The Art of Multiprocessor Programming

Copyright 2007 Elsevier Inc. All rights reserved

Maurice Herlihy Nir Shavit

October 3, 2007

DRAFT COPY

DRAFT COPY

Contents

1 Introduction **13**
1.1 Shared Objects and Synchronization 16
1.2 A Fable 19
1.2.1 Properties of Mutual Exclusion 21
1.2.2 The Moral 22
1.3 The Producer-Consumer Problem 23
1.4 The Readers/Writers Problem 26
1.5 The Harsh Realities of Parallelization 27
1.6 Missive 29
1.7 Chapter Notes 30
1.8 Exercises 30

Principles **35**

2 Mutual Exclusion **37**
2.1 Time 37
2.2 Critical Sections 38
2.3 Two-Thread Solutions 41
2.3.1 The LockOne Class 41
2.3.2 The LockTwo Class 43
2.3.3 The Peterson Lock 44
2.4 The Filter Lock 46
2.5 Fairness 49
2.6 Lamport's Bakery Algorithm 49
2.7 Bounded Timestamps 51
2.8 Lower Bounds on Number of Locations 56
2.9 Chapter Notes 60
2.10 Exercises 60

DRAFT COPY

3 Concurrent Objects **67**
3.1 Concurrency and Correctness 67
3.2 Sequential Objects 71
3.3 Quiescent Consistency 72
3.3.1 Remarks 74
3.4 Sequential Consistency 75
3.4.1 Remarks 76
3.5 Linearizability 79
3.5.1 Linearization Points 79
3.5.2 Remarks 79
3.6 Formal Definitions 80
3.6.1 Linearizability 81
3.6.2 Linearizability is Compositional 82
3.6.3 The Non-Blocking Property 83
3.7 Progress Conditions 84
3.7.1 Dependent Progress Conditions 85
3.8 The Java Memory Model 86
3.8.1 Locks and Synchronized Blocks 88
3.8.2 Volatile Fields 88
3.8.3 Final Fields 89
3.9 Remarks 90
3.10 Chapter Notes 91
3.11 Exercises 91

4 Foundations of Shared Memory **97**
4.1 The Space of Registers 98
4.2 Register Constructions 104
4.2.1 MRSW Safe Registers 105
4.2.2 A Regular Boolean MRSW Register 105
4.2.3 A regular M-valued MRSW register. 106
4.2.4 An Atomic SRSW Register 109
4.2.5 An Atomic MRSW Register 112
4.2.6 An Atomic MRMW Register 115
4.3 Atomic Snapshots 117
4.3.1 An Obstruction-free Snapshot 117
4.3.2 A Wait-Free Snapshot 119
4.3.3 Correctness Arguments 119
4.4 Chapter Notes 121
4.5 Exercises 122

DRAFT COPY

5 The Relative Power of Primitive Synchronization Operations 133
5.1 Consensus Numbers . . . 134
5.1.1 States and Valence . . . 135
5.2 Atomic Registers . . . 138
5.3 Consensus Protocols . . . 140
5.4 FIFO Queues . . . 141
5.5 Multiple Assignment Objects . . . 145
5.6 Read-Modify-Write Operations . . . 148
5.7 Common2 RMW Operations . . . 150
5.8 The compareAndSet() Operation . . . 152
5.9 Chapter Notes . . . 154
5.10 Exercises . . . 155

6 Universality of Consensus 163
6.1 Introduction . . . 163
6.2 Universality . . . 164
6.3 A Lock-free Universal Construction . . . 165
6.4 A Wait-free Universal Construction . . . 169
6.5 Chapter Notes . . . 176
6.6 Exercises . . . 176

Practice 179

7 Spin Locks and Contention 181
7.1 Welcome to the Real World . . . 181
7.2 Test-and-Set Locks . . . 185
7.3 TAS-Based Spin Locks Revisited . . . 188
7.4 Exponential Backoff . . . 190
7.5 Queue Locks . . . 192
7.5.1 Array-Based Locks . . . 192
7.5.2 The CLH Queue Lock . . . 193
7.5.3 The MCS Queue Lock . . . 197
7.6 A Queue Lock with Timeouts . . . 200
7.7 A Composite Lock . . . 203
7.7.1 A Fast-Path Composite Lock . . . 210
7.8 Hierarchical Locks . . . 212
7.8.1 A Hierarchical Backoff Lock . . . 213
7.8.2 A Hierarchical CLH Queue Lock . . . 214
7.9 One Lock To Rule Them All . . . 220

DRAFT COPY

7.10 Chapter Notes 220
7.11 Exercises 221

8 Monitors and Blocking Synchronization 223
8.1 Introduction 223
8.2 Monitor Locks and Conditions 224
8.2.1 Conditions 225
8.2.2 The Lost-Wakeup Problem 228
8.3 Readers-Writers Locks 231
8.3.1 Simple Readers-Writers Lock 231
8.3.2 Fair Readers-Writers Lock 232
8.4 A Reentrant Lock 234
8.5 Semaphores 237
8.6 Chapter Notes 239
8.7 Exercises 240

9 Linked Lists: the Role of Locking 245
9.1 Introduction 245
9.2 List-based Sets 246
9.3 Concurrent Reasoning 248
9.4 Coarse-Grained Synchronization 250
9.5 Fine-Grained Synchronization 252
9.6 Optimistic Synchronization 257
9.7 Lazy Synchronization 261
9.8 A Lock-Free List 267
9.9 Discussion 272
9.10 Chapter Notes 274
9.11 Exercises 274

10 Concurrent Queues and the ABA Problem 277
10.1 Introduction 277
10.2 Queues 279
10.3 A Bounded Partial Queue 279
10.4 An Unbounded Total Queue 285
10.5 An Unbounded Lock-Free Queue 286
10.6 Memory reclamation and the ABA problem 290
10.6.1 A Naïve Synchronous Queue 294
10.7 Dual Data Structures 296
10.8 Chapter Notes 299
10.9 Exercises 299

DRAFT COPY

11 Concurrent Stacks and Elimination 303
11.1 Introduction . . . 303
11.2 An Unbounded Lock-free Stack . . . 303
11.3 Elimination . . . 304
11.4 The Elimination Backoff Stack . . . 305
11.4.1 A Lock-free Exchanger . . . 308
11.4.2 The Elimination Array . . . 311
11.5 Chapter Notes . . . 314
11.6 Exercises . . . 315

12 Counting, Sorting, and Distributed Coordination 321
12.1 Introduction . . . 321
12.2 Shared Counting . . . 321
12.3 Software Combining . . . 322
12.3.1 Overview . . . 323
12.3.2 An Extended Example . . . 330
12.3.3 Performance and Robustness . . . 333
12.4 Quiescently-Consistent Pools and Counters . . . 333
12.5 Counting Networks . . . 334
12.5.1 Networks that count . . . 334
12.5.2 The Bitonic Counting Network . . . 337
12.5.3 Performance and Pipelining . . . 345
12.6 Diffracting Trees . . . 348
12.7 Parallel Sorting . . . 353
12.8 Sorting Networks . . . 354
12.8.1 Designing a Sorting Network . . . 354
12.9 Sample Sorting . . . 357
12.10Distributed Coordination . . . 360
12.11Chapter Notes . . . 361
12.12Exercises . . . 362

13 Concurrent Hashing and Natural Parallelism 367
13.1 Introduction . . . 367
13.2 Closed-Address Hash Sets . . . 368
13.2.1 A Coarse-Grained Hash Set . . . 371
13.2.2 A Striped Hash Set . . . 373
13.2.3 A Refinable Hash Set . . . 376
13.3 A Lock-free Hash Set . . . 379
13.3.1 Recursive Split-ordering . . . 379
13.3.2 The BucketListclass . . . 383

DRAFT COPY

13.3.3 The LockFreeHashSet<T> class . . . 385
13.4 An Open-Addressed Hash Set . . . 388
13.4.1 Cuckoo Hashing . . . 389
13.4.2 Concurrent Cuckoo Hashing . . . 390
13.4.3 Striped Concurrent Cuckoo Hashing . . . 395
13.4.4 A Refinable Concurrent Cuckoo Hash Set . . . 395
13.5 Chapter Notes . . . 396
13.6 Exercises . . . 396

14 Skiplists and Balanced Search 405
14.1 Introduction . . . 405
14.2 Sequential Skiplists . . . 406
14.3 A Lock-Based Concurrent Skiplist . . . 407
14.3.1 A Bird's Eye View . . . 407
14.3.2 The Algorithm . . . 410
14.4 A Lock-Free Concurrent Skiplist . . . 417
14.4.1 A Bird's Eye View . . . 417
14.4.2 The Algorithm in Detail . . . 420
14.5 Concurrent Skiplists . . . 426
14.6 Chapter Notes . . . 428
14.7 Exercises . . . 429

