PURDUE UNIVERSITY

Principles of Concurrency

Lecture 5

Concurrent Data Structures

Material adapted from Herlihy and Shavit, Art of Multiprocessor Programming, Chapters 9 and 10

The Problem

- A concurrent data structure:
  - has state manipulated by a number of methods
  - these methods can be invoked concurrently
- How should we coordinate access?
  - Coarse-grained synchronization:
    - protect all methods with a lock
    - even if acquiring an releasing a lock is efficient, the level of concurrency admitted is low
  - Alternatives:
    - fine-grained synchronization:
      - logically break-up the object into multiple pieces
      - induce coordination only when methods interfere with the accesses they perform
    - optimistic synchronization:
      - assume conflicts won't occur (don't use any kind of synchronization)
      - validate the assumption was correct post-facto
    - lazy synchronization:
      - split a method action into multiple parts, deferring expensive operations
    - non-blocking synchronization
      - eliminate locks entirely, relying on low-level atomic primitives (e.g., CAS)

Running Example

A set specification:

```
data class Set<T> = {
   val add : T -> Boolean
   val remove: T -> Boolean
   val contains: T -> Boolean
}
```

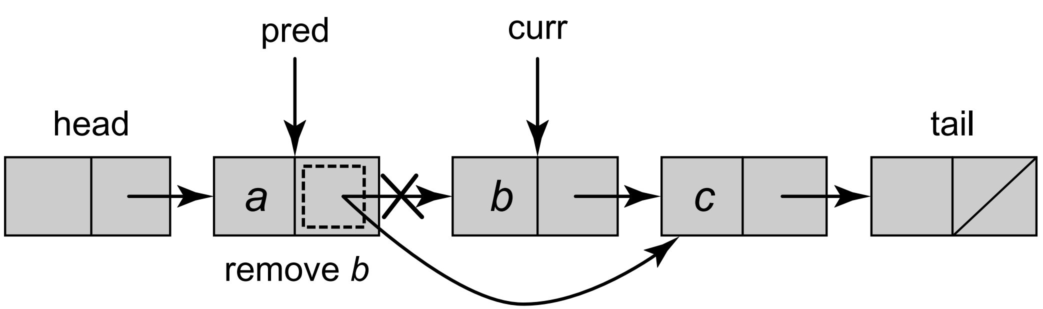
- `add(x)` adds `x` to the set and returns true only if the set did not contain `x` previously
- `remove(x)` removes `x` from the set and returns true only if `x` was in the set previously
- `contains(x)` returns true if the set contains `x`

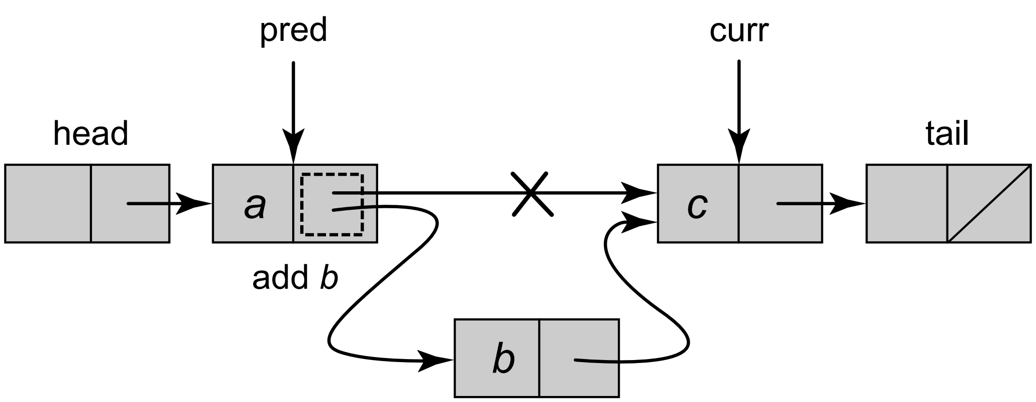
Implementation:

```
data class Node<T> = {
    val item : T
    val key : Integer
    val next : Node
}

val set : List<Node>
```

Sequential Behavior





`head` and `tail` are sentinel nodes used to record the front and end of the list

