

## Linked Lists: Locking, Lock-Free, and Beyond ...

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

- Each method locks the object
  - Avoid contention using queue locks
  - Easy to reason about
    - In simple cases
  - Standard Java model
    - Synchronized blocks and methods
- So, are we done?

- 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 ...

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

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

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

Art of Multiprocessor Programming

7

Fourth:

### Lock-Free Synchronization



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

Art of Multiprocessor Programming

8

### Linked List



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

Art of Multiprocessor Programming

9

### 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
- Sentinel nodes
  - tail reachable from head

Art of Multiprocessor Programming

10

### List-Based Sets



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

Art of Multiprocessor Programming

11

### List Node

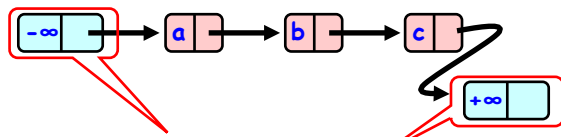


```
public class Node {  
    public T item;  
    public int key;  
    public Node next;  
}
```

Art of Multiprocessor Programming

12

## The List-Based Set



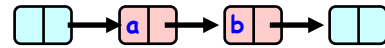
Sorted with Sentinel nodes  
(min & max possible keys)

Art of Multiprocessor Programming

13

## Abstract Data Types

- Concrete representation



- Abstract Type

– {a, b}

Art of Multiprocessor Programming

14

## Abstract Data Types

- Meaning of rep given by abstraction map

–  $S(\text{light blue} \rightarrow \text{a} \rightarrow \text{b} \rightarrow \text{light blue}) = \{a, b\}$

Art of Multiprocessor Programming

15

## Invariants

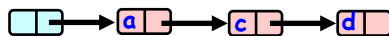
- Sentinel nodes
  - tail reachable from head
- Sorted
- No duplicates

Art of Multiprocessor Programming

16

## Sequential List Based Set

Add()



Remove()

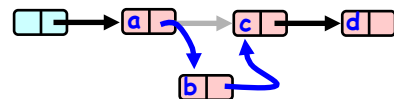


Art of Multiprocessor Programming

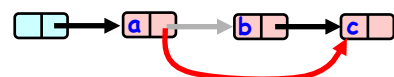
17

## Sequential List Based Set

Add()



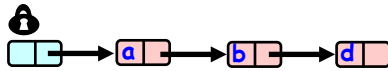
Remove()



Art of Multiprocessor Programming

18

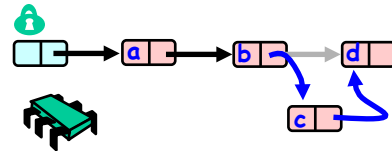
## Course Grained Locking



Art of Multiprocessor Programming

19

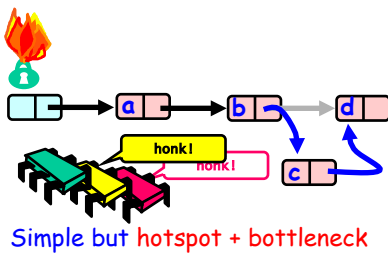
## Course Grained Locking



Art of Multiprocessor Programming

20

## Course Grained Locking



Simple but hotspot + bottleneck

Art of Multiprocessor Programming

21

## 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

Art of Multiprocessor Programming

22

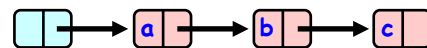
## Fine-grained Locking

- 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

Art of Multiprocessor Programming

23

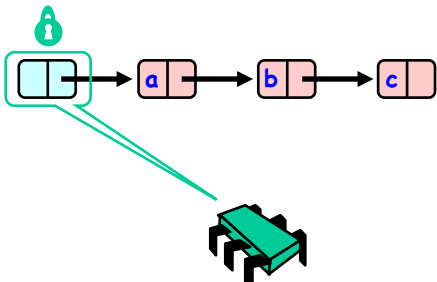
## Hand-over-Hand locking



Art of Multiprocessor Programming

24

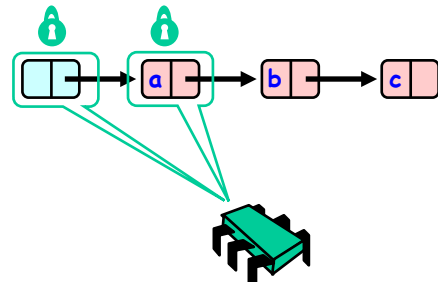
## Hand-over-Hand locking



Art of Multiprocessor Programming

25

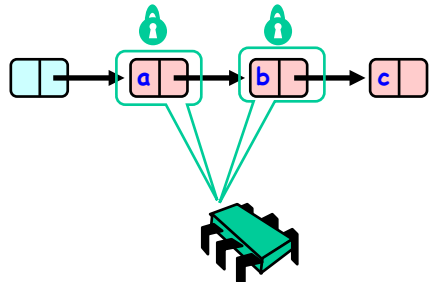
## Hand-over-Hand locking



Art of Multiprocessor Programming

26

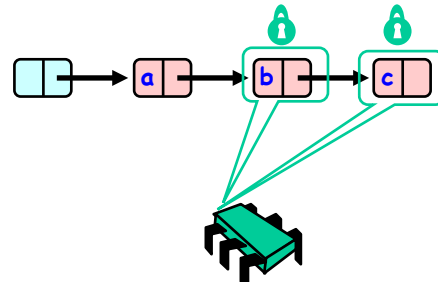
## Hand-over-Hand locking



Art of Multiprocessor Programming

27

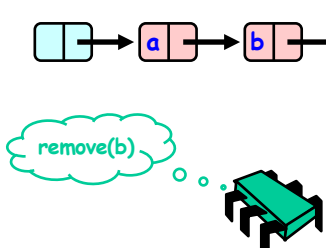
## Hand-over-Hand locking



Art of Multiprocessor Programming

28

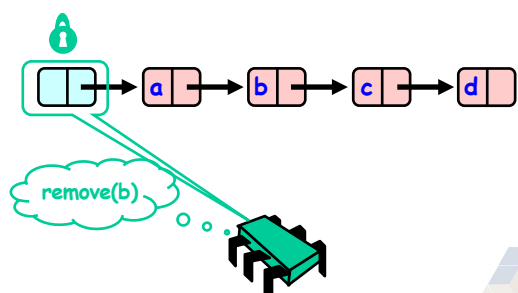
## Removing a Node



Art of Multiprocessor Programming

29

## Removing a Node



Art of Multiprocessor Programming

30

## Removing a Node

İTÜ

remove(b)

Art of Multiprocessor Programming

31

## Removing a Node

İTÜ

remove(b)

Art of Multiprocessor Programming

32

## Removing a Node

İTÜ

remove(b)

Art of Multiprocessor Programming

33

## Removing a Node

İTÜ

remove(b)

Why do we need to always hold 2 locks?

