Java Concurrency and Multithreading - Comprehensive Documentation

Table of Contents

Introduction

This documentation covers a comprehensive collection of Java concurrency problems and their solutions. Each problem demonstrates different aspects of multithreading, synchronization, and concurrent programming patterns. The solutions progress from basic implementations to advanced enhancements, showcasing various concurrency primitives and techniques.

Learning Objectives

- Master Java concurrency primitives (`ReentrantLock`, `Condition`, `Semaphore`, etc.)
- Understand classic synchronization problems and their solutions
- Learn progressive enhancement techniques
- Handle edge cases in concurrent systems
- Implement thread-safe data structures
- Design scalable concurrent systems

Core Concurrency Primitives

ReentrantLock

A reentrant mutual exclusion lock with more flexibility than `synchronized`. Key features: - **Reentrancy**: A thread can acquire the same lock multiple times - **Fairness**: Optional fair ordering (FIFO) with `ReentrantLock(true)` - **Condition Variables**: Support multiple wait queues via `Condition` objects - **Try-lock**: Non-blocking lock attempts with `tryLock()`

Condition Variables

Wait queues associated with a lock. Used for coordinating threads: - `await()`: Release lock and wait until signaled - `signal()`: Wake up one waiting thread - `signalAll()`: Wake up all waiting threads - **Always check conditions in while loops** to handle spurious wakeups

Key Patterns

1. **Lock-Condition Pattern**: `lock.lock() → while(condition) await() → work → signal() → lock.unlock()`
2. **Try-Finally Pattern**: Always unlock in finally block
3. **Predicate Loop**: `while(condition not satisfied) { condition.await(); }`

Problems and Solutions

---

1. Unisex Bathroom Problem

Problem Statement

Design a thread-safe unisex bathroom system where: - Multiple people of the same gender can use the bathroom simultaneously - When someone of one gender is inside, people of the opposite gender must wait - No starvation - both genders should eventually get access

Real-World Applications

- Gender-specific resource allocation
- Resource sharing with mutual exclusion between groups
- Conference room booking with team exclusivity

Base Solution: `Bathroom.java`

Algorithm Overview The bathroom maintains two counters (`menInside`, `womenInside`) and uses a `ReentrantLock` with two `Condition` variables: - `menWait`: Condition for waiting men - `womenWait`: Condition for waiting women

Key Logic **Entry Protocol:**

```java
public void manEnter() throws InterruptedException {
    lock.lock();
    try {
        while(womenInside > 0) {  // Predicate loop
            menWait.await();
        }
        menInside++;
    } finally {
        lock.unlock();
    }
}
```

- Men wait in a loop while women are inside
- The loop protects against spurious wakeups
- Increment counter after wait condition is satisfied

Exit Protocol:

```java
public void manExit() throws InterruptedException {
    lock.lock();
    try {
        menInside--;
        if(menInside == 0) {  // Last man out
            womenWait.signalAll();  // Wake all waiting women
        }
    } finally {
        lock.unlock();
    }
}
```

- Decrement counter on exit
- When last person exits, signal waiting opposite gender
- `signalAll()` wakes all waiters (they'll re-check condition)

Concurrency Tools Used

- **ReentrantLock**: Mutual exclusion for counter updates
- **Condition Variables**: Separate wait queues for men and women
- **Predicate Loop**: `while(condition)` ensures correctness

Edge Cases Handled

- Spurious wakeups: Handled by while-loop condition checking
- Multiple threads entering: Atomic counter increments under lock
- Last person exiting: Signals opposite gender

Limitations

- **Starvation Possible**: If men keep entering, women might wait indefinitely (no fairness)
- **No Priority**: All users treated equally

Enhancement 1: Fairness with Turn-Based Scheduling

Problem Addressed The base solution can lead to starvation - one gender might monopolize the bathroom.

Solution: `BathroomEnhancement1.java` Introduces a **turn-based system** to prevent starvation:

```
private String turn = "WOMEN";  // Whose turn is it?
private int menWaiting = 0;      // Count of waiting men
private int womenWaiting = 0;    // Count of waiting women
```

Enhanced Entry Logic:

```
public void manEnter() throws InterruptedException {
    lock.lock();
    try {
        menWaiting++;  // Track waiting count
        while(womenInside > 0 ||
              ("WOMEN".equals(turn) && womenWaiting > 0)) {
            condition.await();
        }
        menWaiting--;
        menInside++;
    } finally {
        lock.unlock();
    }
}
```

Key Innovation: - Men must wait if: 1. Women are inside (mutual exclusion), OR 2. It's women's turn AND there are women waiting (fairness)

Turn Transfer:

```
public void manExit() throws InterruptedException {
    lock.lock();
    try {
        menInside--;
        if(menInside == 0) {
            turn = "WOMEN";  // Give turn to women
        }
        condition.signalAll();
    } finally {
        lock.unlock();
    }
}
```

Fairness Guarantee

- Turn alternates between genders
- Prevents indefinite starvation
- Uses single `Condition` with `signalAll()`

Enhancement 2: Multi-Group Support

Problem Addressed Support more than two groups (e.g., different categories/castes) with fairness.

Solution: `BathroomEnhancement2.java` Generalizes to **arbitrary groups** using dynamic tracking:

```
private String currentGroup = null;  // Group currently inside
private int inside = 0;              // Total people inside
private Map<String, Integer> waiting = new HashMap<>();  // Waiting per group
private String turnGroup = null;     // Group whose turn it is
```

Dynamic Group Entry:

```
public void enter(String group) throws InterruptedException {
    lock.lock();
    try {
        waiting.put(group, waiting.getOrDefault(group, 0) + 1);

        // Wait if:
        while((inside > 0 && !group.equals(currentGroup)) ||  // Different group inside
              (inside == 0 && turnGroup != null && !turnGroup.equals(group))) {  // Not our
            condition.await();
        }

        waiting.put(group, waiting.get(group) - 1);
        if (inside == 0) {
            currentGroup = group;  // Set current group
        }
        inside++;
    } finally {
        lock.unlock();
    }
}
```

Next Group Selection:

```
private String nextGroup() {
    for (Map.Entry<String, Integer> entry : waiting.entrySet()) {
        if (entry.getValue() > 0) {  // Find first group with waiters
            return entry.getKey();
        }
    }
    return null;
}
```

Key Features

- **Dynamic Groups**: Support any number of groups at runtime
- **Fair Rotation**: Iterates through waiting groups
- **Flexible**: Can add new groups without code changes

Enhancement 3: Multiple Bathrooms

Concept: Scale to multiple bathroom instances with independent locking.

Implementation Approach: - Each `BathroomUnit` has its own lock - Load balancing across bathrooms - Reduces contention

---

Enhancement 4: VIP Priority

Concept: VIP users get priority over normal users within the same gender.

Implementation Strategy: - Maintain separate priority queues (VIP vs Normal) - Process VIP queue first when bathroom becomes available - Use `PriorityQueue` or custom queue ordering

---

Summary: Bathroom Problem

Progression

1. **Base**: Basic mutual exclusion between genders
2. **Enhancement 1**: Fairness with turn-based scheduling

3. **Enhancement 2**: Generalization to multiple groups
4. **Enhancement 3**: Horizontal scaling with multiple units
5. **Enhancement 4**: Priority-based access

Key Takeaways

- **Predicate Loops**: Essential for correctness (`while(condition)`)
- **Signal Strategy**: Use `signalAll()` when multiple threads might be interested
- **Fairness vs Performance**: Fairness adds coordination overhead
- **Scalability**: Multiple independent units reduce contention

---

2. Ordered Printing (Odd/Even)

Problem Statement

Create multiple threads that print numbers in sequential order: - Base: Two threads (odd/even) print 1, 2, 3, 4, … - Enhancement: N threads print in round-robin fashion - Enhancement: Support pause/resume functionality

Real-World Applications

- Turn-based systems
- Sequential task execution across threads
- Coordinated progress tracking

Base Solution: `OrderedPrint1.java`

Two threads coordinate to print numbers in order.

Algorithm

```java
private int MAX = 100;
private int num = 1;
private ReentrantLock lock = new ReentrantLock();
private Condition cond = lock.newCondition();
```

Odd Thread:

```java
public void printOdd() throws Exception {
    while (true) {
        lock.lock();
        try {
            while (num <= MAX && num % 2 == 0) {  // Wait while even
                cond.awaitUninterruptibly();
            }
            if (num > MAX) return;  // Termination

            System.out.println("Odd : " + num);
            num++;
            cond.signalAll();  // Wake even thread
        } finally {
            lock.unlock();
        }
    }
}
```

Even Thread:

```java
public void printEven() {
    while(true) {
        lock.lock();
        try {
            while (num <= MAX && num %2 == 1) {  // Wait while odd
                cond.awaitUninterruptibly();
            }
            if (num > MAX) return;

            System.out.println("Even : " + num);
            num++;
            cond.signalAll();  // Wake odd thread
        } finally {
            lock.unlock();
        }
    }
}
```

Key Concepts

1. **Turn-Based Coordination**: Each thread checks if it's their turn (`num % 2`)
2. **Predicate Loop**: `while(condition)` protects against spurious wakeups
3. **Shared Counter**: `num` is the shared state
4. **Signal Pattern**: Each thread signals after making progress

Why `signalAll()` Instead of `signal()`?

- With only 2 threads, both work
- `signalAll()` is safer for extensibility
- Ensures all waiting threads check their condition

Enhancement 1: Configurable N Threads

Solution: `OrderedPrint2.java`

Generalize to N threads with round-robin turn assignment.

```
private int maxThreads;

public void worker(int id) {
    while (true) {
        lock.lock();
        try {
            // Wait while it's not our turn
            while (num <= MAX && (num-1) % maxThreads != id) {
                condition.awaitUninterruptibly();
            }

            if (num > MAX) return;

            System.out.println("ThreadId : " + id + " --- num : " + num);
            num++;
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

Turn Calculation **Formula**: `(num - 1) % maxThreads == id`

- `num = 1` → `(1-1) % N = 0` → Thread 0's turn
- `num = 2` → `(2-1) % N = 1` → Thread 1's turn
- `num = N+1` → `(N+1-1) % N = 0` → Back to Thread 0

Scalability

- Works for any number of threads
- Each thread checks its specific turn condition
- Single condition variable with `signalAll()`

Enhancement 2: Pause/Resume Support

Solution: `OrderedPrint3.java`

Add external control to pause and resume all threads.

```
private boolean paused = false;
private final Condition pauseCond = lock.newCondition();
```

```
public void worker(int id) {
    while (true) {
        lock.lock();
        try {
            // First check pause state
            while (paused) {
                pauseCond.awaitUninterruptibly();
            }

            // Then check turn
            while (num <= MAX && (num-1) % maxThreads != id) {
                condition.awaitUninterruptibly();
            }

            if (num > MAX) return;

            System.out.println("ThreadId : " + id + " --- num : " + num);
            num++;
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

Control Methods:

```
public void pause() {
    lock.lock();
    try {
        paused = true;
    } finally {
        lock.unlock();
    }
}

public void resume() {
    lock.lock();
    try {
        paused = false;
        pauseCond.signalAll();  // Wake all paused threads
    } finally {
        lock.unlock();
    }
}
```

Design Choice: Separate Condition Why separate `pauseCond`? - Clean Separation: Pause state is independent of turn logic - **Efficiency**: Don't mix predicates - clearer signal semantics - **Maintainability**: Easy to reason about two separate conditions

Condition Checking Order

1. **First check pause**: All threads must respect pause
2. **Then check turn**: Only proceed if it's our turn

This order ensures pause takes precedence.

Summary: Ordered Printing

Progression

1. **Base**: Two threads (odd/even) with binary turn
2. **Enhancement 1**: N threads with modulo-based turn calculation
3. **Enhancement 2**: External pause/resume control

Key Patterns

- **Turn-Based Coordination**: `(num - 1) % N == threadId`
- **Predicate Loops**: Always use `while(condition)` not `if(condition)`
- **Separate Conditions**: Use multiple conditions for independent predicates
- **Broadcast Signals**: `signalAll()` for multiple waiters

Performance Considerations

- **Contention**: All threads compete for same lock (serialized progress)
- **Signal Overhead**: `signalAll()` wakes all threads (most go back to sleep)
- **Scalability**: Limited by single lock - not suitable for high-performance scenarios

---

3. Thread-Safe Bounded Blocking Queue

Problem Statement

Implement a fixed-capacity queue with blocking operations: - `put()`: Block if queue is full - `get()`: Block if queue is empty - Thread-safe for concurrent producers and consumers

Real-World Applications

- Producer-consumer patterns
- Task queues
- Buffer between components with different speeds

Base Solution: `BoundedBlockingQueue1.java`

Classic bounded buffer implementation.

```
protected int capacity;
protected Deque<T> queue = new ArrayDeque<>();
protected ReentrantLock lock = new ReentrantLock();
protected Condition notFull = lock.newCondition();
protected Condition notEmpty = lock.newCondition();
```

Put Operation

```
public void put(T item) throws Exception {
    lock.lock();
    try {
        while (queue.size() == capacity) {  // Full?
            notFull.await();  // Wait for space
        }
        queue.addLast(item);
        notEmpty.signal();  // Wake one consumer
    } finally {
        lock.unlock();
    }
}
```

Logic Flow: 1. Acquire lock 2. Wait while queue is full (predicate loop) 3. Add item when space available 4. Signal one waiting consumer 5. Release lock

Get Operation

```
public T get() throws Exception {
    lock.lock();
    try {
        while(queue.isEmpty()) {  // Empty?
            notEmpty.await();  // Wait for item
        }
        T item = queue.removeFirst();
        notFull.signal();  // Wake one producer
    } finally {
        lock.unlock();
    }
}
```

Logic Flow: 1. Acquire lock 2. Wait while queue is empty (predicate loop) 3. Remove item when available 4. Signal one waiting producer 5. Release lock

---

Concurrency Tools Used

1. **ReentrantLock**: Protects queue operations
2. **Two Condition Variables**:
   - `notFull`: For waiting producers
   - `notEmpty`: For waiting consumers
3. **Predicate Loops**: `while(condition)` for correctness

Why Two Conditions?

