11 Concurrent Stacks and Elimination

11.1 Introduction

The `Stack<T>` class is a collection of items (of type `T`) that provides `push()` and `pop()` methods satisfying the *last-in-first-out* (LIFO) property: the last item pushed is the first popped. This chapter considers how to implement concurrent stacks. At first glance, stacks seem to provide little opportunity for concurrency, because `push()` and `pop()` calls seem to need to synchronize at the top of the stack.

Surprisingly, perhaps, stacks are not inherently sequential. In this chapter, we show how to implement concurrent stacks that can achieve a high degree of parallelism. As a first step, we consider how to build a lock-free stack in which pushes and pops synchronize at a single location.

11.2 An Unbounded Lock-Free Stack

Fig. 11.1 shows a concurrent `LockFreeStack` class, whose code appears in Figs. 11.2, 11.3 and 11.4. The lock-free stack is a linked list, where the `top` field points to the first node (or *null* if the stack is empty.) For simplicity, we usually assume it is illegal to add a *null* value to a stack.

A `pop()` call that tries to remove an item from an empty stack throws an exception. A `push()` method creates a new node (Line 13), and then calls `tryPush()` to try to swing the `top` reference from the current top-of-stack to its successor. If `tryPush()` succeeds, `push()` returns, and if not, the `tryPush()` attempt is repeated after backing off. The `pop()` method calls `tryPop()`, which uses `compareAndSet()` to try to remove the first node from the stack. If it succeeds, it returns the node, otherwise it returns *null*. (It throws an exception if the stack

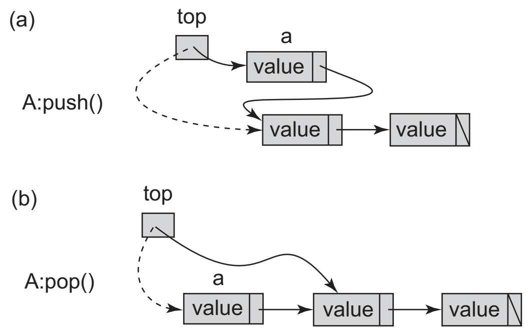

Figure 11.1 A Lock-free stack. In Part (a) a thread pushes value *a* into the stack by applying a `compareAndSet()` to the `top` field. In Part (b) a thread pops value *a* from the stack by applying a `compareAndSet()` to the `top` field.

```
public class LockFreeStack<T> {
  AtomicReference<Node> top = new AtomicReference<Node>(null);
  static final int MIN_DELAY = ...;
  static final int MAX_DELAY = ...;
  Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);

  protected boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node));
  }
  public void push(T value) {
    Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else {
        backoff.backoff();
      }
    }
  }
```

Figure 11.2 The `LockFreeStack<T>` class: in the `push()` method, threads alternate between trying to alter the `top` reference by calling `tryPush()`, and backing off using the `Backoff` class from Fig. 7.5 of Chapter 7.

is empty.) The `tryPop()` method is called until it succeeds, at which point `pop()` returns the value from the removed node.

As we have seen in Chapter 7, one can significantly reduce contention at the `top` field using exponential backoff (see Fig. 7.5 of Chapter 7). Accordingly, both

```
1  public class Node {
2      public T value;
3      public Node next;
4      public Node(T value) {
5        value = value;
6        next  = null;
7      }
8  }
```

Figure 11.3 Lock-free stack list node.

```
 1    protected Node tryPop() throws EmptyException {
 2      Node oldTop = top.get();
 3      if (oldTop == null) {
 4        throw new EmptyException();
 5      }
 6      Node newTop = oldTop.next;
 7      if (top.compareAndSet(oldTop, newTop)) {
 8        return oldTop;
 9      } else {
10        return null;
11      }
12    }
13    public T pop() throws EmptyException {
14      while (true) {
15        Node returnNode = tryPop();
16        if (returnNode != null) {
17          return returnNode.value;
18        } else {
19          backoff.backoff();
20        }
21      }
22    }
```

Figure 11.4 The LockFreeStack<T> class: The pop() method alternates between trying to change the top field and backing off.

the push() and pop() methods back off after an unsuccessful call to tryPush() or tryPop().