Art of Multiprocessor Programming

34

## Concurrent Removes

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

35

## Concurrent Removes

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

36

### Concurrent Removes

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

37

### Concurrent Removes

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

38

### Concurrent Removes

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

39

### Concurrent Removes

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

40

### Concurrent Removes

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

41

### Concurrent Removes

İTÜ

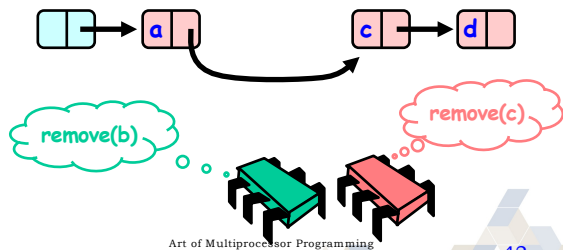
remove(b)

remove(c)

Art of Multiprocessor Programming

42

Uh, Oh



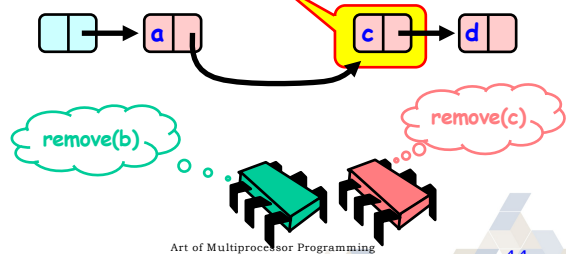
Art of Multiprocessor Programming

43

Uh, Oh



Bad news, C not removed



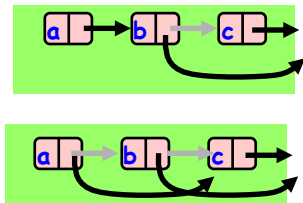
Art of Multiprocessor Programming

44

## 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



Art of Multiprocessor Programming

45

## Insight

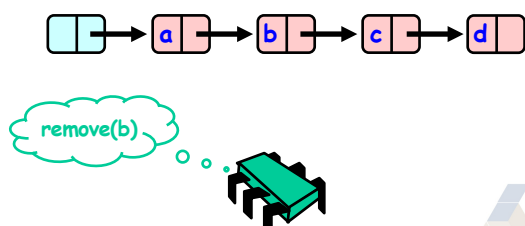


- If a node is locked
  - No one can delete node's successor
- If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works

Art of Multiprocessor Programming

46

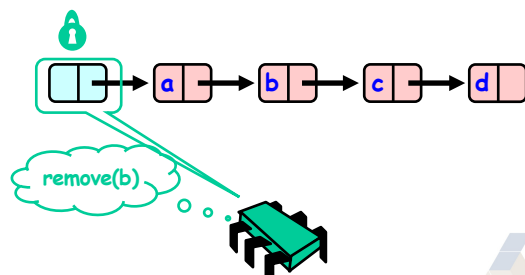
## Hand-Over-Hand Again



Art of Multiprocessor Programming

47

## Hand-Over-Hand Again



Art of Multiprocessor Programming

48



## Hand-Over-Hand Again İTÜ

remove(b)

Art of Multiprocessor Programming

49

## Hand-Over-Hand Again İTÜ

remove(b)

Found it!

Art of Multiprocessor Programming

50

## Hand-Over-Hand Again İTÜ

remove(b)

Found it!

Art of Multiprocessor Programming

51

## Hand-Over-Hand Again İTÜ

remove(b)

Art of Multiprocessor Programming

52

## Removing a Node İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

53

## Removing a Node İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

54

## Removing a Node

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

55

## Removing a Node

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

56

## Removing a Node

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

57

## Removing a Node

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

58

## Removing a Node

İTÜ

remove(b)

remove(c)

Art of Multiprocessor Programming

59

## Removing a Node

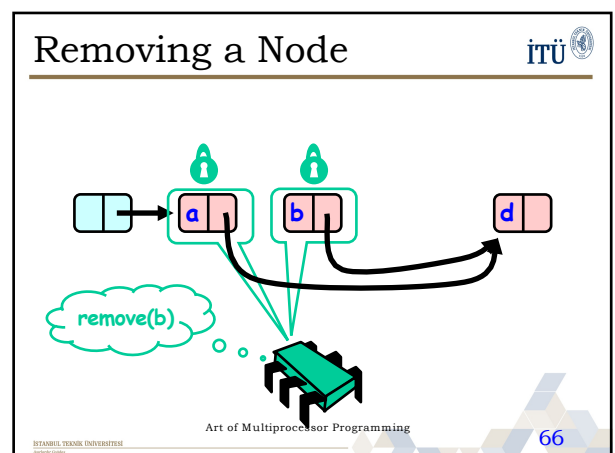
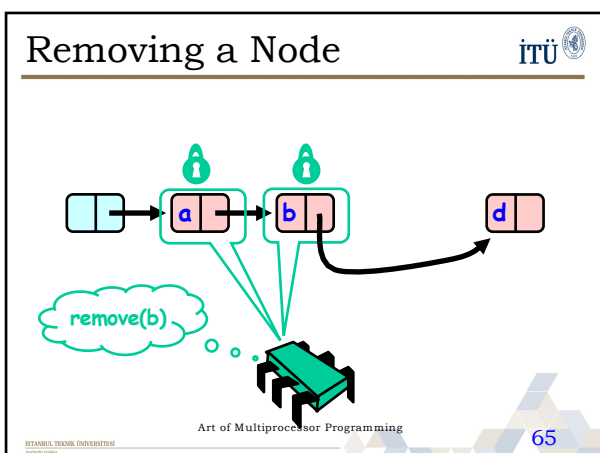
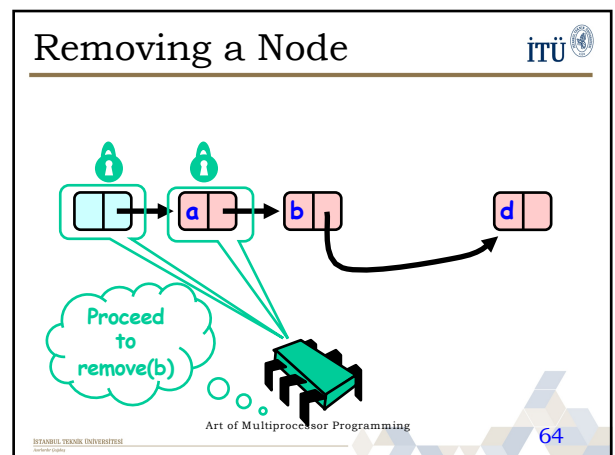
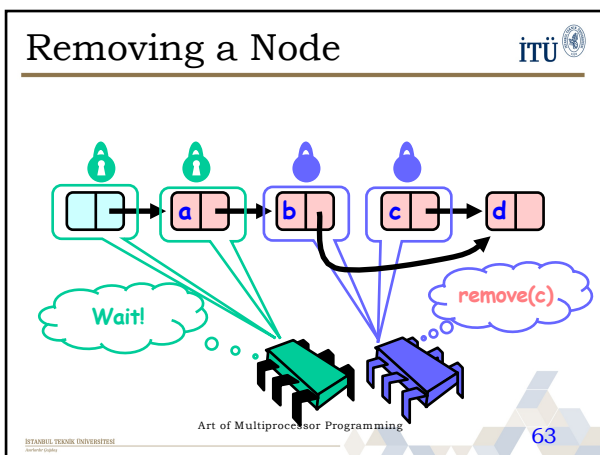
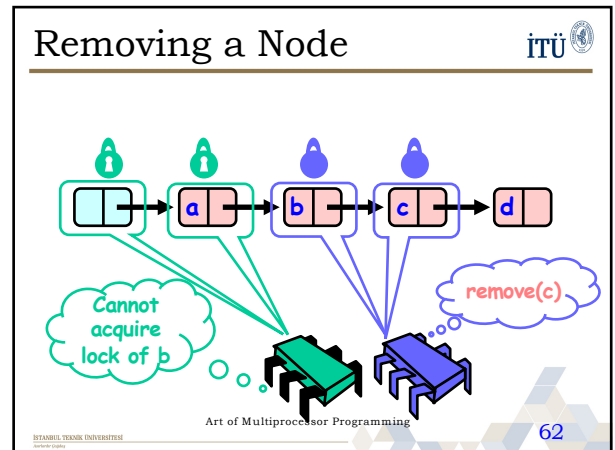
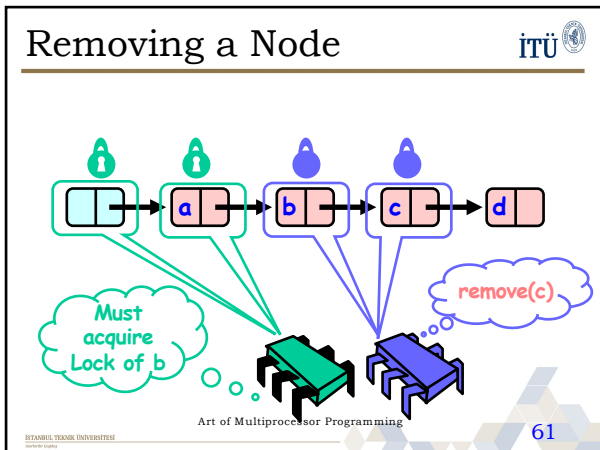
İTÜ

