CH11 SUMMARY

Concurrent Stacks and Elimination: A Complete Guide

Introduction to Stacks

A **stack** is a fundamental data structure that follows a Last-In-First-Out (LIFO) ordering principle. Think of it like a stack of plates - you can only add or remove plates from the top. The stack has two primary operations:

- **push(x)**: Adds an element to the top of the stack
- **pop()**: Removes and returns the element from the top of the stack

In everyday terms, imagine a stack of books. When you add a new book, it goes on top. When you need a book, you take the one on top - not from the middle or bottom.

Concurrent Stacks: The Challenge

When multiple threads (independent sequences of execution) try to access a stack simultaneously, we face the question: **Do stacks provide opportunity for concurrency?**

At first glance, the answer seems negative because:

- Both `push()` and `pop()` operations only access the **top** of the stack
- This creates a natural bottleneck where all operations must touch the same location

However, despite this apparent limitation, **stacks are not inherently sequential** - meaning we can design concurrent implementations that allow some degree of parallel execution.

Lock-Free Stack Implementation

Basic Structure

A lock-free stack uses a **linked-list implementation** where:

- A `top` field points to the first (top-most) node in the stack
- `pop()` on an empty list throws an exception
- `push()` creates a new node and links it to the current top

Lock-free means **atomic** - operations appear to happen instantaneously without locks. Instead, we use a special method called `compareAndSet` (CAS) to safely change link values in the list.

How compareAndSet Works

`compareAndSet` is an atomic operation that:

1. Checks if a value matches an expected value
2. If yes, updates it to a new value
3. Returns `true` if successful, `false` otherwise

This all happens atomically - meaning no other thread can interfere between the check and the update.

The tryPush Operation

```
public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node))
}
```

What this does step-by-step:

1. Read the current top node (`oldTop`)
2. Point the new node's `next` reference to `oldTop`
3. Try to atomically set `top` to point to the new node, but **only if** `top` still equals `oldTop`

When will tryPush return false?

The method returns `false` when there's **high contention** - meaning another thread successfully modified the `top` between when we read it and when we tried to update it. This indicates that multiple threads are competing to modify the stack simultaneously.

The Full Push Operation with Backoff

```
public void push(T value) {
    Node node = new Node(value);
    while (true) {
        if (tryPush(node)) {
            return;
        } else backoff.backoff();
    }
}
```

When `compareAndSet` returns `false`, it indicates high contention. In such scenarios, it's more effective to **backoff** (wait/pause) and try again later rather than immediately retrying. This is called **Exponential Backoff** - the thread waits for increasingly longer periods with each failed attempt.

Why backoff helps: If many threads constantly retry immediately, they create even more contention. By backing off, threads space out their attempts, reducing collisions.

The Pop Operation

```
public T pop() throws EmptyException {
    while (true) {
        Node returnNode = tryPop();
        if (returnNode != null)
            return returnNode.value;
        else
            backoff.backoff();
    }
}

protected Node tryPop() throws EmptyException {
    Node oldTop = top.get();
    if (oldTop == null)
        throw new EmptyException()
    Node newTop = oldTop.next;
    if (top.compareAndSet(oldTop, newTop))
        return oldTop;
    else
        return null;
}
```

How tryPop works:

1. Read the current top node
2. If it's null, the stack is empty - throw an exception
3. Get the next node (which will become the new top)
4. Try to atomically update `top` to point to the next node
5. If successful, return the old top; if not, return `null` and retry

Problems with Lock-Free Stack

Good aspects:

- No locking required - threads never block waiting for locks

Bad aspects:

- Even with backoff, there's **huge contention at the top**
- The stack's `top` field is a **sequential bottleneck**
- Method calls can only proceed **one after the other**
- **No real parallelism** - operations don't actually happen in parallel

- Exponential backoff alleviates contention but doesn't solve the fundamental bottleneck problem

The core issue: Every operation must successfully modify the same memory location (the `top` pointer), creating a serialization point.

The Key Observation: Elimination

Here's a critical insight that enables better performance:

If a `push()` is immediately followed by a `pop()`, the two operations cancel out and the stack doesn't change.

Example:

- Thread A pushes value 42
- Thread B pops and receives 42
- Net effect: The stack is unchanged

We can exploit this by allowing threads to meet "off the stack" and exchange values directly, never touching the actual stack data structure. This is called **elimination**.

How Elimination Works

Instead of both operations contending for the stack's top:

1. The thread calling `push()` exchanges its value with the thread calling `pop()`
2. The `push()` thread considers its push complete
3. The `pop()` thread receives the value directly
4. **No modification to the stack is needed**
5. The two calls **eliminate one another**

After equal numbers of pushes and pops that eliminate each other, the stack remains in the same state - this maintains **linearizability** (the correctness property for concurrent operations).