15 Priority Queues 433
15.1 Introduction . . . 433
15.1.1 Concurrent Priority Queues . . . 434
15.2 An Array-Based Bounded Priority Queue . . . 434
15.3 A Tree-Based Bounded Priority Queue . . . 435
15.4 An Unbounded Heap-Based Priority Queue . . . 439
15.4.1 A Sequential Heap . . . 439
15.4.2 A Concurrent Heap . . . 440
15.5 A Skiplist-Based Unbounded Priority Queue . . . 446
15.6 Chapter Notes . . . 450
15.7 Exercises . . . 451

16 Futures, Scheduling, and Work Distribution 455
16.1 Introduction . . . 455
16.2 Analyzing Parallelism . . . 460
16.3 Realistic Multiprocessor Scheduling . . . 463
16.4 Work Distribution . . . 465
16.4.1 Work Stealing . . . 466

DRAFT COPY

16.4.2 Yielding and Multiprogramming . . . 466
16.5 Work-Stealing Dequeues . . . 467
16.5.1 A Bounded Work-Stealing Dequeue . . . 467
16.5.2 An Unbounded Work-Stealing DEQueue . . . 470
16.5.3 Work Balancing . . . 471
16.6 Chapter Notes . . . 472
16.7 Exercises . . . 473

17 Barriers **493**
17.1 Introduction . . . 493
17.2 Barrier Implementations . . . 495
17.3 Sense-Reversing Barrier . . . 496
17.4 Combining Tree Barrier . . . 497
17.5 Static Tree Barrier . . . 499
17.6 Termination Detecting Barriers . . . 501
17.7 Chapter Notes . . . 507
17.8 Exercises . . . 507

18 Transactional Memory **517**
18.1 Introduction . . . 517
18.1.1 What's Wrong with Locking? . . . 517
18.1.2 What's Wrong with compareAndSet()? . . . 518
18.1.3 What's wrong with Compositionality? . . . 521
18.1.4 What can we do about it? . . . 521
18.2 Transactions and Atomicity . . . 522
18.3 Software Transactional Memory . . . 525
18.3.1 Transactions and Transactional Threads . . . 528
18.3.2 Zombies and Consistency . . . 530
18.3.3 Atomic Objects . . . 531
18.3.4 Dependent or Independent Progress? . . . 532
18.3.5 Contention Managers . . . 534
18.3.6 Implementing Atomic Objects . . . 537
18.3.7 An Obstruction-Free Atomic Object . . . 538
18.3.8 A Lock-Based Atomic Object . . . 542
18.4 Hardware Transactional Memory . . . 546
18.4.1 Cache Coherence . . . 546
18.4.2 Transactional Cache Coherence . . . 547
18.4.3 Enhancements . . . 548
18.5 Chapter Notes . . . 549
18.6 Exercises . . . 549

DRAFT COPY

A Software Basics **563**
A.1 Introduction . . . 563
A.2 Java . . . 563
A.2.1 Threads . . . 563
A.2.2 Monitors . . . 564
A.2.3 Yielding and Sleeping . . . 570
A.2.4 Thread-Local Objects . . . 570
A.3 C# . . . 572
A.3.1 Threads . . . 572
A.3.2 Monitors . . . 573
A.3.3 Thread-Local Objects . . . 576
A.4 Pthreads . . . 577
A.4.1 Thread-Local Storage . . . 579
A.5 The Art of Multiprocessor Programming . . . 579
A.6 Chapter Notes . . . 580

B Hardware Basics **585**
B.1 Introduction (and a Puzzle) . . . 585
B.2 Processors and Threads . . . 588
B.3 Interconnect . . . 589
B.4 Memory . . . 590
B.5 Caches . . . 590
B.5.1 Coherence . . . 591
B.5.2 Spinning . . . 593
B.6 Cache-conscious Programming, or the Puzzle Solved . . . 594
B.7 Multi-Core and Multi-Threaded Architectures . . . 595
B.7.1 Relaxed Memory Consistency . . . 596
B.8 Hardware Synchronization instructions . . . 598
B.9 Chapter Notes . . . 600
B.10 Exercises . . . 600

DRAFT COPY

DRAFT COPY

Chapter 9

Linked Lists: the Role of Locking

9.1 Introduction

In Chapter 7 we saw how to build 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 call acquires and releases that lock. We call this approach *coarse-grained synchronization*.

Often, coarse-grained synchronization works well, but there are important cases where it doesn't. The problem is that a class that uses a single lock to mediate all of its method calls is not always scalable, even if the 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 becomes a sequential bottleneck, forcing threads to wait in line for access.

This chapter introduces several useful 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-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, con-

DRAFT COPY

sist 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.

- *Non-Blocking 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 *items* 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. It is typical that in applications using sets, there are significantly more contains() method calls than add() or remove() calls.

9.2 List-based Sets

This chapter presents a range of concurrent set algorithms, all based on the same basic idea. A set is implemented as a linked list of nodes. As shown

DRAFT COPY

```
1 public interface Set<T> {
2   boolean add(T x);
3   boolean remove(T x);
4   boolean contains(T x);
5 }
```

Figure 9.1: *The Set interface: add() adds an item to the set (no effect if that item is already present), remove() removes it (if present), and contains() returns a Boolean indicating whether the item is present.*

```
1 private class Node {
2   T item;
3   int key;
4   Node next;
5 }
```

Figure 9.2: *The Node<T> class: this internal class keeps track of the item, the item's key, and the next node in the list. Some algorithms require technical changes to this class.*

DRAFT COPY

in Figure 9.2, the Node<T> class has three fields. The item field is the actual item of interest. The key field is the items's hash code. Nodes are sorted in key order, providing an efficient way to detect when an item is absent. The next field is a reference to the next node in the list. (Some of the algorithms we consider require technical changes to this class, such as adding new fields, or changing the types of existing fields.) For simplicity, we assume that each item's hash code is unique (relaxing this assumption is left as an exercise). We associate an item with the same node and key throughout any given example, which allows us to abuse notation and use the same symbol to refer to a node, its key, and its item. That is, node a may have key a and item a, and so on.

The list has two kinds of nodes. In addition to *regular* nodes that hold items in the set, we use two *sentinel* nodes, called head and tail , as the first and last list elements. Sentinel nodes are never added, removed, or searched for, and their keys are the minimum and maximum integer values.[1] Ignoring synchronization for the moment, the top part of Figure 9.3 shows a schematic

[1] All algorithms presented here work for any any ordered set of keys that have maximum and minimum values and that are well-founded, that is, there are only finitely many keys smaller than any given key. For simplicity, we assume here that keys are integers.

description how an item is added to the set. Each thread A has two local variables used to traverse the list: curr_A is the current node and pred_A is its predecessor. To add an item to the set, A sets local variables pred_A and curr_A to head, and moves down the list, comparing curr_A's key to the key of the item being added. If they match, the item is already present in the set, so the call returns *false*. If pred_A precedes curr_A in the list, then pred_A's key is lower than that of the inserted item, and curr_A's key is higher, so the item is not present in the list. The method creates a new node b to hold the item, sets b's nextA field to curr_A, then sets pred_A to b. Removing an item from the set works in a similar way.

9.3 Concurrent Reasoning

Reasoning about concurrent data structures may seem impossibly difficult, but it is a skill that can be learned. 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:

1. The property holds when the object is created, and

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

Most interesting invariants hold trivially when the list is created, so it makes sense to focus on how invariants, once established, are preserved.

Specifically, we can 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 nodes, a property sometimes called freedom from *interference*. In the list algorithms considered here, nodes are internal to the list implementation, so freedom from interference is guaranteed because users of the list have no opportunity to modify its internal nodes.

We require freedom from interference even for nodes that have been removed from the list, since some of our algorithms permit a thread to unlink a node while it is being traversed by others. Fortunately, we do not attempt to reuse list nodes that have been removed from the list, relying instead on a garbage collector to recycle that memory. The algorithms described here work in languages without garbage collection, but sometimes require non-trivial modifications beyond the scope of this chapter.

DRAFT COPY

When reasoning about concurrent object implementations, it is important to understand the distinction between an object's *abstract value* (here, a set of items), and its *concrete representation* (here, a list of nodes).

Not every list of nodes is a meaningful representation for a set. An algorithm's *representation invariant* characterizes which representations make sense as abstract values. If a and b are nodes, we say that *a points to b* if a's next field is a reference to b. We say that b is *reachable* if there is a sequence of nodes, starting at head, and ending at b, where each node in the sequence points to its successor.

The set algorithms in this chapter require the following invariants (some require more, as explained later). First, sentinels are neither added nor removed. Second, nodes are sorted by key, and keys are unique.

