LMAX Disruptor Pattern: High-Performance Inter-Thread Communication

A Deep Dive for Low-Latency Systems

Table of Contents

Overview

The **LMAX Disruptor** is a high-performance inter-thread communication library developed by the LMAX Exchange. It's designed for **low-latency**, **high-throughput** systems where traditional concurrency mechanisms (e.g., `BlockingQueue`) introduce unacceptable overhead.

Key Features

- **Lock-free Ring Buffer**: Eliminates locks by using a circular array (ring buffer).
- **Sequence-based Coordination**: Producers and consumers use sequence numbers to track progress.
- **Batching**: Supports batch processing of events for higher throughput.
- **Wait Strategies**: Customizable strategies for handling backpressure (e.g., `BusySpinWaitStrategy`).
- **Memory Barrier Optimization**: Prevents CPU reordering with memory fences.

Why Disruptor?

Problems with Traditional Queues

Issue	Impact
Lock Contention	Synchronization overhead limits throughput.
Memory Allocation	Frequent object creation causes GC pauses.

Issue	Impact
False Sharing	CPU cache line contention between threads.
Inefficient Batching	Hard to batch events for processing.

Disruptor's Solutions

1. **Preallocated Memory**: Ring buffer is preallocated to avoid GC.
2. **No Locks**: Uses atomic sequence counters for thread coordination.
3. **Cache-Aware Design**: Prevents false sharing with padding.
4. **Batch Processing**: Consumers process multiple events per cycle.

Core Concepts

1. Ring Buffer

- A **circular array** of fixed size (power of 2).
- Stores **events** (data objects) to be processed.
- **No dynamic resizing**—size is set at initialization.

2. Sequence Numbers

- **Sequence**: A monotonically increasing counter.
- **Producers** write to the next available slot using a `Sequence`.
- **Consumers** track their current position with a `Sequence`.

3. Wait Strategies

- `BusySpinWaitStrategy`: Fastest; burns CPU cycles.
- `SleepingWaitStrategy`: Sleeps between polls (lower CPU usage).
- `YieldingWaitStrategy`: Balances CPU and latency.

4. Event Processors

- **Single Producer, Single Consumer (SPSC)**
- **Multiple Producers, Single Consumer (MPSC)**
- **Dependency Chains**: Consumers can depend on other consumers.

Architecture Diagrams

1. Basic Disruptor Architecture

```
+--------------------+       +--------------------+       +--------------------+
|                    |       |                    |       |                    |
|   Producer         |------>|   Ring Buffer      |<------|   Consumer
```

```
|
| (Event Publisher) |       | (Fixed-Size Array)|       | (Event
Processor) |
|                   |       |                   |       |
|
+-------------------+       +-------------------+       +------------------
--+
```

2. Sequence Coordination

```
Ring Buffer (Size = 8)
Slots: [0][1][2][3][4][5][6][7]

Producer Sequence: 7
Consumer Sequence: 3

Next Available Slot: (Producer Sequence + 1) % 8 = 0
```

3. False Sharing Prevention

```java
class PaddedLong {
    @Contended
    private volatile long value;
    // Padding added automatically with @Contended
}
```

How It Works

Step-by-Step Workflow

1. **Initialization**:
    - Create a ring buffer with fixed size (e.g., 1024).
    - Preallocate event objects.
2. **Producer Publishes Events**:
    - Get the next available sequence.
    - Write data to the ring buffer.
    - Publish the sequence.
3. **Consumer Processes Events**:
    - Wait for the next available sequence.
    - Process the event.
    - Update the consumer's sequence.

4. **Batching**:
   - Consumers can process multiple events per cycle.

Code Implementation (Java)

1. Define Event Class

```java
public class TradeEvent {
    private long tradeId;
    private String symbol;
    private double price;

    // Getters and setters
}
```

2. Create Disruptor

```java
import com.lmax.disruptor.*;
import com.lmax.disruptor.dsl.Disruptor;
import com.lmax.disruptor.util.DaemonThreadFactory;

public class DisruptorExample {
    public static void main(String[] args) {
        // Ring buffer size (power of 2)
        int bufferSize = 1024;

        // Create Disruptor with 4 consumer threads
        Disruptor<TradeEvent> disruptor = new Disruptor<>(
            TradeEvent::new,
            bufferSize,
            DaemonThreadFactory.INSTANCE,
            ProducerType.SINGLE,
            new BusySpinWaitStrategy()
        );

        // Define event processor
        disruptor.handleEventsWith((event, sequence, endOfBatch) -> {
            System.out.println("Processing: " + event.getTradeId());
        });

        // Start the disruptor
        disruptor.start();

        // Get the ring buffer
        RingBuffer<TradeEvent> ringBuffer = disruptor.getRingBuffer();

        // Publish events
        for (long i = 0; i < 1000000; i++) {
```

```
            long sequence = ringBuffer.next(); // Get next available slot
            try {
                TradeEvent event = ringBuffer.get(sequence);
                event.setTradeId(i);
                event.setSymbol("AAPL");
                event.setPrice(150.0);
            } finally {
                ringBuffer.publish(sequence); // Publish the event
            }
        }

        disruptor.shutdown(); // Graceful shutdown
    }
}
```

Performance Advantages

Metric	Disruptor	BlockingQueue
Throughput (events/sec)	100M+	10M–100M
Latency (µs)	<1	10–100
CPU Utilization	Lower (no locks)	Higher (locks, GC)
Memory Allocation	None (preallocated)	Frequent (GC pressure)

Use Cases

1. **Financial Trading Systems**: Real-time order matching.
2. **High-Frequency Data Ingestion**: Sensor data from IoT devices.
3. **Real-Time Analytics**: Streaming user behavior analysis.
4. **Game Engines**: Fast event handling for player actions.

Comparison with Traditional Queues

1. `BlockingQueue` Limitations

- Uses locks internally (e.g., `ReentrantLock`).
- Dynamic resizing causes GC overhead.
- No batching support.

2. Disruptor Advantages

- **No locks**: Uses atomic sequences.
- **Batching**: Consumers process multiple events per cycle.
- **Cache-line padding**: Prevents false sharing.

Advanced Topics

1. Dependency Chains

```
disruptor.handleEventsWith(
    new EventHandler1(),
    new EventHandler2(),
    new EventHandler3()
);
```

2. Multiple Consumers

```
disruptor.handleEventsWithWorkerPool(
    new WorkerPool<>(factory, new IgnoreExceptionHandler(), handler1,
handler2)
);
```

3. Custom Wait Strategies

```
public class CustomWaitStrategy implements WaitStrategy {
    @Override
    public long waitFor(long sequence, Sequence cursor, Sequence
dependentSequence, Timeout timeout) {
        // Custom logic
    }
}
```

FAQs

Q1: Why must the ring buffer size be a power of 2?

A1: Allows fast modulo operations using bitmask:

```
sequence & (bufferSize - 1)
```

Q2: How to handle backpressure?

A2: Use `YieldingWaitStrategy` or `SleepingWaitStrategy` to slow down producers.

Q3: Can multiple producers publish events?

A3: Yes, but use `ProducerType.MULTI` and ensure the sequence generator is thread-safe.