The Art of Multiprocessor Programming

Maurice Herlihy Nir Shavit

July 17, 2006

Contents

Chapter 10

Linked Lists: the Role of Locking

10.1 Introduction

In Chapter 9 we saw how to build efficient, scalable spin locks that provide mutual exclusion efficiently even when they are heavily used. You might think that it is now a simple matter to construct scalable concurrent data structures: simply take a sequential implementation of the class, add a scalable lock field and ensure that each method executes only while holding that lock, an approach called *coarse-grained synchronization*.

Often, coarse-grained synchronization works well, but there are important cases where it doesn't. The problem is that an object that uses a single lock to mediate all of its method calls is not always scalable, even if the object lock itself is scalable. Coarse-grained synchronization works well when levels of concurrency are low, but if too many threads try to access the object at the same time, then the object itself becomes a sequential bottleneck, forcing threads to "stand in line".

This chapter introduces a variety of techniques that go beyond coarse-grained locking to allow multiple threads to access a single object at the same time.

- *Fine-grained synchronization:* Instead of using a single lock to synchronize every access to an object, we split the object into independently-

[0] *Portions of this work are from the book* The Art of Multiprocessor Programming *by Maurice Herlihy and Nir Shavit, published by Morgan Kaufmann Publishers, Copyright 2006 Elsevier Inc. All rights reserved..*

synchronized components, ensuring that method calls interfere only when trying to access the same component at the same time.

- *Optimistic synchronization:* Many objects, such as trees or lists, consist of multiple components linked together by references. Some methods search for a particular component (for example, a list or tree node containing a particular key). One way to reduce the cost of fine-grained locking is to search without acquiring any locks at all. If the method finds the sought-after component, it locks that component, and then checks that the component has not changed in the interval between when it was inspected and when it was locked. This technique is worthwhile only if it succeeds more often than not, which is why we call it optimistic.

- *Lazy synchronization:* Sometimes it makes sense to postpone hard work. For example, the task of removing a component from a data structure can be split into two phases: the component is *logically removed* simply by setting a tag bit, and later, the component can be *physically removed* by unlinking it from the rest of the data structure.

- *Lock-Free Synchronization* Sometimes we can eliminate locks entirely, relying on built-in atomic operations such as compareAndSet() for synchronization.

Each of these techniques can be applied (with appropriate customization) to a variety of common data structures. In this chapter we consider how to use linked lists to implement a *set*, a collection of objects that contains no duplicate elements.

For our purposes, a *Set* provides the following three methods:

- The add(x) method adds x to the set, returning *true* if and only if x was not already there.

- The remove(x) method removes x from the set, returning *true* if and only if x was there.

- The contains(x) returns *true* if and only if the set contains x.

For each method, we say that a call is *successful* if it returns *true*, and *unsuccessful* otherwise.

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

Figure 10.1: *Set Interface: the add() method adds an object to the set (no effect if that object is already in the set), the remove() method removes it (if it is present), and the contains() method returns a Boolean indicating whether the object is in the set.*

```
private class Entry {
  T object;
  int key;
  Entry next;
}
```

Figure 10.2: *List entry: an entry keeps track of the object itself (of generic type $\mathcal{T}$) the object's key, and the next entry in the list. Sometimes we will add an additional lock field for fine-grained synchronization.*

10.2 List-based Sets

This chapter presents a range of concurrent set algorithms, all based on the same basic idea. We implement the set as a linked list of entries. As shown in Figure 10.2, the Entry class has three fields. The object field is the actual object of interest. The key field is the object's hash code. Entries are sorted in key order, providing an efficient way to detect when an object is absent from the list. The next field is a reference to the next entry in the list. Later on we will add an additional lock field for fine-grained synchronization. For the sake of simplicity we assume that each object's hash code is unique (relaxing this assumption is left as an exercise). We also make sure to associate an object with the same entry and key throughout any given example, which allows us to abuse notation slightly and use the same symbol to refer to the entry, its associated key, and its associated object (entry a will have key a and associated object a, and so on.).

The list has two kinds of entries. In addition to *regular* entries that hold objects in the set, we use two *sentinel* entries, called `head` and `tail`, as the first and last list elements. Sentinel entries are never added, removed, or

searched for, and their keys are the minimum and maximum integer values. Ignoring synchronization for the moment, the top part of Figure 10.3 shows a schematic description of how an object is added to the set. Each thread has two local variables used to traverse the list: `curr` is the current entry and `pred` is its predecessor. To add a new object to the set, a thread sets the two local variables `pred` and `curr` to `head`, and moves down the list, comparing `curr`'s key to the key of the object being added. If they match, the object is already present in the set, so return *false*. If `pred` precedes `curr` in the list, `pred`'s key is lower than that of the inserted object, and `curr`'s key is higher, then the object is not present in the list. Create a new entry b to hold the object, set b to point to `curr`, then set `pred` to point to b, and the object is now a member of the set. Removing an object from the set works in a similar way.

10.3 Concurrent Reasoning

Reasoning about concurrent data structures may seem impossibly difficult, but it is a skill that can be learned, like many others. Often the key to understanding a concurrent data structure is to understand its *invariants*: properties that always hold. We can show that a property is invariant by showing that these two claims are true:

1. The property holds when the object is created.

2. Once the property holds, then no thread can take a step that makes the property false.

All the invariants we consider here hold trivially when the list is created, so we will focus on the second point.

Specifically, we will check that each invariant is preserved by each invocation of insert(), remove(), and contains() methods. This approach works only if we can assume that these methods are the *only* ones that modify entries, a property sometimes called freedom from *interference*. In the list algorithms considered here, entries are internal to the list implementation, so freedom from interference is guaranteed because users of the list have no opportunity to modify its internal entries.

We will require freedom from interference even for entries that have been removed from the list, since some of the algorithms we consider permit one thread to unlink an entry while it is being traversed by others. Our task is made easier because we do not attempt to reuse list entries that have been

removed from the list, relying instead on the Java garbage collector to recycle that memory. If we were programming in a language without garbage collection, then we would have to reason carefully about any recycling methods as well.

When reasoning about concurrent object algorithms, it is important to make a clear distinction between the object's *abstract value* (here, a set of objects), and its concrete *representation* (here, a list of entries). Not every list of entries is a meaningful representation for a set. The algorithm's *representation invariant* (rep invariant for short) characterizes which representations make sense.

We now introduce some notation that will be useful when discussing the list representation. If a and b are entries, $a \rightarrow b$ is shorthand for saying that a's next is a reference to b. We use $a \rightsquigarrow b$ to mean that b is reachable from a through a sequence of next references. More precisely,

$$a \rightsquigarrow a \tag{10.1}$$

$$\text{If } a \rightsquigarrow b \text{ and } b \rightarrow c \text{ then } a \rightsquigarrow c \tag{10.2}$$

It is easy to see that

$$\text{If } a \rightarrow b \text{ and } b \rightsquigarrow c \text{ then } a \rightsquigarrow c \tag{10.3}$$

For brevity, we say a is *reachable* as shorthand for $\texttt{head} \rightsquigarrow a$.

All the set algorithms in this chapter require at least the following invariant conditions (some require more, as explained later). First, sentinels are neither added nor removed:

$$\texttt{head} \rightsquigarrow \texttt{tail} \tag{10.4}$$

Second, entries are sorted by key, and keys are unique:

$$\text{If } a \rightarrow b \text{ then } a.\texttt{key} < b.\texttt{key} \tag{10.5}$$

Some of the algorithms we consider later will require additional conditions.

Think of the representation invariant as a contract among the object's methods. Each method call preserves the invariant, and relies on the other methods to preserve the invariant. In this way, we can reason about each method in isolation, without having to consider all the possible ways they might interact.