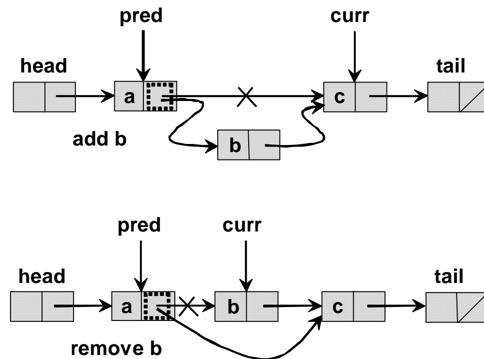

Figure 9.3: *A seqential Set implementation: adding and removing nodes. To insert a node b, a thread uses two variables: curr is the current node, and pred is its predecessor. Move down the list comparing the keys for curr and b. If a match is found, the item is already present, so return false. If curr reaches an node with a higher key, the item is not present Set b's next field to curr, and pred's next field to b. To delete curr, set pred's next field to curr's next field.*

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 also 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 representation invariant, which set does it represent? The meaning of such a list is given by an *abstraction map* carrying lists that satisfy the representation invariant to sets. Here, the abstraction map is simple: an item is in the set if and only if it is reachable from head.

What safety and liveness properties do we need? Our safety property is *linearizability*. As we saw in Chapter 3, to show that a concurrent data structure is a linearizable implementation of a sequentially specified object, it is enough to identify a *linearization point*, a single atomic step where the method call "takes effect". This step can be a read, a write, or a more complex atomic operation. Looking at any execution history of a list-based set, 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 set execution. 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 whether a was already in the set.

Different list algorithms make different progress guarantees. Some use locks, and care is required to ensure they are deadlock and starvation-free. Some *non-blocking* list algorithms do not use locks at all, while others restrict locking to certain methods. Here is a brief summary, from Chapter 3, of the non-blocking properties we use[2]:

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

We are now ready to consider a range of list-based set algorithms. We start with algorithms that use coarse-grained synchronization, and successively refine them to reduce granularity of locking. Formal proofs of correctness lie beyond the scope of this book. Instead, we focus on informal reasoning useful in everyday problem-solving.

As mentioned, in each of these algorithms, methods scan through the list using two local variables: curr is the current node and pred is its predecessor. These variables are thread-local[3], so we use pred_A and curr_A to denote the instances used by thread A.

9.4 Coarse-Grained Synchronization

We start with a simple algorithm using coarse-grained synchronization. Figures 9.4 and 9.5 show the add() and remove() methods for this coarse-grained

[2]Chapter 3 introduces an even weaker non-blocking property called *obstruction-freedom*.

[3]Appendix A describes how thread-local variables work in Java.

DRAFT COPY

```
1  public class CoarseList<T> {
2    private Node head;
3    private Lock lock = new ReentrantLock();
4    public CoarseList() {
5      head  = new Node(Integer.MIN_VALUE);
6      head.next = new Node(Integer.MAX_VALUE);
7    }
8    public boolean add(T item) {
9      Node pred, curr;
10     int key = item.hashCode();
11     lock.lock();
12     try {
13       pred = head;
14       curr = pred.next;
15       while (curr.key < key) {
16         pred = curr;
17         curr = curr.next;
18       }
19       if (key == curr.key) {
20         return false;
21       } else {
22         Node node = new Node(item);
23         node.next = curr;
24         pred.next = node;
25         return true;
26       }
27     } finally {
28       lock.unlock();
29     }
30   }
```

Figure 9.4: *The CoarseList class: the add() method.*

DRAFT COPY

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 acquire. The principal advantage of this algorithm, which should not be discounted, is that it is obviously correct. All methods act on the list only while holding the lock, so the execution is essentially sequential. To simplify matters, we follow the convention (for now) that the linearization point for

```
31   public boolean remove(T item) {
32     Node pred, curr;
33     int key = item.hashCode();
34     lock.lock();
35     try {
36       pred = head;
37       curr = pred.next;
38       while (curr.key < key) {
39         pred = curr;
40         curr = curr.next;
41       }
42       if (key == curr.key) {
43         pred.next = curr.next;
44         return true;
45       } else {
46         return false;
47       }
48     } finally {
49       lock.unlock();
50     }
51   }
```

Figure 9.5: *The CoarseList class: the remove() method: all methods acquire a single lock, which is released on exit by the* **finally** *block.*

DRAFT COPY

any method call that acquires a lock is the instant the lock is acquired.

The CoarseList class satisfies the same progress condition as its lock: if the Lock is starvation-free, so is our implementation. If contention is very low, this algorithm is an excellent way to implement a list. If, however, there is contention, then even if the lock itself performs well, threads will still be delayed waiting for one another.

9.5 Fine-Grained Synchronization

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

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock ();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock ();
8       try {
9         while (curr.key < key) {
10          pred.unlock ();
11          pred = curr;
12          curr = curr.next;
13          curr.lock ();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock ();
24      }
25    } finally {
26      pred.unlock ();
27    }
28  }
```

Figure 9.6: *The* FineList *class: the* add() *method uses hand-over-hand locking to traverse the list. The* **finally** *blocks release locks before returning.*

DRAFT COPY

the list together in a pipelined fashion.

Consider two nodes a and b where a points to b. It is not safe to unlock a before locking b because another thread could remove b from the list in the interval between unlocking a and locking b. Instead, a Thread A must acquire locks in a kind of "hand-over-hand" order: except for the initial head sentinel node, acquire the lock for curr_A only while holding the lock for pred_A. This locking protocol is sometimes called *lock coupling*. (No-

```
29 public boolean remove(T item) {
30   Node pred = null, curr = null;
31   int key = item.hashCode();
32   head.lock ();
33   try {
34     pred = head;
35     curr = pred.next;
36     curr.lock ();
37     try {
38       while (curr.key < key) {
39         pred.unlock ();
40         pred = curr;
41         curr = curr.next;
42         curr.lock ();
43       }
44       if (curr.key == key) {
45         pred.next = curr.next;
46         return true;
47       }
48       return false;
49     } finally {
50       curr.unlock ();
51     }
52   } finally {
53     pred.unlock ();
54   }
55 }
```

Figure 9.7: *The FineList class: the remove() method locks both the node to be removed and its predecessor before removing that node.*

DRAFT COPY

tice that there is no obvious way to implement lock coupling using Java's **synchronized** methods.)

Figure 9.7 shows the FineList algorithm's remove() method. Just as in the coarse-grained list, remove() makes curr_A unreachable by setting pred_A's next field to curr_A's successor. To be safe, remove() must lock both pred_A and curr_A. To see why, consider the following scenario, illustrated in Figure 9.5. Thread A is about to remove node a, the first node in the list, while Thread B is about to remove node b, where a points to b. Suppose A locks head,

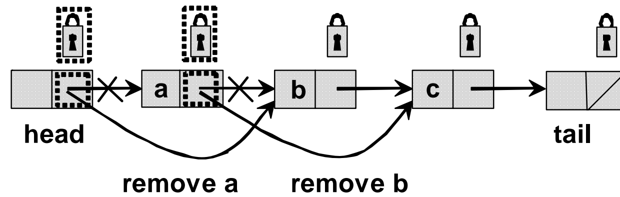

Figure 9.8: *The FineList class: why remove() must acquire two locks. Thread A is about to remove a, the first node in the list, while Thread B is about to remove b, where a points to b. Suppose A locks head, and B locks a. Thread A then sets head.next to b, while B sets a's next field to c. The net effect is to remove a, but not b.*

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.

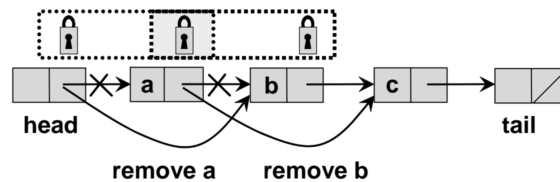

Figure 9.9: *The FineList class: Hand-over-hand locking ensures that if concurrent remove() calls try to remove adjacent nodes, then they acquire conflicting locks. Thread A is about to remove node a, the first node in the list, while Thread B is about to remove node b, where a points to b. Because A 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.*

To guarantee progress, it is important that all methods acquire locks in the same order, starting at head and following next references toward the tail . As Figure 9.10 shows, a deadlock could occur if different method calls were to acquire locks in different orders. In this example, thread A, trying to add a, has locked b and is attempting to lock head, while B, trying to remove b, has locked b and is trying to lock head. Clearly, these method calls will never finish. Avoiding deadlocks is one of the principal challenges of programming with locks.

The FineList algorithm maintains the representation invariant: sentinels are never added or removed, and nodes are sorted by key value without duplicates. The abstraction map is the same as for the course-grained list: an item is in the set if and only if its node is reachable.

The linearization point for an add(a) call depends on whether the call was successful (that is, whether a was already present). A successful call (a absent) is linearized when the node containing a is locked (either Line 7 or 13).

The same distinctions apply to remove(a) calls. A successful call (a present) is linearized when the predecessor node is locked (Lines 36 or 42. A successful call (a absent) is linearized when the node containing the next higher key is locked (Lines 36 or 42. An unsuccessful call (a present) is linearized when the node containing a is locked.

Determining linearization points for contains() is left as an exercise.

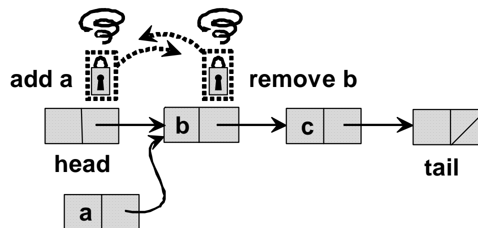

Figure 9.10: *The FineList class: a deadlock can occur if, for example, remove() and add() calls acquire locks in opposite order. Thread A is about to insert a by locking first b and then head, and Thread B is about to remove node b by locking first head and then b. Each thread holds the lock the other is waiting to acquire, so neither makes progress.*

The FineList algorithm is starvation-free, but arguing this property is harder than in the course-grained case. We assume that all individual locks are starvation-free. Because all methods acquire locks in the same down-the-list order, deadlock is impossible. If Thread A attempts to lock head, eventually it succeeds. From that point on, because there are no deadlocks, eventually all locks held by threads ahead of A in the list will be released, and A will succeed in locking pred_A and curr_A.

DRAFT COPY

