CS 636: Concurrent Data Structures

Swarnendu Biswas

Semester 2020-2021-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Need for Concurrent Data Structures

Multithreaded/concurrent programming is now mainstream

Using more hardware resources may not always translate to speedup

Challenges with Concurrent Programming

Less synchronization ⟷ More synchronization

Order, atomicity & sequential consistency violations

Concurrent and correct

Deadlock

Poor performance: lock contention, serialization

Need for Concurrent Data Structures

Less synchronization

More synchronization

Order... sequent... vi...

...ck

Implies that languages and libraries should provide efficient portable data structures as building blocks

Poor performance: lock contention, serialization

Designing a Concurrent Set Data Structure

Designing A Set Data Structure

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

It is expected that there will significantly more calls to `contains()` than `add()` and `remove()`

`add(x)`

- adds x to the set and returns true if and only if x was not already present

`remove(x)`

- removes x from the set and returns true if and only if x was present

`contains(x)`

- returns true if and only if x is present in the set

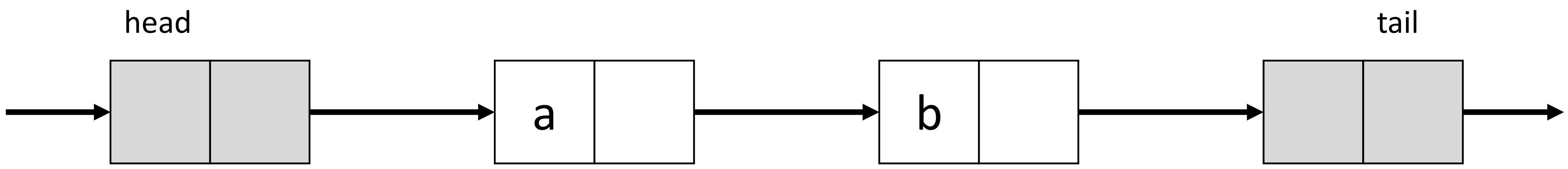
Designing A Set Data Structure using Linked Lists

```
class Node {
  T data;
  int key;
  Node next;
}
```

- Two sentinel nodes `head` and `tail`

- `key` field is the `data`'s hash code to help with efficient search
- Assume that all hash codes are unique
- Removed nodes continue to represent valid memory locations

A Set Instance



Invariants

- No duplicates
- Nodes are sorted based on the `key` value
- Sentinel nodes are immutable, and `tail` is reachable from `head`

A Thread-Unsafe Set Data Structure

```
public class UnsafeList<T> {
  private Node head;

  public UnsafeList() {
    head = new Node(Integer.MIN_VALUE);
    head.next = new Node(Integer.MAX_VALUE);
  }
```

A Thread-Unsafe Set Data Structure: `add()`

```
public boolean add(T x) {
  Node pred, curr;
  int key = x.hashcode();
  pred = head;
  curr = pred.next;
  while (curr.key < key) {
    pred = curr;
    curr = curr.next;
  }
  if (key == curr.key) {
    return false;
  } else {
    Node node = new Node(x);
    node.next = curr;
    prev.next = node;
    return true;
  }
}
```

A Thread-Unsafe Set Data Structure: `remove()`

```
public boolean remove(T x) {
  Node pred, curr;
  int key = x.hashcode();
  pred = head;
  curr = pred.next;
  while (curr.key < key) {
    pred = curr;
    curr = curr.next;
  }
  if (key == curr.key) {
    pred.next = curr.next;
    return true;
  } else {
    return false;
  }
}
```

A Thread-Unsafe Set Data Structure: `contains()`

```
public boolean contains(T x) {
  Node pred, curr;
  int key = x.hashcode();
  pred = head;
  curr = pred.next;
  while (curr.key < key) {
    pred = curr;
    curr = curr.next;
  }
    if (key == curr.key) {
      return true;
    } else {
      return false;
    }
  }
}
```

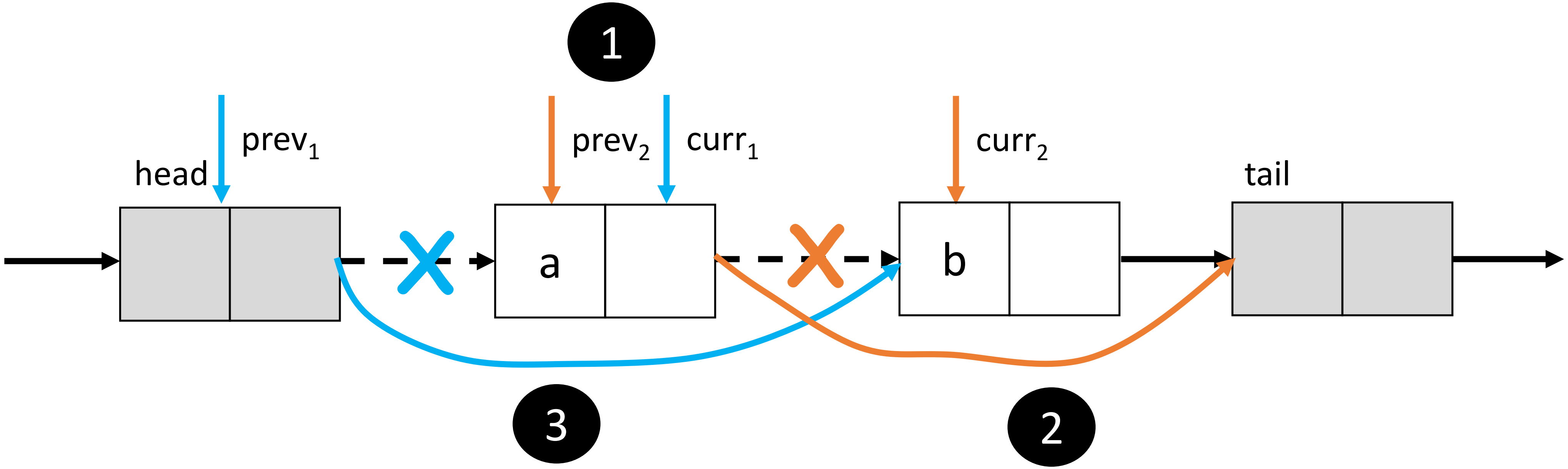
A Thread-Unsafe Set Data Structure: `remove()`

```
public boolean remove(T x) {
  Node pred, curr;
  int key = x.hashcode();
  pred
  curr
  while
    pred = curr;
    curr = curr.next;
  }
```

```
if (key == curr.key) {
  pred.next = curr.next;
  return true;
```

Can you give an example to show `remove()` is not thread-safe?

Unsafe Set: Incorrect `remove()`



- Thread 1 is executing `remove(a)`
- Thread 2 is executing `remove(b)`

A Concurrent Set Data Structure

```
public class CoarseList<T> {
  private Node head;
  private Lock lock = new ReentrantLock();

  public CoarseList() {
    head = new Node(Integer.MIN_VALUE);
    head.next = new Node(Integer.MAX_VALUE);
  }
…
}
```

A Concurrent Set Data Structure: `add()`

```
public boolean add(T x) {
  Node pred, curr;
  int key = x.hashcode();
  lock.lock();
  try {
    pred = head;
    curr = pred.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    if (key == curr.key) {
      return false;
    } else {
      Node node = new Node(x);
      node.next = curr;
      prev.next = node;
      return true;
    }
  } finally {
    lock.unlock();
  }
}
```

A Concurrent Set Data Structure: `remove()`

```
public boolean remove(T x) {
  Node pred, curr;
  int key = x.hashcode();
  lock.lock();
  try {
    pred = head;
    curr = pred.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    if (key == curr.key) {
      pred.next = curr.next;
      return true;
    } else {
      return false;
    }
  } finally {
    lock.unlock();
  }
}
```

A Concurrent Set Data Structure: `contains()`

```
public boolean contains(T x) {
  Node curr;
  int key = x.hashcode();
  boolean found = false;
  lock.lock();
  try {
    curr = head.next;
    while (curr.key < key) {
      curr = curr.next;
    }
    if (key == curr.key) {
      found = true;
    }
  } finally {
    lock.unlock();
  }
  return found;
}
```

Performance Metrics of Concurrent Data Structures

- Speedup measures how effectively is an application utilizing resources
  - Linear speedup is desirable
  - Data structures whose speedup grow with resources is desirable
- Amdahl's law says we need to reduce amount of serialized code
- Lock contention
  - Lock implementations with single memory location can introduce additional coherence traffic and memory traffic due to unsuccessful acquires
- Blocking or nonblocking

Challenges in Designing Concurrent Data Structures

- Multiple threads can access a shared object
  - E.g., a node in our Set data structure
- Situation:
  - Thread 1 is checking for `contains(a)`
  - Thread 2 is executing `remove(a)`
  - How do you reason about the outcome?

We need ways to describe the correctness conditions for operations on a concurrent object

Reasoning about Correctness

- Identify invariants and make sure they always hold
  - An item is in the set if and only if it is reachable from `head`
- Method call is the duration between an invocation event and a response event
  - Pre- and post-conditions encode the invariants before and after a method call
- Correctness (or safety) property is linearizability
- Progress (or liveness) property are starvation and deadlock-freedom

Correctness Condition – Sequential Consistency

Thread 1 `r.write(7)`

time

Thread 2 `r.write(-3)` `r.read(-7)`

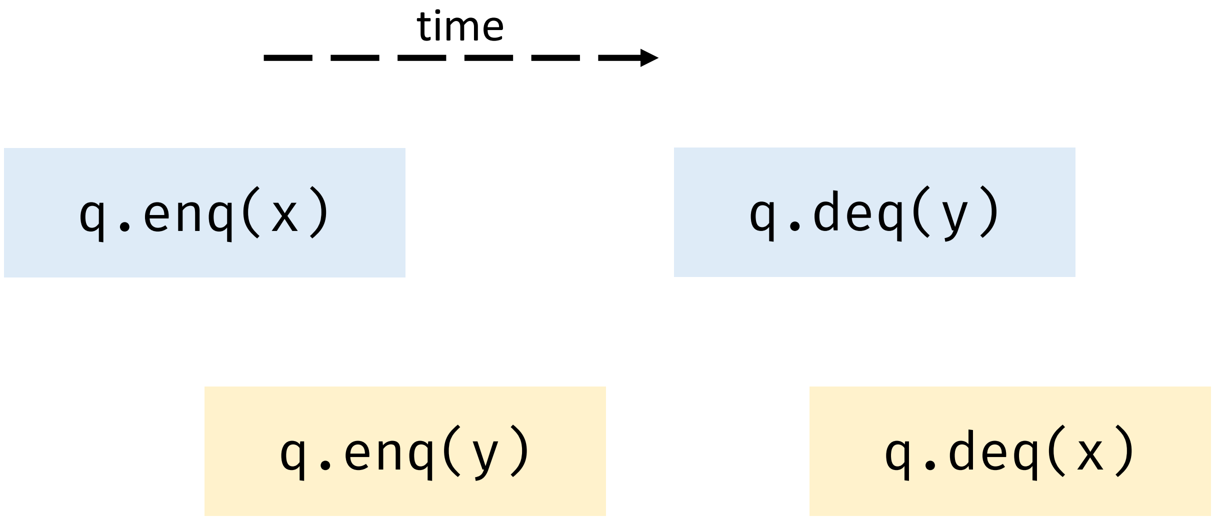
Method calls should appear to happen one-at-a-time in sequential order

