PURDUE UNIVERSITY

Principles of Concurrency

Week 4

Non-blocking 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
}
```

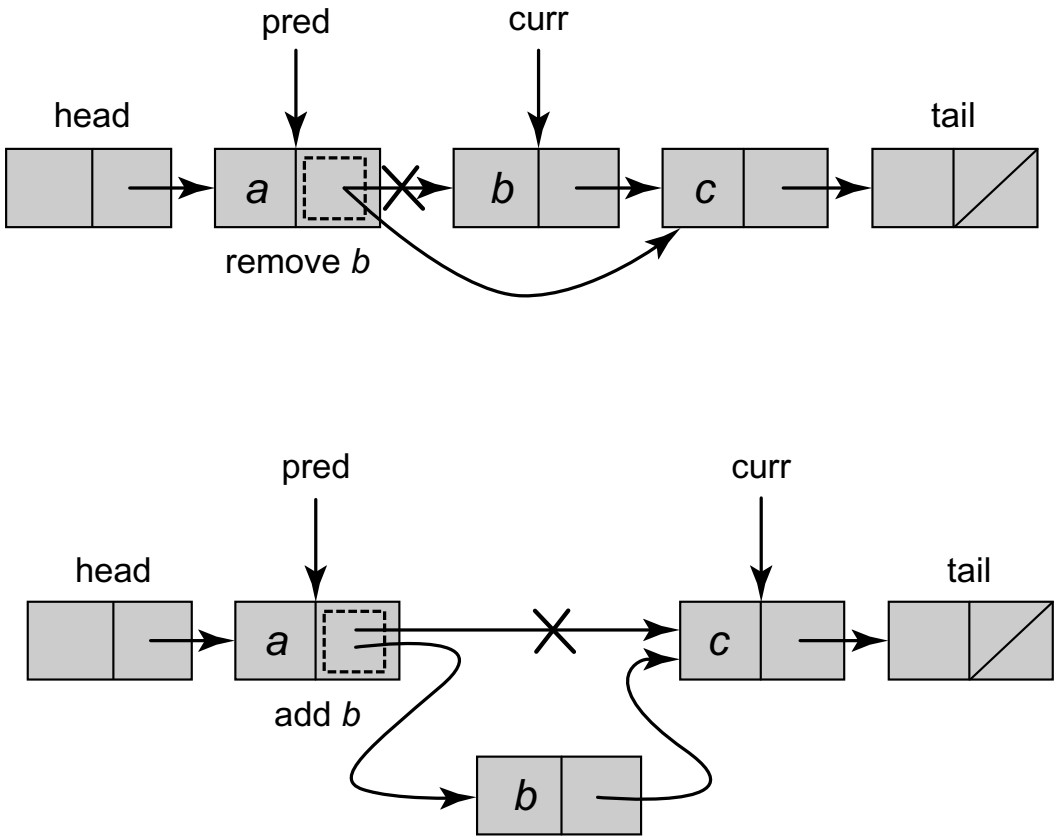
- `add(x)` adds `x` to the set and returns true only if the set did not contain `x` previously
- `remove(x)` removes `x` from the set and returns true only if `x` was in the set previously
- `contains(x)` returns true if the set contains `x`

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:

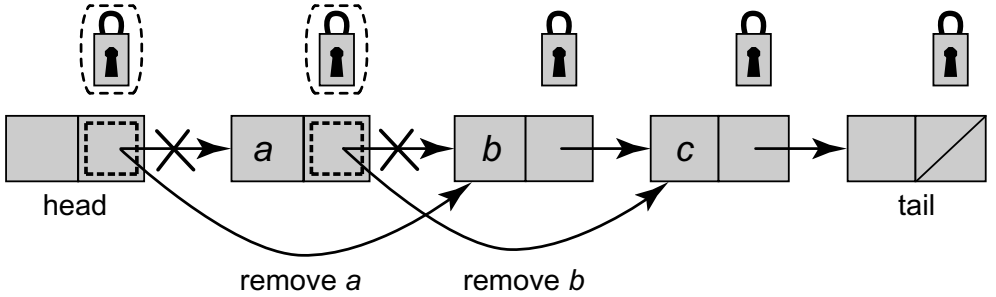