remove(b)

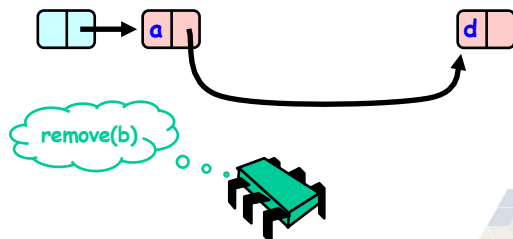
remove(c)

Art of Multiprocessor Programming

60



## Removing a Node



Art of Multiprocessor Programming

67

## Removing a Node



Art of Multiprocessor Programming

68

## Remove method



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

Art of Multiprocessor Programming

69

## Remove method



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

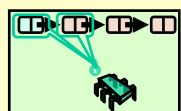
Art of Multiprocessor Programming

70

## 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;
```



Art of Multiprocessor Programming

71

## 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

Art of Multiprocessor Programming

72

## Drawbacks



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

Art of Multiprocessor Programming

73

## Optimistic Synchronization

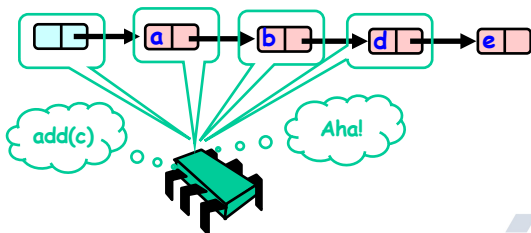


- Find nodes without locking
- Lock nodes
- Check that everything is OK

Art of Multiprocessor Programming

74

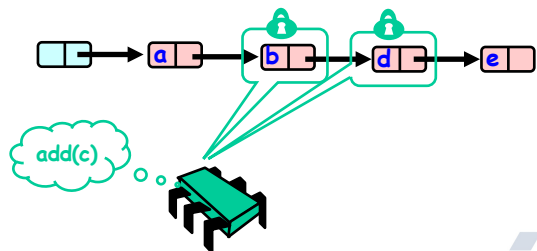
## Optimistic: Traverse without Locking



Art of Multiprocessor Programming

75

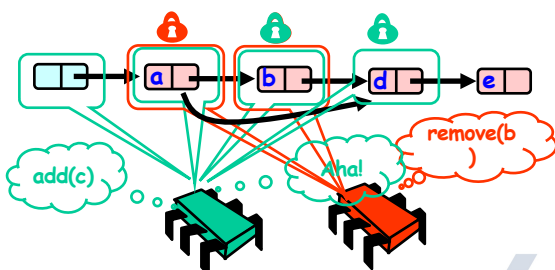
## Optimistic: Lock and Load



Art of Multiprocessor Programming

76

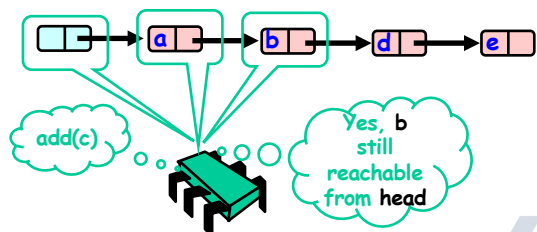
## What could go wrong?



Art of Multiprocessor Programming

77

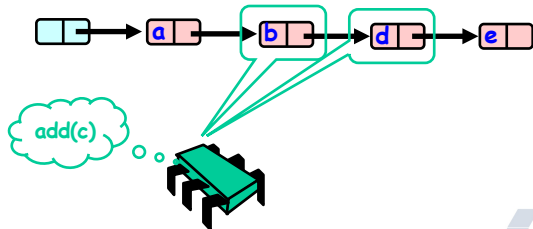
## Validate – Part 1 (while holding locks)



Art of Multiprocessor Programming

78

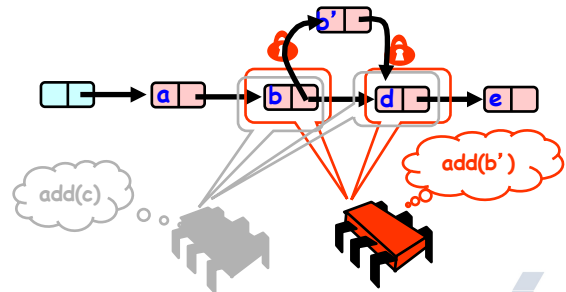
## What Else Can Go Wrong? İTÜ



Art of Multiprocessor Programming

79

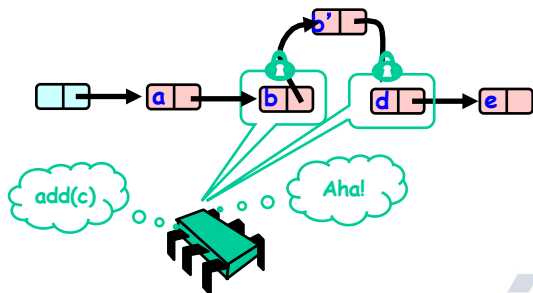
## What Else Can Go Wrong? İTÜ



Art of Multiprocessor Programming

80

## What Else Can Go Wrong? İTÜ

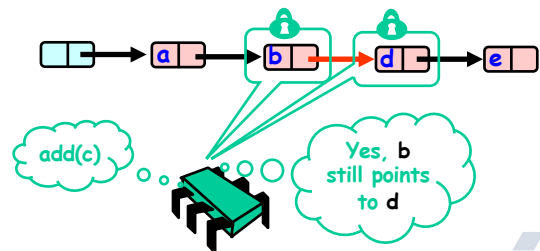


Art of Multiprocessor Programming

81

## Validate Part 2 (while holding locks)

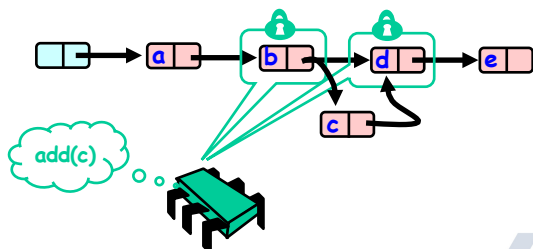
İTÜ



Art of Multiprocessor Programming

82

## Optimistic: Linearization Point İTÜ



Art of Multiprocessor Programming

83

## Invariants

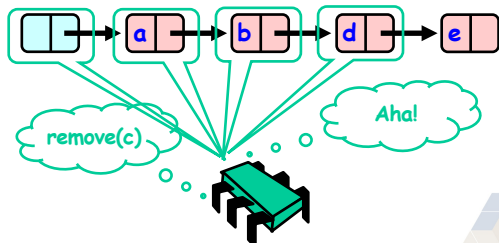
İTÜ

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

Art of Multiprocessor Programming

84

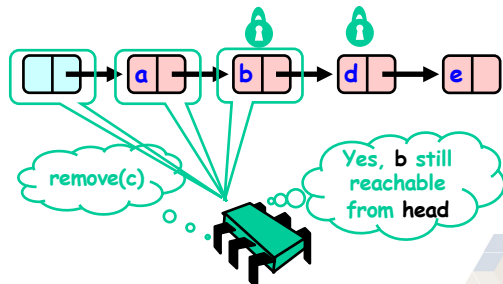
## Unsuccessful Remove



Art of Multiprocessor Programming

85

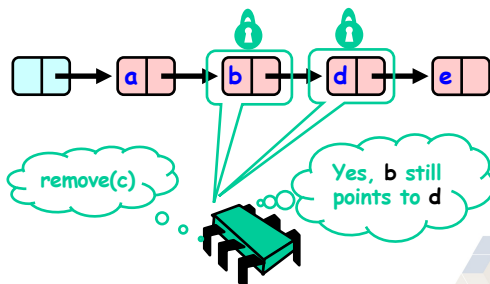
## Validate (1)



Art of Multiprocessor Programming

86

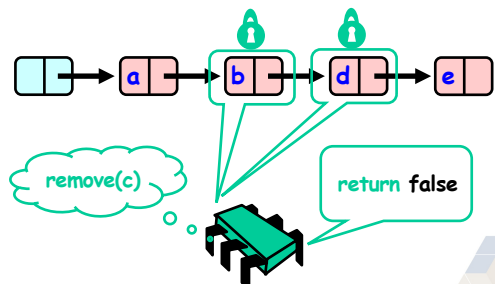
## Validate (2)



Art of Multiprocessor Programming

87

## OK Computer



Art of Multiprocessor Programming

88

## Validation



