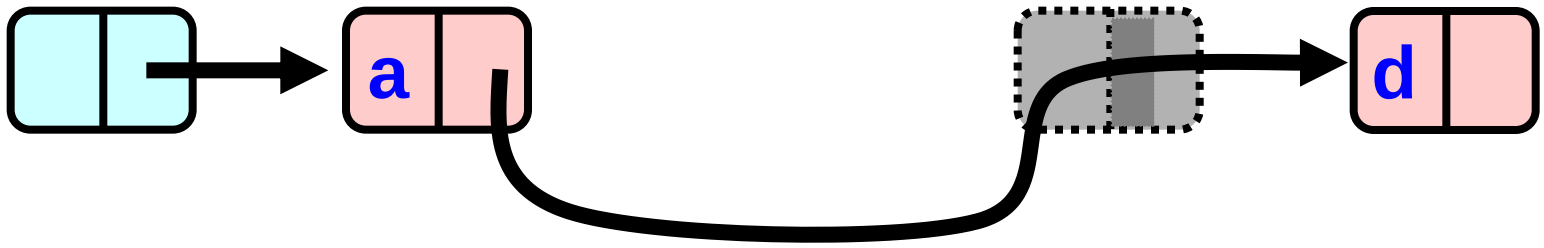
Removing a Node

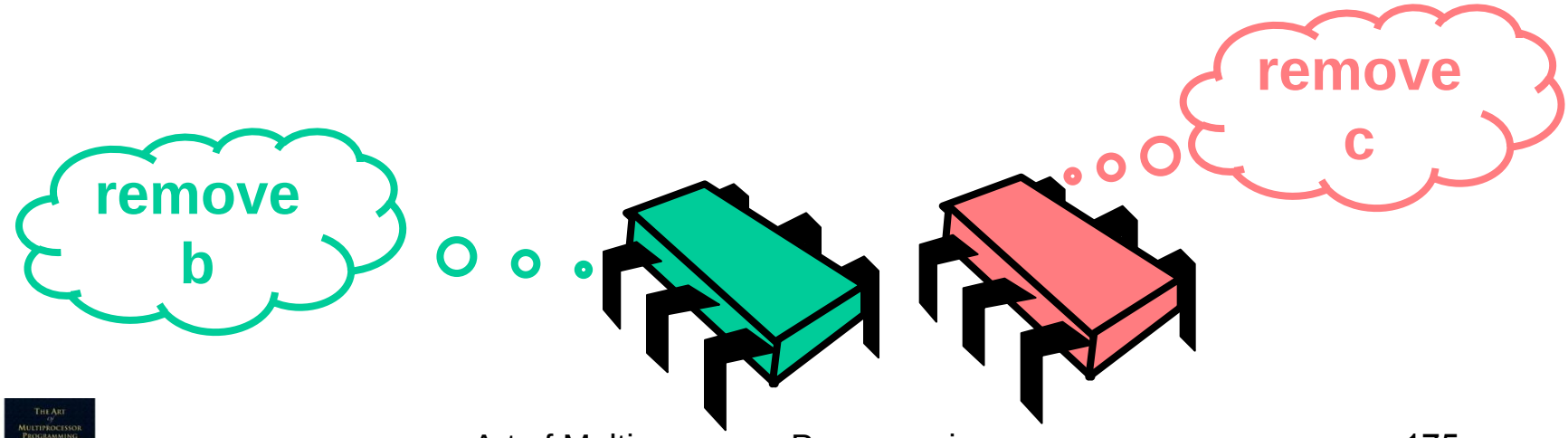


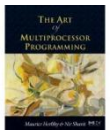




Removing a Node

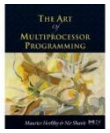




