Errata for

The Art of Multiprocessor Programming

Version of 10 February 2009

In many places, inserted text is highlighted in red.

Preface

p. xx "all of which are useful in structuring concurrent applications."

Chapter 1

p. 4 In Fig. 1.1, line 4 should say

```
  for (int j = (i * block) + 1; j <= (i + 1)* block;
j++) {
```

p. 11 Second paragraph of Mutual Exclusion bullet: "Initially the can is either up or down. Let us say it was down. Then only the pets can go in, and mutual exclusion holds."

p. 18 Replace question 6 bullet 2 with:

- Suppose the method ***M*** accounts for 30% of the program's computation time. Let s_n be the program's speedup on ***n*** processes, assuming the rest of the program is perfectly parallelizable. Your boss tells you to double this speedup: the revised program should have speedup $s'_n \geq s_n/2$. You advertize for a programmer to replace ***M*** with an improved version, ***k*** times faster. What value of ***k*** should you require?

Chapter 2

p. 23 Figure 2.3 should be amended as shown:

```
public long getAndIncrement() {
     lock.lock();
     try {
         long temp = value;
         value = temp + 1;
         return temp;
     } finally {
         lock.unlock();
     }
 }
```

p. 25 Pragma 2.3.1: "We explain the reasons in Chapter 3 and Appendix B."

p. 26 In Fig. 2.5, remove all declarations of variables as *volatile*. Indeed as stated in Pragma 2.3.1 on page 25, one should use memory barriers when implementing these algorithms. However, in this chapter (in all code) we avoid such issues to keep the algorithms simple. Adding volatile to these variables would require more complex coding that would distract readers from the core issues at hand.

p. 35 In the second paragraph the line:
"Suppose A's token is on Node 0, and B's token on Node 1 (so A has the later timestamp)" should read "Suppose A's token is on Node 1, and B's token is on Node 0 (so A has the later timestamp)".

p. 27 In Fig. 2.6, remove all declarations of variables as *volatile*. Indeed as stated in Pragma 2.3.1 on page 25, one should use memory barriers when implementing these algorithms. However, in this chapter (in all code) we avoid such issues to keep the algorithms simple. Adding volatile to these variables would require more complex coding that would distract readers from the core issues at hand.

p. 44 Exercise 19. Instead of 3^n should be $O(3^n)$ and instead of $n2^n$ should say $O(n2^n)$.

Chapter 3

p. 54. First paragraph: "We use the following shorthand: <p.enq(x) A> -> <p.deq(x) B> means that any sequential execution must order A's enqueue of x at p before B's dequeue of x at p, and so on."

p. 67 "... head is the index of the next slot from which to remove an item, and tail is the index of the next slot in which to place an item."

Chapter 4

p.77 In the table of Figure 4.5 in the second line MRMW Boolean Regular should be MRSW Boolean Regular.

p.84 In Fig.4.12, add a new line between 22 and 23 Line

```
if (i == me) continue;
```

In addition, Line 32 should be

```
a_table [ 0 ][ i ] = value;
```

p. 88 Should be "when B's snapshot was taken before A started its Scan() call".

p. 91 In Fig 4.21 Line 20 should be "**return** `newCopy[j].snap;`

p. 90 In Fig 4.19 Line 11 should be "`stamp = label;`"

p. 94 In Exercise 40 " Does Peterson's two thread mutual exclusion algorithm work if the shared atomic `flag` registers are replaced by regular registers".

p. 95. In Figure 4.22, the comment "`N is the total number of threads`" is incorrect. `N` is actually the length of the register.

Chapter 5

P 100 "The reader will notice that since the decide() method of a given consensus object is executed only once by each thread, **and that there are a finite number of threads**, by definition, a lock-free implementation would also be wait-free and vice versa."

p 121. In Exercise 65, Replace "provides the same propose() and decide() method as consensus" by "provides the same decide() method as consensus".

