125 design an unbounded + lock-based Stack (T) implementation based on linked list.

A. class Stack (T) &

Node top = null;

Lock lock = new ReentrantLock();

public void push (T z) &

Node n = new Node.(n),

lock, lock();

n.next = top;

top= n;

lock, unlock ();

<u>ક</u>

try & lack. lock();

Node n = top;

if (n == null) throw new Empty Exception ().

top = n.next;

return n.value;

3 finally &

lock. unlock ();

Z

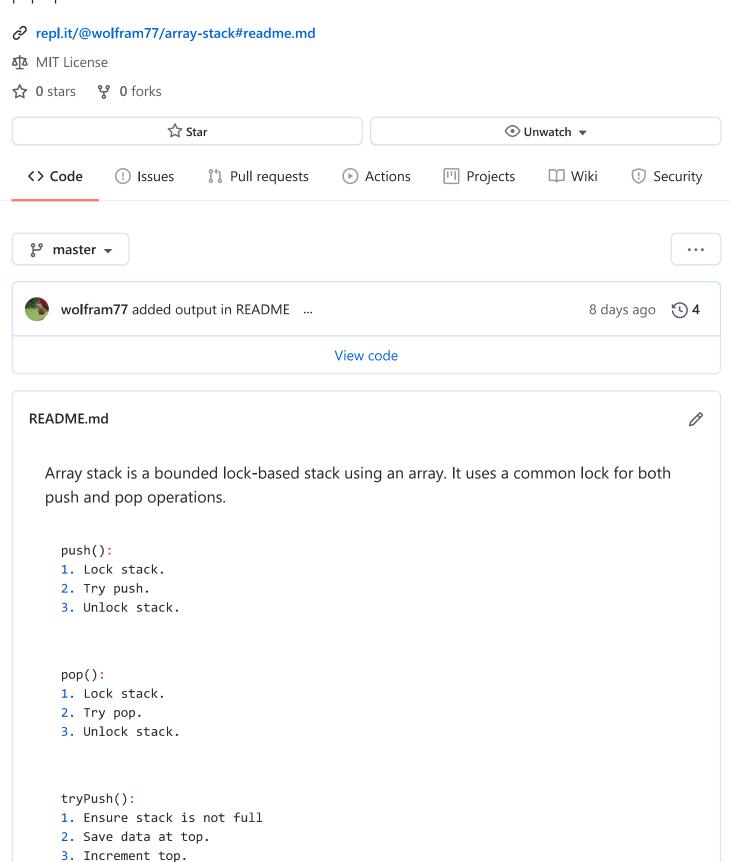
z

I // Stack (T)

public T pop () throws Buffer and flow Exception & lock.lock(); if (top == 0) throw new Buffer Underflow Exception(); top = top-1; return data [top]: 3 finally & Lock - unlock (): 3/1 Stack (T) Pop(): 2. push (x): @ cas (top, top-1); 1 (as (top, top+1); (3 a [old Top] = x; 3 ereturn altop]. consider above algorithm using CAS for push & pop Cexcluding the bound checks). given ODBG is an example execution order, where push only manages to execute the CAS after which a pop completes @, @ and finally push completes (3. so this pope) managed to execute when push () didn't have a chance to actually store the pushed value. hence, popl) would 3 read a garbage value here this is a problem that can occurrer with a bounded array-based lock-free stack, ¥ because set value and modify top are not atomic.

☐ javaf / array-stack

Array stack is a bounded lock-based stack using an array. It uses a common lock for both push and pop operations.



tryPop():

```
1. Ensure stack is not empty.
2. Decrement top.
3. Return data at top.
## OUTPUT
Starting 10 threads with sequential stack
7: failed pop
1: failed pop
5: failed pop
8: failed pop
9: failed pop
1: popped 0/1000 values
5: popped 158/1000 values
7: popped 0/1000 values
8: popped 0/1000 values
9: popped 31/1000 values
Was LIFO? false
Starting 10 threads with array stack
Was LIFO? true
```

See ArrayStack.java for code, Main.java for test, and repl.it for output.