```
private boolean validate(Node pred, Node curr)
{
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}
```

Art of Multiprocessor Programming

89

## Remove: searching



```
public boolean remove(Item item) {
    int key = item.hashCode();
    while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        }
        ...
    }
}
```

Art of Multiprocessor Programming

90

## On Exit from Loop



- If item is present
  - curr holds item
  - pred just before curr
- If item is absent
  - curr has first higher key
  - pred just before curr
- Assuming no synchronization problems

Art of Multiprocessor Programming

91

## Remove Method



```
try {
    pred.lock();
    curr.lock();
    if (validate(pred, curr) {
        if (curr.item == item) {
            pred.next = curr.next;
            return true;
        } else {
            return false;
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

Art of Multiprocessor Programming

92

## Optimistic List



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

Art of Multiprocessor Programming

93

## So Far, So Good



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

Art of Multiprocessor Programming

94

## 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

Art of Multiprocessor Programming

95

## Lazy List



- Like optimistic, except
  - Scan once
  - contains(x) never locks ...
- Key insight
  - Removing nodes causes trouble
  - Do it "lazily"

Art of Multiprocessor Programming

96



## 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)

Art of Multiprocessor Programming

97

## Lazy Removal



Art of Multiprocessor Programming

98

## Lazy Removal



Present in list

Art of Multiprocessor Programming

99

## Lazy Removal



Logically deleted

Art of Multiprocessor Programming

100

## Lazy Removal



Physically deleted

Art of Multiprocessor Programming

101

## Lazy Removal



Physically deleted

Art of Multiprocessor Programming

102

## 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.

Art of Multiprocessor Programming

103

## Validation

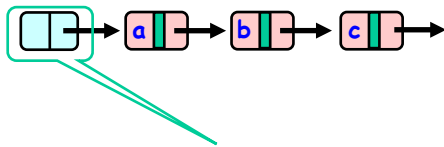


- No need to rescan list!
- Check that pred is not marked
- Check that curr is not marked
- Check that pred points to curr

Art of Multiprocessor Programming

104

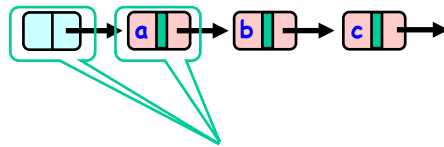
## Business as Usual



Art of Multiprocessor Programming

105

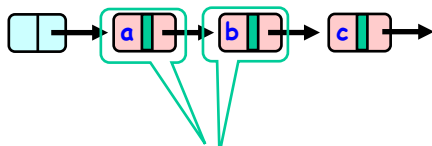
## Business as Usual



Art of Multiprocessor Programming

106

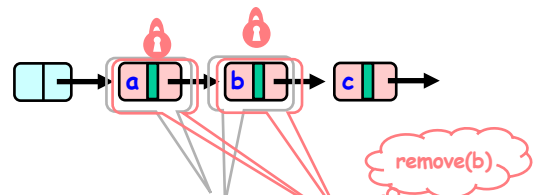
## Business as Usual



Art of Multiprocessor Programming

107

## Business as Usual



Art of Multiprocessor Programming

108

## Business as Usual

İTÜ

Art of Multiprocessor Programming

109

## Business as Usual

İTÜ

Art of Multiprocessor Programming

110

## Business as Usual

İTÜ

Art of Multiprocessor Programming

111

## Business as Usual

İTÜ

Art of Multiprocessor Programming

112

## Business as Usual

İTÜ

Art of Multiprocessor Programming

113

## Validation

İTÜ

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

Art of Multiprocessor Programming

114

## Remove



```
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr) {
        if (curr.key == key) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } finally {
        pred.unlock();
        curr.unlock();
    }
}
```

Art of Multiprocessor Programming

115

## Contains

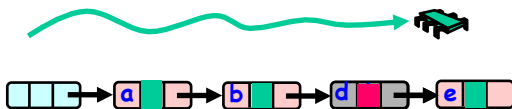


```
public boolean contains(Item item) {
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
```

Art of Multiprocessor Programming

116

## Summary: Wait-free Contains



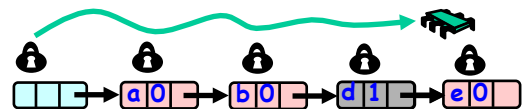
Use Mark bit + Fact that List is ordered

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

Art of Multiprocessor Programming

117

## Lazy List



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

Art of Multiprocessor Programming

118

## 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

Art of Multiprocessor Programming

119

## 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....

Art of Multiprocessor Programming

120

## Lock-free Lists

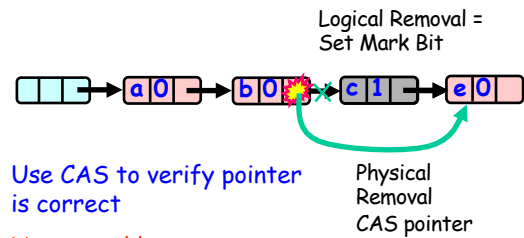


- Next logical step
- Eliminate locking entirely
- contains() wait-free and add() and remove() lock-free
- Use only compareAndSet()
- What could go wrong?

