TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

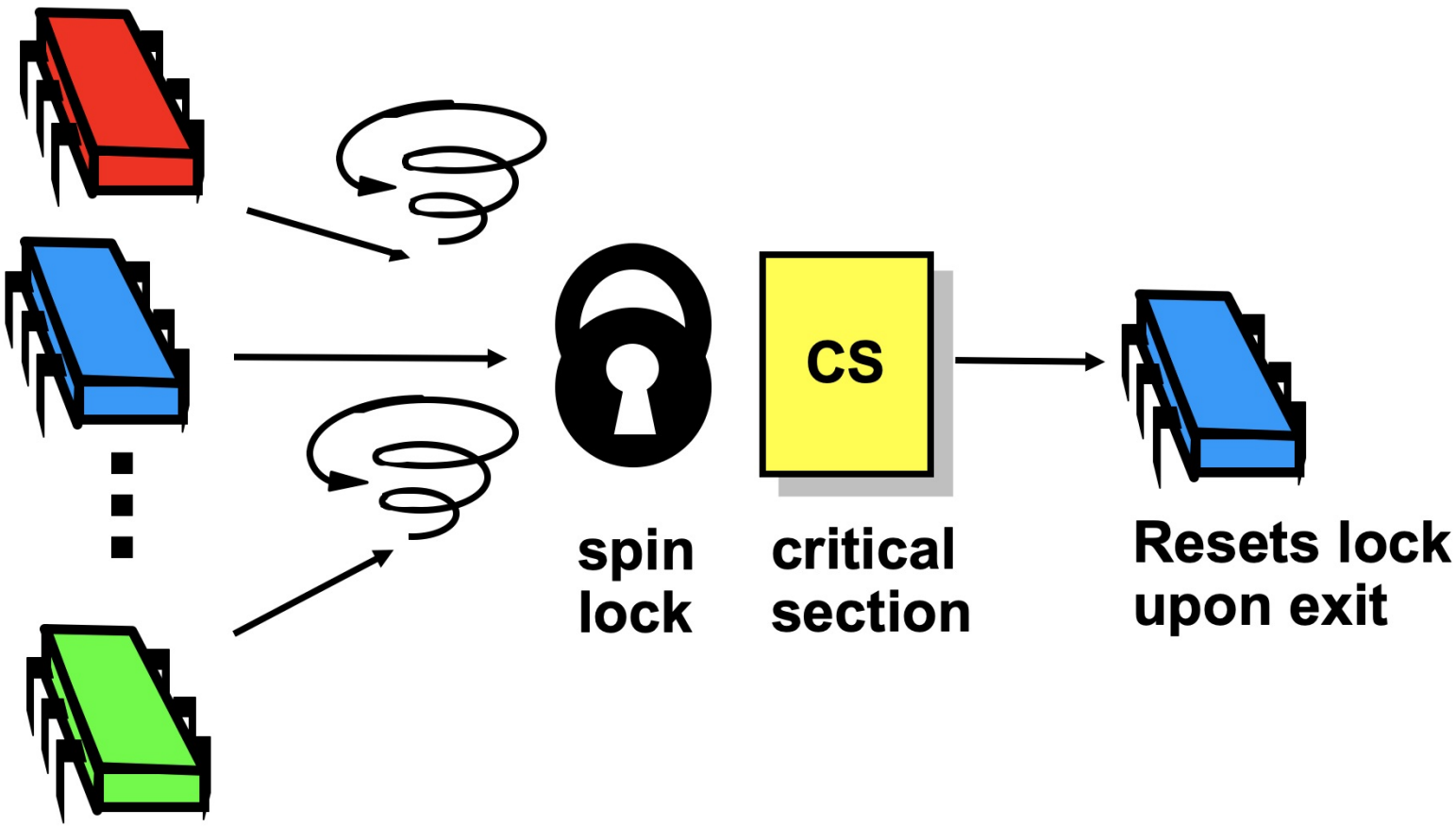
Replication and Consistency

09 Concurrent Linked Lists

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Last lecture: Spin locks



Today: Concurrent Objects

- *Goal 1:* Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by Queue locks
- *Goal 2:* Adding threads should increase throughput
  - Not possible if inherently sequential
  - Surprising things are parallelizable

Today: Concurrent Objects

- *Goal 1:* Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by Queue locks
- *Goal 2:* Adding threads should increase throughput
  - Not possible if inherently sequential
  - Surprising things are parallelizable

- Introduce four "patterns" for highly-concurrent objects
  - Bag of tricks and ideas that work more than once

Pattern 1: Fine-grained synchronization

- Instead of using a single lock, split object into independently-synchronized components
  - E.g. nodes in a list or tree
- Methods conflict when they access
  - the same component
  - at the same time

Pattern 2: Optimistic synchronization

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

Pattern 3: Lazy synchronization

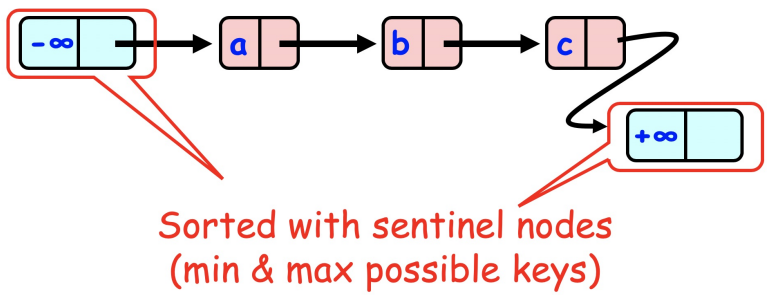
- Postpone hard work
- In particular, removing components can be tricky
- Instead:
  1. Logical removal
     - Mark component to be deleted
  2. Physical removal
     - Do what actually needs to be done

Pattern 4: Lock-free synchronization

- Don't use locks at all
- Use `compareAndSet()` and its relatives
- Advantage:
  - No scheduler assumptions/support required
- Disadvantages:
  - Complex
  - Sometimes high overhead

Set ADT (Abstract Data Type)

- Practical example to illustrate these patterns
- Unordered collection of items
- No duplicates
- List-based implementation with sentinel nodes for head and tail



Methods

- `add(x)`: put `x` in set (returns **`false`** if `x` already in the set)
- `remove(x)`: take `x` out of set (returns **`false`** if `x` not in the set)
- `contains(x)`: tests if `x` in set

Set Interface

```
public interface Set<T> {

  public boolean add(T x);

  public boolean remove(T x);

  public boolean contains(T x);
}
```

Reasoning about Concurrent Objects

- **Invariants**: Properties that "always" holds
- Established because
  - True when object is created
  - Truth preserved by each method
    - i.e. At each step of each method
- Most steps are trivial
  - Usually one step is tricky
  - Often linearization point

Interference

- Invariants make sense only if methods considered are the only modifiers
- Language encapsulation helps
  - *Here:* List nodes not visible outside `List` class
- Freedom from interference needed even for removed nodes
  - Some algorithms traverse removed nodes
  - Careful with `malloc()` & `free()`!
  - Some of the solutions here do **not** work with automatic garbage-collection

Representation Invariants

- **Abstraction map** gives meaning to a concrete representation
  - Here: Item is in the list if corresponding node is reachable from `head`
- **Representation invariants** haracterize legal concrete representations of abstract data type
  - Preserved by methods
  - Relied on by methods
- Contract between methods

Suppose

- `add()` leaves behind two copies of `x`
- `remove()` removes only 1

Which one is incorrect?

Representation Invariants

- **Abstraction map** gives meaning to a concrete representation
  - Here: Item is in the list if corresponding node is reachable from `head`
- **Representation invariants** haracterize legal concrete representations of abstract data type
  - Preserved by methods
  - Relied on by methods
- Contract between methods

Suppose

- `add()` leaves behind two copies of `x`
- `remove()` removes only 1

Which one is incorrect?

- If invariant says no duplicates, `add()` is incorrect
- Otherwise, `remove()` is incorrect

What are Representation Invariants for our linked-list Set?

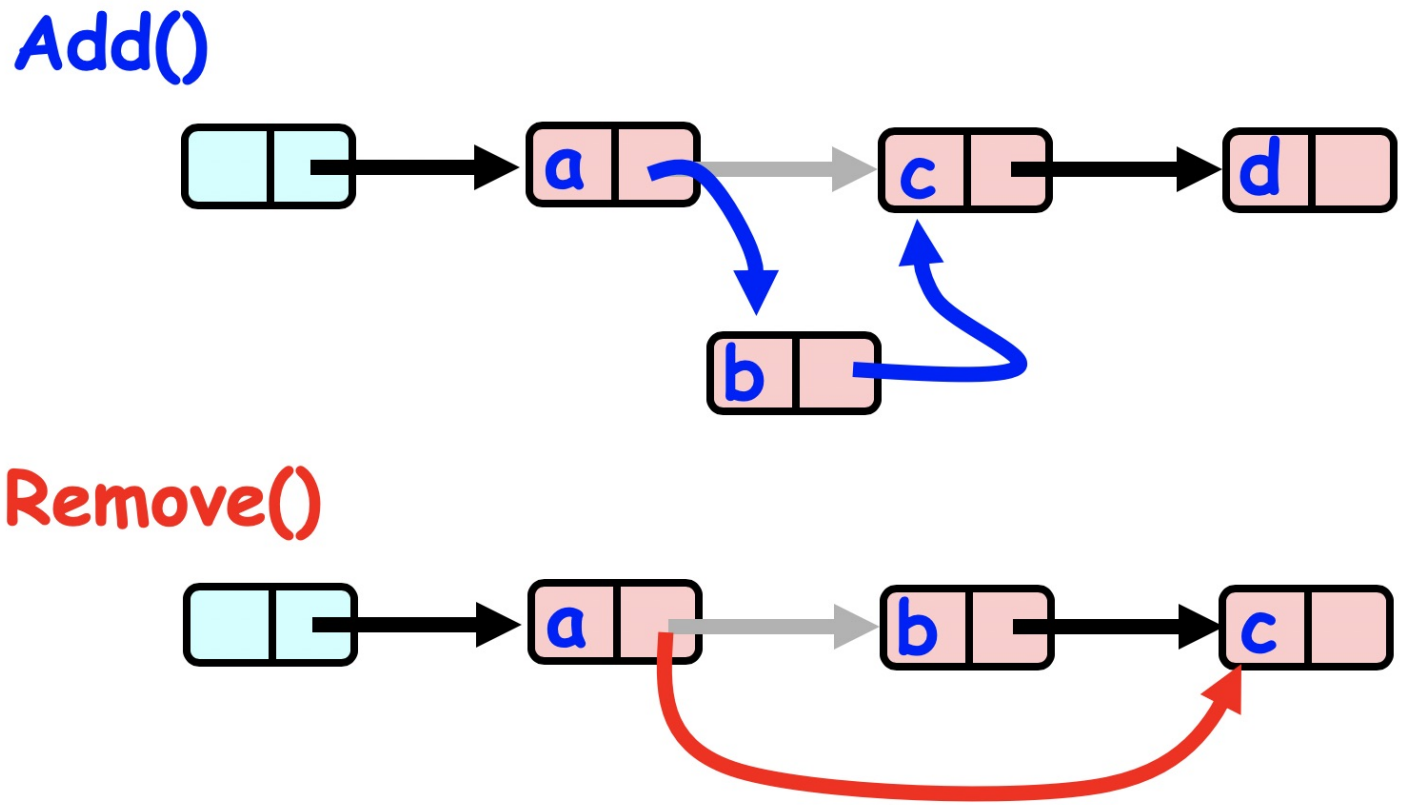
What are Representation Invariants for our linked-list Set?

