The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 12

July 14, 2009

Exercise 126. Design an unbounded lock-based Stack<T> implementation based on a linked list.

Solution. See Figure 1.

Exercise 127. Design a bounded lock-based Stack<T> using an array.

1. Use a single lock and a bounded array.
2. Try to make your algorithm lock-free. Where do you run into difficulty?

Solution. See Figure 2.

The problem with making this approach lock-free is figuring out how to decrement the top and remove the item atomically. Removing the item and decrementing top doesn't work because other threads can modify the stack between those two steps, and similarly if the top is decremented first.

Exercise 128. Modify the unbounded lock-free stack of Section **??** to work in the absence of a garbage collector. Create a thread-local pool of preallocated nodes and recycle them. To avoid the ABA problem, consider using the AtomicStampedReference<T> class from java.util.concurrent.atomic that encapsulates both a reference and an integer *stamp*.

Solution. See Figure 3. The code for the stack is almost identical to the earlier lock-free stack, except that to avoid ABA, we use an AtomicStampedReference in place of an AtomicReference. We use a thread-local object to hold a list of nodes. The method that allocates nodes tries to take one off the free list, and if the list is empty, it calls new. The method that frees nodes simply pushes the node onto the list.

Exercise 129. Discuss the backoff policies used in our implementation. Does it make sense to use the same shared Backoff object for both pushes and pops in our LockFreeStack<T> object? How else could we structure the backoff in space and time in the EliminationBackoffStack<T>?

Solution. For the LockFreeStack<T>, it makes sense to use the same Backoff object for both pushes and pops because they are all competing with one another. For the EliminationBackoffStack<T>, we do not need a Backoff object because that time is better spent in the elimination array.

Exercise 130. Implement a stack algorithm assuming there is a bound, in any state of the execution, on the total difference between the number of pushes and pops to the stack.

```
1  public class ListBasedStack<T> {
2    Node top;
3    Lock lock;
4    ListBasedStack() {
5      top = null;
6      lock = new ReentrantLock();
7    }
8    public void push(T x) {
9      lock.lock();
10     try {
11       top = new Node(x, top);
12     } finally {
13       lock.unlock();
14     }
15   }
16   public T pop() throws EmptyException {
17     lock.lock();
18     try {
19       if (top == null) {
20         throw new EmptyException();
21       } else {
22         T result = top.value;
23         top = top.next;
24         return result;
25       }
26     } finally {
27       lock.unlock();
28     }
29   }
30   private class Node {
31     public T value;
32     public Node next;
33     Node(T myValue, Node myNext) {
34       value = myValue;
35       next = myNext;
36     }
37   }
38 }
```

Figure 1: Unbounded lock-based Stack<T> implementation based on a linked list.

```
1  public class ArrayBasedStack<T> {
2    int capacity; // size of array
3    int top;      // first empty slot
4    T[] items;    // array of items
5    ArrayBasedStack(int myCapacity) {
6      capacity = myCapacity;
7      items = (T[]) new Object[capacity];
8      top = 0;
9    }
10   public void push(T x) throws FullException {
11     if (top == capacity) {
12       throw new FullException();
13     }
14     items[top++] = x;
```

```
1  public class LockFreeStackNoGC<T> {
2    AtomicStampedReference<Node> top = new AtomicStampedReference<Node>(null, 0);
3    static final int MIN_DELAY = ...;
4    static final int MAX_DELAY = ...;
5    Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
6    private boolean tryPush(Node node) {
7      int [] stamp = {0};
8      Node oldTop = top.get(stamp);
9      node.next = oldTop;
10     return (top.compareAndSet(oldTop, node, stamp[0], stamp[0] + 1));
11   }
12   public void push(T value) throws InterruptedException {
13     Node node = (Node<T>) allocateNode();
14     node.value = value;
15     while (true) {
16       if (tryPush(node)) {
17         return;
18       } else {
19         backoff.backoff();
20       }
21     }
22   }
23   protected Node tryPop() throws EmptyException {
24     int [] stamp = {0};
25     Node<T> oldTop = top.get(stamp);
26     if (oldTop == null) {
27       throw new EmptyException();
28     }
29     Node newTop = oldTop.next;
30     if (top.compareAndSet(oldTop, newTop, stamp[0], stamp[0]+1)) {
31       return oldTop;
32     } else {
33       return null;
34     }
35   }
36   public T pop() throws EmptyException, InterruptedException {
37     while (true) {
38       Node<T> node = tryPop();
39       if (node != null) {
40         T value = node.value;
41         freeNode(node);
42         return value;
43       } else {
44         backoff.backoff();
45       }
46     }
47   }
48   // thread-local pool of nodes
49   static ThreadLocal<Node> freeList = new ThreadLocal<Node>() {
50     protected Node initialValue() {
51       return null;
52     }
53   };
54   private static Node allocateNode() {
55     Node node = freeList.get();
```