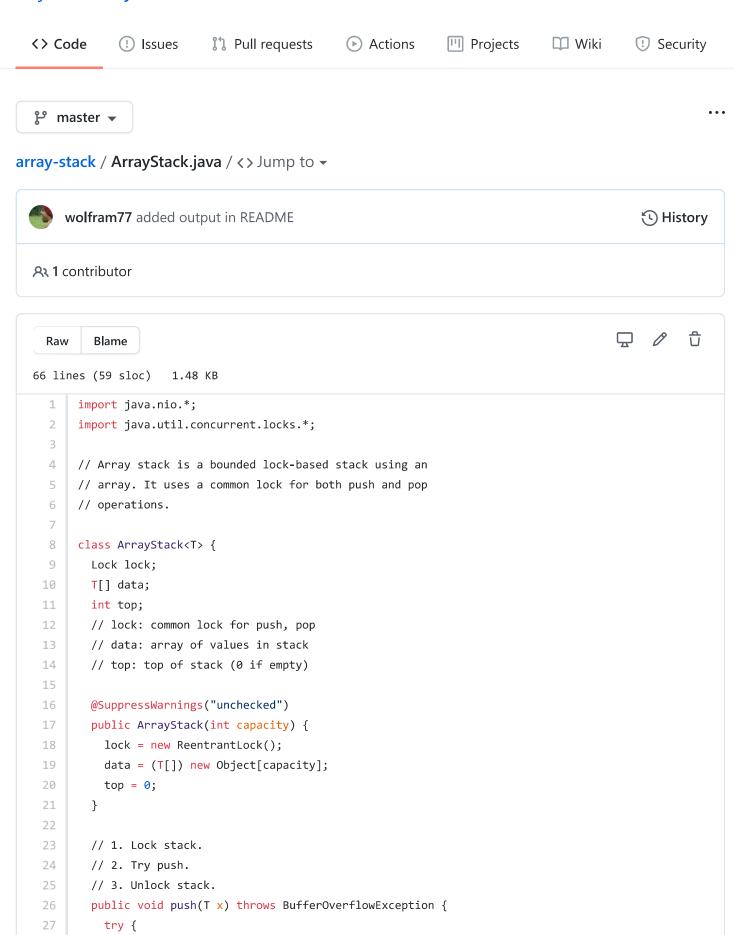
references

• The Art of Multiprocessor Programming :: Maurice Herlihy, Nir Shavit

Languages

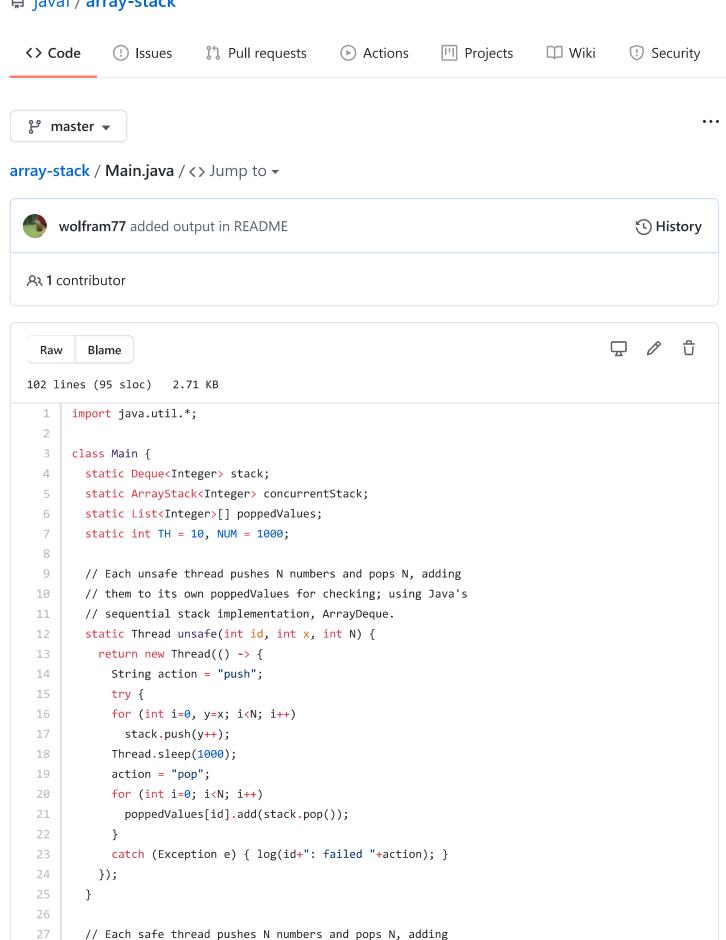
Java 100.0%

☐ javaf / array-stack



```
28
         lock.lock();
                        // 1
29
                        // 2
         tryPush(x);
30
         } finally {
         lock.unlock(); // 3
31
32
         }
33
       }
34
       // 1. Lock stack.
36
       // 2. Try pop.
37
       // 3. Unlock stack.
       public T pop() throws BufferUnderflowException {
38
39
         try {
40
         lock.lock();
                          // 1
41
         return tryPop(); // 2
42
         } finally {
43
         lock.unlock(); // 3
44
         }
45
       }
46
47
       // 1. Ensure stack is not full
48
       // 2. Save data at top.
49
       // 3. Increment top.
50
       protected void tryPush(T x)
51
         throws BufferOverflowException {
52
         if (top == data.length)
                                                 // 1
53
           throw new BufferOverflowException(); // 1
         data[top++] = x; // 2, 3
54
55
       }
56
57
       // 1. Ensure stack is not empty.
58
       // 2. Decrement top.
59
       // 3. Return data at top.
60
       protected T tryPop()
61
         throws BufferUnderflowException {
62
         if (top == 0)
                                                  // 1
63
           throw new BufferUnderflowException(); // 1
64
         return data[--top]; // 2, 3
65
       }
66
     }
```

☐ javaf / array-stack



```
28
       // them to its own poppedValues for checking; using
29
       // ArrayStack.
       static Thread safe(int id, int x, int N) {
30
31
         return new Thread(() -> {
32
           String action = "push";
33
           try {
34
           for (int i=0, y=x; i<N; i++)</pre>
             concurrentStack.push(y++);
           Thread.sleep(1000);
37
           action = "pop";
38
           for (int i=0; i<N; i++)</pre>
39
             poppedValues[id].add(concurrentStack.pop());
           }
41
