Thread-Safe Cache Study Guide

Overview

The **Thread-Safe LRU Cache** demonstrates advanced concurrency patterns for high-performance caching systems with **reader-writer synchronization** and **lazy LRU management**.

Key Concepts Covered

1. Reader-Writer Synchronization

```cpp
std::shared_mutex mutex_;

// Multiple concurrent readers
std::shared_lock<std::shared_mutex> read_lock(mutex_);

// Exclusive writer access
std::unique_lock<std::shared_mutex> write_lock(mutex_);
```

2. LRU (Least Recently Used) Algorithm

```cpp
// Doubly-linked list + hash map = O(1) operations
std::unordered_map<Key, Iterator> cache_map_;
std::list<std::pair<Key, Value>> lru_list_;
```

3. Lazy vs Eager Updates

- **Eager**: Update LRU order on every read (high contention)
- **Lazy**: Update LRU order only on writes (better performance)

Data Structure Design

Hybrid Architecture

```cpp
struct CacheNode {
    Key key;
    Value value;
    // Implicit position in linked list
};

// Fast lookup: O(1)
std::unordered_map<Key, list_iterator> map_;

// LRU ordering: O(1) insert/remove
std::list<CacheNode> lru_list_;
```

Operations Complexity

Operation	Time	Space	Contention
get()	O(1)	O(1)	Shared lock
put()	O(1)	O(1)	Exclusive lock
eviction	O(1)	O(1)	During put()

Real-World Applications

System Caches

- **CPU caches**: L1/L2/L3 cache hierarchies
- **Database buffer pools**: Page caching
- **Web caches**: HTTP response caching
- **CDNs**: Content delivery networks

Application Caches

- **Redis/Memcached**: Distributed caching
- **In-memory caches**: Application-level caching
- **File system caches**: OS page cache
- **Browser caches**: Resource caching

Interview Questions & Answers

Q: "Why use shared_mutex instead of regular mutex?"

A: Performance optimization for read-heavy workloads: - **Concurrent reads**: Multiple threads can read simultaneously - **Exclusive writes**: Only one writer, no readers during writes - **Typical cache ratio**: 90% reads, 10% writes - **Speedup**: 5-10x improvement in read-heavy scenarios

Q: "How do you handle cache eviction?"

A: LRU eviction strategy:

```
if (cache_map_.size() >= capacity_) {
    // Remove least recently used (back of list)
    auto lru_key = lru_list_.back().first;
    cache_map_.erase(lru_key);
    lru_list_.pop_back();
}
```

Q: "What about thread safety during eviction?"

A: Write lock protects entire operation: - Check capacity while holding exclusive lock - Eviction and insertion are atomic - No partial states visible to readers

Q: "Why not update LRU on every read?"

A: Lazy LRU trade-off: - **Pros**: Better performance, less contention - **Cons**: Slightly less accurate LRU ordering - **Real-world**: Most caches use approximations anyway

Design Patterns Demonstrated

1. Adaptive Locking Strategy

```
// Readers use shared locks
std::shared_lock<std::shared_mutex> lock(mutex_);

// Writers use exclusive locks
std::unique_lock<std::shared_mutex> lock(mutex_);
```

2. RAII Lock Management

- Automatic lock release
- Exception safety
- Clear lock scope boundaries

3. Template Specialization

```
template<typename Key, typename Value>
class ThreadSafeCache {
    // Generic for any key-value types
    // Requires Key to be hashable
};
```

Performance Optimization Techniques

1. Lock Granularity

- **Coarse-grained**: One lock for entire cache
- **Fine-grained**: Bucket-level locking (more complex)
- **Trade-off**: Simplicity vs scalability

2. Memory Layout

```
// Cache-friendly access patterns
std::list<std::pair<Key, Value>> lru_list_;  // Sequential access
std::unordered_map<Key, Iterator> map_;      // Random access
```

3. Move Semantics

```
void put(Key key, Value value) {
    // Move to avoid copies
```

```cpp
    lru_list_.emplace_front(std::move(key), std::move(value));
}
```

Cache Replacement Policies

LRU Alternatives

- **LFU (Least Frequently Used)**: Count-based eviction
- **FIFO**: Simple queue-based eviction

- **Random**: Minimal overhead
- **ARC (Adaptive Replacement Cache)**: Balances recency and frequency

When to Use Each

- **LRU**: General-purpose, good temporal locality
- **LFU**: Frequency matters more than recency
- **FIFO**: Simple, predictable behavior
- **Random**: Very low overhead, surprisingly effective

Test Scenarios Covered

1. **BasicOperations**: put/get functionality
2. **EvictionBehavior**: LRU ordering verification
3. **ConcurrentAccess**: Multiple readers/writers
4. **CapacityLimits**: Proper eviction handling
5. **MemoryLeaks**: Resource cleanup verification
6. **StressTest**: High contention scenarios

Common Interview Challenges

1. "Implement LRU cache in 15 minutes"

Key points to hit: - Hash map for O(1) lookup - Doubly-linked list for O(1) eviction - Move recently used to front - Evict from back when full

2. "How would you scale this to multiple machines?"

Distributed cache considerations: - **Consistent hashing**: Distribute keys across nodes - **Replication**: Handle node failures - **Cache coherence**: Keep replicas synchronized - **Hot spotting**: Load balancing strategies

3. "What if the cache is too big for memory?"

Hybrid storage strategies: - **Tiered caching**: Memory + disk - **Compression**: Reduce memory footprint - **Eviction to disk**: Spillover storage - **Memory mapping**: OS-managed paging

Production Considerations

Monitoring & Metrics

- **Hit rate**: Percentage of cache hits
- **Eviction rate**: How often items are removed
- **Memory usage**: Cache size monitoring
- **Contention**: Lock wait times

Configuration Tuning

- **Cache size**: Balance memory vs hit rate
- **Eviction policy**: Match access patterns
- **Concurrency level**: Readers vs writers ratio
- **Warmup strategy**: Pre-populate important data

Advanced Extensions

Lock-Free Cache

- **Hazard pointers**: Safe memory reclamation
- **Atomic operations**: CAS-based updates
- **Memory ordering**: Sequential consistency
- **Complexity**: Much harder to implement correctly

Partitioned Cache

- **Segment locks**: Reduce contention
- **Hash-based partitioning**: Distribute load
- **NUMA awareness**: Thread-local caches

This cache implementation demonstrates sophisticated concurrent data structure design - a key skill for high-performance systems and distributed architectures!