This implementation is lock-free because a thread fails to complete a push() or pop() method call only if there were infinitely many successful calls that modified the top of the stack. The linearization point of both the push() and the pop() methods is the successful compareAndSet(), or the throwing of the exception, in Line 3, in case of a pop() on an empty stack. Note that the compareAndSet() call by pop() does not have an ABA problem (see Chapter 10) because the Java garbage collector ensures that a node cannot be reused by one thread, as long as that node is accessible to another thread. Designing a lock-free stack that avoids the ABA problem without a garbage collector is left as an exercise.

11.3 Elimination

The `LockFreeStack` implementation scales poorly, not so much because the stack's `top` field is a source of *contention*, but primarily because it is a *sequential bottleneck*: method calls can proceed only one after the other, ordered by successful `compareAndSet()` calls applied to the stack's `top` field.

Although exponential backoff can significantly reduce contention, it does nothing to alleviate the sequential bottleneck. To make the stack parallel, we exploit this simple observation: if a `push()` is immediately followed by a `pop()`, the two operations cancel out, and the stack's state does not change. It is as if both operations never happened. If one could somehow cause concurrent pairs of pushes and pops to cancel, then threads calling `push()` could exchange values with threads calling `pop()`, without ever modifying the stack itself. These two calls would *eliminate* one another.

As depicted in Fig. 11.5, threads eliminate one another through an `EliminationArray` in which threads pick random array entries to try to meet complementary calls. Pairs of complementary `push()` and `pop()` calls exchange values and return. A thread whose call cannot be eliminated, either because it has failed to find a partner, or found a partner with the wrong kind of method call (such as a `push()` meeting a `push()`), can either try again to eliminate at a new location, or can access the shared `LockFreeStack`. The combined data structure, array, and shared stack, is linearizable because the shared stack is linearizable, and the eliminated calls can be ordered as if they happened at the point in which they exchanged values.

We can use the `EliminationArray` as a backoff scheme on a shared `LockFreeStack`. Each thread first accesses the `LockFreeStack`, and if it fails

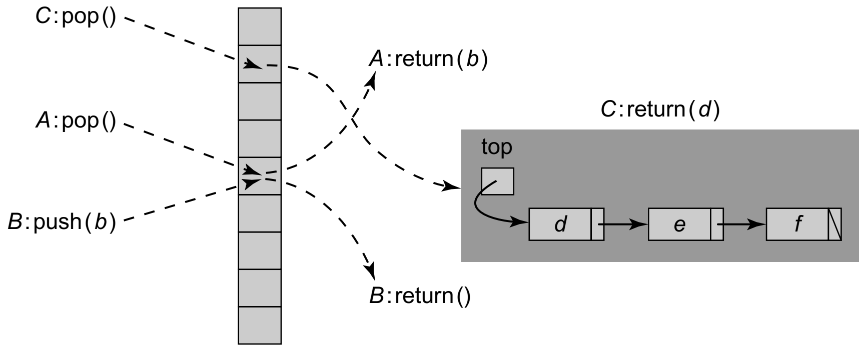

Figure 11.5 The `EliminationBackoffStack<T>` class. Each thread selects a random location in the array. If thread *A*'s `pop()` and *B*'s `push()` calls arrive at the same location at about the same time, then they exchange values without accessing the shared `LockFreeStack`. Thread *C* that does not meet another thread eventually pops the shared `LockFreeStack`.

to complete its call (that is, the `compareAndSet()` attempt fails), it attempts to eliminate its call using the array instead of simply backing off. If it fails to eliminate itself, it calls the `LockFreeStack` again, and so on. We call this structure an `EliminationBackoffStack`.

11.4 The Elimination Backoff Stack

Here is how to construct an `EliminationBackoffStack`, a lock-free linearizable stack implementation.