Art of Multiprocessor Programming

121

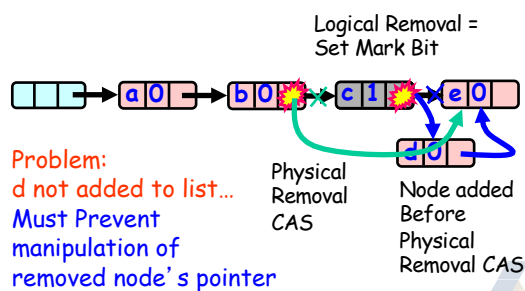
## Remove Using CAS



Art of Multiprocessor Programming

122

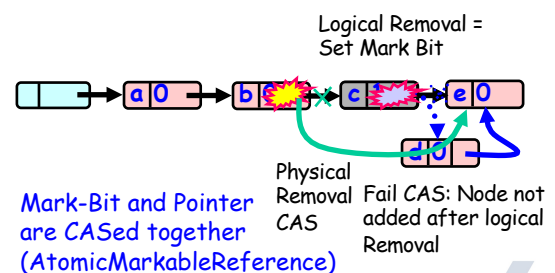
## Problem...



Art of Multiprocessor Programming

123

## The Solution: Combine Bit and Pointer



Art of Multiprocessor Programming

124

## Solution



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

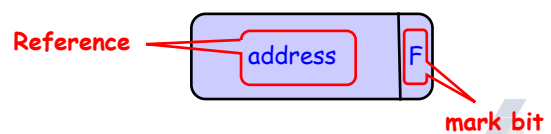
Art of Multiprocessor Programming

125

## Marking a Node



- AtomicMarkableReference class
  - Java.util.concurrent.atomic package



Art of Multiprocessor Programming

126

## Extracting Reference & Mark



```
Public Object get(boolean[] marked);
```

Art of Multiprocessor Programming

127

## Extracting Reference Only



```
public boolean isMarked();
```

Value of  
mark

Art of Multiprocessor Programming

128

## Changing State



```
Public boolean compareAndSet(  
Object expectedRef,  
Object updateRef,  
boolean expectedMark,  
boolean updateMark);
```

Art of Multiprocessor Programming

129

## Changing State

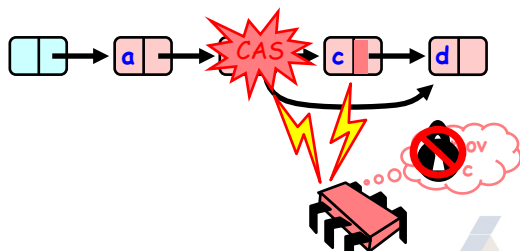


```
public boolean attemptMark(  
Object expectedRef,  
boolean updateMark);
```

Art of Multiprocessor Programming

130

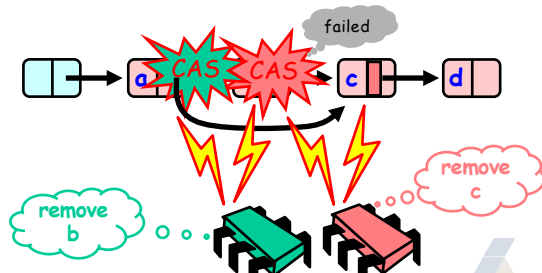
## Removing a Node



Art of Multiprocessor Programming

131

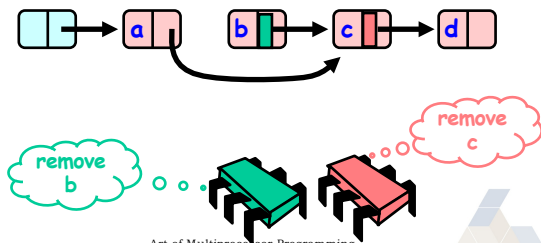
## Removing a Node



Art of Multiprocessor Programming

132

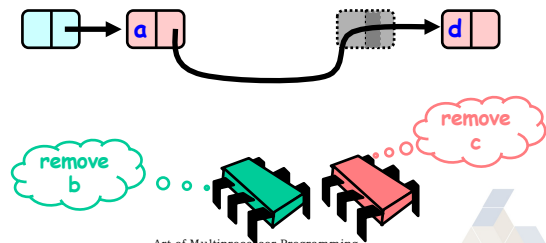
## Removing a Node



Art of Multiprocessor Programming

133

## Removing a Node



Art of Multiprocessor Programming

134

## Traversing the List

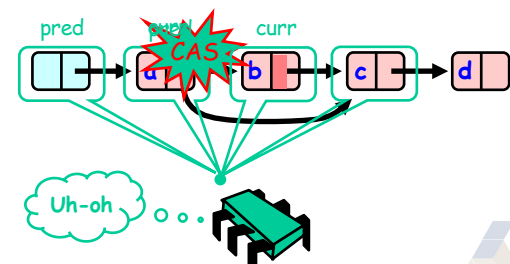


- Q: what do you do when you find a "logically" deleted node in your path?
- A: finish the job.
  - CAS the predecessor's next field
  - Proceed (repeat as needed)

Art of Multiprocessor Programming

135

## Lock-Free Traversal (only Add and Remove)



Art of Multiprocessor Programming

136

## The Window Class



```
class Window {
public Node pred;
public Node curr;
Window(Node pred, Node curr) {
    this.pred = pred;
    this.curr = curr;
}
}
```

Art of Multiprocessor Programming

137

## Using the Find Method



```
window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

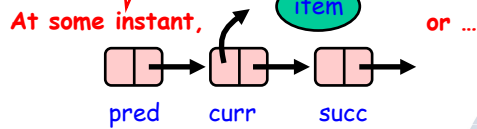
Art of Multiprocessor Programming

138

## The Find Method



`window window = find(item);`



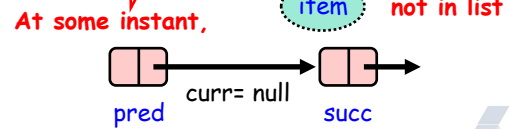
İTAMUL TEKNİK UNIVERSİTESİ

139

## The Find Method



`window window = find(item);`



İTAMUL TEKNİK UNIVERSİTESİ

140

## Lock-free Find