1. Sorted
2. No duplicates
3. Sentinel nodes $\rightarrow$ `tail` reachable from `head`

Safety and Liveness

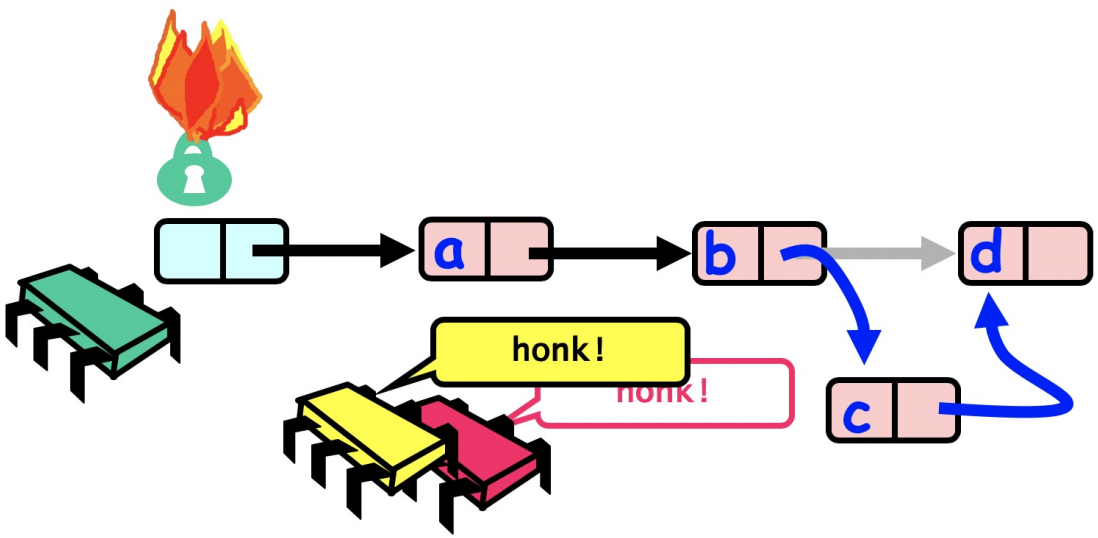
- Safety property: Linearizability
  - Linearization point is atomic step (read, write, CAS, ...) where method "takes effect"
- We will discuss different liveness properties
  - From Deadlock-free
  - To Wait-free

Sequential List-based Set



Coarse-Grained Locking

- Simple and clearly correct
  - [What is the linearization point for each method?]
  - Deserves respect!
- Works poorly under contention
  - Queue locks help
- But bottleneck still an issue



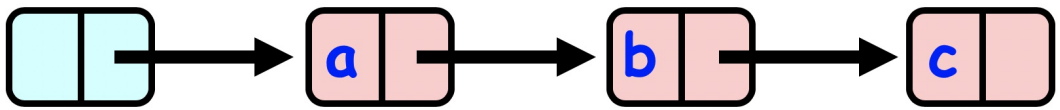
Pattern 1: Fine-Grained Locking

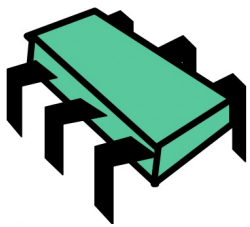
- Requires careful thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger" (J.R.R. Tolkien)

Idea:

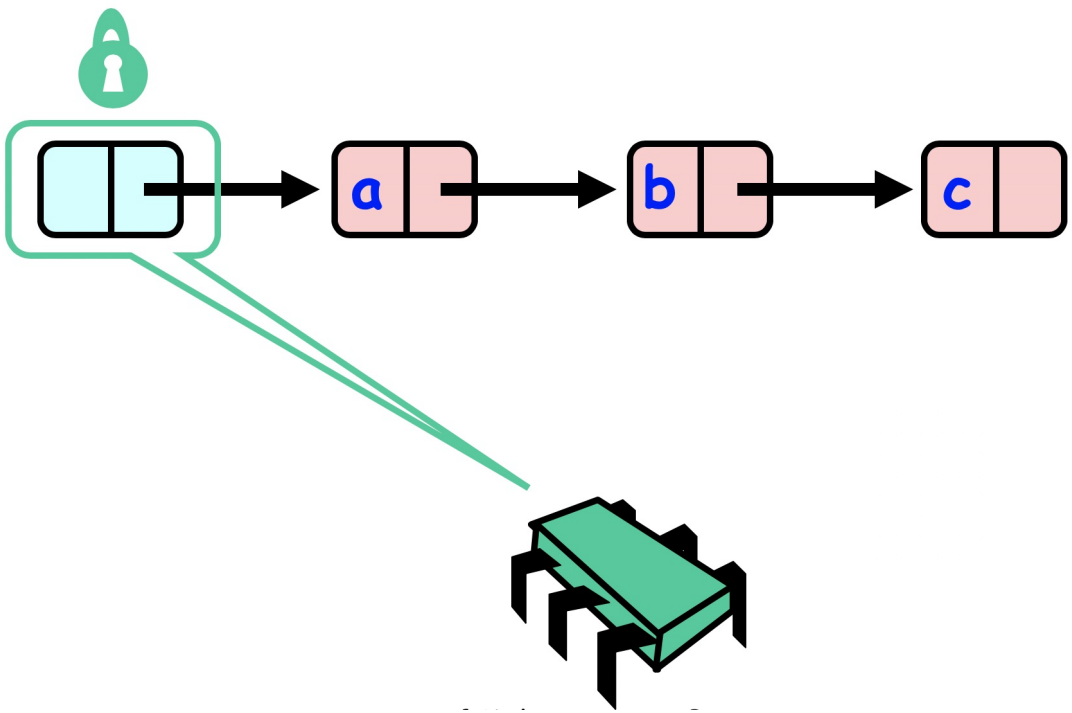
- Split object into pieces
- Each piece has own lock
- Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand Locking