Design Rationale: - **Efficiency**: `notFull.signal()` only wakes producers, `notEmpty.signal()` only wakes consumers - **Targeted Wakeups**: Don't wake threads that can't make progress - **Performance**: Reduces unnecessary context switches

Alternative: Single Condition - Would need `signalAll()` every time (less efficient) - Wakes both producers and consumers (wasteful)

Why `signal()` Instead of `signalAll()`?

```
notEmpty.signal();  // Wake ONE consumer
```

Reasoning: - We added exactly ONE item - Only ONE consumer can take it - No need to wake multiple consumers - **Optimization**: Reduces unnecessary wakeups

When to use `signalAll()`: - When multiple threads might be able to make progress - When condition change affects multiple waiters - For simplicity/correctness (at performance cost)

Enhancement 1: Timeout Support

Feature: `put(timeout)` and `get(timeout)` with time limits.

```
public boolean put(T item, long timeoutMs) throws Exception {
    lock.lock();
    try {
```

```java
        long nanosRemaining = TimeUnit.MILLISECONDS.toNanos(timeoutMs);

        while (queue.size() == capacity) {
            if (nanosRemaining <= 0) {
                return false;  // Timeout
            }
            nanosRemaining = notFull.awaitNanos(nanosRemaining);
        }

        queue.addLast(item);
        notEmpty.signal();
        return true;
    } finally {
        lock.unlock();
    }
}
```

Timeout Handling Pattern **Key Points:** 1. **`awaitNanos()`** returns remaining time 2. **Loop Refinement**: Subtract elapsed time from remaining 3. **Check on Each Iteration**: Timeout might expire mid-wait

Why Track Remaining Time? - `awaitNanos()` can return early (spurious wakeup) - Must re-wait with remaining time - Ensures total wait doesn't exceed timeout

---

Enhancement 2: Fairness

Problem: With unfair lock, threads might be starved.

Solution: Use fair lock

```java
protected ReentrantLock lock = new ReentrantLock(true);  // Fair mode
```

Fairness Implications **Benefits:** - **No Starvation**: FIFO ordering guarantees eventual access - **Predictability**: Threads acquire lock in order

Costs: - **Performance**: Fair locks have higher overhead - **Throughput**: Reduced compared to unfair mode

When to Use: - When fairness is critical requirement - When starvation must be prevented - Acceptable to trade performance for predictability

---

Enhancement 3: Metrics

Feature: Track queue statistics for monitoring.

```
private long addedCount = 0;
private long takenCount = 0;
private int maxWaitingProducers = 0;
private int maxWaitingConsumers = 0;
```

Usage: - Monitor throughput (`addedCount`, `takenCount`) - Detect bottlenecks (waiting producer/consumer counts) - Capacity planning (max waiting threads)

Summary: Bounded Blocking Queue

Key Concepts

- **Producer-Consumer Pattern**: Decouples producers from consumers
- **Backpressure**: Blocking when full provides natural backpressure
- **Two-Condition Pattern**: Separate conditions for different wait reasons

Design Patterns

1. **Mesa Monitor**: `while(condition) await()` pattern
2. **Targeted Signaling**: `signal()` vs `signalAll()`
3. **Timeout Pattern**: `awaitNanos()` with remaining time tracking

Performance Characteristics

- **Throughput**: Limited by single lock (all operations serialized)
- **Latency**: Low when queue not full/empty, high when blocking
- **Scalability**: Not ideal for high-concurrency scenarios

---

4. Producer-Consumer Pipeline

Problem Statement

Build a processing pipeline where: - Producers generate tasks - Consumers process tasks from a queue - Support multi-stage processing - Handle shutdown gracefully - Implement retry mechanisms

Real-World Applications

- ETL (Extract-Transform-Load) pipelines
- Message processing systems
- Stream processing frameworks

Base Solution: `Pipeline1.java` with `SimpleBlockingQueue`

SimpleBlockingQueue Implementation Identical to `BoundedBlockingQueue1` with minor API differences:

```
public class SimpleBlockingQueue<T> {
    private int capacity;
    private Deque<T> q = new ArrayDeque<>();
    private ReentrantLock lock = new ReentrantLock();
    private Condition notEmpty = lock.newCondition();
    private Condition notFull = lock.newCondition();

    public void put(T task) throws Exception { /* ... */ }
    public T take() throws Exception { /* ... */ }
}
```

Simple Pipeline

```
SimpleBlockingQueue<Task> queue = new SimpleBlockingQueue<>(5);

Thread producer = new Thread(() -> {
    for (int i = 0; i < 20; i++) {
        queue.put(new Task(i));
        System.out.println("Produced : " + i);
    }
});

Thread consumer = new Thread(() -> {
    while(true) {
        Task t = queue.take();
        System.out.println("Consumed : " + t.id);
        // Process task
    }
});
```

Pattern

- **Single Producer**: Creates tasks sequentially
- **Single Consumer**: Processes from queue
- **Blocking Queue**: Provides backpressure

Enhancement 1: Multi-Stage Pipeline

Concept: Chain multiple processing stages with intermediate queues.

Architecture:

```
Producer → [Queue1] → Parser → [Queue2] → Transformer → [Queue3] → Writer
```

Implementation Pattern:

```java
SimpleBlockingQueue<RawTask> queue1 = new SimpleBlockingQueue<>(10);
SimpleBlockingQueue<ParsedTask> queue2 = new SimpleBlockingQueue<>(10);
SimpleBlockingQueue<TransformedTask> queue3 = new SimpleBlockingQueue<>(10);

// Producer
Thread producer = new Thread(() -> {
    for (RawTask task : generateTasks()) {
        queue1.put(task);
    }
});

// Parser Stage
Thread parser = new Thread(() -> {
    while (true) {
        RawTask raw = queue1.take();
        ParsedTask parsed = parse(raw);
        queue2.put(parsed);
    }
});

// Transformer Stage
Thread transformer = new Thread(() -> {
    while (true) {
        ParsedTask parsed = queue2.take();
        TransformedTask transformed = transform(parsed);
        queue3.put(transformed);
    }
});

// Writer Stage
Thread writer = new Thread(() -> {
    while (true) {
        TransformedTask task = queue3.take();
        write(task);
    }
});
```

Benefits

- **Modularity**: Each stage is independent
- **Parallelism**: Stages run concurrently
- **Backpressure**: Each queue provides buffering and flow control
- **Scalability**: Can have multiple workers per stage

Enhancement 2: Graceful Shutdown (Poison Pill)

Problem: How to stop pipeline cleanly?

Solution: Poison Pill Pattern

```
static final Task POISON_PILL = new Task(-1);  // Sentinel value

// Producer signals completion
Thread producer = new Thread(() -> {
    for (int i = 0; i < 20; i++) {
        queue.put(new Task(i));
    }
    queue.put(POISON_PILL);  // Signal end
});

// Consumer detects poison pill
Thread consumer = new Thread(() -> {
    while (true) {
        Task t = queue.take();
        if (t == POISON_PILL) {  // Shutdown signal
            break;
        }
        process(t);
    }
});
```

Multi-Stage Poison Pill

```
// Each stage forwards poison pill
Thread parser = new Thread(() -> {
    while (true) {
        RawTask task = queue1.take();
        if (task == POISON_PILL) {
            queue2.put(POISON_PILL);  // Forward to next stage
            break;
        }
        queue2.put(parse(task));
    }
});
```

Alternative: Interrupt-Based Shutdown

```
Thread consumer = new Thread(() -> {
    try {
        while (!Thread.currentThread().isInterrupted()) {
            Task t = queue.take();  // Will throw InterruptedException
            process(t);
        }
    } catch (InterruptedException e) {
        // Cleanup and exit
    }
});

// Shutdown
consumer.interrupt();
```

Comparison:

Approach	Pros	Cons
Poison Pill	Clean shutdown, processes all queued items	Needs sentinel value
Interrupt	Immediate stop, built-in mechanism	Might lose queued work

---

Enhancement 3: Retry Mechanism

Feature: Retry failed tasks with max attempts and dead letter queue.

```
class TaskWithRetry {
    Task task;
    int retryCount;
    int maxRetries = 3;
}

SimpleBlockingQueue<TaskWithRetry> mainQueue = new SimpleBlockingQueue<>(10);
SimpleBlockingQueue<Task> deadLetterQueue = new SimpleBlockingQueue<>(100);

Thread consumer = new Thread(() -> {
    while (true) {
        TaskWithRetry taskWithRetry = mainQueue.take();

        try {
            process(taskWithRetry.task);
        } catch (Exception e) {
            taskWithRetry.retryCount++;

            if (taskWithRetry.retryCount < taskWithRetry.maxRetries) {
                // Retry
                mainQueue.put(taskWithRetry);
            } else {
                // Max retries exceeded - send to dead letter queue
                deadLetterQueue.put(taskWithRetry.task);
            }
        }
    }
});
```

Retry Strategies

1. **Immediate Retry**: Re-queue immediately (shown above)
2. **Delayed Retry**: Sleep before re-queueing
3. **Exponential Backoff**: Increasing delay between retries
4. **Separate Retry Queue**: Different queue for retries

Dead Letter Queue **Purpose:** - Stores tasks that failed all retries - Allows manual inspection/reprocessing - Prevents infinite retry loops

---

Summary: Producer-Consumer Pipeline

Key Patterns

1. **Blocking Queue**: Foundation for producer-consumer
2. **Multi-Stage Pipeline**: Chain processing stages
3. **Poison Pill**: Graceful shutdown signal
4. **Retry Logic**: Handle transient failures

Design Considerations

- **Queue Sizing**: Balance memory vs throughput
- **Backpressure**: Bounded queues provide natural flow control
- **Error Handling**: Decide retry strategy and failure handling
- **Shutdown**: Clean vs immediate termination trade-offs

---

5. Hit Counter

Problem Statement

Count hits (events) within a sliding time window: - Example: Count hits in last 5 minutes (300 seconds) - Support high-throughput concurrent operations - Accurate sliding window (not fixed window)

Real-World Applications

- Rate limiting
- API request tracking
- Metrics collection
- Fraud detection (e.g., "too many login attempts")

Base Solution: `HitCounter.java`

Uses a **circular buffer** approach with time-bucket granularity.

```java
private int WINDOW = 300;  // 5 minutes in seconds
private int[] times = new int[WINDOW];   // Timestamp for each bucket
private int[] counts = new int[WINDOW];  // Hit count for each bucket

private Lock lock = new ReentrantLock();
```

Hit Recording

```java
public void hit(int timestamp) {
    lock.lock();
    try {
        int idx = timestamp % WINDOW;  // Map to bucket

        if (times[idx] != timestamp) {
            // Bucket is stale - reset it
            times[idx] = timestamp;
            counts[idx] = 1;
        } else {
            // Same timestamp - increment
            counts[idx]++;
        }
    } finally {
        lock.unlock();
    }
}
```

Get Hits in Window

```java
public int getHits(int timestamp) {
    lock.lock();
    try {
        int total = 0;
        for (int i = 0; i < WINDOW; i++) {
            if (timestamp - times[i] < WINDOW) {  // Within window?
                total += counts[i];
            }
        }
        return total;
    } finally {
        lock.unlock();
    }
}
```

Algorithm Explanation

Circular Buffer Mapping

- `idx = timestamp % WINDOW` maps timestamp to bucket index
- Same second always maps to same bucket
- Window size = number of buckets

Bucket Staleness Detection

```
if (times[idx] != timestamp) {
    // Bucket holds data from timestamp - WINDOW seconds ago
    // Reset it with new timestamp
    times[idx] = timestamp;
    counts[idx] = 1;
}
```

Example (WINDOW = 300): - At `t=5`: Bucket 5 stores time=5, count=10 - At `t=305`: Bucket 5 (305%300=5) is reused, time=305, count reset

Why This Works

- **Time-based Invalidation**: Old buckets automatically overwritten
- **O(1) Write**: Single bucket update
- **O(WINDOW) Read**: Scan all buckets (could optimize with running total)

Concurrency Analysis

Single Lock Approach **Pros:** - Simple and correct - No complex coordination

Cons: - **Serialization**: All operations serialized (contention bottleneck) - **Read Scalability**: Readers block each other and writers

---

Enhancement 1: Per-User Hit Counters

Feature: Track hits per user/key independently.

```
private ConcurrentHashMap<String, HitCounter> counters = new ConcurrentHashMap<>();

public void hit(String userId, int timestamp) {
    HitCounter counter = counters.computeIfAbsent(userId,
        k -> new HitCounter());
    counter.hit(timestamp);
}
```

```java
public int getHits(String userId, int timestamp) {
    HitCounter counter = counters.get(userId);
    return counter != null ? counter.getHits(timestamp) : 0;
}
```

Using `ConcurrentHashMap`

- **Thread-Safe**: Built-in concurrency support
- **computeIfAbsent**: Atomic "get or create" operation
- **No Global Lock**: Per-user counters are independent

Enhancement 2: Lock Striping

Problem: Single lock creates contention bottleneck.

