CS 610: Concurrent Data Structures

CPUs and GPUs

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2024-25-I

Challenges with Concurrent Programming

Less synchronization

More synchronization

Order, atomicity, and sequential consistency violations

Concurrent and correct

Deadlock

Poor performance: lock contention, serialization

Challenges with Concurrent Programming

Less synchronization

More synchronization

Order,
sequentia
viola

dlock

ormance:
ck contention,
serialization

Languages and libraries should provide efficient portable data structures as building blocks

Designing a Concurrent Set Data Structure

Designing a Concurrent Set

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

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)

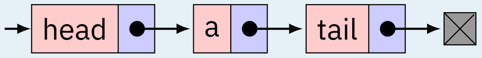
returns true if and only if x is present in the set

There are significantly more calls to `contains()` than `add()` and `remove()`

Designing a Concurrent Set Using Linked Lists

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

Two sentinel nodes head and `tail`



Invariants

- Field key is `data`'s hash code to help with efficient search
- Nodes are sorted based on the key value
- Assume that all hash codes are unique
- Sentinel nodes are immutable, and `tail` is reachable from head
- Removed nodes continue to represent valid memory locations

A Thread-Unsafe Set Data Structure

```java
public class UnsafeList<T> {
  private Node head;
  public UnsafeList() {
    head = new Node(Integer.MIN_VALUE);
    head.next = new Node(Integer.
        MAX_VALUE);
  }
```

```java
public boolean add(T x) {
  int key = x.hashcode();
  Node pred = head;
  Node 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

```
public boolean remove(T x) {
  int key = x.hashcode();
  Node pred = head;
  Node 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;
  }
}
```

```
public boolean contains(T x) {
  int key = x.hashcode();
  Node pred = head;
  Node 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

```
public boolean remove(T x) {
  int key = x.hashcode();
  Node pred = head;
  Node curr = pred
  while (cu
    pred
    cur
  }
  if (k
    pred.
    return
  } else {
    return false;
  }
}
```

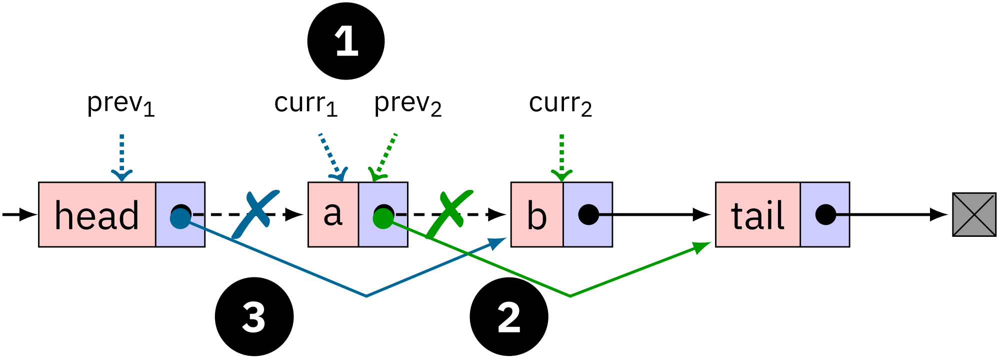
```
public boolean contains(T x) {
  int key = x.hashcode();
  No    ed = head;
        = pred.next;
              key) {


                  {


              e;

  }
}
```

Can you given an example to show that remove() is not thread-safe?

Thread-Unsafe Set: Incorrect `remove()`

Thread 1 is executing `remove(a)`

Thread 2 is executing `remove(b)`



Concurrent Set with Coarse-Grained Synchronization

```java
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);
  }
  ...

  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();
    }
  }
```

Concurrent Set with Coarse-Grained Synchronization

```java
  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();
    }
  }
```

```java
  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;
  }
}
```

Concurrent Set with Fine-Grained Synchronization

Add a lock object to each list node

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

Possible interleaving

Thread 1

```
curr.lock.lock();
next = curr.next;
curr.lock.unlock();

next.lock.lock();
```

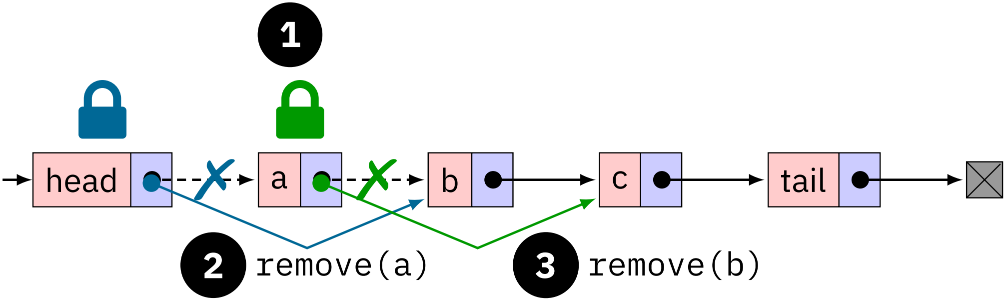
Thread 2

```



// Remove next from list

```

Is Locking One Node Sufficient?

Thread 1 is executing `remove(a)`

Thread 2 is executing `remove(b)`



Concurrent Set with Fine-Grained Synchronization

```java
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();
  }
}
```

Concurrent Set with Fine-Grained Synchronization

```java
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();
  }
}
```

Challenges With Fine-Grained Synchronization

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 predetermined order

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

Challenges With Fine-Grained Synchronization

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 predetermined order

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

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

Evaluating Concurrent Data Structures

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

Reduce lock contention

Lock implementations with single memory location can introduce additional coherence and memory traffic due to unsuccessful acquires

Blocking or nonblocking implementations

Blocking: Delay of any one thread can delay other threads

Nonblocking: Delay of one thread cannot delay other threads

Reasoning about Correctness of Sequential Data Structures

Need to describe how an object's methods behave

- Possibilities include formal specification and API documentation
- Pre-condition describes the object's state before the method call
  - Operations on objects are not instantaneous. Each operation requires an invocation on that object, followed by a response.
  - Method call is the duration between an invocation event and a response event
- Post-condition describes the object's state and return value after the method call

Example

Suppose the state of a queue q is a sequence of items Q (i.e., precondition). Then, a call to `q.enq(z)` changes the state of the queue to Q•z, where • denotes concatenation.

Reasoning about Correctness of 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)`

Using pre- and post-conditions no longer work. How do you reason about the outcome?

Correctness for interleaved operations on concurrent objects is determined by some notion of equivalence with sequential behavior

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

Reasoning about Correctness of Concurrent Data Structures

- Identify invariants and make sure they always hold
  - For example, an item is in the set if and only if it is reachable from head
- Correctness (or safety) property is **linearizability**
- Progress (or liveness) properties are starvation and deadlock-freedom

Definitions

Program order The order in which a single thread issues method calls is called its program order.

- Method calls by different threads are unrelated by program order.

Total method 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 total method is used when the caller thread has something useful to do than wait for certain conditions to be met.

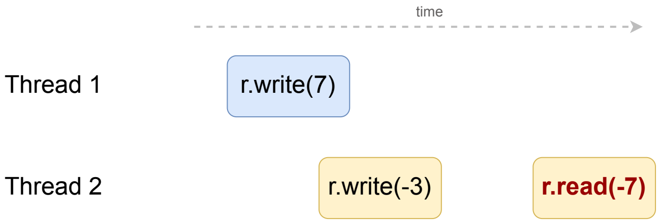
Partial method A partial method is not defined for every object state, it may have to block for certain conditions to hold.

- For example, a partial `Queue::get()` call that tries to remove an item from an empty queue blocks until an item is available to return.