           catch (Exception e) { log(id+": failed "+action);
42
           e.printStackTrace(); }
43
         });
44
       }
45
46
       // Checks if each thread popped N values, and they are
47
       // globally unique.
       static boolean wasLIFO(int N) {
48
49
         Set<Integer> set = new HashSet<>();
50
         boolean passed = true;
51
         for (int i=0; i<TH; i++) {
52
           int n = poppedValues[i].size();
53
           if (n != N) {
54
             log(i+": popped "+n+"/"+N+" values");
             passed = false;
56
           }
57
           for (Integer x : poppedValues[i])
58
             if (set.contains(x)) {
59
               log(i+": has duplicate value "+x);
60
               passed = false;
             }
61
           set.addAll(poppedValues[i]);
62
         }
63
         return passed;
65
       }
       @SuppressWarnings("unchecked")
67
68
       static void testThreads(boolean safe) {
         stack = new ArrayDeque<>();
         concurrentStack = new ArrayStack<>(TH*NUM);
70
         poppedValues = new List[TH];
71
72
         for (int i=0; i<TH; i++)</pre>
73
           poppedValues[i] = new ArrayList<>();
74
         Thread[] threads = new Thread[TH];
75
         for (int i=0; i<TH; i++) {</pre>
```

```
76
            threads[i] = safe?
 77
              safe(i, i*NUM, NUM) :
 78
              unsafe(i, i*NUM, NUM);
 79
            threads[i].start();
          }
80
          try {
 81
          for (int i=0; i<TH; i++)</pre>
 82
 83
            threads[i].join();
          }
 84
 85
          catch (Exception e) {}
 86
        }
 87
 88
        public static void main(String[] args) {
          log("Starting "+TH+" threads with sequential stack");
 89
 90
          testThreads(false);
 91
          log("Was LIFO? "+wasLIFO(NUM));
 92
          log("");
93
          log("Starting "+TH+" threads with array stack");
 94
          testThreads(true);
 95
          log("Was LIFO? "+wasLIFO(NUM));
          log("");
 96
97
        }
98
99
        static void log(String x) {
100
          System.out.println(x);
101
        }
102
      }
```