We are reminded of a story about two friends who are discussing politics on election day, each trying, to no avail, to convince the other to switch sides.

Finally, one says to the other: "Look, it's clear that we are unalterably opposed on every political issue. Our votes will surely cancel out. Why not save ourselves some time and both agree to not vote today?"

The other agrees enthusiastically and they part.

Shortly after that, a friend of the first one who had heard the conversation says, "That was a sporting offer you made."

"Not really," says the second. "This is the third time I've done this today."

The principle behind our construction is the same. We wish to allow threads with pushes and pops to coordinate and cancel out, but must avoid a situation in which a thread can make a sporting offer to more than one other thread. We do so by implementing the `EliminationArray` using coordination structures called *exchangers*, objects that allow exactly two threads (and no more) to rendezvous and exchange values.

We already saw how to exchange values using locks in the synchronous queue of Chapter 10. Here, we need a lock-free exchange, one in which threads spin rather than block, as we expect them to wait only for very short durations.

11.4.1 A Lock-Free Exchanger

A `LockFreeExchanger<T>` object permits two threads to exchange values of type `T`. If thread A calls the object's `exchange()` method with argument a, and B calls the same object's `exchange()` method with argument b, then A's call will return value b and vice versa. On a high level, the exchanger works by having the first thread arrive to write its value, and spin until a second arrives. The second then detects that the first is waiting, reads its value, and signals the exchange. They each have now read the other's value, and can return. The first thread's call may timeout if the second does not show up, allowing it to proceed and leave the exchanger, if it is unable to exchange a value within a reasonable duration.

```
 1  public class LockFreeExchanger<T> {
 2    static final int EMPTY = ..., WAITING = ..., BUSY = ...;
 3    AtomicStampedReference<T> slot = new AtomicStampedReference<T>(null, 0);
 4    public T exchange(T myItem, long timeout, TimeUnit unit)
 5      throws TimeoutException {
 6      long nanos = unit.toNanos(timeout);
 7      long timeBound = System.nanoTime() + nanos;
 8      int[] stampHolder = {EMPTY};
 9      while (true) {
10        if (System.nanoTime() > timeBound)
11          throw new TimeoutException();
12        T yrItem = slot.get(stampHolder);
13        int stamp = stampHolder[0];
14        switch(stamp) {
15        case EMPTY:
16          if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
17            while (System.nanoTime() < timeBound){
18              yrItem = slot.get(stampHolder);
19              if (stampHolder[0] == BUSY) {
20                slot.set(null, EMPTY);
21                return yrItem;
22              }
23            }
24            if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
25              throw new TimeoutException();
26            } else {
27              yrItem = slot.get(stampHolder);
28              slot.set(null, EMPTY);
29              return yrItem;
30            }
31          }
32          break;
33        case WAITING:
34          if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY))
35            return yrItem;
36          break;
37        case BUSY:
38          break;
39        default: // impossible
40          ...
41        }
42      }
43    }
44  }
```

Figure 11.6 The LockFreeExchanger<T> Class.

The LockFreeExchanger<T> class appears in Fig. 11.6. It has a single AtomicStampedReference<T> field,[1] slot. The exchanger has three possible states: EMPTY, BUSY, or WAITING. The reference's stamp records the exchanger's state (Line 14). The exchanger's main loop continues until the timeout limit

1 See Chapter 10, Pragma 10.6.1.

passes, when it throws an exception (Line 10). In the meantime, a thread reads the state of the `slot` (Line 12) and proceeds as follows:

