The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 17

July 14, 2009

```
1  class Prefix extends java.lang.Thread {
2    private int[] a;
3    private int i;
4    public Prefix(int[] myA, int myI) {
5      a = myA;
6      i = myI;
7    }
8    public void run() {
9      int d = 1, sum = 0;
10     while (d < m) {
11       if (i >= d)
12         sum = a[i-d];
13       if (i >= d)
14         a[i] += sum;
15       d = d * 2;
16     }
17   }
18 }
```

Figure 1: Parallel prefix computation.

Exercise 198. Fig. 1 shows how to use barriers to make a parallel prefix computation work on an asynchronous architecture.

A *parallel prefix* computation, given a sequence $a_0, \ldots, a_{m-1}$, of numbers, computes in parallel the partial sums:

$$b_i = \sum_{j=0}^{i} a_j.$$

In a synchronous system, where all threads take steps at the same time, there are simple, well-known algorithms for m threads to compute the partial sums in $\log m$ steps. The computation proceeds in a sequence of rounds, starting at round zero. In round r, if $i \geq 2^r$, thread i reads the value at $a[i - 2^r]$ into a local variable. Next, it adds that value to $a[i]$. Rounds continue until $2^r \geq m$. It is not hard to see that after $\log_2(m)$ rounds, the array a contains the partial sums.

1. What could go wrong if we executed the parallel prefix on $n > m$ threads?

2. Modify this program, adding one or more barriers, to make it work properly in a concurrent setting with n threads. What is the minimum number of barriers that are necessary?

Solution

- In an asynchronous system, threads could read a value before another thread, in an early round, has updated a memory location, or it could attempt to read a value after it has already been overwritten. These errors would then continue to propagate into later rounds and subsequent calculations.

- Each round consists of two phases, a reading phase and then a writing phase, each of which must appear to occur atomically (that is, a writing phase cannot begin while threads are still in the reading phase, and a subsequent reading phase must wait until the preceding writing phase has finished).

  We therefore introduce a barrier between these two phases, and again at the end of the round. The first prevents a thread from entering the writing phase early and updating the array before another thread reads its value. The second prevents a thread from entering the next round and reading a value before that value has been updated by the previous round's writing phase.

Exercise 199. Change the sense-reversing barrier implementation so that waiting threads call wait() instead of spinning.

- Give an example of a situation where suspending threads is better than spinning.

- Give an example of a situation where the other choice is better.

```
1  public SenseBarrier(int n) {
2    count = new AtomicInteger (n);
3    size = n;
4    sense = false;
5    threadSense = new ThreadLocal<Boolean>() {
6       protected Boolean initialValue () { return !sense; };
7    };
8    public void await() {
9      boolean mySense = threadSense.get();
10     int position = count.getAndDecrement();
11     synchronized(count) {
12       if (position == 1) { // I'm last
13         count.set(size);       // reset counter
14         sense = mySense;           // reverse sense
15       } else {
16         while (sense != mySense) {
17           count.wait();          // sleep until we can continue
18         }
```

```
19        }
20        threadSense.set(!mySense);
21        count.notifyAll();        // notify other threads that we're done
22      }
23    }
24  }
```

The modified await() method function calls wait() and notifyAll() instead of busy-waiting to check the barrier's sense. The notify() method is called following the reversal of the sense.

Suspending threads is better if we want to free the processor to do something else while we are waiting. This strategy makes sense only if the wait is expected to be long, and if there is something else the processor could be doing. Also, suspending is better on a non-cache coherent architecture where spinning on a remote location could be expensive. By contrast, spinning is better if the wait is expected to be short, or if there is nothing else for the processor to do.