Solution: Use multiple locks (lock striping).

```java
private static final int NUM_LOCKS = 8;
private ReentrantLock[] locks = new ReentrantLock[NUM_LOCKS];
private int[][] times = new int[NUM_LOCKS][WINDOW/NUM_LOCKS];
private int[][] counts = new int[NUM_LOCKS][WINDOW/NUM_LOCKS];

public void hit(int timestamp) {
    int stripe = timestamp % NUM_LOCKS;  // Which lock?
    locks[stripe].lock();
    try {
        int idx = (timestamp / NUM_LOCKS) % (WINDOW / NUM_LOCKS);
        // ... update logic ...
    } finally {
        locks[stripe].unlock();
    }
}
```

Lock Striping Benefits

- **Reduced Contention**: Different timestamps use different locks
- **Increased Throughput**: Parallel operations on different stripes
- **Trade-off**: More complex, slight memory overhead

Choosing Stripe Count

- **Too Few**: Still have contention
- **Too Many**: Lock overhead, cache effects
- **Rule of Thumb**: 2-16 stripes for most cases

Enhancement 3: Blocking Wait for Threshold

Feature: `waitForHits(threshold)` blocks until threshold reached.

```java
private Condition thresholdReached = lock.newCondition();
private int totalHits = 0;

public void hit(int timestamp) {
    lock.lock();
    try {
        // ... existing hit logic ...
        totalHits++;
        thresholdReached.signalAll();  // Wake waiting threads
    } finally {
        lock.unlock();
    }
}

public void waitForHits(int threshold, int timestamp) throws InterruptedException {
    lock.lock();
    try {
        while (getHits(timestamp) < threshold) {
            thresholdReached.await();
        }
    } finally {
        lock.unlock();
    }
}
```

Use Case

- Wait until N requests received before processing batch
- Threshold-based triggering

Summary: Hit Counter

Key Techniques

1. **Circular Buffer**: Time-based bucket reuse
2. **Lock Striping**: Reduce contention with multiple locks
3. **ConcurrentHashMap**: Per-key independent counters
4. **Condition Variables**: Threshold-based blocking

Performance Optimization

- **Write Path**: O(1) with lock striping
- **Read Path**: O(WINDOW) - could optimize with cumulative counters

- **Scalability**: Lock striping + per-user counters enable horizontal scaling

---

6. Rate Limiter

Problem Statement

Limit request rate to prevent system overload: - Allow maximum N requests per time window - Different algorithms: Fixed Window, Sliding Window, Token Bucket - Support blocking/non-blocking modes - Handle per-user/per-key rate limits

Real-World Applications

- API rate limiting
- DDoS protection
- Resource quota enforcement
- Traffic shaping

Algorithm 1: Fixed Window

Concept: Count requests in fixed time windows.

```java
public class FixedWindowRateLimiter {
    private int maxRequests;
    private long windowMs;
    private int count = 0;
    private long windowStart;

    public synchronized boolean allow() {
        long now = System.currentTimeMillis();

        if (now - windowStart >= windowMs) {
            // New window
            windowStart = now;
            count = 0;
        }

        if (count < maxRequests) {
            count++;
            return true;
        }
        return false;
    }
}
```

Pros/Cons **Pros:** - Simple implementation - Low memory footprint

Cons: - **Burst at Boundary**: 2x rate possible at window edges - Example: 100 requests at t=59s, 100 at t=61s = 200 in 2 seconds

---

Algorithm 2: Sliding Window

Concept: Track exact timestamps of requests.

```java
public class SlidingWindowRateLimiter {
    private int maxRequests;
    private long windowMs;
    private Deque<Long> timestamps = new LinkedList<>();

    public synchronized boolean allow() {
        long now = System.currentTimeMillis();

        // Remove stale timestamps
        while (!timestamps.isEmpty() &&
               now - timestamps.peekFirst() >= windowMs) {
            timestamps.removeFirst();
        }

        if (timestamps.size() < maxRequests) {
            timestamps.addLast(now);
            return true;
        }
        return false;
    }
}
```

Pros/Cons **Pros:** - Accurate sliding window - No burst issues

Cons: - Memory proportional to rate (O(maxRequests)) - More complex cleanup logic

---

Algorithm 3: Token Bucket (Base Implementation)

Concept: Bucket refills at constant rate; request consumes token.

```java
public class TokenBucketRateLimiter {
    private int capacity;              // Max tokens
    private double refillRatePerMs;    // Tokens added per millisecond
    private double tokens;             // Current tokens
    private long lastRefillTime;
    private Lock lock = new ReentrantLock();

    public TokenBucketRateLimiter(int capacity, int tokensPerSecond) {
        this.capacity = capacity;
        this.refillRatePerMs = tokensPerSecond / 1000.0;
        this.tokens = capacity;
        this.lastRefillTime = System.currentTimeMillis();
    }

    public boolean allow() {
        lock.lock();
        try {
            refillTokens();

            if (tokens >= 1) {
                tokens--;
                return true;
            }
            return false;
        } finally {
            lock.unlock();
        }
    }

    private void refillTokens() {
        long now = System.currentTimeMillis();
        long elapsed = now - lastRefillTime;

        if (elapsed <= 0) return;

        double newTokens = elapsed * refillRatePerMs;
        tokens = Math.min(capacity, tokens + newTokens);
        lastRefillTime = now;
    }
}
```

Token Bucket Algorithm Explained

Key Components

1. **Capacity**: Maximum tokens (burst size)
2. **Refill Rate**: Tokens added per time unit
3. **Current Tokens**: Available tokens (fractional for accuracy)

Refill Calculation

```
double newTokens = elapsed * refillRatePerMs;
tokens = Math.min(capacity, tokens + newTokens);
```

- **Linear Refill**: Tokens accumulate proportionally to time
- **Capped at Capacity**: Can't exceed bucket size
- **Fractional Tokens**: Use `double` for sub-token accuracy

Why Token Bucket? **Advantages:** - **Handles Bursts**: Accumulate tokens during idle time - **Smooth Rate**: Average rate limited by refill rate - **Memory Efficient**: O(1) space per limiter - **Industry Standard**: Used by AWS, Google Cloud, etc.

Enhancement 1: Blocking Acquire

Feature: Block until token available instead of returning false.

```
private Condition tokensAvailable = lock.newCondition();

public void acquire() throws InterruptedException {
    lock.lock();
    try {
        refillTokens();

        while (tokens < 1) {
            // Calculate wait time for next token
            long waitMs = (long) ((1 - tokens) / refillRatePerMs);
            tokensAvailable.await(waitMs, TimeUnit.MILLISECONDS);
            refillTokens();  // Refill after wait
        }

        tokens--;
    } finally {
        lock.unlock();
    }
}
```

Smart Waiting Instead of indefinite wait, calculate exact time for next token:

```
long waitMs = (long) ((1 - tokens) / refillRatePerMs);
```

This ensures prompt wakeup when token available.

Enhancement 2: Fairness (FIFO Order)

Problem: Non-blocking `allow()` isn't fair - lucky threads get tokens.

Solution: Queue waiting threads.

```
private Queue<Thread> waitingQueue = new LinkedList<>();

public void acquireFair() throws InterruptedException {
    lock.lock();
    try {
        Thread current = Thread.currentThread();
        waitingQueue.add(current);

        while (tokens < 1 || waitingQueue.peek() != current) {
            // Wait if: not enough tokens OR not my turn
            tokensAvailable.await();
            refillTokens();
        }

        waitingQueue.remove();
        tokens--;

        tokensAvailable.signal();  // Wake next in queue
    } finally {
        lock.unlock();
    }
}
```

Fairness Guarantee   Predicate:

```
while (tokens < 1 || waitingQueue.peek() != current)
```

- Wait if not enough tokens
- Also wait if there's someone ahead in queue
- Only proceed when: (1) tokens available AND (2) first in queue

---

Enhancement 3: VIP Priority

Feature: VIP users get priority over normal users.

```
private Queue<Thread> vipQueue = new LinkedList<>();
private Queue<Thread> normalQueue = new LinkedList<>();

public void acquire(boolean isVIP) throws InterruptedException {
    lock.lock();
    try {
        Queue<Thread> myQueue = isVIP ? vipQueue : normalQueue;
        Thread current = Thread.currentThread();
        myQueue.add(current);

        while (tokens < 1 || !isMyTurn(current)) {
            tokensAvailable.await();
            refillTokens();
        }

        myQueue.remove();
        tokens--;
        tokensAvailable.signal();
    } finally {
        lock.unlock();
    }
}

private boolean isMyTurn(Thread thread) {
    // VIP queue has priority
    if (!vipQueue.isEmpty()) {
        return vipQueue.peek() == thread;
    }
    return normalQueue.peek() == thread;
}
```

Priority Logic

1. **Check VIP queue first**: If non-empty, serve from there
2. **Then normal queue**: Only when VIP queue empty
3. **Within queue**: FIFO fairness

Enhancement 4: Per-User Rate Limiting

Feature: Independent rate limit per user/key.

```
private ConcurrentHashMap<String, TokenBucketRateLimiter> limiters =
    new ConcurrentHashMap<>();

public boolean allow(String userId) {
    TokenBucketRateLimiter limiter = limiters.computeIfAbsent(userId,
        k -> new TokenBucketRateLimiter(capacity, ratePerSecond));
    return limiter.allow();
}
```

Benefits

- **Isolation**: One user can't exhaust another's quota
- **Fairness**: Per-user limits
- **Scalability**: Concurrent access to different user limiters

---

Summary: Rate Limiter

Algorithm Comparison

Algorithm	Memory	Accuracy	Burst Handling	Complexity
Fixed Window	O(1)	Low	Poor	Simple
Sliding Window	O(rate)	High	Good	Medium
Token Bucket	O(1)	High	Excellent	Medium

Key Patterns

1. **Token Bucket**: Industry standard for rate limiting
2. **Fairness via Queue**: FIFO ordering for requests
3. **Priority Queues**: Multi-tier service levels
4. **Per-Key Limiters**: Isolation and fairness

Performance Considerations

- **Contention**: Single lock per limiter
- **Scalability**: Use per-user/per-key limiters
- **Precision**: Fractional tokens for accuracy

---

7. Resource Pool

Problem Statement

Manage a pool of reusable resources (e.g., database connections): - Acquire resource from pool (blocking if none available) - Release resource back to pool - Support maximum pool size - Handle timeouts - Detect and prevent resource leaks - Evict idle resources

Real-World Applications

- Database connection pooling
- Thread pools
- Object recycling (reduce GC pressure)
- Resource quota management

Base Solution: `ResourcePool1.java`

Simple non-blocking pool.

```java
public class ResourcePool1<T extends Resource> {
    private Deque<T> free = new ArrayDeque<>();
    private ReentrantLock lock = new ReentrantLock();

    public T acquire() {
        lock.lock();
        try {
            return free.pollFirst();  // null if empty
        } finally {
            lock.unlock();
        }
    }

    public void release(T r) {
        lock.lock();
        try {
            r.reset();  // Clean resource state
            free.addLast(r);
        } finally {
            lock.unlock();
        }
    }
}
```

Characteristics

- **Non-blocking**: Returns null if pool empty
- **Simple**: Minimal coordination
- **No Size Limit**: Pre-allocated resources

Limitations

- No blocking acquire (caller must handle null)
- No dynamic creation
- No leak detection

Enhancement 1: Max Size + Blocking Acquire

Feature: Block when pool exhausted, allow dynamic creation up to max size.

```java
public class ResourcePool2<T extends Resource> {
    private Deque<T> free = new ArrayDeque<>();
    private int maxSize;
    private int created = 0;

    private ReentrantLock lock = new ReentrantLock();
    private Condition available = lock.newCondition();

    public T acquire() throws InterruptedException {
        lock.lock();
        try {
            while (free.isEmpty() && created >= maxSize) {
                available.await();  // Wait for resource
            }

            if (!free.isEmpty()) {
                return free.removeFirst();
            }

            // Create new resource
            created++;
            return createNewResource();
        } finally {
            lock.unlock();
        }
    }

    public void release(T r) {
        lock.lock();
        try {
            r.reset();
            free.addLast(r);
            available.signal();  // Wake waiting acquirer
        } finally {
            lock.unlock();
        }
    }
}
```

Key Features

1. **Lazy Creation**: Create resources on demand
2. **Max Size Enforcement**: `created >= maxSize`
3. **Blocking**: Wait when pool exhausted
4. **Signal on Release**: Wake one waiting thread

Enhancement 2: Timeout on Acquire

Feature: Fail acquire after timeout instead of waiting indefinitely.

```java
public T acquire(long timeoutMs) throws Exception {
    lock.lock();
    try {
        long nanosRemaining = TimeUnit.MILLISECONDS.toNanos(timeoutMs);

        while (free.isEmpty() && created >= maxSize) {
            if (nanosRemaining <= 0) {
                return null;  // Timeout
            }
            nanosRemaining = available.awaitNanos(nanosRemaining);
        }

        if (!free.isEmpty()) {
            return free.removeFirst();
        }

        created++;
        return createNewResource();
    } finally {
        lock.unlock();
    }
}
```

Timeout Pattern (Revisited) **Standard Pattern:** 1. Convert timeout to nanoseconds 2. Loop: check condition 3. If timeout expired (`nanosRemaining <= 0`), fail 4. `awaitNanos(nanosRemaining)` returns remaining time 5. Update remaining time for next iteration

Why Nanoseconds? - Higher precision than milliseconds - Standard unit for `awaitNanos()`

Enhancement 3: Idle Eviction

Problem: Long-lived resources consume memory/connections unnecessarily.