```
1  private boolean validate(Node pred, Node curr) {
2     Node node = head;
3     while (node.key <= pred.key) {
4        if (node == pred)
5           return pred.next == curr;
6        node = node.next;
7     }
8     return false;
9  }
```

Figure 9.11: *The OptimisticList: validation checks that $pred_A$ points to $curr_A$ and is reachable from head.*

9.6 Optimistic Synchronization

Although fine-grained locking is an improvement over a single, coarse-grained lock, it still imposes a potentially long sequence of lock acquisitions and releases. Moreover, threads accessing disjoint parts of the list may still block one another. For example, a thread removing the second item in the list blocks all concurrent threads searching for later nodes.

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

In Figure 9.12, thread A makes an optimistic add(a). It traverses the list without acquiring any locks (Lines 15 through 17). In fact, it ignores the locks completely. It stops the traversal when $curr_A$'s key is greater than or equal to a's. It then locks $pred_A$ and $curr_A$, and calls validate() to check that $pred_A$ is reachable and its next field still refers to $curr_A$. If validation succeeds, then A proceeds as before: if $curr_A$'s key is greater than a, A adds a new node with item a between $pred_A$ and $curr_A$, and returns *true*. Otherwise it returns *false*. The remove() and contains() methods (Figures 9.13 and 9.14) operate similarly, traversing the list without locking, then locking the target nodes and validating they are still in the list. The following story illustrates the nature of optimistic traversal.

> A tourist takes a taxi in a foreign town. The taxi driver speeds through a red light. The tourist, frightened, asks "What are you are doing?" The driver answers: "Don't worry, I am

DRAFT COPY

```
10    public boolean add(T item) {
11      int key = item.hashCode();
12      while (true) {
13        Node pred = head;
14        Node curr = pred.next;
15        while (curr.key <= key) {
16          pred = curr; curr = curr.next;
17        }
18        pred.lock(); curr.lock();
19        try {
20          if (validate(pred, curr)) {
21            if (curr.key == key) {
22              return false;
23            } else {
24              Node node = new Node(item);
25              node.next = curr;
26              pred.next = node;
27              return true;
28            }
29          }
30        } finally {
31          pred.unlock(); curr.unlock();
32        }
33      }
34    }
```

Figure 9.12: *The OptimisticList class: the add() method traverses the list ignoring locks, acquires locks, and validates before adding the new node.*

DRAFT COPY

> an expert." He speeds through more red lights, and the tourist, on the verge of hysteria, complains again, more urgently. The driver replies, "Relax, relax, you are in the hands of an expert." Suddenly, the light turns green, the driver slams on the brakes, and the taxi skids to a halt. The tourist, picks himself off the floor of the taxi and asks "For crying out loud, why stop now that the light is finally green?" The driver answers "Too dangerous, could be another expert coming".

Traversing any dynamically-changing lock-based data structure while ignoring locks requires careful thought (there are other expert threads out

```
35  public boolean remove(T item) {
36    int key = item.hashCode();
37    while (true) {
38      Node pred = head;
39      Node curr = pred.next;
40      while (curr.key < key) {
41        pred = curr; curr = curr.next;
42      }
43      pred.lock(); curr.lock();
44      try {
45        if (validate(pred, curr)) {
46          if (curr.key == key) {
47            pred.next = curr.next;
48            return true;
49          } else {
50            return false;
51          }
52        }
53      } finally {
54        pred.unlock(); curr.unlock();
55      }
56    }
57  }
```

Figure 9.13: *The OptimisticList class: the remove() method traverses ignoring locks, acquires locks, and validates before removing the node.*

DRAFT COPY

there). We must make sure to use some form of *validation* and guarantee freedom from *interference*.

As Figure 9.15 shows, validation is necessary because the trail of references leading to pred_A or the reference from pred_A to curr_A could have changed between when they were last read by A and when A acquired the locks. In particular, a thread could be traversing parts of the list that have already been removed. For example, the node curr_A and all nodes between curr_A and a (including a) may be removed while A is still traversing curr_A. Thread A discovers that curr_A points to a, and, without validation, "successfully" removes a, even though a is no longer in the list. A validate() call detects that a is no longer in the list, and the caller restarts the method.

Because we are ignoring the locks that protect concurrent modifications,

```
58 public boolean contains(T item) {
59     int key = item.hashCode();
60     while (true) {
61       Entry pred = this.head; // sentinel node;
62       Entry curr = pred.next;
63       while (curr.key < key) {
64         pred = curr; curr = curr.next;
65       }
66       try {
67         pred.lock(); curr.lock();
68         if (validate(pred, curr)) {
69           return (curr.key == key);
70         }
71       } finally {                  // always unlock
72         pred.unlock(); curr.unlock();
73       }
74     }
75   }
```

Figure 9.14: *The OptimisticList class: the contains() method searches, ignoring locks, then it acquires locks, and validates to determine if the node is in the list.*

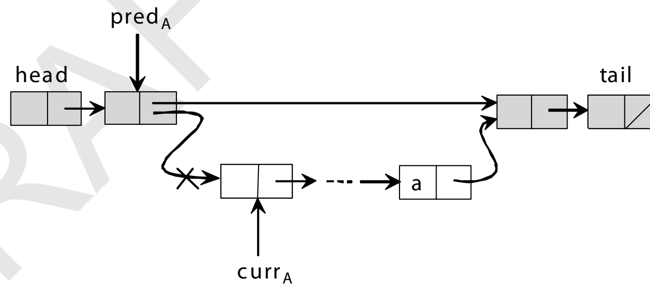

Figure 9.15: *The OptimisticList class: why validation is needed. Thread A is attempting to remove a node a. While traversing the list, $curr_A$ and all nodes between $curr_A$ and a (including a) might be removed (denoted by a lighter node color). In such a case, thread A would proceed to the point where $curr_A$ points to a, and, without validation, would successfully remove a, even though it is no longer in the list. Validation is required to determine that a is no longer reachable from head.*

DRAFT COPY

each of the method calls may traverse nodes that have been removed from the list. Nevertheless, absence of interference implies that once a node has been unlinked from the list, the value of its next field does not change, so following a sequence of such links eventually leads back to the list. Absence of interference, in turn, relies on garbage collection to ensure that no node is recycled while it is being traversed.

The OptimisticList algorithm is not starvation-free even if all node locks are individually starvation-free. A thread might be delayed forever if new nodes are repeatedly added and removed (see Exercise 107). Nevertheless, we would expect this algorithm to do well in practice, since starvation is rare.

9.7 Lazy Synchronization

```
1  private boolean validate(Node pred, Node curr) {
2    return !pred.marked && !curr.marked && pred.next == curr;
3  }
```

Figure 9.16: *The LazyList class: validation checks that neither the pred nor the curr nodes has been logically deleted, and that pred points to curr.*

The OptimisticList 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 this particular algorithm is that contains() acquires locks, which is unattractive since contains() calls are likely to be much more common than calls to other methods.

The next step is to refine this algorithm so that contains() calls are wait-free, and add() and remove() methods, while still blocking, traverse the list only once (in the absence of contention). We add to each node a Boolean marked field indicating whether that node is in the set. Now, traversals do not need to lock the target node, and there is no need to validate that the node is reachable by retraversing the whole list. Instead, the algorithm maintains the invariant that every unmarked node is reachable. If a traversing thread does not find a node, or finds it marked, then the that item is not in the set. As a result, contains() needs only one wait-free traversal. To add an element to the list, add() traverses the list, locks the target's predecessor, and inserts the node. The remove() method is lazy, taking two steps: first, mark the target node, *logically* removing it, and second, redirect its predecessor's next field, *physically* removing it.

DRAFT COPY