`r.write(7)` `r.write(-3)` `r.read(7)`

Method calls should appear to take effect in program order

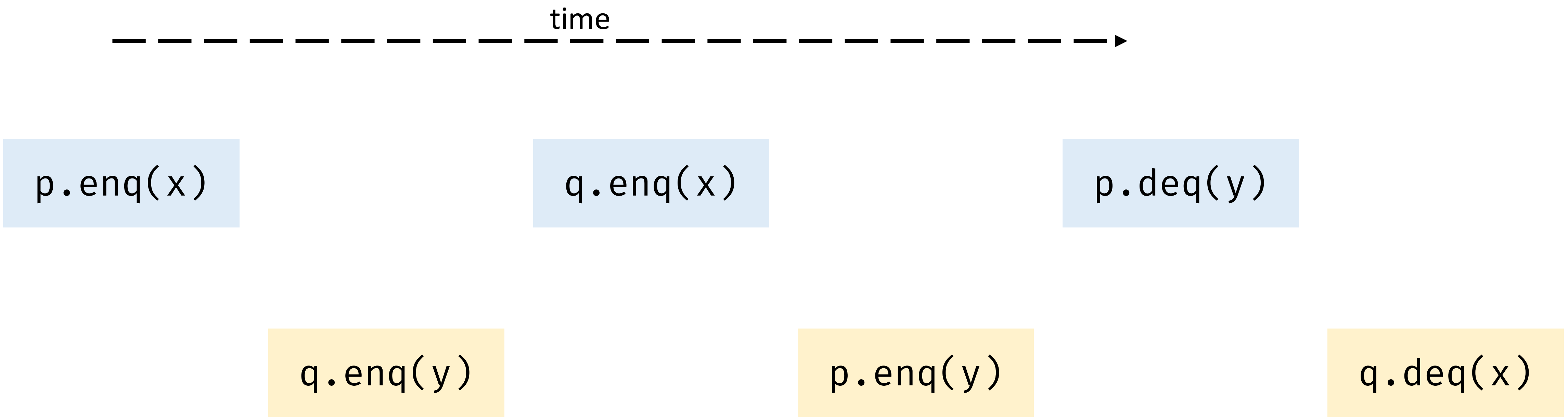
Sequentially Consistent Execution

- Two possible sequential orders



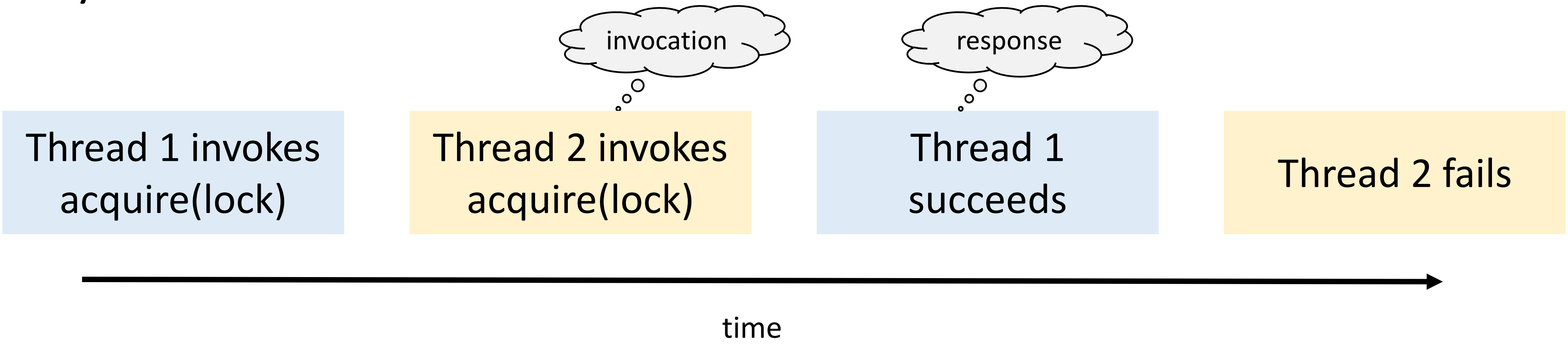
- Total and partial methods
  - A total method is defined for every object state
  - Any pending call to a total method can always be completed under SC (nonblocking)

SC is not Composable



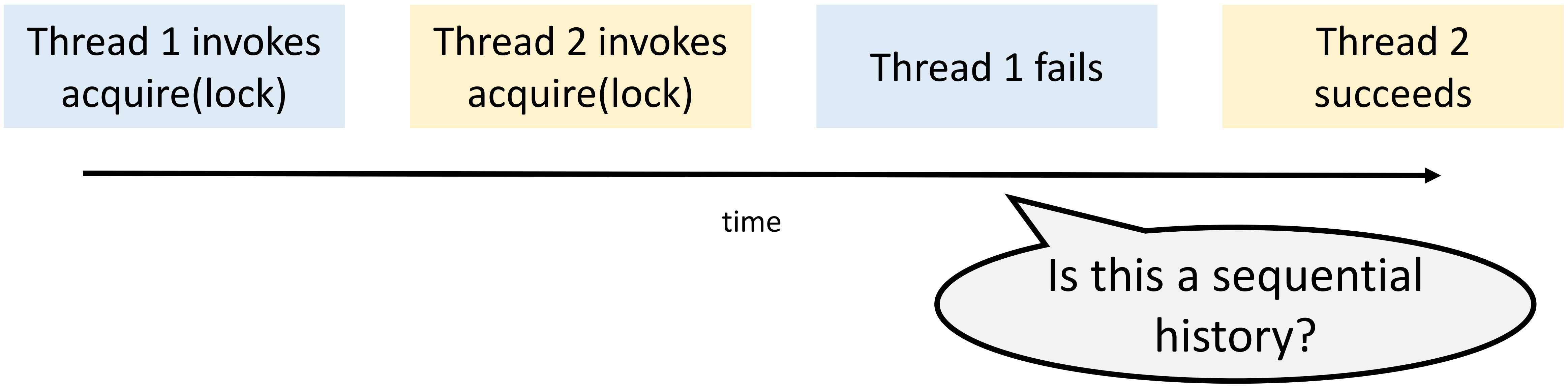
Understanding Linearizability

- Say you perform some operations on an object (e.g., a method call)
  - Each operation requires an invocation on that object, followed by a response
- A **history** is a sequence of invocations and responses on an object made by concurrent threads



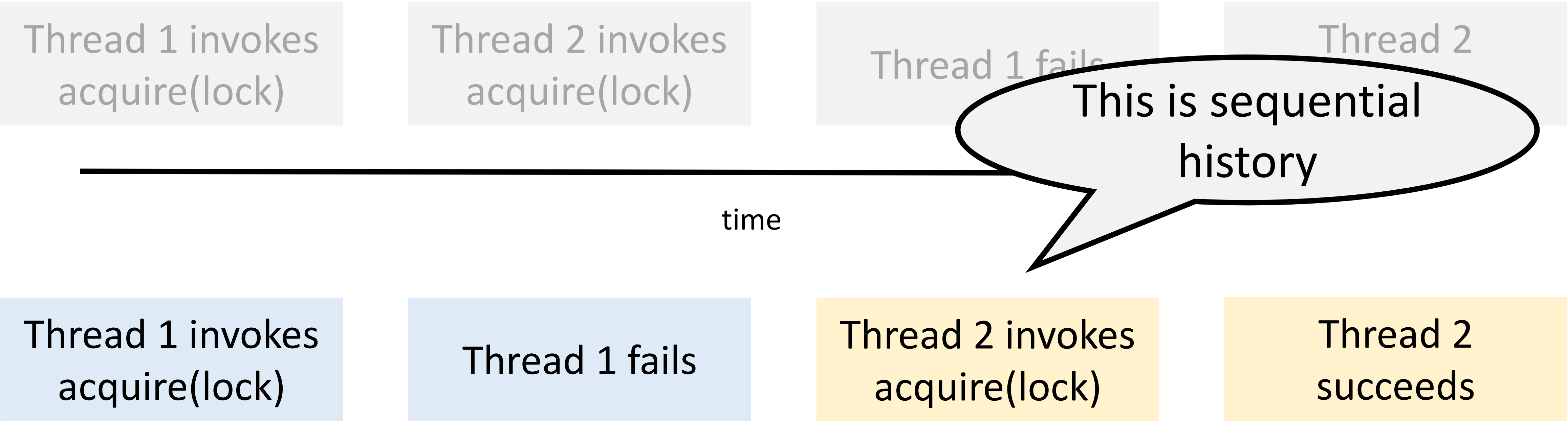
Understanding Linearizability

- **Sequential history** is where all invocations and responses are instantaneous
  - Starts with an invocation, last invocation may not have a response
  - Method calls do not overlap



Understanding Linearizability

- Sequential history is where all invocations and responses are instantaneous



Linearizability

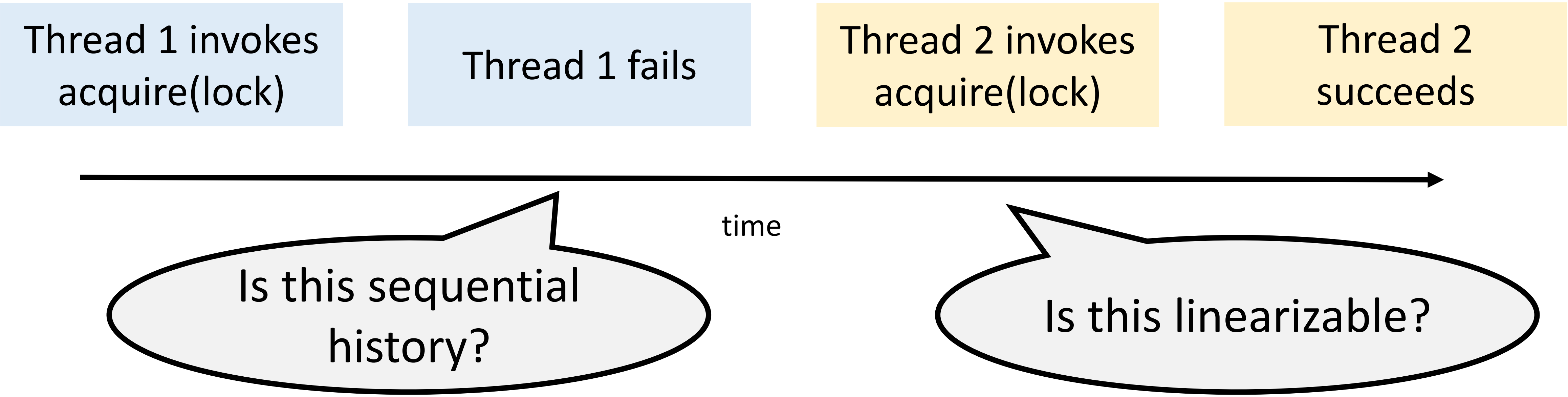
- **Idea**: Every concurrent history is equivalent to some sequential history
  - If one method call precedes another, then the earlier call must have taken effect before the later call
  - If two method calls overlap, we can order them in any way