The `key` is a hash of the node's value used to order elements in the list

Properties

Want the implementation to preserve useful properties expressed by the specification

Assumption:

Freedom from interference: only the set implementation's methods have access to the list representation

Relate the abstraction (a set of values) with the implementation (an ordered list of nodes) using representation invariants:

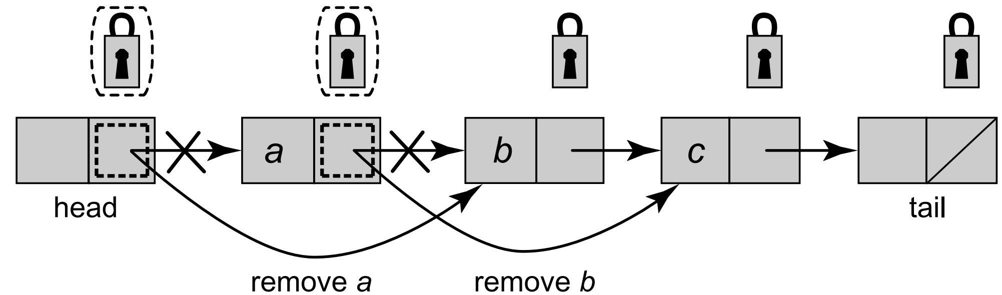
- Constraints on the representation that allow it behaviorally act like a set
- Serves as a contract among the implementation's methods
  - Sentinels are never added or removed
  - Every node (except the tail) in the list is reachable from the next field of another node
  - Distinct values always have distinct keys
  - …

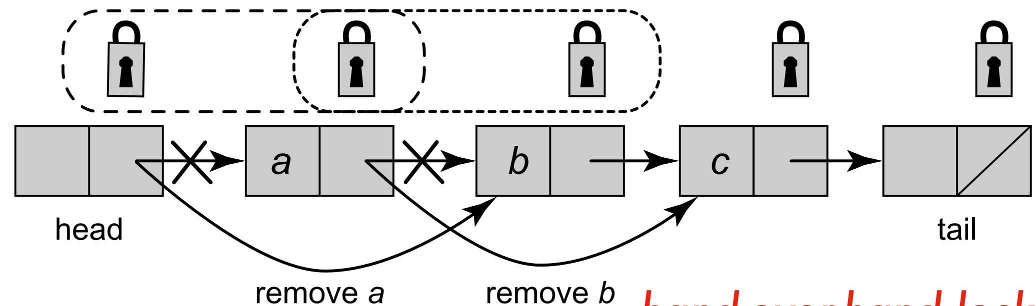
Safety properties:

Concurrent operations on a set are *linearizable* (i.e., methods appear to execute atomically): a set of concurrent method invocations always produces a state that represents a valid sequential execution

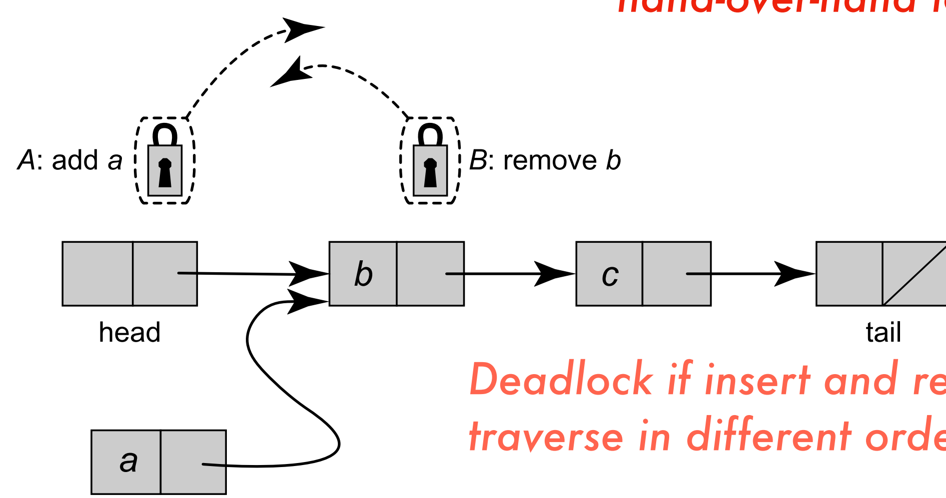
Fine-Grained Synchronization


```
fun <T>insert(item : T) = {
     head.acquire()
     val pred = head
     val curr = pred.next
     curr.acquire()
     while (curr.key < key) {
        pred.release()
        pred = curr
        curr = curr.next
        curr,acquire()
    }
    if (curr.key == key) {
      return false
   }
   val newNode = new Node(item)
   newNode.next = curr
   pred.next = newNode
   curr.release
   pred.release
   return true
}
```





