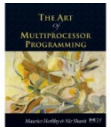
Coarse-grained and fine-grained locking

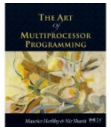
Niklas Fors 2013-12-05

Slides borrowed from:
http://cs.brown.edu/courses/cs176course_information.shtml



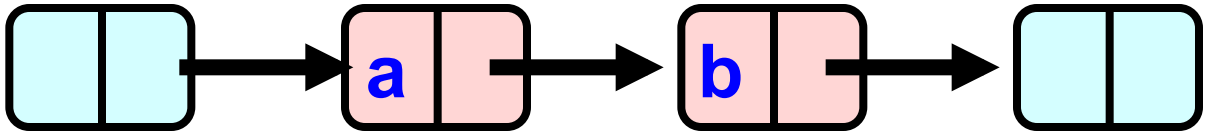
Topics discussed

- Coarse-grained locking
  - One lock
- Fine-grained locking
  - More than one lock

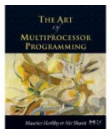


Abstract Data Types

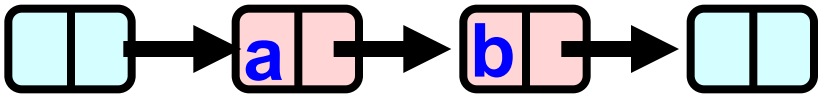
- Concrete representation:

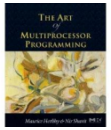


- Abstract Type:
  - {**a**, **b**}

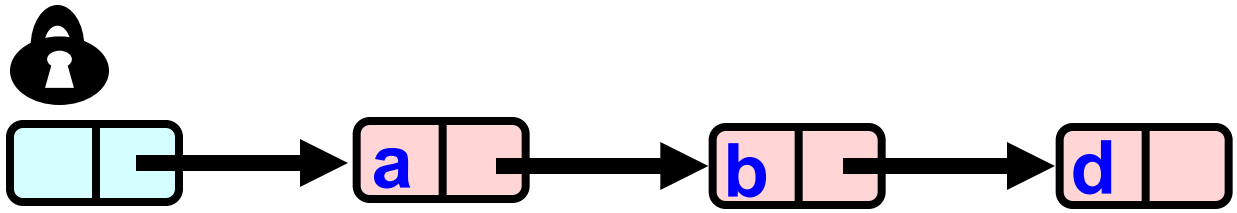


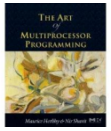
Abstract Data Types

- Meaning of rep given by abstraction map
  - S() = {a,b}

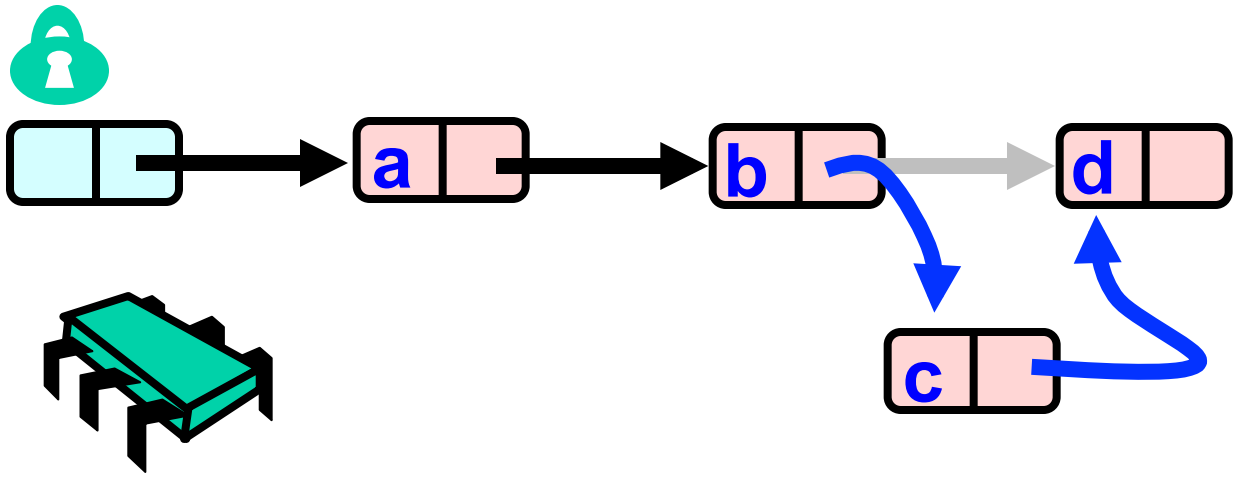


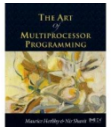
Coarse-Grained Locking



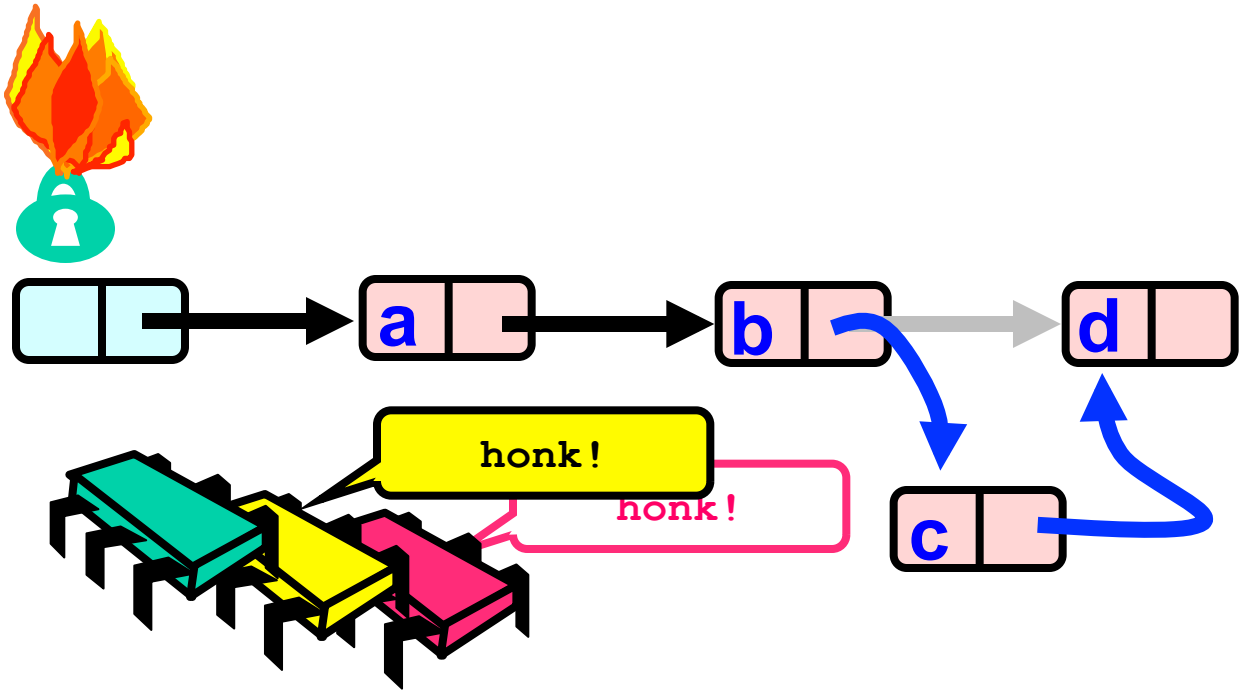


Coarse-Grained Locking

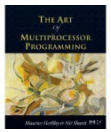




Coarse-Grained Locking

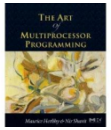


Simple but hotspot + bottleneck



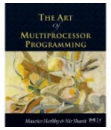
Coarse-Grained Locking

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



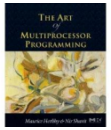
Coarse-Grained Locking

- Easy, same as synchronized methods
  - "One lock to rule them all …"
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention

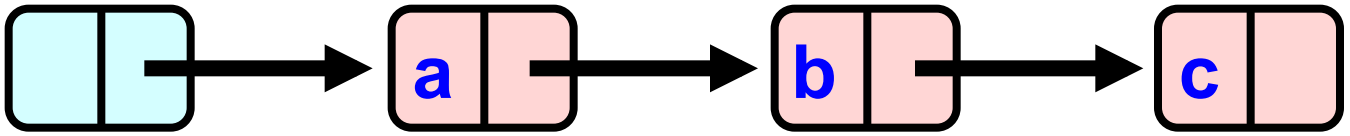


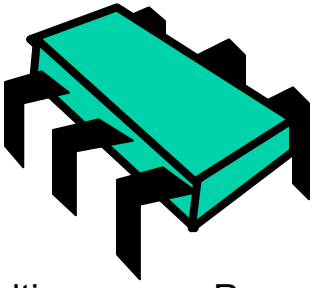
Fine-grained Locking

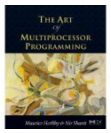
- Requires **careful** thought
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other