- Constraints on the representation that allow it to behaviorally act like a set
- Serves as a contract among the implementation's methods
  - Sentinels are never added or removed
  - Every node (except the tail) in the list is reachable from the next field of another node
  - Distinct values always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

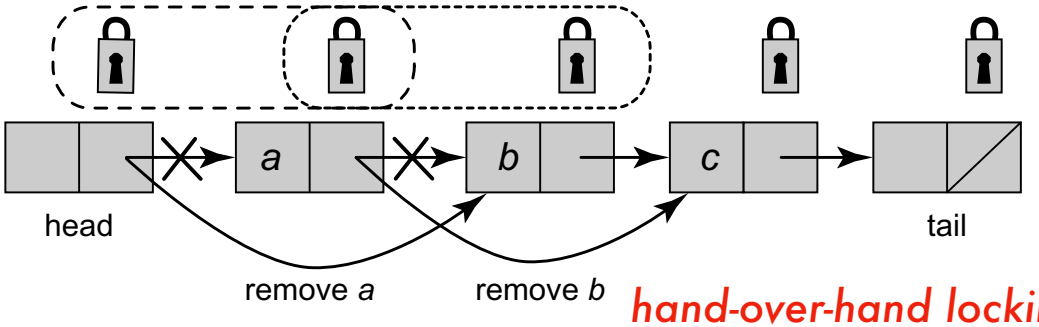
Safety properties:

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

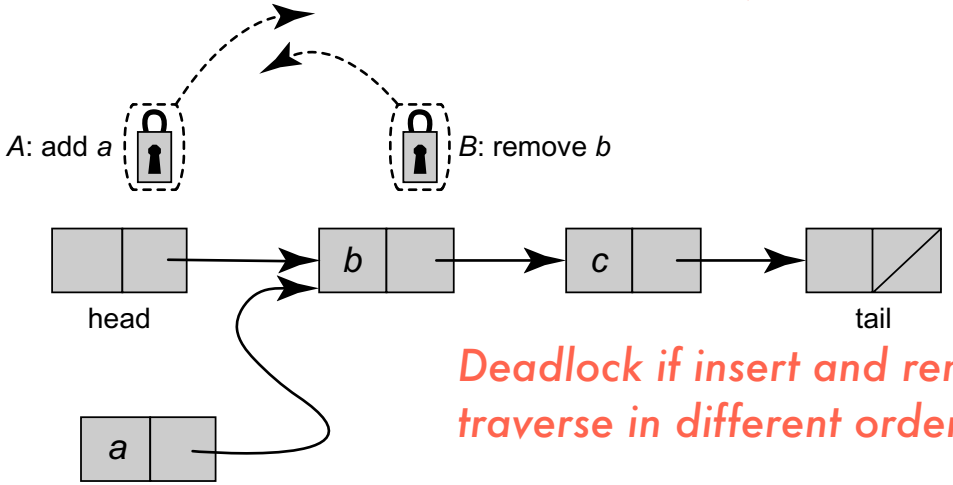
Fine-Grained Synchronization