- Consider a concurrent history (set of method calls) H and a valid sequential history S
- The history H is **linearizable** if
  - For every completed call in H, the call returns the same result as it would return if every operation in H would have been completed one after the other (i.e., in S)
  - If method call m_1 completes before method call m_2 in H, then m_1 precedes m_2 in S

Linearizability

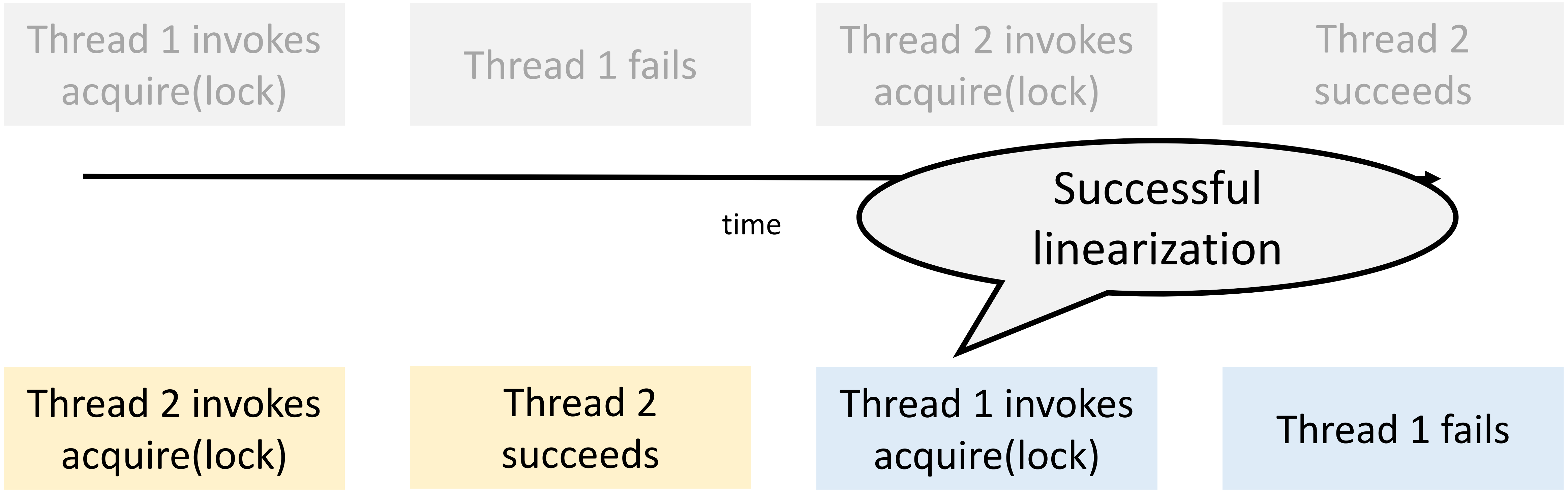
- The history H is **linearizable** if
  - For every completed call in H, the call returns the same result in the execution as it would return if every operation in H would have been completed one after the other (i.e., in S)
  - If method call m_1 completes before method call m_2 in H, then m_1 precedes m_2 in S
- Simpler words
  - Invocations and response can be reordered to form a sequential history
  - Sequential history is correct according to the semantics of the object
  - If a response preceded an invocation in the original history, it must still precede it in the sequential reordering

Understanding Linearizability



Understanding Linearizability

- Sequential history



Linearization Point

- Each method call appear to take effect instantaneously at some moment between its invocation and response
  - Linearization point is between the function invocation and response
  - Represents a single atomic step where the method call "takes effect"
- For lock-based synchronization, the critical section is the linearization point

What are the linearization points for `add()`, `remove()`, and `contains()` for the coarsely synchronized Set?

Sequential Consistency vs Linearizability

Sequential Consistency	Linearizability
• Method calls appear to happen instantaneously in some sequential order	• Method calls appear to happen instantaneously at some point between its invocation and response
• A sequentially consistent history is not necessarily linearizable	• Every linearizable history is sequential ly consistent
• Nonblocking but not composable	• Nonblocking and composable

Linearizability vs Serializability

Invalid semantics, cannot reorder, not linearizable

Thread 1 invokes acquire(lock)	Thread 1's acquire succeeds	Thread 2 invokes release(lock)	Thread 2's release succeeds	Thread 1 invokes release(lock)	Thread 1's release succeeds

time →

Thread 2 invokes release(lock)	Thread 2's release succeeds	Thread 1 invokes acquire(lock)	Thread 1's acquire succeeds	Thread 1 invokes release(lock)	Thread 1's release succeeds

Serializable

Linearizability vs Serializability

Linearizability	Serializability

• Property about operations on individual objects
  • Local property
• Requires real-time ordering
	• Property about transactions or group of operations on one or more objects
	  • Global property
	• Requires output is equivalent to some serial ordering

Linearizability vs Serializability

Linearizability	Serializability

• Property about operations on individual objects
  • Local property
• Requires real-time ordering
	• Property about transactions or group of operations on one or more objects
	  • Global property
	• Requires output is equivalent to some serial ordering

"Linearizability can be viewed as a special case of strict serializability where transactions are restricted to consist of a single operation applied to a single object" – Herlihy and Wing

Progress Guarantees

- A method is wait-free if it guarantees that every call finishes in a finite number of steps
- A method is lock-free if it guarantees that some call always finishes in a finite number of steps

Ideas in Implementing a Concurrent Data Structure

Coarse-grained synchronization

- Easy to get right, low concurrency, not scalable

Fine-grained synchronization

- Difficult to get right, more concurrent and scalable

???

Fine-Grained Synchronization

- Add a lock object to each list node

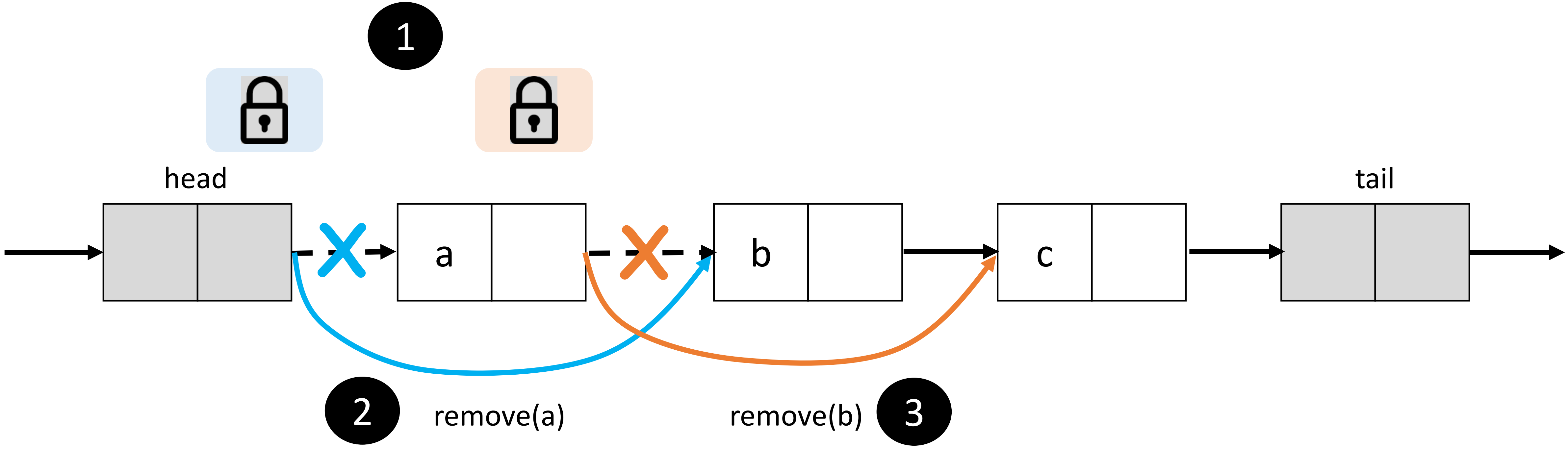
```
class Node {
  T data;
  int key;
  Node next;
  Lock lock;
}
```

What are a few possible ideas to implement `add()` and `remove()`?

Is one lock per node enough?

Thread 1	Thread 2
`curr.lock.lock();`	
`next = curr.next;`	
`curr.lock.unlock();`	
	`// Remove next from list`
`next.lock.lock();`	

Is one lock per node enough?



- Thread 1 is executing `remove(a)`
- Thread 2 is executing `remove(b)`

Fine-Grained Synchronization: add()

```
public boolean add(T x) {
  int key = x.hashcode();
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (key == curr.key) {
        return false;
      } else {
        Node node = new Node(x);
        node.next = curr;
        pred.next = node;
        return true;
      }
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

Fine-Grained Synchronization: add()

```
public boolean add(T x) {
  int key = x.hashcode();
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock()
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
```

```
      if (key == curr.key) {
        return false;
      } else {
        Node node = new Node(x);
        node.next = curr;
        prev.next = node;

    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

Where is the linearization point?

Fine-Grained Synchronization: add()

```
public boolean add(T x) {
  int key = x.hashcode();
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
```

```
if (key == curr.key) {
  return false;
} else {
  Node node = new Node(x);
  node.next = curr;
  prev.next = node;
```

Where is the linearization point?

- x is absent, predecessor’s next pointer is set to the new node
- x is present, the lock for node with value equal to x is acquired

```
    }
```

```
  }
}
```

Fine-Grained Synchronization: remove()

```
public boolean remove(T x) {
  int key = x.hashcode();
  head.lock();
  Node pred = null, curr = null;
  try {
    pred = head; curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (key == curr.key) {
        pred.next = curr.next;
        return true;
      } else {
        return false;
      }
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

Fine-Grained Synchronization: `remove()`

```
public boolean remove(T x) {
  int key = x.hashcode();
  head.lock();
  Node pred = null, curr = null;
  try {
    pred = head; curr = pred.next;

      curr.lock();
    }
```

```
if (key == curr.key) {
  pred.next = curr.next;
  return true;
} else {
  return false;
}
```

Where is the linearization point?

- x is present, predecessor's next pointer is set to the node after x
- x is absent, the lock for node with value greater than x is acquired

Fine-Grained Set Design

- Need to avoid deadlocks
  - Deadlocks are always a problem with fine-grained locking
  - For the Set data structure, each thread must acquire locks in some pre-determined order

Are there other problems with our fine-grained Set design?

Fine-Grained Set Design

- Need to avoid deadlocks
  - Deadlocks are always a problem with fine-grained locking
  - For the Set data structure, each thread must acquire locks in some pre-determined order

Are there other problems with our fine-grained Set design?

- Potentially long sequence of lock acquire and release operations
- Prohibits concurrent accesses to disjoint parts of the data structure

Ideas in Implementing a Concurrent Data Structure

Coarse-grained synchronization

- Easy to get right, low concurrency, not scalable

Fine-grained synchronization

- Difficult to get right, more concurrent and scalable

Optimistic synchronization

- Avoid synchronization to search, good for low contention cases

Lazy synchronization

- Defer expensive data structure manipulation operations

Nonblocking synchronization

Optimistic Synchronization

Optimistic strategy

- Access data without acquiring a lock
- Lock only when required
- **Validate** that the condition before locking is still valid
- If valid, then continue with access/mutation
- If invalid, start over

Optimistic strategy works well if conflicts are rare

Optimistic Synchronization: add()