Exercise 200 Change the tree barrier implementation so that it takes a Runnable object whose run() method is called once after the last thread arrives at the barrier, but before any thread leaves the barrier.

```
1   public class TreeBarrierRunnable implements Barrier {
2     int radix;      // tree fan-in
3     Node[] leaf;    // array of leaf nodes
4     int leaves;     // used to build tree
5     ThreadLocal<Boolean> threadSense; // thread-local sense
6     public TreeBarrierRunnable(int n, int r, Runnable toRun) {
7       radix = r;
8       leaves = 0;
9       leaf = new Node[n / r];
10      int depth = 0;
11      threadSense = new ThreadLocal<Boolean>() {
12        protected Boolean initialValue() { return true; };
13      };
14      // compute tree depth
15      while (n > 1) {
16        depth++;
17        n = n / r;
18      }
19      Node root = new Node(toRun);
20      build(root, depth - 1);
21    }
22    void build(Node parent, int depth) {
23      // are we at a leaf node?
```

```
24      if (depth == 0) {
25        leaf[leaves++] = parent;
26      } else {
27        for (int i = 0; i < radix; i++) {
28          Node child = new Node(parent);
29          build(child, depth - 1);
30        }
31      }
32    }
33    public void await() {
34      int me = ThreadID.get();
35      Node myLeaf = leaf[me / radix];
36      myLeaf.await();
37    }
38    private class Node {
39      Runnable toRun;
40      AtomicInteger count;
41      Node parent;
42      volatile boolean sense;
43      public Node() {
44        sense = false;
45        parent = null;
46        toRun = null;
47        count = new AtomicInteger(radix);
48      }
49      public Node(Runnable myToRun) {
50        this();
51        toRun = myToRun;
52      }
53      public Node(Node myParent) {
54        this();
55        parent = myParent;
56      }
57      public void await() {
58        boolean mySense = threadSense.get();
59        int position = count.getAndDecrement();
60        if (position == 1) {    // I'm last
61          if (parent != null) { // root?
62            parent.await();
63          } else {
64        if (toRun != null) {     // do we have a runnable object?
65          new Thread(toRun).start();
66        }
67          }
68          count.set(radix);       // reset counter
69          sense = mySense;
```

```
70          } else {
71            while (sense != mySense) {};
72          }
73          threadSense.set(!mySense);
74        }
75      }
76    }
```

The constructor (line 6) is modified to accept the Runnable object, a reference to which is passed to a modified Node constructor. The conditional in the Node's await() method (line 65) is then extended so that, in the case of the last node to reach the root, a thread is created from the Runnable object and subsequently started. The threadSense is then set as normal (line 65).

Exercise 201 Modify the combining tree barrier so that nodes can use any barrier implementation, not just the sense-reversing barrier.

Solution See Figure .

Exercise 202 A *tournament tree barrier* (Class TourBarrier in Fig. 3) is an alternative tree-structured barrier. Assume there are n threads, where n is a power of 2. The tree is a binary tree consisting of $2n-1$ nodes. Each leaf is owned by a single, statically determined thread. Each node's two children are linked as *partners*. One partner is statically designated as *active*, and the other as *passive*. Fig. 4 illustrates the tree structure.

Each thread keeps track of the current sense in a thread-local variable. When a thread arrives at a passive node, it sets its active partner's sense field to the current sense, and spins on its own sense field until its partner changes that field's value to the current sense. When a thread arrives at an active node, it spins on its sense field until its passive partner sets it to the current sense. When the field changes, that particular barrier is complete, and the active thread follows the parent reference to its parent node. Note that an active thread at one level may become passive at the next level. When the root node barrier is complete, notifications percolate down the tree. Each thread moves back down the tree setting its partner's sense field to the current sense.

This barrier improves a little on the combining tree barrier of Figure **??**. Explain how.