```
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()

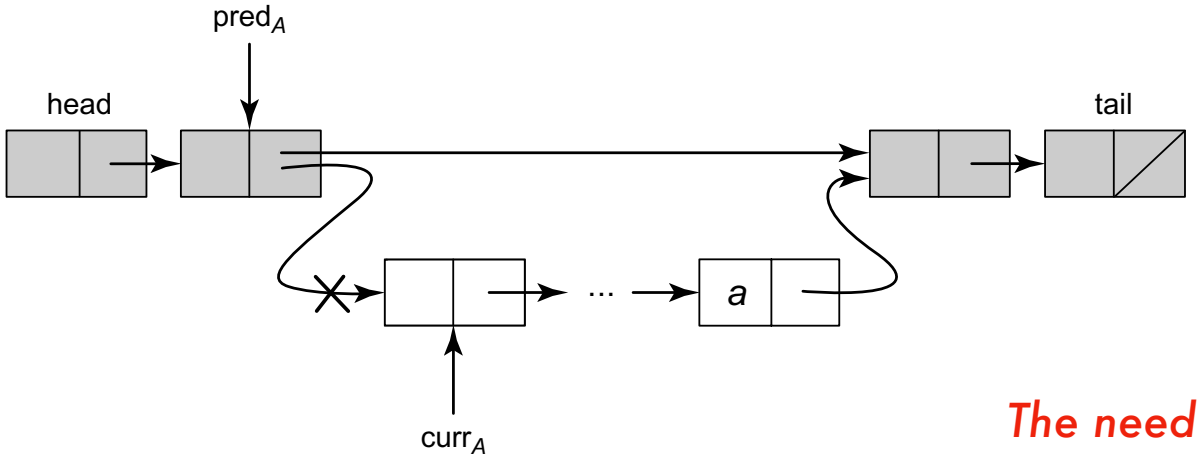
Optimistic Synchronization

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



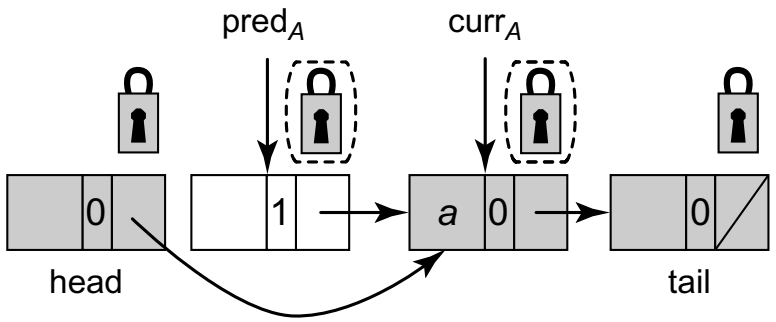
The need for validation

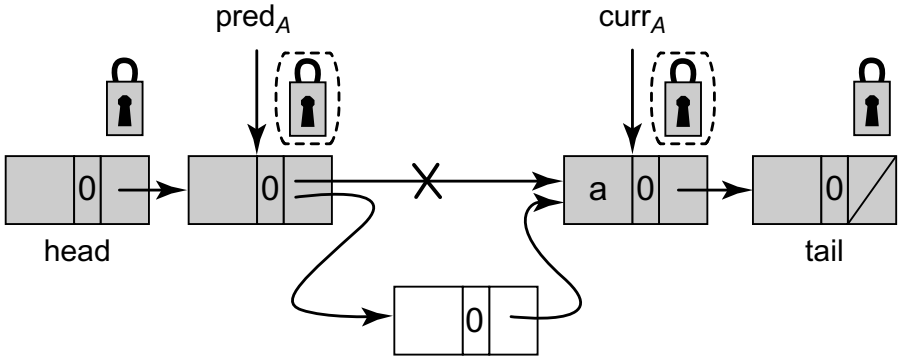
Lazy Synchronization

- 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

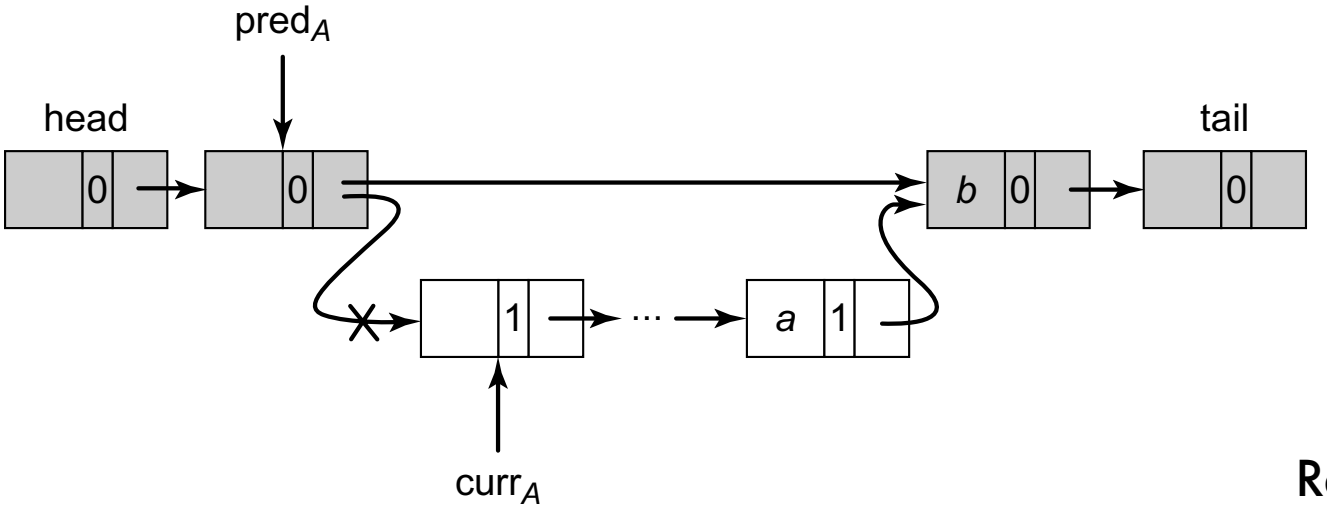
Add an extra marked bit to every node indicating if it is reachable from the head
Invariant: every unmarked node is reachable