```
public boolean add(T x) {
  int key = x.hashcode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) {
          return false;
        } else {
          Node node = new Node(x);
          node.next = curr; prev.next = node;
          return true;
        }
      }
    } finally {
      curr.unlock(); pred.unlock();
    }
  }
}
```

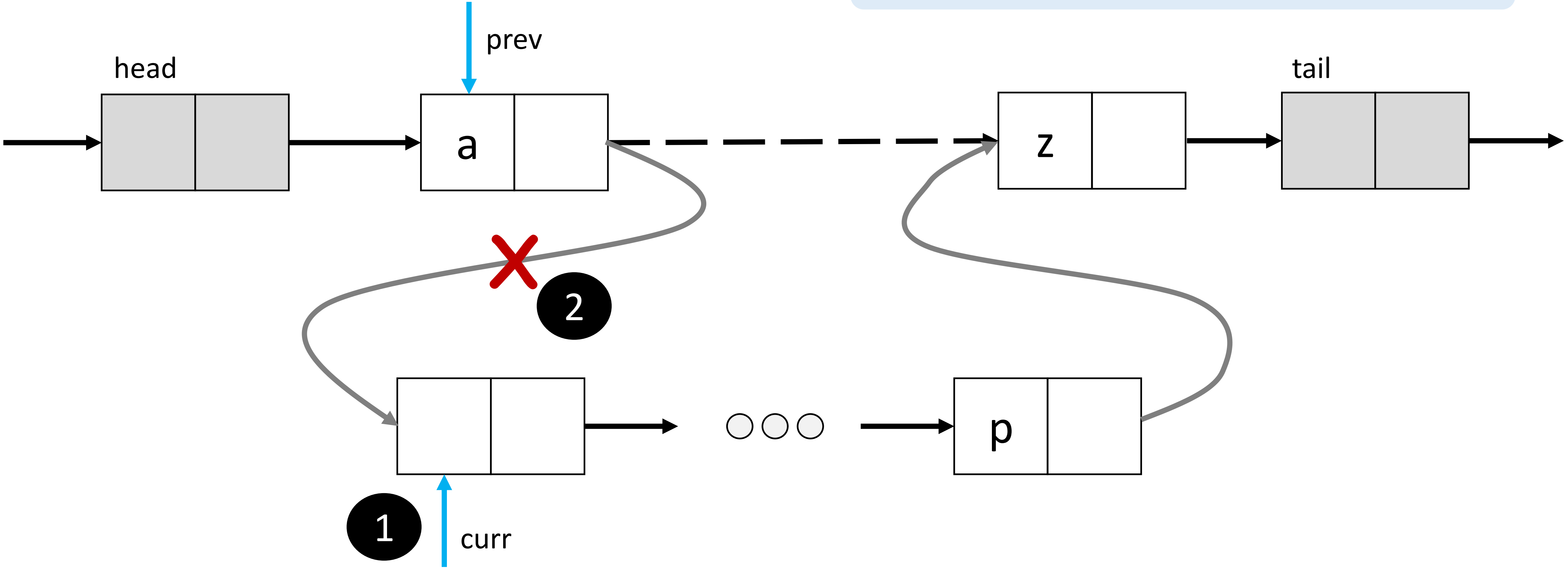
How could you validate?

- Double check that the optimistic result is still valid
- Check that `prev` is reachable from `head` and `prev.next == curr`

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

Is validation necessary?

- Thread 1 is executing `remove(p)`



Optimistic Synchronization: `remove()`

```
public boolean remove(T x) {
  int key = x.hashcode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    pred.lock(); curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key == key) {
            pred.next = curr.next;
            return true;
          } else {
            return false;
          }
        }
      } finally {
        curr.unlock(); pred.unlock();
      }
    }
  }
```

Optimistic Synchronization: `contains()`

```
public boolean contains(T x) {
  int key = x.hashcode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        return curr.key == key;
      }
    } finally {
      curr.unlock(); pred.unlock();
    }
  }
}
```

Optimistic Synchronization Design

Are there problems with our optimistic synchronization-based Set design?

Optimistic Synchronization Design

Are there problems with our optimistic synchronization-based Set design?

- Validation can be costly (for e.g., need to traverse the list)
- Need lock operations for `contains()` which is the most frequent method
  - Bad design in general

Lazy Synchronization

Delay mutation operations for a later time

- Add a mark/flag bit on each node to indicate deletion
- **Invariant**: every unmarked node is reachable from the head

Behavior

- `contains()`: needs only one wait-free traversal
- `add()`: traverses the list, locks the predecessor, and inserts the node
- `remove()`: mark the target node logically removing it, then redirect the predecessor's next link physically removing it

Lazy Synchronization: add()

```
public boolean add(T x) {
  int key = x.hashcode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key == key) {
            return false;
          } else {
            Node node = new Node(x);
            node.next = curr;
            pred.next = node;
            return true;
          } }
      } finally {
        curr.unlock(); }
    } } finally {
    pred.unlock();
  } } }
```

How could you validate?

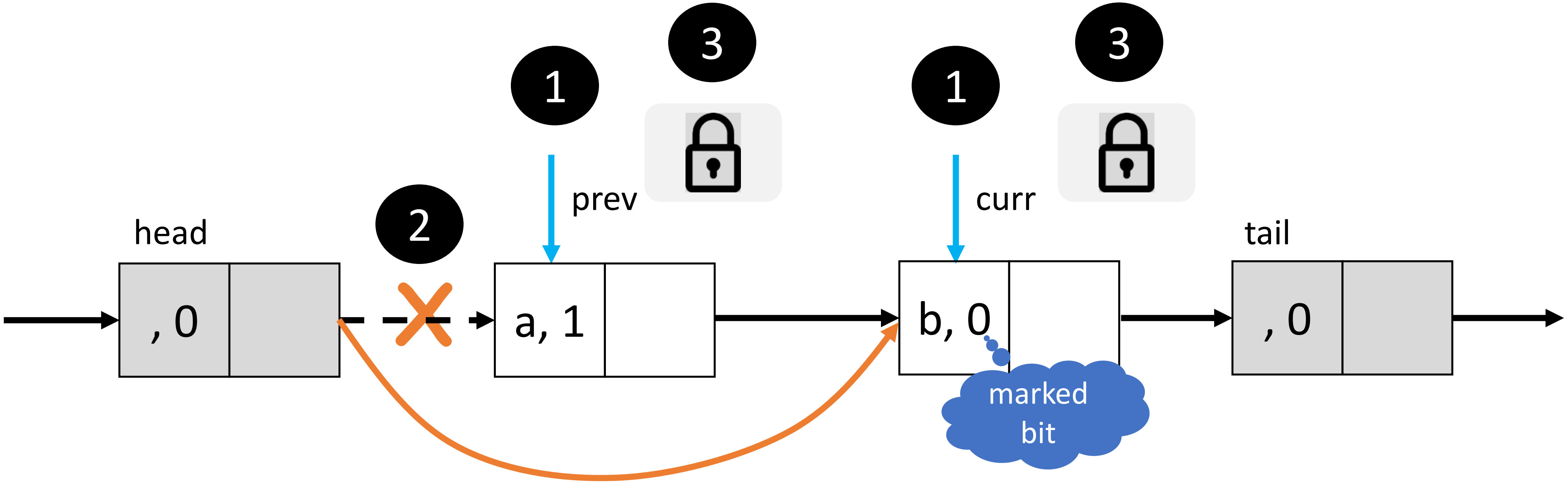
- Check that both `prev` and curr are unmarked and `prev.next == curr`

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

Lazy Synchronization: `remove()`

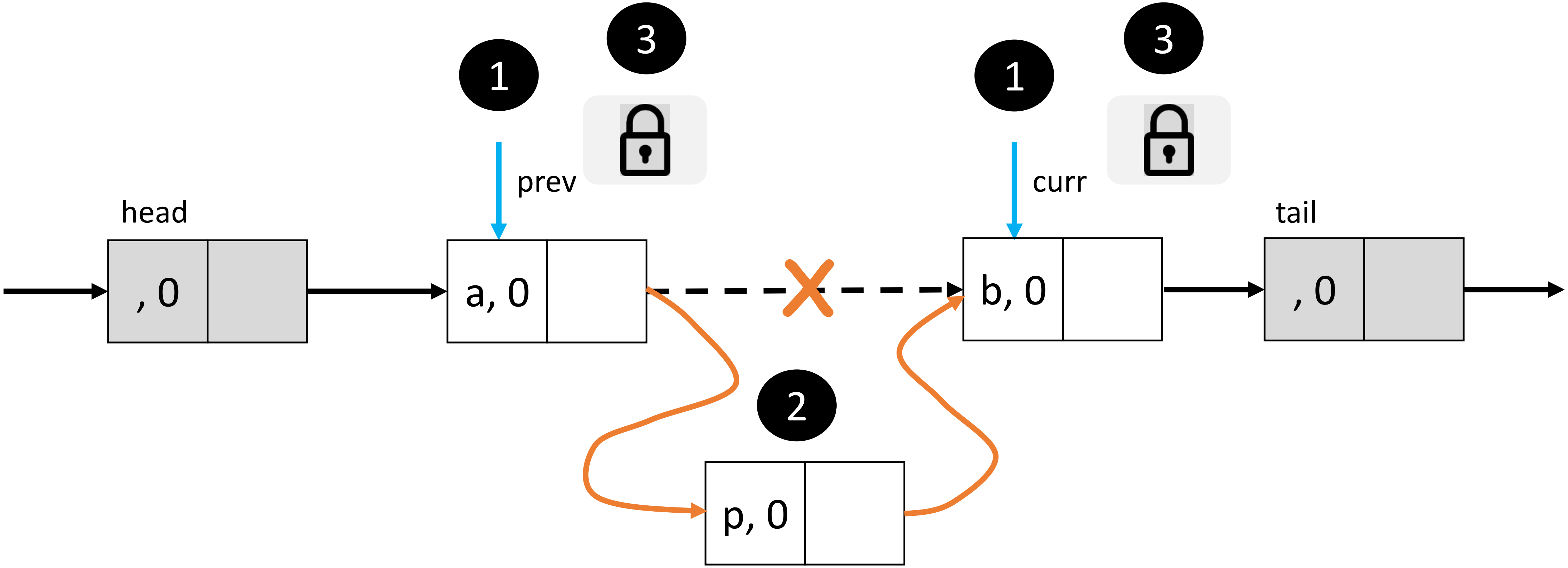
```
public boolean remove(T x) {
  int key = x.hashcode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key != key) {
            return false;
          } else {
            curr.marked = true;
            pred.next = curr.next;
            return true;
          }
        }
      } finally {
        curr.unlock(); }
      }
    } finally {
      pred.unlock();
    } } }
```

Detecting Conflicts: Scenario 1



- Thread 1 is executing `remove(b)`
- Thread 2 is executing `remove(a)`

Detecting Conflicts: Scenario 2



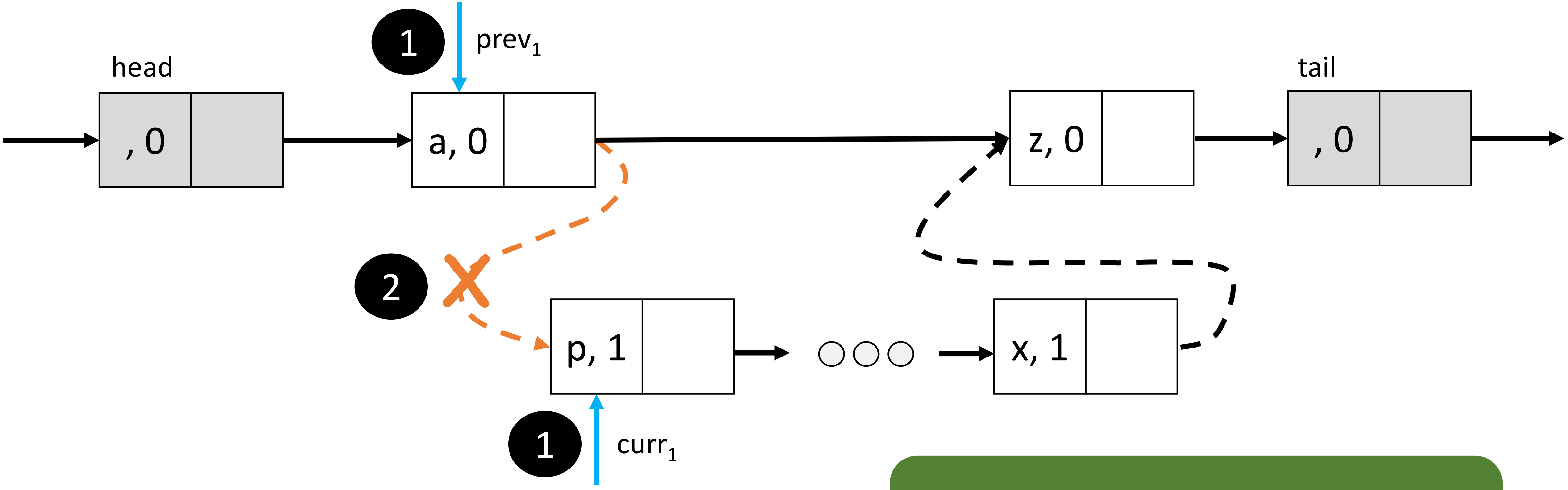
- Thread 1 is executing `remove(b)`
- Thread 2 is executing `add(p)`

Lazy Synchronization: `contains()`