Chapter 6

p 131 In Figure 6.6 line 17

```
Node after = before.decideNext.decide(prefer);
```

p 134 In Figure 6.8 line 17

```
Node after = before.decideNext.decide(prefer);
```

p 137 "Exercise 80. Propose a way to fix the universal construction of Figure 6.8 to work for objects with nondeterministic sequential specifications."

Chapter 7

p. 148 In Figure7.5 Line 7 should be:

```
maxDelay = max;
```

p. 153 Fig. 7.9 omit line 6

p. 156 First paragraph, the reference to Pragma 8.2.3 should be to Pragma 8.2.2.

p. 161 Line 12 should be Line 9

p. 162 In Fig. 7.21, Line 6 is redundant and can be omitted.

p. 163 In Fig. 7.22, remove Line 6.
In Fig. 7.23, remove Line 4.

p 174 Exercise 86 should be:

> First barrier implementation: We have a counter protected by a test-and-test-and-set lock. Each thread locks the counter, increments it, releases the lock, and spins, rereading the counter until it reaches n
>
> Second barrier implementation: We have an n-element array b, all 0. Thread zero sets b[0] to 1. Every thread i, for 0 < i <= n-1, spins until b[i-1] is 1, sets b[i] to 1, and waits until b[i+1] becomes 2, at which point it proceeds to leave the barrier. Thread b[n-1], upon detecting that b[n-1] is 1, sets b[n-1] to 2 and leaves the barrier.
>
> Compare (in ten lines) the behavior of these two implementations on a bus-based cache-coherent architecture. Explain which approach you expect will perform better under low load and high load.

p. 175 In Exercise 86, "Second barrier implementation: We have an n-element boolean array b, all false. Thread zero sets b[0] to true. Every thread i, for 0 < i <= n, spins until b[i-1] is true, sets b[i] to true, and waits until b[n-1] is true"

p. 175 In Exercise 89, Remove "Notice that" and begin with "In the HCLHLock lock…".

Chapter 8

p 182 The constructor for LockedQueue should be:

```
public LockedQueue(int capacity) {
  items = (T[])new Object[capacity];
}
```

p. 185 In Figure 8.9, line 48 should be:

```
      while (readers > 0 || writer) {
```

p. 187. Figure 8.11 should be:

```
private class ReadLock implements Lock {
  public void lock() {
    lock.lock();
    try {
      while (writer) {
        condition.await();
      }
```

```
          readAcquires++;
        } finally {
          lock.unlock();
        }
      }
      public void unlock() {
        lock.lock();
        try {
          readReleases++;
          if (readAcquires == readReleases)
            condition.signalAll();
        } finally {
          lock.unlock();
        }
      }
    }
```

Figure 8.12 should be

```
  private class WriteLock implements Lock {
    public void lock() {
      lock.lock();
      try {
        while (writer) {
          condition.await();
        }
        writer = true;
        while (readAcquires != readReleases) {
          condition.await();
        }
      } finally {
        lock.unlock();
      }
    }
    public void unlock() {
      writer = false;
      condition.signalAll();
    }
}
```

Chapter 9

p. 213. Replace:

Fig. 9.22 shows a Thread A attempting to add node a between nodes predA and currA. It sets a's next field to currA, and then calls compareAndSet() to set predA's next field to a. If B wants to remove currB from the list, it might call CompareAndSet() to set predB's next field to currB's successor. It is 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 Fig. 9.22.

with:

In Fig. 9.22, part (a) shows a Thread A attempting to remove node node a while Thread B is adding a node 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 part (b) of the figure, Thread A attempts 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.