Solution. The bound is actually a bound on the maximum number of items in the stack.

For a lock-based stack using an array, we could use the bound to determine the size of the array.

For a list-based stack, we could use the bound to provide each thread with a local pool of nodes (see Exercise 128).

Exercise 131. Consider the problem of implementing a bounded stack using an array indexed by a top counter, initially zero. In the absence of concurrency, these methods are almost trivial. To push an item, increment top to reserve an array entry, and then store the item at that index. To pop an item, decrement top, and return the item at the previous top index.

Clearly, this strategy does not work for concurrent implementations, because one cannot make atomic changes to multiple memory locations. A single synchronization operation can either increment or decrement the top counter, but not both, and there is no way atomically to increment the counter and store a value.

Nevertheless, Bob D. Hacker decides to solve this problem. He decides to adapt the dual-data structure approach of Chapter **??** to implement a *dual* stack. His DualStack<0>T class splits push() and pop() methods into *reservation* and *fulfillment* steps. Bob's implementation appears in Figure 4.

The stack's top is indexed by the top field, an AtomicInteger manipulated only by getAndIncrement() and getAndDecrement() calls. Bob's push() method's reservation step reserves a slot by applying getAndIncrement() to top. Suppose the call returns index i. If i is in the range $0 \ldots \mathsf{capacity} - 1$, the reservation is complete. In the fulfillment phase, push(x) stores x at index i in the array, and raises the full i flag to indicate that the value is ready to be read. The value field must be **volatile** to guarantee that once flag is raised, the value has already been written to index i of the array.

If the index returned from push()'s getAndIncrement() is less than 0, the push() method repeatedly retries getAndIncrement() until it returns an index greater than or equal to 0. (The index could be less than 0 due to getAndDecrement() calls of failed pop() calls to an empty stack. Each such failed getAndDecrement() decrements the top by one more past the 0 array bound. If the index returned is greater than capacity-1, push() throws an exception because the stack is full.

The situation is symmetric for pop(). It checks that the index is within the bounds and removes an item by applying getAndDecrement() to top, returning index i. If i is in the range $0 \ldots \mathsf{capacity} - 1$, the reservation is complete.. For the fulfillment phase, pop() spins on the full flag of array slot i, until it detects that the flag is true, indicating that the push() call is successful

What is wrong with Bob's algorithm? Is this problem inherent or can you think of a way to fix it?

Solution. Consider the following interleaving. Initially the stack is empty.