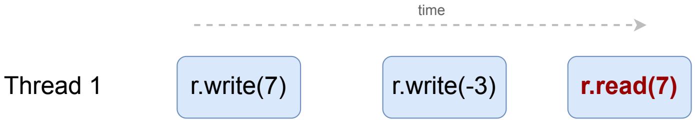
Compositional A correctness property P is compositional if, whenever each object in the system satisfies P, the system as a whole satisfies P.

Correct Behavior Expected From a Concurrent Execution

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



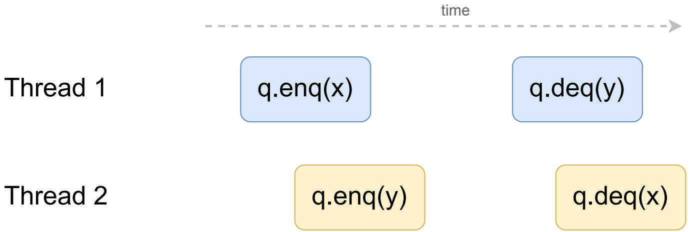
(ii) Method calls should appear to take effect in program order



Sequentially Consistent Execution

An execution is sequentially consistent (SC) if the result is the same as if the operations from all threads were executed in some sequential order, and the operations of each individual thread appear in program order.

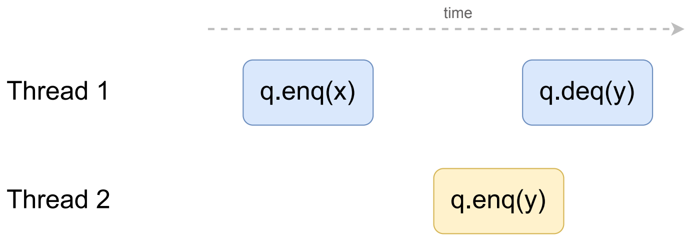
Consider the following operations on a FIFO queue q, where x and y are objects.



There are two possible sequential orders that can justify the above execution

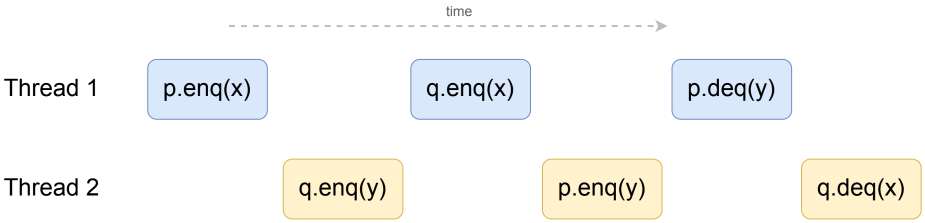
SC can Violate Real-Time Order

Reordering method calls unrelated by program order is allowed in SC, and so can violate real-time order



SC is Not Composable

p and q are each sequentially consistent, but the execution as a whole is not



Linearizability

Linearizability has two requirements

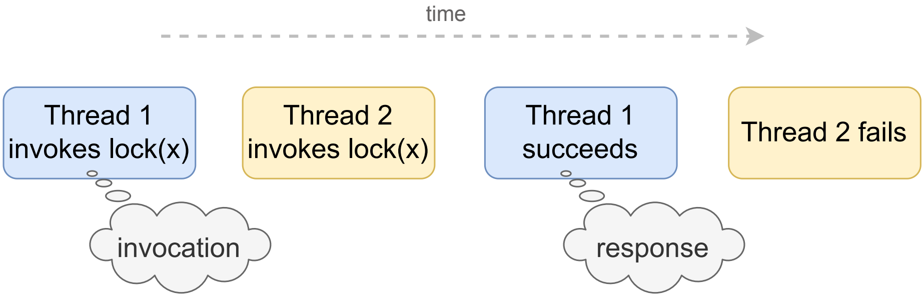
(i) Method calls should appear to happen one-at-a-time in sequential order

(ii) Each method call should appear to take effect instantaneously at some moment between its invocation and response

- Linearization point represents a single atomic step where the method call "takes effect"
  - For coarse-grained lock-based implementations, each method's critical section is its linearization point
  - For implementations that do not use locking, the linearization point is a single step where the effects of the method call become visible to others

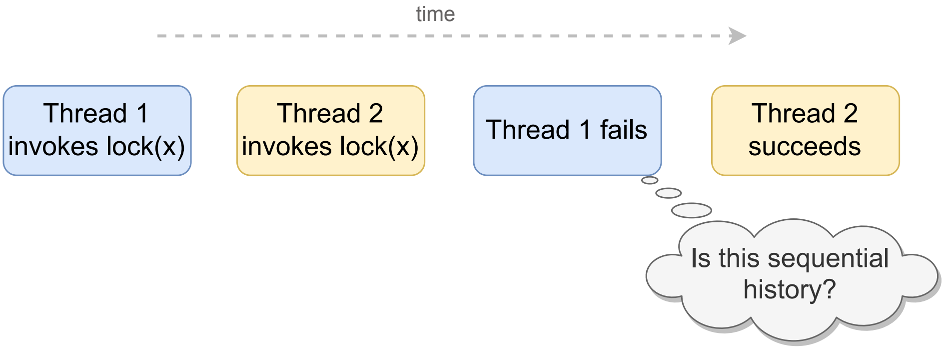
Understanding Linearizability

- Say you perform some operations on an object (e.g., a method call)
- A history is a sequence of invocations and responses on an object made by concurrent threads



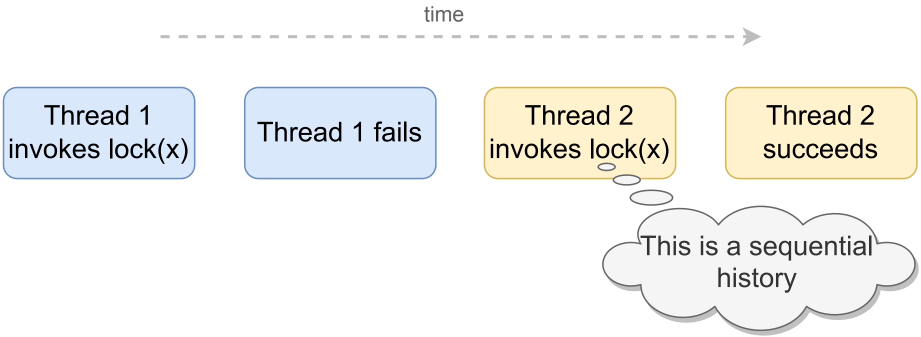
Sequential History

- 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



Sequential History

- 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

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

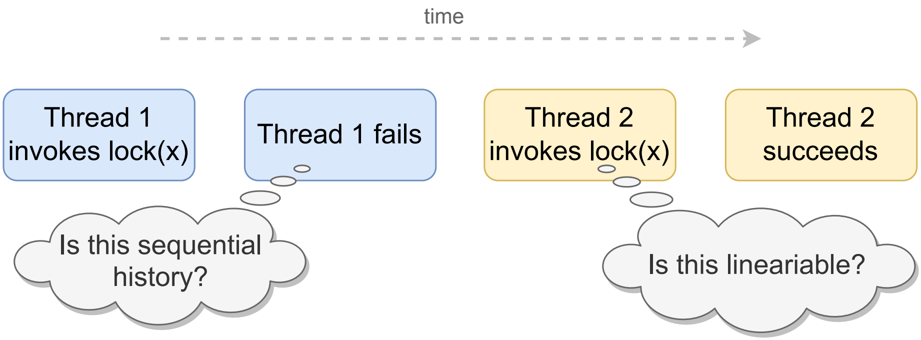
Consider a concurrent history (set of method calls) H and a valid sequential history S. The history H is linearizable if:

