LUND UNIVERSITY

Herlihy Ch 9. Linked Lists: The Role of Locking

Non-Blocking Synchronization

BJÖRN A. JOHNSSON

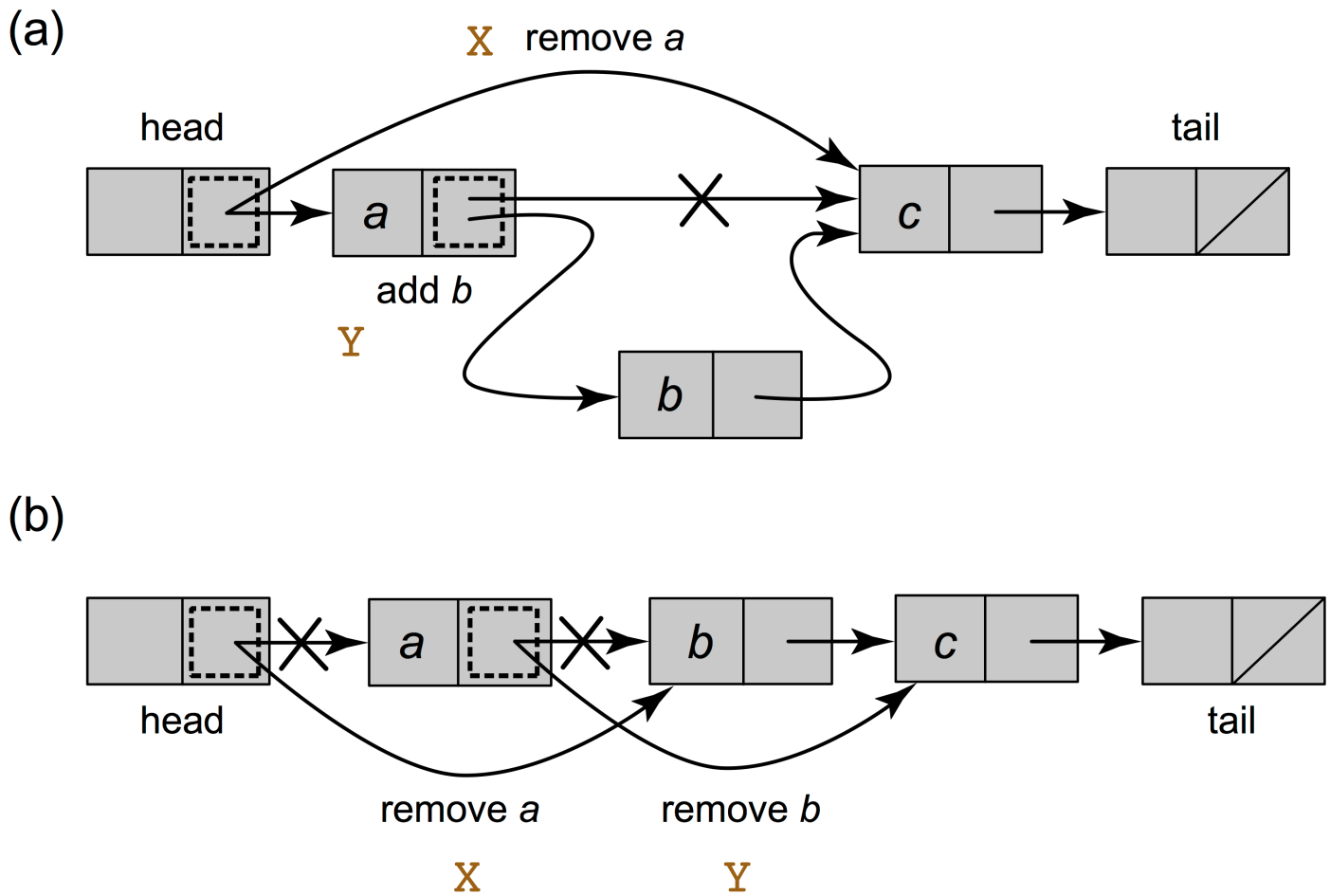
Overview?

- So far:
  - Coarse- and fine-grained synchronization
  - Optimistic synchronization
  - Lazy synchronization

- Now: **Non-blocking** synchronization

bjorn_a.johnsson@cs.lth.se

Naïve: just `compareAndSet()`



bjorn_a.johnsson@cs.lth.se

LUND UNIVERSITY

AtomicMarkableReference<T>

Pragma 9.8.1. An `AtomicMarkableReference<T>` is an object from the `java.util.concurrent.atomic package` that encapsulates both a reference to an object of type `T` and a `Boolean` mark. These fields can be updated atomically, either together or individually. For example, the `compareAndSet()` method tests the expected reference and mark values, and if both tests succeed, replaces them with updated reference and mark values. As shorthand, the `attemptMark()` method tests an expected reference value and if the test succeeds, replaces it with a new mark value. The `get()` method has an unusual interface: it returns the object's reference value and stores the mark value in a Boolean array argument.

```
1  public boolean compareAndSet(T expectedReference,
2                               T newReference,
3                               boolean expectedMark,
4                               boolean newMark);
5  public boolean attemptMark(T expectedReference,
6                             boolean newMark);
7  public T get(boolean[] marked);
```

bjorn_a.johnsson@cs.lth.se

Revisions…

- `Node`'s `next` field now `AtomicMarkableReference<Node>`
- Thread `A` *logically* removes $\texttt{curr}_{\texttt{A}}$ by "marking" it's `next`
- *Physical* removal by other traversing threads
  - `add()` & `remove()` – traverse + physically remove marked nodes on the path to their target node[1]
- `contains()` same as in `LazyList` – performs no modifications to list

[1] Why? Exercise, that's why!

bjorn_a.johnsson@cs.lth.se

Code 1(3)

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

```
class Window {
  public Node pred, curr;
  Window(Node myPred, Node myCurr) {
    pred = myPred; curr = myCurr;
  }
}
```

bjorn_a.johnsson@cs.lth.se

Code 2(3)

```
public boolean add(T item) {
  int key = item.hashCode();
  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 AtomicMarkableReference(curr, false);
      if (pred.next.compareAndSet(curr, node, false, false))
        return true;
    }
  }
}
```

```
public boolean remove(T item) {
  int key = item.hashCode();
  boolean snip; // logical remove OK?
  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.compareAndSet
        (succ, succ, false, true);
      if (!snip)
        continue;
      pred.next.compareAndSet(curr, succ, false, false);
      return true;
    }
  }
}
```

bjorn_a.johnsson@cs.lth.se

LUND UNIVERSITY

Code 2(3)

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

bjorn_a.johnsson@cs.lth.se

Discussion

- Granularity/lock-frequency gradually reduced

  ⇒ Fully nonblocking list!

- `LockFreeList` guarantees progress in the face of arbitrary delays. Price for strong progress guarantees:
  - The need to support atomic modification of `next` has an added performance cost
  - Concurrent cleanup when traversing can cause contention; may force ”unnecessary” traversal restart
- `LazyList` – no progress guarantees (blocks), but:
  - None of `LockFreeList`’s weaknesses
- Decision of approach depends on application

bjorn_a.johnsson@cs.lth.se

LUND
UNIVERSITY

Exercise!

In the `LockFreeList` algorithm, why must threads that add/remove nodes never traverse marked nodes, but instead physically remove them?

Illustrate your answer with a figure(s) similar to those in the chapter. Clarify your illustration(s) with a short *explanation*.

bjorn_a.johnsson@cs.lth.se

LUND UNIVERSITY