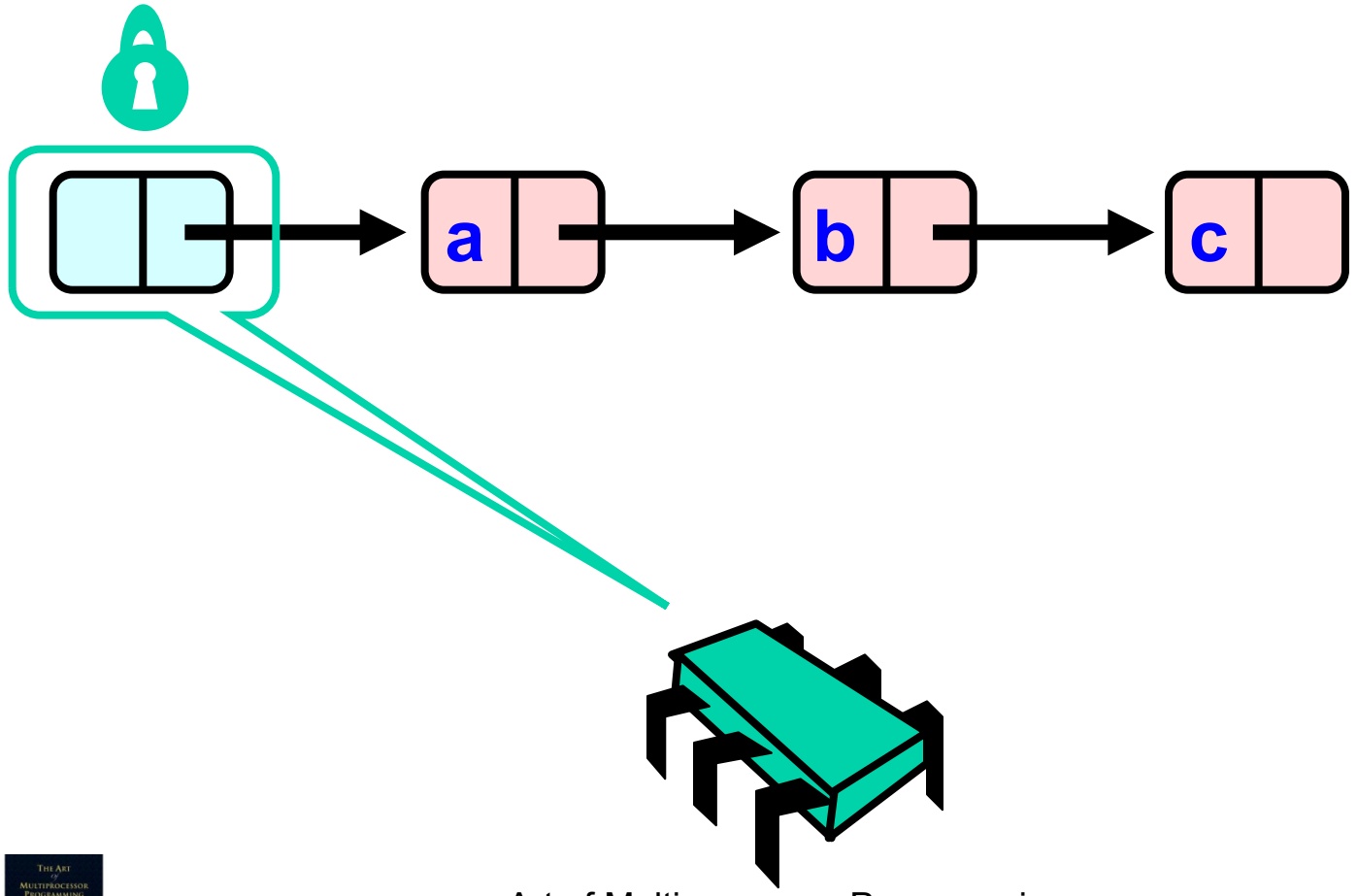
Hand-over-Hand locking

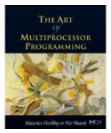




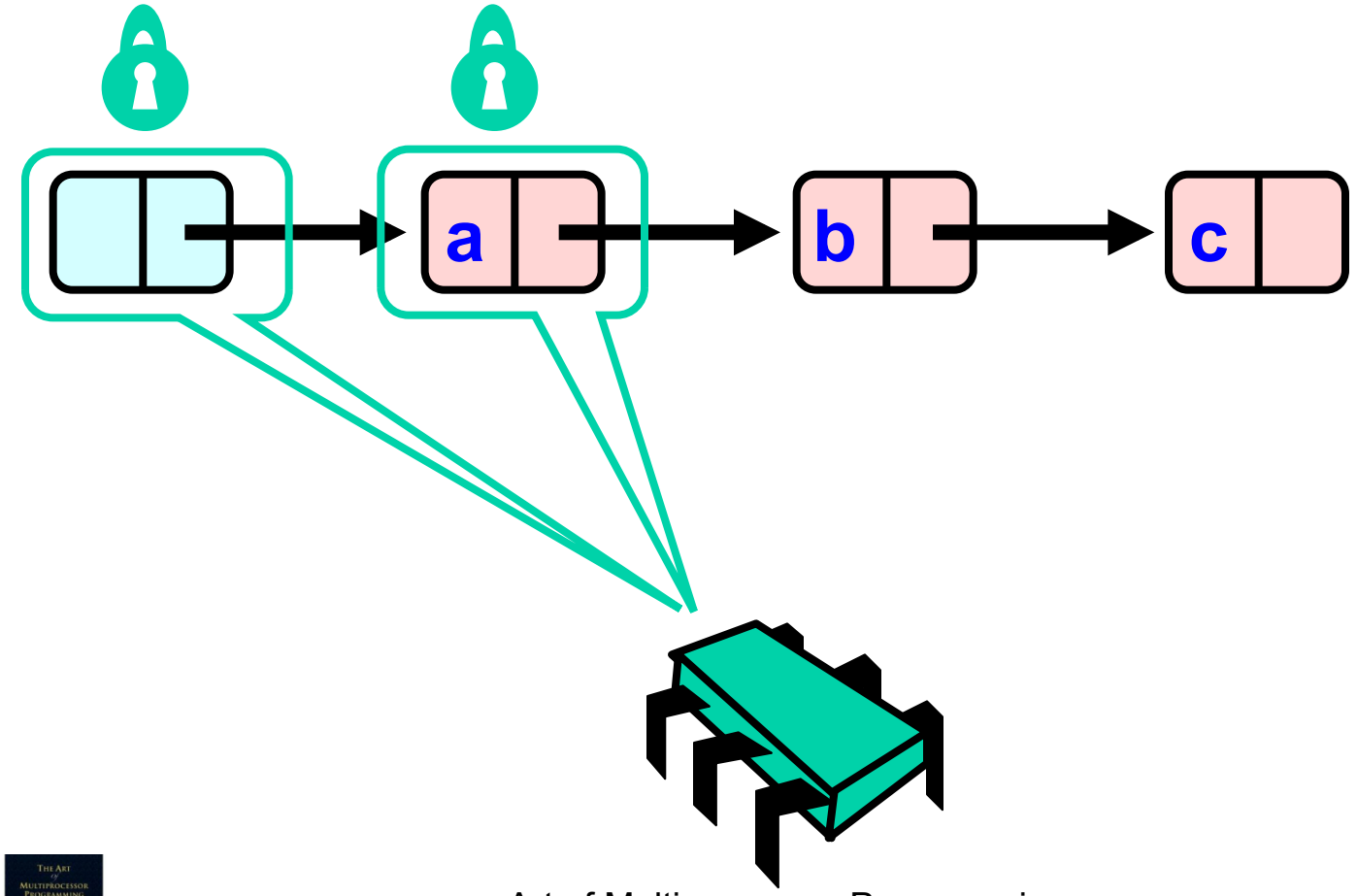


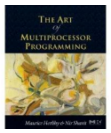
Hand-over-Hand locking



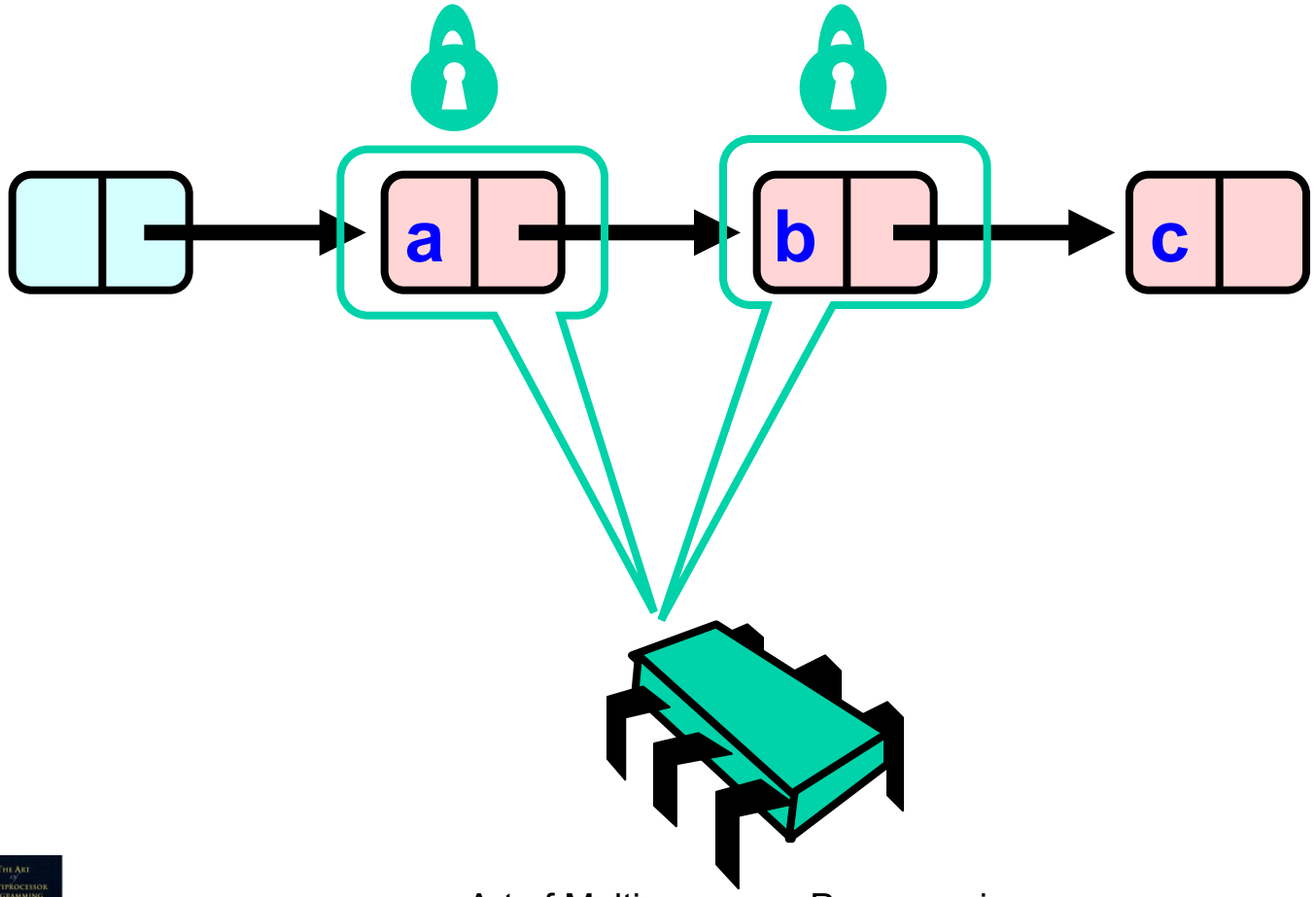


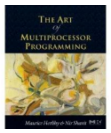
Hand-over-Hand locking



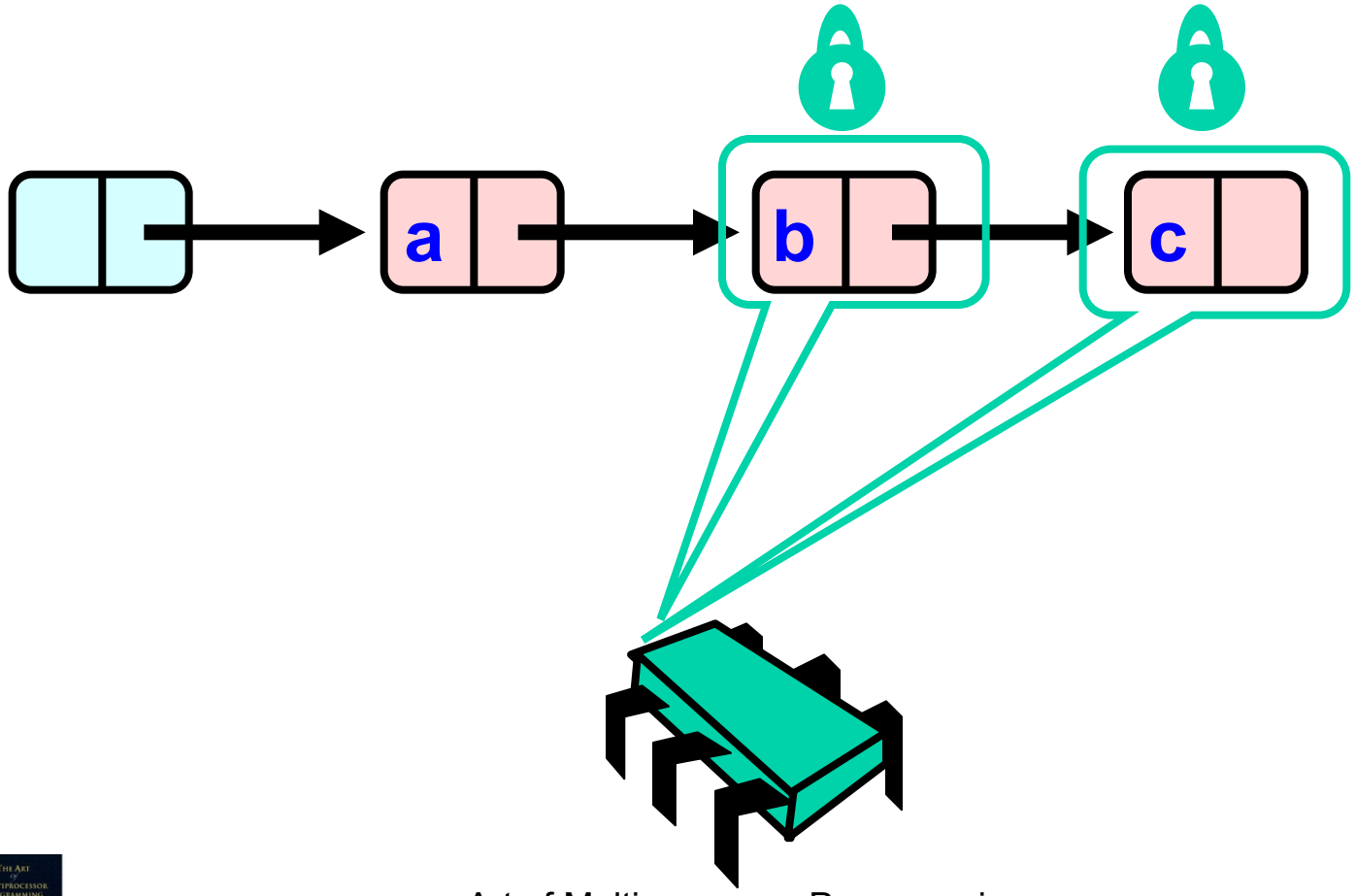


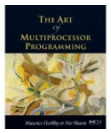
Hand-over-Hand locking



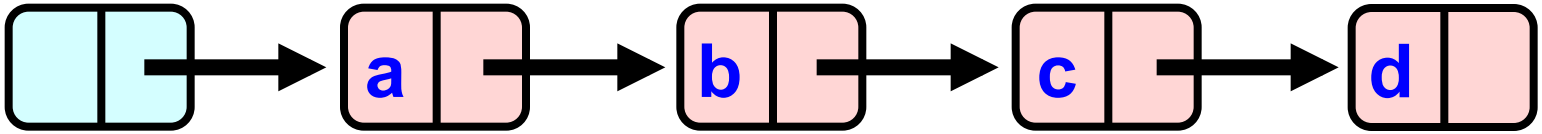


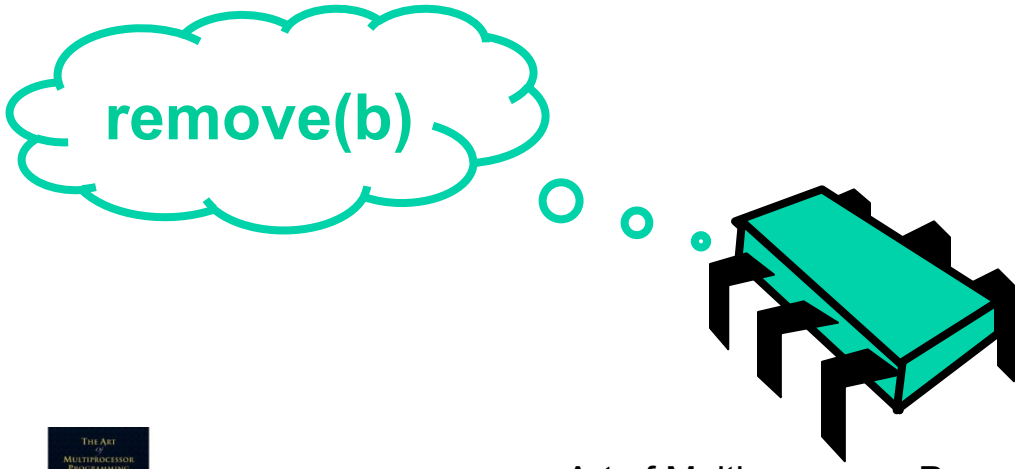
Hand-over-Hand locking

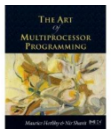




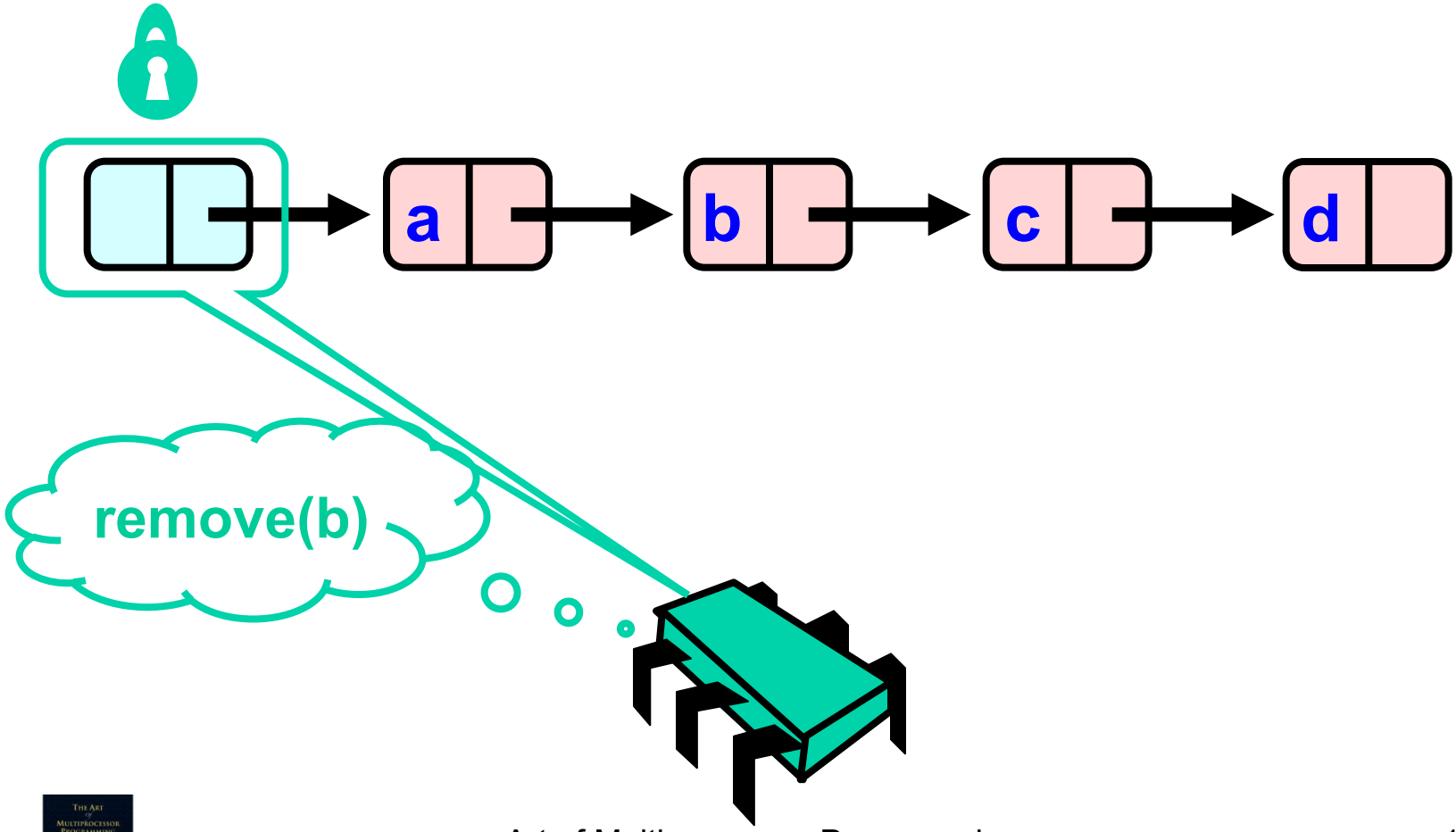
Removing a Node

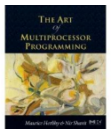




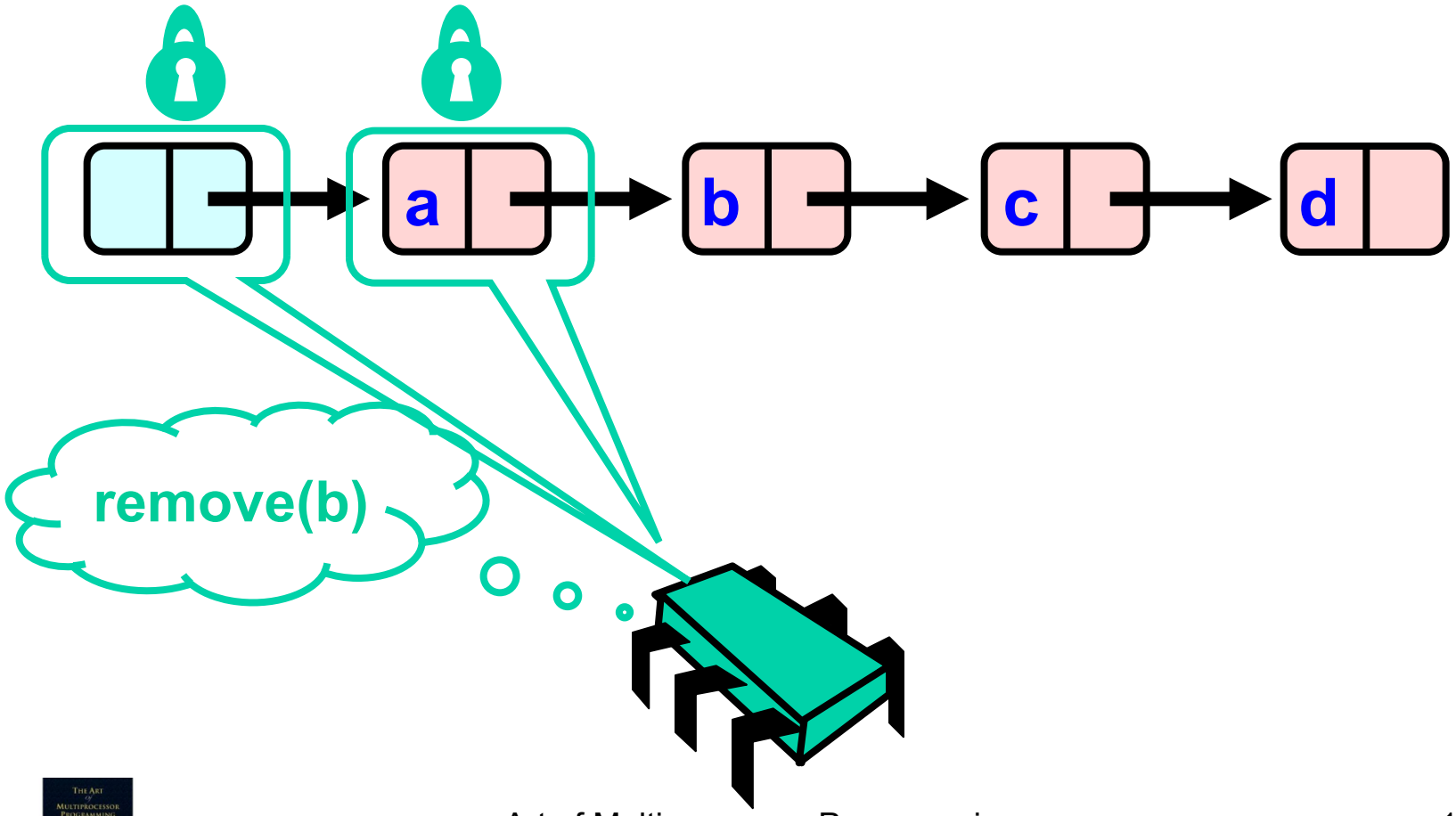


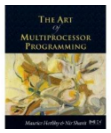
Removing a Node



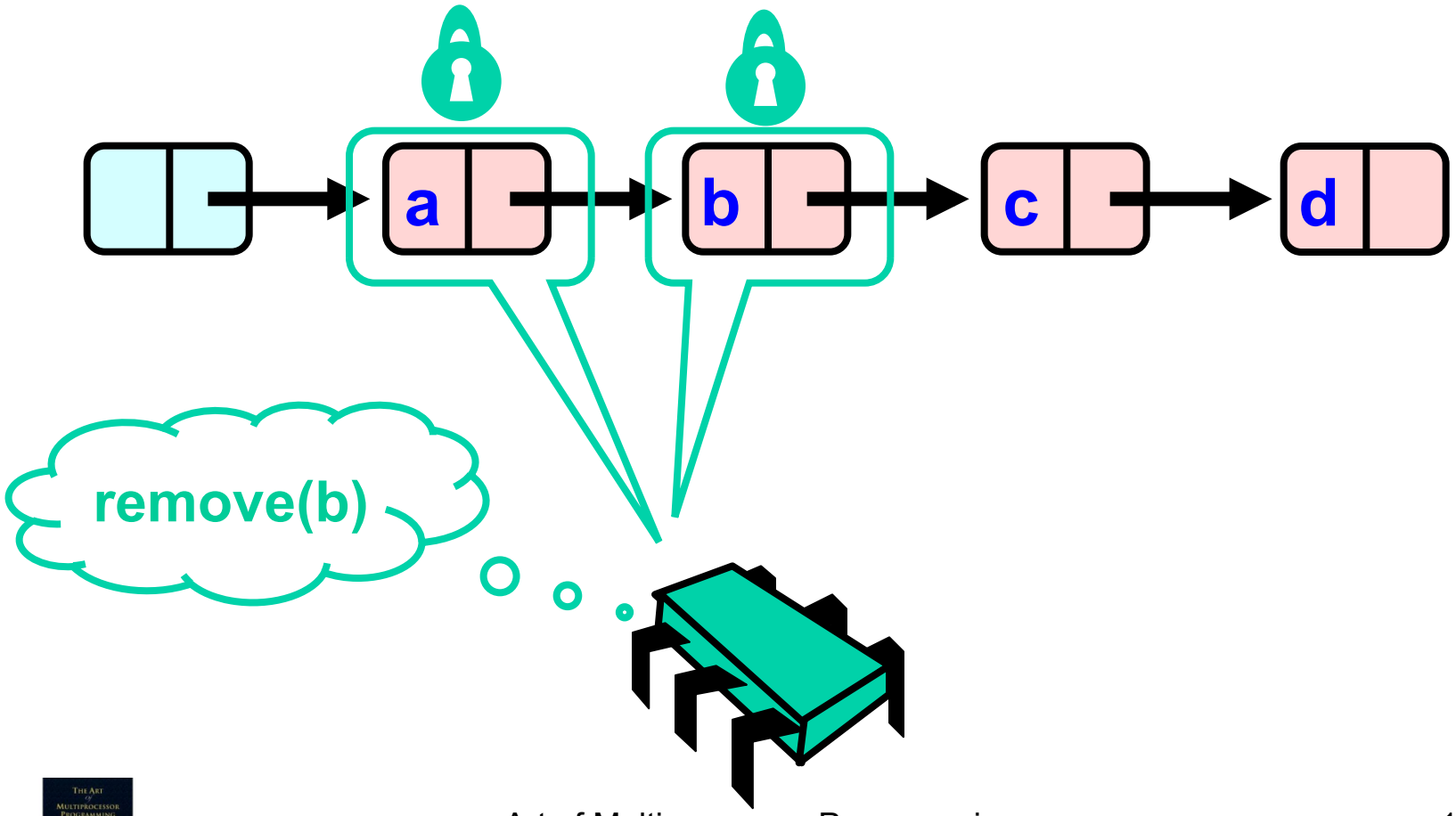


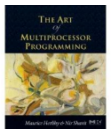
Removing a Node



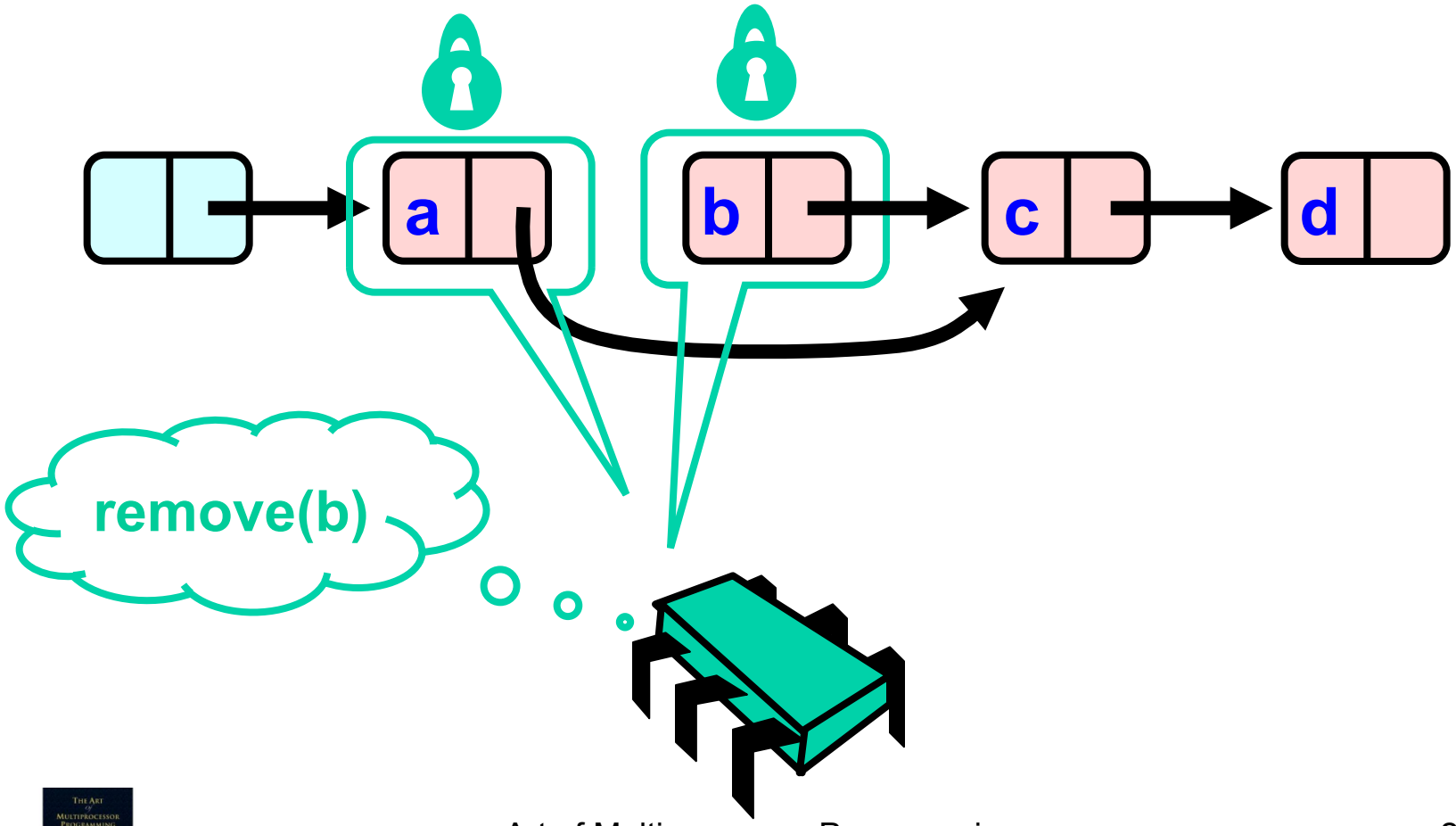


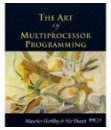
Removing a Node



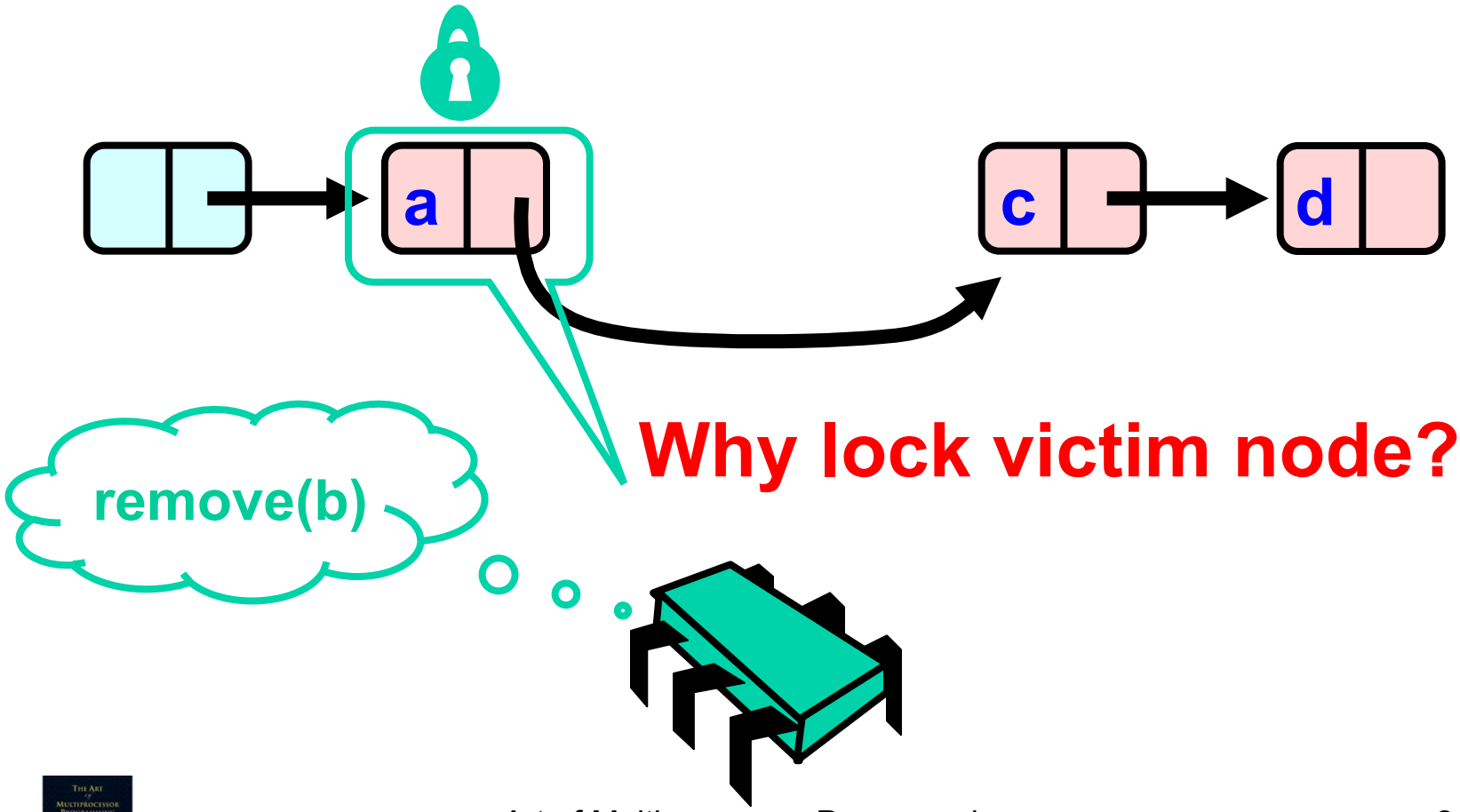


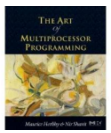
Removing a Node



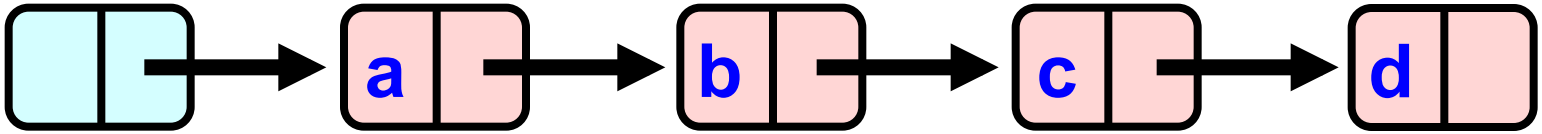


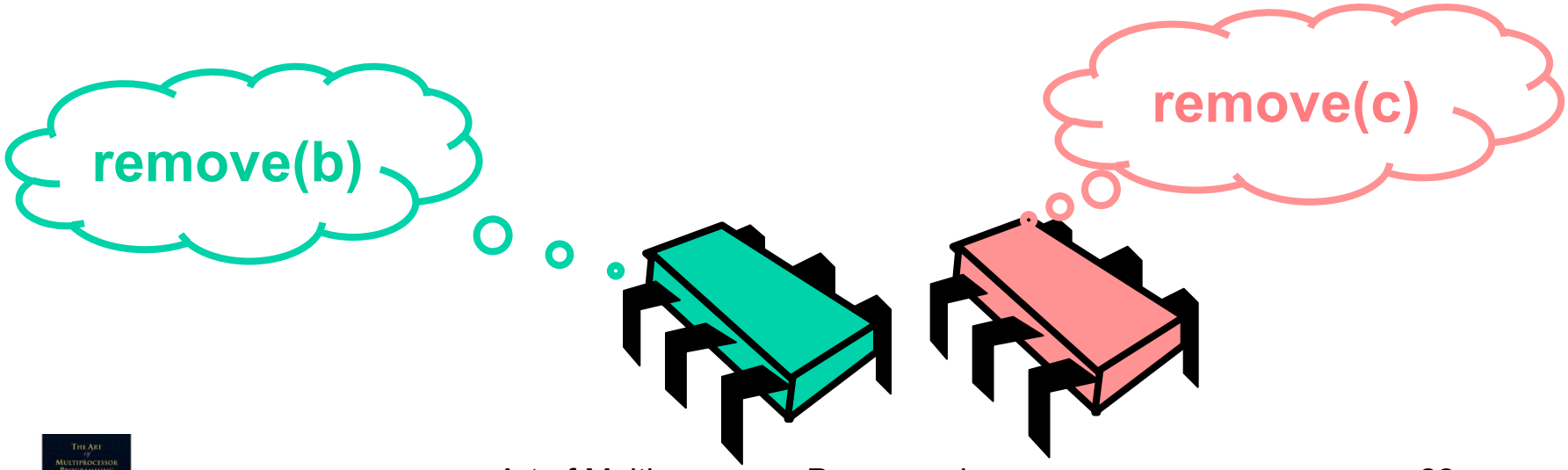
Removing a Node

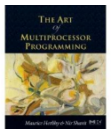