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

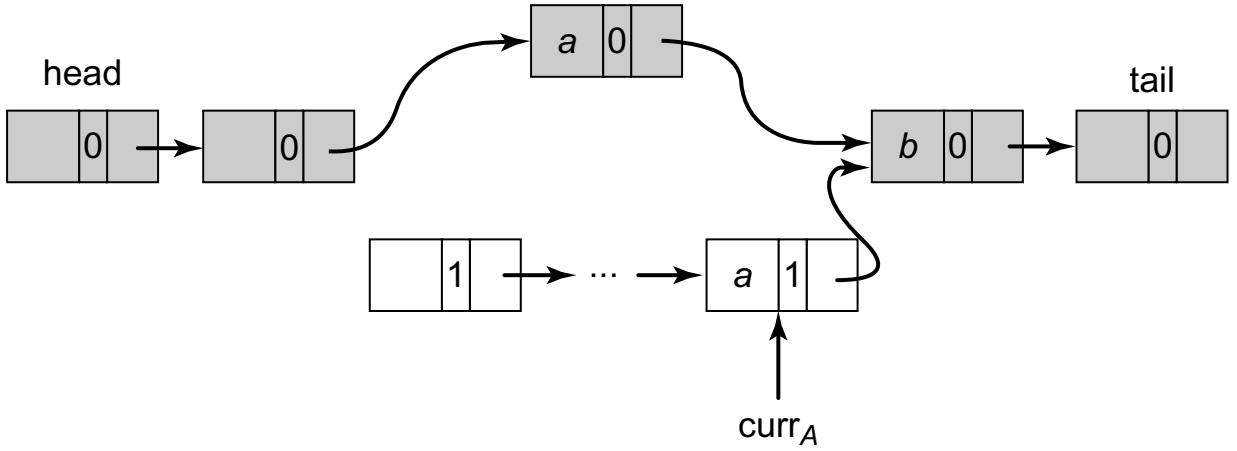




Lazy Synchronization



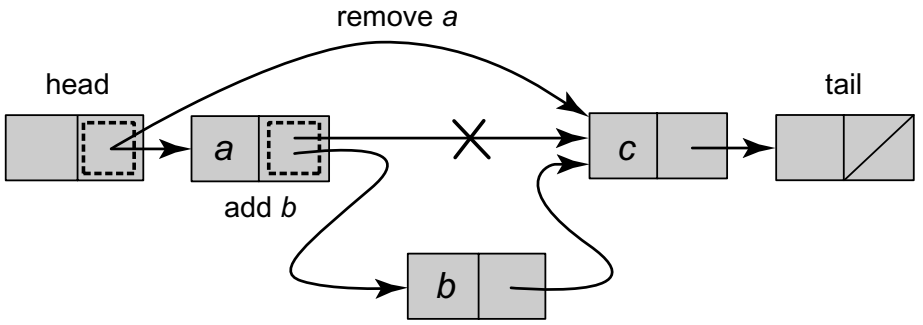
Reasoning about `contains()`



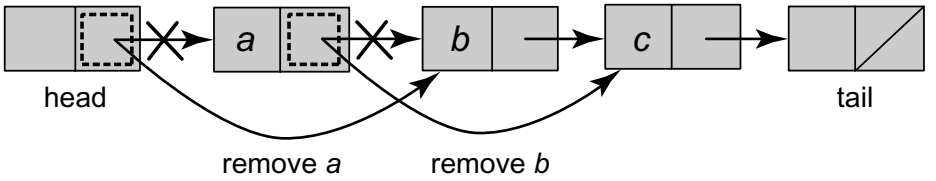
Non-blocking Synchronization

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

The Problem

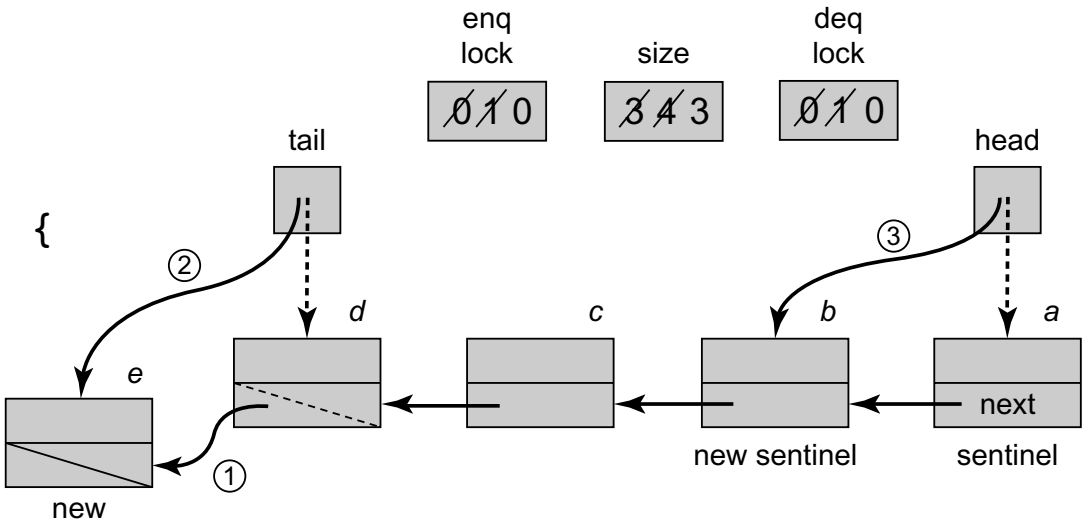
- Data structures like queues and stacks differ from sets:
  - no contains() method
  - an item can appear more than once
- These structures can be:
  - bounded or unbounded
  - total, partial, or synchronous:
    - total: every operation is non-blocking e.g., attempting to retrieve an element from an empty stack simply returns failure
    - partial: certain conditions may need to hold before an operation is allowed to complete, e.g., adding an element to a full bounded queue must wait until an element is removed
    - synchronous: one method waits for another to overlap its call/return interval, e.g., a method that adds an element to a queue blocks until a request to remove an element is received. These implementations implement a form of rendevous protocol.

Queues

A bounded partial queue

- enq() and deq() operations operate over different parts of the queue
- As long as the queue is neither full nor empty, these operations can proceed without interference
- Use locks for enq() and deq() to regulate concurrent enq() and deq() operations

```
fun <T> enq(x : T) = {
    var mustWakeDequeuers = false
    enq.acquire()
    while (size.get() == capacity {
       wait(notFullCondition)