Solution: Background thread evicts idle resources.

```java
public class ResourcePool3<T extends Resource> {
    private Map<T, Long> lastUsedTime = new ConcurrentHashMap<>();
    private long idleTimeoutMs = 60_000;  // 1 minute

    private Thread evictionThread = new Thread(() -> {
        while (true) {
```

```
        try {
            Thread.sleep(10_000);  // Check every 10 seconds
            evictIdleResources();
        } catch (InterruptedException e) {
            break;
        }
    }
});

public ResourcePool3() {
    evictionThread.setDaemon(true);  // Don't prevent JVM exit
    evictionThread.start();
}

private void evictIdleResources() {
    lock.lock();
    try {
        long now = System.currentTimeMillis();
        Iterator<T> it = free.iterator();

        while (it.hasNext()) {
            T resource = it.next();
            long idle = now - lastUsedTime.getOrDefault(resource, now);

            if (idle > idleTimeoutMs) {
                it.remove();
                resource.close();  // Cleanup
                created--;
            }
        }
    } finally {
        lock.unlock();
    }
}

public void release(T r) {
    lock.lock();
    try {
        r.reset();
        lastUsedTime.put(r, System.currentTimeMillis());
        free.addLast(r);
        available.signal();
    } finally {
        lock.unlock();
    }
}
```

```
}
```

Daemon Thread Characteristics: - `setDaemon(true)`: Thread won't prevent JVM shutdown - Runs periodically (e.g., every 10 seconds) - Cleans up resources under lock

Design Considerations **Eviction Frequency:** - Too frequent: Unnecessary overhead - Too infrequent: Resources stay idle too long - Trade-off: Every 10-30 seconds typical

Idle Threshold: - Depends on resource cost (connections expensive → lower threshold) - Should be longer than typical usage interval

---

Enhancement 4: Leak Detection

Problem: Resources not returned cause pool starvation.

Solution: Track resource usage and detect leaks.

```java
public class ResourcePool4<T extends Resource> {
    private Map<T, Long> acquiredTime = new ConcurrentHashMap<>();
    private Map<T, String> acquiredStack = new ConcurrentHashMap<>();
    private long leakDetectionThresholdMs = 300_000;  // 5 minutes

    public T acquire() throws InterruptedException {
        T resource = /* ... acquire logic ... */;

        // Track acquisition
        acquiredTime.put(resource, System.currentTimeMillis());
        acquiredStack.put(resource, getStackTrace());

        return resource;
    }

    public void release(T r) {
        lock.lock();
        try {
            // Remove tracking
            acquiredTime.remove(r);
            acquiredStack.remove(r);

            r.reset();
            free.addLast(r);
            available.signal();
        } finally {
            lock.unlock();
        }
    }

    private void detectLeaks() {
        long now = System.currentTimeMillis();

        for (Map.Entry<T, Long> entry : acquiredTime.entrySet()) {
            long heldTime = now - entry.getValue();

            if (heldTime > leakDetectionThresholdMs) {
                T resource = entry.getKey();
                String stack = acquiredStack.get(resource);

                System.err.println("LEAK DETECTED: Resource held for " +
                    heldTime + "ms");
                System.err.println("Acquired at:\n" + stack);
            }
        }
    }

    private String getStackTrace() {
        StackTraceElement[] stack = Thread.currentThread().getStackTrace();
        // Format and return relevant stack frames
        return /* formatted stack */;
    }
}
```

Leak Detection Strategy

1. **Track Acquisition**: Record time and stack trace
2. **Monitor Held Time**: Periodically check held duration
3. **Alert on Threshold**: Warn when resource held too long
4. **Diagnostic Info**: Provide stack trace for debugging

Metrics Additional useful metrics:

```
private long totalAcquired = 0;
private long totalReleased = 0;
private long totalTimeouts = 0;
private int peakUsage = 0;

public PoolStats getStats() {
    return new PoolStats(
        totalAcquired,
        totalReleased,
        totalTimeouts,
        peakUsage,
        free.size(),
        created
    );
}
```

Summary: Resource Pool

Progression

1. **Base**: Simple non-blocking pool
2. **Enhancement 1**: Blocking acquire with max size
3. **Enhancement 2**: Timeout support
4. **Enhancement 3**: Idle eviction
5. **Enhancement 4**: Leak detection and metrics

Key Patterns

1. **Blocking Acquire**: Wait for available resource
2. **Timeout**: Fail-fast instead of indefinite wait
3. **Background Cleanup**: Daemon thread for eviction
4. **Leak Detection**: Track usage and alert

Best Practices

- **Always release**: Use try-finally
- **Set timeouts**: Prevent indefinite waits
- **Monitor metrics**: Track pool health
- **Tune sizes**: Balance resource usage vs availability

---

8. Reader-Writer Key-Value Store

Problem Statement

Implement thread-safe key-value store with optimizations: - Concurrent reads (multiple readers) - Exclusive writes (one writer, no readers) - Reader-priority vs writer-priority modes - Snapshot reads (consistent view without blocking) - TTL (time-to-live) with automatic expiration

Real-World Applications

- Caching systems
- Configuration stores
- Session management
- Distributed key-value databases

Base Solution: `KVStore1.java`

Simple approach with single lock - serializes all operations.

```
public class KVStore1<K, V> {
    private Map<K, V> map = new HashMap<>();
    private ReentrantLock lock = new ReentrantLock();

    public V get(K key) {
        lock.lock();
        try {
            return map.get(key);
        } finally {
            lock.unlock();
        }
    }

    public void put(K key, V value) {
        lock.lock();
        try {
            map.put(key, value);
        } finally {
            lock.unlock();
        }
    }
}
```

Limitations

- No read concurrency (readers block each other)
- Overkill synchronization for read-heavy workloads

Enhancement 1: Reader-Writer Lock

Goal: Allow concurrent reads, exclusive writes.

Custom RWLock Implementation: `SimpleRWLock.java`

```
public class SimpleRWLock {
    private int readers = 0;
    private int writers = 0;
    private int writeRequests = 0;

    private ReentrantLock lock = new ReentrantLock();
    private Condition canRead = lock.newCondition();
```

```java
private Condition canWrite = lock.newCondition();

public void lockRead() throws InterruptedException {
    lock.lock();
    try {
        while (writers > 0 || writeRequests > 0) {
            canRead.await();
        }
        readers++;
    } finally {
        lock.unlock();
    }
}

public void unlockRead() {
    lock.lock();
    try {
        readers--;
        if (readers == 0) {
            canWrite.signal();  // Wake waiting writer
        }
    } finally {
        lock.unlock();
    }
}

public void lockWrite() throws InterruptedException {
    lock.lock();
    try {
        writeRequests++;

        while (readers > 0 || writers > 0) {
            canWrite.await();
        }

        writeRequests--;
        writers++;
    } finally {
        lock.unlock();
    }
}

public void unlockWrite() {
    lock.lock();
    try {
        writers--;
```

```
            canWrite.signal();    // Wake one waiting writer
            canRead.signalAll(); // Wake all waiting readers
        } finally {
            lock.unlock();
        }
    }
}
```

Algorithm Explanation **Read Lock:** - Wait while: writers active OR write requests pending - Increment reader count - Multiple readers can hold lock simultaneously

Write Lock: - Increment write requests (signals intent) - Wait while: any readers OR any writers active - Decrement requests, increment writers - Exclusive access

Unlock Patterns: - **Read Unlock**: Last reader signals waiting writer - **Write Unlock**: Signal one writer OR all readers

---

Reader-Writer Lock Analysis

Why Track `writeRequests`?

```
while (writers > 0 || writeRequests > 0) {
    canRead.await();
}
```

Purpose: **Writer Priority** - Readers wait if writers are pending - Prevents writer starvation - Trade-off: Readers might wait even when no active writer

Signal Strategy **On Write Unlock:**

```
canWrite.signal();    // Wake one writer
canRead.signalAll(); // Wake all readers
```

Why both? - **Writer-Priority**: Try writer first (`signal()` one) - **Fallback to Readers**: If no writers, readers can proceed - Order matters: `signal()` before `signalAll()`

Using RWLock in KVStore: `KVStore2.java`

```
public class KVStore2<K, V> {
    private Map<K, V> map = new HashMap<>();
    private SimpleRWLock rwLock = new SimpleRWLock();

    public V get(K key) throws InterruptedException {
        rwLock.lockRead();
        try {
            return map.get(key);
        } finally {
            rwLock.unlockRead();
        }
    }

    public void put(K key, V value) throws InterruptedException {
        rwLock.lockWrite();
        try {
            map.put(key, value);
        } finally {
            rwLock.unlockWrite();
        }
    }
}
```

Performance Impact **Read-Heavy Workload (e.g., 90% reads): - Before**: All operations serialized - **After**: Reads concurrent, only writes exclusive - **Improvement**: Significant throughput increase

Write-Heavy Workload: - Minimal benefit (writes still exclusive) - Slight overhead from lock complexity

---

Enhancement 2: Priority Modes

Problem: Default RWLock has writer-priority. Need reader-priority mode too.

Solution: `PriorityRWLock.java`

```
public class PriorityRWLock {
    enum Priority { READER, WRITER }

    private Priority priority;
    private int readers = 0;
    private int writers = 0;
    private int writeRequests = 0;

    private ReentrantLock lock = new ReentrantLock();
    private Condition canRead = lock.newCondition();
    private Condition canWrite = lock.newCondition();

    public PriorityRWLock(Priority priority) {
        this.priority = priority;
    }
    public void lockRead() throws InterruptedException {
        lock.lock();
        try {
            if (priority == Priority.READER) {
                // Reader-priority: only wait for active writers
                while (writers > 0) {
                    canRead.await();
                }
            } else {
                // Writer-priority: wait for writers OR write requests
                while (writers > 0 || writeRequests > 0) {
                    canRead.await();
                }
            }
            readers++;
        } finally {
            lock.unlock();
        }
    }

    // ... other methods similar ...
}
```

Priority Comparison

Priority	Read Condition	Writer Starvation	Reader Starvation
Reader	`writers > 0`	Possible	No
Writer	`writers > 0 || writeRequests > 0`	No	Possible

Choosing Priority: - **Reader-Priority**: Read-heavy workloads, can tolerate write delays - **Writer-Priority**: Need write progress guarantees, freshness important

---

Enhancement 3: Snapshot Reads

Problem: Long reads block writers. Need consistent view without holding lock.

Solution: Versioned Data with Copy-on-Write

```
public class KVStore4<K, V> {
    private volatile Map<K, V> data = new HashMap<>();
    private ReentrantLock writeLock = new ReentrantLock();
    public V get(K key) {
        // Lock-free read of current snapshot
        Map<K, V> snapshot = data;
        return snapshot.get(key);
    }

    public Map<K, V> getSnapshot() {
        // Return reference to current immutable snapshot
        return data;
    }

    public void put(K key, V value) {
        writeLock.lock();
        try {
            // Copy-on-write: create new map
            Map<K, V> newData = new HashMap<>(data);
            newData.put(key, value);
            data = newData;  // Atomic reference update
        } finally {
            writeLock.unlock();
        }
    }
}
```

Copy-on-Write Technique **Characteristics:** 1. **Immutable Snapshots**: Old maps never modified after creation 2. **Atomic Updates**: Reference assignment is atomic (`data = newData`) 3. **Lock-Free Reads**: No synchronization needed for reads 4. **Consistent View**: Each snapshot is internally consistent

Trade-offs: - **Pro**: Maximum read concurrency, no reader blocking - **Con**: Write overhead (full map copy), memory usage - **Use When**: Read:write ratio very high, map size moderate

Volatile Keyword

```
private volatile Map<K, V> data;
```

Purpose: - Ensures visibility of reference updates across threads - Reads always see latest published snapshot - No lock needed due to atomic reference semantics

---

Enhancement 4: TTL (Time-To-Live)

Feature: Auto-expire entries after timeout.

```java
public class KVStore5<K, V> {
    private Map<K, V> map = new HashMap<>();
    private Map<K, Long> expiryTimes = new HashMap<>();

    private ReentrantLock lock = new ReentrantLock();
    private long defaultTTL = 60_000;  // 1 minute

    // Background cleanup thread
    private Thread cleanupThread = new Thread(() -> {
        while (true) {
            try {
                Thread.sleep(5_000);  // Check every 5 seconds
                cleanup();
            } catch (InterruptedException e) {
                break;
            }
        }
    });

    public KVStore5() {
        cleanupThread.setDaemon(true);
        cleanupThread.start();
    }

    public void put(K key, V value) {
        put(key, value, defaultTTL);
    }

    public void put(K key, V value, long ttlMs) {
        lock.lock();
        try {
            map.put(key, value);
            long expiryTime = System.currentTimeMillis() + ttlMs;
            expiryTimes.put(key, expiryTime);
        } finally {
            lock.unlock();
        }
    }

    public V get(K key) {
        lock.lock();
        try {
            Long expiry = expiryTimes.get(key);

            if (expiry != null && System.currentTimeMillis() > expiry) {
                // Expired - lazy deletion
                map.remove(key);
                expiryTimes.remove(key);
                return null;
            }

            return map.get(key);
        } finally {
            lock.unlock();
        }
    }

    private void cleanup() {
        lock.lock();
        try {
            long now = System.currentTimeMillis();
            Iterator<Map.Entry<K, Long>> it = expiryTimes.entrySet().iterator();

            while (it.hasNext()) {
                Map.Entry<K, Long> entry = it.next();
                if (now > entry.getValue()) {
                    map.remove(entry.getKey());
                    it.remove();
                }
            }
        } finally {
            lock.unlock();
        }
    }
}
```

Expiration Strategies

1. **Lazy Deletion**: Check expiry on `get()`, delete if expired
2. **Active Deletion**: Background thread periodically cleans up
3. **Hybrid**: Both strategies (shown above)

Benefits: - Lazy: No overhead when key not accessed - Active: Frees memory even for unaccessed keys - Hybrid: Best of both

Daemon Thread Pattern (Revisited)

```
cleanupThread.setDaemon(true);
```

Why Daemon? - JVM can exit even if cleanup thread running - Cleanup is housekeeping, not critical work - Prevents hanging on shutdown

---

Summary: Reader-Writer KV Store

Progression

1. **Base**: Single lock (no read concurrency)
2. **Enhancement 1**: Reader-writer lock (concurrent reads)
3. **Enhancement 2**: Priority modes (reader vs writer priority)
4. **Enhancement 3**: Snapshot reads (copy-on-write)
5. **Enhancement 4**: TTL with background cleanup

