James Scherick Due 10 November 2022

Midterm 2

Professor: Maurice Herlihy

CS1760: Multiprocessor Synchronization

Please solve *exactly* 3 out of the 4 questions (if you solve more, please specify which 3 you would like us to check).

This is a closed book exam. Please do not consult the textbook, other humans, or the internet.

Good luck!

Problem 1. Describe how to modify each of the linked list algorithms if object hash codes are not guaranteed to be unique.

Problem 2. Show that in the optimistic list algorithm, once tail is reachable from any node, it remains reachable. (Code for the optimistic list appears in the figures.)

Problem 3. 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. 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 $0 \ldots \text{capacity} - 1$, 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 $0 \ldots \text{capacity} - 1$, 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?

Problem 4. Consider the unbounded lock-based queue's deq() method in Figure 5. Is it necessary to hold the lock when checking that the queue is not empty? Explain.

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

Figure 1: The OptimisticList class: remove() method

```
24 public boolean contains(T item) {
25   int key = item.hashCode();
26   while (true) {
27     Node pred = this.head; // sentinel node;
28     Node curr = pred.next;
29     while (curr.key < key) {
30       pred = curr; curr = curr.next;
31     }
32     pred.lock(); curr.lock();
33     try {
34       if (validate(pred, curr)) {
35         return (curr.key == key);
36       }
37     } finally {                    // always unlock
38       pred.unlock(); curr.unlock();
39     }
40   }
41 }
```

Figure 2: The OptimisticList class: contains() method

```
42 public boolean add(T item) {
43   int key = item.hashCode();
44   while (true) {
45     Node pred = head;
46     Node curr = pred.next;
47     while (curr.key < key) {
48       pred = curr; curr = curr.next;
49     }
50     pred.lock(); curr.lock();
51     try {
52       if (validate(pred, curr)) {
53         if (curr.key == key) {
54           return false;
55         } else {
56           Node node = new Node(item);
57           node.next = curr;
58           pred.next = node;
59           return true;
60         }
61       }
62     } finally {
63       pred.unlock(); curr.unlock();
64     }
65   }
66 }
```

Figure 3: The OptimisticList class: add() method

```
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         top.getAndDecrement(); // restore index
21         throw new FullException();
22       } else if (i >= 0) { // i in range, slot reserved
23         stack[i].value = value;
24         stack[i].full = true; // push fulfilled
25         return;
26       }
27     }
28   }
29   public T pop() throws EmptyException {
30     while (true) {
31       int i = top.getAndDecrement() - 1;
32       if (i < 0) {  // is stack empty?
33         top.getAndIncrement() // restore index
34         throw new EmptyException();
35       } else if (i <= capacity - 1) {
36         while (!stack[i].full ){};
37         T value = stack[i].value;
38         stack[i].full = false;
39         return value; // pop fulfilled
40       }
41     }
42   }
43 }
```

Figure 4: Bob's Problematic Dual Stack.

```
1  public class UnboundedQueue<T> {
2    ReentrantLock enqLock, deqLock;
3     volatile Node head, tail ;
4    public UnboundedQueue() {
5      head = new Node(null);
6       tail  = head;
7      enqLock = new ReentrantLock();
8      deqLock = new ReentrantLock();
9    }
10   public void enq(T x) {
11     Node e = new Node(x);
12     enqLock.lock();
13     try {
14        tail .next = e;
15        tail  = e;
16     } finally {
17       enqLock.unlock();
18     }
19   }
20 public T deq() throws EmptyException {
21     T result ;
22     deqLock.lock();
23     try {
24       if (head.next == null) {
25         throw new EmptyException();
26       }
27        result  = head.next.value;
28       head = head.next;
29     } finally {
30       deqLock.unlock();
31     }
32     return result ;
33   }
```

Figure 5: The UnboundedQueue<T> class.