The tournament barrier code uses parent and partner references to navigate the tree. We could save space by eliminating these fields and keeping all the nodes in a single array with the root at index 0, the root's children at indexes 1 and 2, the grandchildren at indexes 3-6, and so on. Re-implement the tournament barrier to use indexing arithmetic instead of references to navigate the tree.

```
1  public class GTreeBarrier implements Barrier {
2    int radix;
3    int leaves;
4    Node[] leaf;
5    public GTreeBarrier(int size, int myRadix) {
6      radix = myRadix;
7      leaves = 0;
8      leaf = new Node[size / radix];
9      int depth = 0;
10     // compute tree depth
11     while (size > 1) {
12       depth++;
13       size = size / radix;
14     }
15     Node root = new Node();
16     build(root, depth - 1);
17   }
18   // recursive tree constructor
19   void build(Node parent, int depth) {
20     // are we at a leaf node?
21     if (depth == 0) {
22       leaf[leaves++] = parent;
23     } else {
24       for (int i = 0; i < radix; i++) {
25         Node child = new Node(parent);
26         build(child, depth - 1);
27       }
28     }
29   }
30   public void await() {
31     int me = ThreadID.get();
32     Node myLeaf = leaf[me / radix];
33     myLeaf.await(me);
34   }
35   private class Node {
36     final Barrier barrier;
37     Node parent;
38     final ThreadLocal<Boolean> threadSense;
39     volatile boolean done;
40     // construct root node
41     public Node() {
42       done = false;
43       threadSense = new ThreadLocal<Boolean>() {
44         protected Boolean initialValue() { return !Node.this.done; };
45       };
46       parent = null;
47       barrier = new SenseBarrier(radix);
48     }
49     public Node(Node _parent) {
50       this();
51       parent = _parent;
52     }
53     public void await(int me) {
54       boolean sense = threadSense.get();
55       barrier.await();
```

```
1   private class Node {
2     volatile boolean flag;        // signal when done
3     boolean active;               // active or passive?
4     Node parent;                  // parent node
5     Node partner;                 // partner node
6     // create passive node
7     Node() {
8       flag    = false;
9       active  = false;
10      partner = null;
11      parent  = null;
12    }
13    // create active node
14    Node(Node myParent) {
15      this();
16      parent = myParent;
17      active = true;
18    }
19    void await(boolean sense) {
20      if (active) { // I'm active
21        if (parent != null) {
22          while (flag != sense) {}; // wait for partner
23          parent.await(sense);      // wait for parent
24          partner.flag = sense;     // tell partner
25        }
26      } else {                      // I'm passive
27        partner.flag = sense;       // tell partner
28        while (flag != sense) {};   // wait for partner
29      }
30    }
31  }
```

Figure 3: The TourBarrier class

.

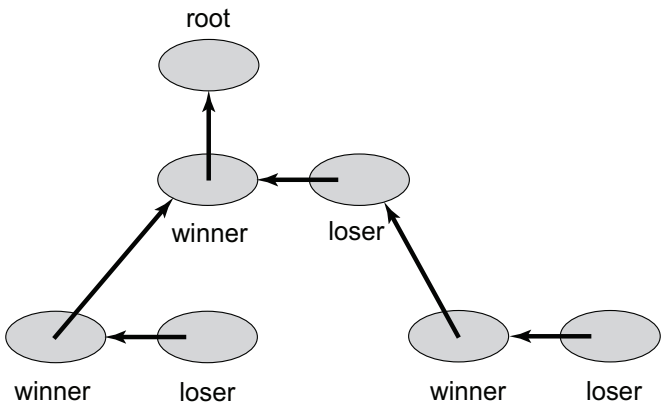

Figure 4: The TourBarrier class: information flow. Nodes are paired statically in active/passive pairs. Threads start at the leaves. Each thread in an active node waits for its passive partner to show up, then it proceeds up the tree. Each passive thread waits for its active partner for notification of completion. Once an active thread reaches the root, all threads have arrived, and notifications flow down the tree in the reverse order.

Solution Unlike the original combining tree barrier, nodes are designated either passive or active when the barrier is created, and this is not dependent on an additional shared variable to determine position when threads arrive at the barrier.

To implement the tree using an array, each node is given its own id (index) into a shared array containing all of the nodes. If a node has access to the global array of nodes, it can access its parent by accessing element $\lfloor \log_2(id-1) \rfloor$ and its partner by $id-1$ if id is even or $id+1$ if id is odd.

Exercise 203 The combining tree barrier uses a single thread-local sense field for the entire barrier. Suppose instead we were to associate a thread-local sense with each node as in Fig. **??**. Either:

- Explain why this implementation is equivalent to the other one, except that it consumes more memory, or.