Key Techniques

1. **Reader-Writer Lock**: Optimize for read-heavy workloads
2. **Priority Control**: Balance reader vs writer progress
3. **Copy-on-Write**: Lock-free reads with snapshots
4. **TTL**: Automatic expiration with daemon cleanup

Design Trade-offs

Approach	Read Perf	Write Perf	Memory	Complexity
Single Lock	Low	Medium	Low	Simple
RW Lock	High	Medium	Low	Medium
Snapshot	Very High	Low	High	Medium
TTL	Medium	Medium	Medium	High

---

9. Multi-Reader Multi-Writer File Simulator

Problem Statement

Simulate concurrent file access with multiple readers and writers: - File divided into chunks - Multiple threads can read/write different chunks concurrently - Prevent conflicting access to same chunk - Support write batching (atomically write multiple chunks) - Detect version conflicts - Support rollback for failed transactions

Real-World Applications

- Database storage engines
- File systems
- Distributed storage
- Version control systems

Base Solution: `FileSim1.java`

Single lock for entire file - serializes all operations.

```java
public class FileSim1 {
    private List<Chunk> chunks;
    private SimpleRWLock fileLock = new SimpleRWLock();

    public byte[] read(int chunkId) throws InterruptedException {
        fileLock.lockRead();
        try {
            return chunks.get(chunkId).getData();
        } finally {
            fileLock.unlockRead();
        }
    }

    public void write(int chunkId, byte[] data) throws InterruptedException {
        fileLock.lockWrite();
        try {
            chunks.get(chunkId).setData(data);
        } finally {
            fileLock.unlockWrite();
        }
    }
}
```

Limitation

- **Coarse-grained**: Lock entire file even for single chunk
- **No Parallelism**: Can't read chunk 0 while writing chunk 10

Enhancement 1: Chunk-Level Locking

Goal: Allow parallel access to different chunks.

```java
public class LockedChunk {
    private byte[] data;
    private SimpleRWLock lock = new SimpleRWLock();

    public byte[] read() throws InterruptedException {
        lock.lockRead();
        try {
            return data.clone();  // Return copy
        } finally {
            lock.unlockRead();
        }
    }

    public void write(byte[] newData) throws InterruptedException {
        lock.lockWrite();
        try {
            this.data = newData.clone();
        } finally {
            lock.unlockWrite();
        }
    }
}

public class FileSim2 {
    private List<LockedChunk> chunks;

    public byte[] read(int chunkId) throws InterruptedException {
        return chunks.get(chunkId).read();
    }

    public void write(int chunkId, byte[] data) throws InterruptedException {
        chunks.get(chunkId).write(data);
    }
}
```

Benefits

- **Fine-Grained**: Each chunk independently lockable
- **Parallelism**: Read chunk 0 while writing chunk 10
- **Scalability**: Contention only on same chunk

Enhancement 2: Write Batching with Deadlock Prevention

Problem: Atomically write multiple chunks. Naive approach causes deadlocks.

Deadlock Scenario:

```
Thread 1: lockWrite(chunk 0) → lockWrite(chunk 5)
Thread 2: lockWrite(chunk 5) → lockWrite(chunk 0)
    → DEADLOCK!
```

Solution: Lock Ordering

```
public void batchWrite(List<Integer> chunkIds, List<byte[]> dataList)
        throws InterruptedException {

    // Sort chunk IDs to ensure consistent lock ordering
    List<Integer> sortedIds = new ArrayList<>(chunkIds);
    Collections.sort(sortedIds);

    // Acquire locks in order
    for (int id : sortedIds) {
        chunks.get(id).lock.lockWrite();
    }

    try {
        // Perform writes
        for (int i = 0; i < chunkIds.size(); i++) {
            int id = chunkIds.get(i);
            chunks.get(id).setData(dataList.get(i));
        }
    } finally {
        // Release locks in reverse order (optional, but good practice)
        for (int i = sortedIds.size() - 1; i >= 0; i--) {
            chunks.get(sortedIds.get(i)).lock.unlockWrite();
        }
    }
}
```

Deadlock Prevention Principle **Rule**: Always acquire locks in **consistent global order**.

Why Sorting Works: - All threads acquire locks in ascending chunk ID order - Impossible for circular wait condition - No deadlocks possible

General Pattern:

```
1. Identify resources needed
2. Sort by global ordering (e.g., ID, address, hash)
3. Acquire in sorted order
4. Release (typically reverse order, but not required)
```

Enhancement 3: Version Conflict Detection (Optimistic Concurrency)

Concept: Track versions, detect conflicts on write.

```
public class VersionedChunk {
    private byte[] data;
    private long version = 0;
    private SimpleRWLock lock = new SimpleRWLock();

    public ReadResult read() throws InterruptedException {
        lock.lockRead();
        try {
            return new ReadResult(data.clone(), version);
        } finally {
            lock.unlockRead();
        }
    }

    public boolean write(byte[] newData, long expectedVersion)
            throws InterruptedException {
        lock.lockWrite();
        try {
            if (version != expectedVersion) {
                return false;  // Conflict!
            }

            this.data = newData.clone();
            this.version++;
            return true;
        } finally {
            lock.unlockWrite();
        }
    }
}

public class ReadResult {
    public byte[] data;
    public long version;

    public ReadResult(byte[] data, long version) {
        this.data = data;
        this.version = version;
    }
}
```

Optimistic Concurrency Pattern   Read-Modify-Write Cycle:

```
// 1. Read with version
ReadResult result = chunk.read();
byte[] data = result.data;
long version = result.version;

// 2. Modify (outside lock)
byte[] modified = modify(data);

// 3. Write with version check
```

```
boolean success = chunk.write(modified, version);

if (!success) {
    // Conflict - retry or fail
}
```

Optimistic vs Pessimistic

Approach	When to Lock	Conflicts	Best For
Pessimistic	Before read	Prevented	High contention
Optimistic	Before write	Detected	Low contention

Optimistic Advantages: - No locks held during computation - Better concurrency for read-heavy patterns - Readers don't block readers

Optimistic Disadvantages: - Wasted work on conflict (must retry) - Poor for high-contention scenarios

---

Enhancement 4: Transaction Rollback

Feature: Multi-chunk update with rollback on failure.

```
public class FileSim5 {
    public boolean transactionalWrite(List<Integer> chunkIds,
                                      List<byte[]> dataList)
            throws InterruptedException {

        List<Integer> sortedIds = new ArrayList<>(chunkIds);
        Collections.sort(sortedIds);

        // Acquire all locks
        for (int id : sortedIds) {
            chunks.get(id).lock.lockWrite();
        }

        try {
            // Save original data (before-images)
            Map<Integer, byte[]> beforeImages = new HashMap<>();
            for (int id : chunkIds) {
                beforeImages.put(id, chunks.get(id).getData().clone());
            }

            // Attempt writes
            try {
                for (int i = 0; i < chunkIds.size(); i++) {
                    int id = chunkIds.get(i);
                    byte[] data = dataList.get(i);

                    // Validate or process
                    if (!validate(data)) {
                        throw new Exception("Validation failed");
                    }

                    chunks.get(id).setData(data);
                }
                return true;  // Success

            } catch (Exception e) {
                // Rollback: restore before-images
                for (Map.Entry<Integer, byte[]> entry : beforeImages.entrySet()) {
                    chunks.get(entry.getKey()).setData(entry.getValue());
                }
                return false;  // Failed
            }

        } finally {
            // Release locks
            for (int i = sortedIds.size() - 1; i >= 0; i--) {
                chunks.get(sortedIds.get(i)).lock.unlockWrite();
            }
        }
    }
}
```

Transaction Pattern Phases: 1. **Acquire Locks**: Lock all resources (in order) 2. **Save Before-Images**: Backup original state 3. **Execute**: Perform modifications 4. **Commit or Rollback**: - Success: Keep changes - Failure: Restore before-images 5. **Release Locks**

ACID Properties

- **Atomicity**: All-or-nothing (rollback on failure)
- **Consistency**: Validation ensures valid states
- **Isolation**: Locks prevent concurrent access
- **Durability**: (Would persist before-images in real system)

Summary: File Simulator

Progression

1. **Base**: Whole-file locking (no parallelism)
2. **Enhancement 1**: Chunk-level locking (fine-grained)
3. **Enhancement 2**: Batch writes with lock ordering (deadlock prevention)
4. **Enhancement 3**: Version conflict detection (optimistic concurrency)
5. **Enhancement 4**: Transactional writes with rollback

Key Patterns

1. **Fine-Grained Locking**: Per-chunk locks for parallelism
2. **Lock Ordering**: Sorted acquisition prevents deadlocks
3. **Optimistic Concurrency**: Version numbers detect conflicts
4. **Transactions**: Before-images enable rollback

Critical Concepts

- **Deadlock Prevention**: Consistent lock ordering is essential
- **Granularity Trade-off**: Fine locks → more parallelism but more overhead
- **Concurrency Control**: Pessimistic (locks) vs optimistic (versions)

---

10. Job Queue

Problem Statement

Implement a thread-safe job queue with various capabilities: - Add and remove jobs - Worker threads fetch jobs - Support blocking when queue is empty - Priority-based job execution - Delayed job execution - Graceful shutdown with metrics

Real-World Applications

- Task execution systems
- Work queues
- Background job processors
- Async operation handlers

Base Solution: `JobQueue1.java`

Simple FIFO queue with fair lock.

```
public class JobQueue1 {
    private Deque<Job> queue = new ArrayDeque<>();
    private ReentrantLock lock = new ReentrantLock(true);  // Fair mode

    public void addJob(Job job) {
        lock.lock();
        try {
            queue.addLast(job);
        } finally {
            lock.unlock();
        }
    }

    public Job getNext() {
        lock.lock();
        try {
            return queue.pollFirst();  // null if empty
        } finally {
            lock.unlock();
        }
    }

    public boolean removeJob(Job job) {
        lock.lock();
        try {
            return queue.remove(job);  // O(n) removal
        } finally {
            lock.unlock();
        }
    }
}
```

Design Choices Fair Lock:

```
private ReentrantLock lock = new ReentrantLock(true);
```

- FIFO ordering for threads waiting on lock
- Prevents starvation
- Slight performance cost vs unfair lock

Non-Blocking `getNext()`: - Returns null if queue empty - Caller must handle null case (poll pattern)

Enhancement 1: Blocking Workers

Feature: Workers block when queue empty instead of polling.

Solution: `JobQueue2.java`

```
public class JobQueue2 {
    private Deque<Job> queue = new ArrayDeque<>();
    private ReentrantLock lock = new ReentrantLock();
    private Condition notEmpty = lock.newCondition();

    public void addJob(Job job) throws Exception {
        lock.lock();
        try {
            queue.addLast(job);
            notEmpty.signal();  // Wake one waiting worker
        } finally {
            lock.unlock();
        }
    }

    public Job takeJob() throws Exception {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                notEmpty.await();  // Block until job available
            }
            return queue.removeFirst();
        } finally {
            lock.unlock();
        }
    }
}
```

Benefits

1. **Efficiency**: No busy-waiting or polling
2. **CPU-Friendly**: Workers sleep when idle
3. **Instant Wakeup**: `signal()` wakes worker immediately when job arrives

Pattern Comparison

Approach	CPU Usage	Latency	Complexity
Polling (`getNext()`)	High	Low	Simple
Blocking (`takeJob()`)	Low	Low	Medium

Enhancement 2: Priority Queue

Feature: Execute higher-priority jobs first.

Solution: `JobQueue3.java`

```java
class PriorityJob implements Job, Comparable<PriorityJob> {
    int priority;
    Runnable task;

    public PriorityJob(int priority, Runnable task) {
        this.priority = priority;
        this.task = task;
    }

    public int compareTo(PriorityJob o) {
        return Integer.compare(o.priority, this.priority);  // Higher first
    }
}

public class JobQueue3 {
    private PriorityQueue<PriorityJob> pq = new PriorityQueue<>();
    private ReentrantLock lock = new ReentrantLock();
    private Condition notEmpty = lock.newCondition();

    public void addJob(PriorityJob job) {
        lock.lock();
        try {
            pq.add(job);  // O(log n)
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public PriorityJob takeJob() throws Exception {
        lock.lock();
        try {
            while (pq.isEmpty()) {
                notEmpty.await();
            }
            return pq.poll();  // O(log n) - returns highest priority
        } finally {
            lock.unlock();
        }
    }
}
```

PriorityQueue Analysis   Comparator:

```java
return Integer.compare(o.priority, this.priority);  // Reverse order
```

- Higher priority value = higher priority
- `PriorityQueue` is min-heap by default, so we reverse

Complexity: - Insert: O(log n) - Remove highest: O(log n) - Peek: O(1)

Thread Safety: - `PriorityQueue` itself is NOT thread-safe - We protect it with `ReentrantLock`

Enhancement 3: Delayed Jobs

Feature: Jobs scheduled for future execution.