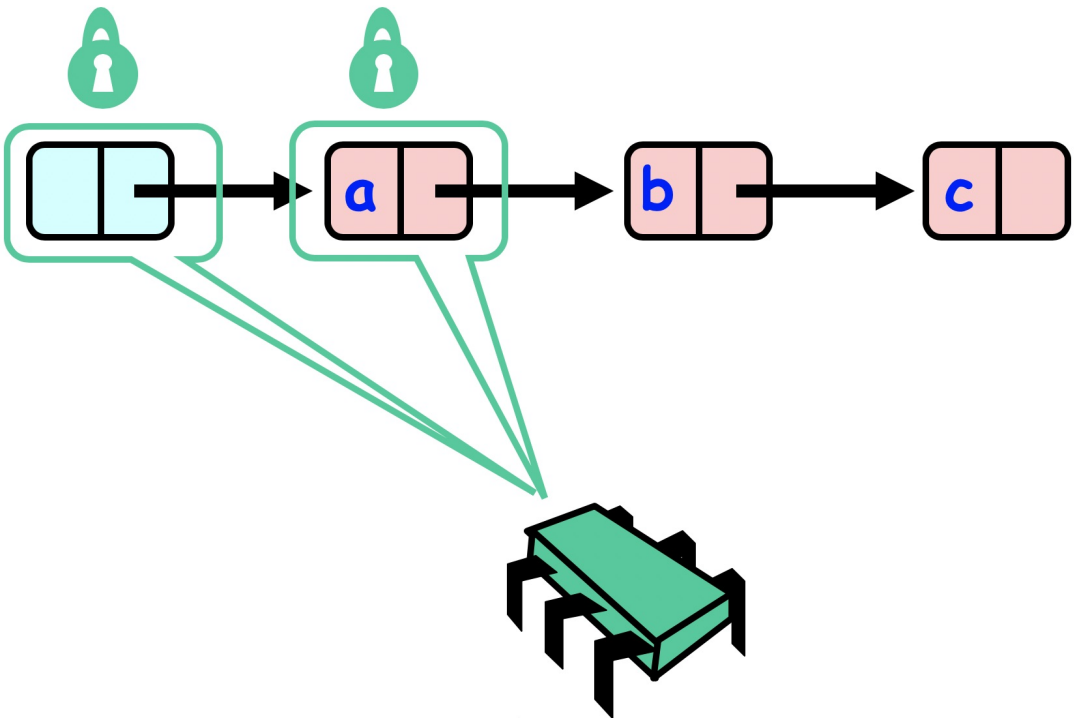




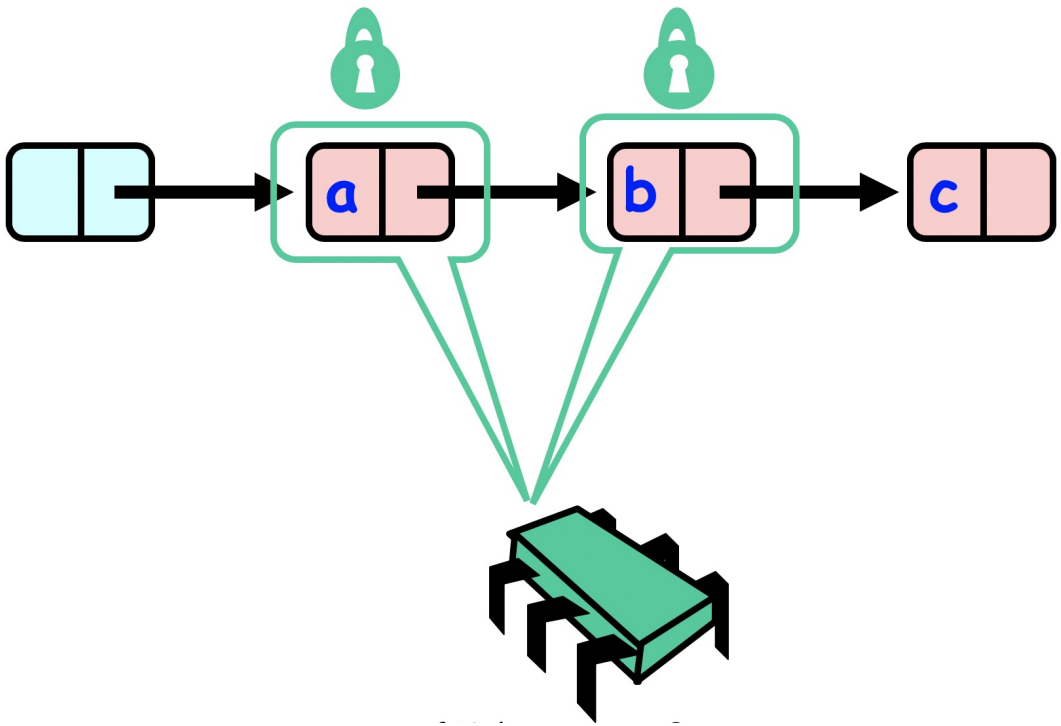
Hand-over-Hand Locking



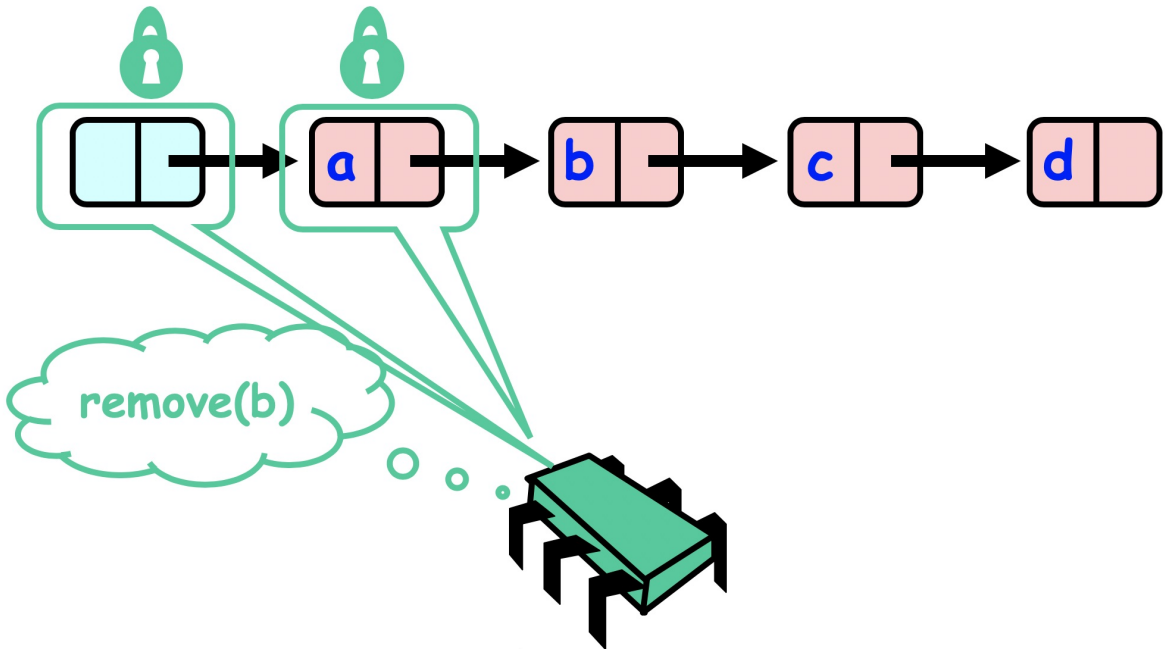
Hand-over-Hand Locking



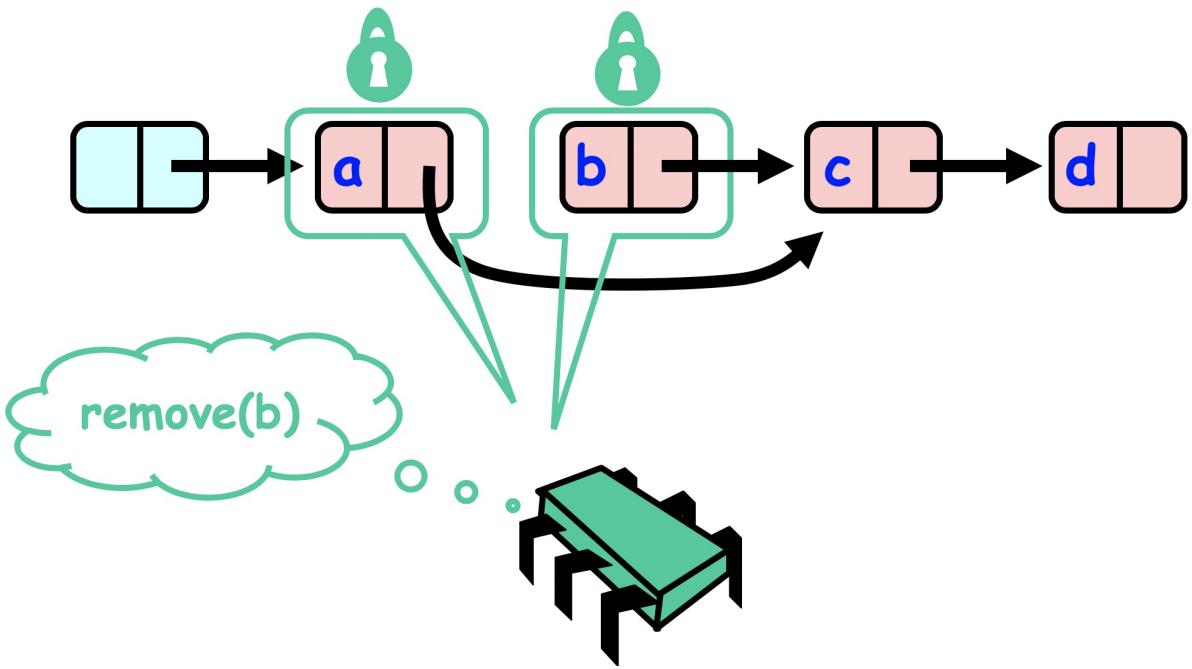
Hand-over-Hand Locking



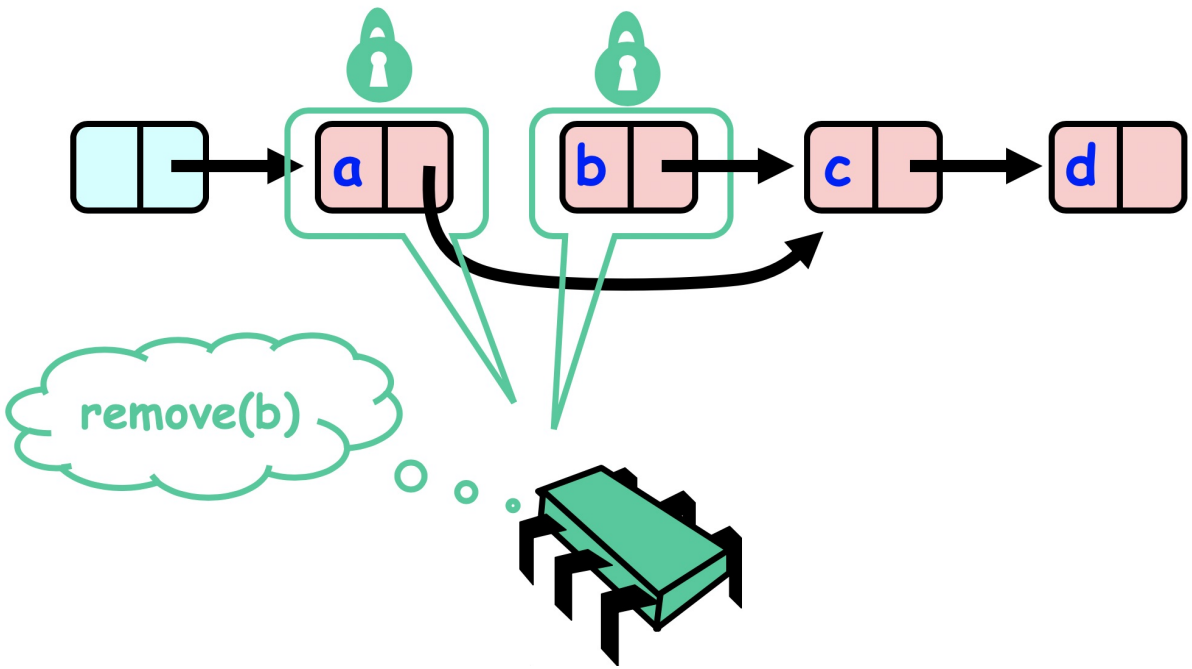
Hand-over-Hand Locking: Removing node



Hand-over-Hand Locking: Removing node

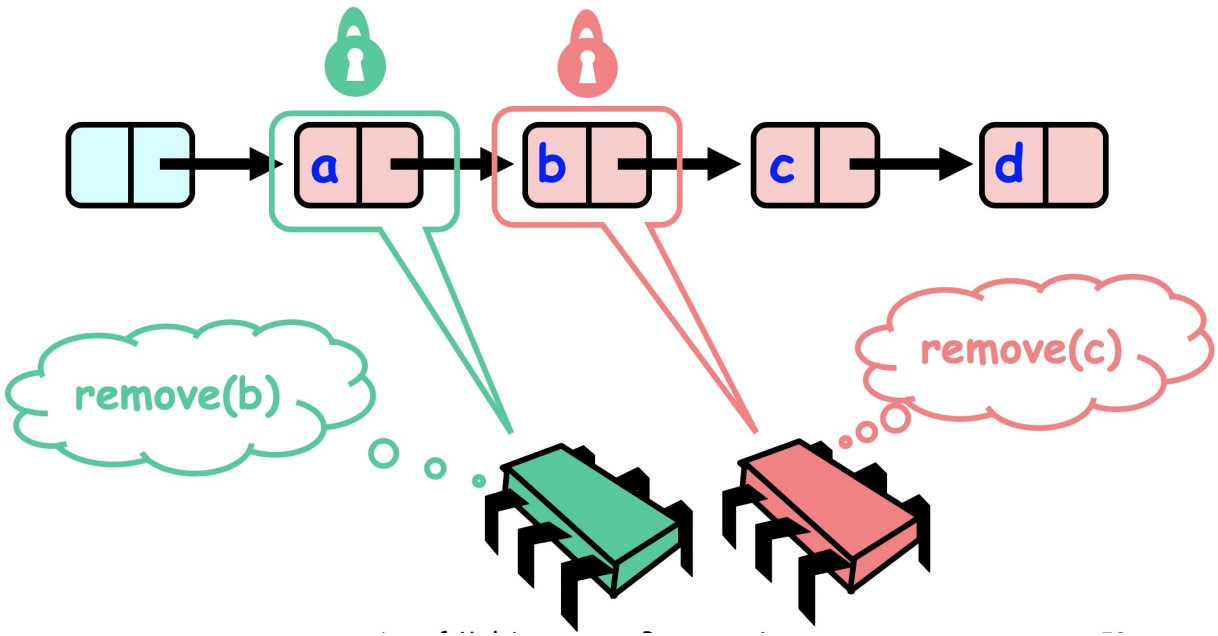


Hand-over-Hand Locking: Removing node

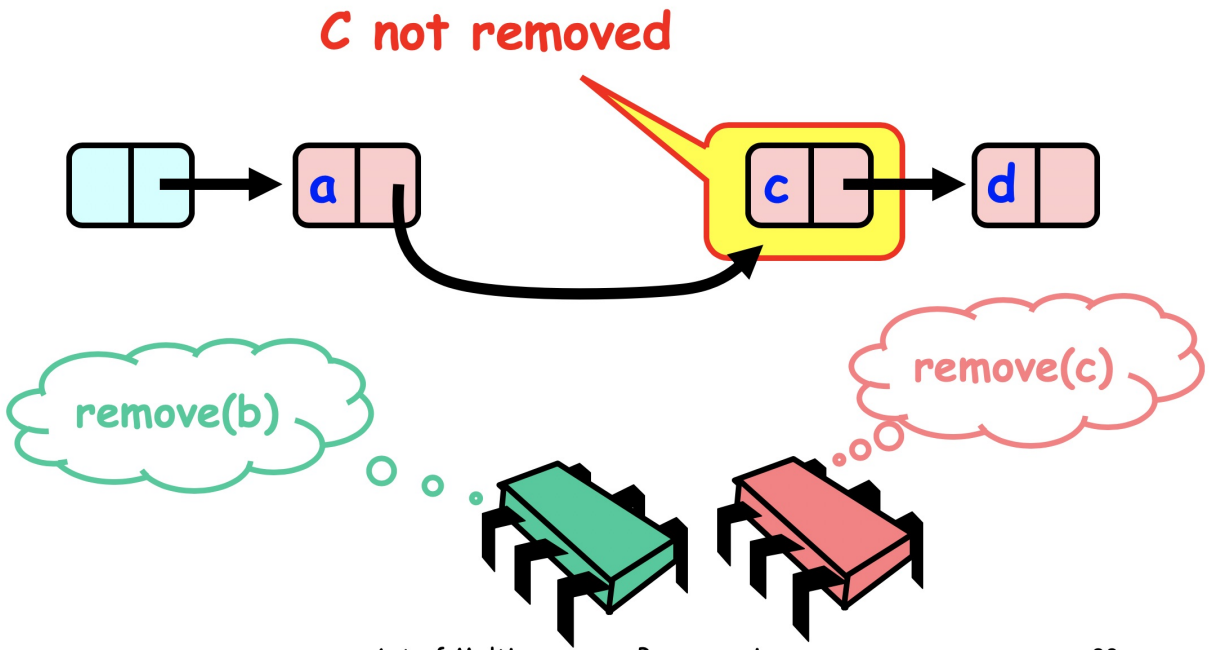


But why do we need to always hold *two* locks?