- If the state is `EMPTY`, then the thread tries to place its item in the slot and set the state to `WAITING` using a `compareAndSet()` (Line 16). If it fails, then some other thread succeeds and it retries. If it was successful (Line 17), then its item is in the slot and the state is `WAITING`, so it spins, waiting for another thread to complete the exchange. If another thread shows up, it will take the item in the slot, replace it with its own, and set the state to `BUSY` (Line 19), indicating to the waiting thread that the exchange is complete. The waiting thread will consume the item and reset the state to `EMPTY`. Resetting to `EMPTY` can be done using a simple write because the waiting thread is the only one that can change the state from `BUSY` to `EMPTY` (Line 20). If no other thread shows up, the waiting thread needs to reset the state of the slot to `EMPTY`. This change requires a `compareAndSet()` because other threads might be attempting to exchange by setting the state from `WAITING` to `BUSY` (Line 24). If the call is successful, it raises a timeout exception. If, however, the call fails, some exchanging thread must have shown up, so the waiting thread completes the exchange (Line 26).
- If the state is `WAITING`, then some thread is waiting and the slot contains its item. The thread takes the item, and tries to replace it with its own by changing the state from `WAITING` to `BUSY` using a `compareAndSet()` (Line 34). It may fail if another thread succeeds, or the other thread resets the state to `EMPTY` following a timeout. If so, the thread must retry. If it does succeed changing the state to `BUSY`, then it can return the item.
- If the state is `BUSY` then two other threads are currently using the slot for an exchange and the thread must retry (Line 37).

Notice that the algorithm allows the inserted item to be *null*, something used later in the elimination array construction. There is no ABA problem because the `compareAndSet()` call that changes the state never inspects the item. A successful exchange's linearization point occurs when the second thread to arrive changes the state from `WAITING` to `BUSY` (Line 34). At this point both `exchange()` calls overlap, and the exchange is committed to being successful. An unsuccessful exchange's linearization point occurs when the timeout exception is thrown.

The algorithm is lock-free because overlapping `exchange()` calls with sufficient time to exchange will fail only if other exchanges are repeatedly succeeding. Clearly, too short an exchange time can cause a thread never to succeed, so care must be taken when choosing timeout durations.

11.4.2 The Elimination Array

An `EliminationArray` is implemented as an array of `Exchanger` objects of maximal size `capacity`. A thread attempting to perform an exchange picks an array entry at random, and calls that entry's `exchange()` method, providing

```
 1  public class EliminationArray<T> {
 2    private static final int duration = ...;
 3    LockFreeExchanger<T>[] exchanger;
 4    Random random;
 5    public EliminationArray(int capacity) {
 6      exchanger = (LockFreeExchanger<T>[]) new LockFreeExchanger[capacity];
 7      for (int i = 0; i < capacity; i++) {
 8        exchanger[i] = new LockFreeExchanger<T>();
 9      }
10      random = new Random();
11    }
12    public T visit(T value, int range) throws TimeoutException {
13      int slot = random.nextInt(range);
14      return (exchanger[slot].exchange(value, duration,
15               TimeUnit.MILLISECONDS));
16    }
17  }
```

Figure 11.7 The EliminationArray<T> class: in each visit, a thread can choose dynamically the sub-range of the array from which it will will randomly select a slot.

its own input as an exchange value with another thread. The code for the EliminationArray appears in Fig. 11.7. The constructor takes as an argument the capacity of the array (the number of distinct exchangers). The EliminationArray class provides a single method, visit(), which takes timeout arguments. (Following the conventions used in the java.util.concurrent package, a timeout is expressed as a number and a time unit.) The visit() call takes a value of type T and either returns the value input by its exchange partner, or throws an exception if the timeout expires without exchanging a value with another thread. At any point in time, each thread will select a random location in a subrange of the array (Line 13). This subrange will be determined dynamically based on the load on the data structure, and will be passed as a parameter to the visit() method.

The EliminationBackoffStack is a subclass of LockFreeStack that overrides the push() and pop() methods, and adds an EliminationArray field. Figs. 11.8 and 11.9 show the new push() and pop() methods. Upon failure of a tryPush() or tryPop() attempt, instead of simply backing off, these methods try to use the EliminationArray to exchange values (Lines 15 and 34). A push() call calls visit() with its input value as argument, and a pop() call with *null* as argument. Both push() and pop() have a thread-local RangePolicy object that determines the EliminationArray subrange to be used.

