# HW4 : SUBHAJIT SAHU : 2018801013

## ELIMINATION BACKOFF STACK

Elimination-backoff stack is an unbounded lock-free LIFO linked list, that eliminates concurrent pairs of pushes and pops with exchanges. It uses compare-and-set (CAS) atomic operation to provide concurrent access with obstruction freedom. In order to support even greater concurrency, in case a push/pop fails, it tries to pair it with another pop/push to eliminate the operation through exchange of values.

class EliminationBackoffStack<T>:

```
push():
1. Create a new node with given value.
2. Try pushing it to stack.
3a. If successful, return.
3b. Otherwise, try exchanging on elimination array.
4a. If found a matching pop, return.
4b. Otherwise, retry 2.

pop():
1. Try popping a node from stack.
2a. If successful, return node's value
2b. Otherwise, try exchanging on elimination array.
3a. If found a matching push, return its value.
3b. Otherwise, retry 1.

tryPush():
1. Get stack top.
2. Set node's next to top.
3. Try push node at top (CAS).

tryPop():
1. Get stack top, and ensure stack not empty.
```

```
2. Try pop node at top, and set top to next (CAS).
```

Elimination array provides a list of exchangers which are picked at random for a given value.

```
class EliminationArray<T>

visit():

1. Try exchanging value on a random exchanger.
```

Exchanger is a lock-free object that permits two threads to exchange values, within a time limit.

class Exchanger<T>

```
exchange():

1. Calculate last wait time.

2. If wait time exceeded, then throw expection.

3. Get slot value and stamp.

4a. If slot is EMPTY (no value):

4b. Try adding 1st value to slot, else retry 2.

4c. Try getting 2nd value from slot, within time limit.

5a. If slot is WAITING (has 1st value):

5b. Try adding 2nd value to slot, else retry 2.

5c. Return 1st value.

6a. If slot is BUSY (has 2nd value):

6b. Retry 2.
```

**CODE: https://repl.it/@wolfram77/elimination-backoff-stack#Node.java**

```java
class Node<T> {
  public T value;
  public Node<T> next;
  public Node(T x) {
    value = x;
  }
}
```

**CODE: https://repl.it/@wolfram77/elimination-backoff-stack#Exchanger.java**

```
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;


// Exchanger is a lock-free object that permits two threads
// to exchange values, within a time limit.


class Exchanger<T> {
  AtomicStampedReference<T> slot;
  static final int EMPTY = 0;
  static final int WAITING = 1;
  static final int BUSY = 2;
  // slot: stores value and stamp
  // EMPTY: slot has no value.
  // WAITING: slot has 1st value, waiting for 2nd.
  // BUSY: slot has 2nd value, waiting to be empty.


  public Exchanger() {
    slot = new AtomicStampedReference<>(null, 0);
  }


  // 1. Calculate last wait time.
  // 2. If wait time exceeded, then throw expection.
  // 3. Get slot value and stamp.
  // 4a. If slot is EMPTY (no value):
  // 4b. Try adding 1st value to slot, else retry 2.
  // 4c. Try getting 2nd value from slot, within time limit.
  // 5a. If slot is WAITING (has 1st value):
  // 5b. Try adding 2nd value to slot, else retry 2.
  // 5c. Return 1st value.
  // 6a. If slot is BUSY (has 2nd value):
  // 6b. Retry 2.
  public T exchange(T y, long timeout, TimeUnit unit)
    throws TimeoutException {
    long w = unit.toNanos(timeout); // 1
    long W = System.nanoTime() + w; // 1
    int[] stamp = {EMPTY};
    while (System.nanoTime() < W) { // 2
      T x = slot.get(stamp); // 3
      switch (stamp[0]) {    // 3
        case EMPTY:    // 4
        if (addA(y)) { // 4
          while (System.nanoTime() < W)            // 4
            if ((x = removeB()) != null) return x; // 4
          throw new TimeoutException(); // 5
        }
        break;
        case WAITING:   // 7
        if (addB(x, y)) // 7
          return x;     // 7
        break;
        case BUSY: // 8
        break;     // 8
        default:
      }
    }
    throw new TimeoutException(); // 2
  }


  // 1. Add 1st value to slot.
  // 2. Set its stamp as WAITING (for 2nd).
  private boolean addA(T y) { // 1, 2
    return slot.compareAndSet(null, y, EMPTY, WAITING);
  }


  // 1. Add 2nd value to slot.
  // 2. Set its stamp as BUSY (for 1st to remove).
  private boolean addB(T x, T y) { // 1, 2
    return slot.compareAndSet(x, y, WAITING, BUSY);
  }


  // 1. If stamp is not BUSY (no 2nd value in slot), exit.
  // 2. Set slot as EMPTY, and get 2nd value from slot.
  private T removeB() {
    int[] stamp = {EMPTY};
    T x = slot.get(stamp);             // 1
    if (stamp[0] != BUSY) return null; // 1
```

```
    slot.set(null, EMPTY); // 2
    return x;              // 2
  }
}
```