hand-over-hand locking



Deadlock if insert and remove traverse in different order

Exercise: implement remove()

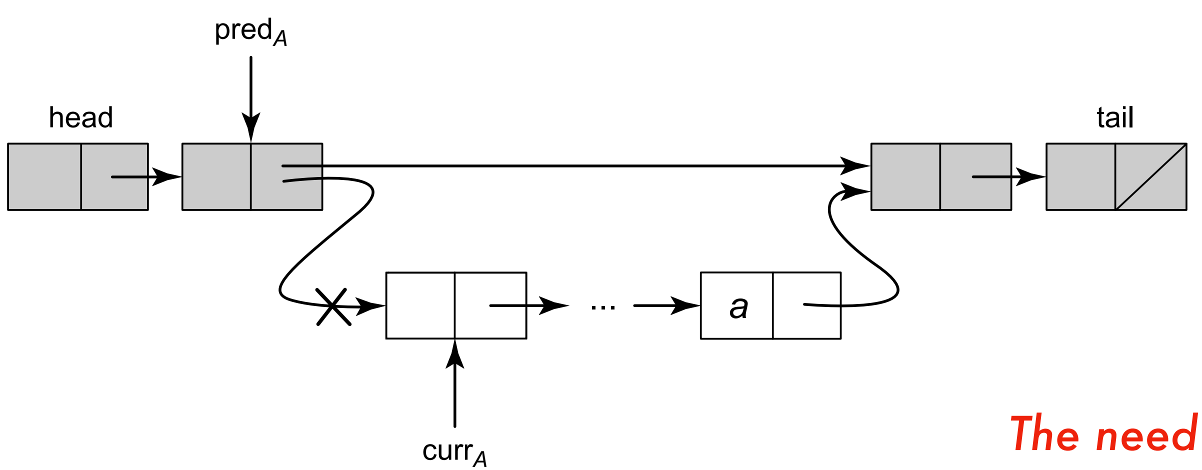
Optimistic Synchronization

Fine-grained locking is an improvement over coarse-grained locking, but:

- potentially involves long series of lock acquisitions and releases
- induces bottlenecks when a thread is manipulating the earlier part of the list

Alternative approach:

- optimistically traverse the list (without using locks)
- when ready to insert, validate that the predecessor and successor nodes are still reachable (use locks for this purpose)
- similar protocol for remove