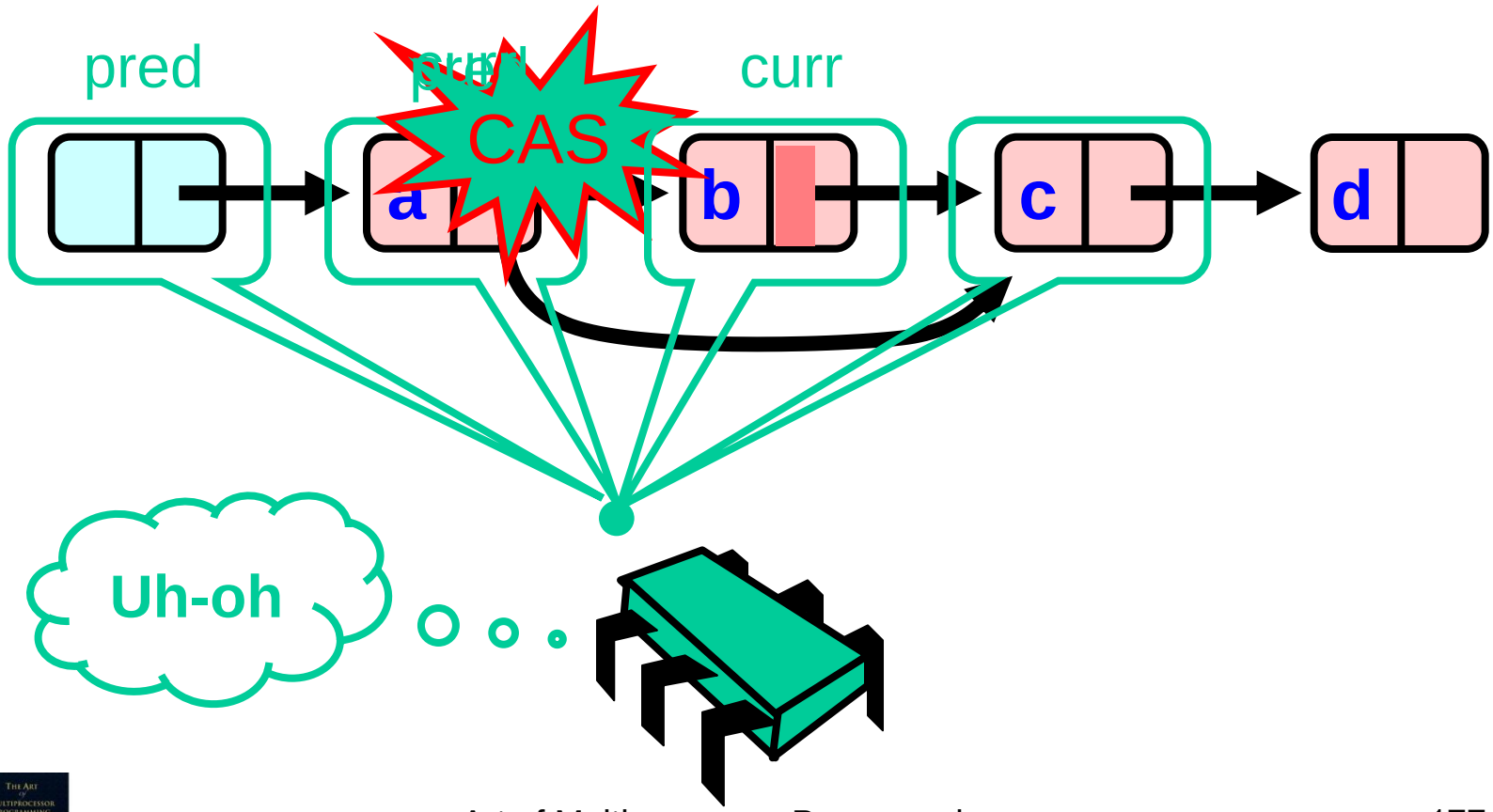


Traversing the List

- Q: what do you do when you find a "logically" deleted node in your path?
- A: finish the job.
  - CAS the predecessor's next field
  - Proceed (repeat as needed)