```
public boolean contains(T x) {
  int key = x.hashcode();
  Node curr = head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

wait-free

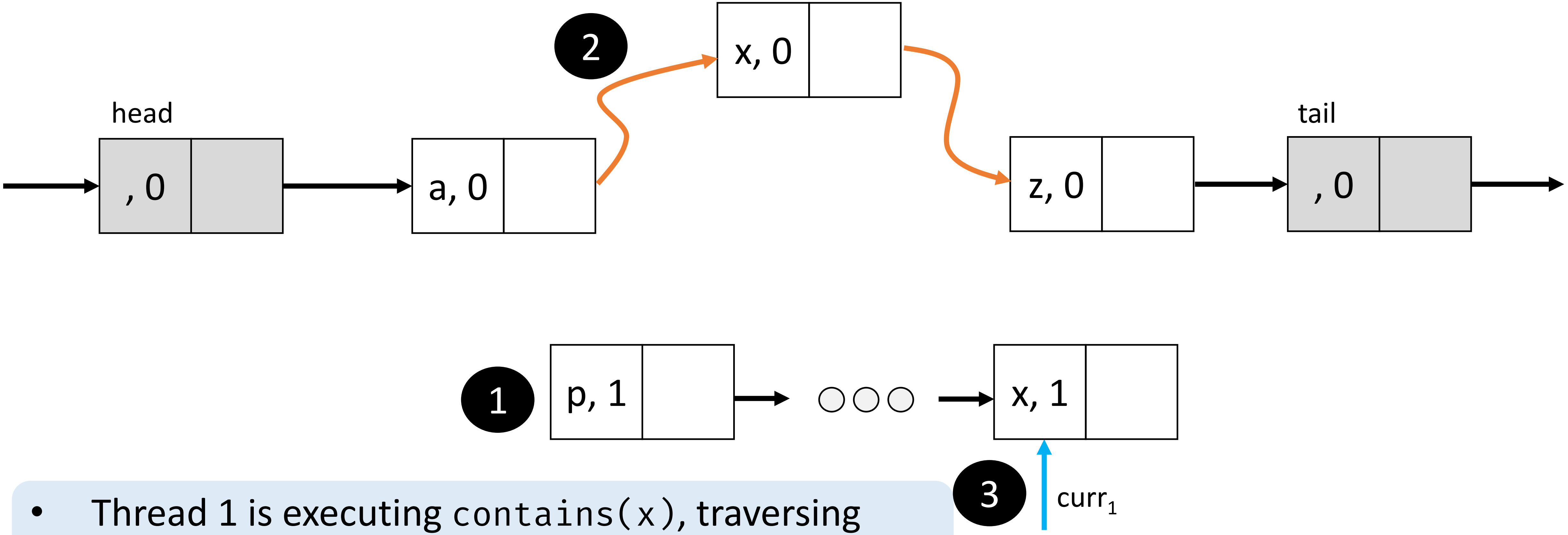
Unsuccessful contains()



- Thread 1 is executing `contains(x)`
- Thread 2 executes `remove(p..x)`

Thread 1's `contain(x)` can be linearized when it sees that x is marked and is no longer in the abstract set

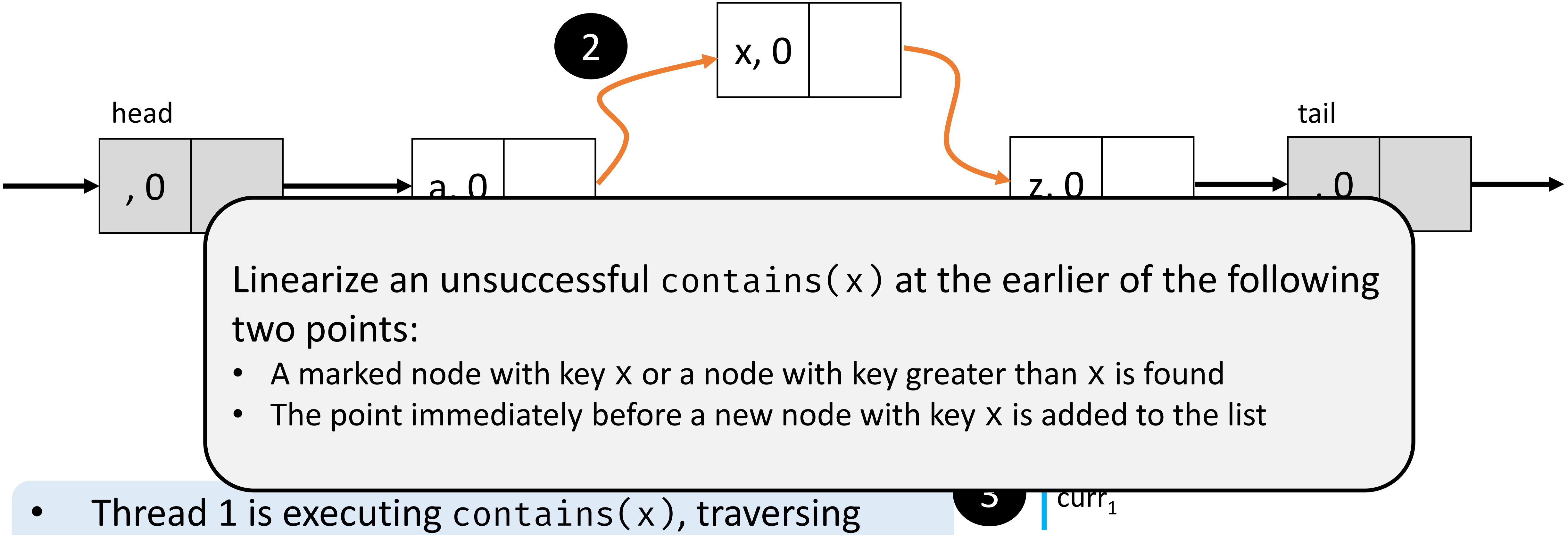
Unsuccessful `contains()`



- Thread 1 is executing `contains(x)`, traversing along the marked portion of the list (p…x)

- Thread 2 is executing `add(x)`

Unsuccessful contains()



Linearize an unsuccessful `contains(x)` at the earlier of the following two points:

- A marked node with key x or a node with key greater than x is found
- The point immediately before a new node with key x is added to the list

- Thread 1 is executing `contains(x)`, traversing along the marked portion of the list (p…x)

- Thread 2 is executing `add(x)`

Nonblocking Synchronization

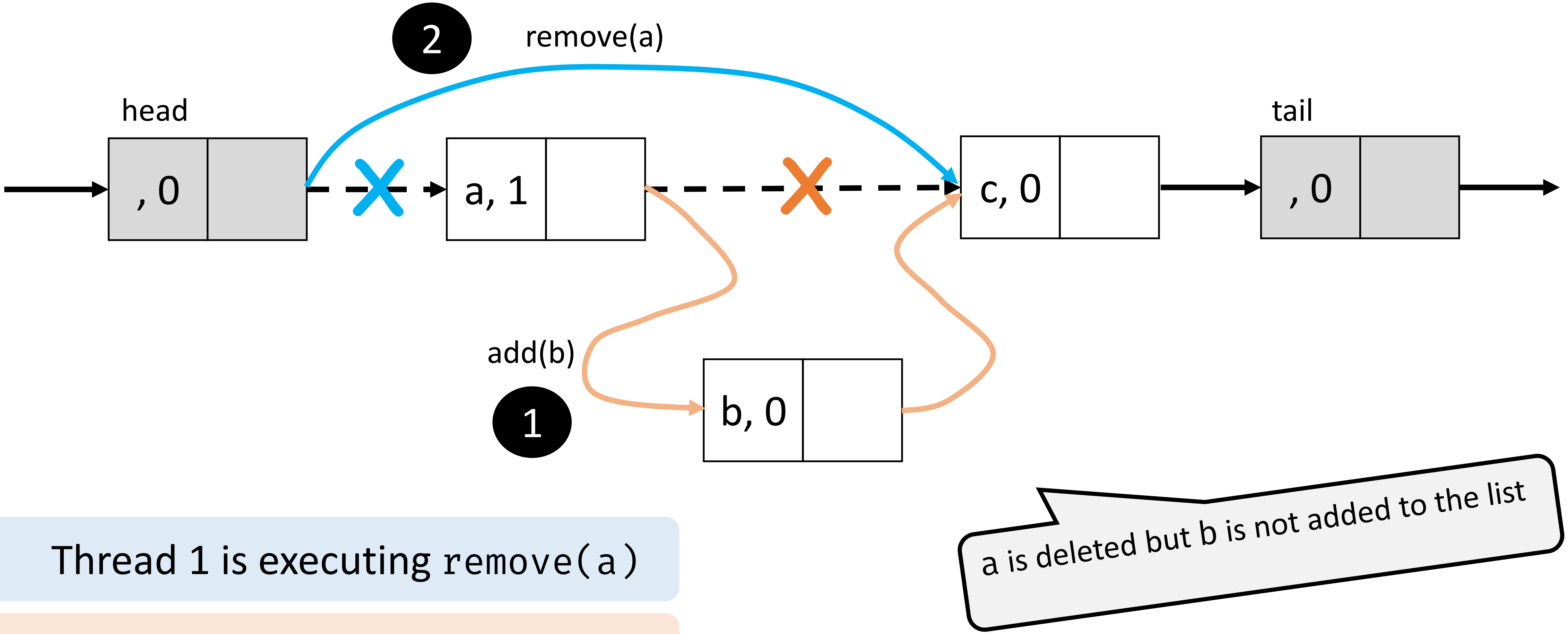
- Why do we need nonblocking designs?
  - Blocked threads do not do useful work, problematic for high-priority or real-time applications
  - Getting the right degree of concurrency and correctness with locks is challenging
  - Use of locks can lead to deadlocks, livelocks, and priority inversion