- 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

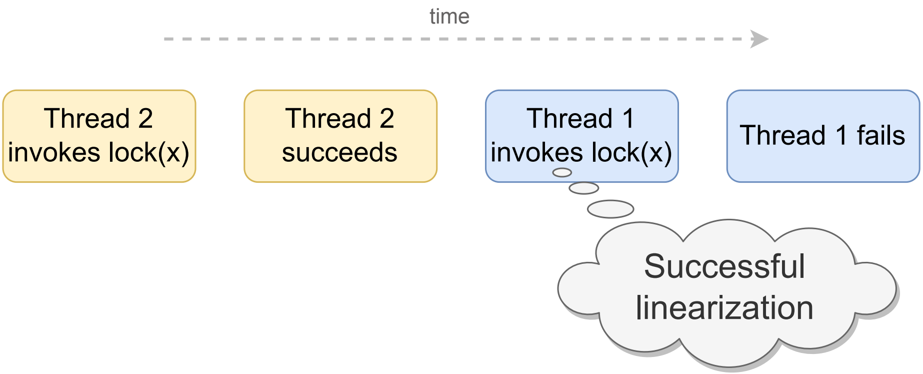
Linearizability in Simpler Words

- Sequential history is correct according to the semantics of the object
- Invocations and responses can be reordered to form a sequential history
- If a response preceded an invocation in the original history, it must still precede it in the sequential reordering

Understanding Linearizability



Understanding Linearizability



Identifying Linearization Points

Linearization point represents a single atomic step where the method call "takes effect", and is between the function invocation and response

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

Sequential Consistency vs Linearizability

Sequential Consistency

- Method calls appear to happen instantaneously in some sequential order
- A sequentially consistent history is not necessarily linearizable
- Nonblocking but not composable

Linearizability

- Method calls appear to happen instantaneously at some point between its invocation and response
- Every linearizable history is sequentially consistent
- Nonblocking and composable

Progress Guarantees

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

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

Designing a Concurrent Set Data Structure

How to Design a Concurrent Set?

Coarse-grained synchronization

Easy to get right, low concurrency, not scalable

Fine-grained synchronization

More concurrent and scalable than coarse-grained synchronization, difficult to get right

Optimistic synchronization

Avoid synchronization to search, good for low contention cases

Lazy synchronization

Defer expensive data structure manipulation operations

Nonblocking synchronization

Rely on atomic operations such as `compareAndSet()` for synchronization

Optimistic Synchronization

Optimistic strategy

- Access data without acquiring a lock
- Lock only when required, and validate that the condition before locking is still valid
- If valid, then continue with access/mutation
- If invalid, restart by locking again

Optimistic strategy works well if conflicts are rare

Concurrent Set with Optimistic Synchronization

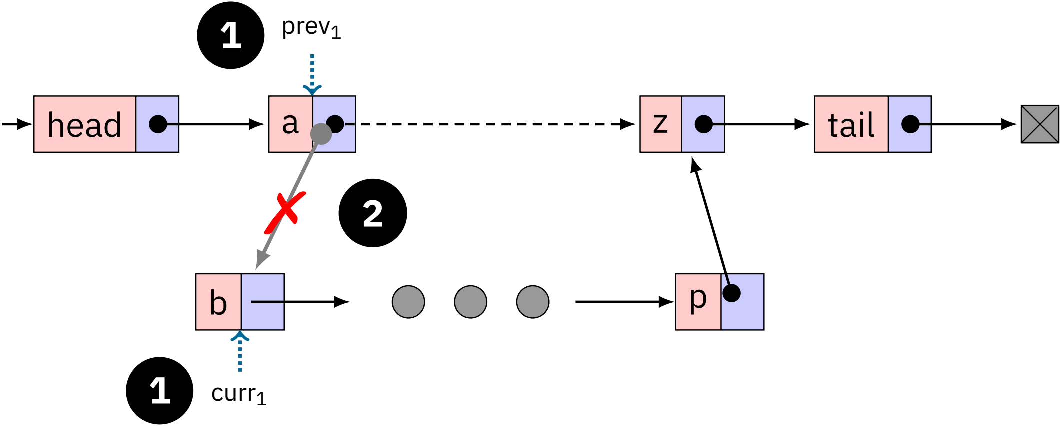
```
public boolean add(T x) {
  int key = x.hashcode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    pred.lock(); curr.lock();
```

```
      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();
      }
    }
}
```

Is Validation Necessary?

Thread 1 is executing `remove(p)`

Other threads execute `remove(b-p)`



How to 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;
}
```

Concurrent Set with Optimistic Synchronization

```java
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();
      }
    }
}
```

Concurrent Set with Optimistic Synchronization

```
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();
      }
    }
  }
```

Concurrent Set with Optimistic Synchronization

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

- Validation can be costly (e.g., need to traverse the list again)
- Needs lock operations for `contains()` which is the most frequent method (bad design)

Lazy Synchronization

Delay mutation operations for a later time

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

Guarantees

`add()`	traverses the list, locks the predecessor, and inserts the node
`remove()`	marks the target node logically removing it, then redirects the predecessor's `next` link physically removing the target node
`contains()`	needs only one wait-free traversal (no locking is required)

Concurrent Set with Lazy Synchronization

```
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 to 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;
}
```

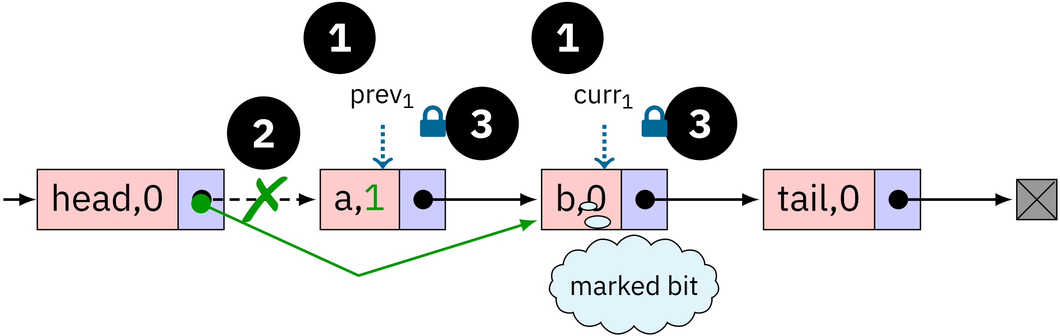
Concurrent Set with Lazy Synchronization