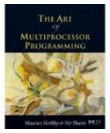


Lock-Free Traversal (only Add and Remove)

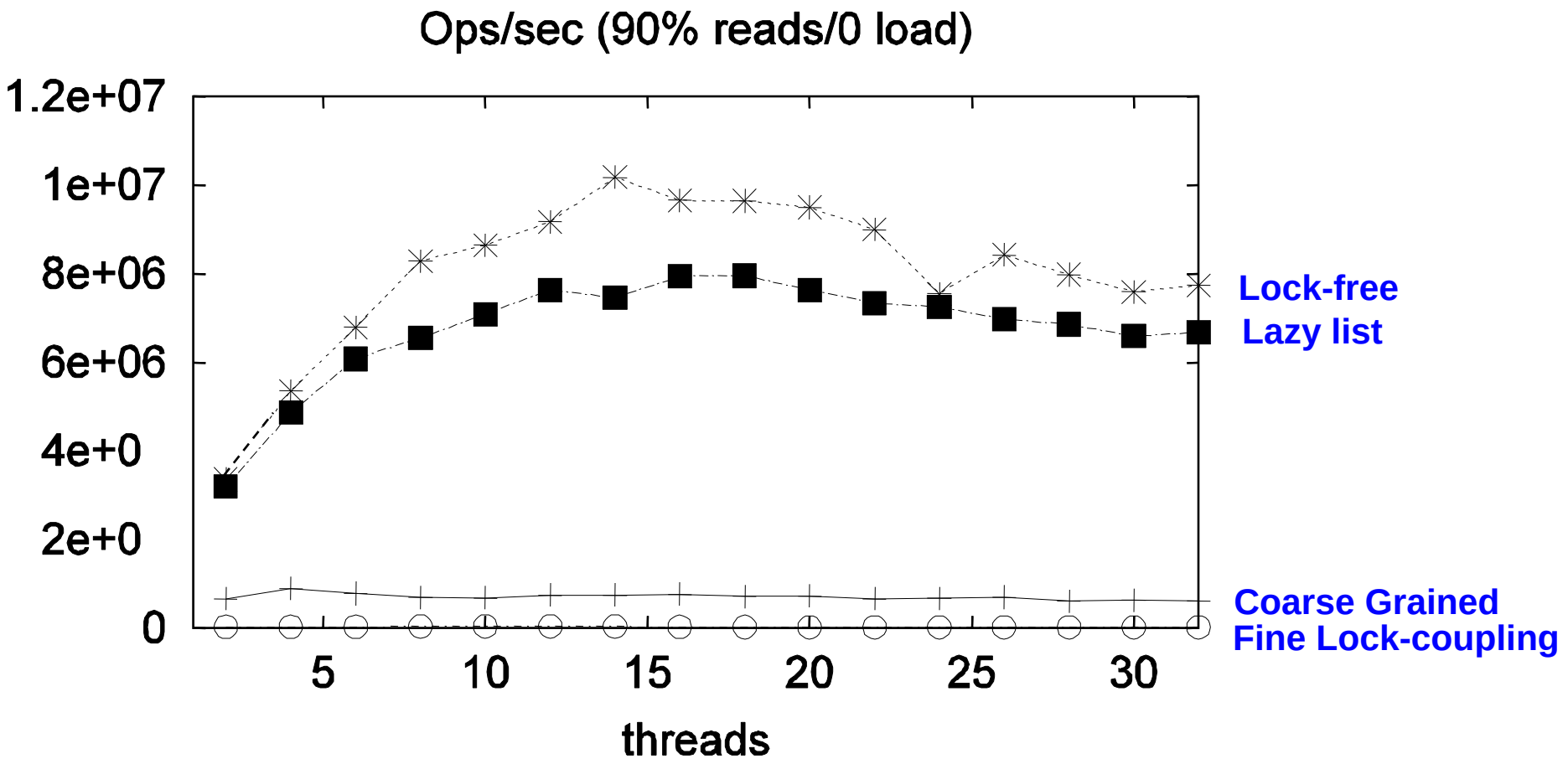


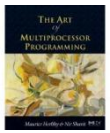
Performance

On 16 node shared memory machine
Benchmark throughput of Java List-based Set algs. Vary % of Contains() method Calls.