When push() calls visit(), it selects a random array entry within its range and attempts to exchange a value with another thread. If the exchange is successful, the pushing thread checks whether the value was exchanged with a pop() method (Line 18) by testing if the value exchanged was *null*. (Recall that pop() always offers *null* to the exchanger while push() always offers a non-*null* value.) Symmetrically, when pop() calls visit(), it attempts an exchange, and if the

```
public class EliminationBackoffStack<T> extends LockFreeStack<T> {
  static final int capacity = ...;
  EliminationArray<T> eliminationArray = new EliminationArray<T>(capacity);
  static ThreadLocal<RangePolicy> policy = new ThreadLocal<RangePolicy>() {
    protected synchronized RangePolicy initialValue() {
      return new RangePolicy();
    }

  public void push(T value) {
    RangePolicy rangePolicy = policy.get();
    Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else try {
        T otherValue = eliminationArray.visit
                          (value, rangePolicy.getRange());
        if (otherValue == null) {
          rangePolicy.recordEliminationSuccess();
          return; // exchanged with pop
        }
      } catch (TimeoutException ex) {
        rangePolicy.recordEliminationTimeout();
      }
    }
  }
}
```

Figure 11.8 The EliminationBackoffStack<T> class: this push() method overrides the LockFreeStack push() method. Instead of using a simple Backoff class, it uses an EliminationArray and a dynamic RangePolicy to select the subrange of the array within which to eliminate.

```
  public T pop() throws EmptyException {
    RangePolicy rangePolicy = policy.get();
    while (true) {
      Node returnNode = tryPop();
      if (returnNode != null) {
        return returnNode.value;
      } else try {
        T otherValue = eliminationArray.visit(null, rangePolicy.getRange());
        if (otherValue != null) {
          rangePolicy.recordEliminationSuccess();
          return otherValue;
        }
      } catch (TimeoutException ex) {
        rangePolicy.recordEliminationTimeout();
      }
    }
  }
```

Figure 11.9 The EliminationBackoffStack<T> class: this pop() method overrides the LockFreeStack push() method.

exchange is successful it checks (Line 36) whether the value was exchanged with a push() call by checking whether it is not *null*.

It is possible that the exchange will be unsuccessful, either because no exchange took place (the call to visit() timed out) or because the exchange was with the same type of operation (such as a pop() with a pop()). For brevity, we choose a simple approach to deal with such cases: we retry the tryPush() or tryPop() calls (Lines 13 and 31).

One important parameter is the range of the EliminationArray from which a thread selects an Exchanger location. A smaller range will allow a greater chance of a successful collision when there are few threads, while a larger range will lower the chances of threads waiting on a busy Exchanger (recall that an Exchanger can only handle one exchange at a time). Thus, if few threads access the array, they should choose smaller ranges, and as the number of threads increase, so should the range. One can control the range dynamically using a RangePolicy object that records both successful exchanges (as in Line 37) and timeout failures (Line 40). We ignore exchanges that fail because the operations do not match (such as push() with push()), because they account for a fixed fraction of the exchanges for any given distribution of push() and pop() calls. One simple policy is to shrink the range as the number of failures increases and vice versa.

There are many other possible policies. For example, one can devise a more elaborate range selection policy, vary the delays on the exchangers dynamically, add additional backoff delays before accessing the shared stack, and control whether to access the shared stack or the array dynamically. We leave these as exercises.

The EliminationBackoffStack is a linearizable stack: any successful push() or pop() call that completes by accessing the LockFreeStack can be linearized at the point of its LockFreeStack access. Any pair of eliminated push() and pop() calls can be linearized when they collide. As noted earlier, the method calls completed through elimination do not affect the linearizability of those completed in the LockFreeStack, because they could have taken effect in any state of the LockFreeStack, and having taken effect, the state of the LockFreeStack would not have changed.