```
1  public boolean add(T item) {
2    int key = item.hashCode();
3    while (true) {
4      Node pred = head;
5      Node 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) {
15             return false;
16           } else {
17             Node node = new Node(item);
18             node.next = curr;
19             pred.next = node;
20             return true;
21           }
22         }
23       } finally {
24         curr.unlock();
25       }
26     } finally {
27       pred.unlock();
28     }
29   }
30 }
```

Figure 9.17: *The LazyList class: add() method.*

In more detail, all methods traverse the list (possibly traversing logically and physically removed nodes) ignoring the locks. The add() and remove() methods lock the pred_A and curr_A nodes as before (Figures 9.17 and 9.18), but validation does not retraverse the entire list (Figure 9.16) to determine whether a node is in the set. Instead, because a node must be marked before being physically removed, validation need only check that curr_A has

DRAFT COPY

```
1  public boolean remove(T item) {
2    int key = item.hashCode();
3    while (true) {
4      Node pred = head;
5      Node 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) {
15             return false;
16           } else {
17             curr.marked = true;
18             pred.next = curr.next;
19             return true;
20           }
21         }
22       } finally {
23         curr.unlock();
24       }
25     } finally {
26       pred.unlock();
27     }
28   }
29 }
```

Figure 9.18: *The LazyList class: the remove() method removes nodes in two steps, logical and physical.*

DRAFT COPY

not been marked. However, as Figure 9.20 shows, for insertion and deletion, since $pred_A$ is the one being modified, one must also check that $pred_A$ itself is not marked, and that that it points to $curr_A$. Logical removals requires a small change to the abstraction map: an item is in the set if and only if it is referred to by an *unmarked* reachable node. Notice that the path along which the node is reachable may contain marked nodes. The reader

```
1   public boolean contains(T item) {
2     int key = item.hashCode();
3     Node curr = head;
4     while (curr.key < key)
5       curr = curr.next;
6     return curr.key == key && !curr.marked;
7   }
```

Figure 9.19: *The LazyList class: the contains() method*

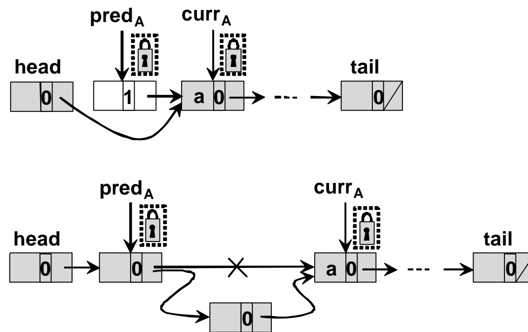

Figure 9.20: *The LazyList class: why validation is needed. In the top part of the figure, thread A is attempting to remove node a. After it reaches the point where $pred_A$ refers to $curr_A$, and before it acquires locks on these nodes, the node $pred_A$ is logically and physically removed. After A acquires the locks, validation will detect the problem. In the bottom part of the figure, A is attempting to remove node a. After it reaches the point where $pred_A$ equals $curr_A$, and before it acquires locks on these nodes, a new node is added between $pred_A$ and $curr_A$. After A acquires the locks, even though neither $pred_A$ or $curr_A$ are marked, validation detects that $pred_A$ is not the same as $curr_A$, and A's call to remove() will be restarted*

should check that any unmarked reachable node remains reachable even if its predecessor is logically or physically deleted. As in the OptimisticList algorithm, add() and remove() are not starvation-free, because list traversals may be arbitrarily delayed by ongoing modifications.

The contains() method (Figure 9.19) traverses the list once ignoring locks and returns *true* if the node it was searching for is present and unmarked,

DRAFT COPY

and *false* otherwise. It is thus is wait-free.[4] A marked node's value is ignored. This method is wait-free. Each time the traversal moves to a new node, the new node has a larger key than the previous one, even if the node is logically deleted.

Logical removal require a small change to the abstraction map: an item is in the set if and only if it is referred to by an *unmarked* reachable node. Notice that the path along which the node is reachable may contain marked nodes. Physical list modifications and traversals occur exactly as in the OptimisticList class, and the reader should check that any unmarked reachable node remains reachable even if its predecessor are logically or physically deleted.

The linearization points for LazyList add() and unsuccessful remove() calls are the same as for OptimisticList. A successful remove() call is linearized when the mark is set (Line 17), and a successful contains() call is linearized when an unmarked matching node is found.

To understand how to linearize an unsuccessful contains(), consider the scenario depicted in Figure 9.21. In part (a), node a is marked as removed (its marked field is set) and Thread A is attempting to find the node matching a's key. While A is traversing the list, curr_A and all nodes between curr_A and a including a are removed, both logically and physically. Thread A would still proceed to the point where curr_A points to a, and would detect that a is marked and no longer in the abstract set. The call could be linearized at this point.

Now consider the scenario depicted in part (b). While A is traversing the removed section of the list leading to a, and before it reaches the removed node a, another thread adds a new node with a key a to the reachable part of the list. Linearizing Thread A's unsuccessful contains() method at the point it found the marked node a would be wrong, since this point occurs *after* the insertion of the new node with key a to the list. We therefore linearize an unsuccessful contains() method call within its execution interval at the earlier of the following points: (1) the point where a removed matching node, or a node with a key greater than the one being searched for, is found, and (2) the point immediately before a new matching node is added to the list. Notice that the second is guaranteed to be within the execution interval because the insertion of the new node with the same key must have happened after the start of the contains() method, or the contains() method would have found that item. As can be seen, the linearization point of

[4]Notice that the list ahead of a given traversing thread cannot grow forever due to newly inserted keys since key size is finite.

DRAFT COPY

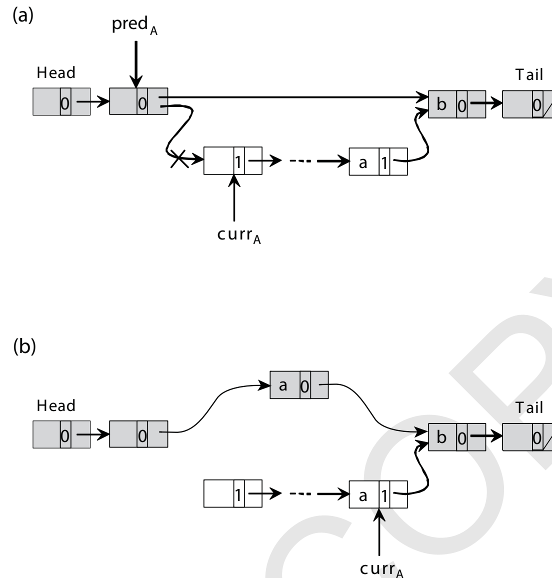

Figure 9.21: *The LazyList class: linearizing an unsuccessful contains() call. Dark nodes are physically in the list and white nodes are physically removed. In part (a), while thread A is traversing the list, a concurrent remove() call disconnects the sublist referred to by curr. Notice that nodes with items a and b are still reachable, so whether an item is actually in the list depends only on whether it is marked. A's call is linearized at the point when it sees that a is marked and is no longer in the abstract set. Alternatively consider the scenario depicted in part (b). While A is traversing the list leading to marked node a, another thread adds a new node with key a. It would be wrong to linearize A's unsuccessful contains() call to when it found the marked node a, since this point occurs* after *the insertion of the new node with key a to the list.*

DRAFT COPY

the unsuccessful contains() is determined by the ordering of events in the execution, and is not a predetermined point in the method's code.

One benefit of lazy synchronization is that we can separate unobtrusive logical steps, such as setting a flag, from disruptive physical changes to the structure, such as physically removing a node. The example presented here is simple because we physically remove one node at a time. In general, however, delayed operations can be batched and performed lazily at a convenient time, reducing the overall disruptiveness of physical modifications to the structure.

The principal disadvantage of the LazyList algorithm is that add() and remove() calls are blocking: if one thread is delayed, then others may also

be delayed.

9.8 A Lock-Free List

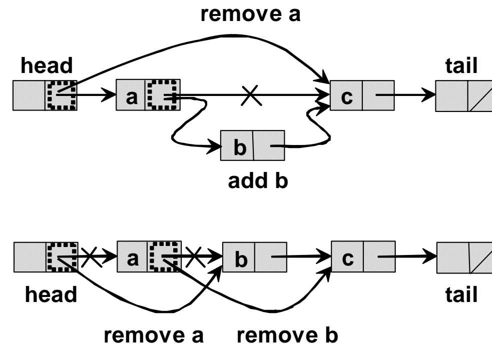

Figure 9.22: *The LazyList class: why mark and reference fields must be modified atomically. In the upper part of the figure, Thread A is about to remove a, the first node in the list, while B is about to add b. Suppose A applies compareAndSet() to head.next, while B applies compareAndSet() to a.next. The net effect is that a is correctly deleted but b is not added to the list. In the lower part of the figure, thread A is about to remove a, the first node in the list, while B is about to remove b, where a points to b. Suppose A applies compareAndSet() to head.next, while B applies compareAndSet() to a.next. The net effect is to remove a, but not b.*

We have seen that it is sometimes a good idea to mark nodes as logically removed before physically removing them from the list. We now show how to extend this idea to eliminate locks altogether, allowing all three methods, add(), remove(), and contains(), to be non-blocking. (The first two methods are lock-free and the last wait-free). A naïve approach would be to use compareAndSet() to change the next fields. Unfortunately, this idea does not work. The bottom part of Figure 9.22 shows a Thread A attempting to add node a between nodes pred_A and curr_A. It sets a's next field to curr_A, and then calls compareAndSet() to set pred_A's next field to a. If B wants to remove curr_B from the list, it might call compareAndSet() to set pred_B's next field to curr_B's successor. It not hard to see that if these two threads try to remove these adjacent nodes 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 9.22.

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

Pragma 9.8.1. An AtomicMarkableReference<T> object, from java.util.concurrent.atomic, encapsulates both a reference to an object of type T and a Boolean mark. These fields can be updated atomically, either together or individually. For example, the compareAndSet() method tests the expected reference and mark values, and if both tests succeed, replaces them with updated reference 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. The get() method has an unusual interface: it returns the object's reference value and stores the mark value in a Boolean array argument. Figure 9.23 illustrates the interfaces of these methods.

In C or C++, one could provide this functionality efficiently by "stealing" a bit from a pointer, using bit-wise operators to extract the mark and the pointer from a single word. In Java, of course, one cannot manipulate pointers directly, so this functionality must be provided by a library.