1. A pushes x, increments top to 1, reserving slot 0.

```
1  public class DualStack<T> {
2    private class Slot {
3      boolean full = false;
4      volatile T value = null;
5    }
6    Slot [] stack;
7    int capacity;
8    private AtomicInteger top = new AtomicInteger(0); // array index
9    public DualStack(int myCapacity) {
10     capacity = myCapacity;
11     stack = (Slot []) new Object[capacity];
12     for (int i = 0; i < capacity; i++) {
13       stack[i] = new Slot();
14     }
15   }
16   public void push(T value) throws FullException {
17     while (true) {
18       int i = top.getAndIncrement();
19       if (i > capacity - 1) {  // is stack full ?
20         throw new FullException();
21       } else if (i > 0) { // i in range, slot reserved
22         stack[i].value = value;
23         stack[i].full = true; //push fulfilled
24         return;
25       }
26     }
27   }
28   public T pop() throws EmptyException {
29     while (true) {
30       int i = top.getAndDecrement();
31       if (i < 0) {  // is stack empty?
32         throw new EmptyException();
33       } else if (i < capacity - 1) {
34         while (!stack[i].full ){};
35         T value = stack[i].value;
36         stack[i].full = false;
37         return value; //pop fulfilled
38       }
39     }
40   }
41 }
```

Figure 4: Bob's Problematic Dual Stack.

```
1 public interface Rooms {
2   public interface Handler {
3     void onEmpty();
4   }
5   void enter(int i);
6   boolean exit();
7   public void setExitHandler(int i, Rooms.Handler h);
8 }
```

Figure 5: The Rooms interface

2. B pops, decrements top to 0, spins waiting for the full bit.

3. C pushes y, increments top to 1, also reserving slot 0.

4. C writes y to slot 0.

5. A writes x to slot 0, causing y to be lost.

A similar disaster can happen between concurrent pops.

To fix this problem, we would need a way to store a value and set the full bit atomically, as well as remove a value and clear the full bit atomically. (These are sometimes called "full-empty" bits.)

Exercise 132. In Exercise 97 we ask you to implement the Rooms interface, reproduced in Fig. 5. The Rooms class manages a collection of *rooms*, indexed from 0 to m (where m is a known constant). Threads can enter or exit any room in that range. Each room can hold an arbitrary number of threads simultaneously, but only one room can be occupied at a time. The last thread to leave a room triggers an onEmpty() handler, which runs while all rooms are empty.

Fig. 6 shows an incorrect concurrent stack implementation.

1. Explain why this stack implementation does not work.

2. Fix it by adding calls to a two-room Rooms class: one room for pushing and one for popping.

Solution. Problems arise if a push() runs concurrently with a pop(). Suppose push() increments top, and then a concurrent pop() decrements it. The pop() method will then remove and return a meaningless value from the stack.

To fix this problem, we employ rooms synchronization. When a thread calls a stack push() or pop() method, it enters a "room" associated with that method. Each room can hold an arbitrary number of threads simultaneously, but only one room can be occupied at a time. For example, any number of threads might enter the *push room* and each will execute the push() method

```
1  public class Stack<T> {
2    private AtomicInteger top;
3    private T[] items;
4    public Stack(int capacity) {
5      top = new AtomicInteger();
6      items = (T[]) new Object[capacity];
7    }
8    public void push(T x) throws FullException {
9      int i = top.getAndIncrement();
10     if (i >= items.length) {  // stack is full
11       top.getAndDecrement(); // restore state
12       throw new FullException();
13     }
14     items[i] = x;
15   }
16   public T pop() throws EmptyException {
17     int i = top.getAndDecrement() - 1;
18     if (i < 0) {              // stack is empty
19       top.getAndIncrement(); // restore state
20       throw new EmptyException();
21     }
22     return items[i];
23   }
24 }
```