Hand-over-Hand Locking: Concurrent Remove



Hand-over-Hand Locking: Concurrent Remove



Remarks

- To delete node `c`, swing node `b`'s `next` field to `d`
- *Problem:* Someone deleting `b` concurrently could direct a pointer to `c`
- 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

Implementation: Node

```
class Node {
  T item;
  int key;    // used for ordering, typically hash
  Node next;
}
```

Implementation: Remove

```
public boolean remove(T item) {
  int key = item.hashCode();
  Node pred, curr;
  try {
    ... // next slide
  } finally { // never forget to unlock in the end!
    curr.unlock();
    pred.unlock();
  }
}
```

Implementation: Remove

```
try {
  // get and lock predecessor
  // initially: pred = head
  pred = this.head;
  pred.lock();
  // get and lock current
  curr = pred.next;
  curr.lock();
  ...
} finally { ... }
```

Implementation: Remove

```
// traverse list and search for element to remove
// at start of each loop, curr and pred are locked
  while (curr.key <= key) {  // search range
    if (item == curr.item) {
      // item found -> remove node
      pred.next = curr.next;
      return true;
    }
    pred.unlock();
    // only one node locked!
    pred = curr;
    curr = curr.next;
    curr.lock();
    // lock invariant again restored
  }
  // item not found
  return false;
```

Adding an item

- To add a node
  - we must lock predecessor
  - we must lock successor
- Then, neither can be deleted

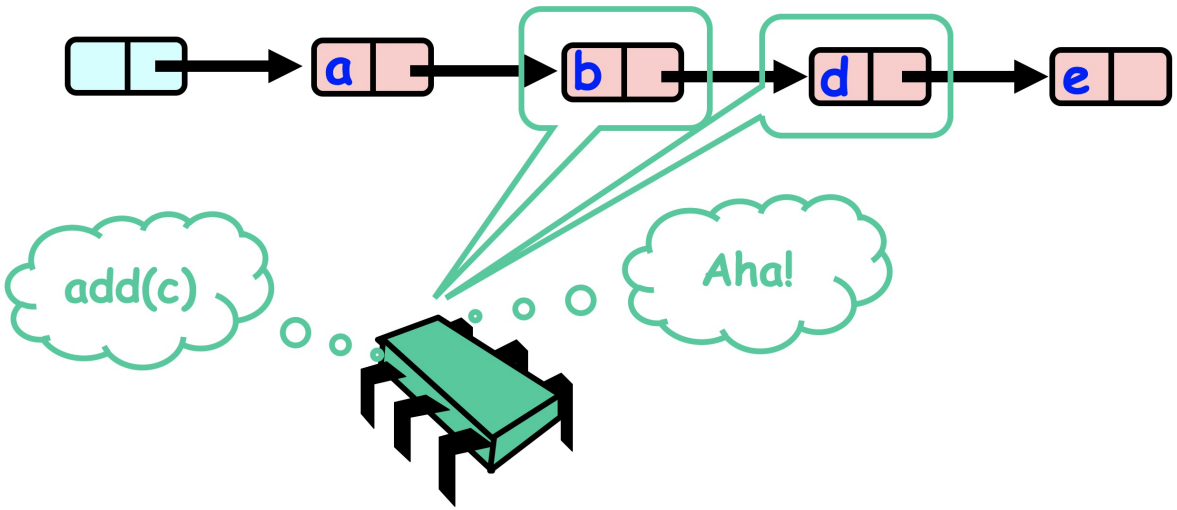
Analysis

- Deadlock-free (why?)
- Starvation-free if locks are starvation-free (why?)
- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
- Inefficient

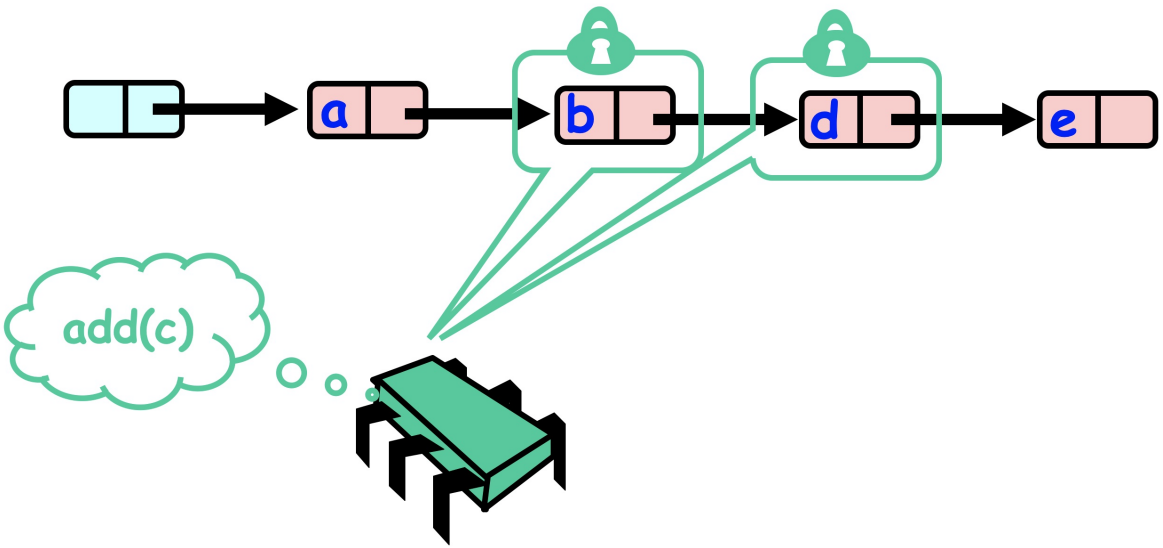
Pattern 2: Optimistic synchronization

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

Optimistic: Traverse without locking

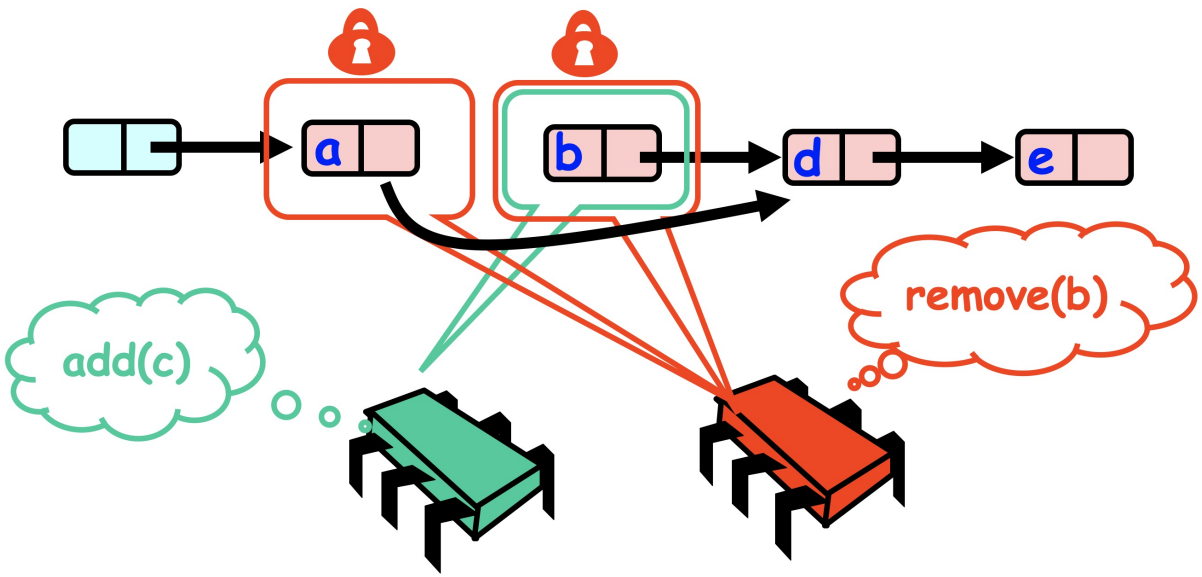


Optimistic: Lock and load

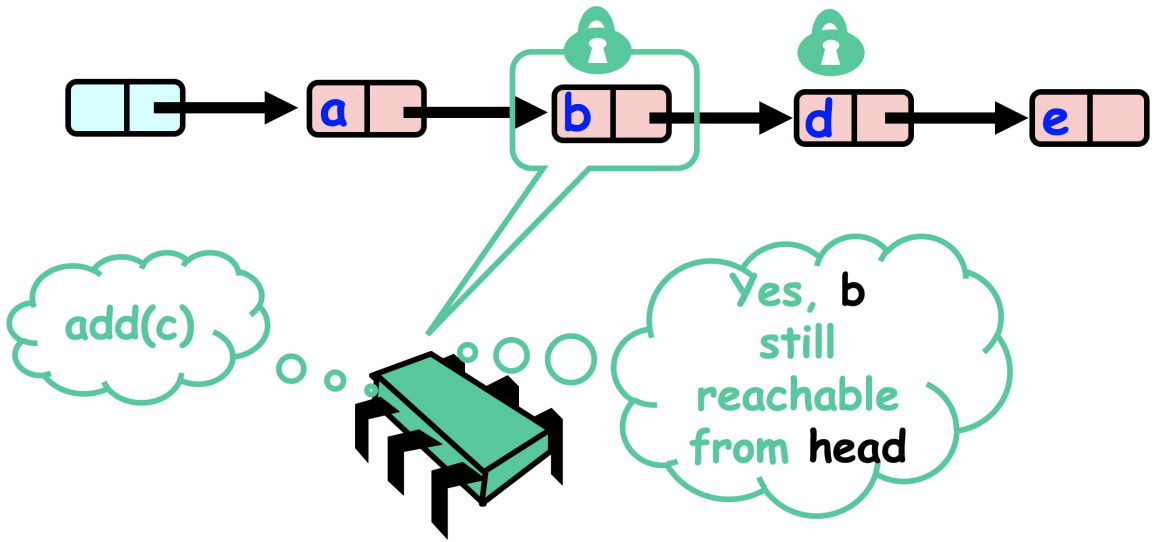


What can go wrong?