```
1  public boolean compareAndSet(T expectedReference,
2                               T newReference,
3                               boolean expectedMark,
4                               boolean newMark);
5  public boolean attemptMark(T expectedReference,
6                             boolean newMark);
7  public T get(boolean[] marked);
```

Figure 9.23: *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.*

As described in detail in Pragma 9.8.1, an AtomicMarkableReference<<>T¿ object encapsulates both a reference to an object of type T and a Boolean mark. These fields can be atomically updated either together or individually.

We make each node's next field an AtomicMarkableReference<Node>. Thread A logically removes curr_A by setting the mark bit in the node's next field, and shares the physical removal with other threads performing add() or

DRAFT COPY

remove(): as each thread traverses the list, it cleans up the list by physically removing (using compareAndSet()) any marked nodes it encounters. In other words, threads performing add() and remove() do not traverse marked nodes, they remove them before continuing. The contains() method remains the same as in the LazyList algorithm, traversing all nodes 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 a design decision that differentiates the LockFreeList algorithm from the LazyList algorithm. Why do threads that add or remove nodes never traverse marked nodes, and instead physically remove all marked nodes they encounter? Suppose that Thread A were to traverse marked nodes without physically removing them, and after logically removing curr_A, were to attempt to physically remove it as well. It could do so by calling compareAndSet() to try to redirect pred_A's next field, simultaneously verifying that pred_A is not marked and that it that it refers to curr_A. The difficulty is that because A is not holding locks on pred_A and curr_A, other threads could insert new nodes or remove pred_A before the compareAndSet() call.

Consider a scenario in which another thread marks pred_A. As illustrated in Figure 9.22, we cannot safely redirect the next field of a marked node, so A would have to restart the physical removal by retraversing the list. This time, however, A would have to physically remove pred_A before it could remove curr_A. Even worse, if there is a sequence of logically removed nodes leading to pred_A, A must remove them all, one after the other, before it can remove curr_A itself.

This example illustrates why add() and remove() calls do not traverse marked nodes: when they arrive at the node to be modified, they may be forced to retraverse the list to remove previous marked nodes. Instead, we choose to have both add() and remove() physically remove any marked nodes on the path to their target node. The contains() method, by contrast, performs no modification, and therefore need not participate in the cleanup of logically removed nodes, allowing it, as in the LazyList, to traverse both marked and unmarked nodes.

In presenting our LockFreeList algorithm, we factor out functionality common to the add() and remove() methods by creating an inner Window class to help navigation. As shown in Figure 9.24, a Window object is a structure with pred and curr fields. The Window class's find() method takes a head node and a key a, and traverses the list, seeking to set pred to the node with the largest key less than a, and curr to the node with the least key greater than or equal to a. As Thread A traverses the list, each time it

DRAFT COPY

```
1   class Window {
2     public Node pred, curr;
3     Window(Node myPred, Node myCurr) {
4       pred = myPred; curr = myCurr;
5     }
6   }
7   public Window find(Node head, int key) {
8     Node pred = null, curr = null, succ = null;
9     boolean[] marked = {false};
10    boolean snip;
11    retry: while (true) {
12      pred = head;
13      curr = pred.next.getReference();
14      while (true) {
15        succ = curr.next.get(marked);
16        while (marked[0]) {
17          snip = pred.next.compareAndSet(curr, succ, false, false);
18          if (!snip) continue retry;
19          curr = succ;
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 9.24: *The* Window *class: the* find() *method returns a structure containing the nodes on either side of the key. It removes marked nodes when it encounters them.*

DRAFT COPY

advances curr_A, it checks whether that node is marked (Line 16). If so, it calls compareAndSet() to attempt to physically remove the node by setting pred_A's next to curr_A's next field. This call tests both the field's reference and Boolean mark values, and fails if either value has changed. A concurrent thread could change the mark value by logically removing pred_A, or it could change the reference value by physically removing curr_A. If the call fails, A

restarts the traversal from the head of the list, and otherwise the traversal continues.

The LockFreeList algorithm uses the same abstraction map as the LazyList algorithm: an item is in the set if and only if it is in an *unmarked* reachable node. The compareAndSet() call at Line 17 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 node does not change the value of the abstraction map.

Figure 9.25 shows the LockFreeList classm's add() method. Suppose Thread A calls add(a). A uses find() to locate pred_A and curr_A. If curr_A's key is equal to a's, the call returns *false*. Otherwise, add() initializes a new node a to hold a, and sets a to refer to curr_A. It then calls compareAndSet() (Line 10) to set pred_A to a. Because the compareAndSet() tests both the mark and the reference, it succeeds only if pred_A is unmarked and refers to curr_A. If the compareAndSet() is successful, the method returns *true*, and otherwise it starts over.

```
1  public boolean add(T item) {
2    int key = item.hashCode();
3    while (true) {
4      Window window = find(head, key);
5      Node pred = window.pred, curr = window.curr;
6      if (curr.key == key) {
7        return false;
8      } else {
9        Node node = new Node(item);
10       node.next = new AtomicMarkableReference(curr, false);
11       if (pred.next.compareAndSet(curr, node, false, false)) {
12         return true;
13       }
14     }
15   }
16 }
```

Figure 9.25: *The LockFreeList class: the add() method calls find() to locate pred_A and curr_A. It adds a new node only if pred_A is unmarked and refers to curr_A.*

Figure 9.26 shows the LockFreeList algorithm's remove() method. When A calls remove() to remove item a, it uses find() to locate pred_A and curr_A. If curr_A's key fails to match a's, the call returns *false*. Otherwise, remove()

DRAFT COPY

```
17   public boolean remove(T item) {
18     int key = item.hashCode();
19     boolean snip;
20     while (true) {
21       Window window = find(head, key);
22       Node pred = window.pred, curr = window.curr;
23       if (curr.key != key) {
24         return false;
25       } else {
26         Node succ = curr.next.getReference();
27         snip = curr.next.attemptMark(succ, true);
28         if (!snip)
29           continue;
30         pred.next.compareAndSet(curr, succ, false, false);
31         return true;
32       }
33     }
34   }
```

Figure 9.26: *The LockFreeList class: the remove() method calls find() to locate $pred_A$ and $curr_A$, and atomically marks the node for removal.*

calls attemptMark() to mark $curr_A$ as logically removed (Line 27). This call succeeds only if no other thread has set the mark first. If it succeeds, the call returns *true*. A single attempt is made to physically remove the node, but there is no need to try again because the node will be removed by the next thread to traverse that region of the list. If the attemptMark() call fails, remove() starts over.

The LockFreeList algorithm's contains() method is virtually the same as that of the LazyList (Figure 9.27). 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*.