Figure 6: Unsynchronized concurrent stack

```
1  public class RoomStack<T> {
2    /**
3     * Used by push() method calls
4     */
5    private final static int PUSH_ROOM = 0;
6    /**
7     * Used by pop() method calls
8     */
9    private final static int POP_ROOM = 1;
10   /**
11    * Index of top element in stack
12    */
13   private AtomicInteger top;
14   /**
15    * Array of elements.
16    */
17   private T[] items;
18   /**
19    * Rooms bookkeeping.
20    */
21   private Rooms rooms;
22   /**
23    * constructor
24    * @param n size of stack
25    */
26   public RoomStack(int n) {
27     top   = new AtomicInteger(0);
28     items = (T[]) new Object[n];
29     rooms = new TicketRooms(2);
30   }
31   /**
32    * push object onto stack
33    * @param x Object to push
34    * @throws rooms.FullException When stack is full .
35    */
36   public void push(T x) throws FullException {
37     try {
38       rooms.enter(PUSH_ROOM);
39       int i = top.getAndIncrement();
40       if (i >= items.length) {  // stack is full
41         top.getAndDecrement(); // restore state
42         throw new FullException();
43       }
44       items[i] = x;
45     } finally {
46       rooms.exit();
47     }
48   }
49   /**
50    * pop object from stack
51    * @throws rooms.EmptyException When stack is empty
52    * @return element removed from top of stack
53    */
54   public T pop() throws Empty
55   /**
```

```
1  public class DynamicStack<T> {
2    private final static int PUSH_ROOM = 0;
3    private final static int POP_ROOM = 1;
4    private AtomicInteger top;
5    private T[] items;
6    private Rooms rooms;
7    private boolean overflow;
8    public DynamicStack(int n) {
9      top = new AtomicInteger(0);
10     items = (T[]) new Object[n];
11     rooms = new TicketRooms(2);
12     rooms.setExitHandler(PUSH_ROOM, new MyHandler());
13   }
14   public void push(T x) {
15     rooms.enter(PUSH_ROOM);
16     int i = top.getAndIncrement();
17     if (i >= items.length) {// stack is full
18       overflow = true;       // ask for help
19       top.getAndDecrement();
20       rooms.exit();          // leave room
21       push(x);               // try again
22     }
23     items[i] = x;
24     rooms.exit();
25   }
```

Figure 8: The dynamic stack and its push() method

concurrently with the others, but no thread can enter the *pop room* while the push room is occupied.

See Figure 7. Rooms are indexed by integers. The enter(i) method enters room i, and the exit() method exits the currently active room.

Exercise 133. This exercise is a follow-on to Exercise 132. Instead of having the push() method throw FullException, exploit the push room's exit handler to resize the array. Remember that no thread can be in any room when an exit handler is running, so (of course) only one exit handler can run at a time.

Solution. In the DynamicStack class illustrated in Figures 8 and 9, the push() method checks whether the current stack size is about to overflow. If so, it sets a bit indicating an overflow condition, leaves the push room, and then calls push() again. The eventual emptiness property implies that the push room will eventually become empty, and the last thread to leave the room will copy the

```
1   public T pop() throws EmptyException {
2     try {
3       rooms.enter(POP_ROOM);
4       int i = top.getAndDecrement() − 1;
5       if (i < 0) {                    // stack is empty
6         top.getAndIncrement(); // back off
7         throw new EmptyException();
8       }
9       return items[i];
10    } finally {
11      rooms.exit();
12    }
13  }
14  class MyHandler implements Rooms.Handler {
15    public void onEmpty() {
16      DynamicStack stack = DynamicStack.this;
17      if (stack.overflow) {
18        stack.overflow = false;
19        T[] newItems = (T[]) new Object[2 ∗ stack.items.length];
20        System.arraycopy(stack.items, 0, newItems, 0, stack.items.length);
21        stack.items = newItems;
22      }
23    }
24  }
25 }
```

Figure 9: Dynamic stack's pop() method and exit handler

old version of the stack to a new version twice the size. This example illustrates the individual importance of the three properties. The mutual exclusion property ensures that no other thread is accessing the stack while it is being copied. the starvation-freedom property ensures that every push() that encounters an overflowing stack will be able to exit. Finally, the eventual emptiness property ensures that the push room eventually becomes empty, the stack is eventually copied, and the push() call eventually succeeds. Without this guarantee, concurrent push() calls could effectively deadlock, repeatedly entering and exiting a non-empty room.