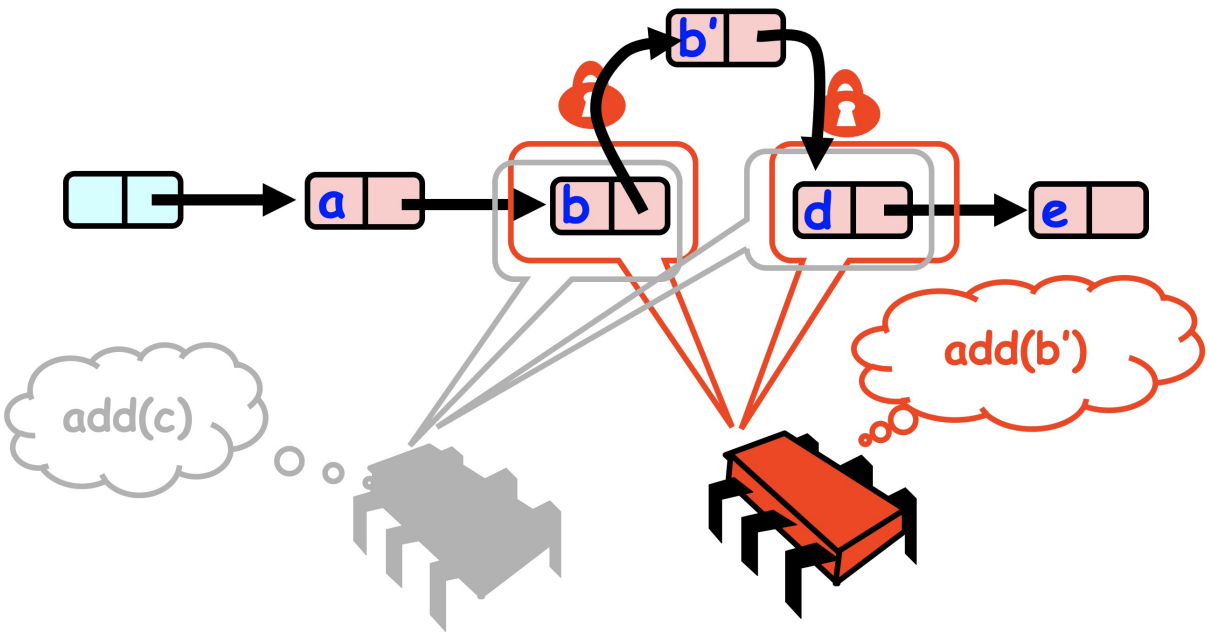
What can go wrong?



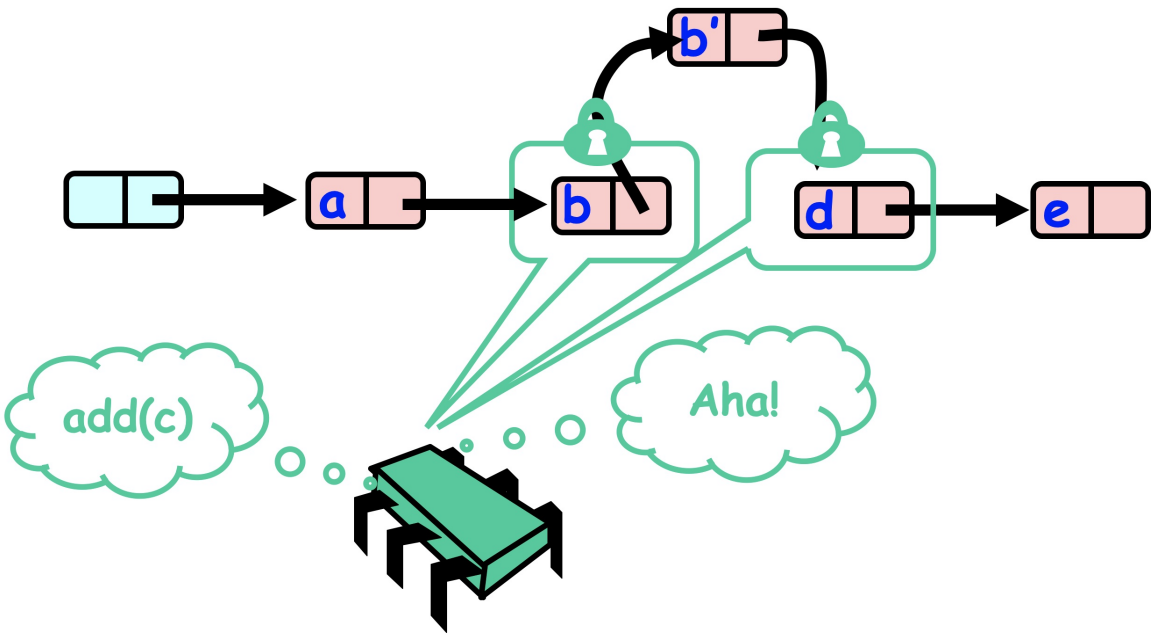
Trick 1: Validate that predecessor is still reachable



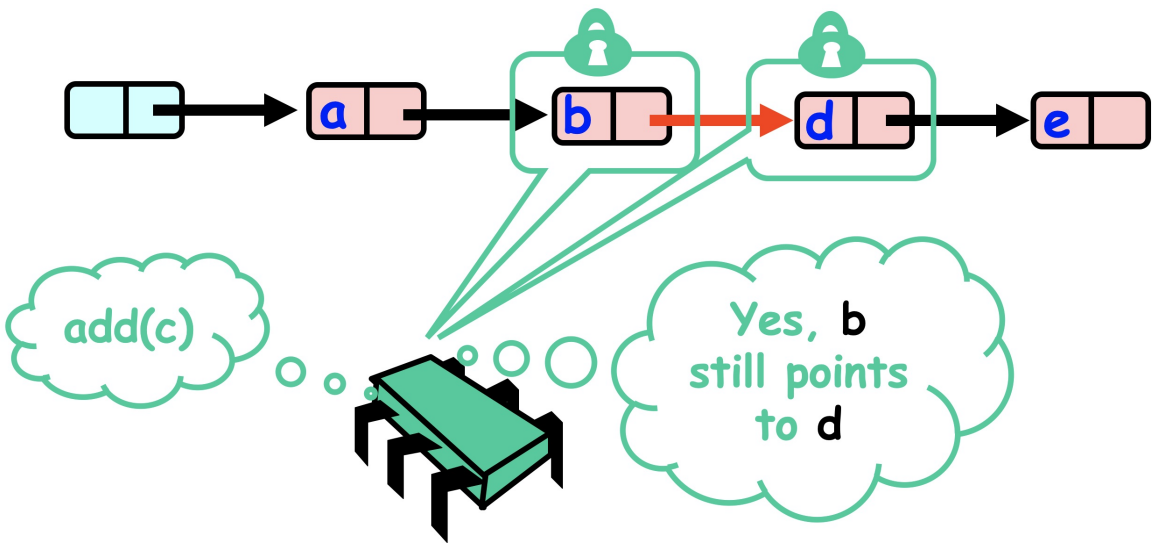
What else can go wrong?



What else can go wrong?



Trick 2: Validate that insertion point is still ok



Is this correct?

Remember our **invariants**!

- Careful: We may traverse deleted nodes
  - `next` is not modified even for unlinked nodes
  - Further traversing leads back to list
  - Be careful with `malloc() / free()` here!
- Establish properties by
  - validation
  - after we lock target nodes

Correctness of Remove (Part 1)

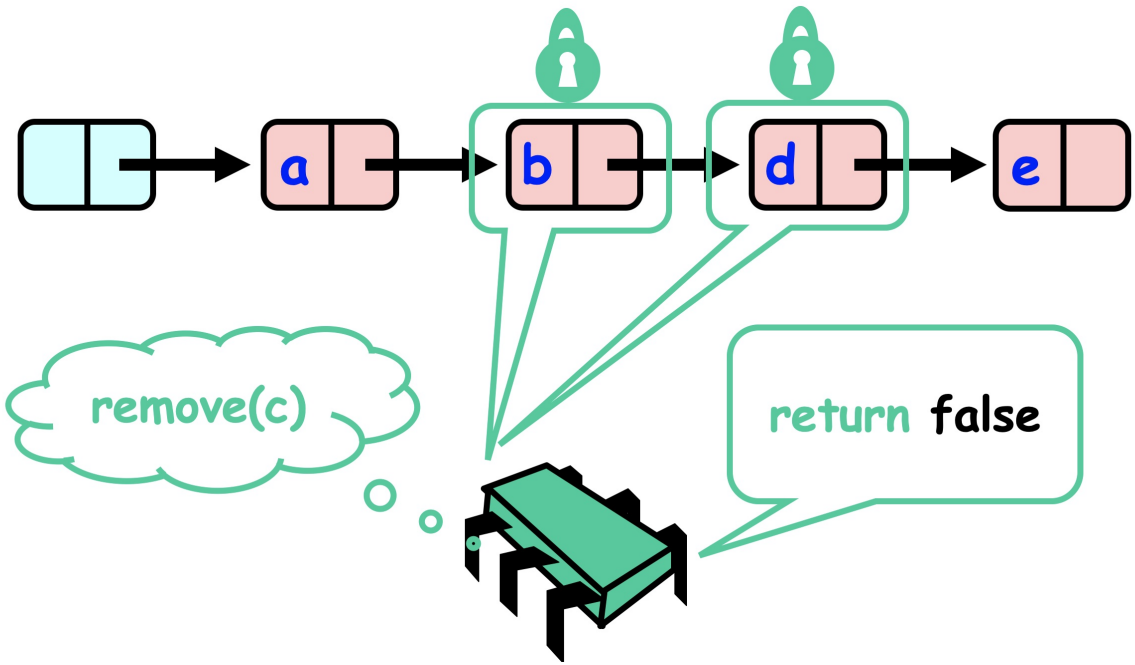
If

- Nodes `b` and `c` both locked
- Node `b` still reachable from `head`
- Node `c` still successor to `b`

Then

- Neither `b` nor `d` will be deleted by other thread
- OK to delete `c` and return **`true`**

Unsuccessful remove



Correctness of Remove (Part 2)

If

- Nodes `b` and `d` both locked
- Node `b` still reachable from `head`
- Node `d` still successor to `b`

Then

- Neither will be deleted
- No thread can add `c` after `b`
- OK to return **`false`**

Validation

```
private boolean validate(Node pred, Node curr) {
  Node node = head;
  // search range
  while (node.key <= pred.key) {
    // predecessor reached from head
    if (node == pred)
      // insertion point still valid?
      return pred.next == curr;
    node = node.next;
  }
  return false;
}
```

Removing - Part 1

```
public boolean remove(Item item) {
  int key = item.hashCode(); // search key
  retry: while (true) {
    // iterate over items
    Node pred = this.head;
    Node curr = pred.next;
    while (curr.key <= key) {
      if (item == curr.item)
        break; // stop if item found
      pred = curr;
      curr = curr.next;
    } ...
```

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]

Removing - Part 2

```
try {
  pred.lock();
  curr.lock();
  if (validate(pred, curr) {
    // successful validation
    if (curr.item == item) {
      // item found
      pred.next = curr.next;
      return true;
    } else {
      // item not found
      return false;
    }
  }
} finally { // always unlock!
    pred.unlock();
    curr.unlock();
}}}
```

Optimistic Lock

- Limited number of hot-spots where threads fiddle with locks
- Targets of `add()`, `remove()`, `contains()`
- No contention on traversals
- Traversals are wait-free, but threads might starve (why?)

Food for thought. . .