```
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 {
            // Logical deletion
            curr.marked = true;
            // Physical deletion
            pred.next = curr.next;
            return true;
          }
        }
      } finally {
        curr.unlock(); }
      }
    } finally {
      pred.unlock();
    }
  }
}
```

Detecting Conflicts: Scenario 1

Thread 1 is executing `remove(b)`

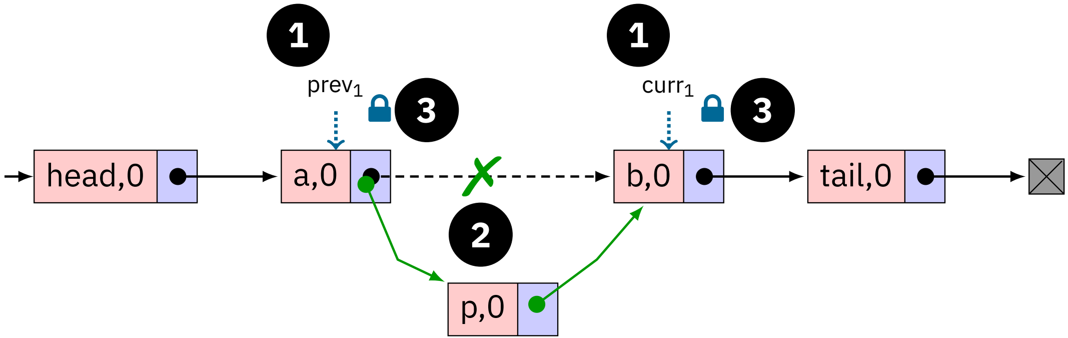
Thread 2 is executing `remove(a)`



Detecting Conflicts: Scenario 2

Thread 1 is executing `remove(b)`

Thread 2 is executing `add(p)`



Concurrent Set with Lazy Synchronization

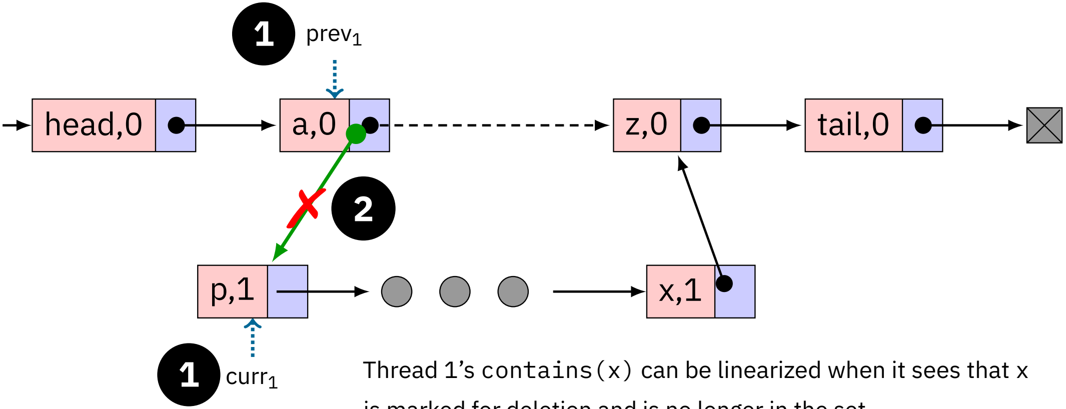
```java
public boolean contains(T x) {
  int key = x.hashcode();
  Node curr = head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

wait-free

Unsuccessful `contains()`: Scenario 1

Thread 1 is executing `contains(x)`

Thread 2 is executing `remove(p...x)`

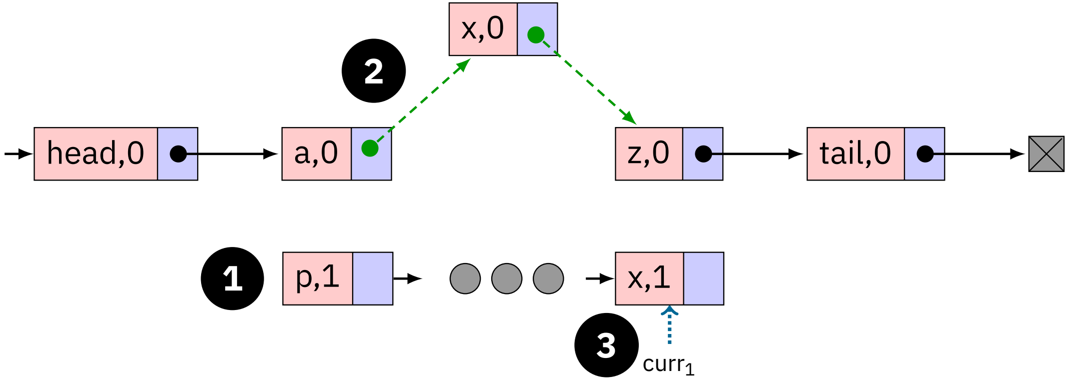


Thread 1's `contains(x)` can be linearized when it sees that x is marked for deletion and is no longer in the set

Unsuccessful contains(): Scenario 2

Thread 1 is executing contains(x)

Thread 2 is executing add(x)



Thread 1 is traversing along the marked portion of the list p...x

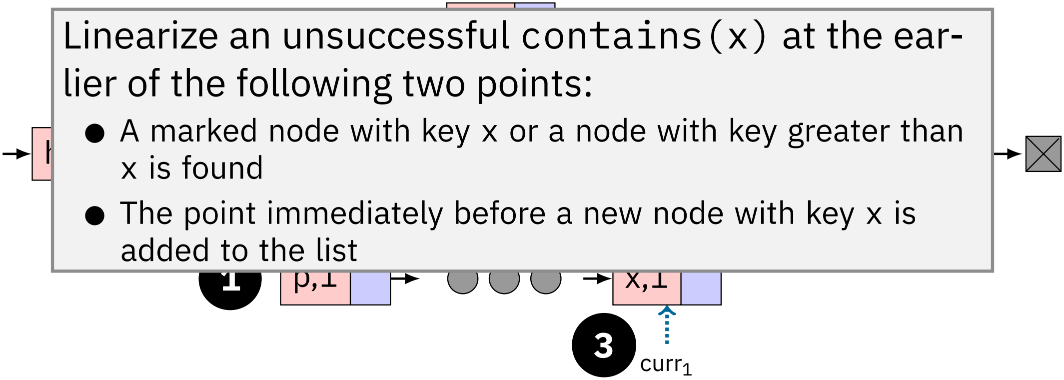
Unsuccessful `contains()`: Scenario 2

Thread 1 is executing `contains(x)`

Thread 2 is executing `add(x)`

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 traversing along the marked portion of the list `p...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 the 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 and livelock)

Lock-free does not **only** imply absence of synchronization locks

Compare-and-Swap (CAS) Primitive

- Modern architectures provide many atomic read-modify-write (RMW) instructions for synchronization
  - For example, test-and-set, fetch-and-add, compare-and-swap, and load-linked/store-conditional

- Compare-and-Swap (CAS) compares the contents of a memory location with a given value and, only if they are the same, updates the contents of that memory location to a new given value

```
bool CAS(word* loc, word oldval, word newval) {
  atomic { // Code block will execute atomically
    res := (*loc == oldval);
    if (res)
      *loc := newval;
    return res;
  }
}
```

Compare-and-Swap (CAS) Primitive

- CAS is implemented as the compare-and-exchange (CMPXCHG) instruction in x86 architectures
  - On a multiprocessor, the LOCK prefix must be used

- CAS is a popular synchronization primitive for implementing both lock-based and nonblocking concurrent data structures

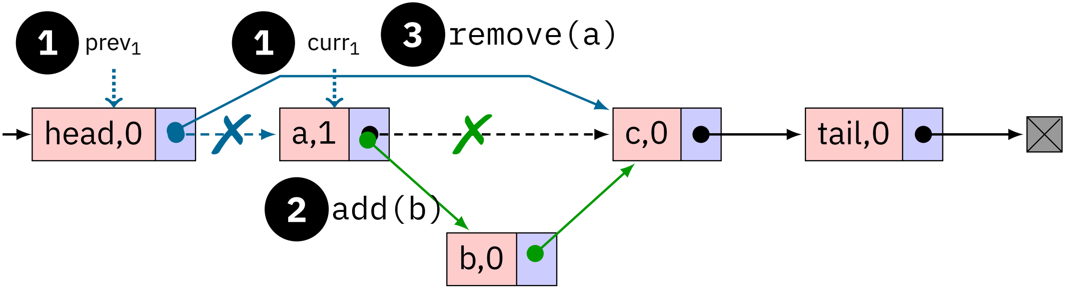
```
        xor %ecx, %ecx /*ecx=0*/
        inc %ecx /*ecx=1*/
RETRY:  xor %eax, %eax /*eax=0*/
        lock compxchg %ecx, &lock
        jnz RETRY
        ret