- Give a counterexample showing that this implementation is incorrect.

```
1  private class Node {
2     AtomicInteger count;
3     Node parent;
4      volatile boolean sense;
5     int d;
6     // construct root node
7     public Node() {
8       sense = false;
9       parent = null;
10      count = new AtomicInteger(radix);
11      ThreadLocal<Boolean> threadSense;
12      threadSense = new ThreadLocal<Boolean>() {
13        protected Boolean initialValue () { return true; };
14    };
15    }
16    public Node(Node myParent) {
17      this();
18      parent = myParent;
19    }
20    public void await() {
21      boolean mySense = threadSense.get();
22      int position = count.getAndDecrement();
23      if (position == 1) {    // I'm last
24        if (parent != null) { // root?
25          parent.await();
26        }
27        count.set(radix);       // reset counter
28        sense = mySense;
29      } else {
30        while (sense != mySense) {};
31      }
32      threadSense.set(!mySense);
33    }
34  }
```

Figure 5: Thread local tree barrier.

Solution This implementation is incorrect and could lead to a disagreement in the thread-local sense fields among nodes after the first round.

Consider four threads in a tree of depth two. Threads 0 and 1 use node 1 while threads 2 and 3 use node 2 (and node 0 is the root).

The first time the barrier is used, thread 0 reaches the root and flips both its local sense fields for nodes 1 and 0 from the initial value of *true to* false, while thread 1 only flips its local sense for node 1. In the original case, both threads would have flipped the their local copy of the barrier's sense to *false. In the next round, if thread* 1 *reaches the root this time, its local sense variable at that node will still be in the initial setting of true, and not false as it should be.*

Exercise 204 *The tree barrier works "bottom-up," in the sense that barrier completion moves from the leaves up to the root, while wake-up information moves from the root back down to the leaves. Figs. 6 and 7 show an alternative design, called a* reverse tree barrier*, which works just like a tree barrier except for the fact that barrier completion starts at the root and moves down to the leaves. Either:*

- *Sketch an argument why this is correct, perhaps by reduction to the standard tree barrier, or*
- *Give a counterexample showing why it is incorrect.*

Solution *This top-down approach does not work as threads may be falsely notified that the barrier is complete. The problem stems from the fact that at each level, the active node notifies the parent regardless of whether the passive node has reached the barrier yet. Consider the case where a node is passive at the first level, and never reaches the barrier. Its active partner will begin by notifying its parent that itself has arrived, and as other threads reach the barrier, the barrier will complete. There will be no indication that this thread hasn't reached the barrier until the wake-up messages propagate back down to the parent node and the* getAndDecrement() *on Line 64 is performed, at which point other threads may already be awake.*

Exercise 205. *Implement an n-thread reusable barrier from an n-wire counting network and a single boolean variable. Sketch a proof that the barrier works.*

Solution *Initially the variable is set to* 1*. Each thread keeps track of the round parity (sense). The thread that emerges on wire n reverses the variable's sense while the others spin waiting for the sense to change. No thread emerges on wire n until a multiple of n threads has entered the network.*

Exercise 206 *Can you devise a "distributed" termination detection algorithm for the executer pool in which threads do not repeatedly update or test a central*

```
1  public class RevBarrier implements Barrier {
2    int radix;
3    ThreadLocal<Boolean> threadSense;
4    int leaves;
5    Node[] leaf;
6    public RevBarrier(int mySize, int myRadix) {
7      radix = myRadix;
8      leaves = 0;
9      leaf = new Node[mySize / myRadix];
10     int depth = 0;
11     threadSense = new ThreadLocal<Boolean>() {
12       protected Boolean initialValue() { return true; };
13     };
14     // compute tree depth
15     while (mySize > 1) {
16       depth++;
17       mySize = mySize / myRadix;
18     }
19     Node root = new Node();
20     root.d = depth;
21     build(root, depth - 1);
22   }
23   // recursive tree constructor
24   void build(Node parent, int depth) {
25     // are we at a leaf node?
26     if (depth == 0) {
27       leaf[leaves++] = parent;
28     } else {
29       for (int i = 0; i < radix; i++) {
30         Node child = new Node(parent);
31         child.d = depth;
32         build(child, depth - 1);
33       }
34     }
35   }
```