- **Idea**: Use RMW instructions like CAS to update `next` field
  - Eliminate locks altogether

Nonblocking Algorithms

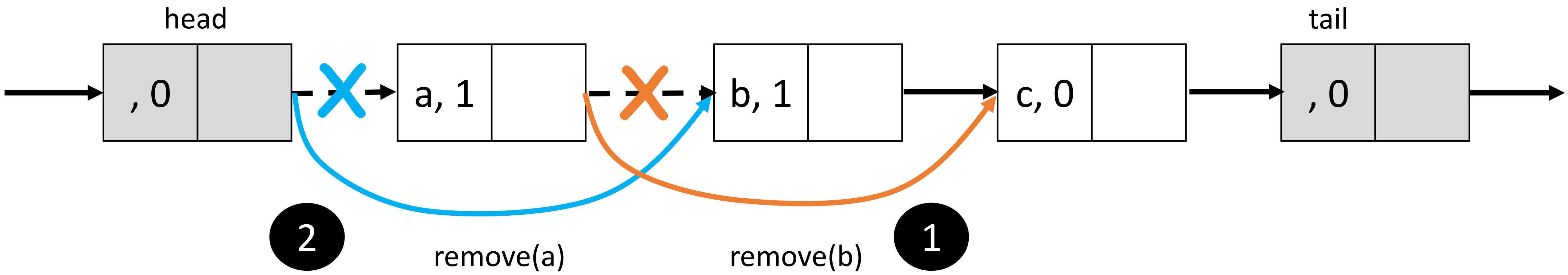
- Failure or suspension of a thread does not impact other threads
- Guaranteed system-wide progress implies lock-freedom, while per-thread progress implies wait-freedom
- Wait-freedom is the strongest nonblocking progress guarantee
  - Lock-freedom allows an individual thread to starve
  - All wait-free algorithms are lock-free
- Lock-free implies “locking up” the application in some way (e.g., deadlock, livelock)
  - Lock-free does not only imply absence of synchronization locks

Nonblocking Synchronization with CAS



- Thread 1 is executing `remove(a)`
- Thread 2 is executing `add(b)`

Nonblocking Synchronization with CAS



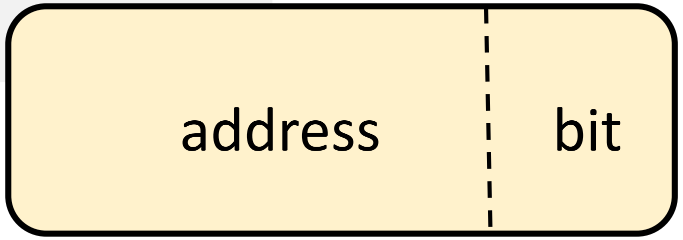
a is deleted but b is not deleted from the list

- Thread 1 is executing `remove(a)`
- Thread 2 is executing `remove(b)`

Possible Workaround

- Cannot allow updates to a node once it has been logically or physically removed from the list
- Treat the `next` and `marked` fields as atomic
  - An attempt to update the `next` field when the `marked` field is true will fail

Java provides the AtomicMarkableReference<T> in the java.util.concurrent.atomic package



AtomicMarkableReference<T>

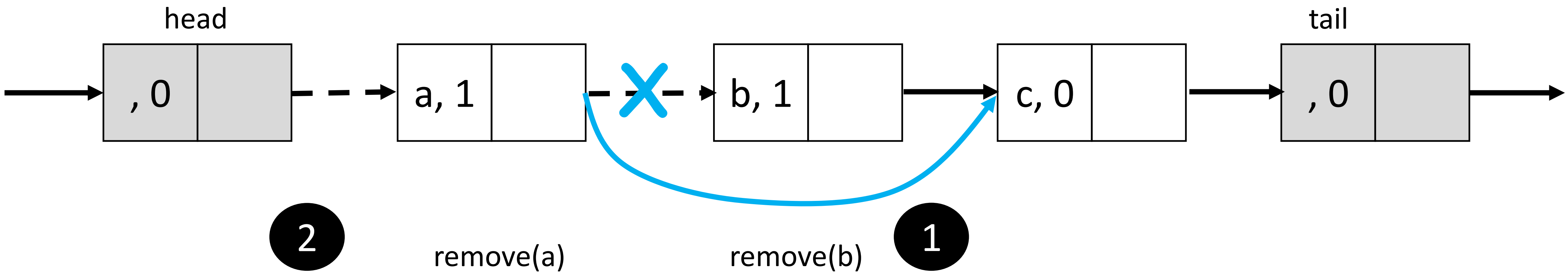
```
public boolean compareAndSet(T expectedReference,
                             T newReference,
                             boolean expectedMark,
                             boolean newMark);

public T get(boolean[] marked);

public T getReference();

public Boolean isMarked();
```

Designing the Nonblocking Set

- The `next` field is of type AtomicMarkableReference<Node>
- A thread logically removes a node by setting the `marked` bit in the `next` field
- As threads traverse the list, they clean up the list by physically removing marked nodes
- Threads performing `add()` and `remove()` do not traverse marked nodes, they **remove them before** continuing

Why?

Challenge in traversing marked nodes



Thread 1 does not delete the marked node a => Thread 1 cannot redirect `a.next`

- Thread 1 is executing `remove(b)`
- Thread 2 marks `a`

Helper Code

- Helper method `public Window find(Node head, int key)`
  - Traverses the list seeking to set `pred` to the node with the largest key less than `key`, and `curr` to the node with the least key greater than or equal to `key`

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

Helper Code

```
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]) {
        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;
    }
  }
}
```

Nonblocking Synchronization: add( )

```
public boolean add(T x) {
  int key = x.hashcode();
  while (true) {
    Window w = find(head, key);
    Node pred = w.pred, curr = w.curr;
    if (curr.key == key) return false;
    else {
      Node node = new Node(x);
      node.next = new AtomicMarkableReference(curr, false);
      if (pred.next.compareAndSet(curr, node, false, false))
        return true;
    } } }
```

Nonblocking Synchronization: `remove()`

```
public boolean remove(T x) {
  int key = x.hashcode();
  boolean snip;
  while (true) {
    Window w = find(head, key);
    Node pred = w.pred, curr = w.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;
} } }
```

Nonblocking Synchronization: `contains()`

```
public boolean contains(T x) {
  int key = x.hashcode();
  Node curr = head;
  while (curr.key < key) {
    curr = curr.next.getReference();
  }
  return curr.key == key && !curr.next.isMarked();
}
```

Pool Data Structure

Pools

Allows duplicates

May not support membership test (i.e., no `contains()` method)

Examples: stack, queue, bounded/unbounded buffers

Data Structure Variants

- Bounded vs Unbounded
  - Different requirements and implementation challenges

```
public interface Pool<T> {
  void put(T item);
  T get();
}
```

- Different method call invocation semantics
  - Blocking vs nonblocking
  - Synchronous vs asynchronous
  - Total vs partial

Method Call Semantics

Synchronous and asynchronous

- A synchronous call waits for a concurrent action to hold before returning
- An asynchronous call requests to start the computation and returns to potentially execute other operations

Blocking and nonblocking

- Blocking call waits for an event to hold and puts the caller thread on a wait queue
- Nonblocking call does not put the thread on the wait queue, it returns even if the result is unavailable

Total and partial

- A method is total if it is defined for every object state, i.e., it does not need to wait for certain conditions to become true
- A partial method is not defined for every object state, it may have to block for certain conditions to hold

Method Call Semantics

A blocking call is always synchronous

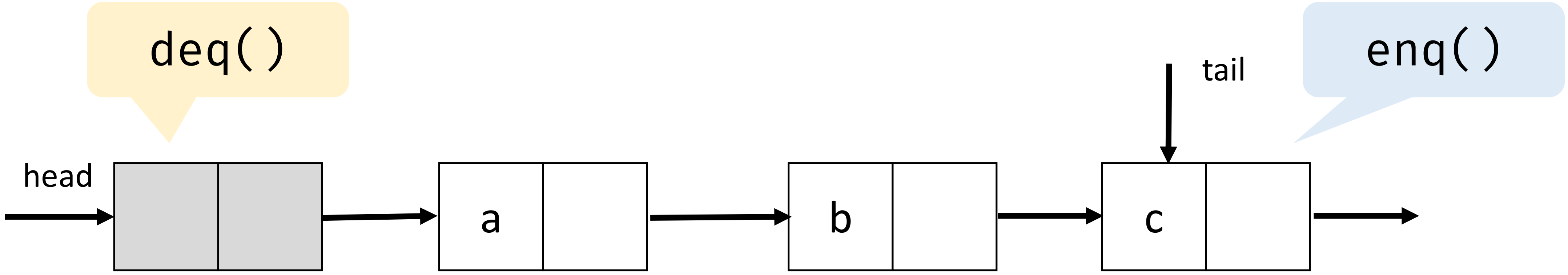
Nonblocking calls can be either wait-free or lock-free

- In the nonblocking design for Set, `add()` and `remove()` are lock-free and `contains()` is wait-free

Synchronous calls can be either blocking or nonblocking (e.g., busy wait or spinning)

Partial methods can be either synchronous or asynchronous

Bounded Partial Queue



Enqueue and dequeue operations are at the two ends – allows for concurrent modifications

Bounded Partial Queue

- Given these requirements, what do we need to have a correct concurrent implementation?

Bounded Partial Queue

- Given these requirements, what do we need to have a correct concurrent implementation?

- Lock for mutual exclusion of enqueues and dequeues?

Bounded Partial Queue

- Given these requirements, what do we need to have a correct concurrent implementation?

Possible Java classes we can use
- `ReentrantLock`

- Lock for mutual exclusion of concurrent enqueues
- Lock for mutual exclusion of concurrent dequeues

Bounded Partial Queue

- Given these requirements, what do we need to have a correct concurrent implementation?