```

```
void spinLock(lock* lk) {
  // flag attribute is set when the
  // lock is acquired
  while (CAS(&lk->flag, 0, 1) == 1) {
    // Keep spinning
  }
}
```

Nonblocking Synchronization with CAS

Thread 1 is executing `remove(a)`

Thread 2 is executing `add(b)`

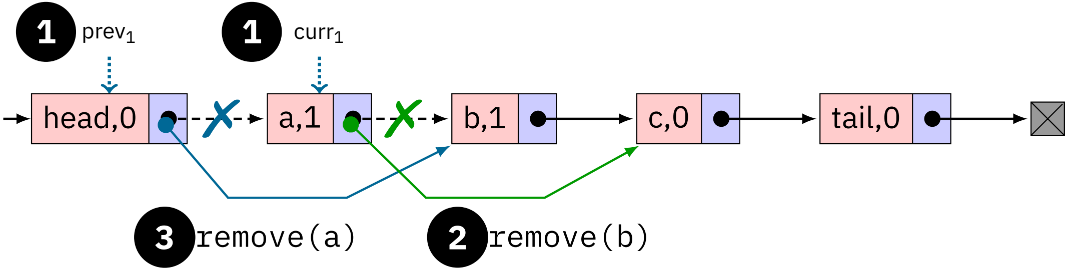


a is deleted but b is not added to the list

Nonblocking Synchronization with CAS

Thread 1 is executing `remove(a)`

Thread 2 is executing `remove(b)`



a is deleted but b is not deleted from the list

Disallow Updates to Deleted Nodes

- 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 Class `AtomicMarkableReference<T>` in the `java.util.concurrent.atomic` package

```java
public boolean compareAndSet(T expectedReference, T newReference, boolean
    expectedMark, boolean newMark);
public T get(boolean[] marked);
public T getReference();
public Boolean isMarked();
```

Designing a Nonblocking Set

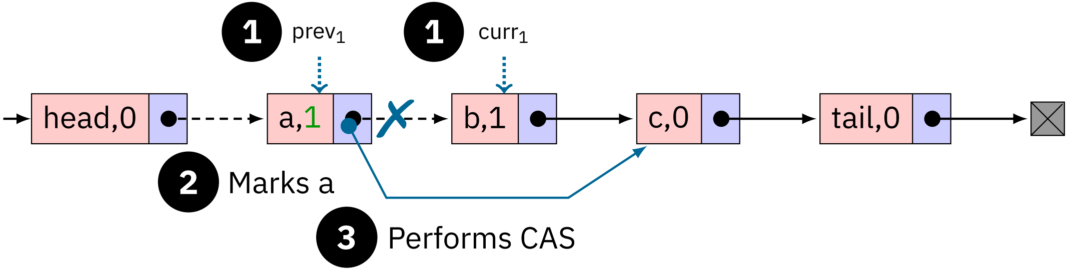
- The `next` field is of type `AtomicMarkableReference<Node>`
- A thread logically removes a node by setting the `marked` bit in the `next` field
- As threads traverse the list, they clean up the list by physically removing marked nodes
  - Threads performing `add()` and `remove()` do not traverse marked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

Why?

Challenge in Traversing Marked Nodes

Thread 1 is executing `remove(b)`

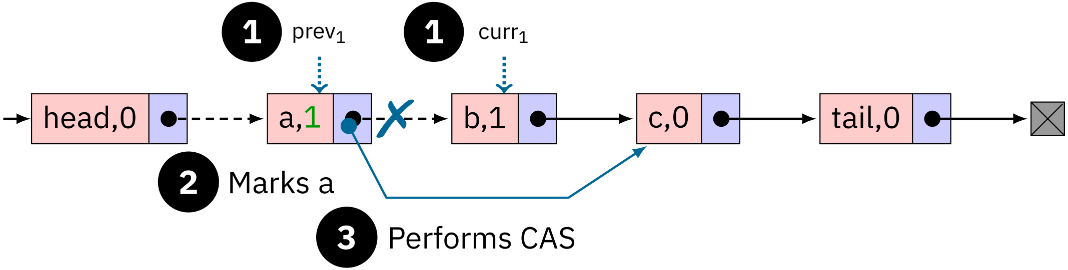
Thread 2 marks a



Challenge in Traversing Marked Nodes

Thread 1 is executing `remove(b)`

Thread 2 marks a



Thread 1 does not delete the marked node a $\implies$ Thread 1 cannot redirect `a.next`

Helper Method

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

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

Helper Method

```java
  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;
      }
    } }
```

Concurrent Set with Nonblocking Synchronization

```java
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;
    }
  }
}
```

Concurrent Set with Nonblocking Synchronization

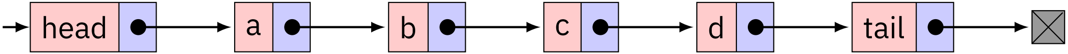
```
public boolean remove(T x) {
  int key = x.hashcode();
  boolean snip;
  while (true) {
    Window w = find(head, key);
    Node pred = w.pred, curr = w.curr;
    if (curr.key != key) return false;
    else {
      Node succ = curr.next.getReference();
      snip = curr.next.compareAndSet(succ, succ, false, true);
      if (!snip) continue;
      pred.next.compareAndSet(curr, succ, false, false);
      return true;
    }
  }
}
```

Concurrent Set with Nonblocking Synchronization

```
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();
}
```

Lock-Free Programming and ABA Problem

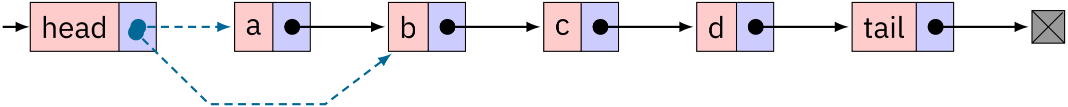
Thread 1 is executing `deq(a)`



Assume deleted nodes can be reused

Lock-Free Programming and ABA Problem

Thread 1 is executing deq(a)

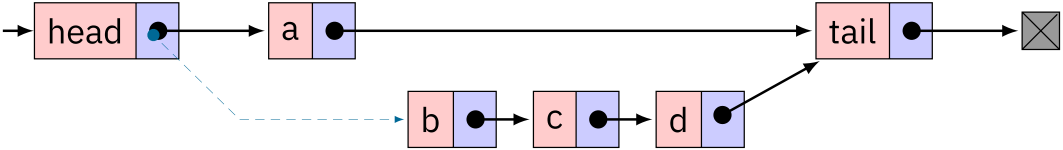


Thread 1 sees head points to a, but gets delayed while executing deq(a)

Lock-Free Programming and ABA Problem

Thread 1 is executing `deq(a)`

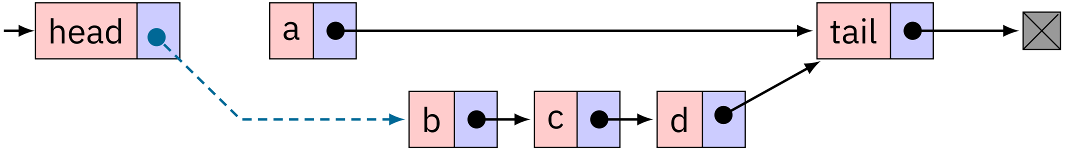
Other threads execute `deq(a, b, c, d)`, then execute `enq(a)`



Lock-Free Programming and ABA Problem

Thread 1 is executing `deq(a)`

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



Thread 1's CAS succeeds, incorrectly setting head to the recycled node b

Avoiding ABA Problem

- Common workaround is to add extra "tag" to the memory address being compared
  - Tag can be a counter that tracks the number of updates to the reference
  - Can steal lower order bits of memory address or use a separate tag field if 128-bit CAS is available

Concurrent Hash Sets for CPU

Hash Sets

Closed addressing Each table entry refers to a set of items, called a bucket. Closed addressing is also known as chaining.