     }
    var node = new Node(x)
    tail.next = tail =  node
    if (size.getAndIncrement() == 0) {
       mustWakeDequeuers = true
    enq.release()
    if (mustWakeDequeuers) {
      deq.acquire()
      signal(notEmptyCondition)
      deq.release()
    }
```



Abstract queue's head and tail not necessarily the same as implementation's. Why is this ok?

Unbounded Lock-Free Queue

- An unbounded total queue always successfully enqueues and item, with deq() throwing an exception if the queue is empty.
- This implementation does not requiring checking any conditions before proceeding with an operation.

```
fun <T>enq(item : T) = {
   var node = new Node(value)
   while (true) {
      val last = tail.get()
      val next = last.next.get()
      if (last == tail.get()) {
        if (next == null) {
           if (last.next.CAS(next, node))
              tail.CAS(last, node)
              return
          } else {
              tail.CAS(last, next)
           }
     }
}
```

create the node

locate presumed last element

get this last element's next pointer (when can it be non-null?)

if no new element concurrently added, append node

set the tail to point to the new node, return successfully even if the CAS fails

tail node already has a successor (half-finished prior enq()), help that prior node to complete

Appending a node to the list, and setting the tail to point to that node are not atomic. Use a "helping" mechanism to allow other nodes to finish the `enq()` of an operation that was preempted between these two actions.