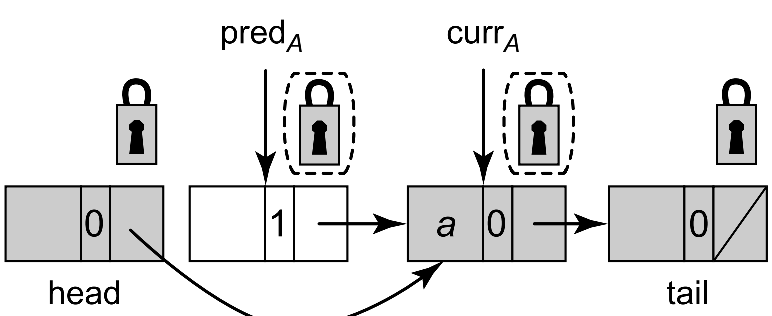
The need for validation

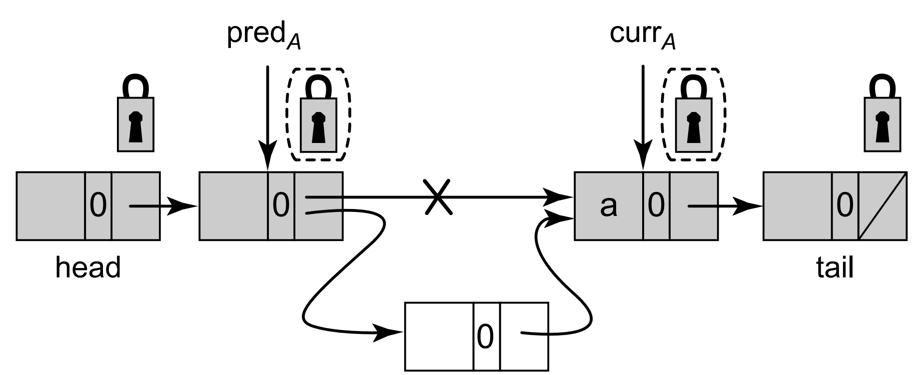
Lazy Synchronization

- Optimistic synchronization works well if the cost of traversing a list twice (without locks) is faster than traversing it once with locks
- Can we improve this so that insert() and remove() traverse a list only once

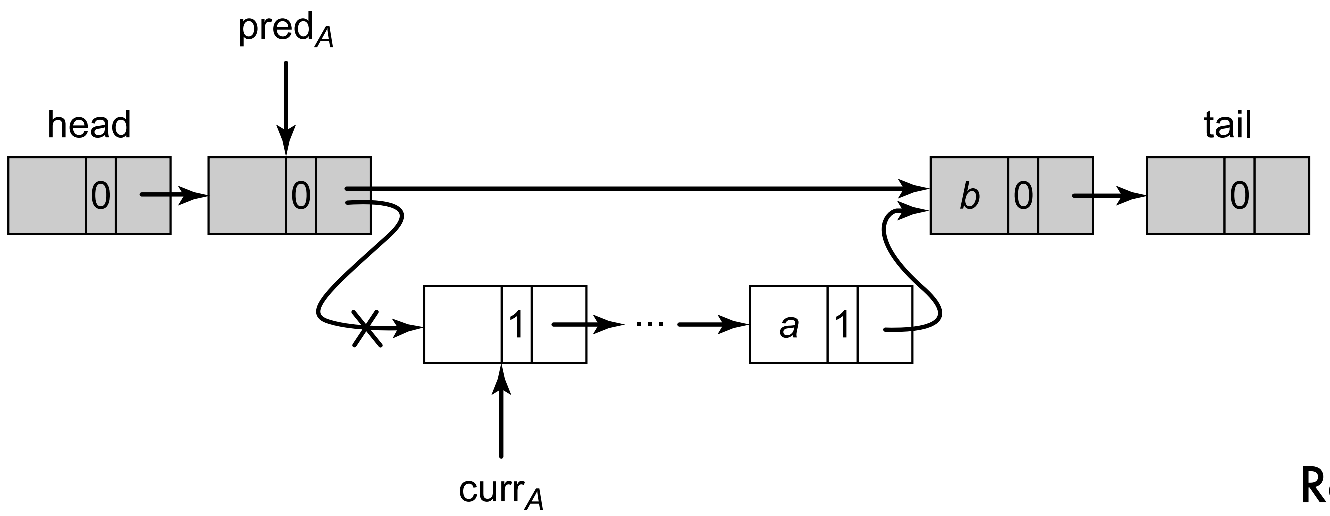
Add an extra marked bit to every node indicating if it is reachable from the head
Invariant: every unmarked node is reachable

No need to lock target node
Insert locks predecessor
Remove: (1) marks the target node (logical removal)
(2) updates the predecessor to point to the target's successor (physical removal)
Validation checks if the predecessor and current nodes are not marked and the current predecessor still points to the current node