Given a list satisfying the rep invariant, which set does it represent? The meaning of such a list is given by an *abstraction map* carrying lists that

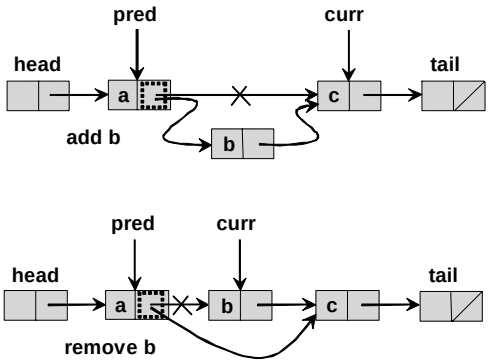

Figure 10.3: *Insertion and removal of list entries. As noted earlier, for brevity, we slightly abuse notation by using the same variable name to refer to a node, its associated key, and the set object it represents. To insert a new entry b into a list, a thread uses two variables:* `curr` *is the current entry, and* `pred` *is its predecessor. Move down the list comparing the keys for* `curr` *and b. If a match is found, the object is present, so return false. As soon as* `curr` *reaches an entry with a higher key, we know the object is not already present in the set. Set b's next to point to* `curr`*, then set* `pred`*'s next to point to b. To delete* `curr`*, the thread sets* `pred`*'s next field to* `curr`*'s next value.*

satisfy the rep invariant to sets. In this case, the abstraction map is simple: an object is in the set if and only if it is referred to by some reachable entry. Bearing in mind that `head` is the start of the list, we define the abstraction map $\mathcal{S}$ as follows:

$$\mathcal{S}(\texttt{head}) = \{\S | \text{ there exists } \dashv \text{ such that } \texttt{head} \rightsquigarrow \ddagger \text{ and } \dashv.\texttt{object} = \S\} \tag{10.6}$$

What safety and liveness properties must we show to prove our representation correct? Our safety property is linearizability. As we saw in Chapter 5, to show that a concurrent data structure algorithm is a linearizable implementation of some sequentially specified object, it is enough to identify a *linearization point*, a single atomic memory operation when the method call "takes effect". This operation can be a read, a write, or a more complex atomic operation. Looking at any execution history of our list-based representation of sets, it must be the case that if the abstraction map is applied to the representation at the linearization points, the resulting sequence of states and method calls defines a valid sequential execution of a set object. Here, add(a) adds a to the abstract set, remove(a) removes a from the abstract set, and contains(a) returns *true* or *false* depending on

```
public class CoarseList<T> {
  private Entry head;
  private Lock lock = new ReentrantLock();
  public CoarseList() {
    this.head  = new Entry(Integer.MIN_VALUE);
    this.head.next = new Entry(Integer.MAX_VALUE);
  }
  public boolean add(T object) {
    Entry pred, curr;
    int key = object.hashCode();
    lock.lock();
    try {
      pred = this.head;
      curr = pred.next;
      while (curr.key < key) {
        pred = curr;
        curr = curr.next;
      }
      if (key == curr.key) {  // present
        return false;
      } else {                // not present
        Entry entry = new Entry(object);
        entry.next = curr;
        pred.next = entry;
        return true;
      }
    } finally {               // always unlock
      lock.unlock();
    }
  }
```

Figure 10.4: *Coarse-Grained add(): all methods acquire a single lock, which is released on exit by the* **finally** *block.*

whether a is in the set.

Our liveness properties will vary for different list representations. Some of our implementations use locks, and will require no deadlocks or lock-outs. Other implementations will be free of locks, and will require the lock-free progress condition.

```
public boolean remove(T object) {
  Entry pred, curr;
  int key = object.hashCode();
  lock.lock();
  try {
    pred = this.head;
    curr = pred.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    if (key == curr.key) { // present
      pred.next = curr.next;
      return true;
    } else {
      return false;          // not present
    }
  } finally {                // always unlock
    lock.unlock();
  }
}
```

Figure 10.5: *Coarse-Grained remove(): all methods acquire a single lock, which is released on exit by the* **finally** *block.*

We are now ready to consider a range of list-based set algorithms, successively refining the algorithms in an attempt to reduce the extent and granularity of locking. As we explained earlier, in each of these algorithms the various methods scan through the list using two local variables: `curr` is the current entry and `pred` is its predecessor. Because these variables are in use only during certain method calls, we adopt the convention that when those variables are out-of-scope, they have value *null*. Because these variables are local to each thread, we will often use $\texttt{pred}_A$ and $\texttt{curr}_A$ to denote the variables local to thread A.

10.4 Coarse-Grained Synchronization

We start with a simple algorithm using coarse-grained synchronization. Figures 10.4 and 10.5 show the add() and remove() methods for this coarse-

grained algorithm. (The contains() method works in much the same way, and is left as an exercise.) The list itself has a single Lock which every method call must hold while it is running. The principal advantage of this algorithm, which should not be discounted, is that it is obviously correct. All methods perform operations on the list while holding the lock, so the execution is essentially sequential and for each method it is easy to verify that any point during which a method is holding the Lock is a valid linearization point given the abstraction map of Equation 10.6. The implementation inherits its progress conditions from those of the Lock, and so, if this Lock implementation is lockout-free, our implementation will also be lockout-free. If contention is low, this is an excellent way to implement a list. If contention is high, however, then even if the lock itself performs well, threads will be delayed waiting for one another.

10.5 Fine-Grained Synchronization

We can improve concurrency by locking individual entries, rather than locking the list as a whole. Instead of placing a lock on the entire list, let us add a Lock to each entry, along with lock() and unlock() methods. As a thread traverses the list, it locks each entry when it first visits, and sometime later releases it. Such *fine-grained* locking permits concurrent threads to traverse the list together in a pipelined fashion.

Consider two entries a and b where $a \rightarrow b$. A moment's thought shows that it is not safe for the thread to unlock a before locking b because another thread could remove b from the list in the interval between the unlocking of a and the locking of b. Instead, a thread A will acquire locks in a kind of "hand-over-hand" order: except for the initial sentinel entry, acquire the lock for $\texttt{curr}_A$ only while holding the lock for $\texttt{pred}_A$. This locking protocol is sometimes called *lock coupling*. (Notice that there is no obvious way to implement lock coupling using Java's `synchronized` methods.)

Figure 10.7 shows the fine-grained algorithm's remove() method. Just as in the coarse-grained list, remove() makes $\texttt{curr}_A$ unreachable by setting $\texttt{pred}_A$'s next field to $\texttt{curr}_A$'s successor. To be safe, remove() must lock both $\texttt{pred}_A$ and $\texttt{curr}_A$. To see why, consider the following scenario, illustrated in Figure 10.8. Thread A is about to remove entry a, the first entry in the list, while thread B is about to remove entry b, where $a \rightarrow b$. Suppose A locks `head`, and B locks a. A then sets `head.next` to b, while B sets `a.next` to `c`. The net effect is to remove a, but not b. The problem is that there is no overlap between the locks held by the two remove() calls.

```
public boolean add(T object) {
  int key = object.hashCode();
  Entry pred = this.head; // sentinel node;
  pred.lock();
  try {
    Entry curr = pred.next;
    curr.lock();
    try {
      while (curr.key <= key) {
        if (object.equals(curr.object)) {
          return false; // already in list
        }
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      // link into list
      Entry newEntry = new Entry(object);
      newEntry.next = curr;
      pred.next = newEntry;
      return true;
    } finally {        // always unlock
      curr.unlock();
    }
  } finally {          // always unlock
    pred.unlock();
  }
}
```