Concept:

```
class DelayedJob implements Comparable<DelayedJob> {
    long scheduledTime;  // When to execute
    Runnable task;

    public int compareTo(DelayedJob o) {
        return Long.compare(this.scheduledTime, o.scheduledTime);
    }
}

public class JobQueue4 {
    private PriorityQueue<DelayedJob> pq = new PriorityQueue<>();
    private ReentrantLock lock = new ReentrantLock();
    private Condition notEmpty = lock.newCondition();

    public DelayedJob takeJob() throws Exception {
        lock.lock();
        try {
            while (true) {
                if (pq.isEmpty()) {
                    notEmpty.await();
                    continue;
                }

                DelayedJob job = pq.peek();
                long now = System.currentTimeMillis();

                if (job.scheduledTime <= now) {
                    // Ready to execute
                    return pq.poll();
                }

                // Wait until job is ready
                long waitMs = job.scheduledTime - now;
                notEmpty.await(waitMs, TimeUnit.MILLISECONDS);
            }
        } finally {
            lock.unlock();
        }
    }
}
```

Delayed Execution Logic Smart Waiting:

```
long waitMs = job.scheduledTime - now;
notEmpty.await(waitMs, TimeUnit.MILLISECONDS);
```

- Calculate exact wait time until next job ready
- Timeout wakes thread when job ready
- Also wakes if new job added (`signal()`)

Why Peek Then Poll?

```
DelayedJob job = pq.peek();  // Look without removing
// ... check time ...
return pq.poll();  // Remove and return
```

- Check if ready without removing
- Only remove when actually ready

Enhancement 4: Shutdown Support

Features: - Graceful shutdown (finish queued jobs) - Immediate shutdown (stop accepting new jobs) - Metrics (jobs added, taken, rejected)

```
public class JobQueue5 {
    private volatile boolean shutdown = false;
    private long jobsAdded = 0;
    private long jobsTaken = 0;
    private long jobsRejected = 0;

    public void addJob(Job job) throws Exception {
        lock.lock();
        try {
            if (shutdown) {
                jobsRejected++;
                throw new IllegalStateException("Queue is shutdown");
            }
            queue.addLast(job);
            jobsAdded++;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public Job takeJob() throws Exception {
        lock.lock();
        try {
            while (queue.isEmpty() && !shutdown) {
                notEmpty.await();
            }

            if (queue.isEmpty() && shutdown) {
                return null;  // Signal worker to exit
            }

            Job job = queue.removeFirst();
            jobsTaken++;
            return job;
        } finally {
            lock.unlock();
        }
    }

    public void shutdown() {
        lock.lock();
        try {
            shutdown = true;
            notEmpty.signalAll();  // Wake all waiting workers
        } finally {
            lock.unlock();
        }
    }

    public QueueMetrics getMetrics() {
        lock.lock();
        try {
            return new QueueMetrics(jobsAdded, jobsTaken, jobsRejected);
        } finally {
            lock.unlock();
        }
    }
}
```

Shutdown Protocol **On `shutdown()`:** 1. Set `shutdown = true` (volatile for visibility) 2. `signalAll()` wakes all workers 3. Workers check `shutdown` flag and exit gracefully

Worker Exit Condition:

```
if (queue.isEmpty() && shutdown) {
    return null;  // Worker should exit
}
```

- Process remaining jobs before exiting
- Only exit when queue drained AND shutdown signaled

Summary: Job Queue

Progression

1. **Base**: Simple FIFO with fair lock
2. **Enhancement 1**: Blocking workers with condition variables
3. **Enhancement 2**: Priority-based execution
4. **Enhancement 3**: Delayed/scheduled jobs
5. **Enhancement 4**: Shutdown and metrics

Key Patterns

1. **Producer-Consumer**: `addJob()` vs `takeJob()`
2. **Priority Scheduling**: `PriorityQueue` with custom comparator
3. **Delayed Execution**: Timed waits with `await(duration)`
4. **Graceful Shutdown**: Drain queue before exit

---

11. Task Scheduler

Problem Statement

Execute tasks with concurrency control: - Limit maximum parallel executions - Return values from tasks (Future pattern) - Support task delays - Task cancellation - Task caching (deduplicate identical tasks) - Failure propagation - Task dependencies

Real-World Applications

- Thread pool executors
- Async task runners
- Rate-limited API clients
- Workflow orchestration

Base Solution: `Scheduler1.java`

Limits concurrent executions using a counting semaphore pattern.

```java
public class Scheduler1 {
    private int maxParallel;
    private int running;

    private ReentrantLock lock = new ReentrantLock();
    private Condition isAvailable = lock.newCondition();

    public Scheduler1(int maxParallel) {
        this.maxParallel = maxParallel;
    }

    public void submit(Runnable task) {
        Thread t = new Thread(() -> {
            try {
                acquireSlot();
                task.run();
            } catch (InterruptedException e) {
                // Handle interruption
            } finally {
                releaseSlot();
            }
        });
        t.start();
    }

    private void acquireSlot() throws InterruptedException {
        lock.lock();
        try {
            while(running == maxParallel) {
                isAvailable.awaitUninterruptibly();
            }
            running++;
        } finally {
            lock.unlock();
        }
    }

    private void releaseSlot() {
        lock.lock();
        try {
            running--;
            isAvailable.signal();  // Wake one waiting task
        } finally {
            lock.unlock();
        }
    }
}
```

Semaphore Pattern Manual Counting Semaphore: - `running` tracks active tasks - `acquireSlot()` blocks when `running == maxParallel` - `releaseSlot()` decrements and signals waiter

Why Not Use `Semaphore` Class? - Educational purposes (show underlying mechanism) - More flexible for extensions - Could use `Semaphore(maxParallel)` for simplicity

Thread Creation

```
Thread t = new Thread(() -> {
    // acquire -> run -> release
});
t.start();
```

Characteristics: - Creates new thread per task - Thread dies after task completes - Not a thread pool (no reuse)

Enhancement 1: Return Values with Future Pattern

Feature: Get task results asynchronously.

Solution: `Scheduler2.java`

```
class SimpleFuture<T> {
    private T value;
    private boolean done = false;

    private ReentrantLock lock = new ReentrantLock();
    private Condition finished = lock.newCondition();

    public void set(T v) {
        lock.lock();
        try {
            value = v;
            done = true;
            finished.signalAll();  // Wake all waiters
        } finally {
            lock.unlock();
        }
    }
    public T get() throws InterruptedException {
        lock.lock();
        try {
            while(!done) {
                finished.await();  // Block until value ready
            }
            return value;
        } finally {
            lock.unlock();
        }
    }
}
```

Using SimpleFuture:

```
public <T> SimpleFuture<T> submit(Callable<T> task) {
    SimpleFuture<T> future = new SimpleFuture<>();

    Thread t = new Thread(() -> {
        acquireSlot();
        try {
            T result = task.call();
            future.set(result);  // Publish result
        } catch (Exception e) {
            // Could enhance to store exception
        } finally {
            releaseSlot();
        }
    });

    t.start();
    return future;
}
```

Future Pattern Explained Producer Side (Task Thread):

```
T result = task.call();
future.set(result);
```

- Compute result
- Publish to future
- Signal waiting threads

Consumer Side (Caller):

```
SimpleFuture<Integer> future = scheduler.submit(() -> 42);
// Do other work...
```

```java
Integer result = future.get();  // Block until ready
```

Comparison with `java.util.concurrent.Future`

Feature	SimpleFuture	Java Future
Blocking get		
Timeout		
Cancellation		
Exception handling		
`isDone()`		

SimpleFuture is educational; production code should use `CompletableFuture`.

Enhancement 2: Task Delay

Feature: Delay task execution by specified time.

```java
public <T> SimpleFuture<T> submitWithDelay(Callable<T> task, long delayMs) {
    SimpleFuture<T> future = new SimpleFuture<>();

    Thread t = new Thread(() -> {
        try {
            Thread.sleep(delayMs);  // Initial delay
        } catch (InterruptedException e) {
            return;
        }

        acquireSlot();
        try {
            future.set(task.call());
        } finally {
            releaseSlot();
        }
    });

    t.start();
    return future;
}
```

Note: Delay happens BEFORE acquiring slot (not counted toward concurrency limit while waiting).

Enhancement 3: Shutdown Support

Feature: Stop accepting new tasks, optionally wait for completion.

```java
private volatile boolean shutdown = false;

public void submit(Runnable task) {
    if (shutdown) {
        throw new RejectedExecutionException("Scheduler is shutdown");
    }
    // ... create and start thread ...
}

public void shutdown() {
    shutdown = true;
}

public void awaitTermination(long timeoutMs) throws InterruptedException {
    long deadline = System.currentTimeMillis() + timeoutMs;

    lock.lock();
    try {
        while (running > 0) {
            long remaining = deadline - System.currentTimeMillis();
            if (remaining <= 0) {
                return;  // Timeout
            }
            isAvailable.await(remaining, TimeUnit.MILLISECONDS);
        }
    } finally {
        lock.unlock();
    }
}
```

Enhancement 4: Task Cancellation

Feature: Cancel tasks before or during execution.

```java
class CancellableFuture<T> extends SimpleFuture<T> {
    private volatile boolean cancelled = false;

    public boolean cancel() {
        lock.lock();
        try {
            if (done) {
                return false;  // Already completed
            }
            cancelled = true;
            finished.signalAll();
            return true;
        } finally {
            lock.unlock();
        }
    }

    public boolean isCancelled() {
        return cancelled;
    }

    public T get() throws InterruptedException {
        lock.lock();
        try {
            while (!done && !cancelled) {
                finished.await();
            }
            if (cancelled) {
                throw new CancellationException();
            }
            return value;
        } finally {
            lock.unlock();
        }
    }
}
```

Task Must Check Cancellation:

```
Callable<Integer> task = () -> {
    for (int i = 0; i < 1000; i++) {
        if (Thread.currentThread().isInterrupted()) {
            return null;  // Or throw exception
        }
        // Do work
    }
    return result;
};
```

Cancellation is Cooperative - task must check flag.

Enhancement 5: Task Caching

Feature: Deduplicate identical concurrent tasks.

```java
private ConcurrentHashMap<String, SimpleFuture<T>> cache = new ConcurrentHashMap<>();

public SimpleFuture<T> submitCached(String key, Callable<T> task) {
    SimpleFuture<T> future = cache.computeIfAbsent(key, k -> {
        SimpleFuture<T> newFuture = new SimpleFuture<>();

        Thread t = new Thread(() -> {
            acquireSlot();
            try {
                T result = task.call();
                newFuture.set(result);
            } finally {
                releaseSlot();
                cache.remove(key);  // Clean up after completion
            }
        });

        t.start();
        return newFuture;
    });

    return future;
}
```

Use Case Without Caching:

```
Thread 1: submit(fetchUser(123)) → starts task A
Thread 2: submit(fetchUser(123)) → starts task B (duplicate!)
```

With Caching:

```
Thread 1: submitCached("user-123", ...) → starts task A, returns futureA
Thread 2: submitCached("user-123", ...) → returns futureA (reuses!)
```

Benefits: - Avoid redundant work - Save resources - Consistent results

---

Enhancement 6: Task Dependencies

Feature: Task B waits for Task A to complete.

```java
public <T> SimpleFuture<T> submitAfter(SimpleFuture<?> dependency, Callable<T> task) {
    SimpleFuture<T> future = new SimpleFuture<>();

    Thread t = new Thread(() -> {
        try {
            dependency.get();  // Wait for dependency
        } catch (Exception e) {
            // Propagate failure
            return;
        }

        acquireSlot();
        try {
            future.set(task.call());
        } finally {
            releaseSlot();
        }
    });

    t.start();
    return future;
}
```

Usage:

```
SimpleFuture<Data> fetchFuture = scheduler.submit(() -> fetchData());
SimpleFuture<Result> processFuture = scheduler.submitAfter(fetchFuture,
    () -> processData());
```

Chain:

```
fetchData() → [dependency wait] → processData()
```

Summary: Task Scheduler

Progression

1. **Base**: Concurrency-limited execution
2. **Enhancement 1**: Return values with Future pattern
3. **Enhancement 2**: Delayed execution
4. **Enhancement 3**: Shutdown support
5. **Enhancement 4**: Cancellation support
6. **Enhancement 5**: Task caching (deduplication)
7. **Enhancement 6**: Task dependencies

Key Patterns

1. **Counting Semaphore**: Manual `acquireSlot()`/`releaseSlot()`
2. **Future Pattern**: Async result retrieval
3. **Cooperative Cancellation**: Tasks check flag
4. **Task Caching**: `ConcurrentHashMap.computeIfAbsent()`
5. **Dependency Chain**: Sequential task execution

---

12. Priority Job Scheduler

Problem Statement

Advanced job scheduler with priority support: - Submit jobs with priority levels - Workers fetch highest-priority job - Dynamic priority changes (even for queued jobs) - Cooperative preemption for running jobs - Multi-worker execution - Graceful shutdown with metrics

Real-World Applications

- Operating system schedulers
- Request prioritization
- Quality of Service (QoS) systems
- Emergency task handling

Base Solution: `PriorityScheduler1.java`

Priority queue with blocking take.

```java
public class PriorityScheduler1 {
    protected PriorityQueue<PJob> pq = new PriorityQueue<>();
    protected ReentrantLock lock = new ReentrantLock();
    protected Condition notEmpty = lock.newCondition();

    public void submit(PJob job) {
        lock.lock();
        try {
            pq.add(job);
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public PJob take() throws Exception {
        lock.lock();
        try {
            while (pq.isEmpty()) {
                notEmpty.await();
            }
            return pq.poll();  // Returns highest priority
        } finally {
            lock.unlock();
        }
    }
}
```

Job Structure:

```java
class PJob implements Comparable<PJob> {
    String id;
    int priority;
    Runnable task;

    public int compareTo(PJob o) {
        return Integer.compare(o.priority, this.priority);  // Higher first
    }
}
```

Enhancement 1: Multi-Worker Execution

Solution: `PriorityScheduler2.java`

Create worker threads that continuously process jobs.

```java
public class PriorityScheduler2 extends PriorityScheduler1 {
    private List<Thread> workers = new ArrayList<>();

    public PriorityScheduler2(int numWorkers) {
        for (int i = 0; i < numWorkers; i++) {
            Thread t = new Thread(this::workerAction);
            t.start();
            workers.add(t);
        }
    }

    private void workerAction() {
        while (true) {
            try {
                PJob job = take();  // Block until job available
                job.task.run();
            } catch (Exception e) {
                // Handle error
            }
        }
    }
}
```

Worker Pool Benefits **Advantages:** - **Thread Reuse**: No thread creation overhead per task - **Parallelism**: Multiple jobs execute concurrently -

Resource Control: Fixed thread count (no thread explosion)

Considerations: - Worker threads run forever (need shutdown mechanism) - Exception handling important (one bad job shouldn't kill worker)

Enhancement 2: Dynamic Priority Change

Problem: Change priority of queued job.