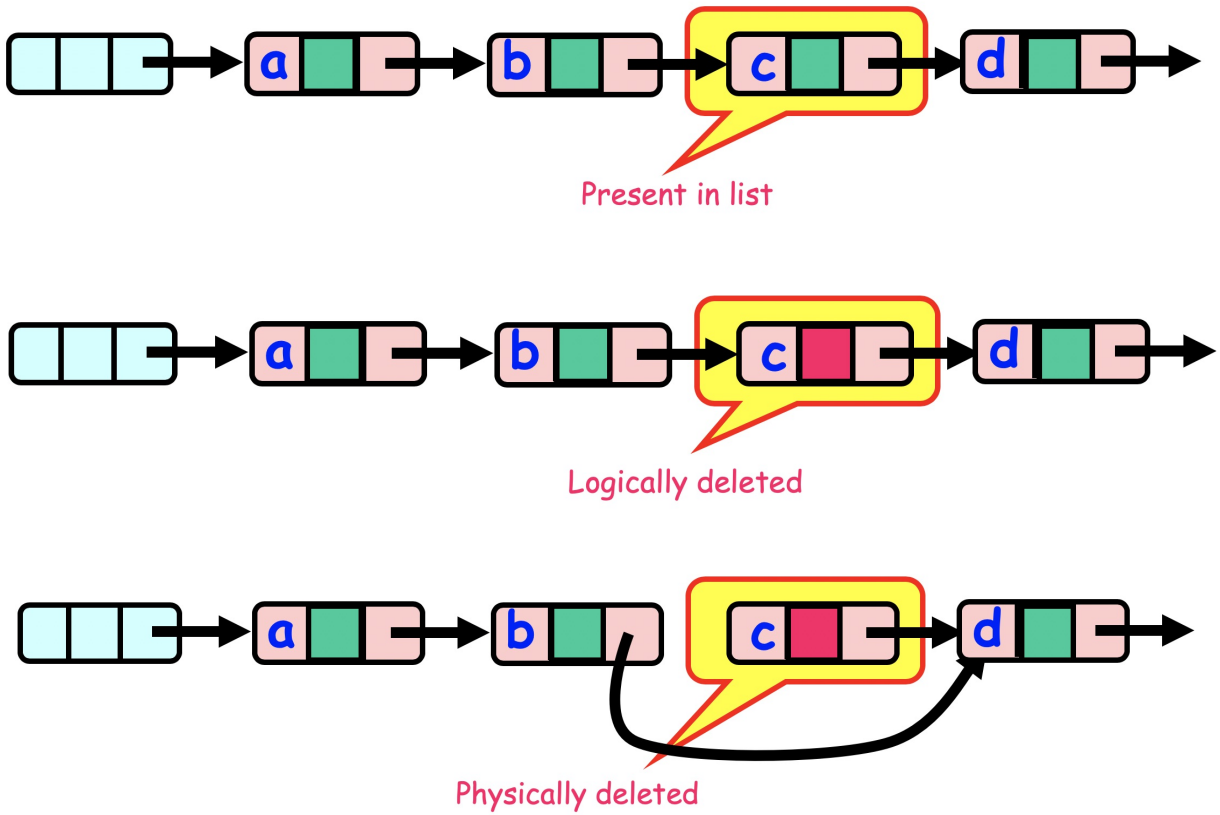
So far, so good

- Much less lock acquisition/release
  - Good for performance
  - Good for concurrency
- Problems
  - Need to traverse list **twice**
  - Optimistic is effective if cost of scanning twice without locks is less than cost of scanning once with locks
  - `contains()` method acquires locks (90% of calls in many apps)

Pattern 4: Lazy Synchronization

- Like optimistic, except
  - scan once for each method
  - `contains(x)` never locks ...
- Key insight
  - Removing nodes causes trouble
  - So, do it "lazily"
- E.g. in method `remove()`:
  - Scans list (as before)
  - Locks predecessor & current (as before)
  - **Logical delete**: Marks current node as removed (new!)
  - **Physical delete**: Redirects predecessor's next (as before)

Lazy List


a
b
c
d
Present in list
a
b
c
d
Logically deleted
a
b
c
d
Physically deleted

Remarks

- Different abstraction map

Remarks

- Different abstraction map
  - An item is in the set iff it is referred to by an *unmarked* reachable node
- Different representation invariant
  - Every unmarked node is reachable
- All methods scan through locked and marked nodes
- Removing a node doesn't slow down other method calls
- Must still lock `pred` and `curr` nodes

Validation

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

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

Remove

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

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;
}
```

Analysis

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

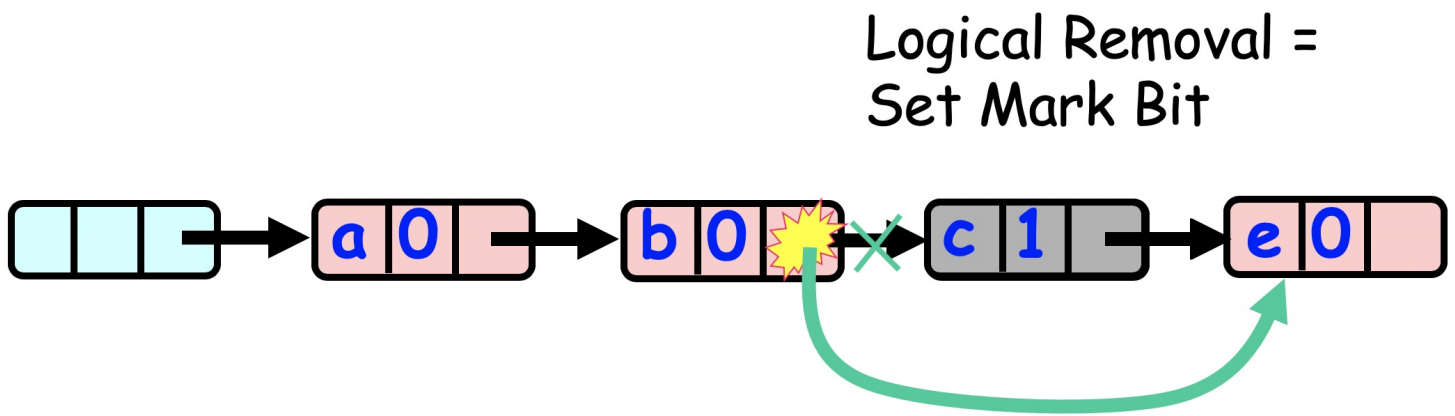
"Traffic jam"

- Any concurrent data structure based on mutual exclusion has a weakness
  - If one thread enters critical section
  - And "eats the big muffin" (e.g. cache miss, page fault, descheduled ...)
  - Everyone else using that lock is stuck!
- Need to trust the scheduler ...
- **Lock-free data structures**
  - Guarantees minimal progress in *any* execution
  - i.e. some thread will always complete a method call
  - Even if others halt at malicious times
  - Implies that implementation can't use locks

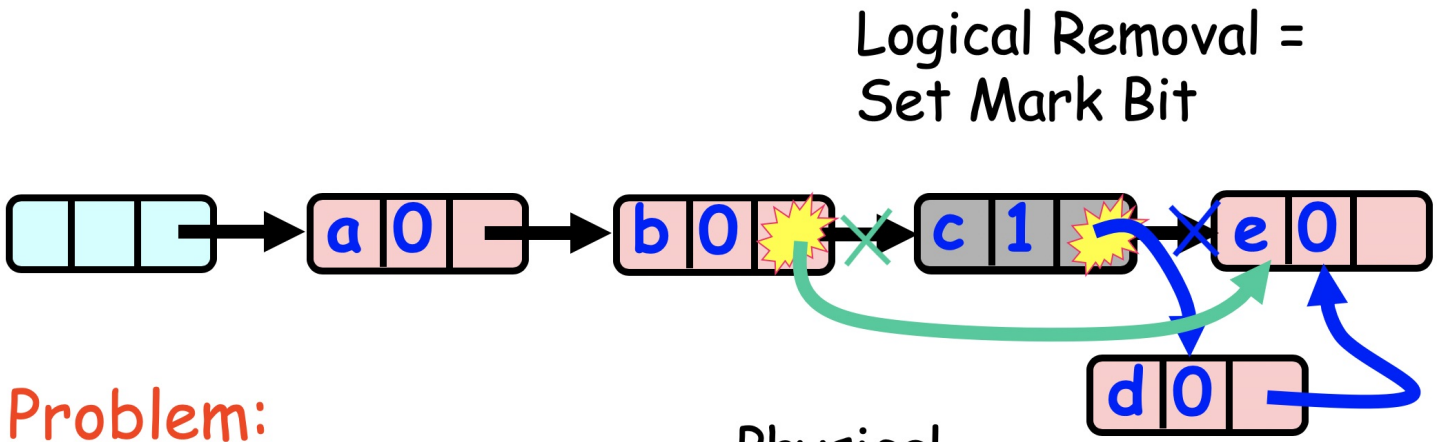
Pattern 4: Lock-free Synchronization

- Eliminate locking entirely
- `contains()` wait-free and `add()` and `remove()` lock-free
- Use only `compareAndSet()`

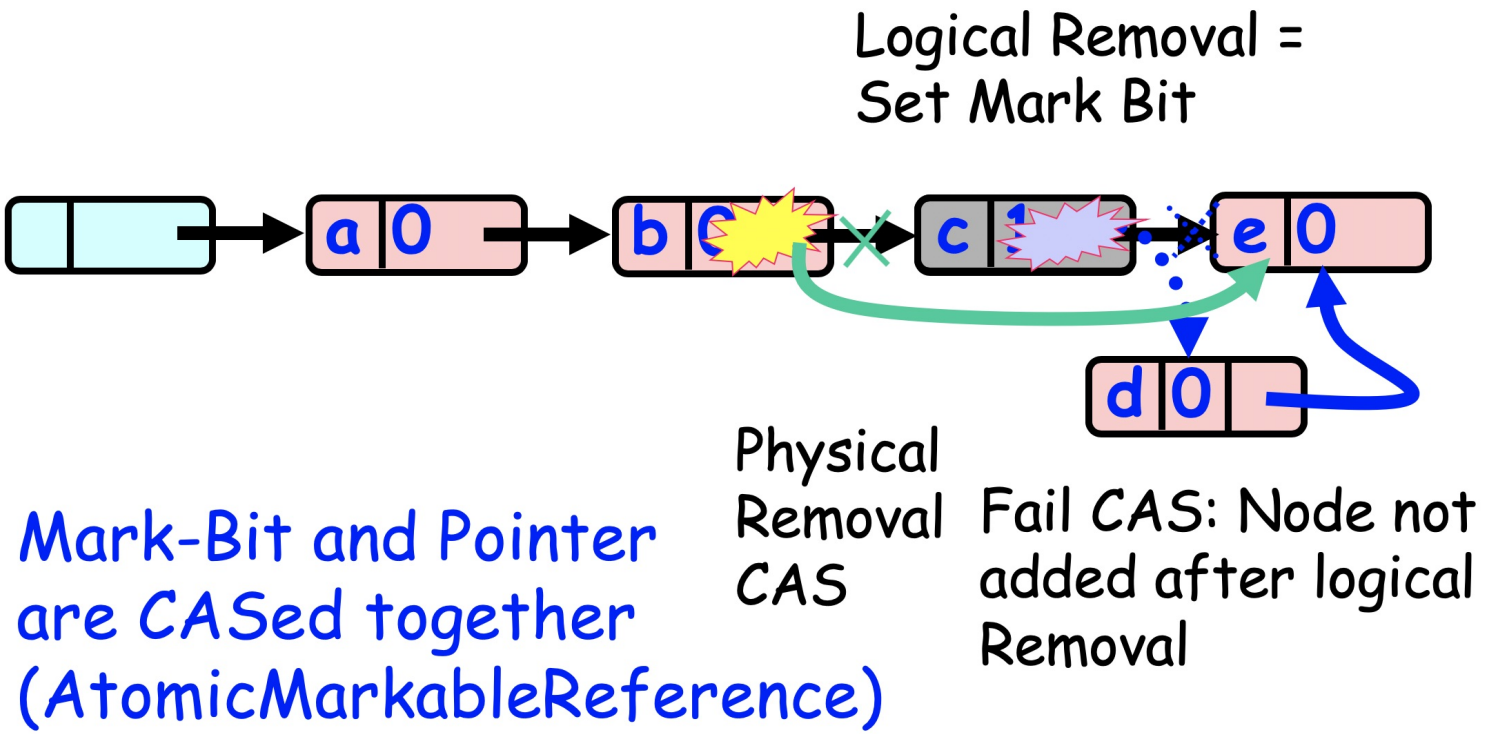
What could go wrong?



Problem


Logical Removal =
Set Mark Bit
a 0
b 0
c 1
e 0
d 0
Problem:
d not added to list
Must Prevent
manipulation of
removed node's pointer
Physical
Removal
CAS
Node added
Before
Physical
Removal CAS

Solution: Combine Bit and Pointer

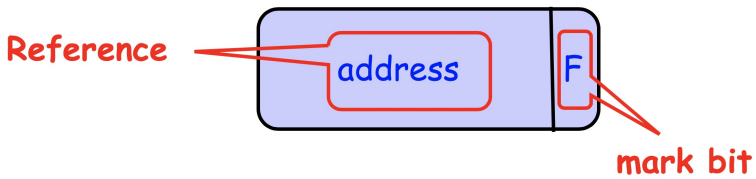


- Ensure that node's fields cannot be updated after node has been logically or physically removed from the list
- Treat `next` and `marked` field as atomic unit

Use AtomicMarkableReference

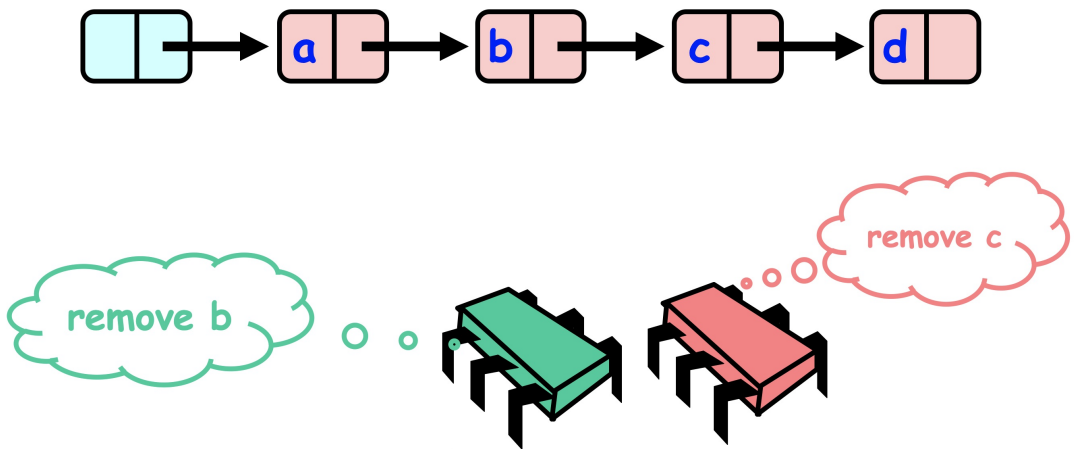
- In C/C++: Embed flag in pointer
- In Java: Class in `java.util.concurrent.atomic` package
- Atomically swing reference and update flag
- Remove in two steps
  - Set mark bit in `next` field
  - Redirect predecessor's pointer

AtomicMarkableReference class

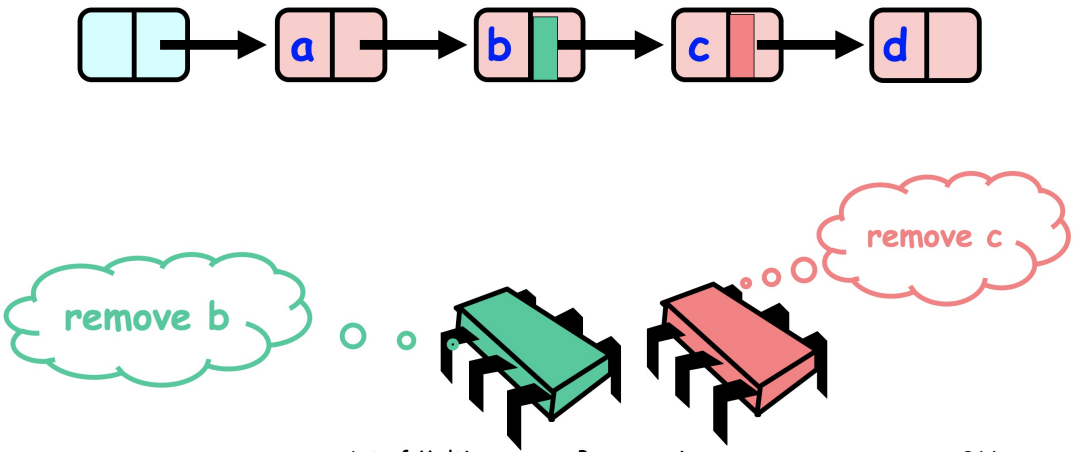