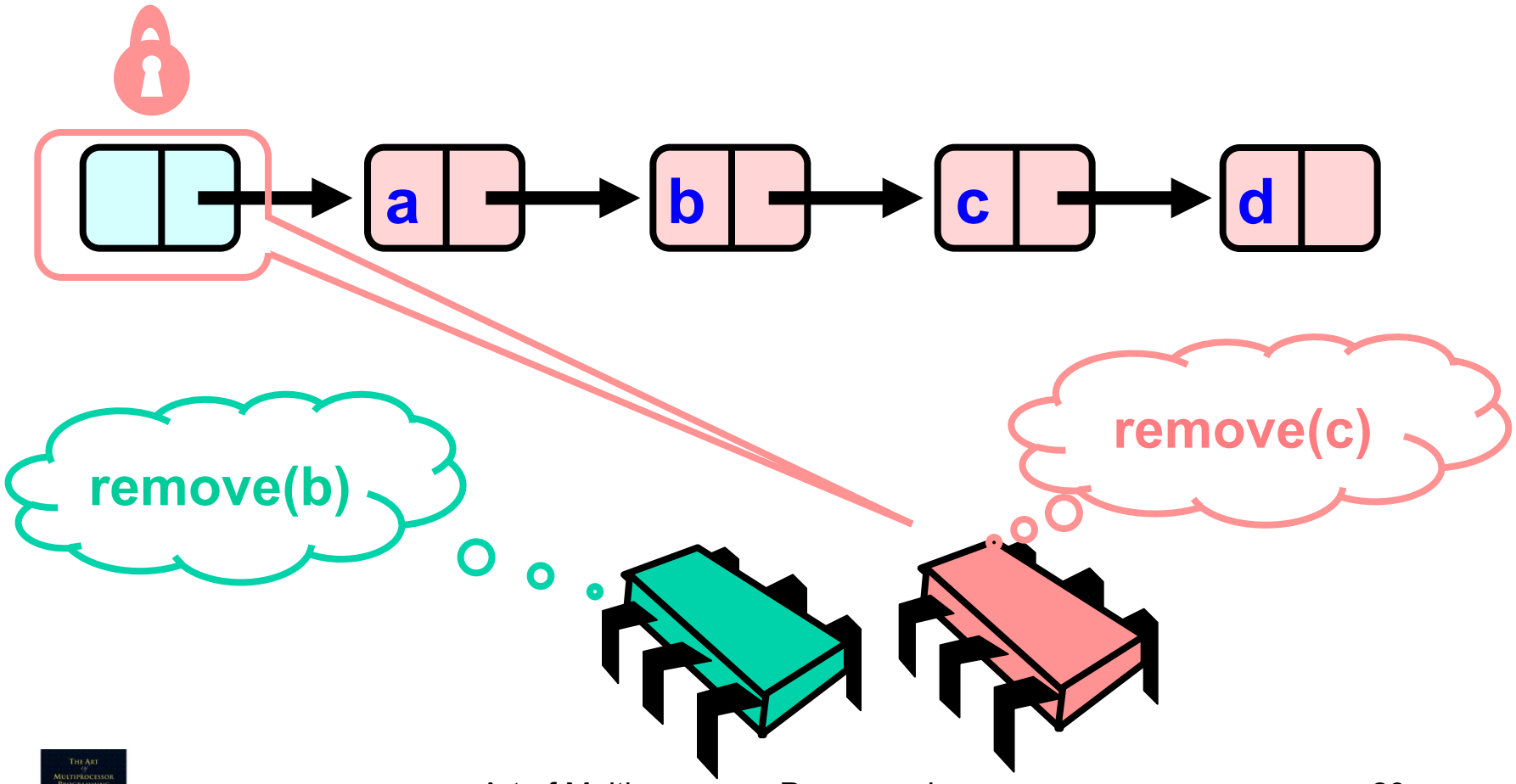
Concurrent Removes

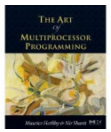




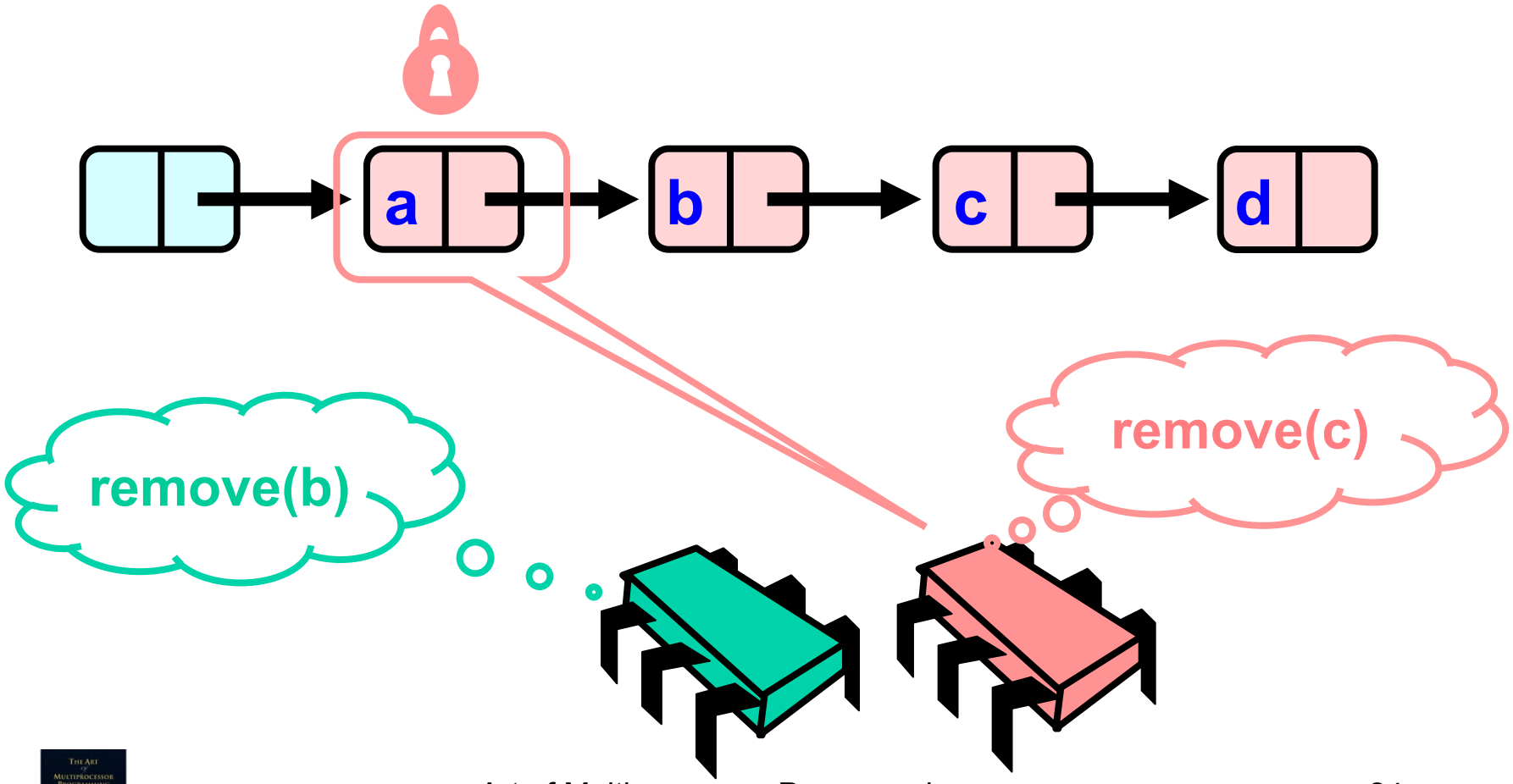


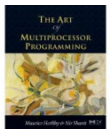
Concurrent Removes



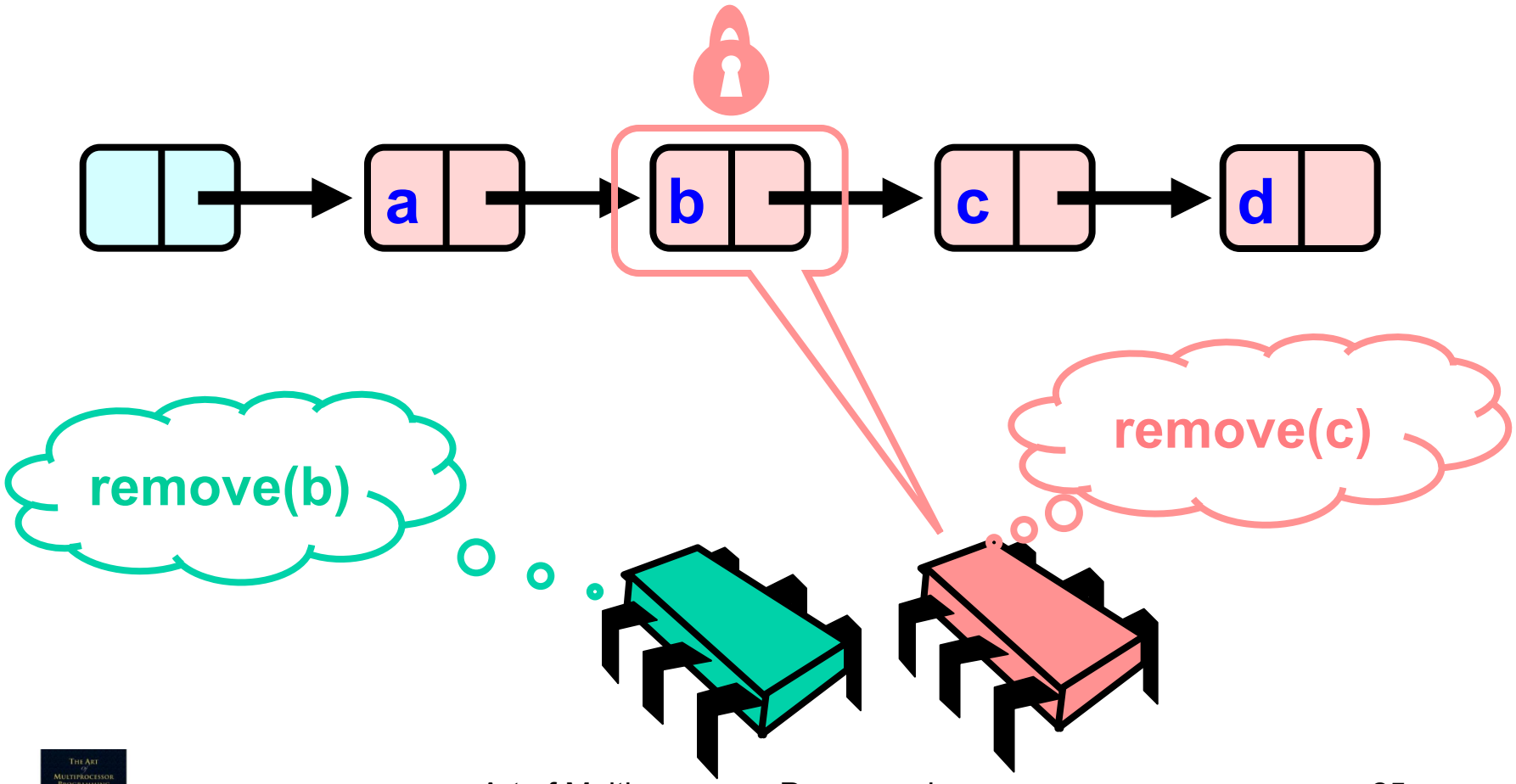


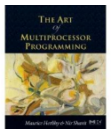
Concurrent Removes



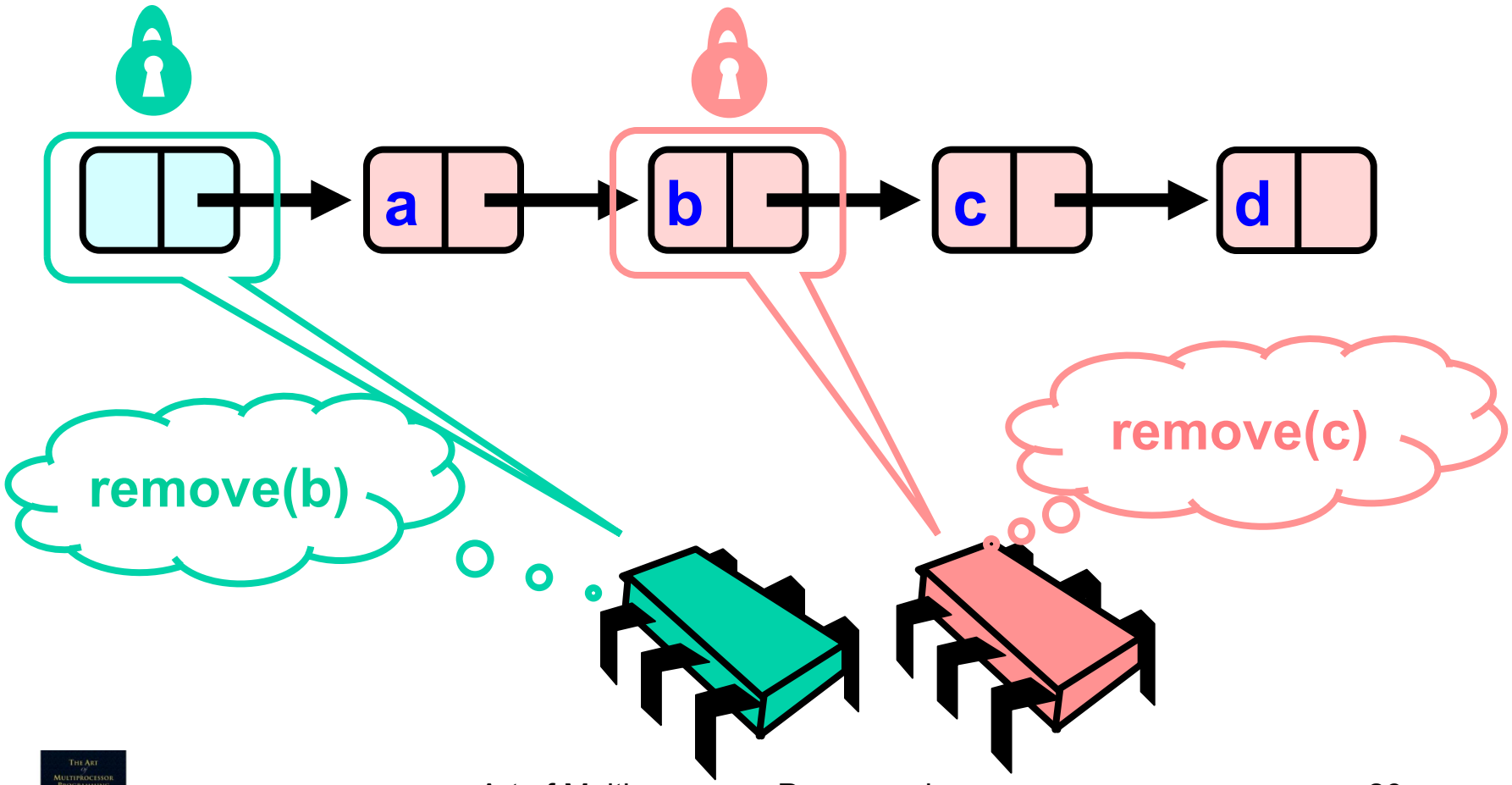


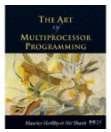
Concurrent Removes



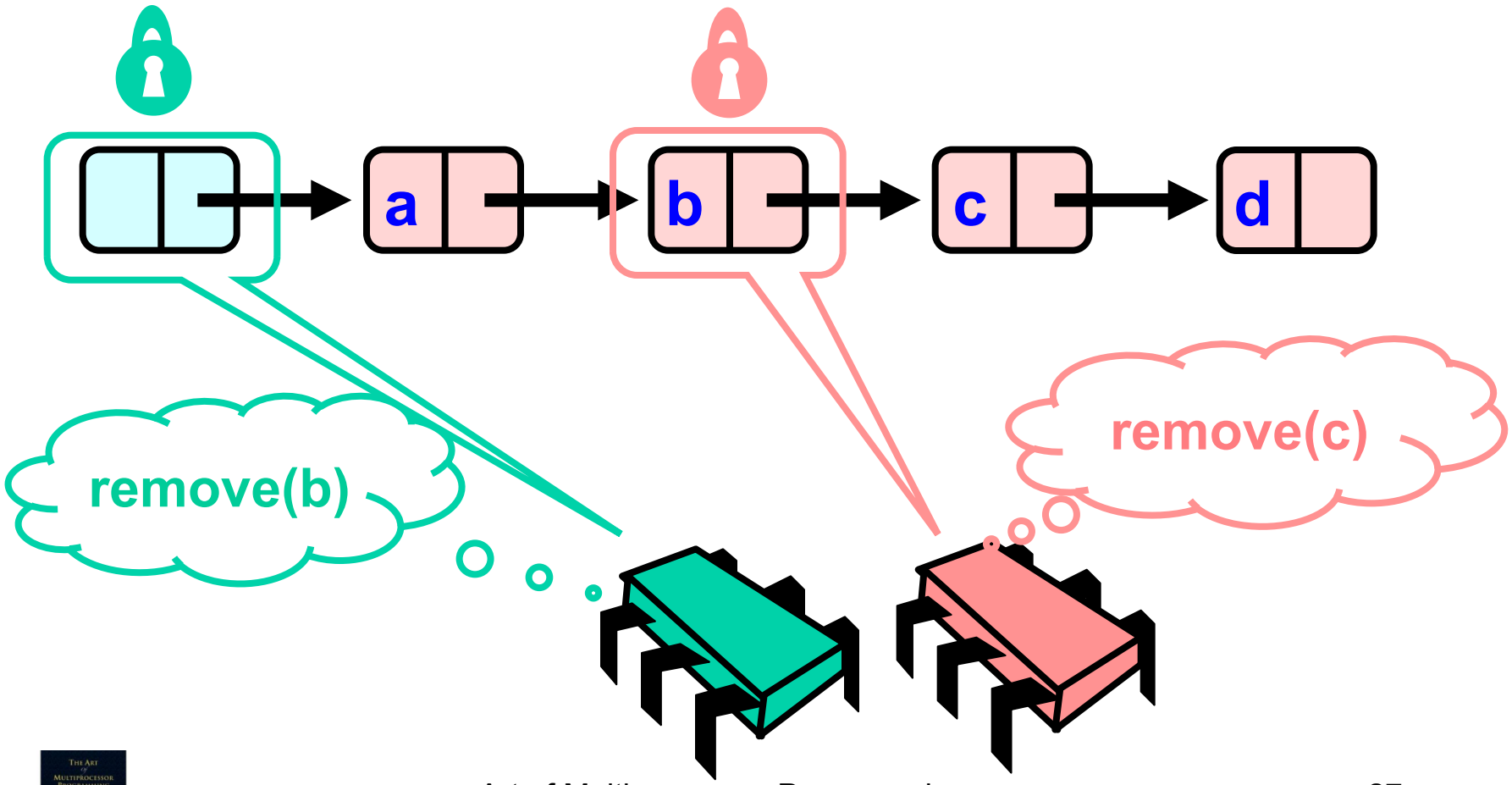


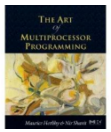
Concurrent Removes



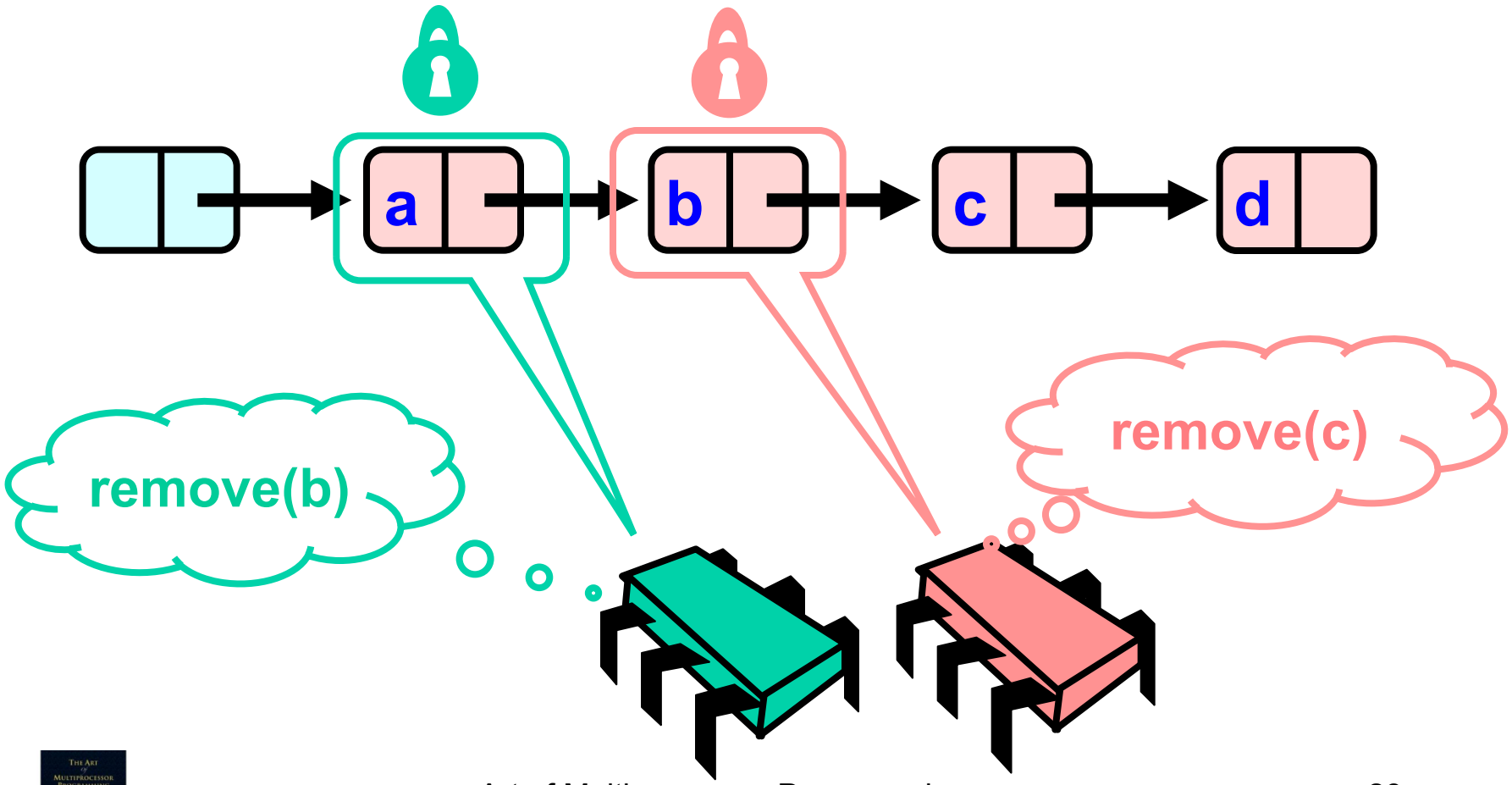


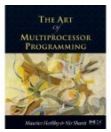
Concurrent Removes



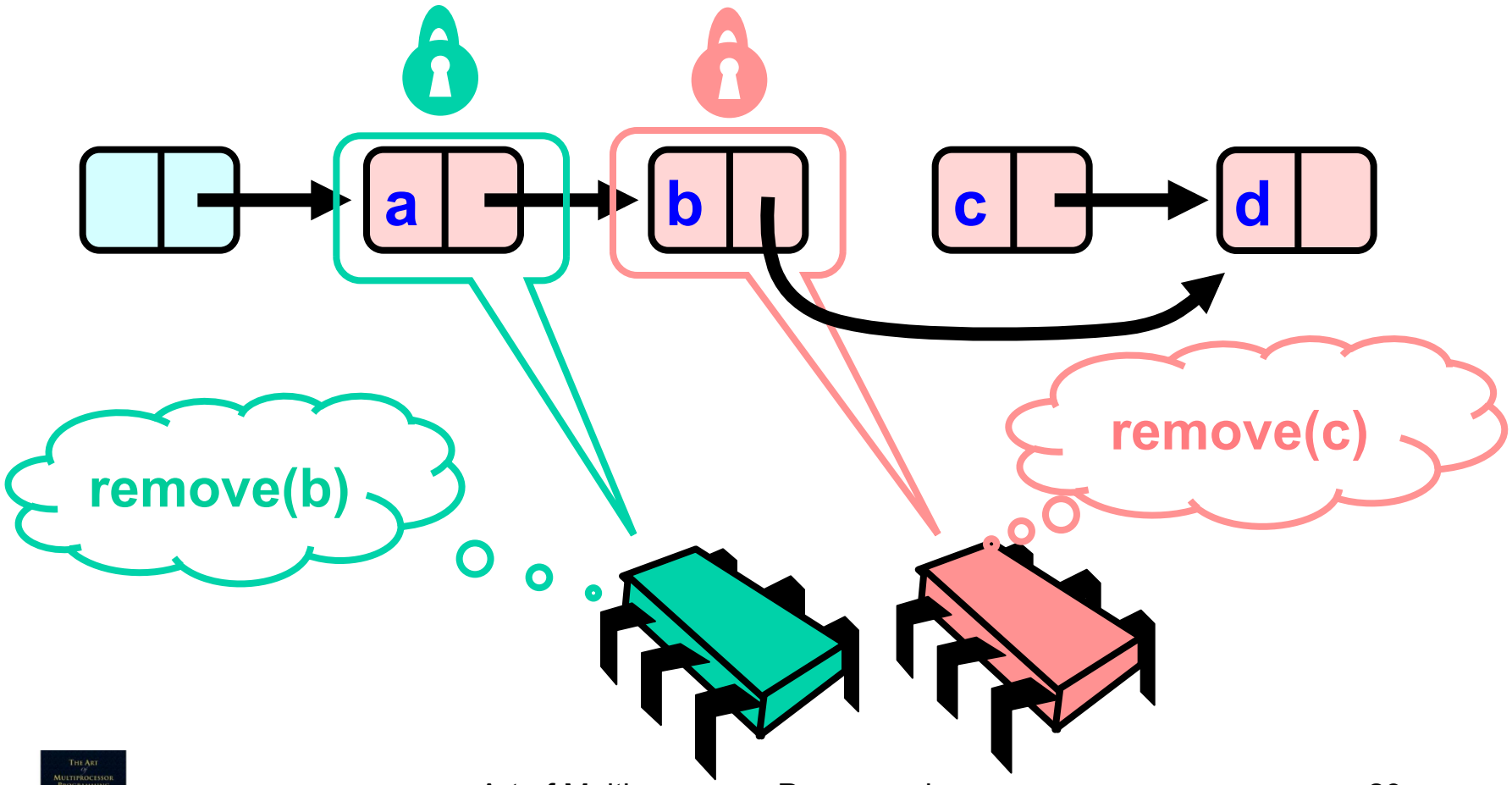


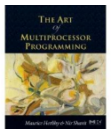
Concurrent Removes



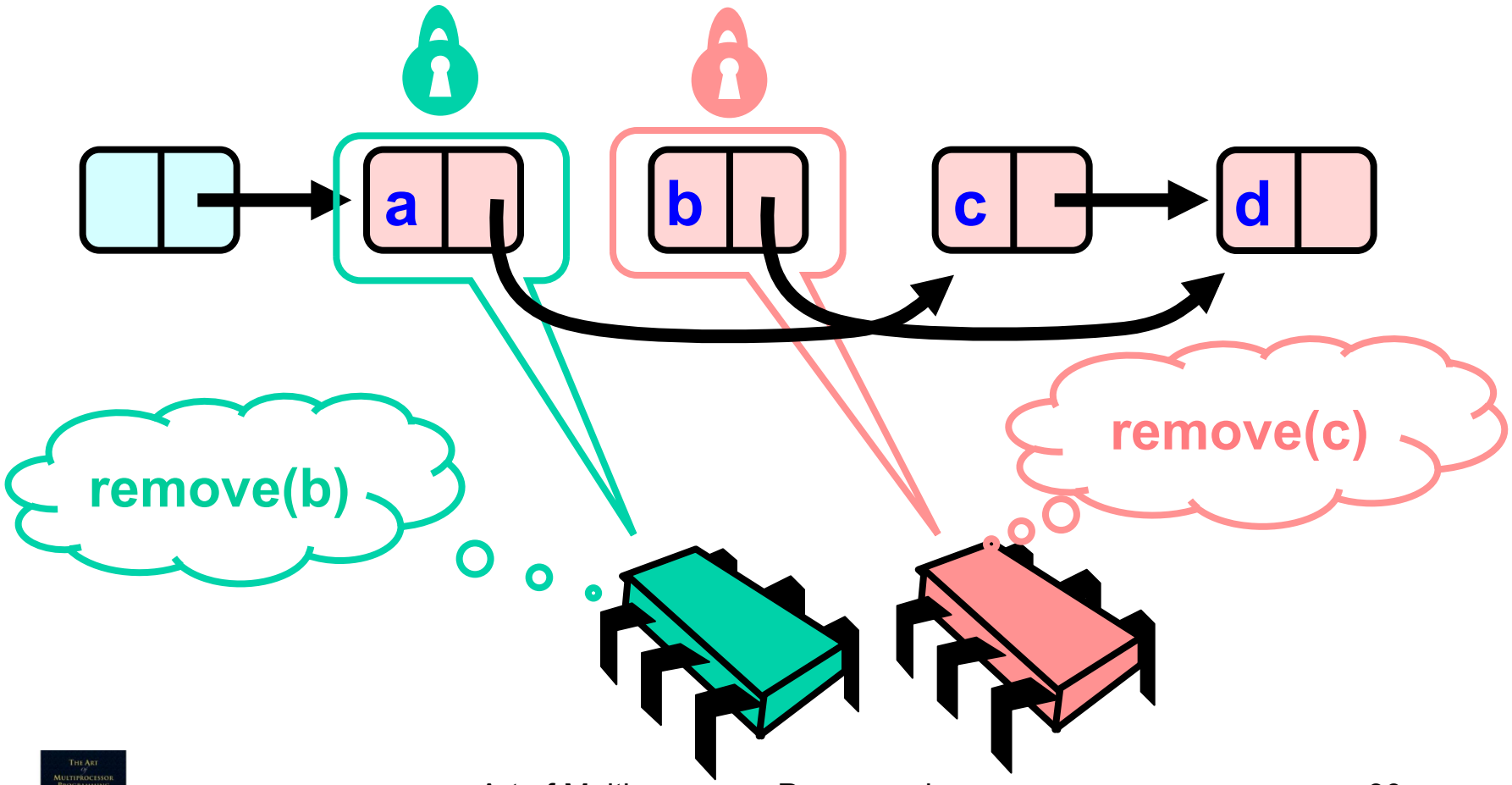


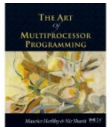
Concurrent Removes



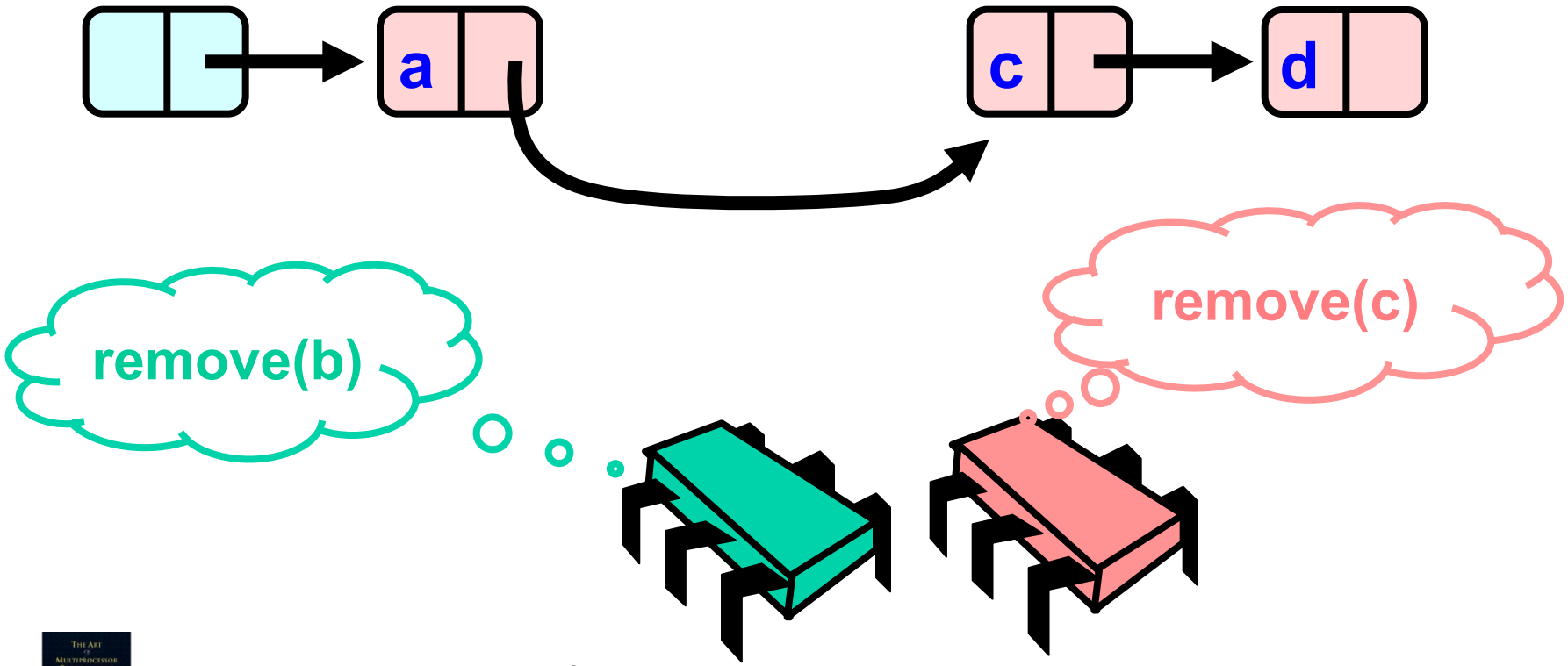


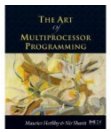
Concurrent Removes





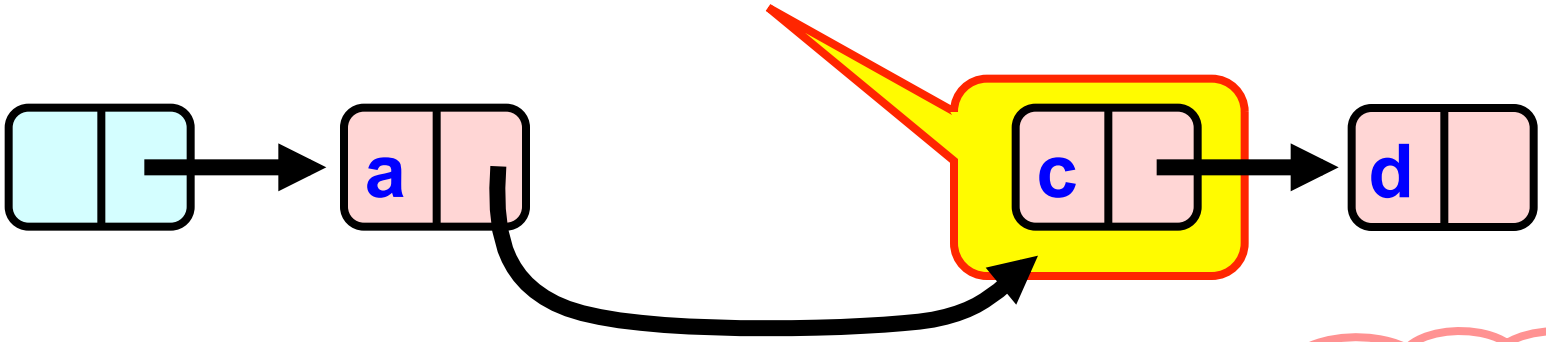
Uh, Oh

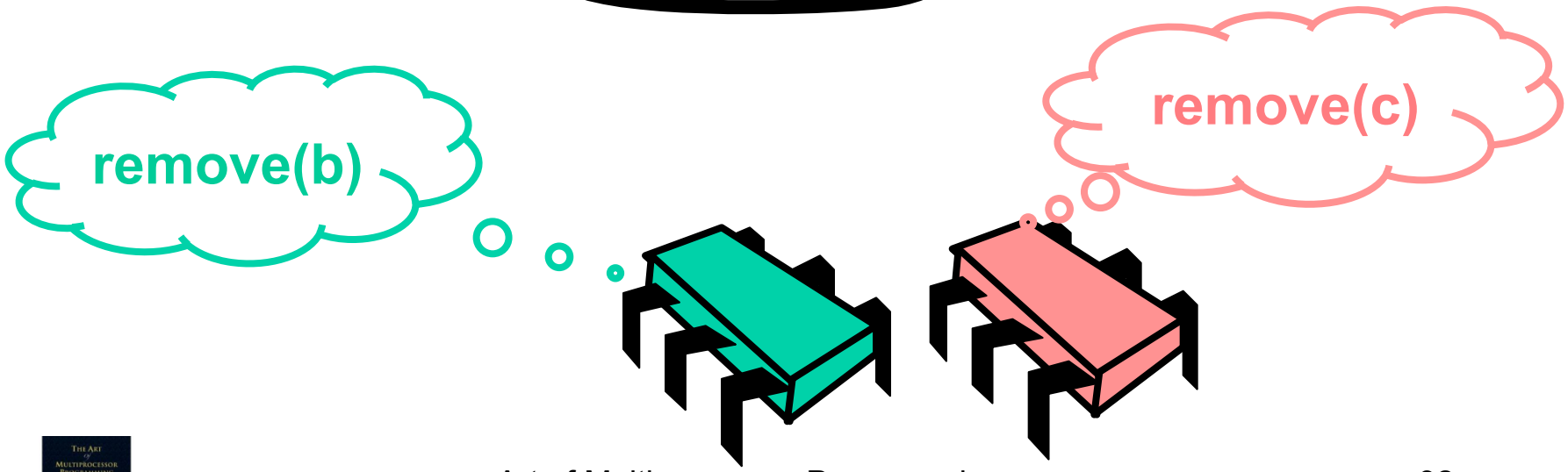


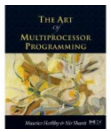


Uh, Oh

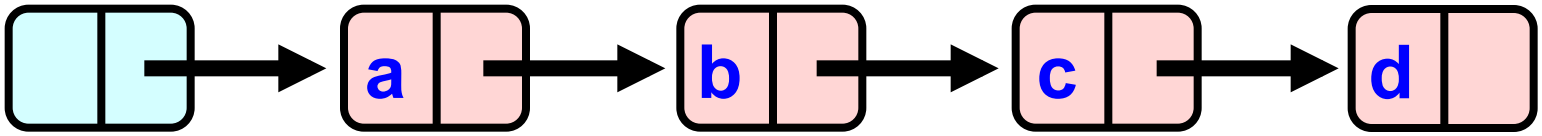
Bad news, c not removed

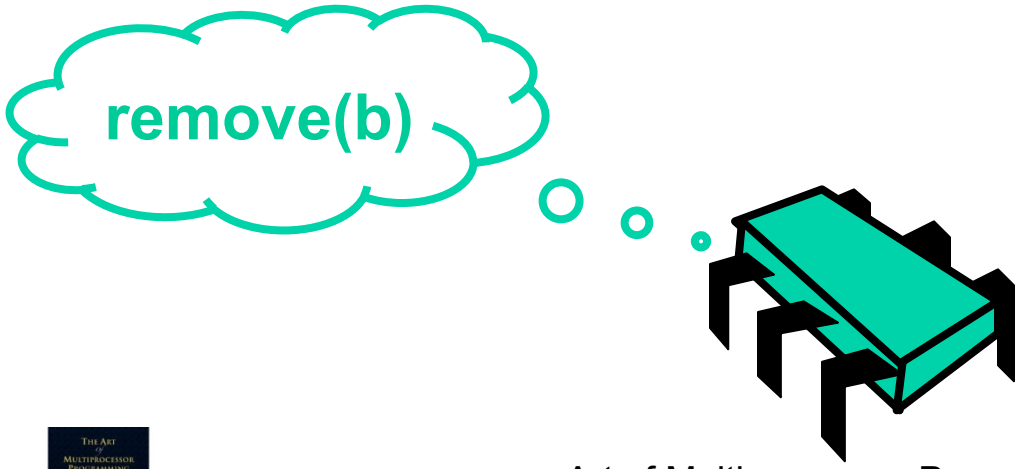


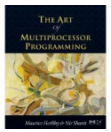