Figure 10.6: *The fine-grained add() method: uses hand-over-hand locking to traverse the list looking for an object. It relies on the finally clause to release locks before returning.*

To guarantee progress, it is important that all methods acquire locks in the same order, starting at the `head` and following next references toward the `tail`. As Figure 10.10 shows, if different method calls were to acquire locks in different orders, a *deadlock* could occur. In this example, to insert a, thread A, has acquired the lock for b and is attempting to acquire the

```
1  public boolean remove(T object) {
2      Entry last = null, pred = null, curr = null;
3      int key = object.hashCode();
4      this.head.lock(); // lock prececessor
5      try {
6        pred = this.head; // sentinel node
7        curr = pred.next;
8        curr.lock();        // lock current
9        try {
10         while (curr.key <= key) {
11           if (object.equals(curr.object)) { // unlink entry
12             pred.next = curr.next;
13             return true;
14           }
15           last = pred;
16           pred = curr;
17           last.unlock();
18           curr = pred.next;
19           curr.lock();
20         }
21         return false; // not present
22       } finally {      // always unlock
23         curr.unlock();
24       }
25     } finally {          // always unlock
26       pred.unlock();
27     }
28   }
```

Figure 10.7: *The fine-grained remove() method. The method locks both the entry to be removed and its predecessor before removing that entry.*

lock for `head`, while thread B, to remove b, has acquired the lock for b and is attempting to acquire the lock for `head`. Clearly, these method calls will never finish. As we will see when constructing other data structures, avoiding deadlocks is one of the principal challenges of programming with locks.

The fine-grained algorithm maintains the rep invariant: sentinels are never added or removed, and entries are sorted by key value without dupli-

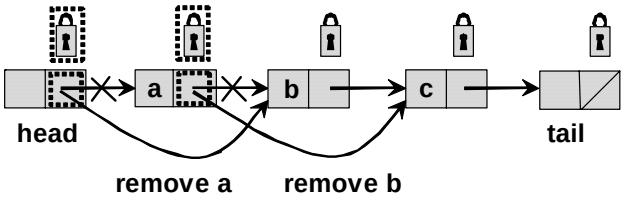

Figure 10.8: *Why remove() must acquire two locks. Thread A is about to remove a, the first entry in the list, while thread B is about to remove b, where $a \rightarrow b$. Suppose A locks* `head`*, and B locks a. Thread A then sets* `head.next` *to b, while B sets a's* `next`*pointer to c. The net effect is to remove a, but not b.*

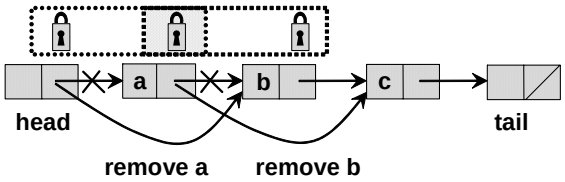

Figure 10.9: *Hand-over-hand locking ensures that if concurrent remove() calls try to remove adjacent entries, then they acquire conflicting locks. Thread A is about to remove entry a, the first entry in the list, while thread B is about to remove entry b, where $a \rightarrow b$. Because A must lock both* `head` *and a and B must lock both a and b, they are guaranteed to conflict on a, forcing one call to wait for the other.*

cates. The abstraction map is the same as for the course-grained list: an object is in the set if and only if its entry is reachable. The linearization point for a successful remove() call is when $\texttt{curr}_A$ becomes unreachable, and the linearization point for an unsuccessful call occurs when $\texttt{curr}_A$ is set to an entry with a higher key.

To complete the linearizability proof, we need to introduce two additional invariant properties that our implementation must meet. The first property says that if A and B are distinct threads, and $\texttt{pred}_A$ is not **null**, then $\texttt{pred}_A$ is distinct from $\texttt{pred}_B$.

$$\text{If } A \neq B \text{ and } \texttt{pred}_A \neq \textbf{null} \text{ then } \texttt{pred}_A \neq \texttt{pred}_B. \tag{10.7}$$

The second property says that for all threads A, if $\texttt{pred}_A$ is not **null**, then

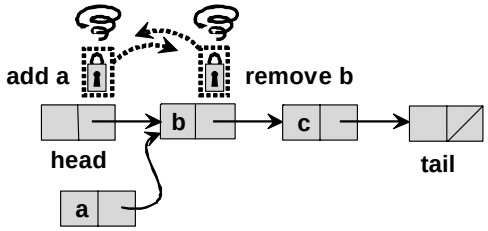

Figure 10.10: *A deadlock can occur if, for example, remove() and add() calls acquire locks in opposite order. Thread A is about to insert a by locking first b and then* `head`*, and thread B is about to remove entry b by locking first* `head` *and then b. Each thread holds the lock the other is waiting to acquire, so neither one can make progress.*

$\texttt{pred}_A$ is reachable from `head`, and `tail` is reachable from $\texttt{pred}_A$.

$$\text{If } \texttt{pred}_A \neq \textbf{null} \text{ then } \texttt{head} \rightsquigarrow \texttt{pred}_A \rightsquigarrow \texttt{tail}. \tag{10.8}$$

We check that Property 10.7 is invariant. When the list is initialized, $\texttt{pred}_A$ is **null** for all A, so the property is clearly true. If, at any time, neither $\texttt{pred}_A$ nor $\texttt{pred}_B$ are **null**, then A holds the lock for $\texttt{pred}_A$, and B holds the lock for $\texttt{pred}_B$, so the two entries must be distinct.

To see why the remove() method (Figure 10.7) does not violate Property 10.8, note that A holds the lock for $\texttt{pred}_A$ while it traverses the list: it starts by locking `head` (line 4), sets $\texttt{pred}_A$ to `head` (line 6), and repeated locks $\texttt{curr}_A$ (line 8) and sets $\texttt{pred}_A$ to $\texttt{curr}_A$ (line 16). Because the entry referred to by $\texttt{pred}_A$ is always locked, no other thread can remove it from the list, and $\texttt{head} \rightsquigarrow \texttt{pred}_A$ holds throughout the call. In the same way, `tail` remains reachable from $\texttt{pred}_A$.

Having established the invariance of Properties 10.7 and 10.8, we can put the pieces together to see why, for example, the fine-grained remove(a) method is linearizable. Let a be the object being removed by thread A. There are two cases to consider: a successful call that finds a in the list, and an unsuccessful call that doesn't. If the thread reaches line 11, then the call was successful. Immediately before that line, $\texttt{pred}_A \rightarrow \texttt{curr}_A$, and the value field of $\texttt{curr}_A$ is a. We know that $\texttt{head} \rightsquigarrow \texttt{pred}_A \rightarrow \texttt{curr}_A$, and therefore a is in the abstract set. Moreover, no other thread can remove a from the set because A is holding the lock on its predecessor. The linearization point for a successful remove() call occurs at line 12, where $\texttt{curr}_A$ becomes no longer reachable.

```
private boolean validate(Entry pred, Entry curr) {
  Entry entry = head;
  while (entry.key <= pred.key) {
    if (entry == pred)
      return pred.next == curr;
    entry = entry.next;
  }
  return false;
}
```

Figure 10.11: *Optimistic validation: check that* `head` $\rightsquigarrow$ $\texttt{pred}_A \rightarrow \texttt{curr}_A$.

If the thread reaches line 21, then the call was unsuccessful. Immediately before that line, $\texttt{pred}_A \rightarrow \texttt{curr}_A$, $\texttt{pred}_A$'s key is less than a's, and $\texttt{curr}_A$'s key is greater. The rep invariant guarantees that if a were reachable, then it would lie between these two entries. The linearization point for an unsuccessful remove() call occurs at the moment `curr` is set to an entry with a higher key.