```
// returns reference + mark at array index 0
public Object get(boolean[] marked);
// returns mark
public boolean isMarked();
// double CAS
public boolean compareAndSet(
  Object expectedRef, Object updateRef,
  boolean expectedMark, boolean updateMark);
// update mark if ref as expected
public boolean attemptMark(
  Object expectedRef, boolean updateMark);
```

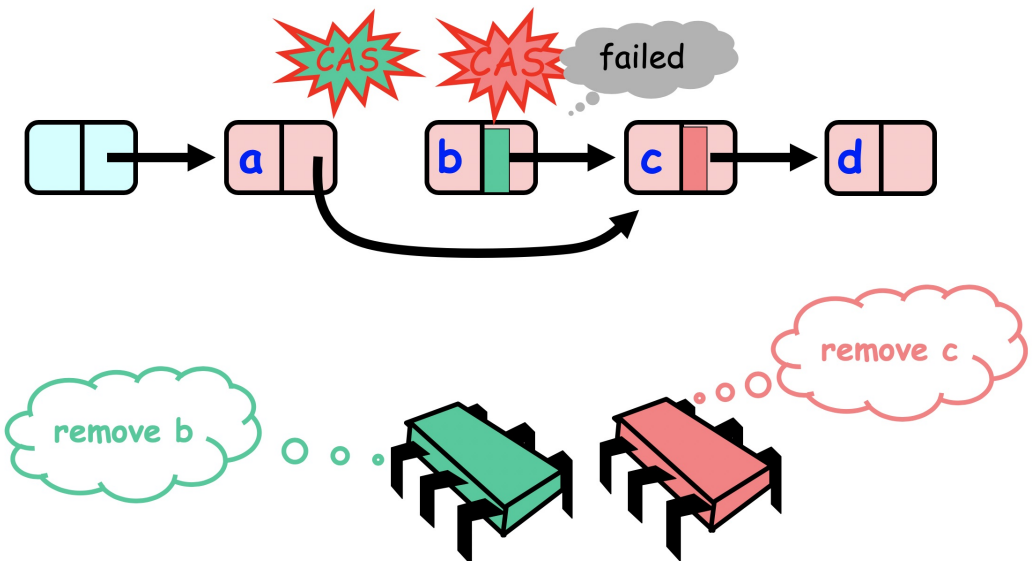
Remove



Remove



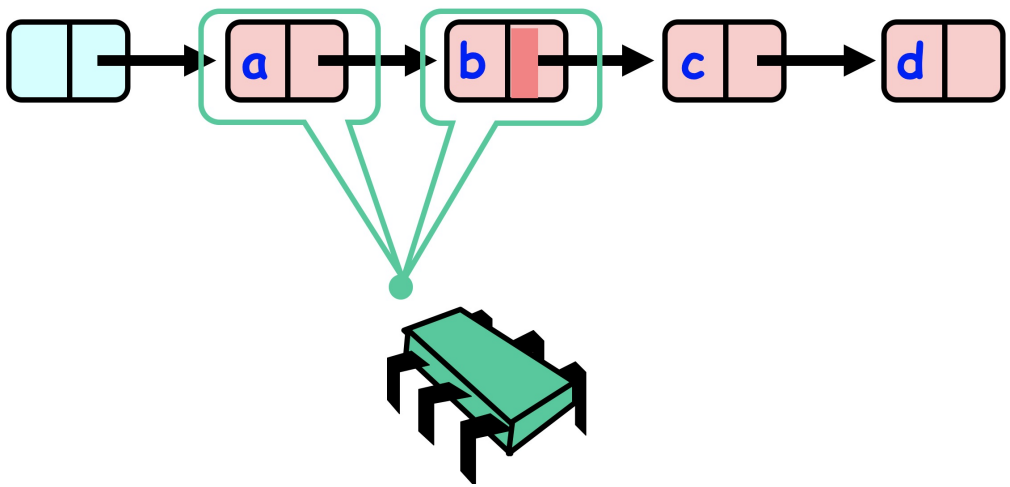
Remove



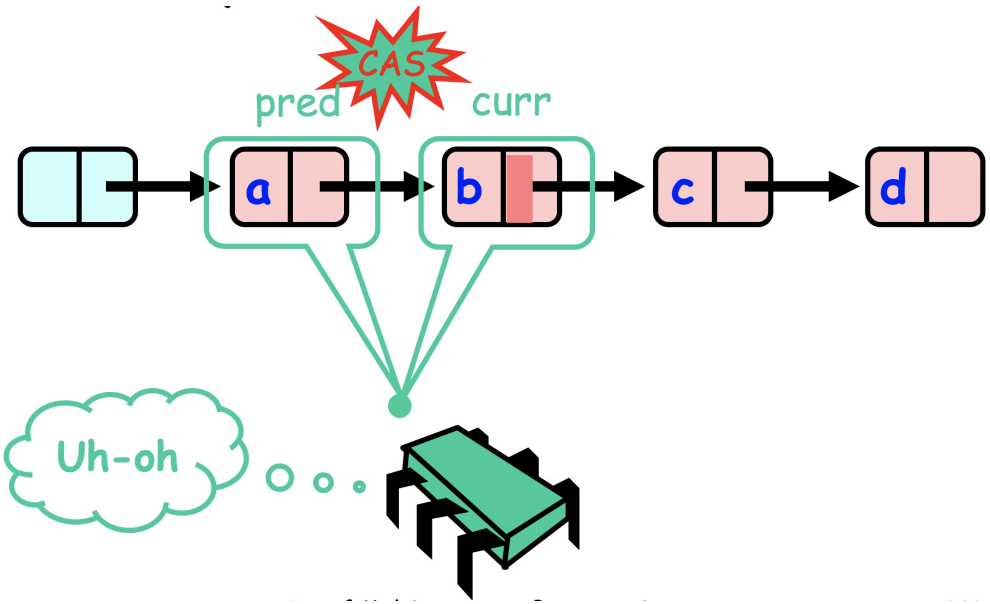
Traversing the list

- What do you do when you find a “logically” deleted node in your path?
- Finish the job!
  - CAS the predecessor’s `next` field
  - Proceed (repeat as needed)