**CODE: https://repl.it/@wolfram77/elimination-backoff-stack#EliminationArray.java**

```
import java.util.*;
import java.util.concurrent.*;


// Elimination array provides a list of exchangers which
// are picked at random for a given value.


class EliminationArray<T> {
  Exchanger<T>[] exchangers;
  final long TIMEOUT;
  final TimeUnit UNIT;
  Random random;
  // exchangers: array of exchangers
  // TIMEOUT: exchange timeout number
  // UNIT: exchange timeout unit
  // random: random number generator


  @SuppressWarnings("unchecked")
  public EliminationArray(int capacity, long timeout, TimeUnit unit) {
    exchangers = new Exchanger[capacity];
    for (int i=0; i<capacity; i++)
      exchangers[i] = new Exchanger<>();
    random = new Random();
    TIMEOUT = timeout;
    UNIT = unit;
  }


  // 1. Try exchanging value on a random exchanger.
  public T visit(T x) throws TimeoutException {
    int i = random.nextInt(exchangers.length);
    return exchangers[i].exchange(x, TIMEOUT, UNIT);
  }
}
```

**CODE: https://repl.it/@wolfram77/elimination-backoff-stack#EliminationBackoffStack.java**

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;


// Elimination-backoff stack is an unbounded lock-free LIFO
// linked list, that eliminates concurrent pairs of pushes
// and pops with exchanges.  It uses compare-and-set (CAS)
// atomic operation to provide concurrent access with
// obstruction freedom. In order to support even greater
// concurrency, in case a push/pop fails, it tries to
// pair it with another pop/push to eliminate the operation
// through exchange of values.


class EliminationBackoffStack<T> {
  AtomicReference<Node<T>> top;
  EliminationArray<T> eliminationArray;
  static final int CAPACITY = 100;
  static final long TIMEOUT = 10;
  static final TimeUnit UNIT = TimeUnit.MILLISECONDS;
  // top: top of stack (null if empty)
  // eliminationArray: for exchanging values between push, pop
  // CAPACITY: capacity of elimination array
  // TIMEOUT: exchange timeout for elimination array
  // UNIT: exchange timeout unit for elimination array


  public EliminationBackoffStack() {
    top = new AtomicReference<>(null);
    eliminationArray = new EliminationArray<>(
      CAPACITY, TIMEOUT, UNIT
    );
  }


  // 1. Create a new node with given value.
  // 2. Try pushing it to stack.
  // 3a. If successful, return.
  // 3b. Otherwise, try exchanging on elimination array.
  // 4a. If found a matching pop, return.
  // 4b. Otherwise, retry 2.
  public void push(T x) {
```

```
  Node<T> n = new Node<>(x); // 1
  while (true) {
    if (tryPush(n)) return;  // 2, 3a
    try {
    T y = eliminationArray.visit(x); // 3b
    if (y == null) return;           // 4a
    }
    catch (TimeoutException e) {}
  } // 4b
}


// 1. Try popping a node from stack.
// 2a. If successful, return node's value
// 2b. Otherwise, try exchanging on elimination array.
// 3a. If found a matching push, return its value.
// 3b. Otherwise, retry 1.
public T pop() throws EmptyStackException {
  while (true) {
    Node<T> n = tryPop();          // 1
    if (n != null) return n.value; // 2a
    try {
    T y = eliminationArray.visit(null); // 2b
    if (y != null) return y;            // 3a
    }
    catch (TimeoutException e) {} // 3b
  }
}


// 1. Get stack top.
// 2. Set node's next to top.
// 3. Try push node at top (CAS).
protected boolean tryPush(Node<T> n) {
  Node<T> m = top.get(); // 1
  n.next = m;                     // 2
  return top.compareAndSet(m, n); // 3
}