The proof for the add() and contains() methods is similar. Note also that all methods preserve the rep invariant: they do not add or delete sentinels, and the list remains sorted and without duplicates.

The fine-grained algorithm is lockout-free, but arguing this property is harder than in the course-grained case. We assume that all individual locks provide lockout-freedom. Because all methods acquire locks in the same down-the-list order, there are no deadlocks. Thus, if a given thread A attempts to lock `head`, eventually it will do so. From that point on, given that there are no deadlocks, eventually all locks held by threads ahead of A in the list will be freed, and A will succeed in acquiring the locks of the $\texttt{pred}_A$ and $\texttt{curr}_A$ entries necessary to complete its operation.

10.6 Optimistic Synchronization

Although fine-grained synchronization is an improvement over a single, coarse-grained lock, it still requires a potentially long sequence of lock acquisitions and releases. Moreover, threads accessing disjoint parts of the list may still block one another for long periods. For example, a thread removing the second item in the list locks that entry, which may block concurrent threads searching for entries later in the list.

```
1  public boolean add(T object) {
2    int key = object.hashCode();
3    while (true) {
4      Entry pred = this.head;
5      Entry curr = pred.next;
6      while (curr.key <= key) {
7        pred = curr; curr = curr.next;
8      }
9      pred.lock(); curr.lock();
10     try {
11       if (validate(pred, curr)) {
12         if (curr.key == key) { // present
13           return false;
14         } else {                  // not present
15           Entry entry = new Entry(object);
16           entry.next = curr;
17           pred.next = entry;
18           return true;
19         }
20       }
21     } finally {                   // always unlock
22       pred.unlock(); curr.unlock();
23     }
24   }
25 }
```

Figure 10.12: *Optimistic add(): searches without locking, acquires locks, and validates before adding the new entry.*

One way to reduce locking costs is to take a chance: search without acquiring locks, lock the entries we find, and then confirm that we locked the correct entries. If a synchronization conflict caused us to lock the wrong entries, then we release the locks and start over. We expect this kind of conflict to be rare, which is why we call this technique *optimistic synchronization*.

Figure 10.12 shows an optimistic add() algorithm. Thread A searches the list without acquiring any locks (lines 6 through 8), stopping when $\texttt{curr}_A$'s key is greater than or equal to a. It then locks $\texttt{pred}_A$ and $\texttt{curr}_A$, and calls

```
public boolean remove(T object) {
  int key = object.hashCode();
  while (true) {
    Entry pred = this.head;
    Entry 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) { // present in list
          pred.next = curr.next;
          return true;
        } else {               // not present in list
          return false;
        }
      }
    } finally {                // always unlock
      pred.unlock(); curr.unlock();
    }
  }
}
```

Figure 10.13: *Optimistic remove(): searches without locking, acquires locks, and validates before removing the entry.*

validate() to check that:

$$\texttt{head} \rightsquigarrow \texttt{pred}_A \rightarrow \texttt{curr}_A. \tag{10.9}$$

If the validation succeeds, then A proceeds as before: if $\texttt{curr}_A$'s key is greater than k, A adds a new entry between $\texttt{pred}_A$ and $\texttt{curr}_A$ and returns *true*, and otherwise it returns *false*. As Figure 10.14 shows, validation is crucial as either of the "arrows" in Equation 10.9 could have changed between when they were observed and when the thread acquired the locks. For a thread A attempting to remove a node a, $\texttt{curr}_A$ and all entries between $\texttt{curr}_A$ and a (including a) may be removed while A is still traversing $\texttt{curr}_A$. Thread A will then proceed to the point where $\texttt{curr}_A$ points to a, and, without a validation, "successfully" remove a, even though a is no longer in the

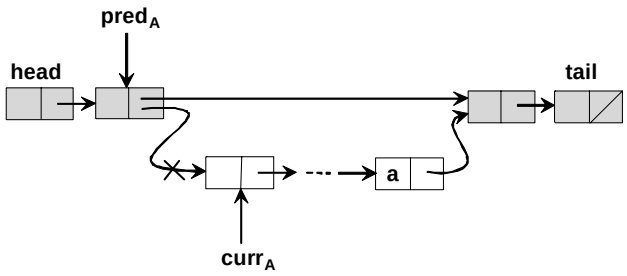

Figure 10.14: *Validation is necessary in the optimistic algorithm. Thread A is attempting to remove an entry a. While traversing the list,* $\texttt{curr}_A$ *and all entries between* $\texttt{curr}_A$ *and a (including a) might be removed. In such a case thread A would proceed to the point where* $\texttt{curr}_A$ *points to a, and, if there were no validation, would "successfully" remove a, even though it is no longer in the list. Validation determines that a is no longer reachable from the* `head`*.*

list. A call to validate () will detect that a is no longer in the list and cause the method to retry the removal attempt.

Properties 10.7 and 10.8 are not invariants for the optimistic list algorithm. Nothing prevents $\texttt{pred}_A$ from being the same as $\texttt{pred}_B$ for distinct threads A and B. Moreover, there is no guarantee that $\texttt{pred}_A$ is always reachable from `head`. Instead of being invariants, Properties 10.7 and 10.8 are more like *goals*. When the optimistic add() is done searching, it gambles on locking the two entries, and then (effectively) checks whether the properties hold. By locking $\texttt{pred}_A$, a thread A establishes that $\texttt{pred}_A$ is distinct from any *locked* $\texttt{pred}_B$. To derive an invariant property, we refine Property 10.7:

$$\text{If } A \neq B, \texttt{pred}_A, \texttt{pred}_B \neq \textbf{null} \text{ and locked, then } \texttt{pred}_A \neq \texttt{pred}_B. \quad (10.10)$$

If $\texttt{pred}_A$ is locked by A and reachable, then no other thread can make it unreachable. Removing an earlier or later entry will not make $\texttt{pred}_A$ unreachable, and no other thread can remove $\texttt{pred}_A$ itself because it is locked by A. We leave as an exercise to show that if $\texttt{pred}_A$ is not **null**, then $\texttt{pred}_A \rightsquigarrow \texttt{tail}$, even if $\texttt{pred}_A$ is unreachable. It follows that Property 10.8 holds if validation succeeds.

Since in the optimistic list, threads traverse nodes that have been removed from the list, we need to state the following invariant that guarantees

```
private boolean validate(Entry pred, Entry curr) {
  return !pred.marked && !curr.marked && pred.next == curr;
}
```

Figure 10.15: *Lazy Validation: we need to check that neither the* `pred` *nor the* `curr` *has been logically deleted, and check that* `pred` $\rightarrow$ `curr`.

that if an item is in the list, it will be found.

$$\text{For all } a \in \mathcal{S} \text{ if } \texttt{pred}_{\mathcal{A}} \neq \textbf{null}, \texttt{pred}_{\mathcal{A}}.\texttt{key} \leq \dashv.\texttt{key} \text{ then } \texttt{pred}_{\mathcal{A}} \rightsquigarrow \dashv. \tag{10.11}$$

It says that for all threads A, an entry a will remain reachable from $\texttt{pred}_A$ as long as it is reachable from `head`. The invariant follows immediately from the freedom from interference property of list methods, which implies that the next pointers of all removed entries remain unchanged as long as they are reachable by some thread.

We have established that Property 10.10 is invariant, and 10.8, while no longer invariant, holds at the linearization points of all the methods. Given the invariance of Property 10.11, the rest of the analysis is essentially the same as for the fine-grained algorithm.

The optimistic algorithm is not lockout-free even if all individual entry locks are lockout-free, as a thread might be forever delayed due to repeatedly added new entries (see homework Exercise 10.10.5).[1] However, we would expect it to do well in practice, as chances of such bad scenarios occurring are slim.