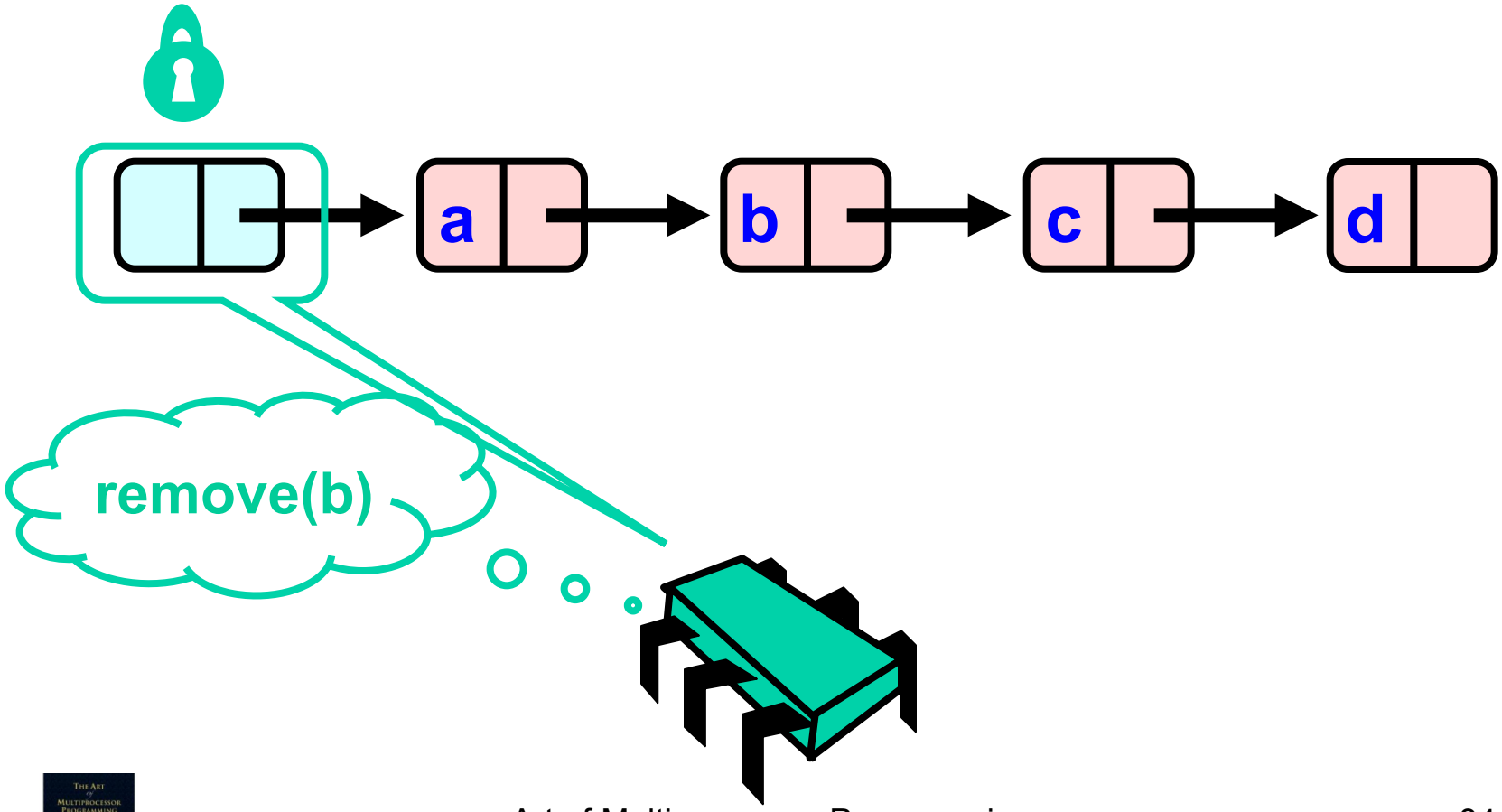
Hand-Over-Hand Again

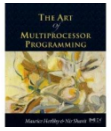




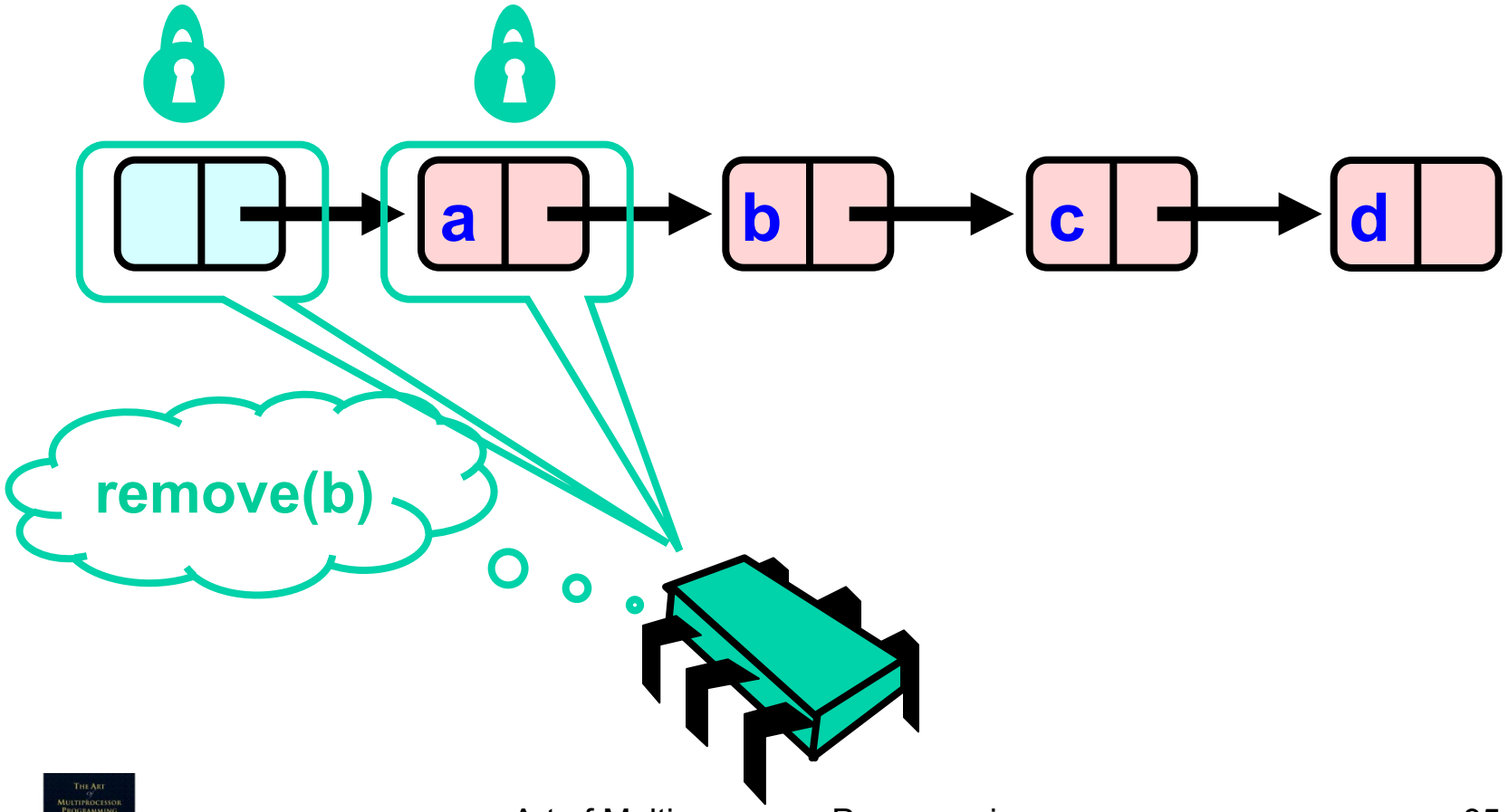


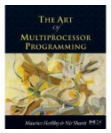
Hand-Over-Hand Again



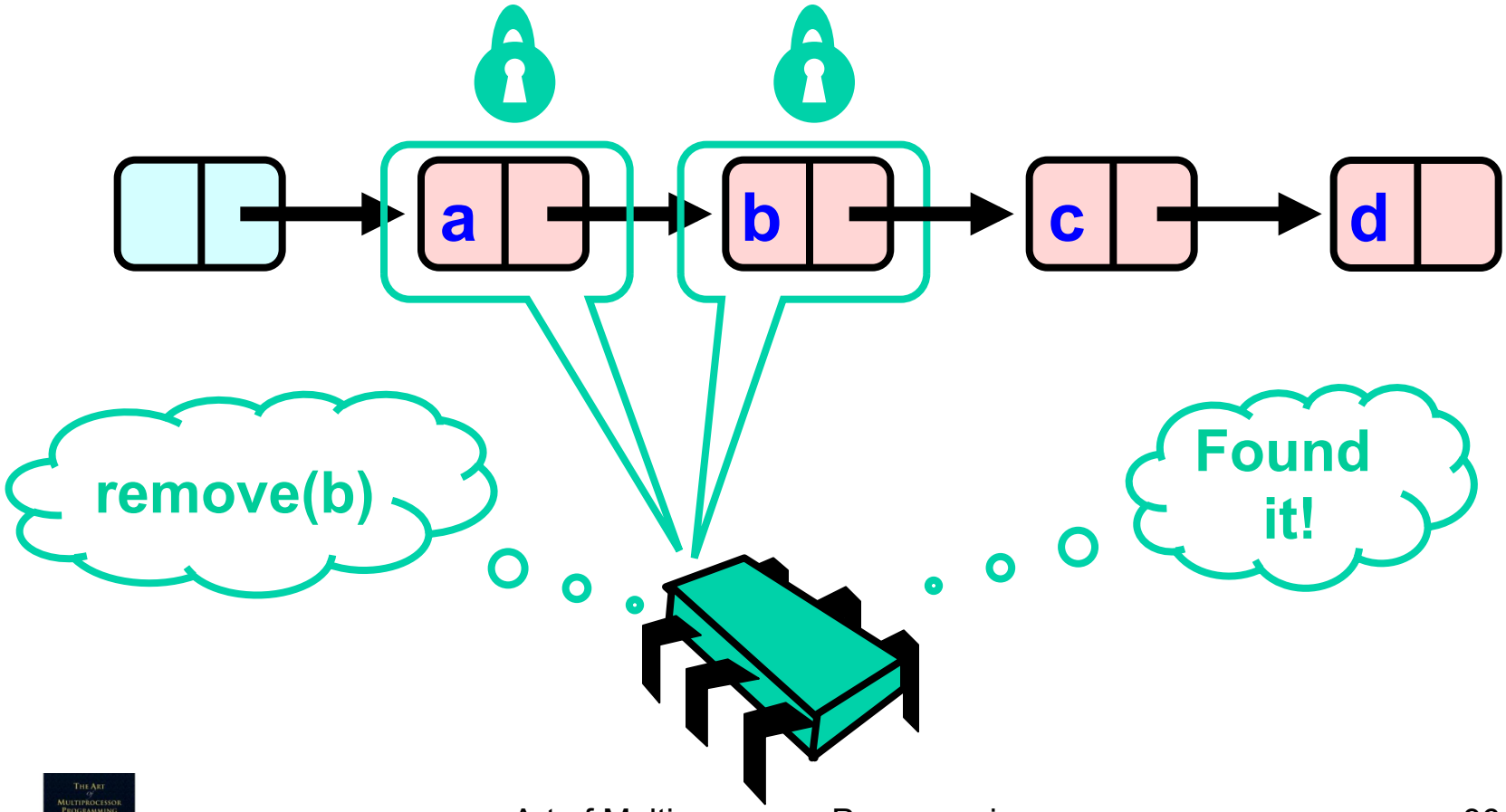


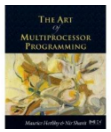
Hand-Over-Hand Again



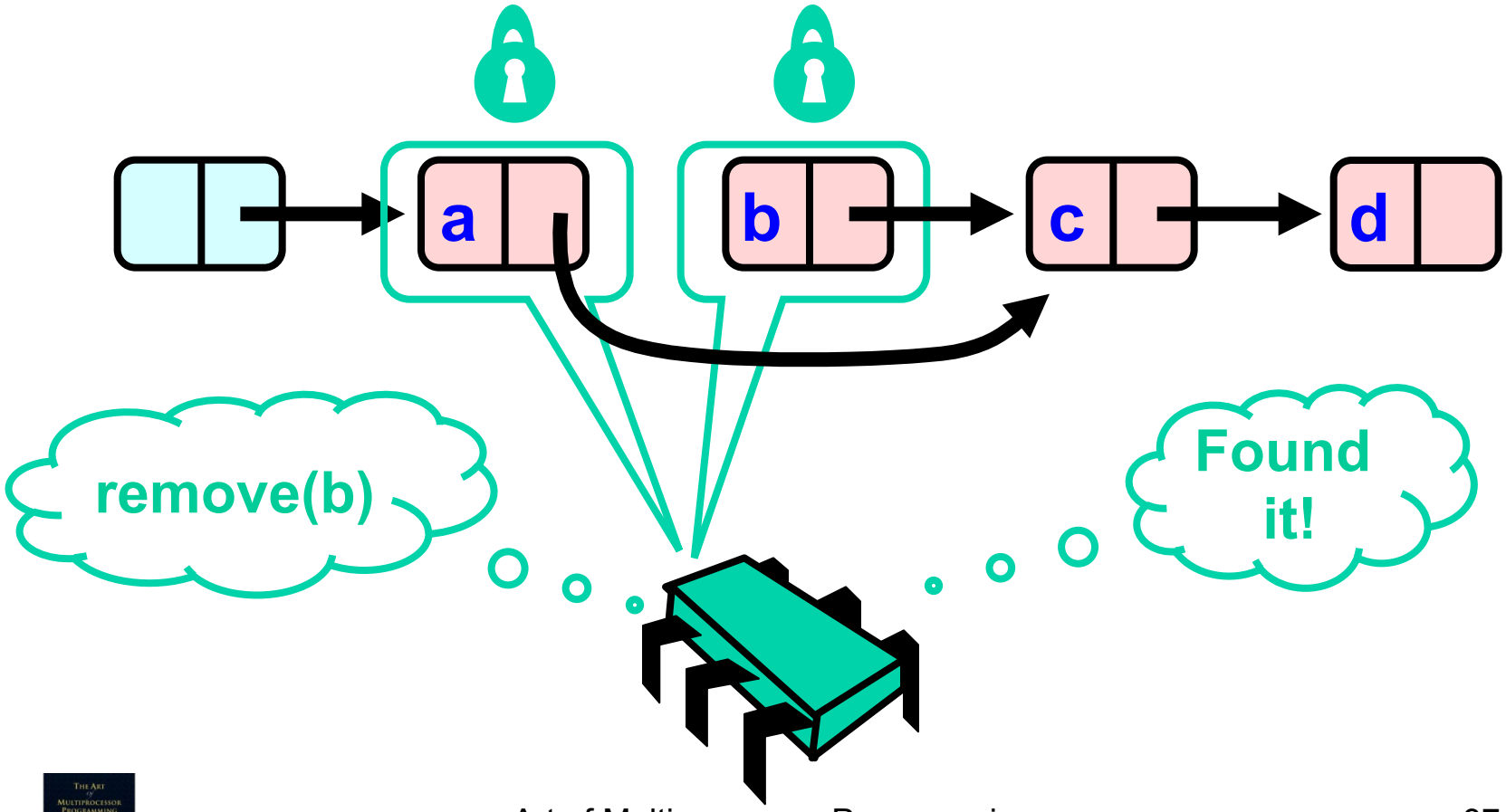


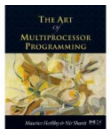
Hand-Over-Hand Again



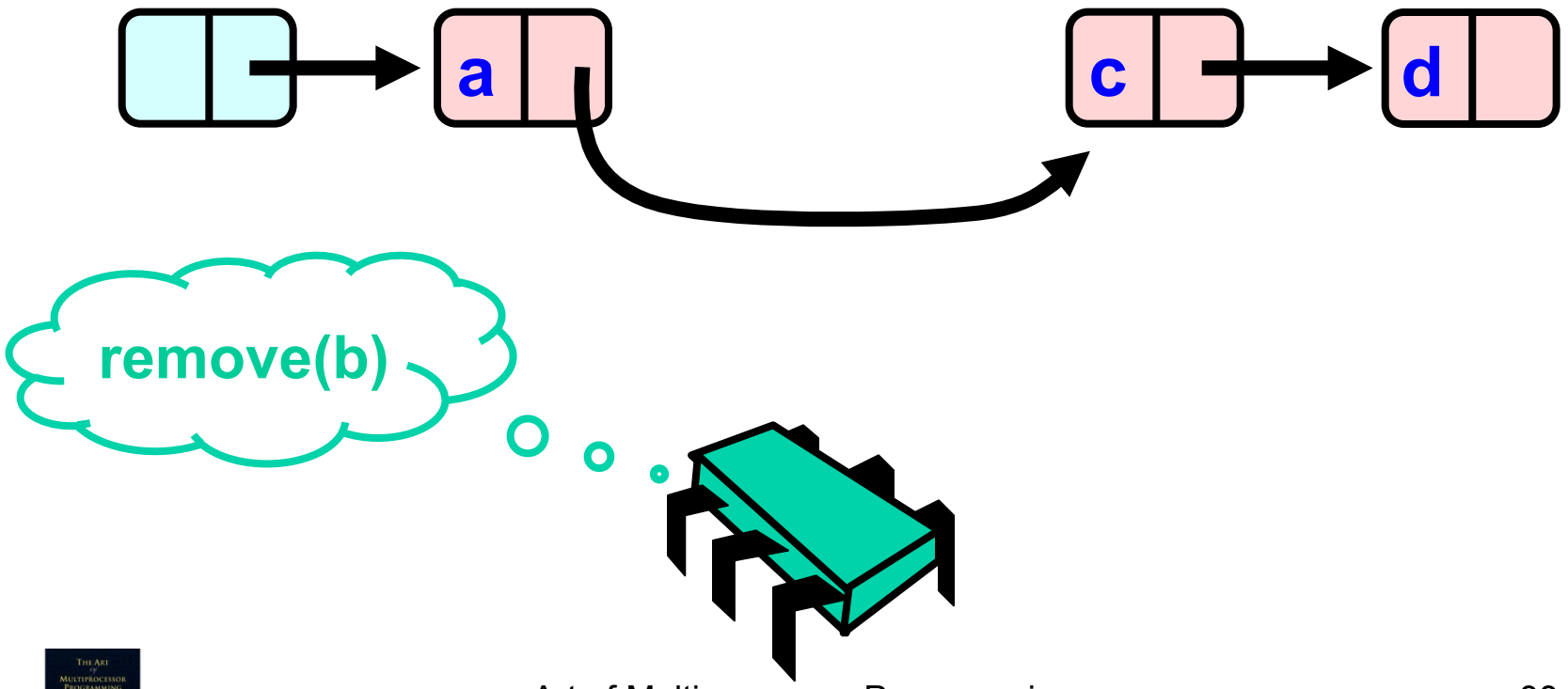


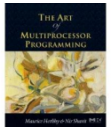
Hand-Over-Hand Again



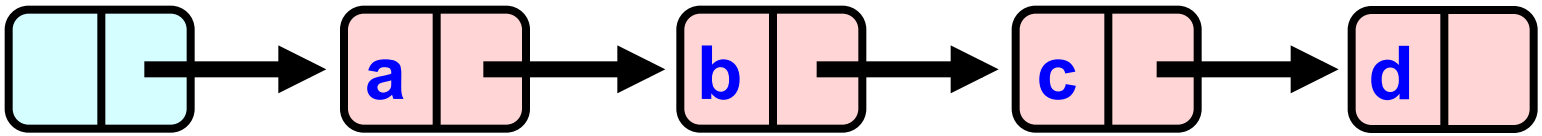


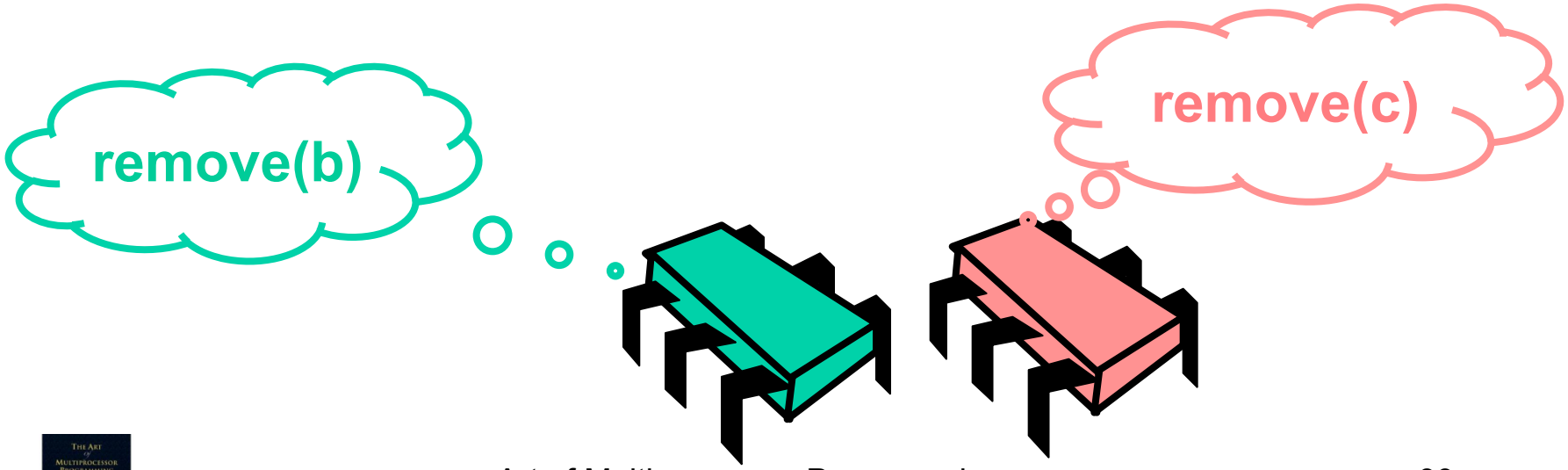
Hand-Over-Hand Again

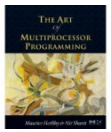




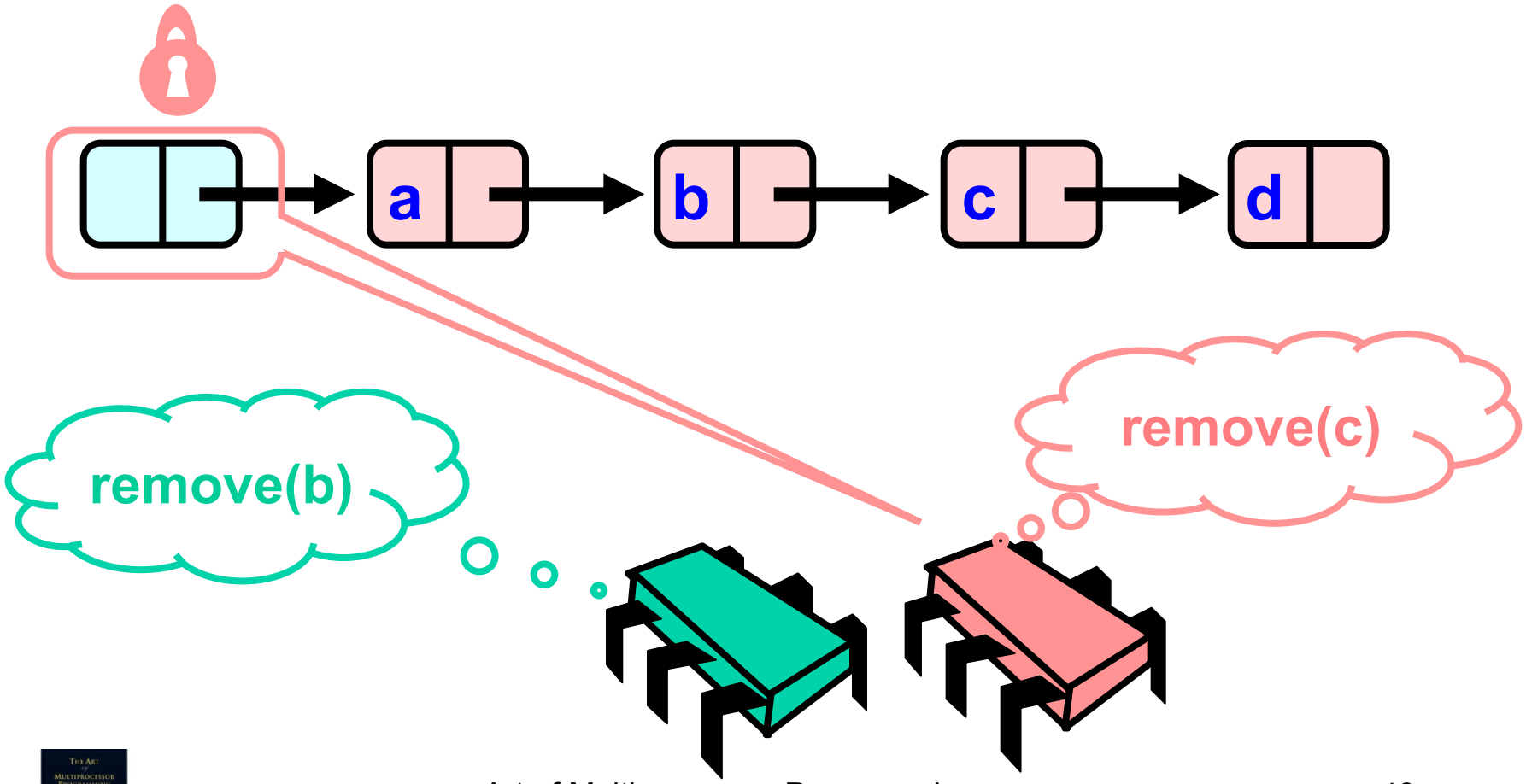
Removing a Node

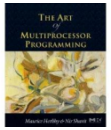




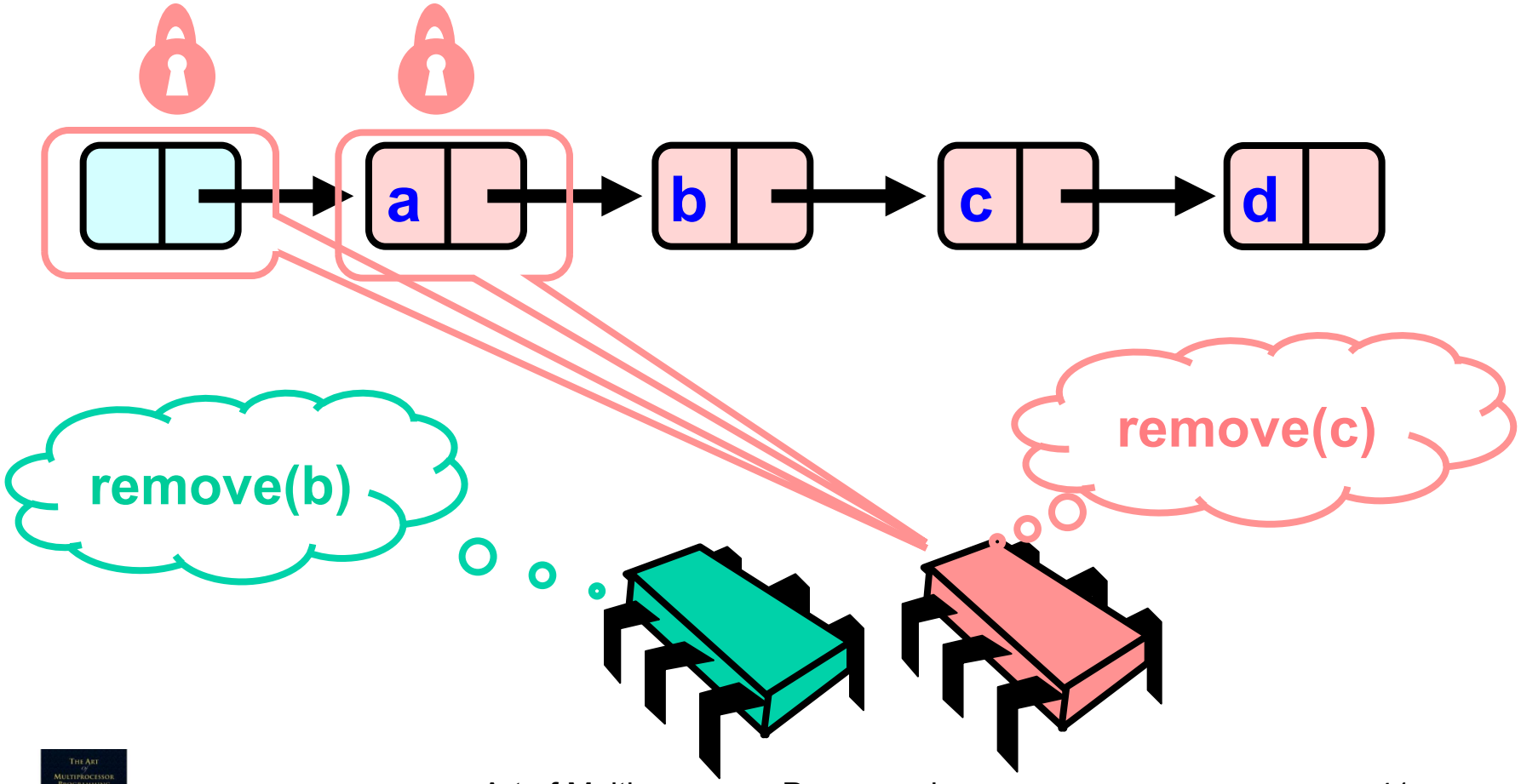


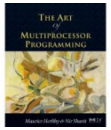
Removing a Node



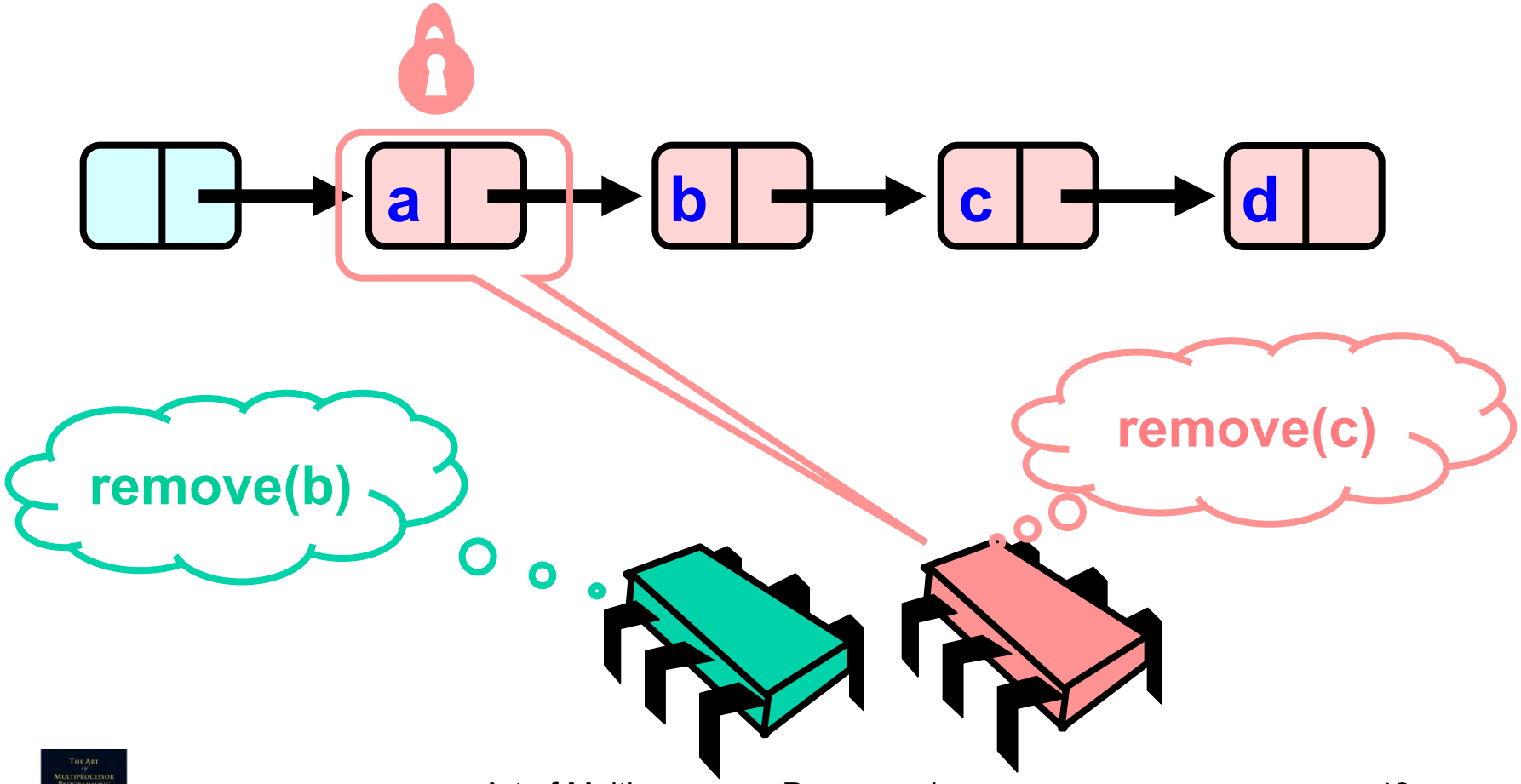


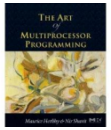
Removing a Node



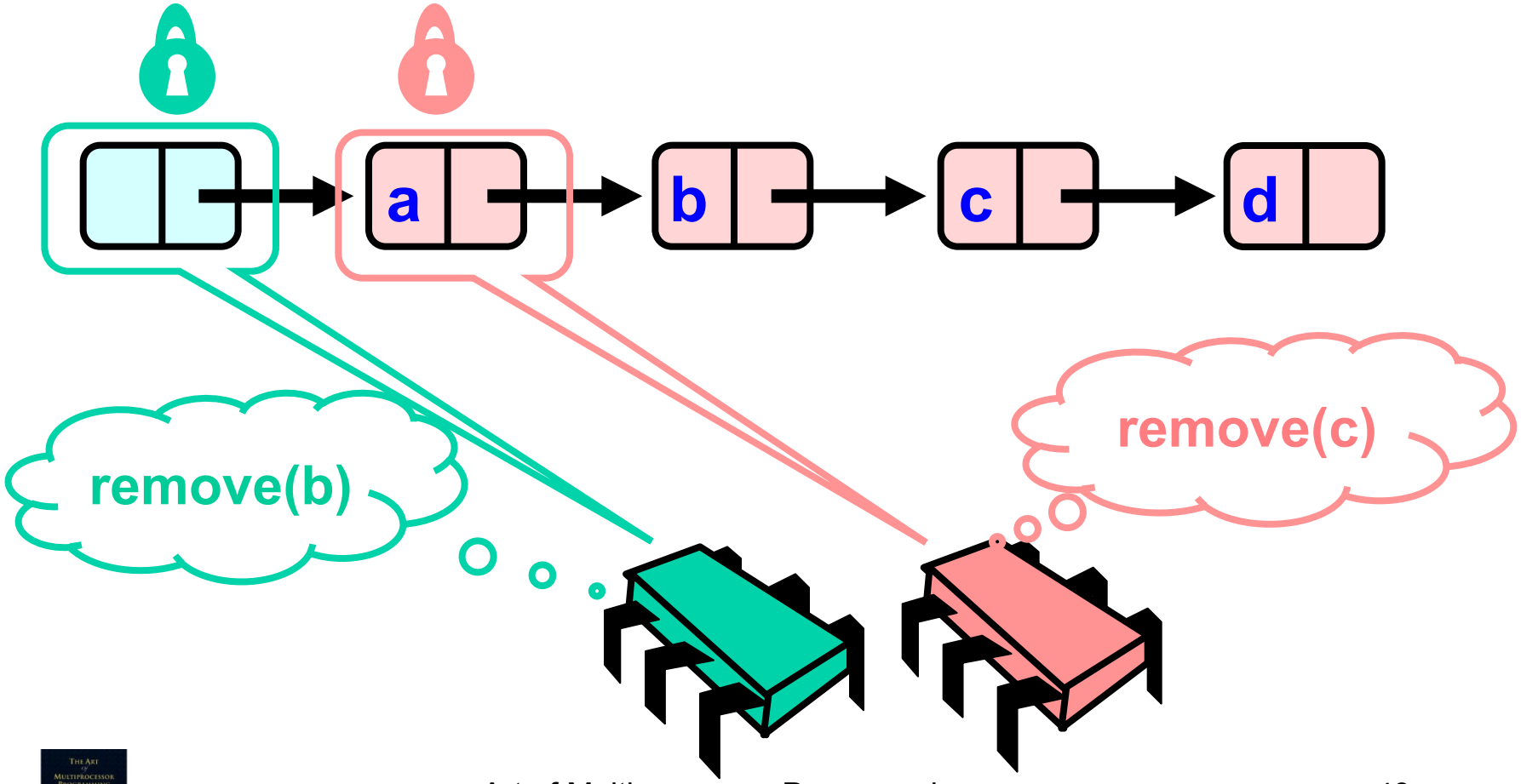


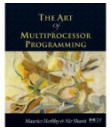
Removing a Node



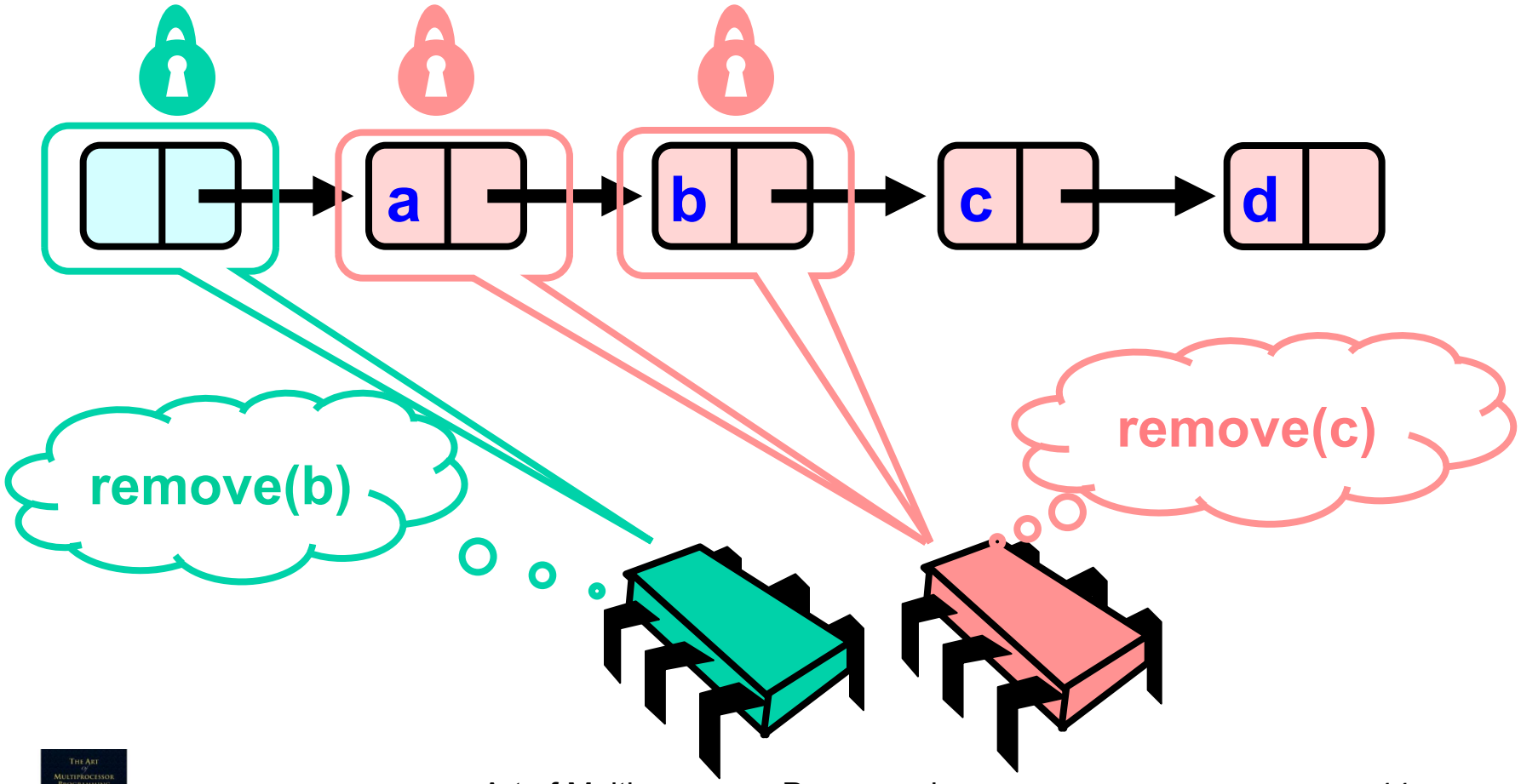


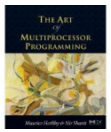
Removing a Node



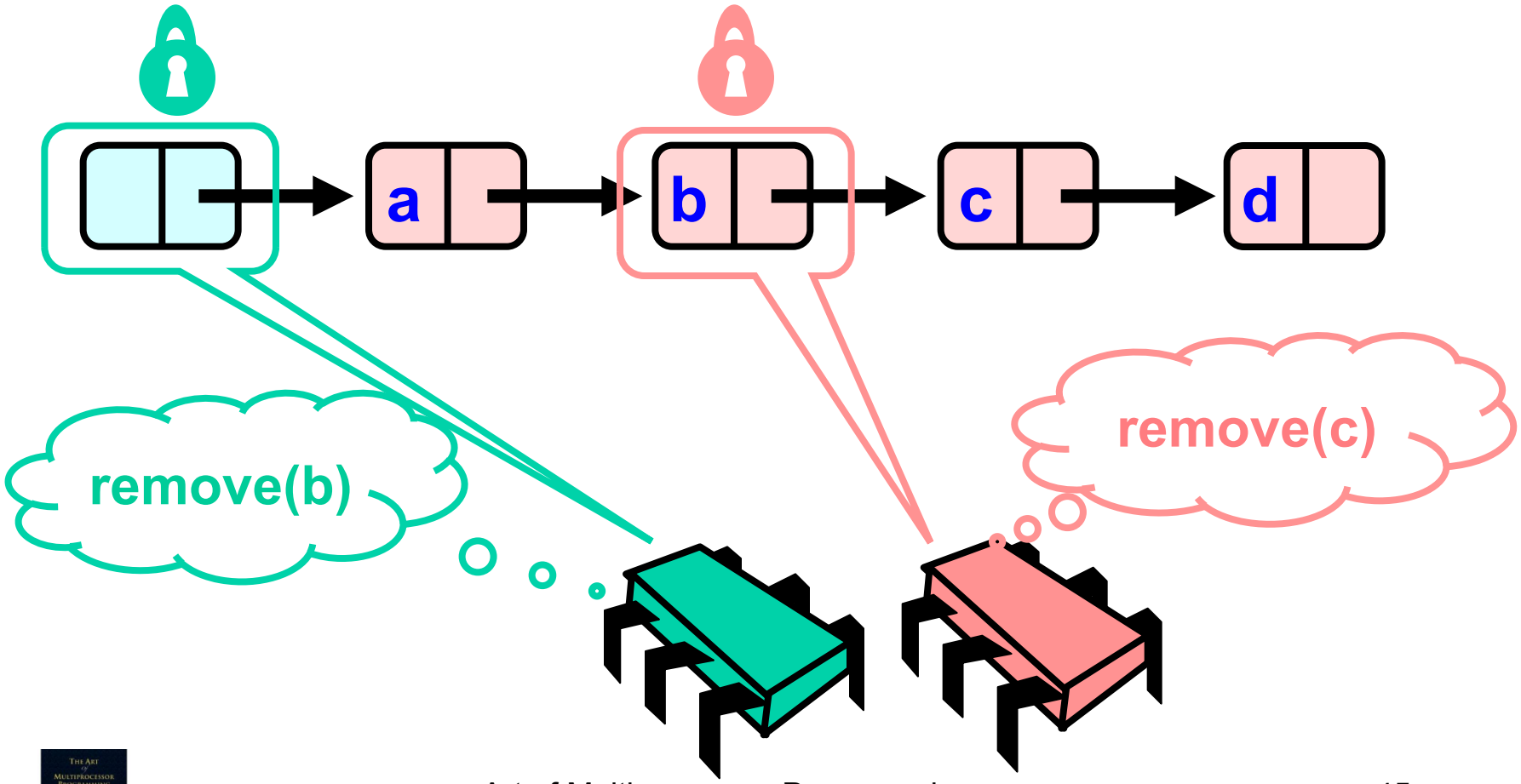