// 1. Get stack top, and ensure stack not empty.
// 2. Try pop node at top, and set top to next (CAS).
protected Node<T> tryPop() throws EmptyStackException {
  Node<T> m = top.get();                          // 1
  if (m == null) throw new EmptyStackException(); // 1
```

```
    Node<T> n = m.next;                       // 2
    return top.compareAndSet(m, n)? m : null; // 2
  }
}
```

**CODE: https://repl.it/@wolfram77/elimination-backoff-stack#Main.java**

```
import java.util.*;


class Main {
  static Deque<Integer> stack;
  static EliminationBackoffStack<Integer> concurrentStack;
  static List<Integer>[] poppedValues;
  static int TH = 10, NUM = 1000;


  // Each unsafe thread pushes N numbers and pops N, adding
  // them to its own poppedValues for checking; using Java's
  // sequential stack implementation, ArrayDeque.
  static Thread unsafe(int id, int x, int N) {
    return new Thread(() -> {
      String action = "push";
      try {
      for (int i=0, y=x; i<N; i++)
        stack.push(y++);
      Thread.sleep(1000);
      action = "pop";
      for (int i=0; i<N; i++)
        poppedValues[id].add(stack.pop());
      }
      catch (Exception e) { log(id+": failed "+action); }
    });
  }


  // Each safe thread pushes N numbers and pops N, adding
  // them to its own poppedValues for checking; using
  // BackoffStack.
  static Thread safe(int id, int x, int N) {
    return new Thread(() -> {
      String action = "push";
      try {
      for (int i=0, y=x; i<N; i++)
        concurrentStack.push(y++);
    Thread.sleep(1000);
    action = "pop";
    for (int i=0; i<N; i++)
      poppedValues[id].add(concurrentStack.pop());
    }
    catch (Exception e) { log(id+": failed "+action); }
  });
}


// Checks if each thread popped N values, and they are
// globally unique.
static boolean wasLIFO(int N) {
  Set<Integer> set = new HashSet<>();
  boolean passed = true;
  for (int i=0; i<TH; i++) {
    int n = poppedValues[i].size();
    if (n != N) {
      log(i+": popped "+n+"/"+N+" values");
      passed = false;
    }
    for (Integer x : poppedValues[i])
      if (set.contains(x)) {
        log(i+": has duplicate value "+x);
        passed = false;
      }
    set.addAll(poppedValues[i]);
  }
  return passed;
}


@SuppressWarnings("unchecked")
static void testThreads(boolean backoff) {
  stack = new ArrayDeque<>();
  concurrentStack = new EliminationBackoffStack<>();
  poppedValues = new List[TH];
  for (int i=0; i<TH; i++)
    poppedValues[i] = new ArrayList<>();
  Thread[] threads = new Thread[TH];
  for (int i=0; i<TH; i++) {
    threads[i] = backoff?
      safe(i, i*NUM, NUM) :
      unsafe(i, i*NUM, NUM);
    threads[i].start();
    }
    try {
    for (int i=0; i<TH; i++)
      threads[i].join();
    }
    catch (Exception e) {}
  }


  public static void main(String[] args) {
    log("Starting "+TH+" threads with sequential stack");
    testThreads(false);
    log("Was LIFO? "+wasLIFO(NUM));
    log("");
    String name = "elimination backoff stack";
    log("Starting "+TH+" threads with "+name);
    testThreads(true);
    log("Was LIFO? "+wasLIFO(NUM));
    log("");
  }


  static void log(String x) {
    System.out.println(x);
  }
}
```

**OUTPUT: https://elimination-backoff-stack.wolfram77.repl.run**

```
Starting 10 threads with sequential stack
4: failed push
2: failed pop
3: failed pop
0: failed pop
5: failed pop
1: failed pop
0: popped 346/1000 values
1: popped 403/1000 values
2: popped 1/1000 values
2: has duplicate value 9881
3: popped 6/1000 values
3: has duplicate value 9654
3: has duplicate value 9652
4: popped 0/1000 values
5: popped 6/1000 values
```

```
5: has duplicate value 9359
7: has duplicate value 9247
Was LIFO? false


Starting 10 threads with elimination backoff stack
Was LIFO? true
```