- If threads don’t clean up while traversing, they may be forced to re-traverse the list to remove previous marked nodes

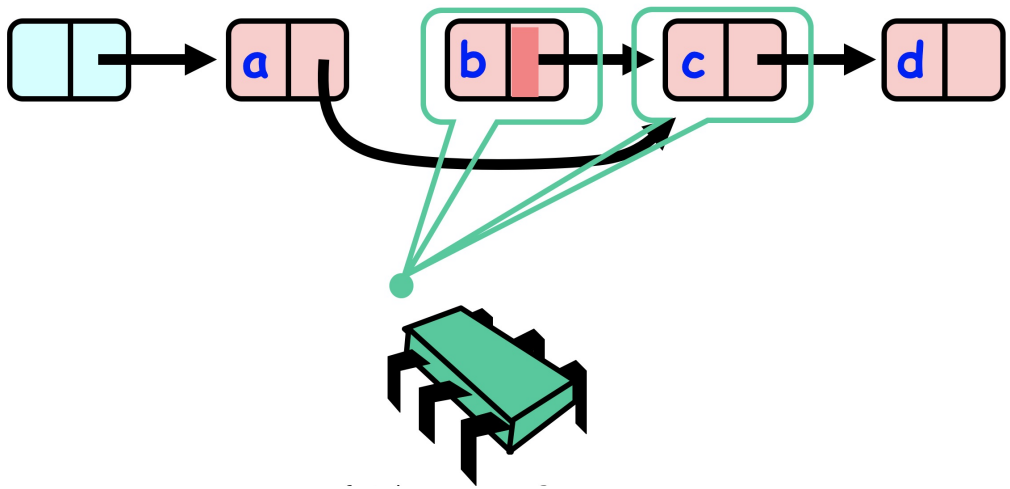
Lock-free Traversal (only `add()` + `remove()`)



Lock-free Traversal (only `add()` + `remove()`)



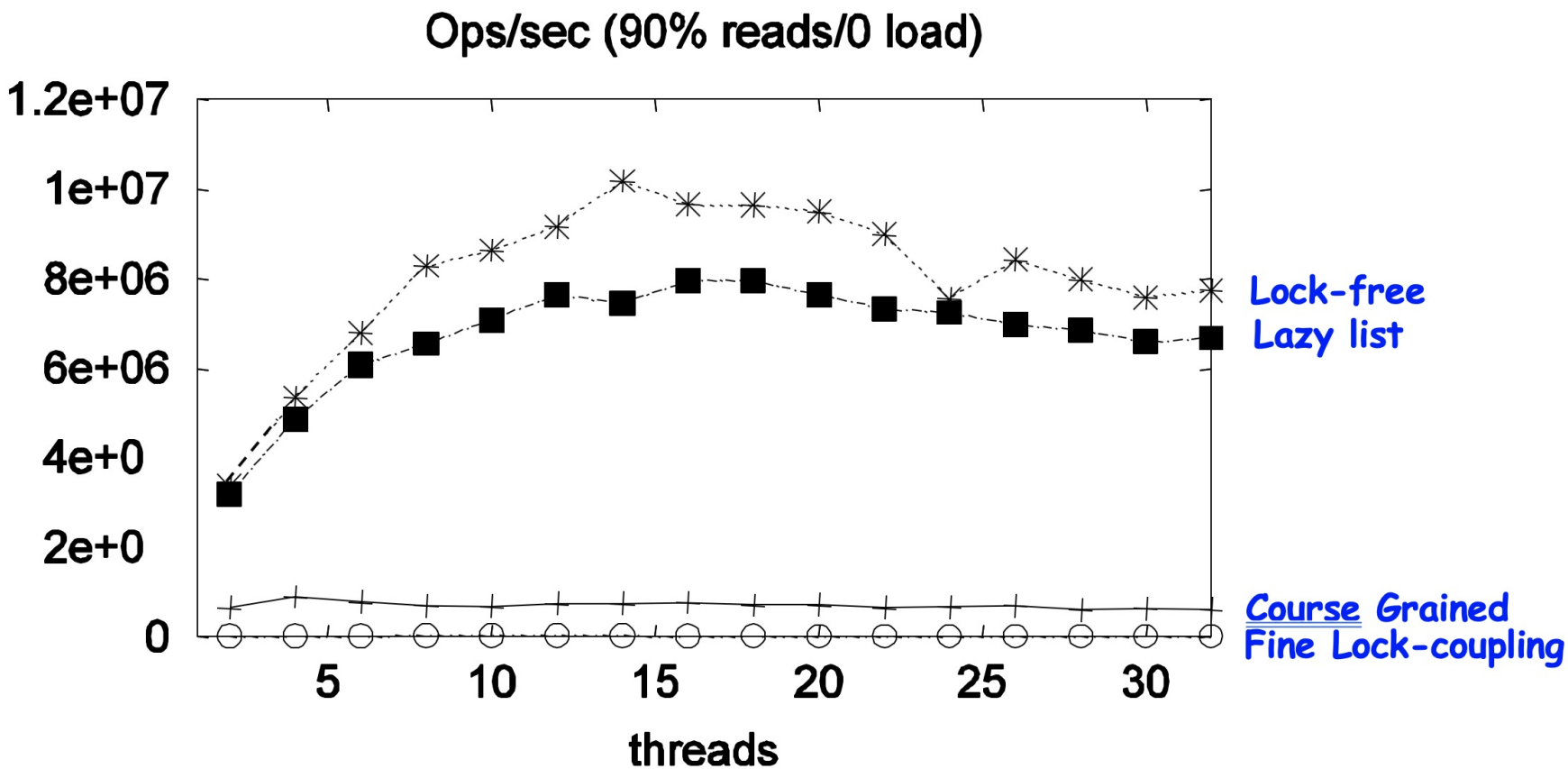
Lock-free Traversal (only `add()` + `remove()`)



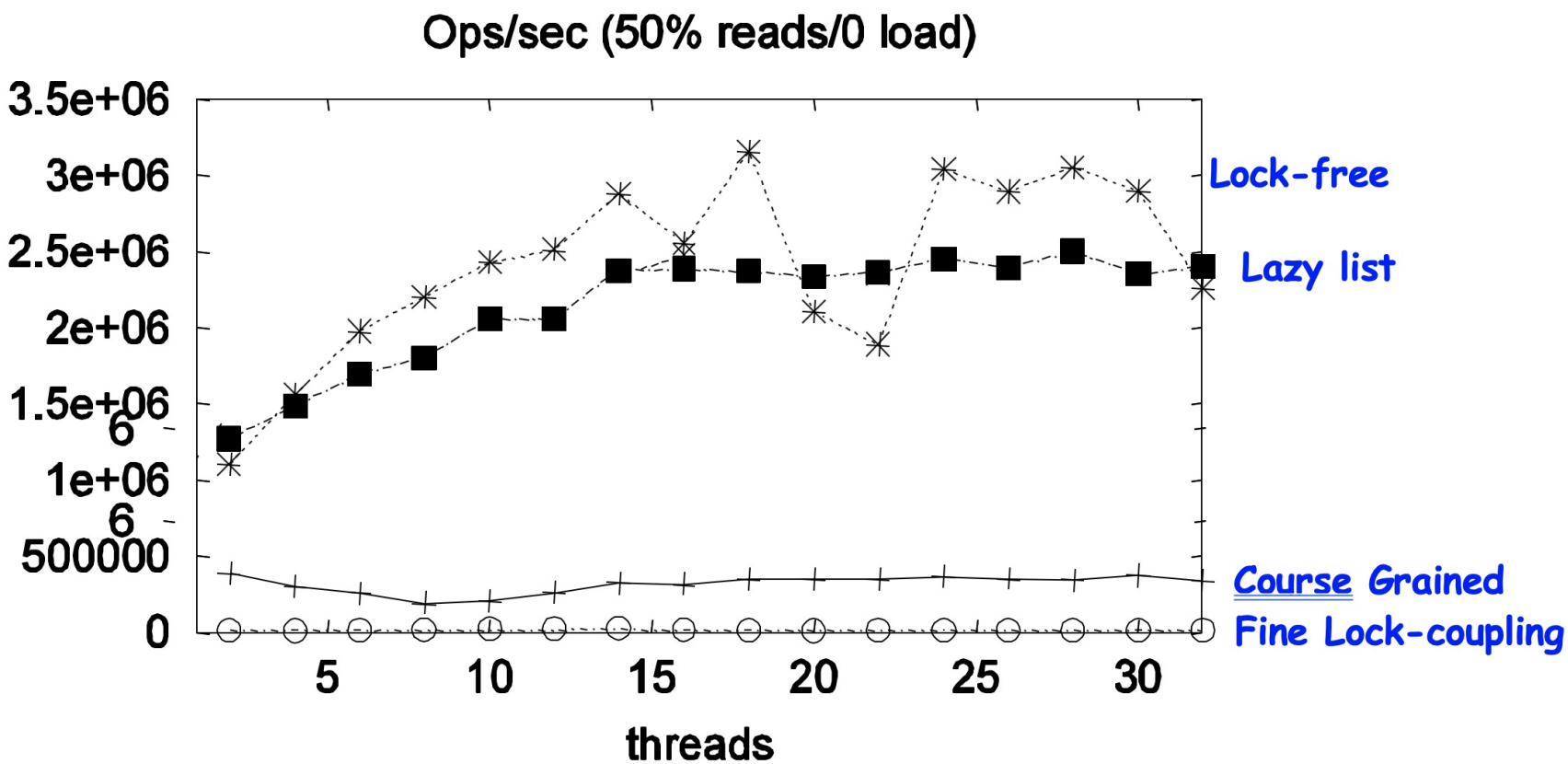
Performance

- Benchmark for throughput of Java List-based Set
- 16-node shared memory machine
- Different algorithms
- Vary percentage of `contains()` method calls

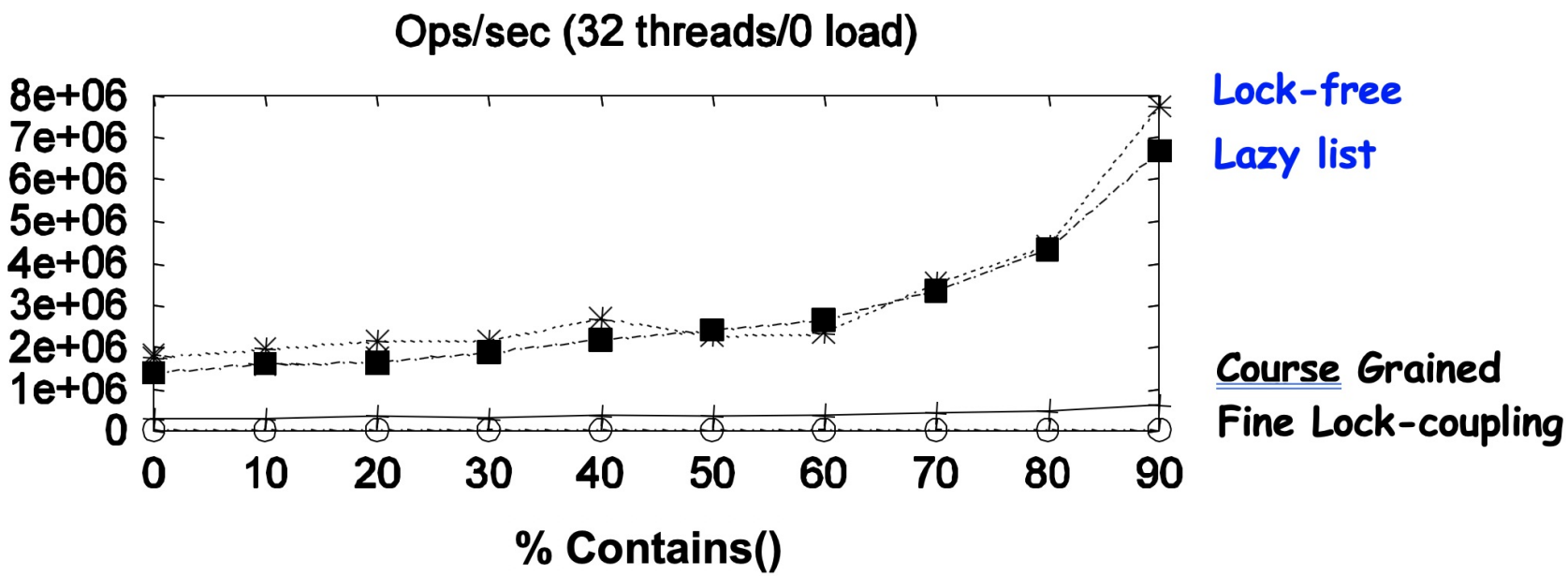
High contains ratio



Low contains ratio



Increasing contains ratio



Summary: "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