Open addressing Each table entry maps to a single item. Open addressing requires a deterministic probing scheme to search for free slots.

Let l be the number of probing attempts, c be the capacity of the hash set, and $s(k, l)$ the l-th element in the probe sequence where $s(k, 0) = h(k)$.

Linear probing Probing sequence is $s(k, l) = (h(k) + l) \bmod c$. Cache efficient but suffers from clustering.

Quadratic probing Probing sequence is $s(k, l) = (h(k) + l^2) \bmod c$. Incurs more cache misses but avoids primary clustering.

Chaotic probing Probing sequence is $s(k, l) = (h(k) + l \cdot g(k)) \bmod c$ where $g(k)$ is a second hash function. Incurs more cache misses but avoids primary clustering. Chaotic probing is also known as double hashing.

Hash Set with Closed Addressing: Abstract Base Class

```java
public abstract class BaseHashSet<T> {
  protected List<T>[] table;
  protected int setSize;
  public BaseHashSet(int capacity) {
    setSize = 0;
    table = (List<T>[]) new List[capacity];
    for (int i = 0; i < capacity; i++)
      table[i] = new ArrayList<T>();
  }
  public boolean contains(T x) {
    acquire(x);
    try {
      int myBucket = x.hashCode() % table.length;
      return table[myBucket].contains(x);
    } finally {
      release(x);
    }
  }
```

Hash Set with Closed Addressing: Abstract Base Class

```java
  public boolean add(T x) {
    boolean result = false;
    acquire(x);
    try {
      int myBucket = x.hashCode() % table.length;
      result = table[myBucket].add(x);
      setSize = result ? setSize + 1 : setSize;
    } finally {
      release(x);
    }
    if (policy()) // When to resize the hash set?
      resize();
    return result;
  }
}
```

Policies: average bucket size exceeds a fixed threshold, more than 1/4 of the buckets exceed a bucket threshold, or if any single bucket exceeds a global threshold

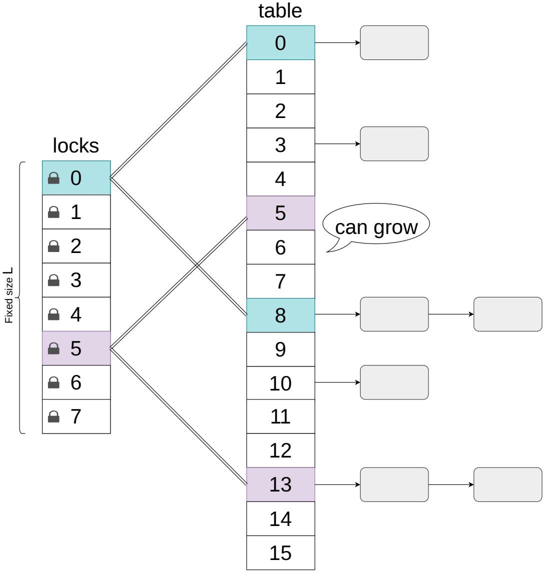
Hash Set with Closed Addressing: Coarse-Grained Locking

```java
public class CoarseHashSet<T> extends BaseHashSet<T>{
  final Lock lock;
  CoarseHashSet(int capacity) {
    super(capacity);
    lock = new ReentrantLock();
  }
  public final void acquire(T x) {
    lock.lock();
  }
  public void release(T x) {
    lock.unlock();
  }
  public boolean policy() {
    // Average size of a bucket is > 4
    return setSize / table.length > 4;
  }
```

Hash Set with Closed Addressing: Coarse-Grained Locking

```java
public void resize() {
  int oldCapacity = table.length;
  lock.lock();
  try {
    if (oldCapacity != table.length)
      return; // someone beat us to it
    int newCapacity = 2 * oldCapacity;
    List<T>[] oldTable = table;
    table = (List<T>[]) new List[newCapacity];
    for (int i = 0; i < newCapacity; i++)
      table[i] = new ArrayList<T>();
    for (List<T> bucket : oldTable)
      for (T x : bucket)
        table[x.hashCode() % table.length].add(x);
  } finally {
    lock.unlock();
  }
}
```

Hash Set with Closed Addressing: Striped Locking



- Each lock protects N/L buckets
- Allows more concurrency than coarse-grained lock

Hash Set with Closed Addressing: Striped Locking

```java
public class StripedHashSet<T> extends BaseHashSet<T>{
  final ReentrantLock[] locks;
  public StripedHashSet(int capacity) {
    super(capacity);
    // Number of locks is initially same as the length of the table. The table
    // can dynamically grow, but not locks. Increasing the number of locks is
    // challenging.
    locks = new Lock[capacity];
    for (int j = 0; j < locks.length; j++)
      locks[j] = new ReentrantLock();
  }
  public final void acquire(T x) {
    locks[x.hashCode() % locks.length].lock();
  }
  public void release(T x) {
    locks[x.hashCode() % locks.length].unlock();
  }
```

Hash Set with Closed Addressing: Striped Locking

```
  public void resize() {
    int oldCapacity = table.length;
    for (Lock lock : locks)
      lock.lock();
    try {
      if (oldCapacity != table.length) return; // someone beat us to it
      int newCapacity = 2 * oldCapacity;
      List<T>[] oldTable = table;
      table = (List<T>[]) new List[newCapacity];
      for (int i = 0; i < newCapacity; i++)
        table[i] = new ArrayList<T>();
      for (List<T> bucket : oldTable)
        for (T x : bucket)
          table[x.hashCode() % table.length].add(x);
    } finally {
      for (Lock lock : locks)
        lock.unlock();
    }
  }
}
```

Hash Set with Closed Addressing: Refinable Hash Set

```java
// Allow resizing number of locks
public class RefinableHashSet<T> extends BaseHashSet<T> {
  // Identifies the owner thread who is resizing, and the boolean is set to true.
  // Used for mutual exclusion with other mutation methods (e.g., add()).
  AtomicMarkableReference<Thread> owner;
  volatile ReentrantLock[] locks;
  public RefinableHashSet(int capacity) {
    super(capacity);
    locks = new ReentrantLock[capacity];
    for (int i = 0; i < capacity; i++)
      locks[i] = new ReentrantLock();
    owner = new AtomicMarkableReference<Thread>(null, false);
  }
  public void release(T x) {
    locks[x.hashCode() % locks.length].unlock();
  }
  protected void quiesce() { // Visit each lock and wait until it is free
    for (ReentrantLock lock : locks)
      while (lock.isLocked()) {}
  }
```

Hash Set with Closed Addressing: Refinable Hash Set

```java
public void acquire(T x) {
  boolean[] mark = {true};
  Thread me = Thread.currentThread();
  Thread who;
  while (true) {
    do { // Wait while some other thread is the owner
      who = owner.get(mark);
    } while (mark[0] && who != me);
    ReentrantLock[] oldLocks = locks;
    ReentrantLock oldLock = oldLocks[x.hashCode() % oldLocks.length];
    oldLock.lock();
    // Check again to see if the locks array has been resized in the meantime
    who = owner.get(mark);
    // locks array has not changed, mark is not set or mark is set and I am the owner
    if ((!mark[0] || who == me) && locks == oldLocks) {
      return;
    } else {
      oldLock.unlock();
    }
  }
}
```