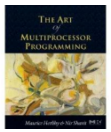
Removing a Node



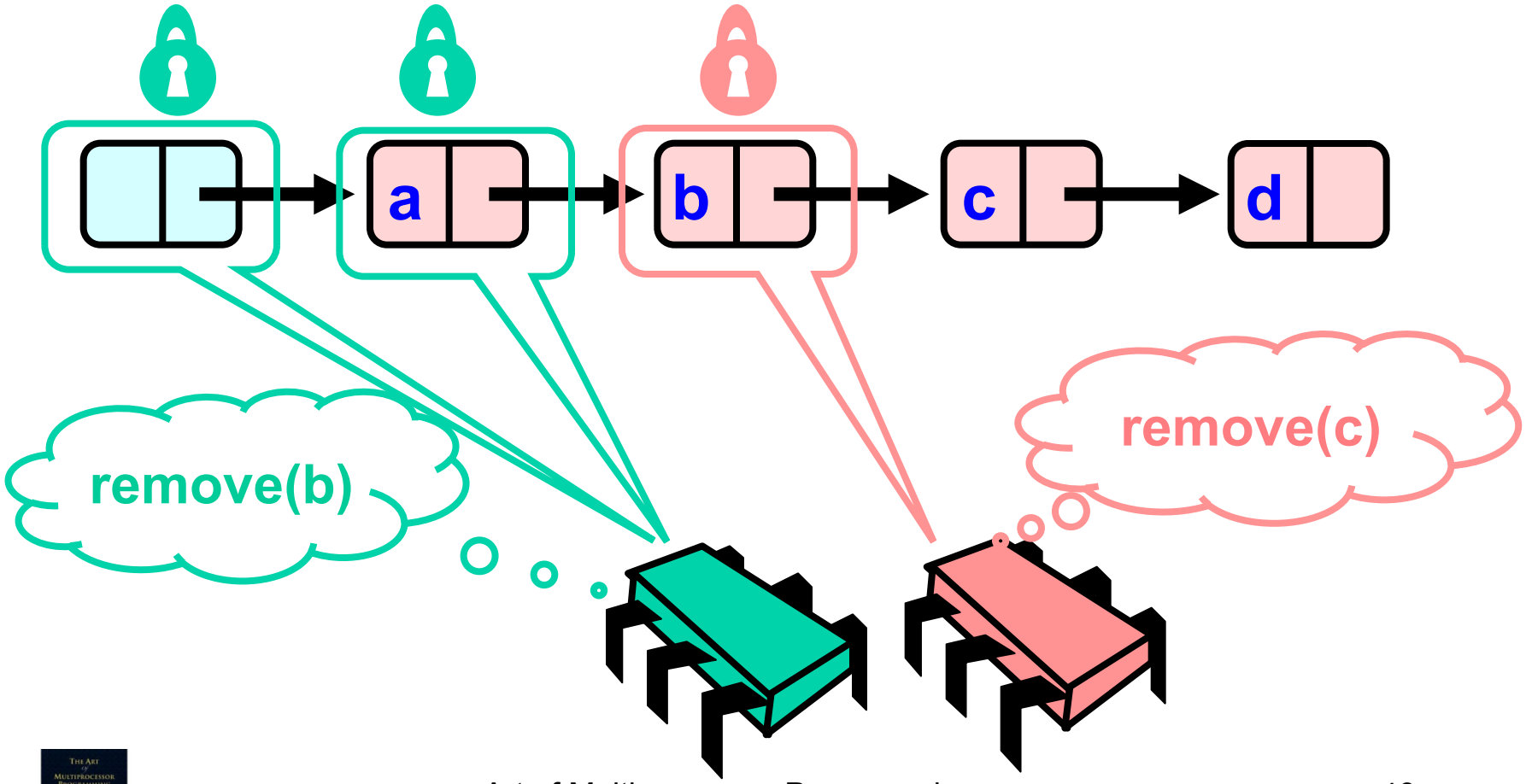


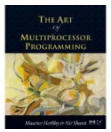
Removing a Node



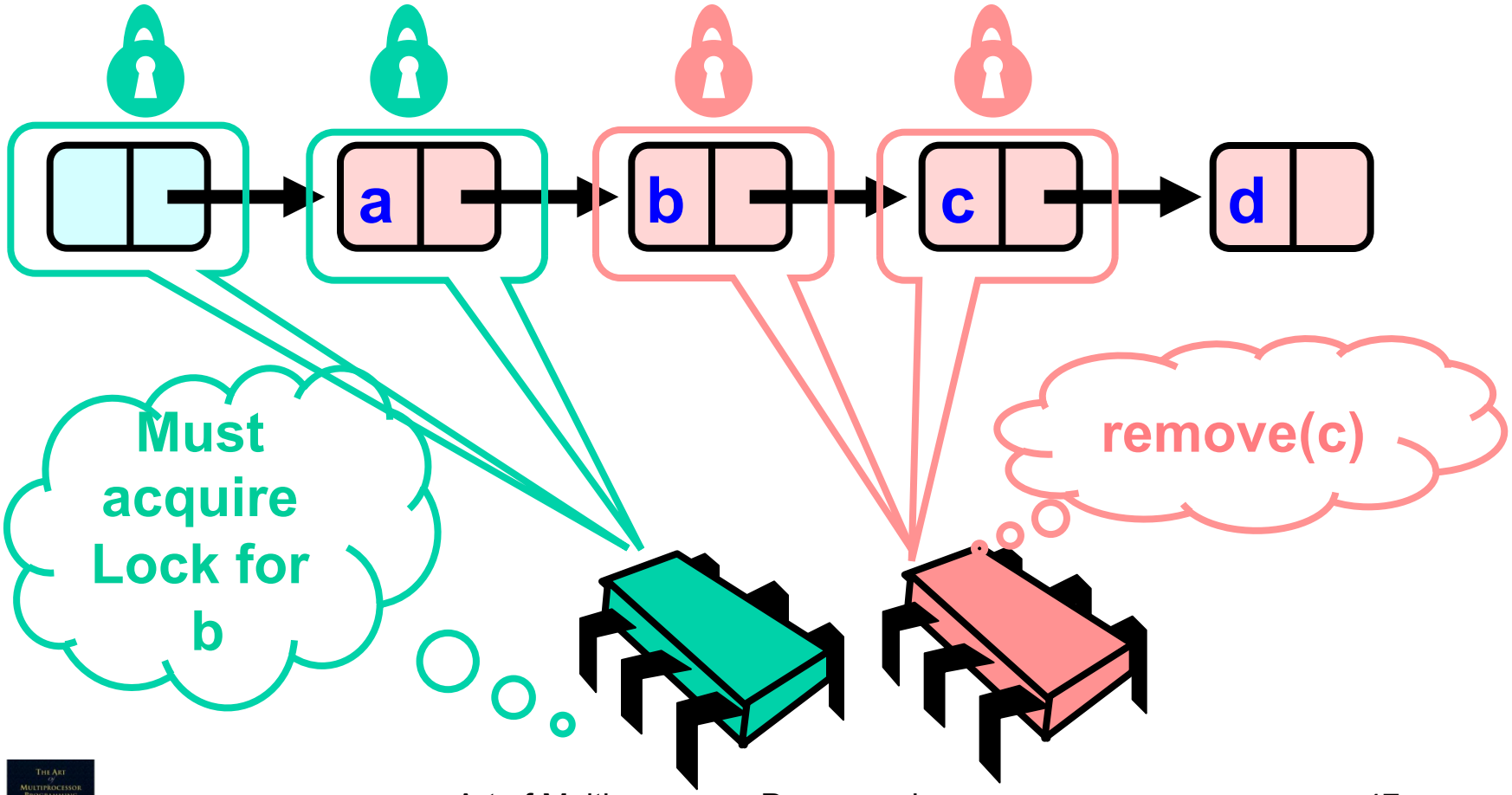


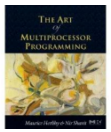
Removing a Node



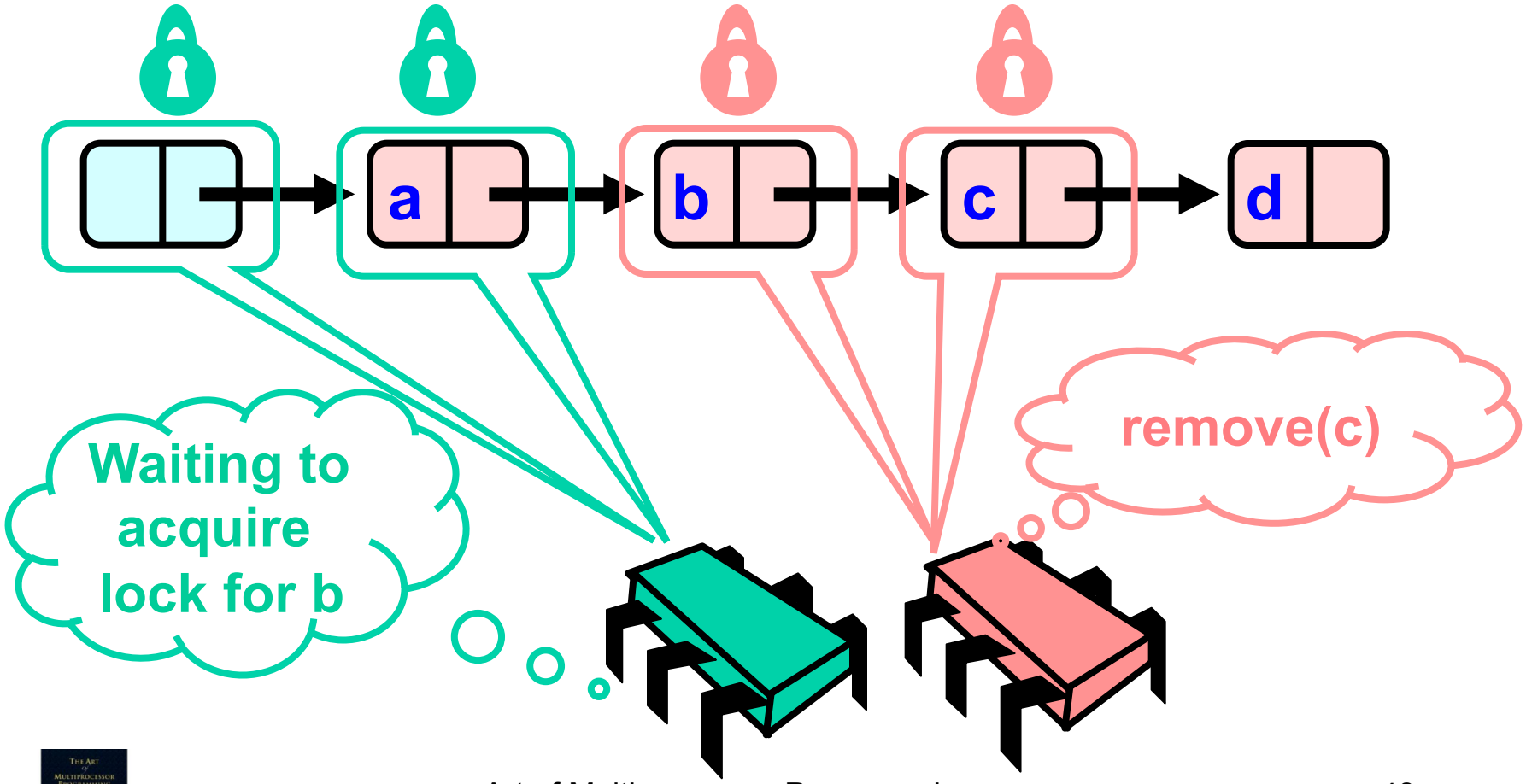


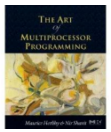
Removing a Node



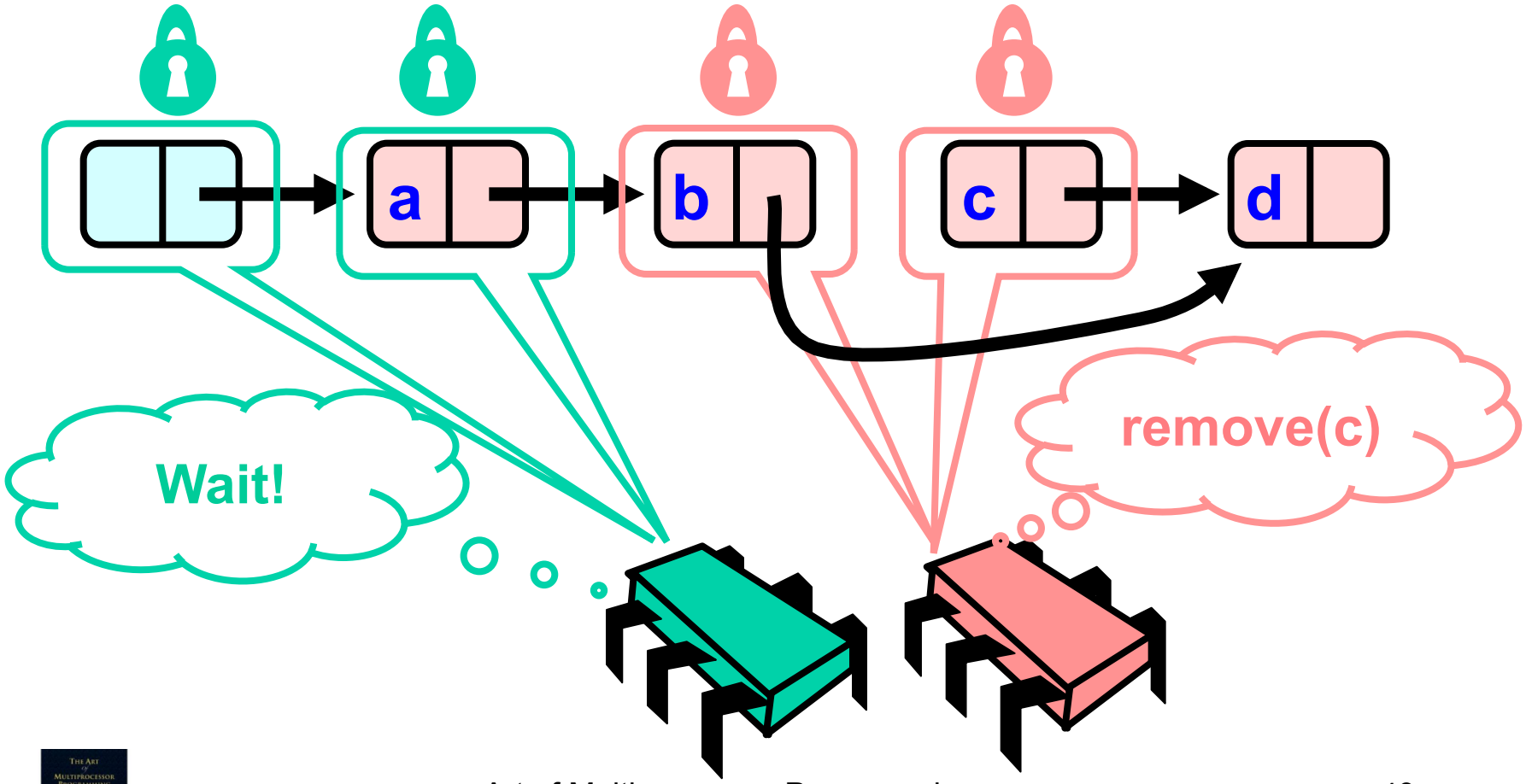


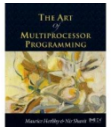
Removing a Node



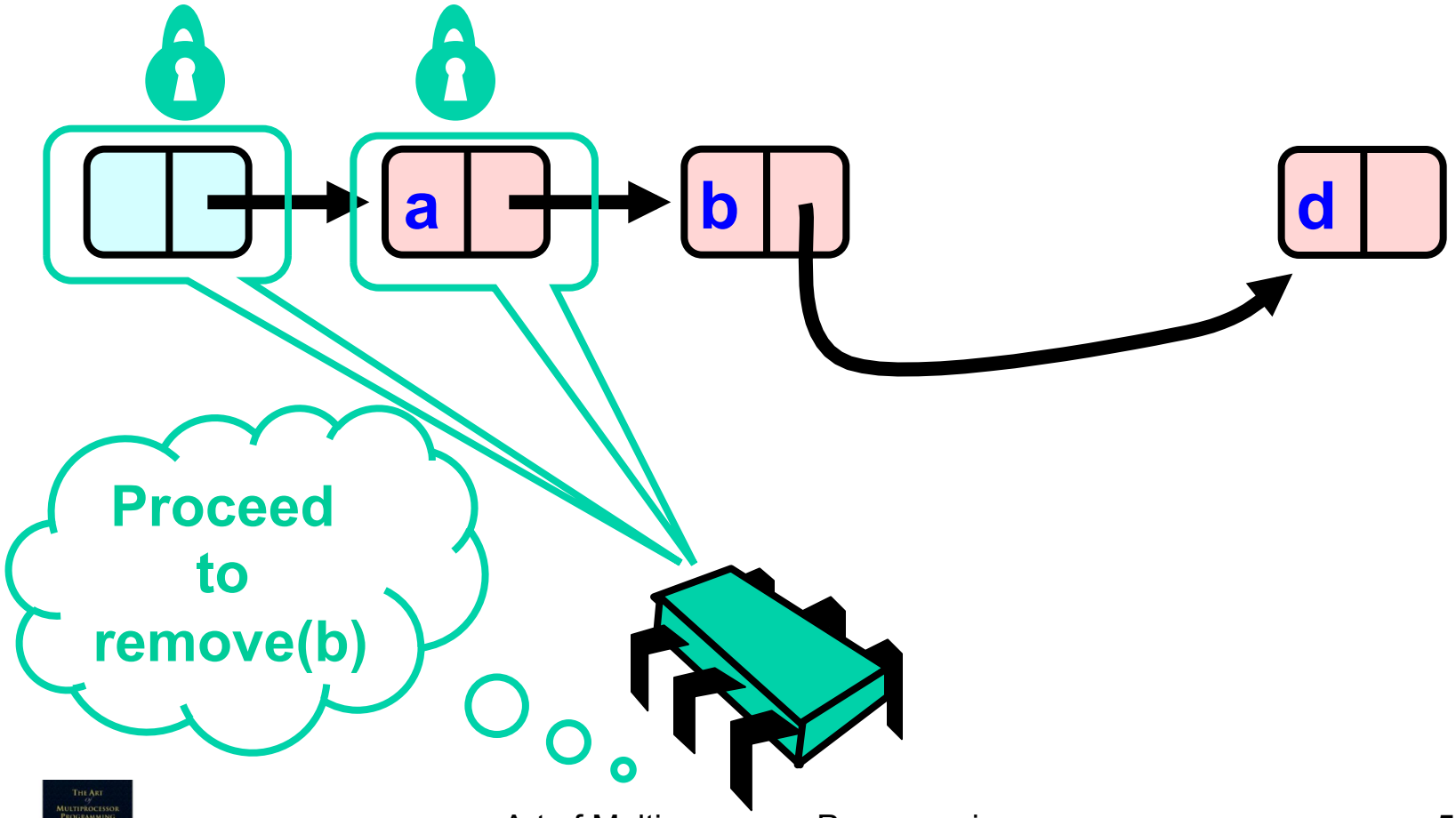


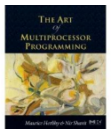
Removing a Node



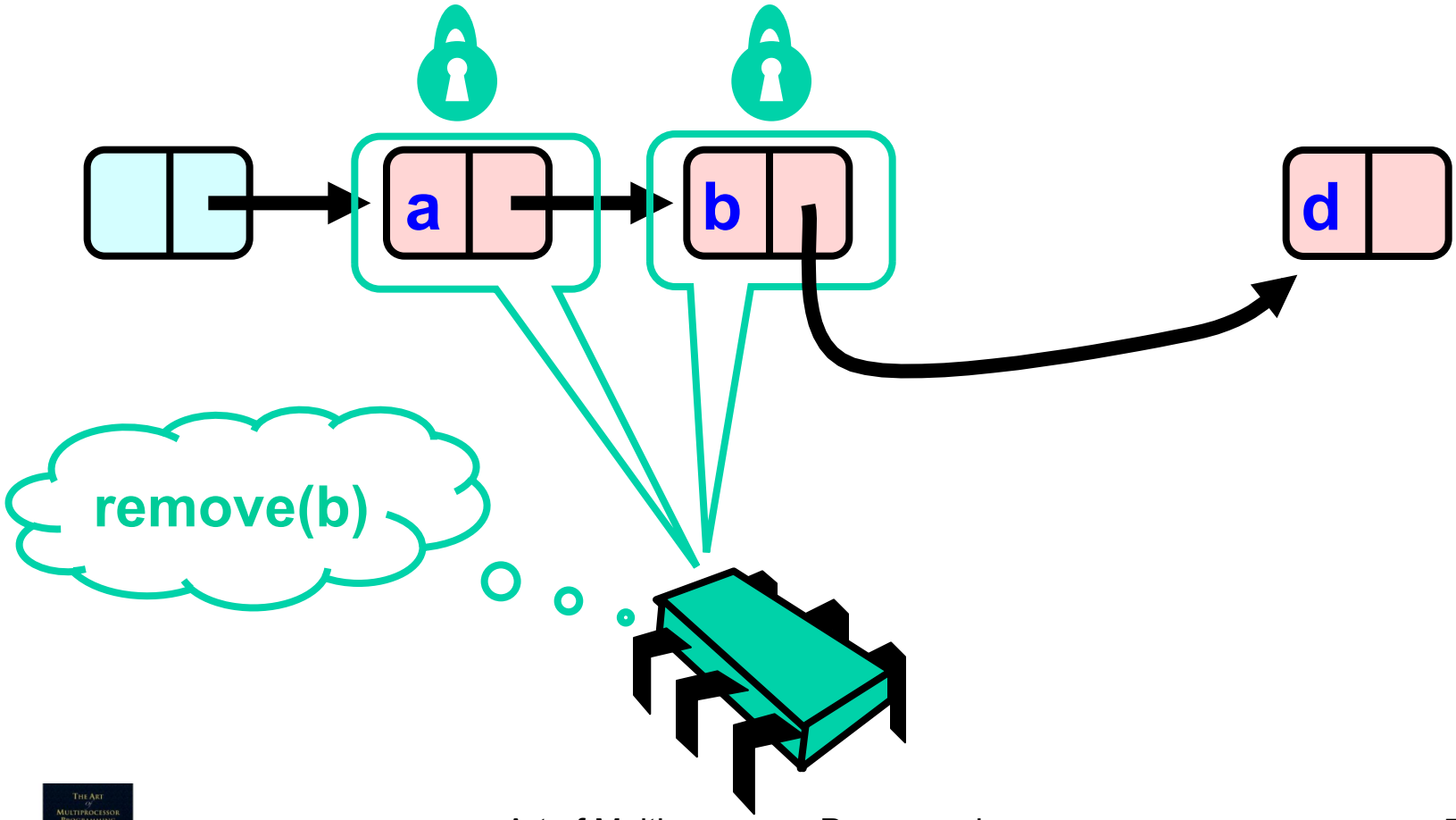


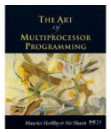
Removing a Node



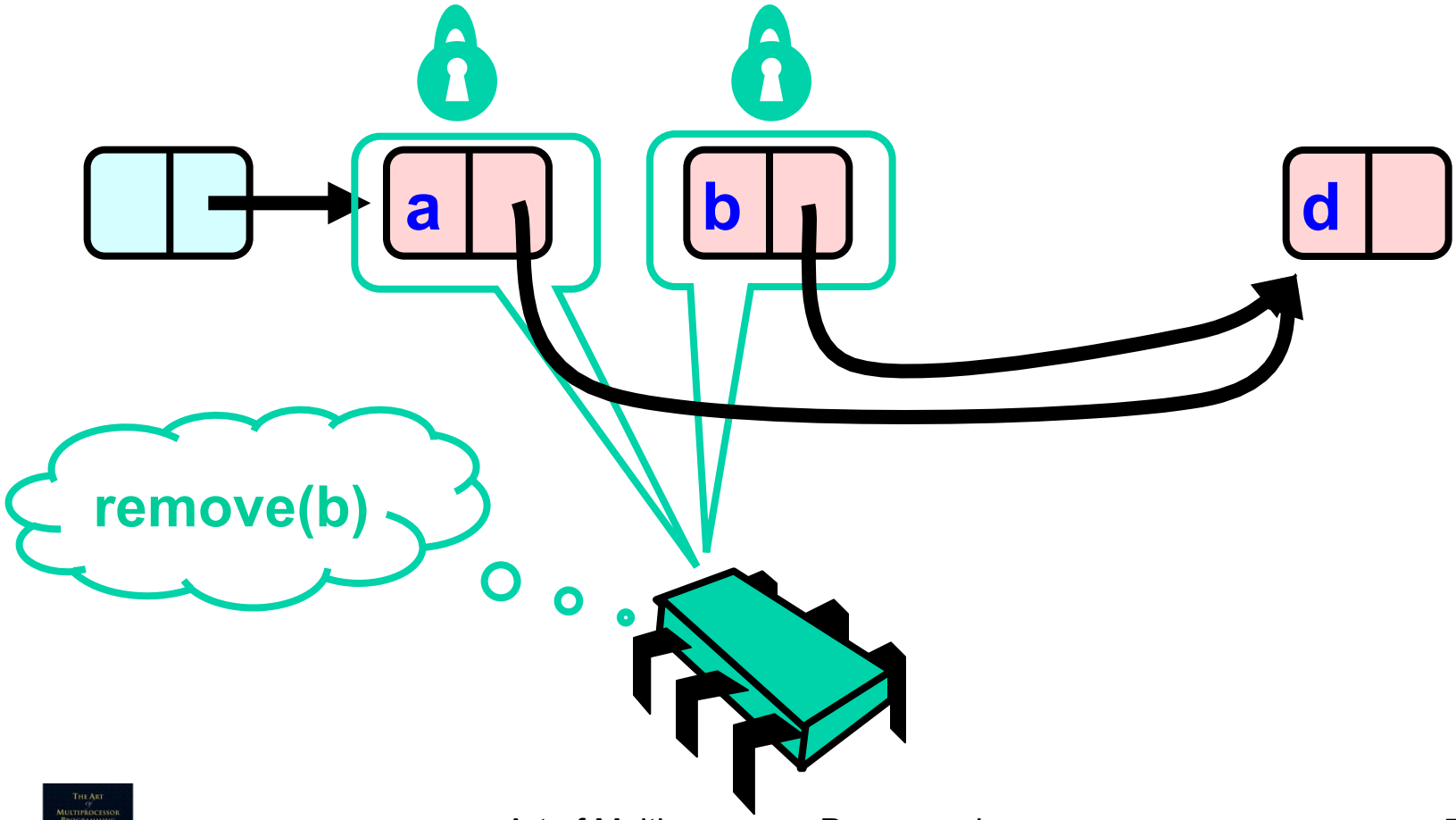


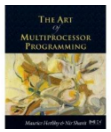
Removing a Node



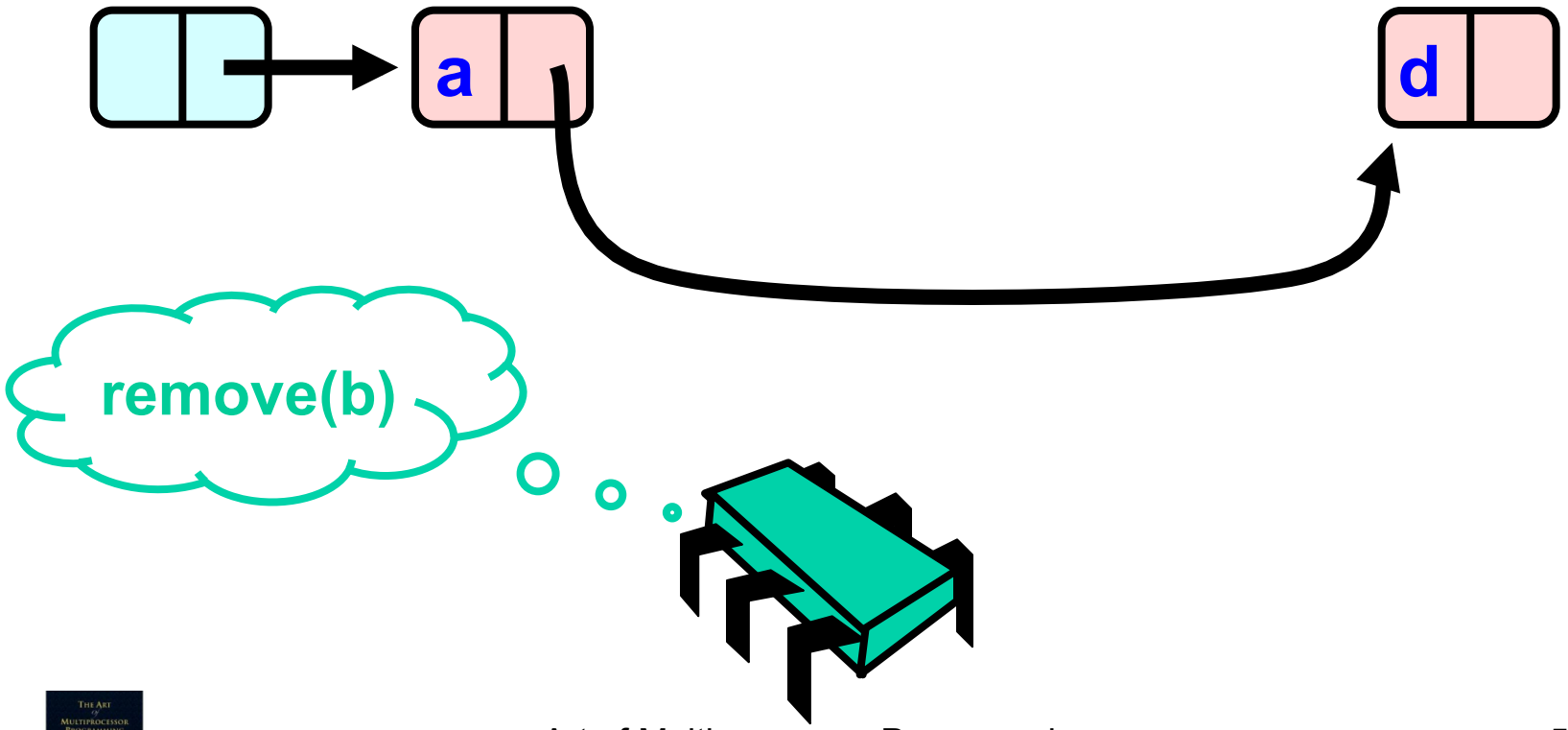


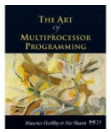
Removing a Node



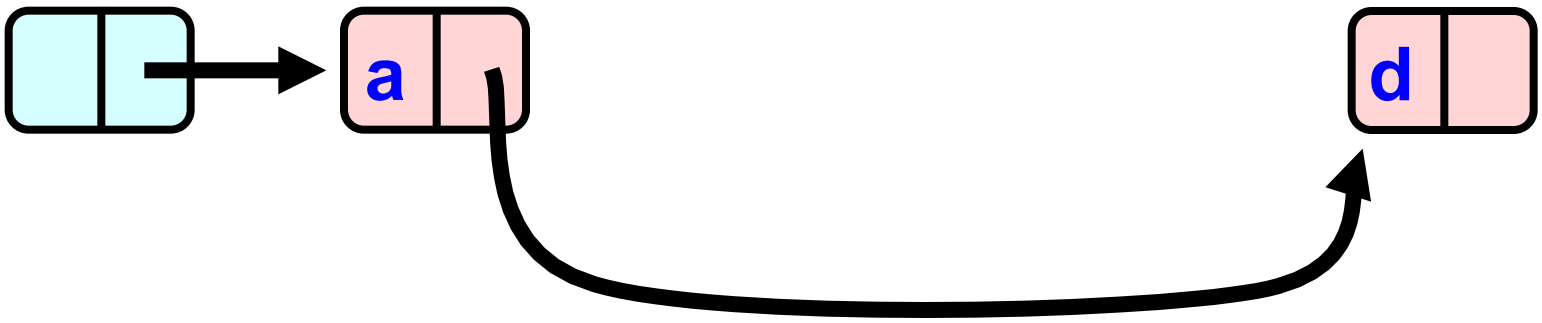


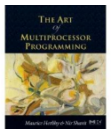
Removing a Node





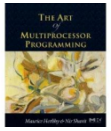
Removing a Node





Remove method

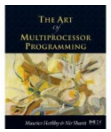
```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
    …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```



Remove method