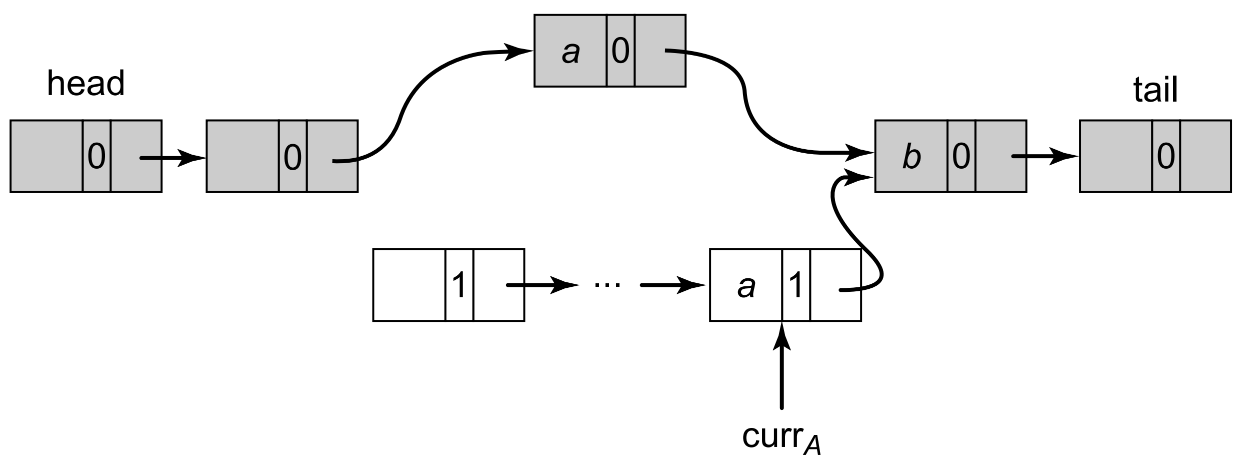




Lazy Synchronization



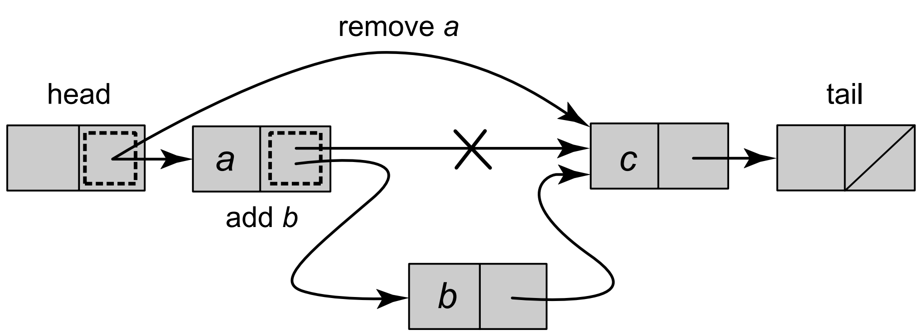
Reasoning about `contains()`



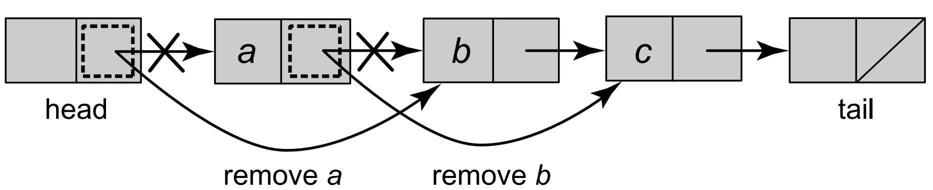
Non-blocking Synchronization

All previous approaches involve locks at some point in the implementation.
Can we devise a solution that eliminates locks altogether?

Need a way to ensure a node's fields cannot be updated after it has been logically or physically removed. *Approach*: treat a node's next and marked fields as an atomic unit: attempting to update the next field if the marked bit is set will fail



Use a variant of compare-and-swap() to achieve this behavior



The need for atomic update of mark and next fields

Non-blocking Synchronization


```
fun <T>add(item :T) = {
  val key = item.hash()
  while (true) {
     val (pred, curr) = find(head, key)
     if (curr.key == key) {
        return false
     } else {
         val node = new Node(item)
         node.next = AtomicMarkableReference(curr, false)
         if pred.next.CAS(curr, node, false, false) {
          return true
       }
    }
}
```

Atomically update the new node's mark bit and its next field

Atomically set the predecessor's next field to node and set its marked bit to false