Figure 6: Reverse tree barrier Part 1

```
36    public void await() {
37      int me = ThreadInfo.getIndex();
38      Node myLeaf = leaf[me / radix];
39      myLeaf.await(me);
40    }
41    private class Node {
42      AtomicInteger count;
43      Node parent;
44       volatile boolean sense;
45      int d;
46      // construct root node
47      public Node() {
48        sense = false;
49        parent = null;
50        count = new AtomicInteger(radix);
51      }
52      public Node(Node myParent) {
53        this();
54        parent = myParent;
55      }
56      public void await(int me) {
57        boolean mySense = threadSense.get();
58        // visit parent first
59        if ((me % radix) == 0) {
60          if (parent != null) { // root?
61            parent.await(me / radix);
62          }
63        }
64        int position = count.getAndDecrement();
65        if (position == 1) {    // I'm last
66          count.set(radix);       // reset counter
67          sense = mySense;
68        } else {
69          while (sense != mySense) {};
70        }
71        threadSense.set(!mySense);
72      }
73    }
74  }
```

Figure 7: Reverse tree barrier Part 2: correct or not?

location for termination, but rather use only local uncontended variables? Variables may be unbounded, but state changes should take constant time, (so you cannot parallelize the shared counter).

Hint: adapt the atomic snapshot algorithm from Chapter 4.

Solution *See Figure 8. The simplest solution is to have an* AtomicStampedReference<Boolean> *array, one for each thread. Each* AtomicStampedReference<Boolean> *object encompasses a Boolean value and a stamp (version number). A thread declares itself active (inactive) by setting its own Boolean field to true (false). Each time it updates its status, it increments the associated stamp. The* isTerminated() *method takes an obstruction-free snapshot of the array by reading it twice. The first time, it records each stamp. If any reference's Boolean field is true, then* isTerminated() *returns false because there are active threads. The second time, it rereads each stamp. If there is a version mismatch, or if any reference is true, then* isTerminated() *returns false because threads are still changing state. Otherwise, the method has completed a consistent snapshot in which all entries were false, (See Chapter* **??***, so it is safe to return true.*

Exercise 207. *A* dissemination barrier *is a symmetric barrier implementation in which threads spin on statically-assigned locally-cached locations using only loads and stores. As illustrated in Fig. 9, the algorithm runs in a series of rounds. At round r, thread i notifies thread $i + 2^r \pmod n$, (where n is the number of threads) and waits for notification from thread $i - 2^r \pmod n$.*

For how many rounds must this protocol run to implement a barrier? What if n is not a power of 2? Justify your answers.

Solution *We argue by induction that if thread i completes round r, then threads $i, i-1, \ldots, i-2^r+1$ have arrived at the barrier. In the base case, when thread i finishes round 0, threads i and $i-1$ have both arrived at the barrier. Consider round $r+1$. By the induction hypothesis, threads $i, i-1, \ldots, i-2^r+1$ have arrived at the barrier. When thread i is notified by thread $i-2^r$, the induction hypothesis implies tha threads $i-2^r, i-2^r-1, \ldots, i-2^{r+1}-1$ have arrived at the barrier. It follows that the dissemination barrier must run for $\lceil \log_2 n \rceil$ rounds*

Exercise 208 *Give a reusable implementation of a dissemination barrier in Java.*