```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

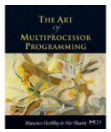
Key used to order node



Remove method

```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  currNode.unlock();
  predNode.unlock();
 }}
```

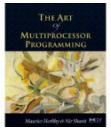
Predecessor and current nodes



Remove method

```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

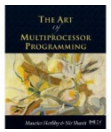
Make sure locks released



Remove method

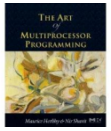
```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   ...
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

Everything else



Remove method

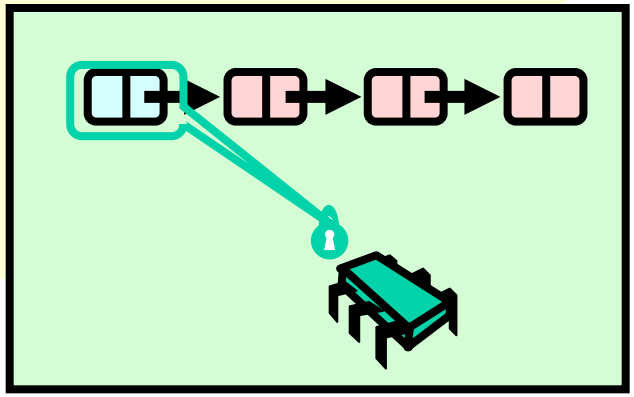
```
try {
 pred = head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

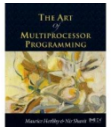


Remove method

lock pred == head

```
try {
  pred = head;
  pred.lock();
  curr = pred.next;
  curr.lock();
  …
} finally { … }
```

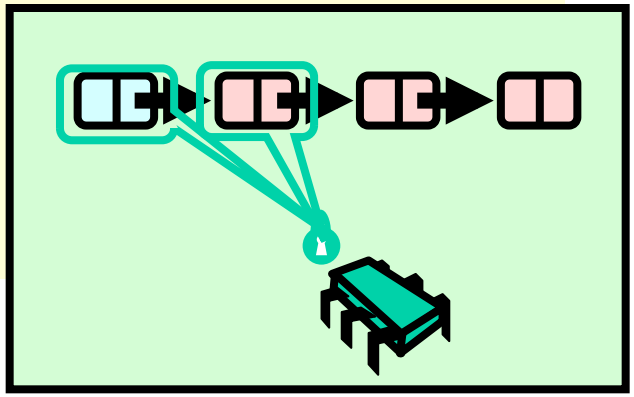




Remove method

```
try {
  pred = head;
  pred.lock();
  curr = pred.next;
  curr.lock();
  …
} finally { … }
```

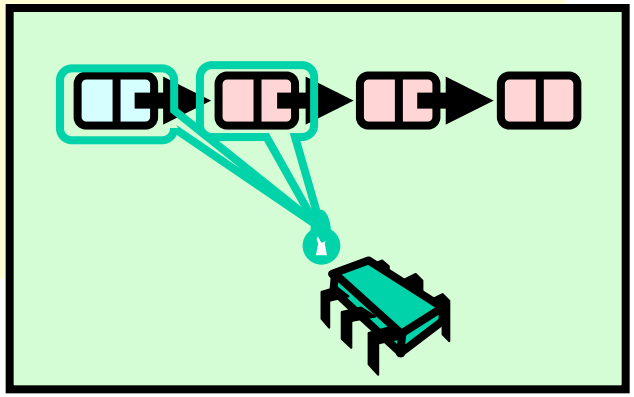
Lock current



Remove method

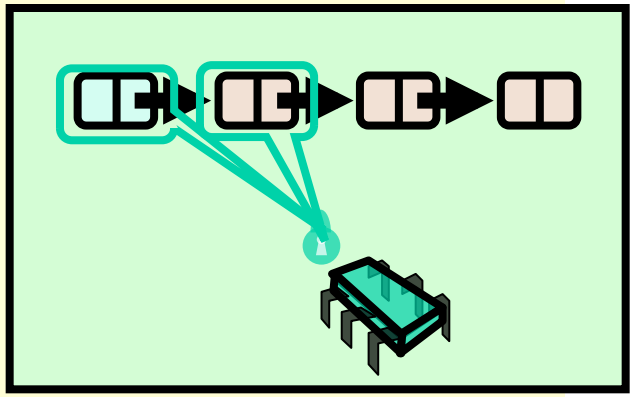
```
try {