10.7 Lazy Synchronization

The optimistic implementation works best if the cost of traversing the list twice without locking is significantly less than the cost of traversing the list once with locking. One drawback of the optimistic approach is that the contains() method must acquire locks, which is unattractive since we expect contains() calls to occur much more often than calls to other methods.

Our next step is to refine this technique so that the contains() calls do not need to acquire locks, and the add() and remove() methods traverse the list only once (in the absence of contention). In fact, the contains() method

[1] Note that even though this means that a thread is delayed by other threads making progress, it is lockout-free and not lock-free because if any thread holding a lock is delayed, it will not make any progress.

```
public boolean add(T object) {
  int key = object.hashCode();
  while (true) {
    Entry pred = this.head;
    Entry curr = head.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key == key) { // present
            return false;
          } else {                    // not present
            Entry entry = new Entry(object);
            entry.next = curr;
            pred.next = entry;
            return true;
          }
        }
      } finally { // always unlock
        curr.unlock();
      }
    } finally { // always unlock
      pred.unlock();
    }
  }
}
```

Figure 10.16: *The Lazy add().*

will meet the lock-free progress condition: it will only be delayed if other methods succeed infinitely often.

We add to each entry a Boolean marked field indicating if its associated object is in the set. Removing an object from the set is done in two steps: first $\texttt{curr}_A$ is marked, indicating that $\texttt{curr}_A$'s object has been removed from

```
1  public boolean remove(T object) {
2    int key = object.hashCode();
3    while (true) {
4      Entry pred = this.head;
5      Entry curr = head.next;
6      while (curr.key < key) {
7        pred = curr; curr = curr.next;
8      }
9      pred.lock();
10     try {
11       curr.lock();
12       try {
13         if (validate(pred, curr)) {
14           if (curr.key != key) {     // present
15             return false;
16           } else {                   // absent
17             curr.marked = true;      // logically remove
18             pred.next = curr.next;   // physically remove
19             return true;
20           }
21         }
22       } finally {                    // always unlock curr
23         curr.unlock();
24       }
25     } finally {                      // always unlock pred
26       pred.unlock();
27     }
28   }
29 }
```

Figure 10.17: *Lazy remove(): removes entries in two steps, logical and physical.*

the set (*logical* removal), and then $\texttt{pred}_A$'s next field is redirected and the entry is removed (*physical* removal). The logical removal of an entry does not change the ability of threads to continue to traverse it, maintaining the property that traversals are not delayed by ongoing modifications. The linearization point of the remove() method call occurs when the entry is logically removed. Physically removing the entry is a clean-up step that does not affect the current value of the abstraction map.

```
public boolean contains(T object) {
  int key = object.hashCode();
  Entry curr = this.head;
  while (curr.key < key)
    curr = curr.next;
  return curr.key == key && !curr.marked;
}
```

Figure 10.18: *Lazy List: the contains() method.*

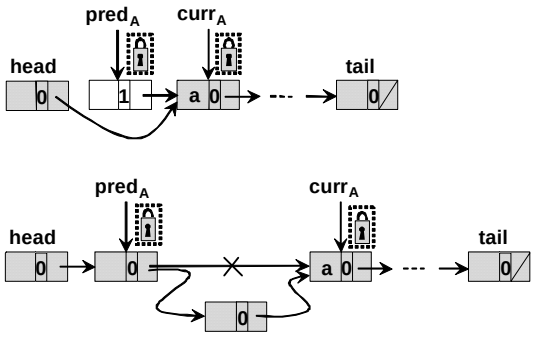

Figure 10.19: *Why the lazy list must validate that* `pred` *and* `curr` *point to one another and are not marked. In the top figure a thread A is attempting to remove an entry a. After it reaches the point where* $\texttt{pred}_A$ *points to* $\texttt{curr}_A$*, and before it acquires locks on these entries, the entry pointed by* $\texttt{pred}_A$ *is logically and physically removed. After it acquires the locks, validation will detect this when checking* $\texttt{pred}_A$*'s mark. In the bottom figure, the same thread A is attempting to remove an entry a. After it reaches the point where* $\texttt{pred}_A$ *points to* $\texttt{curr}_A$*, and before it acquires locks on these entries, a new entry is added between* $\texttt{pred}_A$ *and* $\texttt{curr}_A$*. After it acquires the locks, even though neither* $\texttt{pred}_A$ *or* $\texttt{curr}_A$ *are marked, validation will detect that* $\texttt{pred}_A$ *does not point to* $\texttt{curr}_A$*,h and the removal attempt of a will be restarted from the* `head`*.*

In more detail, all methods traverse the list without acquiring locks. The add() and remove() methods must still lock the $\texttt{pred}_A$ and $\texttt{curr}_A$ entries just as before (Figures 10.16 and 10.17), but validation no longer requires retraversing the entire list (Figure 10.15) to determine whether an entry is in the set. Instead, because an entry must be marked before being physically removed, validation need only check that $\texttt{curr}_A$ has not been marked. How-

ever, as Figure 10.19 shows, for insertion and deletion, since $\texttt{pred}_A$ is the one being modified, one must also check that $\texttt{pred}_A$ itself is not marked, and that that $\texttt{pred}_A \rightarrow \texttt{curr}_A$. The contains() method (Figure 10.18) traverses nodes as in the optimistic list (possibly traversing logically and physically removed entries) and returns *true* if the entry it was searching for is present and unmarked and *false* otherwise. We can interpret the searched entry being marked as meaning that the value is absent because the invariant that the list is sorted without duplication holds for marked entries also. As in the optimistic algorithm, add() and remove() are not lockout-free, as list traversals may be delayed forever by ongoing modifications.

We must make a small change to the abstraction map. An object is in the set if and only if it is referred to by some *unmarked* reachable entry.

$$\mathcal{S}(\texttt{head}) = \{\text{x}| \text{ exists a such that } \texttt{head} \rightsquigarrow \text{a, a not marked, and a.}\texttt{object} = \text{x}\} \quad (10.12)$$

Note that this statement says that being in the set $\mathcal{S}$ is determined by being unmarked and being reachable from the head along a path that may contain marked nodes.

Since all list modifications and traversals are controlled by exactly the same mechanism as in the optimistic algorithm, and since marking an entry does not violate them, Properties 10.10 and 10.11 hold for the lazy algorithm. The following invariant replaces Property 10.8:

If $\texttt{pred}_A \neq$ **null** and is not marked, then $\texttt{head} \rightsquigarrow \texttt{pred}_A \rightsquigarrow \texttt{tail}$. (10.13)

This property clearly holds at the start of a traversal, when $\texttt{pred}_A$ is head. No add() or contains() method calls can violate the property, because they do not make any entries unreachable. What about remove() calls? Marking an entry does not violate the property, because it does not make any entries unreachable. Neither does setting $\texttt{pred}_A$'s next field to $\texttt{curr}_A$'s successor, because the entry that becomes unreachable is already marked. Property 10.8 justifies why validation does not need to retraverse the entire list, and why contains() does not need to acquire locks: an unmarked reachable entry will remain reachable as long as it remains unmarked.