Because the EliminationArray is effectively used as a backoff scheme, we expect it to deliver performance comparable to the LockFreeStack at low loads. Unlike the LockFreeStack, it has the potential to scale. As the load increases, the number of successful eliminations will grow, allowing many operations to complete in parallel. Moreover, contention at the LockFreeStack is reduced because eliminated operations never access the stack.

11.5 Chapter Notes

The LockFreeStack is credited to Treiber [145]. Actually it predates Treiber's report in 1986. It was probably invented in the early 1970s to motivate the

CAS operation on the IBM 370. The EliminationBackoffStack is due to Danny Hendler, Nir Shavit, and Lena Yerushalmi [57]. An efficient exchanger, which quite interestingly uses an elimination array, was introduced by Doug Lea, Michael Scott, and Bill Scherer [136]. A variant of this exchanger appears in the Java Concurrency Package. The EliminationBackoffStack we present here is modular, making use of exchangers, but somewhat inefficient. Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit present a highly effective implementation of an EliminationArray [120].

11.6 Exercises

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

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?

Exercise 128. Modify the unbounded lock-free stack of Section 11.2 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*.

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>?

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.

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 10 to implement a *dual* stack. His `DualStack<T>` class splits `push()` and `pop()` methods into *reservation* and *fulfillment* steps. Bob's implementation appears in Fig. 11.10.

The stack's top is indexed by the `top` field, an `AtomicInteger` manipulated only by `getAndIncrement()` and `getAndDecrement()` calls. Bob's `push()`

```
public class DualStack<T> {
  private class Slot {
    boolean full = false;
    volatile T value = null;
  }
  Slot[] stack;
  int capacity;
  private AtomicInteger top = new AtomicInteger(0); // array index
  public DualStack(int myCapacity) {
    capacity = myCapacity;
    stack = (Slot[]) new Object[capacity];
    for (int i = 0; i < capacity; i++) {
      stack[i] = new Slot();
    }
  }
  public T pop() throws EmptyException {
    while (true) {
      int i = top.getAndDecrement();
      if (i <= 0) { // is stack empty?
        throw new EmptyException();
       } else if (i-1 < capacity){
         while (!stack [i-1].full ) {};
         T value = stack[i-1].value;
         stack[i-1].full = false;
         return value ; //pop fulfilled
        }
     }
  }
  public T pop() throws EmptyException {
    while (true) {
      int i = top.getAndDecrement();
      if (i < 0) { // is stack empty?
        throw new EmptyException();
      } else if (i < capacity - 1) {
        while (!stack[i].full){};
        T value = stack[i].value;
        stack[i].full = false;
        return value; //pop fulfilled
      }
    }
  }
}
```

Figure 11.10 Bob's problematic dual stack.

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 . . . `capacity` − 1, the reservation is complete. In the fulfillment phase, `push(x)` stores *x* at index *i* in the array, and raises the `full` 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 . . . `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 an inherent problem or can you think of a way to fix it?

Exercise 132. In Exercise 97 we ask you to implement the `Rooms` interface, reproduced in Fig. 11.11. 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. 11.12 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.

```
public interface Rooms {
  public interface Handler {
    void onEmpty();
  }
  void enter(int i);
  boolean exit();
  public void setExitHandler(int i, Rooms.Handler h) ;
}
```

Figure 11.11 The `Rooms` interface.

```
public class Stack<T> {
  private AtomicInteger top;
  private T[] items;
  public Stack(int capacity) {
    top = new AtomicInteger();
    items = (T[]) new Object[capacity];
  }
  public void push(T x) throws FullException {
    int i = top.getAndIncrement();
    if (i >= items.length) { // stack is full
      top.getAndDecrement(); // restore state
      throw new FullException();
    }
    items[i] = x;
  }
  public T pop() throws EmptyException {
    int i = top.getAndDecrement() - 1;
    if (i < 0) {           // stack is empty
      top.getAndIncrement(); // restore state
      throw new EmptyException();
    }
    return items[i];
  }
}
```

Figure 11.12 Unsynchronized concurrent stack.

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.