Read-Write Lock Study Guide

Overview

The Read-Write Lock exercise demonstrates **reader-writer synchronization** using `std::shared_mutex` - a crucial pattern for optimizing concurrent read-heavy workloads.

Key Concepts Covered

1. Shared vs Exclusive Locking

```cpp
std::shared_mutex rw_lock;

// Multiple readers can acquire simultaneously
std::shared_lock<std::shared_mutex> read_lock(rw_lock);

// Only one writer, blocks all readers
std::unique_lock<std::shared_mutex> write_lock(rw_lock);
```

2. Lock Compatibility Matrix

Lock Type	Reader Waiting	Writer Waiting
No locks	Proceed	Proceed
Reader active	Proceed	Block
Writer active	Block	Block

3. Real-World Applications

- **Caching systems**: Frequent reads, rare cache updates
- **Configuration management**: Read config often, update rarely
- **Database indexes**: Read queries dominate, writes are periodic
- **Shared data structures**: Multiple readers, occasional modifications

Performance Benefits

Read Scalability

```cpp
// Traditional mutex: Serial access
std::mutex m;
std::lock_guard<std::mutex> lock(m);  // Only ONE thread at a time

// Shared mutex: Parallel reads
std::shared_mutex sm;
std::shared_lock<std::shared_mutex> lock(sm);  // MULTIPLE readers
```

When to Use Read-Write Locks

- **High read-to-write ratio** (10:1 or higher)
- **Long read operations** (make parallelism worthwhile)
- **Contention exists** (benefit from reduced blocking)

Warning: RW locks have overhead. For short reads or low contention, regular mutex may be faster!

Implementation Patterns

1. Basic Reader-Writer

```
class SharedCounter {
    mutable std::shared_mutex mutex_;
    int counter_ = 0;

public:
    int get() const {
        std::shared_lock lock(mutex_);  // Shared read
        return counter_;
    }

    void increment() {
        std::unique_lock lock(mutex_);  // Exclusive write
        ++counter_;
    }
};
```

2. Upgraded Lock Pattern

```
// Start with read lock, upgrade to write if needed
std::shared_lock read_lock(mutex_);
if (needs_modification(data)) {
    read_lock.unlock();                    // Release read
    std::unique_lock write_lock(mutex_);   // Acquire write
    modify(data);
    // Write lock auto-releases
}
```

3. RAII Lock Guards

```
template<typename T>
class ReadGuard {
    std::shared_lock<std::shared_mutex> lock_;
    const T& data_;
public:
    ReadGuard(std::shared_mutex& m, const T& data)
```

```cpp
        : lock_(m), data_(data) {}

    const T& get() const { return data_; }
};
```

Common Pitfalls

1. Writer Starvation

```cpp
// Problem: Continuous readers can starve writers
while (true) {
    std::shared_lock lock(mutex_);
    read_data();  // If this never ends, writers wait forever
}
```

Solution: Use writer-preferring implementations or bounded reader limits.

2. Lock Ordering

```cpp
// Deadlock risk with multiple RW locks
void transfer(Account& from, Account& to) {
    std::unique_lock lock1(from.mutex_);  // Writer lock
    std::unique_lock lock2(to.mutex_);    // Writer lock - potential deadlock
}
```

Solution: Consistent lock ordering (e.g., by address).

3. Exception Safety

```cpp
void risky_operation() {
    std::unique_lock lock(mutex_);
    may_throw_exception();  // Lock auto-releases on exception
    // RAII ensures cleanup
}
```

Interview Questions

Q: When would you choose read-write lock over regular mutex?

A: When reads significantly outnumber writes (high read-to-write ratio) and read operations are substantial enough to benefit from parallelization. Typical scenarios include caches, configuration stores, and lookup tables.

Q: What's the overhead of read-write locks?

A: RW locks have higher overhead than regular mutex due to: - More complex internal state tracking - Reader counting mechanisms
- Fairness algorithms to prevent writer starvation

For short critical sections or low contention, regular mutex is often faster.

Q: How do you prevent writer starvation?

A: Strategies include: - **Writer preference**: New readers block if writers are waiting - **Bounded readers**: Limit concurrent reader count - **Time-based fairness**: Guarantee writer access within time bounds - **Queue-based**: FIFO ordering of lock requests

Q: Can you upgrade from read to write lock?

A: Standard `std::shared_mutex` doesn't support atomic upgrade. You must: 1. Release read lock 2. Acquire write lock
3. Re-validate state (data may have changed)

Some libraries provide upgradable read-write locks for this pattern.

Performance Characteristics

Best Case

- Many concurrent readers
- Rare writers
- Long read operations
- **Result**: Near-linear scaling with reader threads

Worst Case

- Frequent writes
- Short read operations
- High lock contention
- **Result**: Slower than regular mutex due to overhead

Measurement

```cpp
// Benchmark read vs write performance
auto start = std::chrono::high_resolution_clock::now();
{
    std::shared_lock lock(rw_mutex);  // or std::unique_lock
    do_work();
}
auto duration = std::chrono::high_resolution_clock::now() - start;
```

Key Takeaways

1. **Read-write locks optimize read-heavy workloads** by allowing parallel readers
2. **Use when read-to-write ratio is high** (typically 10:1 or better)

3. **Beware of writer starvation** - continuous readers can block writers indefinitely
4. **RAII is essential** for exception safety and automatic cleanup
5. **Measure performance** - RW locks have overhead that may not always pay off

This pattern is fundamental for building scalable systems with shared data structures!