The linearizability points for add() and unsuccessful remove() of the lazy list are the same as in the optimistic list. A successful remove() method must be linearized when the mark is set in Line , and an successful contains() method must be linearized when an unmarked matching entry is found. Finally, linearizing an unsuccessful contains() methods is a bit tricky, and is

a good example of how, as we explained in Chapter **??**, the choice of linearization point can be non-deterministic. Choosing the linearization point of an entry a as the point where a marked a or an entry greater than a is found will not suffice. Consider the scenario depicted in Figure 10.14 for the optimistic list, but think of the entries as being those of the lazy list. Assume that entry a is marked as removed and thread A is attempting to find the entry matching a's key. While A is traversing the list, $\texttt{curr}_A$ and all entries between $\texttt{curr}_A$ and a including a are removed logically and physically. Thread A would still proceed to the point where $\texttt{curr}_A$ points to a, and, would detect that a is marked and therefor no longer in the list. Linearizing at this point is correct. However, consider what happens if while thread A is traversing the removed section of the list leading to a, and before it reaches the removed a, another thread adds a new entry with a key a to the reachable part of the list (in the figure it would be added by redirecting the "long" arrow to point to it). Linearizing the unsuccessful contains() method at the point it found the marked entry a would be wrong, since it occurs *after* the insertion of the new entry with key a to the list. To remain consistent we must thus linearize an unsuccessful contains() within its execution interval at the earlier of the following points: (1) the point where a removed matching entry is found and (2) the point immediately before a new matching entry is added to the list. As can be seen, this linearization point is non-deterministic since it is determined by the ordering of events in the execution.

The principal disadvantage of the lazy algorithm is that, as in all the earlier lock-based algorithms, the add() and remove() methods are blocking: if one thread is delayed, then other threads may also be delayed. This is a potentially serious problem in a multiprocessor architecture where delays are unpredictable and potentially quite long.

10.8 A Lock-Free List

We have seen that it can be effective to mark entries as logically removed before physically removing them from the list, thus allowing for a lock-free implementation of the contains() method. We now show how to extend this idea to eliminate locks altogether, allowing all three methods, add(), remove(), and contains() to be lock-free. A naive approach would be to use compareAndSet() calls to change the next references. As one could expect, this is problematic. The bottom part of Figure 10.20 shows a thread A attempting to add entry a between entries $\texttt{pred}_A$ and $\texttt{curr}_A$. It will

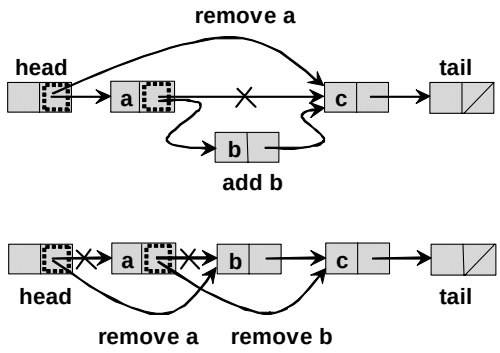

Figure 10.20: *Why we need to modify marks and pointers atomically. In the upper part of the figure Thread A is about to remove a, the first entry in the list, while thread B is about to add b. Suppose A performs a compareAndSet() on* `head.next`*, while B performs a compareAndSet()* `a.next`*. The net effect is that a is correctly deleted but b is not added to the list. In the lower part of the figure, as in the case of fine-grained locks, thread A is about to remove a, the first entry in the list, while thread B is about to remove b, where* $a \rightarrow b$*. Suppose A performs a compareAndSet() on* `head.next`*, while B performs a compareAndSet()* `a.next`*. The net effect is to remove a, but not b.*

set a's next field to $\texttt{curr}_A$, and then call compareAndSet() to set $\texttt{pred}_A$'s next field to a. If B wants to remove $\texttt{curr}_B$ from the list, it might call compareAndSet() to set $\texttt{pred}_B$'s next field to $\texttt{curr}_B$'s successor. It not hard to see that if these two threads try to remove these adjacent entries concurrently, the list would end up with b not being removed. A similar situation for a pair of concurrent add() and remove() methods is depicted in the upper part of Figure 10.20.

Clearly, we need a way to ensure that an entry's fields cannot be updated after that entry has been logically or physically removed from the list. Our approach is to treat the entry's next and marked fields as a single atomic unit: any attempt to update the next field when the marked field is false will fail.

Conveniently, the `java.util.concurrent.atomic` package provides a class with exactly this functionality. An AtomicMarkableReference<T> encapsulates both a reference to an object of type `T` and a Boolean *mark*. These fields can be atomically updated either together, or individually. For example, the compareAndSet() method tests expected reference and mark arguments, and if both tests succeed, replaces them with updated reference

```
public boolean compareAndSet(T expectedReference,
                             T newReference,
                             boolean expectedMark,
                             boolean newMark);
public boolean attemptMark(T expectedReference,
                             boolean newMark);
public T get(boolean marked[]);
```

Figure 10.21: *Some AtomicMarkableReference<T> methods: the* `CompareAndSet` *method tests and updates both the mark and reference fields, while the* `attemptMark` *method updates the mark if the reference field has the expected value. The get() method returns the encapsulated reference and stores the mark at position 0 in the argument array.*

and mark values. As shorthand, the attemptMark() method tests an expected reference value and if the test succeeds, replaces it with a new mark value. Finally, the get() method returns the object's reference value, and stores the mark value in a Boolean array argument. Figure 10.21 illustrates the signatures for these methods.

We change each entry's next field to be an AtomicMarkableReference<Entry>. Thread A logically removes $\texttt{curr}_A$ by setting the mark bit in the entry's next field, and shares the physical removal with other threads performing add() or remove(): as each thread traverses the list, it "cleans up" the list by physically removing (using a compareAndSet()) any marked entries it encounters. In other words, threads performing add() and remove() do not traverse marked entries, they remove them before continuing. The contains() method remains the same as in the lazy algorithm, traversing all entries whether they are marked or not, and testing if an item is in the list based on its key and mark. It is worth pausing to consider two design decisions that differentiate the lock-free algorithm from the lazy algorithm. First, why doesn't the thread that logically deletes an entry also try to physically delete it, and second why can't threads that add or remove entries traverse marked entries without physically removing them? These questions are related.

Suppose that thread A, after logically removing $\texttt{curr}_A$, were to attempt to physically remove it as well. It could do so by calling compareAndSet() to try to redirect $\texttt{pred}_A$'s next field, simultaneously verifying that $\texttt{pred}_A$ is not marked and that it that it refers to $\texttt{curr}_A$. The difficulty is that because A is not holding locks on $\texttt{pred}_A$ and $\texttt{curr}_A$, other threads could insert new nodes or remove $\texttt{pred}_A$ before the compareAndSet() call.

Consider a scenario in which another thread marks $\texttt{pred}_A$. As illustrated in Figure 10.20, we cannot safely redirect the next field of a marked entry, so A would have to restart the physical removal by retraversing the list. This time, however, A would have to physically remove $\texttt{pred}_A$ before it could remove $\texttt{curr}_A$. Even worse, if there is a sequence of logically removed entries leading to $\texttt{pred}_A$, A must remove them all, one after the other, before it can remove $\texttt{curr}_A$ itself.

This example illustrates why add() and remove() cannot traverse marked entries: when they arrive at the entry to be modified, they may be forced to retraverse the list to remove previous marked entries. Instead, both add() and remove() must physically remove any marked entries before proceeding. The contains() method, by contrast, performs no modification, and therefore need not participate in the cleanup of logically removed entries, allowing it, as in the lazy list, to traverse both marked and unmarked entries.

In presenting our lock-free algorithm, we factor out functionality common to the add() and remove() methods by creating a nested Window class to help navigation. As shown in Figure 10.22, a Window object is a simple structure with pred and curr fields, both entries. The Window class's find() method takes a `head` entry and a key a, and traverses the list, seeking to set pred to the entry with the largest key less than a, and curr to the entry with the least key greater than or equal to a. As thread A traverses the list, each time it advances $\texttt{curr}_A$, it checks whether that entry is marked (line 17). If so, it calls compareAndSet() to attempt to physically remove the entry by setting $\texttt{pred}_A$'s next to $\texttt{curr}_A$'s next field. This call tests both the field's reference and Boolean mark values, and it could fail if either value has changed. A concurrent thread could change the mark value by logically removing $\texttt{pred}_A$, or it could change the reference value by physically removing $\texttt{curr}_A$. If the call fails, A restarts the traversal from the head of the list, and otherwise the traversal continues.