Challenge: `PriorityQueue` doesn't support priority updates directly.

Solution: Remove and Re-insert

```
public class PriorityScheduler3 extends PriorityScheduler2 {
    private Map<String, PJob> jobMap = new HashMap<>();  // Track jobs by ID

    public void submit(PJob job) {
        lock.lock();
        try {
            pq.add(job);
            jobMap.put(job.id, job);
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public boolean changePriority(String jobId, int newPriority) {
        lock.lock();
        try {
            PJob job = jobMap.get(jobId);
            if (job == null) {
                return false;  // Job not found or already executing
            }

            // Remove from queue
            pq.remove(job);  // O(n) operation

            // Update priority
            job.priority = newPriority;

            // Re-insert with new priority
            pq.add(job);  // O(log n)

            return true;
        } finally {
            lock.unlock();
        }
    }
}
```

Complexity Analysis **`changePriority()` Operations:** 1. `jobMap.get(jobId)`: O(1) 2. `pq.remove(job)`: O(n) - linear scan 3. `pq.add(job)`: O(log n) - heap insert

Total: O(n) where n = queue size

Optimization Opportunity: Could use `TreeMap<Priority, Queue<Job>>` for faster priority changes, but more complex.

Enhancement 3: Cooperative Preemption

Problem: Change priority of RUNNING job (not just queued).

Challenge: Can't forcefully stop running thread safely in Java.

Solution: Cooperative Preemption

Preemptible Job Interface:

```
interface PreemptibleJob {
    void run();
    void yield();  // Signal to stop
    boolean shouldYield();  // Check if should stop
}

class PreemptibleJobImpl implements PreemptibleJob {
    private volatile boolean yieldFlag = false;
    private Runnable task;

    public void run() {
        // Task must periodically check shouldYield()
        // Pseudo-code for task:
        // while (hasWork() && !shouldYield()) {
        //     doWork();
        // }
    }

    public void yield() {
        yieldFlag = true;
    }

    public boolean shouldYield() {
        return yieldFlag;
    }
}
```

Preemption Logic:

```
public class PriorityScheduler4 extends PriorityScheduler3 {
    private Map<String, PreemptibleJob> runningJobs = new HashMap<>();

    public boolean changePriority(String jobId, int newPriority) {
        lock.lock();
        try {
            // Check if queued
            PJob job = jobMap.get(jobId);
            if (job != null) {
                // Queued - just update priority
                return super.changePriority(jobId, newPriority);
            }

            // Check if running
            PreemptibleJob running = runningJobs.get(jobId);
            if (running != null) {
                // Running - signal to yield
                running.yield();

                // Re-queue with new priority
                PJob newJob = new PJob(jobId, newPriority, () -> running.run());
                submit(newJob);
                return true;
            }

            return false;  // Job not found
        } finally {
            lock.unlock();
        }
    }
}
```

Cooperative Preemption Pattern Key Points: 1. **Voluntary**: Task must check `shouldYield()` periodically 2. **Graceful**: Task can clean up before stopping 3. **Safe**: No forced thread termination

Task Implementation:

```
PreemptibleJob job = new PreemptibleJob() {
    public void run() {
        for (int i = 0; i < 1000; i++) {
            if (shouldYield()) {
                // Save state and return
                return;
            }
            // Do work
        }
    }
};
```

Limitations: - Requires task cooperation - Can't preempt tasks that don't check flag - Delay between yield signal and actual stop

---

Enhancement 4: Metrics and Monitoring

Features: - Jobs submitted/completed counts - Average wait time - Average execution time - Current queue size

```
public class PriorityScheduler5 extends PriorityScheduler4 {
    private long jobsSubmitted = 0;
    private long jobsCompleted = 0;
    private long totalWaitTimeMs = 0;
    private long totalExecutionTimeMs = 0;

    public void submit(PJob job) {
        lock.lock();
        try {
            job.submitTime = System.currentTimeMillis();
            pq.add(job);
            jobMap.put(job.id, job);
            jobsSubmitted++;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    private void workerAction() {
        while (!shutdown) {
            try {
                PJob job = take();

                long startTime = System.currentTimeMillis();
                long waitTime = startTime - job.submitTime;

                job.task.run();
                long endTime = System.currentTimeMillis();
                long executionTime = endTime - startTime;

                lock.lock();
                try {
                    jobsCompleted++;
                    totalWaitTimeMs += waitTime;
                    totalExecutionTimeMs += executionTime;
                } finally {
                    lock.unlock();
                }

            } catch (Exception e) {
                // Handle error
            }
        }
    }

    public SchedulerMetrics getMetrics() {
        lock.lock();
        try {
            double avgWait = jobsCompleted > 0 ?
                (double) totalWaitTimeMs / jobsCompleted : 0;
            double avgExecution = jobsCompleted > 0 ?
                (double) totalExecutionTimeMs / jobsCompleted : 0;

            return new SchedulerMetrics(
                jobsSubmitted,
                jobsCompleted,
                avgWait,
                avgExecution,
                pq.size()
            );
        } finally {
            lock.unlock();
        }
    }
}
```

Metric Uses

- **Performance Tuning**: Identify bottlenecks
- **Capacity Planning**: Determine worker count needs
- **SLA Monitoring**: Track wait/execution times
- **Alerting**: Detect queue buildup

Summary: Priority Job Scheduler

Progression

1. **Base**: Priority queue with blocking take
2. **Enhancement 1**: Multi-worker thread pool
3. **Enhancement 2**: Dynamic priority for queued jobs
4. **Enhancement 3**: Cooperative preemption for running jobs
5. **Enhancement 4**: Comprehensive metrics

Key Concepts

1. **Priority Scheduling**: `PriorityQueue` with custom comparator
2. **Worker Pool**: Fixed threads continuously process jobs
3. **Dynamic Priority**: Remove and re-insert pattern
4. **Cooperative Preemption**: Voluntary yield with flag checking
5. **Metrics**: Track wait/execution times

Design Trade-offs

- **Priority Update Cost**: O(n) removal from heap
- **Preemption**: Cooperative (safe) vs Forced (dangerous)
- **Worker Count**: More workers = more parallelism but more overhead

---

13. LRU Cache

Problem Statement

Implement a thread-safe Least Recently Used (LRU) cache: - Fixed capacity - O(1) get and put operations - Evict least recently used item when full - Thread-safe for concurrent access

Real-World Applications

- Web browser caches
- CDN edge caching
- Database query caches
- API response caching

Solution: `LRUCache.java`

Uses **HashMap + Doubly-Linked List** data structure.

Data Structure Design **Components:** 1. **HashMap**: O(1) key → node lookup 2. **Doubly-Linked List**: O(1) add/remove operations - Head: Most recently used - Tail: Least recently used

Node Structure:

```
private class Node {
    K key;
    V value;
    Node prev, next;

    public Node(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

Cache State:

```
private int capacity;
private Map<K, Node> map = new HashMap<>();
private Node head;  // Most recent
private Node tail;  // Least recent

private ReentrantLock lock = new ReentrantLock();
```

Get Operation

```
public V get(K key) {
    lock.lock();
    try {
        Node node = map.get(key);
        if (node == null) {
            return null;  // Cache miss
        }
        moveToHead(node);  // Update recency
        return node.getValue();
    } finally {
        lock.unlock();
    }
}
```

Logic: 1. Look up in HashMap: O(1) 2. If found, move to head (mark as recently used) 3. Return value

Move to Head:

```
private void moveToHead(Node node) {
    if (node != head) {
        removeNode(node);  // Remove from current position
        addToHead(node);   // Add to head position
    }
}
```

Put Operation

```
public void put(K key, V value) {
    lock.lock();
    try {
        Node node = map.get(key);

        if (node == null) {
            // New key
            node = new Node(key, value);
            map.put(key, node);
        } else {
            // Update existing
            node.setValue(value);
        }

        addToHead(node);

        if (map.size() > capacity) {
            // Evict LRU
            Node removed = trimTail();
            map.remove(removed.getKey());
        }
    } finally {
        lock.unlock();
    }
}
```

Logic: 1. Check if key exists 2. Create new node or update existing 3. Move to head (most recent) 4. If over capacity, evict tail (least recent)

List Operations

Add to Head:

```
private void addToHead(Node node) {
    node.setNext(head);
    node.setPrev(null);

    if (head != null) {
        head.setPrev(node);
    }
    head = node;

    if (tail == null) {
        tail = node;  // First node
    }
}
```

Remove Node:

```
private void removeNode(Node node) {
    if (node == null) return;

    if (node.getPrev() != null) {
        node.getPrev().setNext(node.getNext());
    } else {
        head = node.getNext();  // Removing head
    }

    if (node.getNext() != null) {
        node.getNext().setPrev(node.getPrev());
    } else {
        tail = node.getPrev();  // Removing tail
    }
}
```

Trim Tail (Evict LRU):

```
private Node trimTail() {
    if (tail == null) return null;

    Node evicted = tail;
    removeNode(evicted);
    return evicted;
}
```

Complexity Analysis

Operation	Time	Explanation
get(key)	O(1)	HashMap lookup + list operations
put(key, value)	O(1)	HashMap insert + list operations

Operation	Time	Explanation
Eviction	O(1)	Remove tail node

All operations are O(1) - this is the key property of LRU cache!

Thread Safety

Single Lock Strategy:

```
private ReentrantLock lock = new ReentrantLock();
```

- All operations protected by single lock
- Simple and correct
- Serializes all cache operations

Performance Consideration: - Contention bottleneck for high-concurrency - Could optimize with lock striping or concurrent data structures

Alternative: `ReadWriteLock`

```
private ReadWriteLock rwLock = new ReentrantReadWriteLock();

public V get(K key) {
    rwLock.readLock().lock();
    try {
        // ... but moveToHead() modifies list!
        // Would need write lock anyway
    }
}
```

Problem: Even `get()` modifies the list (moves node to head), so read lock insufficient.

Usage Example

```
LRUCache<Integer, String> cache = new LRUCache<>(3);

cache.put(1, "one");
cache.put(2, "two");
cache.put(3, "three");
// Cache: [3→2→1]

cache.get(1);  // Access 1
// Cache: [1→3→2]
```

```java
cache.put(4, "four");  // Evicts 2 (LRU)
// Cache: [4→1→3]

String val = cache.get(2);  // Returns null (evicted)
```

Summary: LRU Cache

Key Techniques

1. **HashMap + Doubly-Linked List**: O(1) operations
2. **Recency Tracking**: Move accessed items to head
3. **Eviction**: Remove tail when over capacity

Design Patterns

- **Cache Aside**: Application manages cache explicitly
- **Write-Through**: All writes go through cache (not implemented here)

Optimization Opportunities

1. **Lock Striping**: Multiple segments like `ConcurrentHashMap`
2. **Read-Heavy Optimization**: Separate read path with optimistic locking
3. **Batching**: Group multiple operations

Real-World Considerations

- **TTL**: Add expiration times
- **Stats**: Track hit rate, eviction rate
- **Persistence**: Serialize cache to disk
- **Distributed**: Use Redis or Memcached

---

14. Matrix Multiplication

Problem Statement

Parallelize matrix multiplication: - Multiply two matrices A and B - Distribute work across multiple threads - Maximize CPU utilization - No synchronization overhead (if possible)

Real-World Applications

- Scientific computing
- Machine learning (neural networks)
- Graphics rendering
- Image processing

Base Solution: Row-Wise Parallelization

File: `ParallelMatrixMultiply.java`

```java
public class ParallelMatrixMultiply {
    public static int[][] multiply(int[][] A, int[][] B) throws InterruptedException {
        int m = A.length;         // Rows in A
        int n = A[0].length;      // Cols in A = Rows in B
        int p = B[0].length;      // Cols in B

        if (B.length != n) {
            throw new IllegalArgumentException("Invalid matrix dimensions");
        }

        int[][] C = new int[m][p];

        // Create one thread per row
        Thread[] threads = new Thread[m];
        for (int i = 0; i < m; i++) {
            threads[i] = new RowWorker(A, B, C, i);
            threads[i].start();
        }

        // Wait for all threads to complete
        for (int i = 0; i < m; i++) {
            threads[i].join();
        }

        return C;
    }
}
```

Row Worker:

```java
class RowWorker extends Thread {
    private int[][] A, B, C;
    private int row;

    public RowWorker(int[][] A, int[][] B, int[][] C, int row) {
        this.A = A;
        this.B = B;
        this.C = C;
        this.row = row;
    }
    public void run() {
        int n = A[0].length;
        int p = B[0].length;

        // Compute row 'row' of result matrix C
        for (int j = 0; j < p; j++) {
            int sum = 0;
            for (int k = 0; k < n; k++) {
                sum += A[row][k] * B[k][j];
            }
            C[row][j] = sum;
        }
    }
}
```

Why No Synchronization Needed?

Critical Question: Multiple threads writing to C[][] - why no locks?

Answer: No Write Conflicts

```
Thread 0: writes C[0][*]
Thread 1: writes C[1][*]
Thread 2: writes C[2][*]
...
```

Each thread writes to DIFFERENT rows - No two threads access same memory location - No race conditions - No need for synchronization

Memory Safety: - Each row is independent chunk of memory - Different cache lines (assuming typical layout) - No false sharing (rows large enough)

Alternative: Block-Based Parallelization

File: `BlockParallelMatrixMultiply.java`

Divide matrix into blocks instead of rows.