Unbounded Lock-free Queue

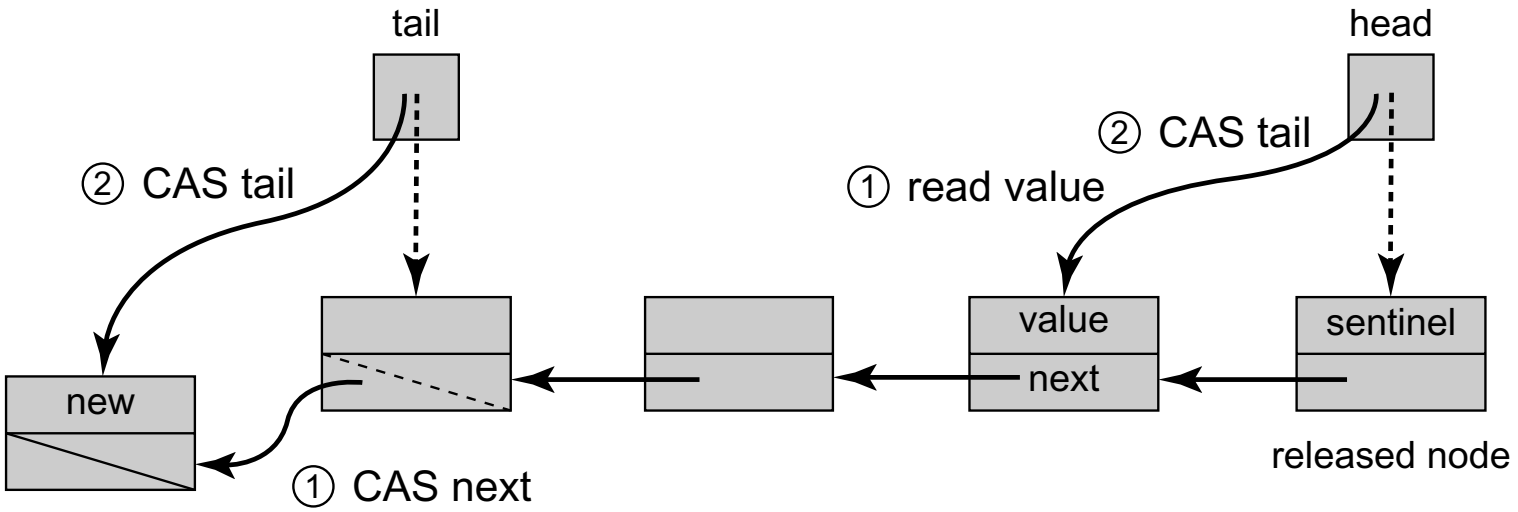

```
fun <T>deq() {
   while (true) {
      val first = head.get()
      val last = tail.get()
      val next = first.next.get()
      if (first == head.get()) {
        if (first == last) {
          if (next == null) {
            throw EmptyExn()
          }
          tail.CAS(last, next)
      } else {
          val value = next.value
          if (head.CAS(first, next) {
           return value
        }
      }
  }
}
```

Queue is empty: head and tail pointing to the same node, head's next is null

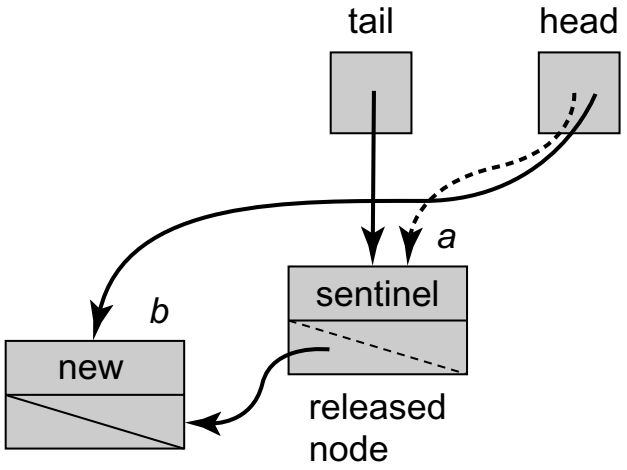
Tail is out-of-sync, help to move it to the next element