 pred = head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

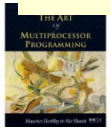
Traversing list



Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

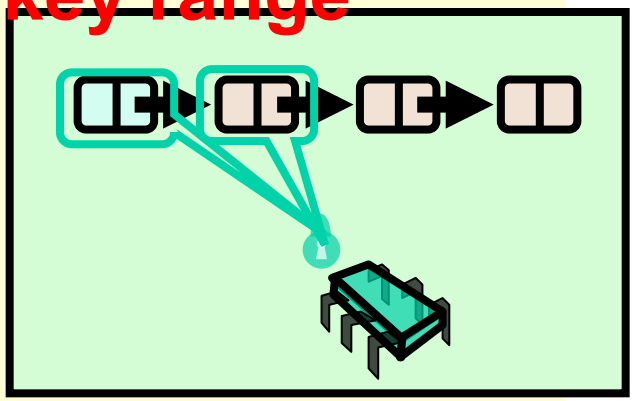


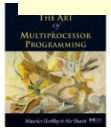


Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Search key range

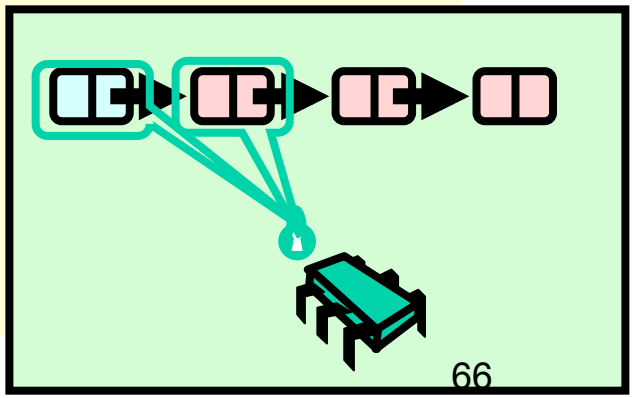


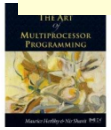


Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

At start of each loop:
curr and pred locked

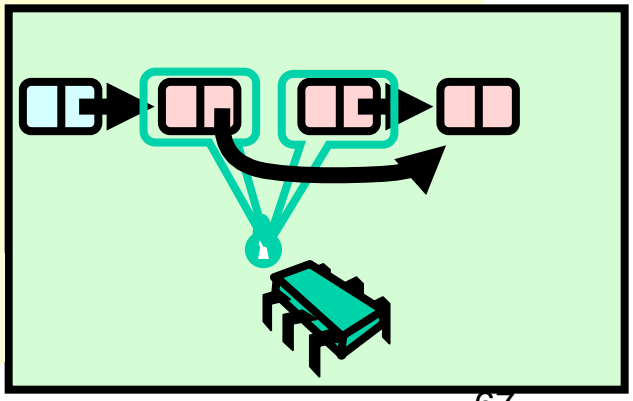


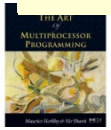


Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
}
```

If item found, remove node

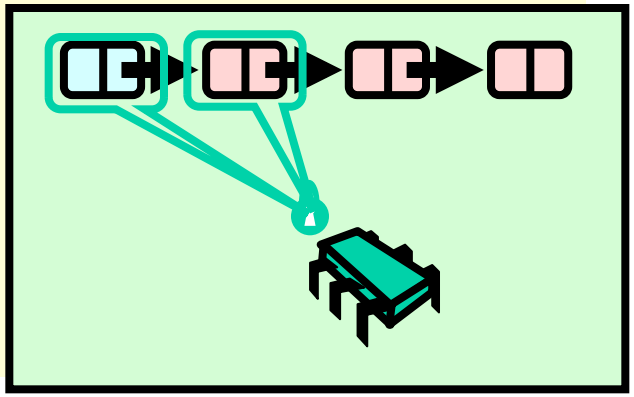


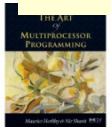


Remove: searching

Unlock predecessor

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

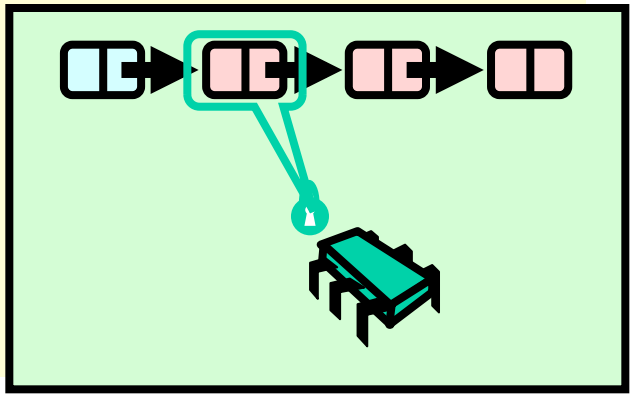


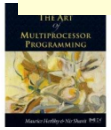


Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
    pred.next = curr.next;
    return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
}
return false;
```

demote current

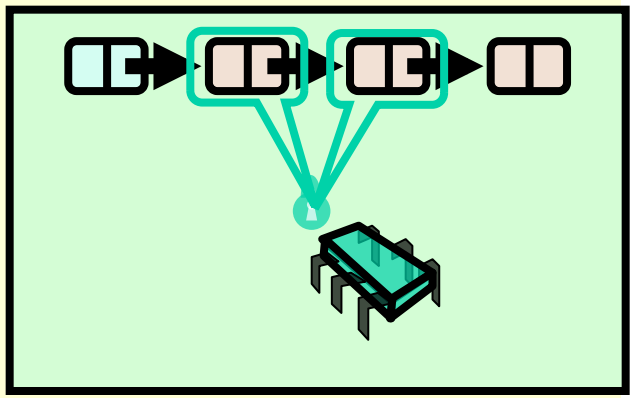




Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
    pred.next = curr.next;