Possible Java classes we can use

- `ReentrantLock`
- `Condition`

- Lock for mutual exclusion of concurrent enqueues
- Lock for mutual exclusion of concurrent dequeues
- Condition variable to indicate queue is empty
- Condition variable to indicate queue is full

Bounded Partial Queue

- Given these requirements, what do we need to have a correct concurrent implementation?

Possible Java classes we can use
- `ReentrantLock`
- `Condition`
- `AtomicInteger`

- Lock for mutual exclusion of concurrent enqueues
- Lock for mutual exclusion of concurrent dequeues
- Condition variable to indicate queue is empty
- Condition variable to indicate queue is full
- Atomic variable to track the current size

Bounded Partial Queue: enq()

```
public void enq(T x) {
  boolean wakeDeq = false;
  Node e = new Node(x);
  enqLock.lock();
  try {
    while (size.get() == MAX_CAPACITY)
      notFull.await();
    tail.next = e; // Add the new node
    tail = e; // Update the tail pointer
    if (size.getAndIncrement() == 0)
      wakeDeq = true;
  } finally {
    enqLock.unlock();
  }
```

```
  if (wakeDeq) {
    deqLock.lock();
    try {
      notEmpty.signalAll();
    } finally {
      deqLock.unlock();
    }
  } // end if (wakeDeq)
} // end enq()
```

Bounded Partial Queue: enq()

```
public void enq(T x) {
  boolean wakeDeq = false;
  Node e = new Node(x);
  enqLock.lock();
  try {
    while (size.get() == MAX_CAPACITY)
      notFull.await();
    tail.next = e;
    tail = e;
    if (size.getAndIncrement() == 0)
      wakeDeq = true;
  } finally {
    enqLock.unlock();
  }
  if (wakeDeq) {
    deqLock.lock();
    try {
      notEmpty.signalAll();
    } finally {
      deqLock.unlock();
    }
  } // end if (wakeDeq)
} // end enq()
```

Where is the linearization point?

Bounded Partial Queue: enq()

```
public void enq(T x) {
  boolean wakeDeq = false;
  Node e = new Node(x);
  enqLock.lock();
  try {
    while (size.get() == MAX_CAPACITY)
      notFull.await();
    tail.next = e;
    tail = e;
    if (size.getAndIncrement() == 0)
      wakeDeq = true;
  } finally {
    enqLock.unlock();
  }
  if (wakeDeq) {
    deqLock.lock();
    try {
      notEmpty.signalAll();
    } finally {
      deqLock.unlock();
    }
  } // end if (wakeDeq)
} // end enq()
```

Where is the linearization point if the queue was unbounded and the methods are total?

Bounded Partial Queue: deq( )

```
public void deq() {
  boolean wakeEnq = false;
  T result;
  deqLock.lock();
  try {
    while (head.next == null)
      notEmpty.await();
    result = head.next.value;
    head = head.next;
    if (size.getAndDecrement() == MAX_CAPACITY)
      wakeEnq = true;
  } finally {
    deqLock.unlock();
  }
  if (wakeEnq) {
    enqLock.lock();
    try {
      notFull.signalAll();
    } finally {
      enqLock.unlock();
    }
  }
  return result;
}
```

Evaluating the Bounded Partial Queue

- Need to ensure correct interleaving of concurrent calls to `enq()` and `deq()`
  - Special cases: Queue has zero or one element (`size` can become negative temporarily)
- Shared updates to the `size` variable could be a bottleneck
  - Can we do something about it?

Unbounded Total Queue

- `enq()` always enqueues an item
  - It may run in to OOM error which we will ignore
- `deq()` returns an error if the queue is empty
- Simpler conditions, no need for condition variables and no need to track the size

Unbounded Total Queue

```
public void enq(T x) {
  Node e = new Node(x);
  enqLock.lock();
  try {
    tail.next = e;
    tail = e;
  } finally {
    enqLock.unlock();
  }
}
```

```
public T deq() {
  T result;
  deqLock.lock();
  try {
    if (head.next == null)
      return null;
    result = head.next.value;
    head = head.next;
  } finally {
    deqLock.unlock();
  }
  return result;
}
```

Unbounded Total Queue

```
public void enq(T x) {
  Node e = new Node(x);
  enqLock.lock();
  try {
    tail.next = e;
    tail = e;
  } finally {
    enqLock.unlock();
  }
}
```

```
public T deq() {
  T result;
  deqLock.lock();
  try {
    if (head.next == null)
      return null;
    result = head.next.value;
    head = head.next;
  } finally {
    deqLock.unlock();
  }
  return result;
}
```

Can these methods deadlock?

A Natural Next Step!

- Unbounded lock-free queue

Possible Java classes we can use

- `AtomicReference<T>`

```
public class Node {
  public T val;
  public AtomicReference<Node> next;
  public Node(T value) {
    this.val = val;
    next = AtomicReference<Node>(null);
  }
}
```

```
public class LockFreeQueue<T> {
  AtomicReference<Node> head, tail;
  public LockFreeQueue() {
    Node node = new Node(null);
    head = new AtomicReference(node);
    end = new AtomicReference(node);
  }
  …
}
```

Unbounded Lock-Free Queue: enq( )

```
public void enq(T x) {
  Node node = new Node(x);
  while (true) {
    Node last = tail.get();
    Node next = last.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        }
      }
      } else {
        // Indicates a concurrent enqueuer
        // tail not yet updated
        tail.compareAndSet(last, next);
        // Retry
      }
    } // end if (last == …
  } // end while (true)
} // end enq()
```

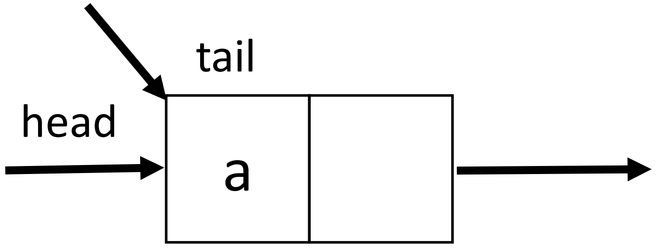
This call can fail!

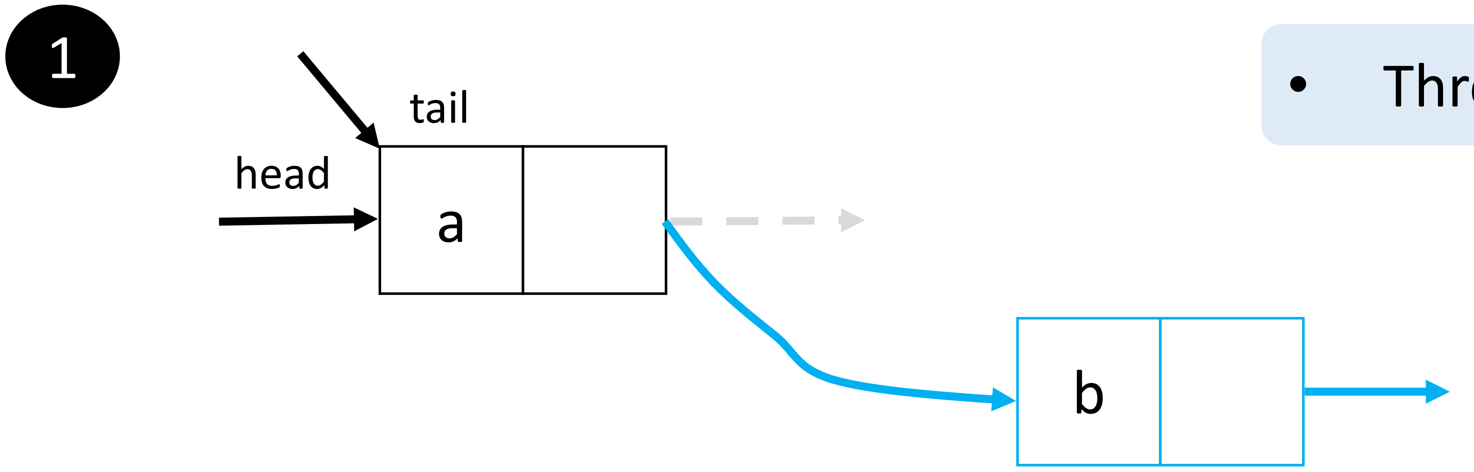
Unbounded Lock-Free Queue: enq()

```
public void enq(T x) {
  Node node = new Node(x);
  while (true) {
    Node last = tail.get();
    Node next = last.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        }
      }
      } else {
        // Indicates a concurrent enqueuer
        // tail not yet updated
        tail.compareAndSet(last, next);
        // Retry
      }
    } // end if (last == …
  } // end while (true)
} // end enq()
```

Where is the linearization point?

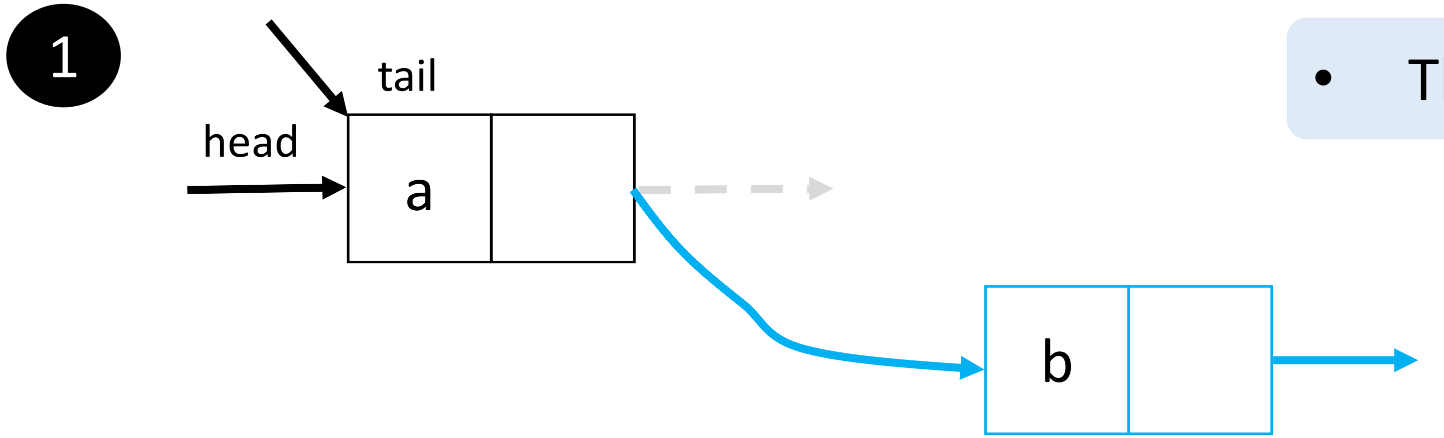
Ensure that `tail` remains valid!