```
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false};
    boolean snip;
    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);
            while (marked[0]) {
                // ...
            }
            if (curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
        }
    }
}
```

İTAMUL TEKNİK UNIVERSİTESİ

141

## Lock-free Find



```
retry: while (true) {
    ...
    while (marked[0]) {
        snip = pred.next.compareAndSet(curr, succ,
                                         false, false);
        if (!snip) continue retry;
        curr = succ;
        succ = curr.next.get(marked);
    }
    ...
}
```

Art of Multiprocessor Programming

142

## Remove



```
public boolean remove(T item) {
    boolean snip;
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key != key) {
            return false;
        } else {
            Node succ = curr.next.getReference();
            snip = curr.next.attemptMark(succ, true);
            if (!snip)
                continue;
            pred.next.compareAndSet(curr, succ, false, false);
            return true;
        }
    }
}
```

İTAMUL TEKNİK UNIVERSİTESİ

Art of Multiprocessor Programming

143

## Add



```
public boolean add(T item) {
    boolean splice;
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key == key) {
            return false;
        } else {
            Node node = new Node(item);
            node.next = new AtomicMarkableRef(curr, false);
            if (pred.next.compareAndSet(curr, node, false, false))
                return true;
        }
    }
}
```

İTAMUL TEKNİK UNIVERSİTESİ

Art of Multiprocessor Programming

144



## Wait-free Contains



```
public boolean contains(Tt item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key)
        curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
}
```

Art of Multiprocessor Programming

145

## Performance

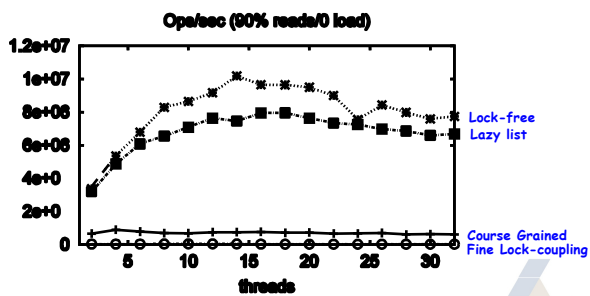


On 16 node shared memory machine benchmark throughput of Java List-based Set algs. vary % of Contains() method Calls.

Art of Multiprocessor Programming

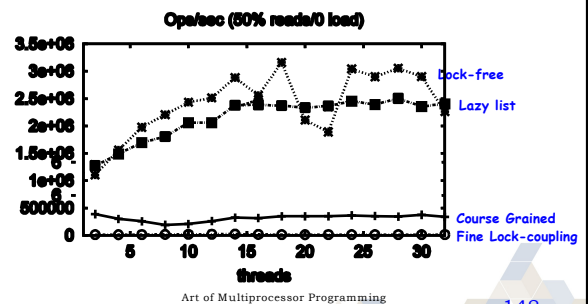
146

## High Contains Ratio



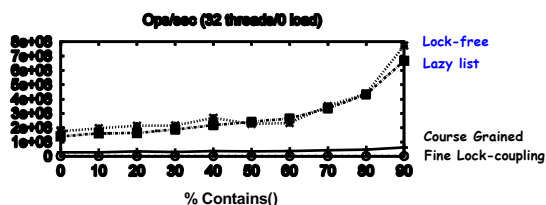
147

## Low Contains Ratio



148

## As Contains Ratio Increases



149

## Summary



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

Art of Multiprocessor Programming

150

## “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

Art of Multiprocessor Programming

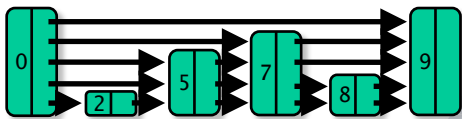
151

İTÜ 

## Concurrent Skip Lists

## Skip Lists

- Probabilistic Data Structure
- No global rebalancing
- Logarithmic-time search



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Skip List Property

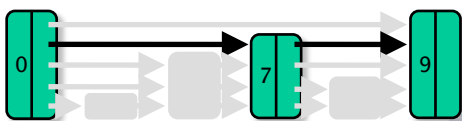
- Each layer is sublist of lower-levels



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Skip List Property

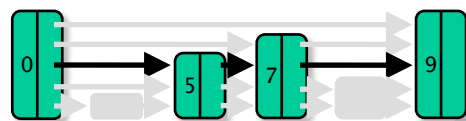
- Each layer is sublist of lower-levels



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Skip List Property

- Each layer is sublist of lower-levels

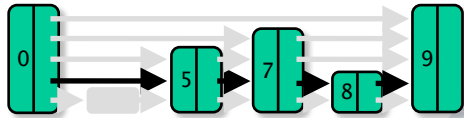


Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Skip List Property



- Each layer is sublist of lower-levels

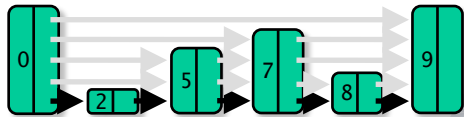


Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Skip List Property



- Each layer is sublist of lower-levels
- Not easy to preserve in concurrent implementations ...

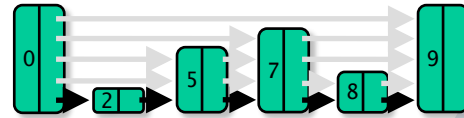


Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Skip List Property

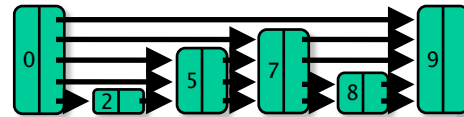


- Each layer is sublist of lower-levels
- Lowest level is entire list



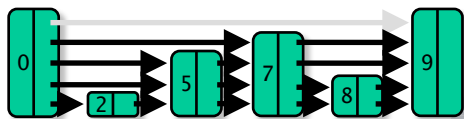
Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Search



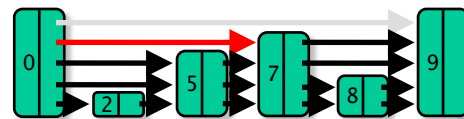
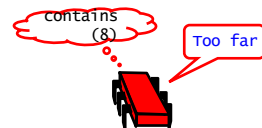
Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Search



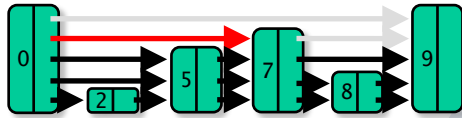
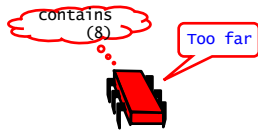
Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Search



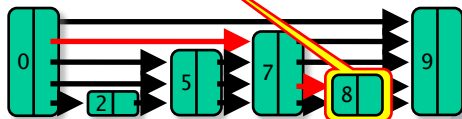
Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Search



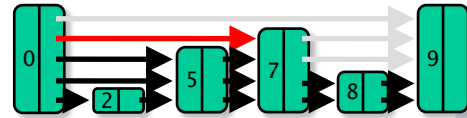
Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Search



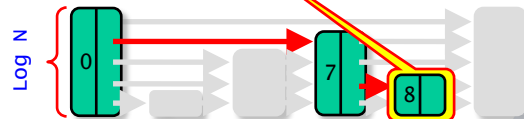
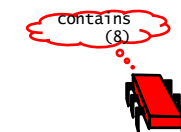
Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Search



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Logarithmic

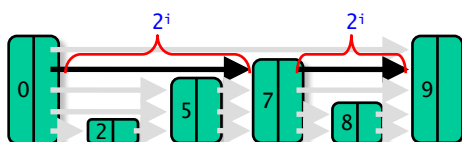


Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Why Logarithmic



- Property: Each pointer at layer  $i$  jumps over roughly  $2^i$  nodes
- Pick node heights randomly so property guaranteed probabilistically



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Find() -- Sequential