The Elimination Array

Core Concept

The **EliminationArray** is an array of meeting points where threads can attempt to find complementary operations to eliminate with.

How it works:

1. Threads pick **random array entries** when trying to eliminate
2. If a `push()` and `pop()` pick the **same location**, they can exchange values

3. If successful, both operations return without accessing the stack

Visual analogy: Imagine a parking lot with multiple spaces. Threads looking to exchange drive to random spots. If a pusher and popper happen to park at the same spot, they exchange items and drive away. If they park alone or meet someone doing the same operation, they eventually drive to the actual stack.

When Push Collides with Pop

When a push and pop successfully meet at the same array location:

- The push thread gives its value to the pop thread
- Both threads continue without accessing the stack
- This eliminates contention at the stack's top

When No Collision Occurs

If a thread fails to eliminate because:

- It picked an empty location (no partner)
- It met a thread doing the same operation (push meeting push, or pop meeting pop)

Then the thread accesses the actual stack instead.

Elimination as a Backoff Scheme

The EliminationArray serves as an intelligent backoff mechanism:

1. Each thread **first tries to access the stack directly**
2. **If that fails** (compareAndSet fails due to contention), it attempts to eliminate using the array
3. **If elimination fails**, it tries the stack again
4. This process repeats...

This approach is called an **EliminationBackoffStack** - combining stack access with elimination-based backoff.

Key advantage: Threads that would otherwise just wait during backoff can potentially do useful work by eliminating with complementary operations.

Exchangers: The Building Blocks

An **Exchanger** is an object that allows **exactly two threads** (and no more) to rendezvous and exchange values.

Why we need this: We must avoid a situation where a thread can make an exchange offer to more than one thread simultaneously - this would cause inconsistency.

LockFreeExchanger Design

A **LockFreeExchanger<T>** allows two threads to exchange values of type T:

- If thread A calls `exchange()` with value `a`
- And thread B calls `exchange()` with value `b`
- Then A returns `b` and B returns `a`

High-Level Exchange Protocol

The exchange process:

1. The **first thread** arrives, writes its value, and spins (busy-waits) until a second thread arrives
2. The **second thread** detects the first is waiting, reads its value, and signals the exchange is complete
3. The first thread may **timeout** if a second thread doesn't show up within a time limit

Three States of an Exchanger

The exchanger uses three states tracked by an **AtomicStampedReference** (which stores both a value and an integer "stamp" atomically):

1. **EMPTY**: No thread is currently using this exchanger
2. **WAITING**: One thread has placed its item and is waiting for a partner
3. **BUSY**: Two threads are currently exchanging items

The Exchange Algorithm: EMPTY State

When a thread encounters an EMPTY exchanger:

```
if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING))
```

Step 1: Try to place your item in the slot and change state to WAITING

Step 2: If successful, spin while waiting:

```
while (System.nanoTime() < timeBound) {
    yrItem = slot.get(stampHolder);
    if (stampHolder[0] == BUSY) {
        slot.set(null, EMPTY);
        return yrItem;
    }
}
```

- Keep checking if another thread changed the state to BUSY

- If BUSY, take their item, reset to EMPTY, and return

Step 3: Handle timeout:

```
if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
    throw new TimeoutException();
} else {
    yrItem = slot.get(stampHolder);
    slot.set(null, EMPTY);
    return yrItem;
}
```

- If time runs out, try to reset state to EMPTY
- If successful, timeout (no partner arrived)
- If failed, someone showed up at the last moment! Take their item and return it

Step 4: If initial CAS failed:

- Someone else changed state from EMPTY to WAITING concurrently
- Retry from the beginning

The Exchange Algorithm: WAITING State

When a thread encounters a WAITING exchanger:

```
if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY))
    return yrItem;
```

What this means: Someone is already waiting with their item:

1. Try to replace their item with yours and change state to BUSY
2. If successful, return their item (exchange complete!)
3. If failed, another thread beat you to it, so retry

The first thread (now seeing BUSY) will take your item and reset to EMPTY.

The Exchange Algorithm: BUSY State

```
case BUSY:
    break;
```

If the state is BUSY, two other threads are currently using this slot for an exchange. Simply retry (try another slot or come back later).