Hint: you may want to keep track of both the parity and the sense of the current phase.

```
public class DisBarrier implements Barrier {
  int size;
  int logSize;
```

```
1  public class T2Barrier implements TDBarrier {
2    int size;
3    AtomicStampedReference<Boolean>[] active;
4    T2Barrier(int mySize) {
5      size = mySize;
6      active = (AtomicStampedReference<Boolean>[]) new AtomicStampedReference[size];
7      for (int i = 0; i < size; i++) {
8        active[i] = new AtomicStampedReference<Boolean>(true, 0);
9      }
10   }
11   public void setActive(boolean state) {
12     int[] stamp = {0};
13     int me = ThreadID.get();
14     boolean oldState = active[me].get(stamp);
15     active[me].set(state, stamp[0] + 1);
16   }
17   public boolean isTerminated() {
18     int[] stamps = new int[size];
19     int[] stamp = {0};
20     // first collect
21     for (int i = 0; i < size; i++) {
22       boolean oldState = active[i].get(stamp);
23       if (oldState) {  // someone still active
24         return false;
25       }
26       stamps[i] = stamp[0];
27     }
28     // second collect
29     for (int i = 0; i < size; i++) {
30       boolean oldState = active[i].get(stamp);
31       if (oldState) {  // someone still active
32         return false;
33       }
34       if (stamp[0] != stamp[i]) { // version mismatch, something happened
35         return false;
36       }
37     }
38     return true;
39   }
40 }
```

Figure 8: Snapshot-based Termination Barrier

```
  Node[][] node;
  ThreadLocal<Boolean> mySense;
  ThreadLocal<Integer> myParity;
  ...
   public void await() {
    int parity = myParity.get();
    boolean sense = mySense.get();
    int i = ThreadID.get();
    for (int r = 0; r < logSize; r++) {
      node[r][i].partner.flag[parity] = sense;
      while (node[r][i].flag[parity] != sense) {}
    }
    if (parity == 1) {
      mySense.set(!sense);
    }
    myParity.set(1 - parity);
  }
  private class Node {
    boolean[] flag = {false, false}; // signal when done
    Node partner;                    // partner node
  }
}
```

Exercise 209 *Create a table that summarizes the total number of operations in the static tree, combining tree, and dissemination barriers.*

Solution *[]*

Notifications	*static tree* $O(\log n)$	*combining tree* $O(n)$	*dissemination* $O(n \log n)$

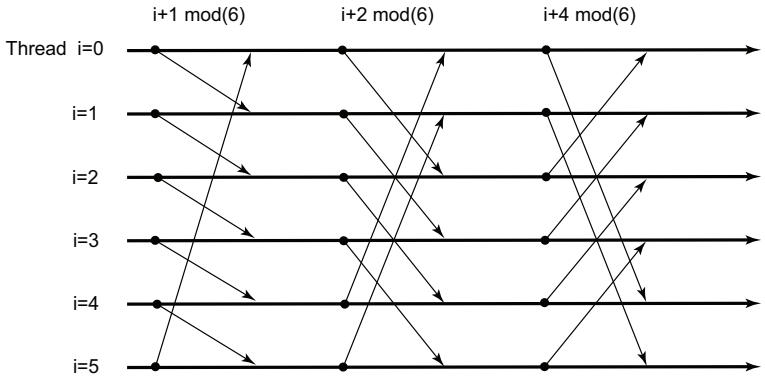

Figure 9: Communication in the dissemination barrier. In each round r a thread i communicates with thread $i + 2^r \pmod n$.

Exercise 210 *In the termination detection barrier, the state is set to active before stealing the task; otherwise the stealing thread could be declared inactive; then it would steal a task, and before setting its state back to active, the thread it stole from could become inactive. This would lead to an undesirable situation in which all threads are declared inactive yet the computation continues. Can you devise a terminating executer pool in which the state is set to active only* after *successfully stealing a task?*

Solution *It's impossible. Assume the* isTerminated() *method makes its decision based only on the active/inactive status of each thread, and (generously) assume it can take an atomic snapshot of those bits. A thread announces itself as busy only*

Imagine there is one task left in the queue for thread A. All threads but A are marked inactive. Thread B, currently inactive, steals the task, but then pauses before it can change its status. A notices that it has no work, and that its queue is empty, and marks itself inactive. At this point, all threads appear to be inactive. There is no way to tell that B will, when it awakens, declare itself active.