point head to the next element

Unbounded Lock-free Queue



Basic operation



Dequeuer's help fix-up lagging tail from concurrent enqueue operations

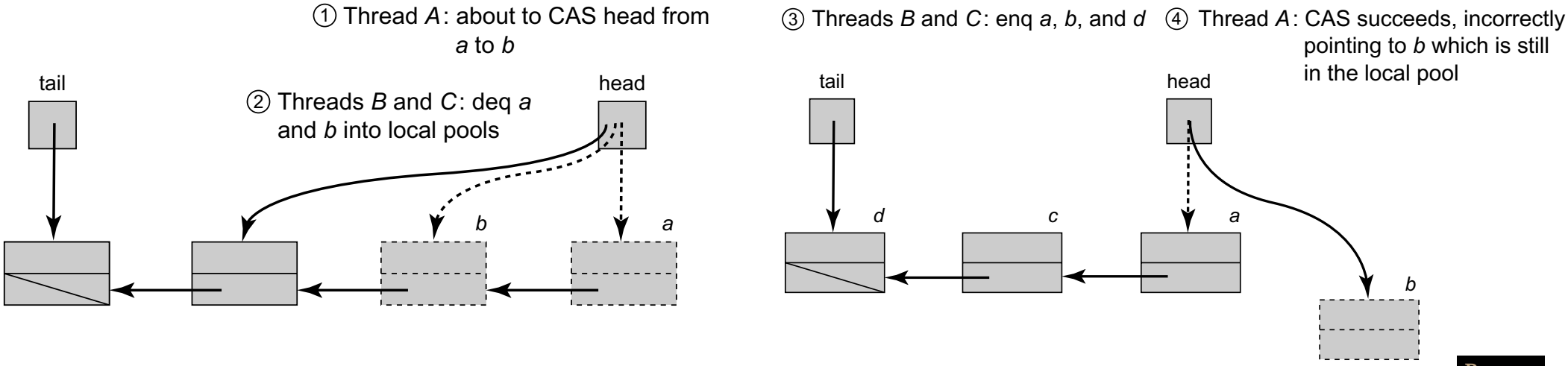
The ABA Problem

How do we deal with dequeued nodes? In a language without garbage collection, we need to provide our own memory management mechanism.

Idea: every thread maintains its own local node freelist. An enqueuing operation removes a node from the freelist (or allocates a new node if the list is empty). A dequeueing thread adds the node back to the freelist.

Subtle problem:

- A `deq()` operation observes a node `a` followed by `b`
- It attempts to update head to point to `b` using CAS and is preempted
- Concurrently, other `deq()` operations remove both `a` and `b`
- Node `a` is recycled
- The preempted node resumes and (incorrectly) observes that head still points to `a` and completes the CAS operation so that head now points to `b` (a recycled node)



Lock-free Stack


```
fun tryPush(n : Node) =  {
   val oldTop = top.get()
   n.next = oldTop
   top.CAS(oldTop, node)
}

fun <T>push(value : T) = {
   var node = new Node(value)
   while (true)  {
      if (tryPush(node)) {
        return
     }
}
```

When can the CAS operations in tryPush() and tryPop() fail?

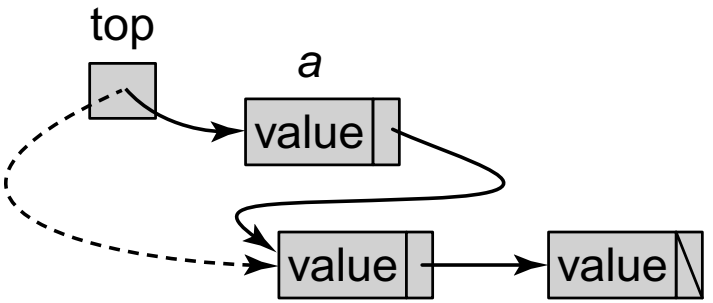
```
fun tryPop()  = {
   val oldTop = top.get()
   if (oldTop == null) {
      throw EmptyExn();
   }
   val newTop = oldTop.next()
   if (top.CAS(oldTop, newTop)) {
      return oldTop
   } else {
        return null
    }
}

fun <T>pop() = {
   while (true) {
      val returnNode = tryPop()
      if (returnNode != null) {
         return returnNode.value
     }
  }
}
```