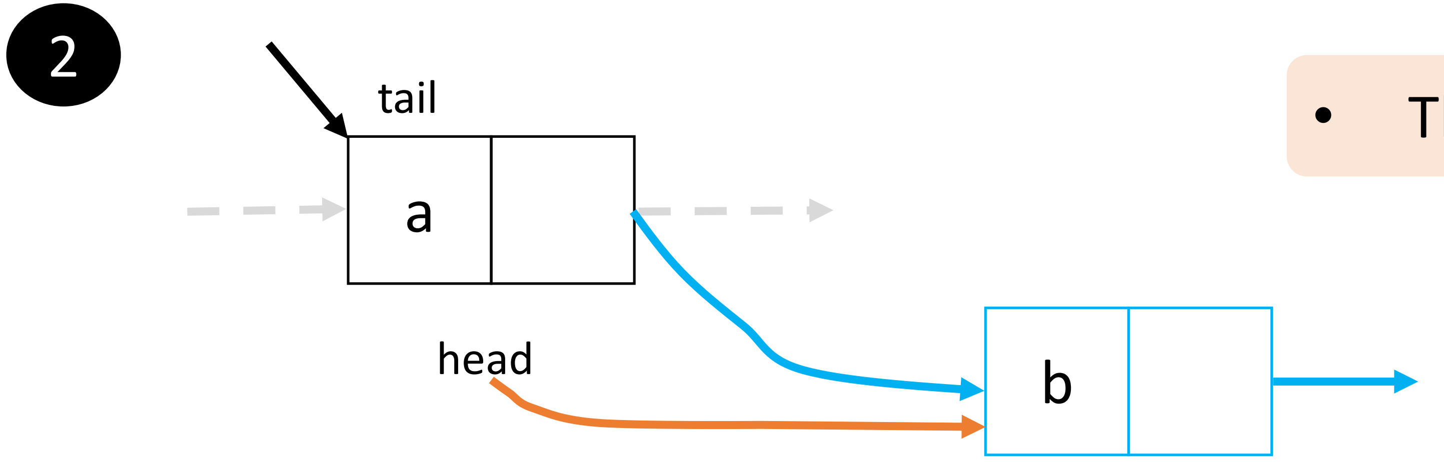


- Thread 1 is executing `enq(b)`

Ensure that `tail` remains valid!



- Thread 1 is executing `enq(b)`



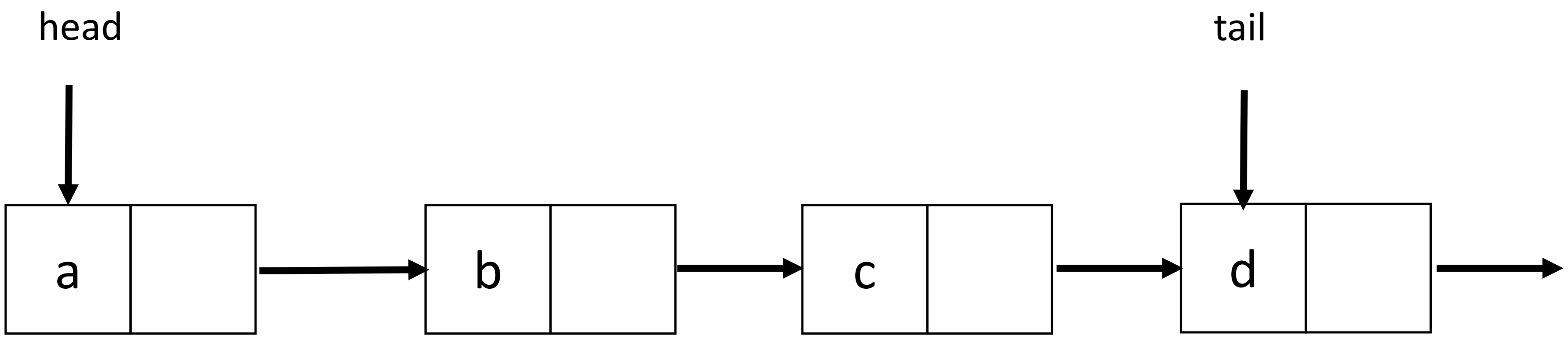
- Thread 2 is executing `deq(a)`

Unbounded Lock-Free Queue: `deq()`

```
public void deq(void) throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null)
          throw new EmptyException();
       // tail is lagging head
        tail.compareAndSet(last, next);
      } else {
        T val = next.value;
        if (head.compareAndSet(first, next))
          return val;
      } // end else
    } // end if (first == head…)
  } // end while (true)
} // end deq()
```

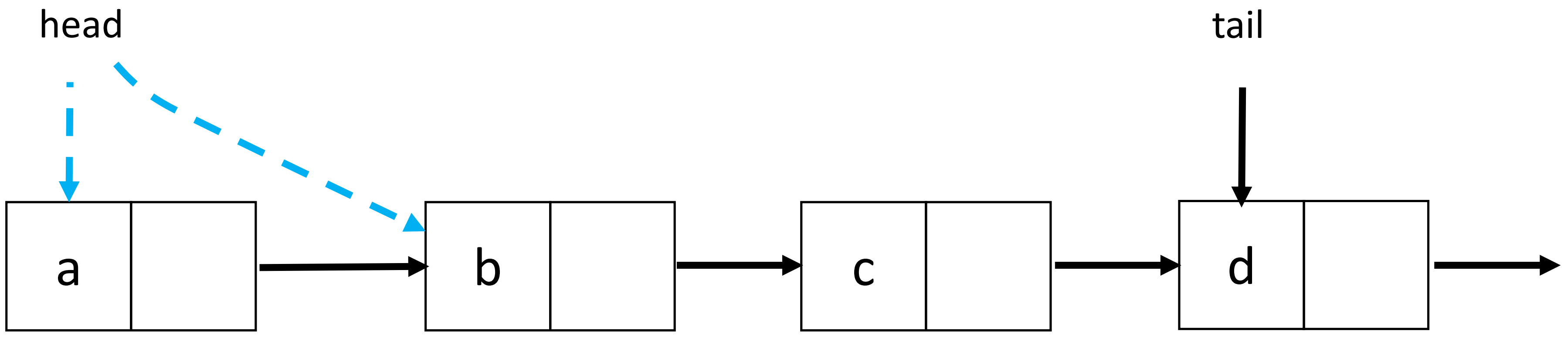
Lock-Free Programming and ABA Problem

The system recycles old nodes



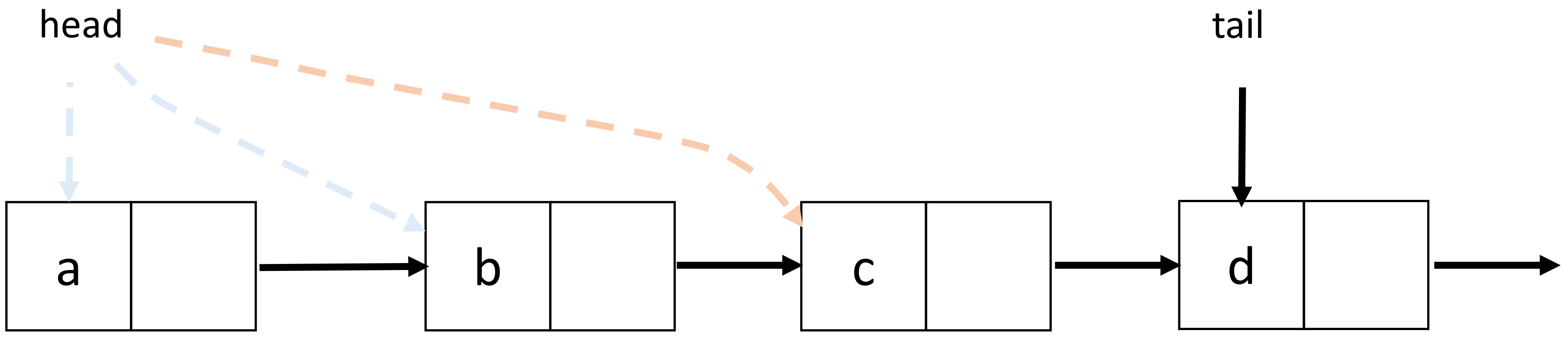
- Thread 1 will execute `deq(a)`

Lock-Free Programming and ABA Problem



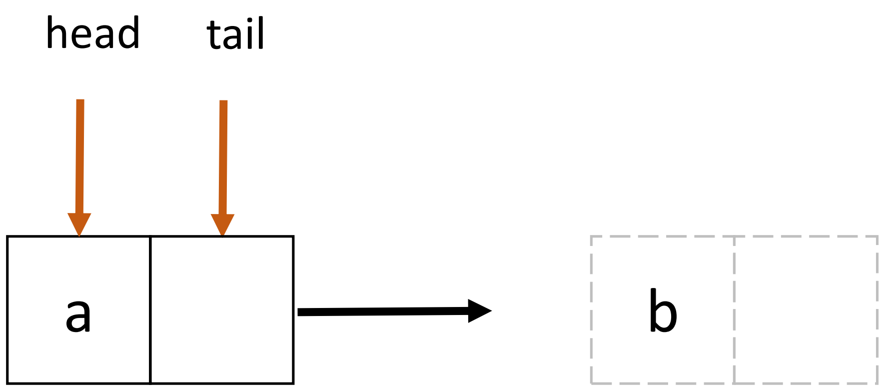
- Thread 1 is executing `deq(a)`, gets delayed

Lock-Free Programming and ABA Problem



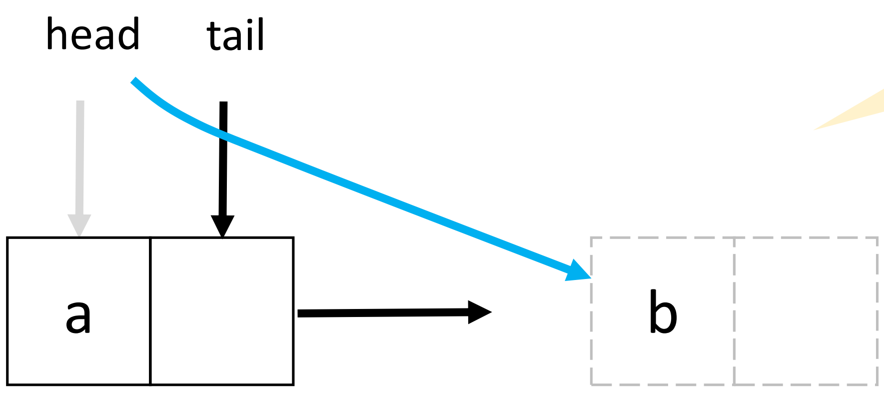
- Other threads execute `deq(a, b, c, d)`, then execute `enq(a)`

Lock-Free Programming and ABA Problem



- Other threads execute `deq(a, b, c, d)`, then execute `enq(a)`

Lock-Free Programming and ABA Problem



`head.compareAndSet(first, next)`

- Thread 1 is executes CAS for `deq(a)`, CAS succeeds

To Lock or Not to Lock!

Use a middle path more often than not

- Combine blocking and nonblocking schemes
- For e.g., lazily synchronized Set
  - `add()` and `remove()` were blocking, `contains()` was nonblocking

Spend several hours reasoning about the correctness of your concurrent data structures, if you are writing one!

References

- M. Herlihy and N. Shavit – The Art of Multiprocessor Programming, Chapters 9 and 10.
- M. Moir and N. Shavit – Concurrent Data Structures.
- Stephen Tu – Techniques for Implementing Concurrent Data Structures on Modern Multicore Machines.