9.9 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 non-blocking list. The final transition from the LazyList to the LockFreeList exposes some of the design decisions that face concurrent

DRAFT COPY

```
35    public boolean contains(T item) {
36      boolean[] marked = false{};
37      int key = item.hashCode();
38      Node curr = head;
39      while (curr.key < key) {
40        curr = curr.next;
41        Node succ = curr.next.get(marked);
42      }
43      return (curr.key == key && !marked[0])
44    }
```

Figure 9.27: *The LockFreeList class: the wait-free contains() method is the almost the same as in the LazyList class. There is one small difference: it calls curr.next.get(marked) to test whether curr is marked.*

programmers.

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

- The need to support atomic modification of a reference and a Boolean mark has an added performance cost.[5]
- As add() and remove() traverse the list, they must engage in concurrent cleanup of removed nodes, introducing the possibility of contention among threads, sometimes forcing threads to restart traversals, even if there was no change near the node 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 does not require each node to include an atomically markable reference. It also does not require traversals to clean up logically removed nodes; they progress down the list, ignoring marked nodes.

Which approach is preferable depends on the 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.

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

DRAFT COPY

9.10 Chapter Notes

Lock coupling was invented by Rudolf Bayer and Mario Schkolnick [17]. The first designs of lock-free linked-list algorithms are due to John Valois [142]. The Lock-free list implementation shown here is a variation on the lists of Maged Michael [112], who based his work on earlier linked-list algorithms by Tim Harris [50]. Michael's algorithm is the one used in the Java Concurrency Package. The OptimisticList algorithm was invented for this chapter, and the lazy algorithm is due to Heller et al. [52].

9.11 Exercises

Exercise 103. Describe how to modify each of the linked list algorithms if object hash codes are not guaranteed to be unique. 9.5

Exercise 104. Explain why the fine-grained locking algorithm is not subject to deadlock.

Exercise 105. Explain why the fine-grained list's add() method is linearizable.

Exercise 106. Explain why the optimistic and lazy locking algorithms are not subject to deadlock.

Exercise 107. Show a scenario in the optimistic algorithm where a thread is forever attempting to delete an node. *Hint*: since we assume that all the individual node locks are starvation-free, the livelock is not on any individual lock, and a bad execution must repeatedly add and remove nodes from the list.

Exercise 108. Provide the code for the contains() method missing from the fine-grained algorithm. Explain why your implementation is correct.

Exercise 109. Is the optimistic list implementation still correct if we switch the order in which add() locks the pred and curr entries?

Exercise 110. Show that in the optimistic list algorithm, if pred_A is not *null*, then tail is reachable from pred_A, even if pred_A itself is not reachable.

Exercise 111. Show that in the optimistic algorithm, the add() method needs to lock only pred.

Exercise 112. 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.

Either explain why this alternative is linearizable, or give a counterexample showing it is not.

DRAFT COPY

Exercise 113. 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?

Exercise 114. In the lazy algorithm, can pred_A ever be unreachable? Justify your answer.

Exercise 115. Your new employee claims that the lazy list's validation method (Figure 9.16) can be simplified by dropping the check that pred.next is equal to curr. After all, the code always sets pred to the old value of curr, and before pred.next can be changed, the new value of curr must be marked, causing the validation to fail. Explain the error in this reasoning.

Exercise 116. Can you modify the lazy algorithm's remove() so it locks only one node?

Exercise 117. In the lock-free algorithm, argue the benefits and drawbacks of having the contains() method help in the cleanup of logically removed entries.

Exercise 118. 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 119. 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 120. The add() method of the lock-free algorithm never finds a marked node with the same key. Can one modify the algorithm so that it will simply insert its new added object into the existing marked node with same key if such an node exists in the list, thus saving the need to insert a new node?

Exercise 121. Explain why it cannot happen in the LockFreeList algorithm that a node with item x will be logically but not yet physically removed by some thread, then the same item x will be added into the list by another thread, and finally a contains() call by a third thread will traverse the list, finding the logically removed node, and returning false, even though the linearization order of the remove() and add() implies that x is in the set.

DRAFT COPY

Appendix

DRAFT COPY

Bibliography

DRAFT COPY

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

[3] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 538–547, New York, NY, USA, 1995. ACM Press.

[4] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot (extended abstract). In *Symposium on Principles of Distributed Computing*, pages 71–80, 2000.

[5] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 159–170, New York, NY, USA, 1993. ACM Press.

[6] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 396–406, May 1989.

[7] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. *ACM SIGPLAN Notices*, 34(10):207–222, 1999.

[8] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. *Combinatorica*, 3:1–19, 1983.

[9] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, Atlantic City, NJ, April 1967. AFIPS Press, Reston, VA.

[10] James H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.

[11] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 184–193, New York, NY, USA, 1995. ACM Press.

[12] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[13] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129. ACM Press, 1998.

[14] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[15] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: Featherweight synchronization for java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998.

[16] K. Batcher. Sorting networks and their applications. In *Proceedings of AFIPS Joint Computer Conference*, pages 338–334, 1968.

[17] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

[18] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[19] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 261–268, New York, NY, USA, 2005. ACM Press.

DRAFT COPY

[20] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 41–51, New York, NY, USA, 1993. ACM Press.

[21] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[22] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 222–231, New York, NY, USA, 1987. ACM Press.

[23] Costas Busch and Marios Mavronicolas. A combinatorial treatment of balancing networks. *J. ACM*, 43(5):794–839, 1996.

[24] Tushar Deepak Chandra, Prasad Jayanti, and King Tan. A polylog time wait-free construction for closed objects. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 287–296, New York, NY, USA, 1998. ACM Press.

[25] Graham Chapman, John Cleese, Terry Gilliam, Eric Idle, Terry Jones, and Michael Palin. Monty phyton and the holy grail, 1975.

[26] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM Press.

[27] Alonzo Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1936.

[28] Intel Corporation. *Pentium Processor User's Manual*. Intel Books, 1993. ISB: 1555121934.

[29] T. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Department of Computer Science, February 1993.

[30] Dave Dice and Nir Shavit. Transactional locking ii. In *20th International Symposium on Distributed Computing*, September 2006.

DRAFT COPY

[31] David Dice. Implementing fast java monitors with relaxed-locks. In *Java Virtual Machine Research and Technology Symposium*, pages 79–90, 2001.

[32] Edsger W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[33] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM Journal of Computing*, 26(2):418–455, 1997.

[34] Martin Dowd, Yehoshua Perl, Larry Rudolph, and Michael Saks. The periodic balanced sorting network. *J. ACM*, 36(4):738–757, 1989.

[35] Arthur Conan Doyle. *A Study in Scarlet and the Sign of Four*. Berkley Publishing Group, 1994. ISBN 0425102408.

[36] Cynthia Dwork and Orli Waarts. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *Journal of the ACM (JACM)*, 46(5):633–666, 1999.

[37] C. Ellis. Concurrency in linear hashing. *ACM Transactions on Database Systems (TODS)*, 12(2):195–217, 1987.

[38] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[39] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proc. of the Java TM Virtual Machine Research and Technology Symposium*, 2001.

[40] K. Fraser. *Practical Lock-Freedom*. Ph.D. dissertation, Kings College, University of Cambridge, Cambridge, England, September 2003.

[41] B. Gamsa, O. Kreiger, E.W. Parsons, and M. Stumm. Performance issues for multiprocessor operating systems. Technical report, Computer Systems Research Institute, University of Toronto, 1995.

[42] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1):21–42, 2005.

DRAFT COPY

[43] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 64–75. ACM Press, 1989.

[44] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification,*. Prentice Hall PTR, third edition edition, 2005. ISBN 0321246780.

[45] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[46] Michael Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using dcas. In *21st ACM Symposium on Principles of Distributed Computing*, pages 260–269, July 2002.

[47] S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM*, 42(1):186–203, 1995.

[48] Sibsankar Haldar and Paul Vitányi. Bounded concurrent timestamp systems using vector clocks. *Journal of the ACM (JACM)*, 49(1):101–126, 2002.

[49] Per Brinch Hansen. Structured multi-programming. *Communications of the ACM*, 15(7):574–578, 1972.

[50] Tim Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of 15th International Symposium on Distributed Computing (DISC 2001), Lisbon, Portugal*, volume 2180 of *Lecture Notes in Computer Science*, pages 300—314. Springer Verlag, October 2001.

[51] Tim Harris, Simon Marlowe, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming (PPOPP)*, 2005.

[52] Steven Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, Nir Shavit, and William N Scherer III. Lazy concurrent list-based set algorithm. In *9th International Conf. of Principles of Distributed Systems (OPODIS 2005)*, December 2005.

DRAFT COPY

[53] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 280–289. ACM Press, 2002.

[54] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM Press.

[55] J.L. Hennessy and D.A. Patterson. *Computer Architecture: An Quantitative Approach*. Morgan Kaufmann Publishers, 1995.