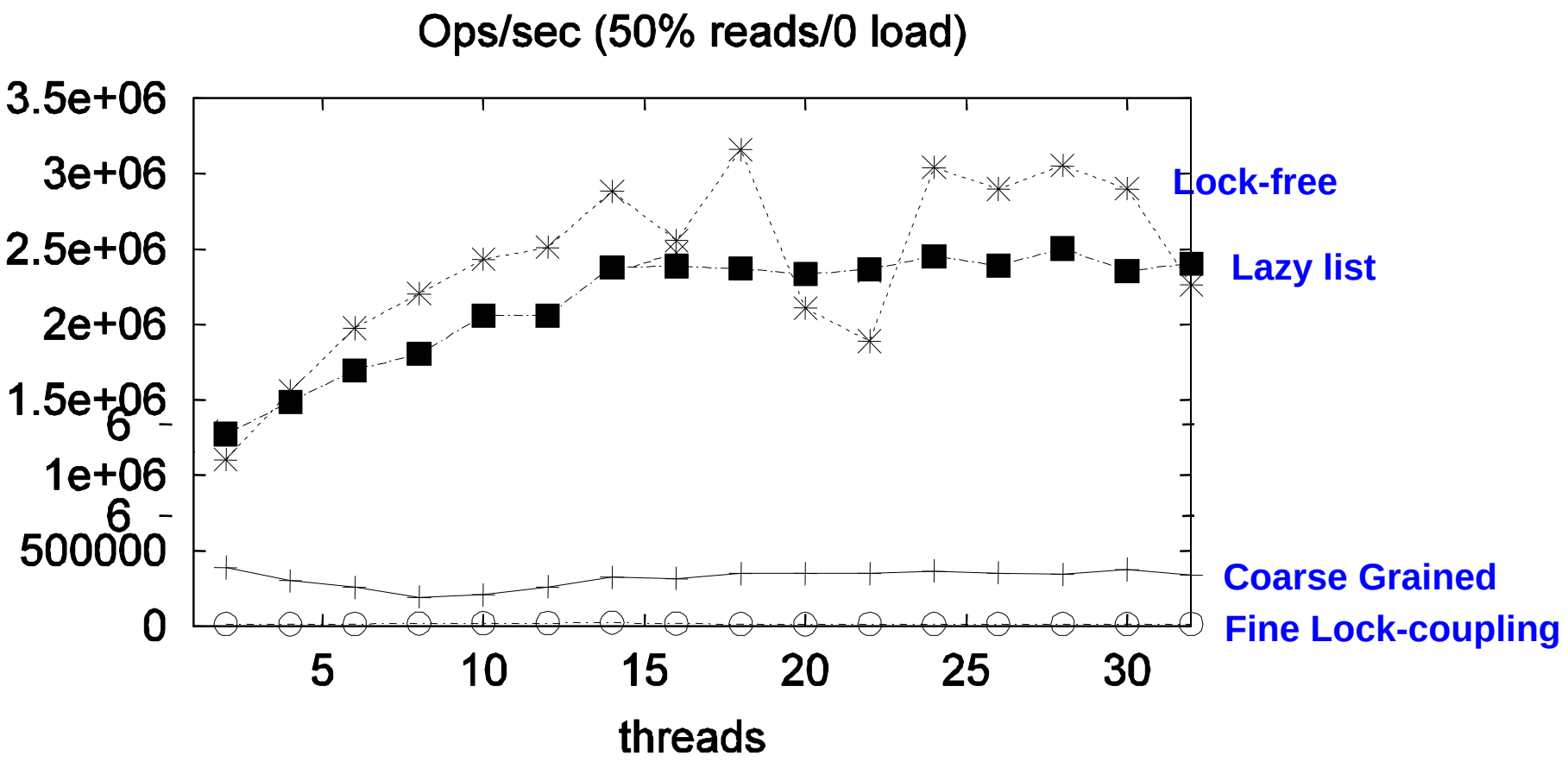


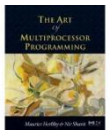
High Contains Ratio



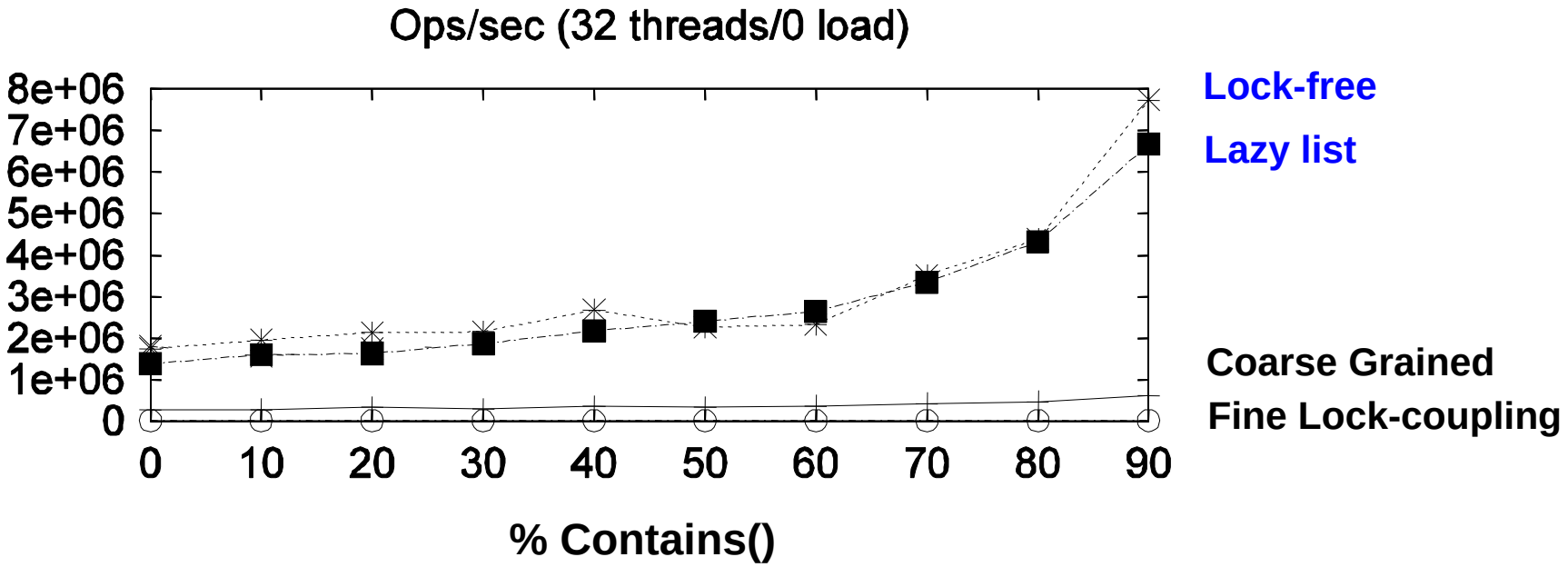


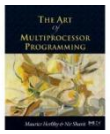
Low Contains Ratio





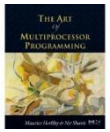
As Contains Ratio Increases





Summary

- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization
- Lock-free synchronization



“To Lock or Not to Lock”

- Locking vs. Non-blocking: Extremist views on both sides
- The answer: nobler to compromise, combine locking and non-blocking
  - Example: Lazy list combines blocking add() and remove() and a wait-free contains()
  - Remember: Blocking/non-blocking is a property of a method