```
int find(T x, Node<T>[] preds, Node<T>[] succs)
{
    ...
}
```

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Find() -- Sequential



```
int find(T x, Node<T>[] preds, Node<T>[] succs)
{
    ..
}
```

object height  
(-1 if not there)

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Find() -- Sequential



```
int find(T x, Node<T>[] preds, Node<T>[] succs)
{
    ..
}
```

object height  
(-1 if not there)  
Object sought

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Find() -- Sequential



```
int find(T x, Node<T>[] preds, Node<T>[] succs)
{
    ..
}
```

object height  
(-1 if not there)  
Object sought  
return predecessors

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Find() -- Sequential

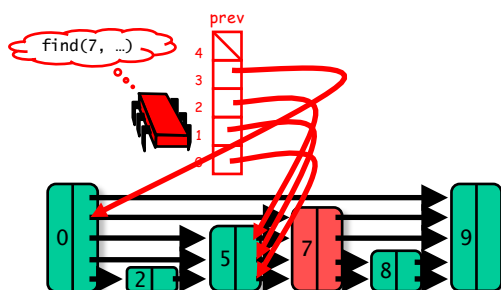


```
int find(T x, Node<T>[] preds, Node<T>[] succs)
{
    ..
}
```

object height  
(-1 if not there)  
Object sought  
return predecessors  
return successors

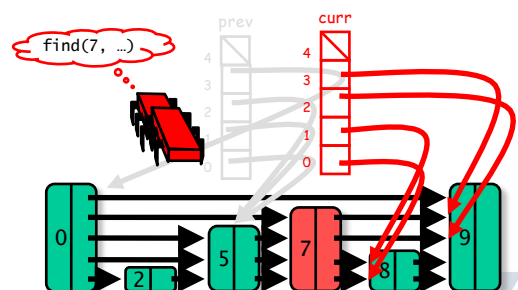
Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Node predecessors



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Node successors



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Lazy Skip List



- Mix blocking and non-blocking techniques:
  - Use optimistic-lazy locking for add() and remove()
  - Wait-free contains()
- Remember: typically lots of contains() calls but few add() and remove()

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Review: Lazy List Remove



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Review: Lazy List Remove



Present in list

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Review: Lazy List Remove



Logically deleted

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Review: Lazy List Remove



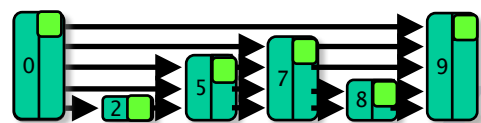
Physically deleted

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Lazy Skip Lists



- Use a mark bit for logical deletion



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

### add(6)

- Create node of (random) height 4

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

### add(6)

- find() predecessors

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

### add(6)

- find() predecessors
- Lock them

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

### add(6)

- find() predecessors
- Lock them
- Validate

optimistic approach

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

### add(6)

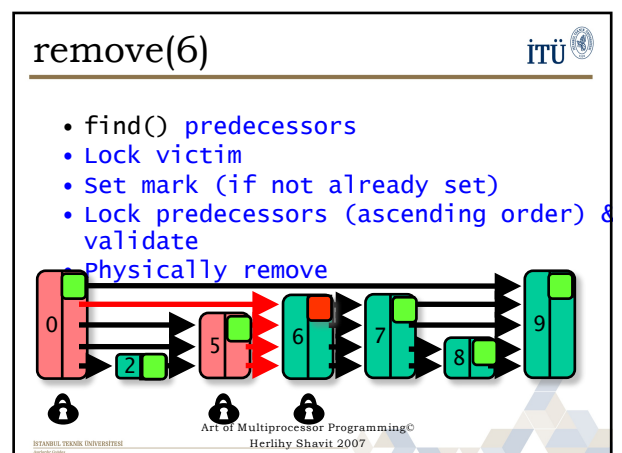
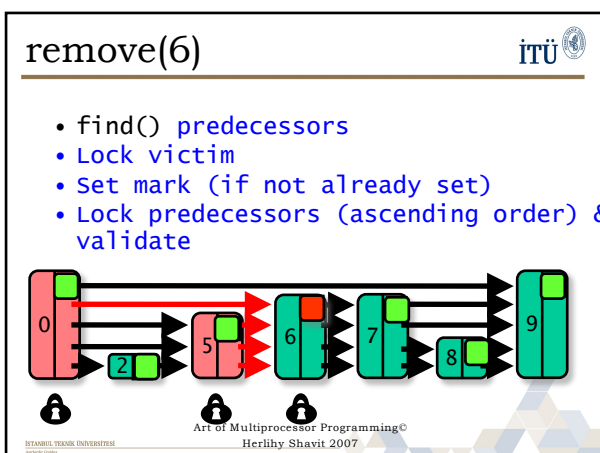
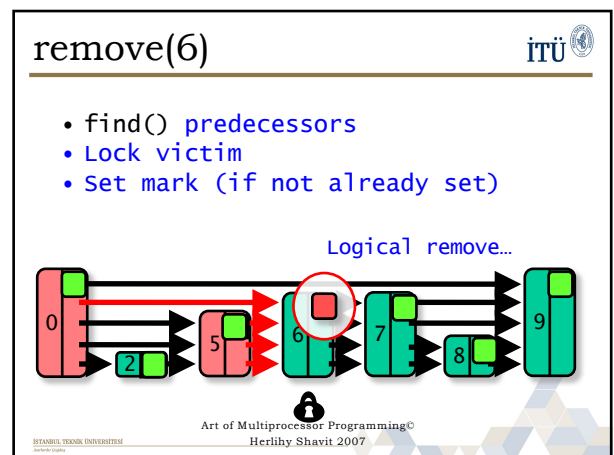
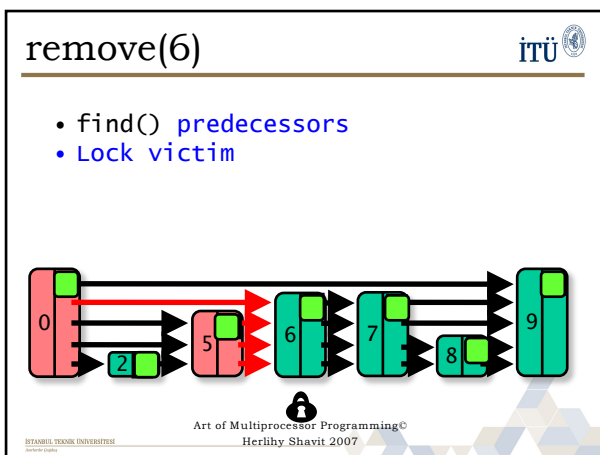
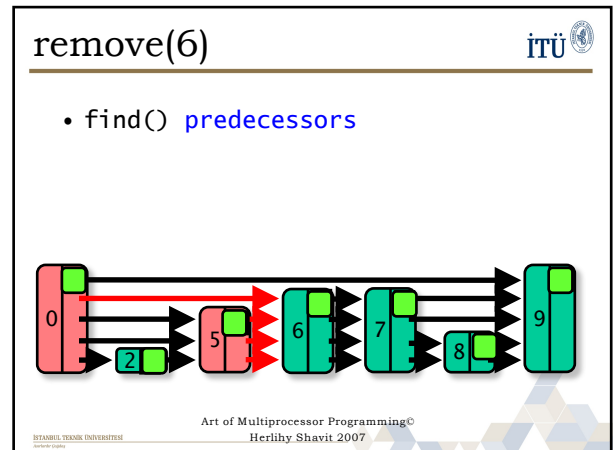
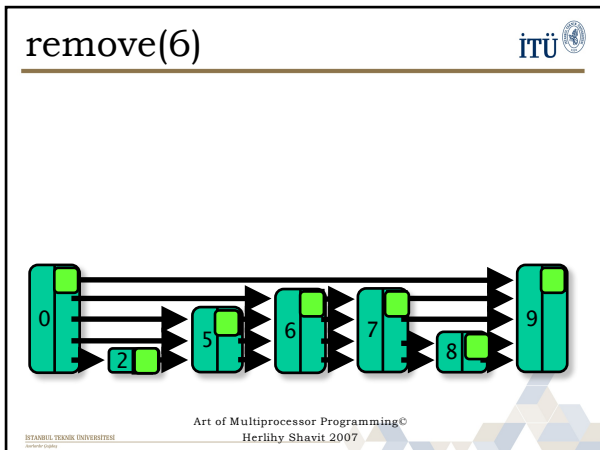
- find() predecessors
- Lock them
- validate
- splice