[56] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 0885-7458 1988.

[57] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[58] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE, 2003.

[59] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[60] Maurice Herlihy, Yossi Lev, and Nir Shavit. A lock-free concurrent skiplist with wait-free search, 2007.

[61] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, 1995.

[62] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.

[63] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the*

DRAFT COPY

20th annual international symposium on Computer architecture, pages 289–300. ACM Press, 1993.

[64] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Concurrent cuckoo hashing. Technical report, Brown University, 2007.

[65] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[66] C. A. R. Hoare. "partition: Algorithm 63," "quicksort: Algorithm 64," and "find: Algorithm 65.". *Communications of the ACM*, 4(7):321–322, 1961.

[67] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[68] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[69] M. Hsu and W. Yang. Concurrent operations in extendible hashing. In *Symposium on very large data bases*, pages 241–247, 1986.

[70] J. Huang and Y. Chow. Cc-radix: A cache conscious sorting based on radix sort. *In Proc. of the 7th Computer Software and Applications Conference*, page 627631, 1983.

[71] J.S. Huang and Y.C. Chow. Parallel sorting and data partitioning by sampling. In *Proceedings of the IEEE Computer Society's Seventh International Computer Software and Applications Conference*, pages 627–631, 1983.

[72] Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Process. Lett.*, 60(3):151–157, 1996.

[73] A. Israeli and L. Rappaport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 14–17 1994. Los Angeles, CA.

[74] Amos Israeli and Ming Li. Bounded time stamps. *Distributed Computing*, 6(5):205–209, 1993.

DRAFT COPY

[75] Amos Israeli and Ming Li. Bounded time-stamps. *Distrib. Comput.*, 6(4):205–209, 1993.

[76] Amos Israeli and Amnon Shaham. Optimal multi-writer multi-reader atomic register. In *Symposium on Principles of Distributed Computing*, pages 71–82, 1992.

[77] Mohammed Gouda James Anderson, Ambuj Singh. The elusive atomic register. Technical Report TR 86.29, University of Texas at Austin, 1986.

[78] Prasad Jayanti. Robust wait-free hierarchies. *J. ACM*, 44(4):592–614, 1997.

[79] Prasad Jayanti. A lower bound on the local time complexity of universal constructions. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 183–192, New York, NY, USA, 1998. ACM Press.

[80] Prasad Jayanti and Sam Toueg. Some results on the impossibility, universality, and decidability of consensus. In *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 69–84, London, UK, 1992. Springer-Verlag.

[81] Lefteris M. Kirousis, Paul G. Spirakis, and Philippas Tsigas. Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. In *Workshop on Distributed Algorithms*, pages 229–241, 1991.

[82] M. R. Klugerman. Small-depth counting networks and related topics. Technical Report MIT/LCS/TR-643, MIT Laboratory for Computer Science, 1994.

[83] Michael Klugerman and C. Greg Plaxton. Small-depth counting networks. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 417–428, New York, NY, USA, 1992. ACM Press.

[84] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms, Volume 3.* Addison-Wesley, 1973.

[85] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Transactions*

DRAFT COPY

on Programming Languages and Systems (TOPLAS), 10(4):579–601, 1988.

[86] V. Kumar. Concurrent operations on extendible hashing and its performance. *Communications of the ACM*, 33(6):681–694, 1990.

[87] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[88] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(5):543–545, 1974.

[89] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[90] Leslie Lamport. Time, clocks, and the ordering of events. *Communications of the ACM*, 21(7):558–565, July 1978.

[91] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690, September 1979.

[92] Leslie Lamport. Invited address: Solved problems, unsolved problems and non-problems in concurrency. In *Proceedings of the Third Annual Acm Symposium on Principles of Distributed Computing*, pages 1–11, 1984.

[93] Leslie Lamport. The mutual exclusion problem: part ia theory of interprocess communication. *Journal of the ACM (JACM)*, 33(2):313–326, 1986.

[94] Leslie Lamport. The mutual exclusion problem: part ii statement and solutions. *Journal of the ACM (JACM)*, 33(2):327–348, 1986.

[95] Butler Lampson and David Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 2(23):105–117, 1980.

[96] James R. Larus and Ravi Rajwar. *Transactional Memory.* Morgan and Claypool, 2006.

[97] D. Lea. Concurrent hash map in JSR166 concurrency utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`.

[98] Doug Lea, 2007.

DRAFT COPY

[99] Douglas Lea. Java community process, jsr 166, concurrency utilities. `http://www.jcp.org/en/jsr/`, 2003.

[100] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, and Andrew Sohn. Partitioned parallel radix sort. *J. Parallel Distrib. Comput.*, 62(4):656–668, 2002.

[101] C. Leiserson and H. Prokop. A minicourse on multithreaded programming, 1998.

[102] Y. Lev, M. Herlihy, V. Luchangco, and N. Shavit. A provably correct scalable skiplist (brief announcement). In *Proc. of the 10th International Conference On Principles Of Distributed Systems (OPODIS 2006)*, 2006.

[103] Ming Li, John Tromp, and Paul M. B. Vitányi. How to share concurrent wait-free variables. *J. ACM*, 43(4):723–746, 1996.

[104] Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: wait-free hierarchies are not robust. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 579–588, New York, NY, USA, 1997. ACM Press.

[105] I. Lotan and N. Shavit. Skiplist-based concurrent priority queues. In *Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–268, 2000.

[106] Victor Luchangco, Daniel Nussbaum, and Nir Shavit. A hierarchical clh queue lock. In *Euro-Par*, pages 801–810, 2006.

[107] P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing (IPPS)*, pages 165–171. IEEE Computer Society, April 1994.

[108] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[109] Paul E. McKenney. Selecting locking primitives for parallel programming. *Commun. ACM*, 39(10):75–82, 1996.

DRAFT COPY

[110] John Mellor-Crummey and Michael Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[111] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[112] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM Press, 2002.

[113] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM Press, 1996.

[114] Jaydev Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(1):142–153, 1986.

[115] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228, New York, NY, USA, 1997. ACM Press.

[116] Mark Moir. Laziness pays! using lazy synchronization mechanisms to improve non-blocking constructions. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 61–70, New York, NY, USA, 2000. ACM Press.

[117] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 253–262, New York, NY, USA, 2005. ACM Press.

[118] Richard Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 232–248, New York, NY, USA, 1987. ACM Press.

DRAFT COPY

[119] Isaac Newton, I. Bernard Cohen (Translator), and Anne Whitman (Translator). *The Principia : Mathematical Principles of Natural Philosophy.* University of California Press, 1999.

[120] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[121] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[122] Gary Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[123] Gary L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

[124] S. A. Plotkin. Sticky bits and universality of consensus. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 159–175, New York, NY, USA, 1989. ACM Press.

[125] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, 1989.

[126] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *ACM Transactions on Database Systems*, 33(6):668–676, 1990.

[127] Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. In *DISC*, pages 108–121, 2005.

[128] Zoran Radović and Erik Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Ninth International Symposium on High Performance Computer Architecture*, pages 241–252, Anaheim, California, USA, February 2003.

[129] John H. Reif and Leslie G. Valiant. A logarithmic time sort for linear size networks. *J. ACM*, 34(1):60–76, 1987.

[130] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *In Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245. ACM Press, July 1991.

DRAFT COPY

[131] Michael Saks, Nir Shavit, and Heather Woll. Optimal time randomized consensus — making resilient algorithms fast in practice. In *SODA '91: Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 351–362, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[132] Michael L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 31–40, New York, NY, USA, 2002. ACM Press.

[133] Michael L. Scott and William N. Scherer. Scalable queue-based spin locks with timeout. *ACM SIGPLAN Notices*, 36(7):44–52, 2001.

[134] Maurice Sendak. *Where the Wild Things Are.* Publisher: Harper-Collins, 1988. ISBN: 0060254920.

[135] Ori Shalev and Nir Shavit. Split-ordered lists: lock-free extensible hash tables. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 102–111. ACM Press, 2003.

[136] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. ACM Press, 1995.

[137] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.

[138] Eric Shenk. The consensus hierarchy is not robust. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, page 279, New York, NY, USA, 1997. ACM Press.

[139] Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register. *J. ACM*, 41(2):311–339, 1994.

[140] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[141] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. Lond. Math. Soc*, 1937.

DRAFT COPY

[142] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM Press, 1995.

[143] Paul Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In *In 27th Annual Symposium on Foundations of Computer Science*, pages 233–243, Los Angeles, Ca., USA, October 1986. IEEE Computer Society Press.

[144] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(2):249–282, 1989.

[145] III William N. Scherer, Doug Lea, and Michael L. Scott. Scalable synchronous queues. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 147–156, New York, NY, USA, 2006. ACM Press.

[146] III William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM Press.

[147] P. Yew, N. Tzeng, and D. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

DRAFT COPY