    return true;
  }
  pred.unlock();
  pred = currNode;
  curr = curr.next;
  curr.lock();
}
return false;
```

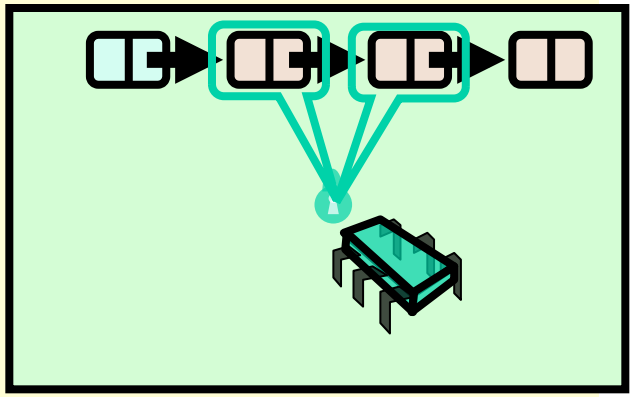
Find and lock new current



Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
    pred.next = curr.next;
    return true;
  }
  pred.unlock();
  pred = currNode;
  curr = curr.next;
  curr.lock();
}
return false;
```

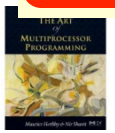
Lock invariant restored



Remove: searching

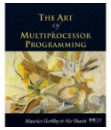
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Otherwise, not present



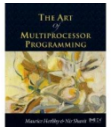
Why does this work?

- To remove node e
  - Must lock e
  - Must lock e's predecessor
- Therefore, if you lock a node
  - It can't be removed
  - And neither can its successor



Adding Nodes

- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted



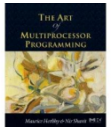
Same Abstraction Map

- S(head) =

  { x | there exists a such that

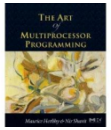
  - a reachable from head and
  - a.item = x

  }



Rep Invariant

- Easy to check that
  - tail always reachable from head
  - Nodes sorted, no duplicates



Drawbacks

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