The lock-free algorithm uses the same abstraction map as the lazy algorithm:

$$\mathcal{S}(\texttt{head}) = \{\text{x}| \text{ exists a such that } \texttt{head} \rightsquigarrow \text{a, a not marked, and a.}\texttt{object} = \text{x}\} \quad (10.14)$$

The compareAndSet() call at line 10.8 of the find() method is an example of a *benevolent side-effect*: it changes the concrete list without changing the abstract set, because removing a marked entry does not change the value of the abstraction map.

Figure 10.23 shows the lock-free algorithm's add() method. Suppose

```
1   class Window {
2     public Entry pred;
3     public Entry curr;
4     Window(Entry pred, Entry curr) {
5       this.pred = pred; this.curr = curr;
6     }
7   }
8   public Window find(Entry head, int key) {
9     Entry pred = null, curr = null, succ = null;
10    boolean[] marked = {false}; // is curr marked?
11    boolean snip;
12    retry: while (true) {
13      pred = head;
14      curr = pred.next.getReference(); // look for key
15      while (true) {
16        succ = curr.next.get(marked); // remove if marked
17        while (marked[0]) {
18          snip = pred.next.compareAndSet(curr, succ, false, false);
19          if (!snip) continue retry;
20          succ = curr.next.get(marked);
21        }
22        if (curr.key >= key)
23          return new Window(pred, curr);
24        pred = curr;
25        curr = succ;
26      }
27    }
28  }
```

Figure 10.22: *The Window class: the* `find()` *method returns a structure containing the entries on either side of the key. It removes marked entries when it encounters them.*

thread A calls add(a). A uses find() to locate $\texttt{pred}_A$ and $\texttt{curr}_A$. If $\texttt{curr}_A$'s key is equal to a's, the call returns *false*. Otherwise, add() initializes a new entry a to hold a, and sets $a \rightarrow \texttt{curr}_A$. It then calls compareAndSet() (line 14) to set $\texttt{pred}_A \rightarrow a$. Because the compareAndSet() tests both the mark and the reference, it will succeed only if $\texttt{pred}_A$ is unmarked, and if $\texttt{pred}_A \rightarrow \texttt{curr}_A$. If the call succeeds, the method returns *true*, and

```
1  public boolean add(T object) {
2      int key = object.hashCode();
3      boolean splice;
4      while (true) {
5        // find predecessor and curren entries
6        Window window = find(head, key);
7        Entry pred = window.pred, curr = window.curr;
8        // is the key present?
9        if (curr.key == key) {
10         return false;
11       } else {
12         // splice in new entry
13         Entry entry = new Entry(object);
14         entry.next = new AtomicMarkableReference(curr, false);
15         if (pred.next.compareAndSet(curr, entry, false, false)) {
16           return true;
17         }
18       }
19     }
20   }
```

Figure 10.23: *Lock-free add(): uses find() to locate* $\texttt{pred}_A$ *and* $\texttt{curr}_A$*. It Inserts a new entry only if* $\texttt{pred}_A$ *is unmarked.*

otherwise it starts over.

Figure 10.24 shows the lock-free algorithm's remove() method. When A calls remove() to remove object a, it uses find() to locate $\texttt{pred}_A$ and $\texttt{curr}_A$. If $\texttt{curr}_A$'s key fails to match a's, the call returns *false*. Otherwise, remove() calls attemptMark() to mark $\texttt{curr}_A$ as logically removed (line 14). This call will succeed only if no other thread has set the mark first. If it succeeds, the call returns *true*. There is no need to remove the entry physically, because it will be removed by the next thread to traverse that region of the list. If the attemptMark() call fails, remove() starts over from the very beginning.

The lock-free algorithm's contains() method is virtually the same as that of the lazy list, as depicted in Figure 10.18. There is one small change: to test if `curr` is marked we must apply `curr.next.get(marked)` and check that `marked[0]` is true.

All of the correctness arguments of the lazy list carry through for the contains() method, and we therefore concentrate on verifying the correctness

```
1  public boolean remove(T object) {
2    int key = object.hashCode();
3    boolean snip;
4    while (true) {
5      // find predecessor and curren entries
6      Window window = find(head, key);
7      Entry pred = window.pred, curr = window.curr;
8      // is the key present?
9      if (curr.key != key) {
10       return false;
11     } else {
12       // snip out matching entry
13       Entry succ = curr.next.getReference();
14       snip = curr.next.attemptMark(succ, true);
15       if (!snip)
16         continue;
17       pred.next.compareAndSet(curr, succ, false, false);
18       return true;
19     }
20   }
21 }
```

Figure 10.24: *Lock-free List: the remove() method uses find() to locate* $\texttt{pred}_A$ *and* $\texttt{curr}_A$*, and atomically marks the entry for removal.*

of the add() and remove() methods. Property 10.11 is invariant because, as in the lazy list, all the methods maintain freedom from interference. We claim the lock-free algorithm satisfies invariant Property 10.13: if $\texttt{pred}_A$ is not marked, then it is reachable. To see why, first note that if an unmarked entry is reachable, then it will stay reachable for as long is it stays unmarked, because none of the statements that changes the list structure (**??**, 14, and 17) can make an unmarked entry unreachable. We must now check that $\texttt{pred}_A$ cannot not become unmarked and unreachable at line 10.8 when it is set to $\texttt{curr}_A$. If $\texttt{curr}_A$ is marked, then so is the new value of $\texttt{pred}_A$, and the invariant holds trivially. Suppose $\texttt{curr}_A$ is unmarked. We know that $\texttt{pred}_A \rightarrow \texttt{curr}_A$ immediately after either line 22 (if no marked entries were removed) or (if some were). If $\texttt{curr}_A$ is unmarked at line 24 then it was marked before, and since it was both marked and reachable at some earlier instant, it remains reachable. Notice that like $\texttt{pred}_A$, if $\texttt{curr}_A$ is unmarked,

then it is also reachable.

We can now use this invariant to analyze the lock-free add() method. Suppose A tries to add object a. If A observes that $\texttt{curr}_A$'s key is equal to a's, then at some point during the find() call, there was a reachable unmarked entry for a, so the add() returns *false*.

If A successfully executes the compareAndSet() call at line 17, then we know that immediately before that call, $\texttt{pred}_A$ was unmarked and reachable, $\texttt{pred}_A \rightarrow \texttt{curr}_A$, $\texttt{pred}_A$'s key was less than a's key, and $\texttt{curr}_A$'s key was greater. Because the rep invariant guarantees that entries are sorted by key, a was not in the abstract set. Immediately after line **??**, $\texttt{pred}_A$ remains unmarked and reachable, and the new entry is unmarked and reachable, so a is now in the abstract set. The reachability of all other unmarked entries is unchanged, so no other objects have been added to or removed from the set.

Suppose A tries to remove object a. If $\texttt{curr}_A$'s key fails to match a's, then we know that at some point during the call, there was a gap between two reachable unmarked entries where a would have been. So the remove() call returns *false*.

If A successfully executes the attemptMark() call at line 14, then we know that immediately before that call, $\texttt{curr}_A$ was unmarked, and therefore a was present in the set. Immediately afterwards, $\texttt{curr}_A$ is marked, so a is no longer present. The reachability of all other unmarked entries is unchanged, so no other objects have been added to or removed from the set.