If B wants to remove currB from the list, it might call compareAndSet() to set predB's next field to currB's successor. It is 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 Fig. 9.22.

p. 201. Last paragraph: "except for the initial head sentinel node, acquire the lock for a node only while holding the lock for its predecessor"

p.204 Figure 9.9 caption: ``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."

p.217 In Line 16 "calls attemptMark() to mark currA as logically removed (Line 27)" should be "uses a compareAndSet() to attempt to mark currA as logically removed (Line 27)".

p.217 In Line 20 "If the attemptMark() call fails, remove() starts over." should be replaced by "If the compareAndSet() call fails, remove() starts over."

p. 218 In Figure 9.26, Line 27, replace "27 snip = curr.next.attemptMark(succ,true);" by "27 snip = curr.next.compareAndSet(succ, succ, false, true);"

p. 221 In Exercise 118 "Expalin why the following cannot happen…"

Chapter 10

p. 224 Paragraph 4: "Pools provide different fairness guarantees. They can be first-in-first-out (a queue), last-in-first-out (a stack),"

p. 226 Figure 10.4 should be

```
public T deq() {
 T result;
 boolean mustWakeEnqueuers = false;
 deqLock.lock();
 try {
   while (size.get() == 0)
     notEmptyCondition.await();
   result = head.next.value;
   head = head.next;
   if (size.getAndDecrement() == capacity) {
     mustWakeEnqueuers = true;
```

```
      }
    } finally {
      deqLock.unlock();
    }
    if (mustWakeEnqueuers) {
      enqLock.lock();
      try {
        notFullCondition.signalAll();
      } finally {
        enqLock.unlock();
      }
    }
    return result;
  }
```

Fig. 10.3, line 24 should be replaced by:

```
tail.next = e;
tail = tail.next;
```

p. 227 "The deq() method proceeds as follows. It reads the size field to check whether the queue is empty. If so, the dequeuer must wait until an item is enqueued."

p. 228 Paragraph 2: "The dequeuer then decrements size and releases deqLock."

p. 234 In Fig. 10.14, "Threads B and C: enq a, c, and d."

Chapter 11

p. 246 In paragraph 1, "push()" should be "pop()"

p. 246 In Figure 11.1 part (b) the dashed line emanating from *top* should be solid and the solid line directed from *top* to *A* should be dashed (dashed lines symbolize past states).

p. 251 "The waiting thread will consume the item and reset the state to EMPTY. Resetting to EMPTY can be done using a simple write"

p. 255 The LockFreeStack is credited to Treiber [145]. The EliminationBackoffStack is due to Danny Hendler, Nir Shavit, and Lena Yerushalmi.

Chapter 12

p. 259 "Interestingly, for some data structures based on distributed coordination, high throughput does not necessarily mean low latency."

p. 264 5 lines from the top, "Fig. 12.6" should be "in Part (a) of Fig. 12.3"

p. 264 "In Line 20, the thread waits until the locked field is false"

p. 273 Caption should read "as depicted in Fig. 12.11". Also "The gray `Merger[4]` network has as inputs the even wires coming out of the top `Bitonic[4]` network, and the odd ones from the lower `Bitonic[4]` network."

p. 275 In Fig. 12.15, the `traverse` method is missing a line at the end:

```
return (2 * output) + layer[output].traverse();
```

p. 276 In Fig. 12.16, replace Line 17 with:

```
return merger.traverse((input >= (size / 2) ?
```

```
(size / 2) : 0) + output);
  }
```

P. 292 8 lines from the bottom of the page, "Beng-Hong Lim et al." should be "Maurice Herlihy, Beng-Hong Lim, and Nir Shavit"

Chapter 13

p. 317	Fig. 13.21 should be
	`public boolean add(T x) {`
	`  if (contains(x)) {`
	`    return false;`
	`  }`
	`  for (int i = 0; i < LIMIT; i++) {`
	`    if ((x = swap(0, hash0(x), x)) == null) {`
	`      return true;`
	`    } else if ((x = swap(1, hash1(x), x)) == null) {`
	`      return true;`
	`    }`
	`  }`
	`  resize();`
	`  Add(x);`
	`}`
	The caption to Fig. 13.22 should say:
	A sequence of displacements started when an item with key 14 finds both locations Table[0][h_0 (14)] and Table[1][h_1(14)] taken by the values} 3 and 25, …

Chapter 14

p. 331. Last paragraph: "as in the LazyList class…"

p. 337. In Figure 14.7, Line 102 should be
`if (ismarked ||`

p. 343 In Figure 14.11, Line 53 is redundant and can be omitted.