Visualization of Exchange Process

Step-by-step example:

1. **Thread A arrives**: Slot is EMPTY → A places item, sets WAITING
2. **Thread A waits**: Spinning and checking for state change
3. **Thread B arrives**: Slot is WAITING → B takes A's item, places its own, sets BUSY
4. **Thread A notices**: Sees BUSY → Takes B's item, resets to EMPTY, returns
5. **Thread B returns**: Already has A's item, operation complete

EliminationBackoffStack Implementation

The EliminationArray Structure

The EliminationArray is implemented as an **array of Exchanger objects**. When a thread wants to attempt elimination:

1. It picks an array entry **at random**
2. Calls that entry's `exchange()` method
3. Waits to see if a complementary operation shows up

The Complete Push Algorithm

```java
public void push(T value) {
    Node node = new Node(value);
    while (true) {
        if (tryPush(node)) {
            return;
        } else try {
            T otherValue = eliminationArray.visit(value);
            if (otherValue == null)
                return;
        } catch (TimeoutException e) {}
    }
}
```

Detailed flow:

1. **First attempt:** Try to push directly onto the stack using `tryPush(node)`
   - Returns `true` if successful (we're done!)
   - Returns `false` if high contention (another thread modified `top`)
2. **Elimination attempt:** If direct push failed, visit the elimination array with your value
   - Pick a random exchanger slot
   - Try to exchange with a pop operation
   - If `otherValue == null`, a pop thread took your value → success!
3. **Timeout:** If exchange times out (caught exception), retry from step 1

Key insight: For a push, receiving `null` from a pop thread means the pop took your value - the push is complete!

The Complete Pop Algorithm

```java
public T pop() throws EmptyException {
    while (true) {
        Node returnNode = tryPop();
        if (returnNode != null)
            return returnNode.value;
        else try {
            T otherValue = eliminationArray.visit(null);
            if (otherValue != null)
                return otherValue;
        } catch (TimeoutException e) {}
    }
}
```

Detailed flow:

1. **First attempt:** Try to pop directly from the stack using `tryPop()`
   - Returns a node if successful
   - Returns `null` if high contention or empty stack
2. **Elimination attempt:** If direct pop failed, visit elimination array with `null` as your value
   - Pick a random exchanger slot
   - Try to exchange with a push operation
   - If `otherValue != null`, a push thread gave you their value → success!
3. **Timeout:** If exchange times out, retry from step 1

Key insight: For a pop, receiving a non-null value means a push thread gave you their value - the pop is complete!

Why This Works

The EliminationBackoffStack achieves better performance because:

Choosing the right slot matters:

- If a thread picks a slot where state is BUSY, elimination fails (retry needed)
- If a thread collides with an unsuitable method (push-push or pop-pop), elimination fails

But successful eliminations:

- Remove operations from the stack bottleneck
- Allow multiple push-pop pairs to complete simultaneously in the elimination array
- Provide real parallelism that the basic lock-free stack cannot achieve

Failure Cases and Recovery

A thread fails to eliminate when:

1. **Empty slot timeout:** No partner arrived at that exchanger
2. **Wrong operation type:** Met another thread doing the same operation
3. **Busy slot:** Two other threads were already exchanging

In all cases, the thread simply retries - either attempting another elimination or accessing the stack directly.

Performance Characteristics

Low Contention Scenario

At **low levels of contention** (few threads competing):

- EliminationBackoffStack performance is **comparable** to LockFreeStack
- Most operations succeed on their first try accessing the stack
- Elimination overhead is minimal since there are few collisions

High Contention Scenario

At **high contention** (many threads competing):

- **Number of successful eliminations increases** significantly
- More operations complete in parallel through the elimination array
- **Much better performance** than LockFreeStack
- The stack's top is no longer a total bottleneck

Why it scales better:

- Multiple push-pop pairs can exchange simultaneously in different array slots
- Only operations that fail to eliminate need to access the actual stack
- This distributes the load across multiple memory locations

Key Concepts Summary

Sequential Bottleneck

A point in the algorithm where operations must happen one at a time, preventing parallelism. The stack's `top` pointer is a sequential bottleneck in basic implementations.

Linearizability

A correctness property ensuring that concurrent operations appear to occur in some sequential order consistent with real-time ordering. Elimination maintains linearizability because the net effect is the same as if operations occurred sequentially on the stack.

Exponential Backoff

A strategy where threads wait increasingly longer periods after each failed attempt, reducing contention by spacing out retries.

Lock-Free

A progress guarantee where at least one thread makes progress in a finite number of steps, even if other threads are delayed. Uses atomic operations like compareAndSet instead of locks.

Elimination

The technique of canceling out complementary operations (push-pop pairs) by having threads exchange values directly, bypassing the shared data structure.

Practical Implications

The EliminationBackoffStack demonstrates an important principle in concurrent programming: **sometimes the best way to reduce contention isn't to make access faster, but to provide alternative paths that bypass the bottleneck entirely.**

This approach trades some complexity (the elimination array and exchange protocol) for significant performance gains under high contention - a common trade-off in concurrent algorithm design.

The technique of elimination can be applied to other data structures beyond stacks, wherever complementary operations can cancel out without affecting the observable behavior of the structure.