```
public class BlockParallelMatrixMultiply {
    public static int[][] multiply(int[][] A, int[][] B, int blockSize) {
        int m = A.length;
        int p = B[0].length;
        int[][] C = new int[m][p];

        List<Thread> threads = new ArrayList<>();
        // Create threads for each block
        for (int i = 0; i < m; i += blockSize) {
            for (int j = 0; j < p; j += blockSize) {
                final int startRow = i;
                final int startCol = j;

                Thread t = new Thread(() -> {
                    computeBlock(A, B, C, startRow, startCol, blockSize);
                });

                t.start();
                threads.add(t);
            }
        }

        // Wait for completion
        for (Thread t : threads) {
            t.join();
        }

        return C;
    }

    private static void computeBlock(int[][] A, int[][] B, int[][] C,
                                     int startRow, int startCol, int blockSize) {
        int m = A.length;
        int n = A[0].length;
        int p = B[0].length;

        int endRow = Math.min(startRow + blockSize, m);
        int endCol = Math.min(startCol + blockSize, p);

        for (int i = startRow; i < endRow; i++) {
            for (int j = startCol; j < endCol; j++) {
                int sum = 0;
                for (int k = 0; k < n; k++) {
                    sum += A[i][k] * B[k][j];
                }
                C[i][j] = sum;
            }
        }
    }
}
```

Row-Wise vs Block-Based

Approach	Granularity	Thread Count	Cache Locality	Use Case
Row-Wise	Coarse	= # rows	Lower	Simple, small matrices
Block-Based	Fine	= (m/block) × (p/block)	Higher	Large matrices

Block-Based Advantages: - Better cache locality (blocks fit in L1/L2 cache) - Flexible parallelism (tune block size) - Handles large matrices better

Row-Wise Advantages: - Simpler implementation - Less overhead - Good for moderate-sized matrices

---

Performance Considerations

Factors Affecting Performance:

1. **Thread Overhead**
   - Creating/destroying threads has cost
   - For small matrices, overhead > benefit
   - Use thread pool for production
2. **Cache Effects**
   - Matrix B accessed column-wise (poor cache locality)
   - Transpose B first for better cache hits
   - Block-based helps with cache
3. **Load Balancing**
   - Equal-sized rows/blocks → balanced load
   - Irregular workloads need work-stealing
4. **Optimal Thread Count**
   - Too few: Underutilized CPU
   - Too many: Context switching overhead
   - Rule of thumb: # cores to 2× # cores

Optimization: Tiled Matrix Multiplication

Advanced Technique (not in provided code):

```
// Compute C = A × B using cache-friendly tiling
for (int ii = 0; ii < m; ii += TILE_SIZE) {
    for (int jj = 0; jj < p; jj += TILE_SIZE) {
        for (int kk = 0; kk < n; kk += TILE_SIZE) {
            // Multiply tile A[ii:ii+TILE][kk:kk+TILE]
            //   with tile B[kk:kk+TILE][jj:jj+TILE]
            multiplyTile(A, B, C, ii, jj, kk, TILE_SIZE);
        }
    }
}
```

Benefits: - Keeps tiles in L1/L2 cache - Reduces memory bandwidth - 10-100× speedup for large matrices

---

Summary: Matrix Multiplication

Key Concepts

1. **Data Parallelism**: Independent computations can run in parallel
2. **No Synchronization**: Disjoint write sets eliminate need for locks
3. **Thread Join**: Wait for all threads before returning result

Parallelization Strategies

1. **Row-Wise**: One thread per row (simple)
2. **Block-Based**: Divide into blocks (better cache)
3. **Tiled**: Cache-optimized (production)

Performance Factors

- **Thread overhead vs matrix size**
- **Cache locality** (block/tile size)
- **Load balancing** (equal work per thread)

---

15. Thread-Safe Key-Value Store

Problem Statement

Simple thread-safe map wrapper using synchronized methods.

Solution: `ThreadSafeKeyValueStore.java`

```java
public class ThreadSafeKeyValueStore {
    private final Map<String, String> store = new HashMap<>();

    synchronized void setValue(String key, String value) {
        this.store.put(key, value);
    }
    private synchronized String getValue(String key) {
        return this.store.get(key);
    }

    private synchronized void deleteValue(String key) {
        this.store.remove(key);
    }

    private synchronized int size() {
        return this.store.size();
    }
}
```

Analysis

Synchronization Strategy Intrinsic Lock (Monitor):

```java
synchronized void setValue(String key, String value) {
    // Equivalent to:
    // synchronized(this) {
    //     this.store.put(key, value);
    // }
}
```

- Every object has one intrinsic lock (monitor)
- `synchronized` method acquires `this` monitor
- Released automatically on method exit

Correctness Thread Safety: - All access to `store` protected by `synchronized` - Serializes all operations - No race conditions possible

Why `final` on `store`?

```java
private final Map<String, String> store = new HashMap<>();
```

- Reference is immutable (can't reassign to different Map)
- Helps with thread safety (one less variable to worry about)

Performance Characteristics

Pros: - Simple and correct - No explicit Lock management - Clear semantics

Cons: - **Coarse-Grained**: Single lock for entire map - **No Read Concurrency**: Readers block each other - **Limited Scalability**: Contention bottleneck

---

Comparison with Alternatives

1. ConcurrentHashMap   Better Approach:

```
public class ThreadSafeKeyValueStore {
    private final ConcurrentHashMap<String, String> store = new ConcurrentHashMap<>();

    void setValue(String key, String value) {
        store.put(key, value);  // No synchronization needed
    }

    String getValue(String key) {
        return store.get(key);
    }
}
```

Benefits: - Lock-free reads - Fine-grained locking (segment-level) - Much higher concurrency

2. ReentrantReadWriteLock   For HashMap:

```
public class ThreadSafeKeyValueStore {
    private final Map<String, String> store = new HashMap<>();
    private final ReadWriteLock rwLock = new ReentrantReadWriteLock();

    void setValue(String key, String value) {
        rwLock.writeLock().lock();
        try {
            store.put(key, value);
        } finally {
            rwLock.writeLock().unlock();
        }
    }

    String getValue(String key) {
        rwLock.readLock().lock();
        try {
            return store.get(key);
        } finally {
            rwLock.readLock().unlock();
        }
    }
}
```

Benefits: - Concurrent reads - Still uses `HashMap` (less memory than `ConcurrentHashMap`)

Summary: Thread-Safe KV Store

Key Concepts

1. **Synchronized Methods**: Intrinsic lock (monitor) pattern
2. **Coarse-Grained Locking**: Entire object protected
3. **Simplicity vs Performance**: Trade-off

When to Use

- **Educational**: Learn synchronization basics
- **Low Contention**: Few threads, infrequent access
- **Simple Requirements**: Don't need high performance

Production Recommendation Use `ConcurrentHashMap` for production - it's optimized for concurrent access.

---

16. Singleton Pattern

Problem Statement

Ensure only one instance of a class exists across the application with thread-safe lazy initialization.

Solution: `Singleton.java`

```
public class Singleton {
    private static volatile Singleton instance;

    Singleton() {
        // Private constructor
    }

    public static Singleton getInstance() {
        synchronized (Singleton.class) {
            if (instance == null) {
                instance = new Singleton();
            }
        }
        return instance;
    }
}
```

Analysis

Current Implementation Issues **Problem: Always Synchronized**

```java
public static Singleton getInstance() {
    synchronized (Singleton.class) {  // Every call acquires lock!
        if (instance == null) {
            instance = new Singleton();
        }
    }
    return instance;
}
```

- Lock acquired on EVERY call to `getInstance()`
- Even after instance created
- Unnecessary contention and overhead

Double-Checked Locking (DCL)

Better Implementation:

```java
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {  // First check (no lock)
            synchronized (Singleton.class) {
                if (instance == null) {  // Second check (with lock)
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

DCL Explanation **First Check (Unsynchronized):**

```java
if (instance == null) {  // Fast path
```

- Most calls skip synchronization (instance already created)
- No lock overhead

Synchronized Block:

```java
synchronized (Singleton.class) {
    if (instance == null) {  // Double-check
        instance = new Singleton();
    }
}
```

- Only entered if instance appears null
- Second check needed: another thread might have created instance while we waited for lock

Why `volatile`?

```java
private static volatile Singleton instance;
```

Without `volatile`: - Thread A: Creates instance, writes to memory - Thread B: Might see partially constructed instance (!) - Due to memory reordering - Constructor not fully complete

With `volatile`: - Happens-before relationship guaranteed - Writes before `volatile` write visible to readers - No partially constructed objects

---

Alternative: Initialization-on-Demand Holder

Best Practice (Bill Pugh Singleton):

```java
public class Singleton {
    private Singleton() {}

    private static class Holder {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return Holder.INSTANCE;
    }
}
```

Why This is Better Lazy Initialization: - `Holder` class not loaded until `getInstance()` called - No explicit synchronization needed - ClassLoader handles thread safety

Thread Safety: - JVM guarantees class initialization is thread-safe - No race conditions possible - Simpler and more efficient than DCL

No Synchronization Overhead: - No locks ever acquired - Just field access

---

Alternative: Enum Singleton

Simplest Approach:

```java
public enum Singleton {
    INSTANCE;

    public void doSomething() {
        // ...
    }
}

// Usage:
Singleton.INSTANCE.doSomething();
```

Enum Benefits

1. **Thread-Safe**: JVM guarantees single instance
2. **Serialization-Safe**: Enum serialization handled correctly
3. **Reflection-Safe**: Can't create second instance via reflection
4. **Concise**: Minimal code

Drawback: Enum can't extend classes (can implement interfaces)

Singleton Pattern Comparison

Approach	Thread-Safe	Lazy Init	Complexity	Reflection-Safe
Eager Init			Simple	
Synchronized Method			Simple	
Double-Checked Locking			Medium	
Holder Pattern			Simple	
Enum			Simple	

Summary: Singleton Pattern

Key Concepts

1. **Double-Checked Locking**: Optimize synchronized access
2. **Volatile**: Prevent partial construction visibility
3. **Holder Pattern**: Use class initialization for thread safety
4. **Enum**: Simplest and safest singleton

Best Practices

- **Prefer**: Holder pattern or Enum
- **Avoid**: Simple synchronized (current implementation)
- **Use DCL**: Only if need lazy init AND can't use Holder
- **Always**: Make constructor private

Common Pitfalls

- Forgetting `volatile` in DCL
- Not making constructor private
- Not handling serialization/cloning properly

Conclusion

This documentation has covered 16 comprehensive concurrency problems in Java, demonstrating:

Core Concurrency Primitives

- **ReentrantLock**: Explicit locking with fairness and condition support
- **Condition Variables**: Thread coordination and signaling
- **Semaphore Pattern**: Resource counting and throttling
- **ReadWriteLock**: Optimize for read-heavy workloads
- **Volatile**: Visibility guarantees without locks
- **Synchronized**: Intrinsic locks (monitors)

Design Patterns

1. **Producer-Consumer**: Bounded queues with backpressure
2. **Future Pattern**: Asynchronous result retrieval
3. **Object Pool**: Resource reuse and lifecycle management
4. **Reader-Writer**: Concurrent reads, exclusive writes
5. **Priority Scheduling**: QoS and fairness
6. **Copy-on-Write**: Lock-free reads with immutable snapshots
7. **Singleton**: Thread-safe single instance

Key Principles

- **Predicate Loops**: Always `while(condition)` not `if(condition)`
- **Lock Ordering**: Prevent deadlocks with consistent acquisition order
- **Signal Strategy**: `signal()` vs `signalAll()` based on predicate
- **Fairness vs Performance**: Trade-offs in lock and queue implementations
- **Fine vs Coarse Locking**: Granularity affects parallelism and complexity
- **Cooperative Cancellation**: Safe thread interruption patterns

Progressive Enhancement Approach

Each problem demonstrates evolution from: 1. Basic correctness 2. Performance optimization 3. Fairness guarantees 4. Advanced features (metrics, priorities, caching) 5. Production-ready concerns (shutdown, leak detection, monitoring)

Performance Optimization Techniques

- **Lock Striping**: Reduce contention with multiple locks
- **Lock-Free Algorithms**: ConcurrentHashMap, atomic operations
- **Batching**: Amortize lock acquisition cost
- **Background Cleanup**: Daemon threads for housekeeping
- **Timeout Patterns**: Prevent indefinite blocking

Testing and Debugging

- **Metrics**: Track throughput, latency, queue sizes
- **Leak Detection**: Monitor resource usage
- **Visibility**: Use monitoring tools (JConsole, VisualVM)

Recommended Study Path

1. **Start with**: Ordered Printing, Bathroom Problem (basic coordination)
2. **Progress to**: Bounded Blocking Queue, Producer-Consumer (classic patterns)
3. **Explore**: Resource Pool, Job Queue (resource management)
4. **Advanced**: File Simulator, Priority Scheduler (complex coordination)
5. **Optimize**: Hit Counter, Rate Limiter (performance patterns)
6. **Master**: LRU Cache, Reader-Writer Store (data structure design)

Additional Resources

Java Concurrency Utilities

- `java.util.concurrent` package
- `ExecutorService` for thread pools
- `CompletableFuture` for async operations
- `ConcurrentHashMap` for concurrent collections

Further Reading

- "Java Concurrency in Practice" by Brian Goetz
- "The Art of Multiprocessor Programming" by Herlihy and Shavit
- Java Memory Model (JSR-133)
- Happens-Before relationships

Tools

- **JMH**: Microbenchmarking framework
- **JProfiler/YourKit**: Profiling and thread analysis
- **Thread Sanitizer**: Detect data races
- **Stress Testing**: `jcstress` framework

End of Documentation

This comprehensive guide covers fundamental to advanced concurrency patterns in Java. Each problem builds upon previous concepts, demonstrating real-world applications and best practices for writing correct, efficient, and maintainable concurrent code.