Lock-free Stack

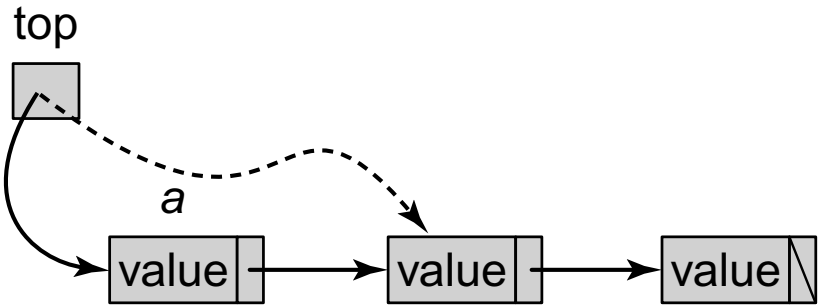
(a)

A:push()



(b)

A:pop()



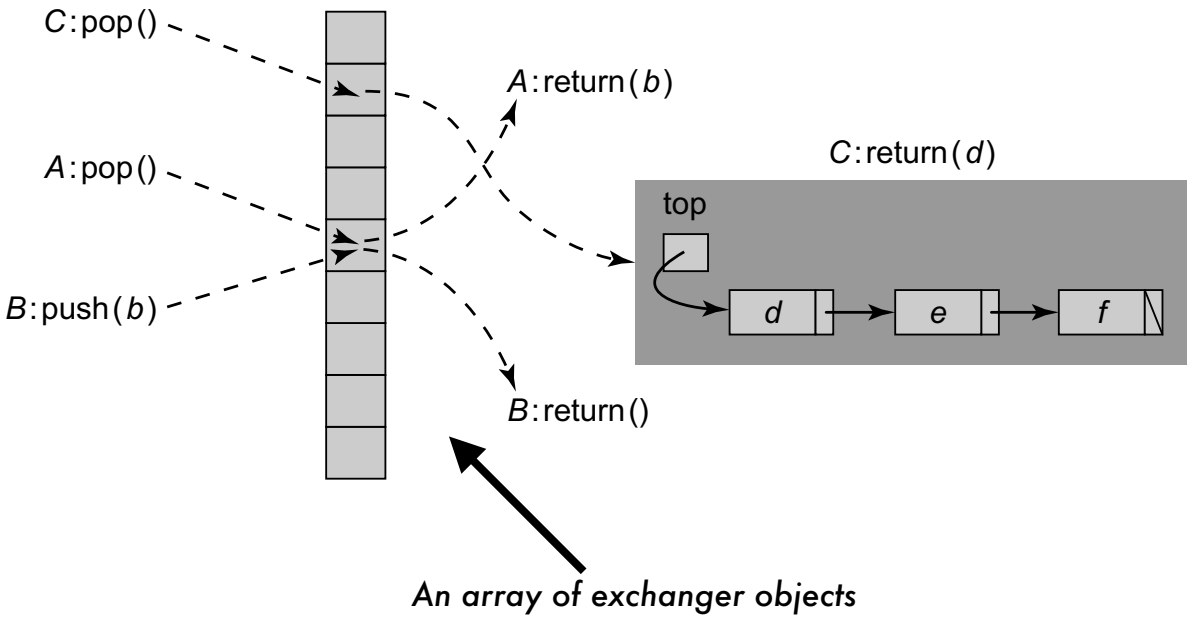
Elimination Stack

The lockfree stack has poor scalability:

- CAS operations sequentialize access to the stack's top field

Alternative approach:

- Pair concurrent pushes and pops - threads calling `push()` exchange values with threads calling `pop()`
- This exchange happens without modifying the stack



LockFreeExchanger:

- permits two threads to atomically exchange values
- First thread spins waiting for the second until a timeout

Three state automaton:

- EMPTY
- BUSY
- WAITING

1. Slot initially EMPTY
2. First thread sets it to WAITING using CAS
   - if not successful, retries
   - if successful, spins
3. Another thread that accesses this slot sets the state to BUSY
4. Item can be consumed and the state reset to EMPTY