Hash Set with Closed Addressing: Refinable Hash Set

```
    public void resize() {
      int oldCapacity = table.length;
      boolean[] mark = {false};
      int newCapacity = 2 * oldCapacity;
      Thread me = Thread.currentThread();
      if (owner.compareAndSet(null, me, false, true)) { // Try to make yourself the owner
        try {
          if (table.length != oldCapacity) return; // someone else resized first
          quiesce();
          List<T>[] oldTable = table;
          table = (List<T>[]) new List[newCapacity];
          for (int i = 0; i < newCapacity; i++)
            table[i] = new ArrayList<T>();
          locks = new ReentrantLock[newCapacity];
          for (int j = 0; j < locks.length; j++)
            locks[j] = new ReentrantLock();
          initializeFrom(oldTable);
        } finally {
          owner.set(null, false);
        }
      } } }
```

Hash Set with Closed Addressing: Lock-free Hash Set

Challenging to design a correct algorithm with synchronization primitives like CAS

- Not enough to make individual buckets lock-free
- Resizing the table requires atomically moving entries from old buckets to new buckets
- If the table doubles in capacity, then items in the old bucket must be distributed between two new buckets
- CAS operates only on one memory location

Hash Set with Open Addressing: Cuckoo Hashing

Cuckoo hashing is a type of open addressing scheme where collisions are resolved by displacing any earlier item occupying the same slot with a newly added item

- Uses two hash functions which provides two possible locations for each key
- Assume a hash set of size $N = 2k$, and two tables each of size k (denoted by `table[0]` and `table[1]`
- Two independent hash functions h_0 and h_1 map the keys to $0, \ldots, k-1$

`contains(x)` Tests whether either `table[0]`[h_0(x)] or `table[1]`[h_1(x)] is equal to x

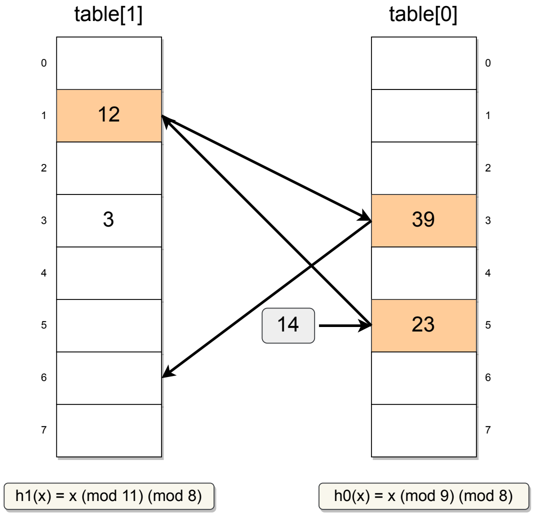
`remove(x)` Checks whether x is in either `table[0]`[h_0(x)] or `table[1]`[h_1(x)] and removes it if found

`add(x)` Repeatedly displace conflicting items until every key has a slot

- May not find an empty slot if the table is full or the sequence of displacements form a cycle
- Need to resize the hash table, choose new hash functions, and restart the add operation after a THRESHOLD of successive displacements is reached

Sequential Cuckoo Hashing: add()

```
public boolean add(T x) {
  if (contains(x)) {
    return false;
  }
  for (int i = 0; i < THRESHOLD; i++) {
    if ((x = swap(h0(x), x)) == null) {
      return true;
    } else if ((x = swap(h1(x), x)) == null)
         {
      return true;
    }
  }
  resize();
  add(x);
}
```



Concurrent Cuckoo Hashing

Challenge is in the possibly long sequence of swap operations during add()

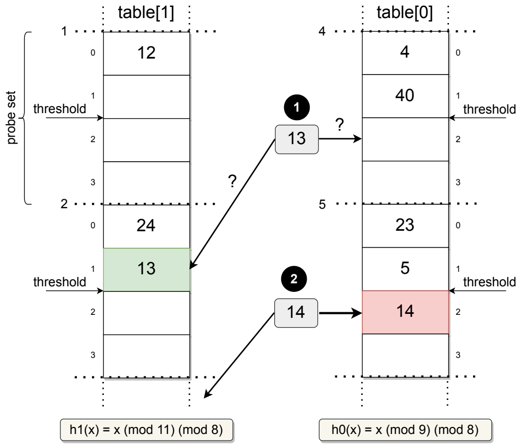
Break up each method call into a sequence of phases, where each phase adds, removes, or displaces a single item x

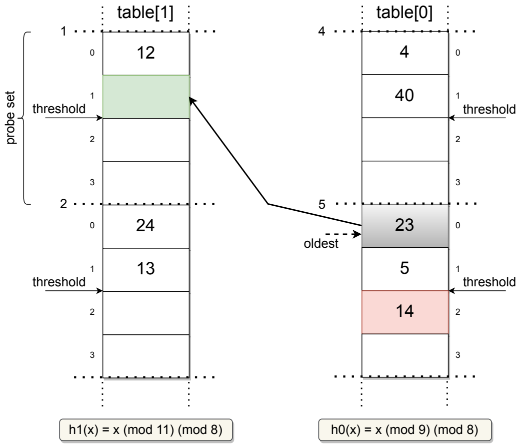
- Hash table is organized as a 2D table of probe sets
- Probe set is a constant-sized set of items with the same hash code
- Each probe set holds at most `PROBE_SIZE` items
- Implementation tries to ensure that when the set is quiescent (i.e., no method calls are in progress) each probe set holds no more than `THRESHOLD < PROBE_SIZE` items

Concurrent Cuckoo Hashing using Phases

```java
public abstract class PhasedCuckooHashSet<T> {
  volatile int capacity;
  volatile List<T>[][] table;
  public PhasedCuckooHashSet(int size) {
    capacity = size;
    table = (List<T>[][]) new java.util.ArrayList[2][capacity];
    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < capacity; j++) {
        table[i][j] = new ArrayList<T>(PROBE_SIZE);
      }
    }
  }
```

Concurrent Cuckoo Hashing using Phases





Example of adding to the Hash Set

Example of relocation

Concurrent Cuckoo Hashing using Phases

```java
public boolean remove(T x) {
  acquire(x);
  try {
    List<T> set0 = table[0][hash0(x) % capacity];
    if (set0.contains(x)) {
      set0.remove(x);
      return true;
    } else {
      List<T> set1 = table[1][hash1(x) % capacity];
      if (set1.contains(x)) {
        set1.remove(x);
        return true;
      }
    }
    return false;
  } finally {
    release(x);
  }
}
```

Concurrent Cuckoo Hashing using Phases

```java
public boolean add(T x) {
  T y = null;
  acquire(x);
  int h0 = hash0(x) % capacity;
  int h1 = hash1(x) % capacity;
  int i = -1, h = -1;
  boolean mustResize = false;
  try {
    if (present(x)) return false;
    List<T> set0 = table[0][h0];
    List<T> set1 = table[1][h1];
    if (set0.size() < THRESHOLD) {
      set0.add(x); return true;
    } else if (set1.size() < THRESHOLD) {
      set1.add(x); return true;
    } else if (set0.size() < PROBE_SIZE) {
      set0.add(x); i = 0; h = h0;
    } else if (set1.size() < PROBE_SIZE) {
      set1.add(x); i = 1; h = h1;
    } else {
      mustResize = true;
    }
  } finally {
    release(x);
  }
  if (mustResize) {
    resize();
    add(x);
  } else if (!relocate(i, h)) {
    resize();
  }
  // x must have been present
  return true;
}
```