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

### add(6)

- find() predecessors
- Lock them
- validate
- splice
- unlock

Art of Multiprocessor Programming©  
Herlihy Shavit 2007

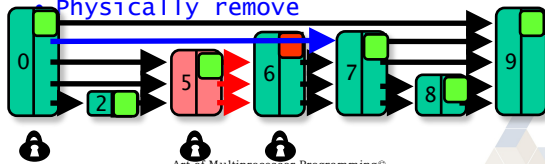




## remove(6)



- find() predecessors
- Lock victim
- Set mark (if not already set)
- Lock predecessors (ascending order) & validate
- Physically remove

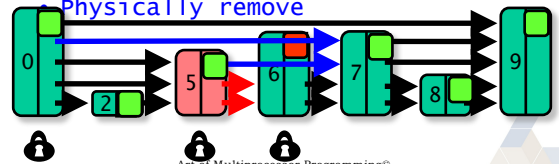


Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## remove(6)



- find() predecessors
- Lock victim
- Set mark (if not already set)
- Lock predecessors (ascending order) & validate
- Physically remove

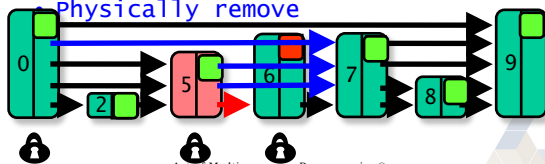


Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## remove(6)



- find() predecessors
- Lock victim
- Set mark (if not already set)
- Lock predecessors (ascending order) & validate
- Physically remove

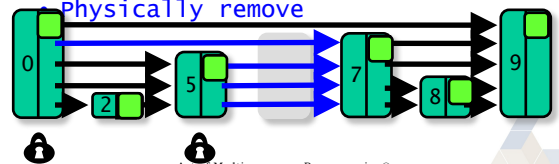


Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## remove(6)



- find() predecessors
- Lock victim
- Set mark (if not already set)
- Lock predecessors (ascending order) & validate
- Physically remove

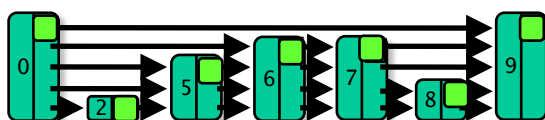


Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## contains(8)



- Find() & not marked

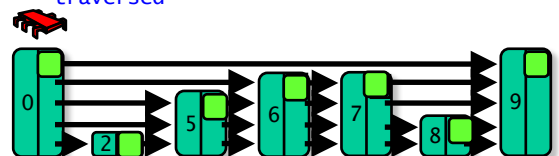


Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## contains(8)



Node 6 removed while traversed

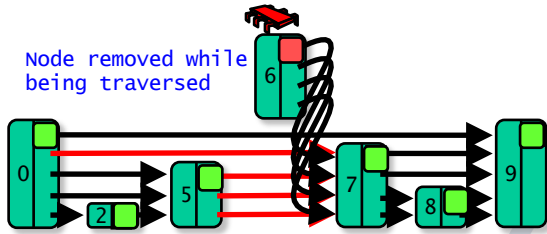


Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## contains(8)



Node removed while  
being traversed



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

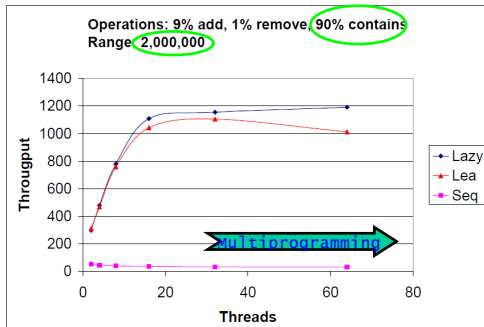
## A Simple Experiment



- Each thread runs 1 million iterations, each either:
  - add()
  - remove()
  - contains()
- Item and method chosen in random from some distribution

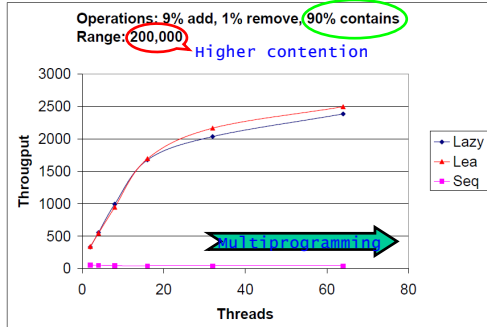
Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Lazy Skip List: Performance



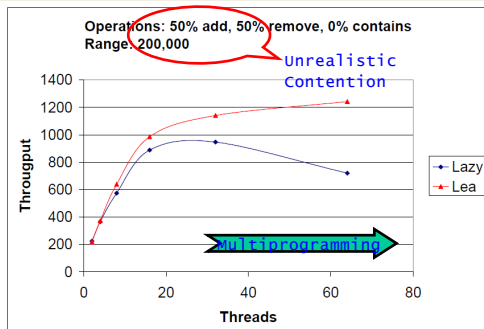
Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Lazy Skip List: Performance



Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Lazy Skip List: Performance





Art of Multiprocessor Programming©  
Herlihy Shavit 2007

## Summary



- Lazy Skip List
  - Optimistic fine-grained Locking
- Performs as well as the lock-free solution in “common” cases
- Simple

Art of Multiprocessor Programming©  
Herlihy Shavit 2007



This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](#).

- **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to "The Art of Multiprocessor Programming" (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

ISTANBUL TECHNICAL UNIVERSITY  
İTÜ

Art of Multiprocessor Programming

205