p. 344 In Figure 14.12, Line 87, replace
" `nodeToRemove.next[level].attemptMark(succ, true);`"
with
"`nodeToRemove.next[level].compareAndSet(succ, succ, false, true);`"

p. 345 Lines 4 and 5 should end with "applying a `compareAndSet()`" instead of "applying `attemptMark()`"

p. 345 Each mention of `victim` should be `nodeToRemove`.

p. 347 Figure 14.14, Line 145 should read:
`curr = curr.next[ level ]. getReference();`

Chapter 15

p.362 Fig 15.12 (c). Node with the value 10 occurs twice, while node 7 is absent. The bottom 10 should be 7 instead (the state before the two swaps).

p.364 Fig 15.13, drop the curly brackets in Lines 19 and 24, so the lines will read

```
If (!curr.marked.get()) {
   if(curr.marked.compareAndSet(false,true))
     return curr;
   }
   else {
      curr=curr.next[0].getReference();
   }
```

Chapter 16

p. 370 In Fig 16.1, line 5. `ymA` should be `myA`

p. 372 Last paragraph, "Line 19" should be "Line 20 of Fig. 16.4".

p. 383 In Fig. 16.10, The `isEmpty()` method should be:

```
boolean isEmpty() {
  int localTop = top.getReference();
  int localBottom = bottom;
  return localBottom <= localTop;
}
```

p. 386 Paragraph 4 should reference corrected Fig. 16.10 (above). "On the other hand if bottom is greater than top"

p. 393 Exercise 186: "optimized version with T'_1 = 1024 seconds, and T'_∞ = 8 seconds. Why is it optimized? When you run it on your 32-processor machine, the running time is 40 seconds, as predicted by our formula."

Chapter 17

p. 398 Last paragraph: “This barrier class may look like it works, but it does not.”
p. 399 First paragraph: “Unfortunately, the attempt to make the barrier reusable causes it to break”
Line 13 of figure 17.3 should be
```
while (count.get() != 0){};
```

Chapter 18

p. 422 Figs 18.6 and 18.7 are switched with respect to the text.
p. 428 “Z later reads x, and sees value 2, which is inconsistent with the value it read for y.”
p. 429 In Fig. 18.16, Line 3 should be
```
protected T internalInit;
```
p. 430 In Figure 18.17, Line 21 should be
```
target.next = next;
```
p. 433 In Fig. 18.19, line 8 should be
```
if (other != previous)
```
p. 434 “Setters are implemented in a symmetric way, calling the setter in the second step.”
“This class has a single `AtomicObject<SSkipNode>` field. The constructor takes
as argument values to initialize the `AtomicObject<SSkipNode>` field.”

Appendix A

p. 455 In Fig. A1, Caption should read “This method initializes a number of Java threads, starts them, and waits for them to finish.”

Appendix B

p. 473 In Fig. B.4, the right and left-hand sides are reversed.
This problem is known as the *ABA* problem,
discussed in detail in Chapter 10.
p. 481. “John Hennessey and David Patterson”'

Acknowledgements

Here is a partial list of people who sent us errata. If you sent us an error, and you’re not acknowledged, please let us know so we can fix it.

Rajeev Alur
Liran Barsisa
Igor Berman
Martin Buchholz
Neill Clift
Eran Cohen

Daniel B. Curtis
David Dice
Brian Goetz
K. Gopinath
Jason T. Greene
Dan Grossman
Tim Halloran
Matt Hayes
Francis Hools
Omar Khan
Namhyung Kim
Guy Korland
Sergey Kotov
Doug Lea
Yossi Lev
Adam MacBeth
Mike Maloney
Tim McIver
Sergejs Melderis
Bartosz Milewski
Adam Morrison
Jonathan Perry
Amir Pnueli
Pat Quillen
Assaf Schuster
Joseph P. Skudlarek
James Stout
Deqing Sun
Seth Syberg
Fuad Tabba
Jaeheon Yi
Zhenyuan Zhao
Ruiwen Zuo