When A calls remove() to remove object a, it uses find() to locate $\texttt{pred}_A$ and $\texttt{curr}_A$. If $\texttt{curr}_A$'s key fails to match a's, then we know that at some point during the call, there was gap between two adjacent unmarked entries where a would have been, so a was not present in the set.

Line 14 is the serialization point for a successful remove() call. If A successfully executes the attemptMark() call, then we know that immediately before that call, $\texttt{curr}_A$ was unmarked (hence reachable), with key value matching a's, so a was in the set. Immediately afterwards, $\texttt{curr}_A$ is marked, so a is no longer in the set. As usual, the reachability of all other unmarked entries is unchanged, so no other objects have been added to or removed from the set.

When A calls contains() to find object a, it traverses the list to locate $\texttt{curr}_A$. If $\texttt{curr}_A$'s key matches a's, then we know that at some instant during the call, there was an unmarked reachable entry holding a, so a was in the set. Otherwise, because Property 10.11 is invariant, a was not present in the set.

10.9 Discussion

We have seen a progression of list-based lock implementations in which the granularity and frequency of locking was gradually reduced, eventually reaching a fully lock-free list. The final transition from the lazy list to the lock-free list exposes some of the design decisions that face concurrent programmers.

On the one hand, the lock-free list algorithm guarantees progress in the face of arbitrary delays. However, there is a price for this strong progress guarantee:

- Atomic modification of a reference and a Boolean mark must be supported, a requirement that has an added performance cost in most architectures and languages.[2]

- Traversal operations in the add() and remove() methods must engage in concurrent cleanup of removed entries, introducing the possibility of contention among threads, and forcing threads into costly restarts of their traversal even if there was no change in the vicinity of the entry each was trying to modify.

On the other hand, the lazy lock-based list does not guarantee progress in the face of arbitrary delays: its add() and remove() methods are blocking. However, unlike the lock-free algorithm, it has the benefit that it does not require each entry to include an atomically markable reference. It also does not require traversals to perform cleanup of logically removed entries, they can progress down the list, ignoring if an entry is marked or not.

As in many other aspects of life, there is not a single clear choice of which implementation to use: it depends on the context (application). In the end, the balance of factors such as the potential for arbitrary thread delays, the relative frequency of calls to the add() and remove() methods, the overhead of implementing an atomically markable reference, and so on, determine the choice of whether to lock, and if so at what granularity.

10.10 Exercises

Exercise 10.10.1 Describe how to modify each of the linked list algorithms and invariants if object hash codes are not guaranteed to be unique. 10.5

[2] In the Java Concurrency Package, for example, this cost is reduced somewhat by using a pointer to an intermediate dummy node to signify that the marked bit is set.

Exercise 10.10.2 Explain why the fine-grained locking algorithm is not subject to deadlock.

Exercise 10.10.3 Argue the linearizability of the add() of the fine-grained locking algorithm.

Exercise 10.10.4 Explain why the optimistic and lazy locking algorithms are not subject to deadlock.

Exercise 10.10.5 Show a scenario in the optimistic algorithm where a thread is forever attempting to delete an entry. *Hint*: since we assume that all the individual entry locks are lockout-free, the livelock is not on any individual lock, and a bad execution must repeatedly add and remove nodes from the list.

Exercise 10.10.6 Provide the code for the contains() method missing from the fine-grained algorithm. Explain why your implementation is correct.

Exercise 10.10.7 Is the optimistic list implementation still correct if we switch the order in which add() locks the `prev` and `curr` entries?

Exercise 10.10.8 Show that in the optimistic list algorithm, if $\texttt{pred}_A$ is not *null*, then $\texttt{pred}_A \rightsquigarrow \texttt{tail}$, even if $\texttt{pred}_A$ itself is not reachable.

Exercise 10.10.9 Show that in the optimistic algorithm, the add() method needs to lock only `pred`.

Solution: This can be shown using case analysis. We examine the three different possible overlaps with a call to the add() method:

- Two concurrent add() calls: For two concurrent adds to cause any problems, they must be adding an entry at the same position. Since both threads must lock the predecessor, one thread will have to wait for the other. If one thread is trying to insert an element between entries a and b and another is trying to insert one between b and c, then there is no conflict since b is only modified by the thread inserting between b and c. If a thread is trying to insert after a value that is currently being added by another thread, then it must have validated it successfully in order to succeed, and so the insertion must have completed.

- remove() call overlapping with mAnadd call: First, let's see what happens when the node marked as `curr` in the add() method is being deleted. Then one method would lock the other out since both try to acquire the same `pred` lock. If the node marked as `pred` is the one being deleted, then we are saved again by the locks since the node is being locked as the `pred` in add() and as the currEntry in remove().

- contains() overlaps add(): The contains() method does not cause any problems since it simply goes through the list and checks if it finds an element in the list. By not locking `curr` in add(), contains() is left completely as it was.

Exercise 10.10.10 In the optimistic algorithm, the contains() method locks two entries before deciding whether a key is present. Suppose, instead, it locks no entries, returning *true* if it observes the value, and *false* otherwise.

Explain why this alternative is linearizable, or give a counterexample.

Exercise 10.10.11 Would the lazy algorithm still work if we marked a node as removed simply by setting its next field to **null**? Why or why not? What about the lock-free algorithm?

Solution: The lazy algorithm would not work. The problem is that by removing the node by simply setting its next field to null we lose information - specifically, the rest of the list! This could be modified to work if we make a couple of changes. First, we must save a pointer to the rest of the list after setting the removed node's next field to null. We also need to have a special sentinel node to mark the end of the list instead of using the null pointer. Note that this "fix" is pointless since we aren't really setting the next field to null, we are just copying it to another location.

Not only would the LockFreeList algorithm not work if we remove a node by setting its next field to null, our "solution" above would also not work because of the compareAndSet()() operation. We would need to lock the tail of the list to remedy this problem, but then the list would no longer be lock-free.

Exercise 10.10.12 In the lazy algorithm, can $\texttt{pred}_A$ ever be unreachable? Justify your answer.

Exercise 10.10.13 In the lazy algorithm, explain why validating that `pred` and `curr` are not marked while holding locks, does not imply that `pred` $\rightarrow$ `curr`, and we must test for this property explicitly. (Note that from the code

it would seem that because `pred` is always set to the old value of `curr` and in order for its next pointer to change the new `curr` must first be marked, this will always cause the validation test of `curr` to fail.)

Exercise 10.10.14 Can you modify the remove() of the lazy algorithm so it locks only one entry?

Exercise 10.10.15 In the lock-free algorithm, argue the benefits and drawbacks of having the contains() method help in the cleanup of logically removed entries.

Exercise 10.10.16 In the lock-free algorithm, if an add() method call fails because `pred` does not point to `curr`, but `pred` is not marked, do we need to traverse the list again from `head` in order to attempt to complete the call.

Exercise 10.10.17 Would the contains() method of the lazy and lock-free algorithms still be correct if logically removed entries were not guaranteed to be sorted?

Exercise 10.10.18 The add() method of the lock-free algorithm never finds a marked entry with the same key. Can one modify the algorithm so that it will simply insert its new added object into the existing marked entry with same key if such an entry exists in the list, thus saving the need to insert a new entry?

10.11 Chapter Notes

Lock coupling was invented by Rudolf Bayer and Mario Schkolnick [**?**]. The first designs of lock-free linked-list algorithms are due to John Valois [**?**]. The Lock-free list implementation shown here is a variation on the lists of Maged Michael [**?**], who based his work on earlier linked-list algorithms by Timothy Harris [**?**]. Michael's algorithm is the one used in the Java Concurrency Package. The optimistic algorithm was invented for this chapter, and the lazy algorithm is due to Heller et al. [**?**].