Concurrent Cuckoo Hashing using Phases

```
  protected boolean relocate(int i, int hi) {
    int hj = 0, j = 1 - i;
    for (int round=0; round<LIMIT; round++) {
      List<T> iSet = table[i][hi];
      T y = iSet.get(0);
      switch (i) {
        case 0: hj = hash1(y)%capacity; break;
        case 1: hj = hash0(y)%capacity; break;
      }
      acquire(y);
      List<T> jSet = table[j][hj];
      try {
        if (iSet.remove(y)) {
          if (jSet.size() < THRESHOLD) {
            jSet.add(y); return true;

          } else if (jSet.size() < PROBE_SIZE) {
            jSet.add(y);
            i = 1 - i; hi = hj; j = 1 - j;
          } else {
            iSet.add(y); return false;
          }
        } else if (iSet.size() >= THRESHOLD) {
          continue;
        } else {
          return true;
        }
      } finally {
        release(y);
      }
    }
    return false;
  }
}
```

Concurrent Cuckoo Hashing using Striped Locking

```
public class StripedCuckooHashSet<T> extends PhasedCuckooHashSet<T>{
  final ReentrantLock[][] lock;
  public StripedCuckooHashSet(int capacity) {
    super(capacity);
    lock = new ReentrantLock[2][capacity];
    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < capacity; j++)
        lock[i][j] = new ReentrantLock();
    }
  }
  public final void acquire(T x) {
    lock[0][hash0(x) % lock[0].length].lock();
    lock[1][hash1(x) % lock[1].length].lock();
  }
  public final void release(T x) {
    lock[0][hash0(x) % lock[0].length].unlock();
    lock[1][hash1(x) % lock[1].length].unlock();
  }
```

Concurrent Cuckoo Hashing using Striped Locking

```java
  public void resize() {
    int oldCapacity = capacity;
    for (Lock aLock : lock[0]) {
      aLock.lock();
    }
    try {
      if (capacity != oldCapacity) {
        return;
      }
      List<T>[][] oldTable = table;
      capacity = 2 * capacity;
      table = (List<T>[][]) new List[2][capacity];
      for (List<T>[] row : table) {
        for (int i = 0; i < row.length; i++) {
          row[i] = new ArrayList<T>(PROBE_SIZE);
        }
      }
      for (List<T>[] row : oldTable) {
        for (List<T> set : row) {
          for (T z : set) {
            add(z);
          }
        }
      }
    } finally {
      for (Lock aLock : lock[0]) {
        aLock.unlock();
      }
    }
  }
}
```

Concurrent Data Structures for GPU

Concurrent Hashing on GPUs

Requirements

- Need to support high throughput for concurrent accesses to the hash tables
- Need to devise nonblocking algorithms that are tuned to the GPU programming model for good performance

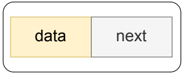
Challenges in designing an efficient hash table

- Lock-based synchronization will not scale to thousands of GPU threads
- Accessing linked-list-based data structures imply making random (uncoalesced) memory accesses from threads in a warp
  - High memory bandwidth is achieved when threads in a warp access consecutive memory locations with a fixed stride
- Dynamically allocating linked list nodes for numerous GPU threads is a bottleneck

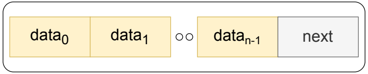
Concurrent Hashing on GPUs with SlabHash

SlabHash builds a concurrent hash table using slab lists instead of linked lists

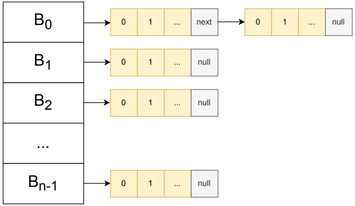
- A slab consists of multiple words of data (i.e., unordered set) and a single next pointer, thereby reducing memory overhead
- Slabs on NVIDIA GPUs can be of 32 words
- Can search a slab list node using a single ballot instruction



Linked list node



Slab list node



Slab list node

S. Ashkiani et al. A Dynamic Hash Table for the GPU, IPDPS’18.

Operations on a Slab List

- **search** Search from the head of the list, load the next slab if the item cannot be found in the current slab
- **insert** Start from the head of the list, use an atomic CAS to insert the new key-value pair into the first empty data slot. An unsuccessful CAS implies someone else occupied the empty slot, so restart looking for an empty slot. Load the next slab if no empty slots are found. Allocate a new slab at the end of the list if all slabs are full.
- **replace** Similar to insert except that the entire slab list needs to be searched
- **delete** Similar to insert except that the entire slab list needs to be searched (depending on whether duplicates are allowed)

More about SlabHash

Provides a dynamic hash table that uses chaining for collision resolution

Universal hash function of the form $h(k; a, b) = ((ak + b) \bmod p) \bmod B$ is used, where a, b are random arbitrary integers, p is a random prime number, and B is the number of buckets.

Threads are assigned independent tasks (i.e., keys), but work is done in parallel per-warp, called warp-cooperative work sharing (WCWS)

- One-to-one mapping maps each thread to a single key, threads in a warp process their 32 keys individually
- Advantage of WCWS is that it significantly reduces branch divergence when compared to per-thread processing
- Disadvantage of WCWS is that all threads within a warp should be active

Concurrent Hashing on GPUs with WarpCore

WarpCore argues that open addressing with linear probing is more amenable to the SIMT execution model on GPUs

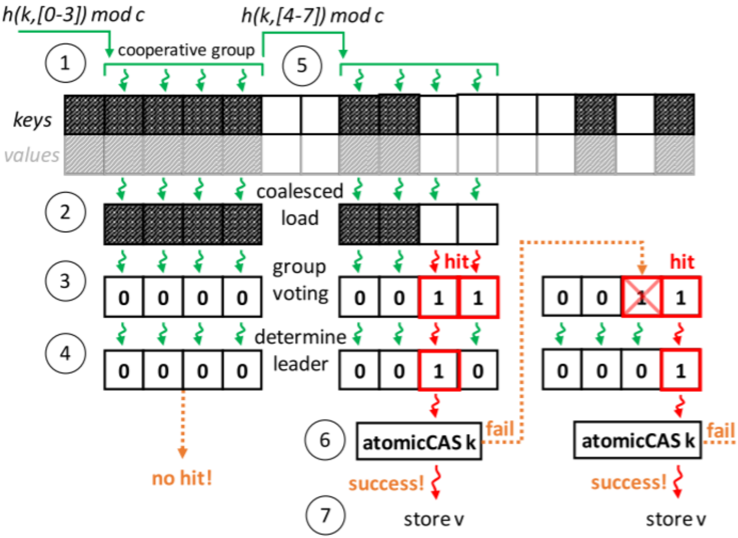
Linked lists are cache-inefficient and lock-free insertion and deletion of nodes in linked lists is complicated because of the ABA problem

Parallel probing scheme

- Mapping each thread to a key will lead to a different probing sequence for threads in a warp, leading to non-coalesced global memory accesses
- Can use an entire warp of 32 threads per input key k, such that each thread with lane ID `t` probes a different hash table position $h(k, t)$ mod c, but linear probing suffers from clustering
- WarpCore uses double hashing with an inner intra-warp linear probing, double hashing determines the starting offset for linear probing

D. Jünger et al. WarpCore: A Library for fast Hash Tables on GPUs, HiPC'20.

Example of Insertion Operation with WarpCore



References

- M. Herlihy et al. The Art of Multiprocessor Programming. Chapters 3, 9, 10, 13, 2nd edition, Morgan and Claypool.
- Mark Moir and Nir Shavit. Concurrent Data Structures In Handbook of Data Structures and Applications, Chapman and Hall/CRC Press, 2004.