C++ Memory Model and Atomic Operations - Exercise Sheet

Easy Exercises (Understanding Basics)

Exercise 1: Basic Atomic Operations

Create a program with two threads: one incrementing an atomic counter 1000 times, another decrementing it 1000 times. Verify the final value is 0.

Exercise 2: Atomic Flag Spinlock

Implement a spinlock using `std::atomic_flag`. Use it to protect a shared integer that 4 threads increment 1000 times each. The final value should be 4000.

Exercise 3: Compare and Exchange

Implement an atomic increment function using `compare_exchange_weak()` in a loop. Test with 10 threads each incrementing 1000 times.

Exercise 4: Lock-Free Status Check

Create an atomic boolean flag that one thread sets to true after doing some work, while another thread polls until it becomes true. Measure how long the polling takes.

Exercise 5: Atomic Pointer Operations

Create an atomic pointer to an integer. One thread allocates and stores new integers, another thread reads and processes them. Use `exchange()` to swap pointers safely.

Medium Exercises (Memory Ordering)

Exercise 6: Relaxed Ordering Demonstration

Create two atomic booleans x and y. Thread 1 stores true to x then y (relaxed ordering). Thread 2 waits for y to be true, then checks x. Run multiple times to observe cases where x might still be false.

Exercise 7: Acquire-Release Synchronization

Implement a simple producer-consumer where the producer fills an array and sets a flag with release semantics. The consumer waits for the flag with acquire semantics then reads the array.

Exercise 8: Sequential Consistency vs Acquire-Release

Implement the same algorithm using both sequential consistency and acquire-release ordering. Compare performance and verify both produce correct results.

Exercise 9: Release Sequence

Create a scenario with one producer thread and multiple consumer threads using an atomic counter. The producer sets the count, consumers use `fetch_sub()` to claim work items. Verify the release sequence ensures all consumers see the producer's data.

Exercise 10: Memory Fences

Rewrite the relaxed ordering example from Exercise 6 using `std::atomic_thread_fence()` to ensure proper synchronization while keeping the atomic operations relaxed.

Hard Exercises (Advanced Patterns)

Exercise 11: Lock-Free Stack

Implement a lock-free stack using atomic pointers and compare-exchange operations. Handle the ABA problem appropriately.

Exercise 12: Hazard Pointers

Implement a simple hazard pointer system to safely reclaim memory in the lock-free stack from Exercise 11.

Exercise 13: SPSC Queue

Create a Single Producer Single Consumer queue using atomic operations with relaxed ordering where possible and acquire-release only where necessary.

Exercise 14: RCU-like Pattern

Implement a Read-Copy-Update pattern where readers can access data without locks while a writer creates new versions. Use atomic pointers and appropriate memory ordering.

Exercise 15: Lock-Free Reference Counting

Implement a thread-safe reference-counted smart pointer using atomic operations. Handle the race condition between increment/decrement and destruction.

Exercise 16: Work-Stealing Queue

Create a work-stealing deque where the owner can push/pop from one end while thieves can steal from the other end, all using atomic operations.

Exercise 17: Seqlock Implementation

Implement a sequence lock (seqlock) that allows multiple readers to proceed concurrently with writers, using atomic operations and memory barriers.

Exercise 18: Memory Pool with Atomic Free List

Design a lock-free memory pool using an atomic stack for the free list. Handle allocation and deallocation without traditional locks.

Expert Exercises (Complex Synchronization)

Exercise 19: MPMC Queue

Implement a Multi-Producer Multi-Consumer queue using atomic operations. Consider cache-line optimization and minimizing contention.

Exercise 20: Epoch-Based Reclamation

Implement an epoch-based memory reclamation system that can be used with lock-free data structures to safely defer memory deallocation.

Exercise 21: Flat Combining

Implement a flat combining pattern where one thread becomes the combiner and executes operations on behalf of other threads, reducing contention.

Exercise 22: Read-Write Lock

Create a reader-writer lock using only atomic operations (no mutexes). Ensure writers have priority and no reader starvation.

Exercise 23: Load Balancing Work Queue

Design a work queue system where multiple worker threads can steal work from each other using atomic operations, with proper load balancing.

Exercise 24: Atomic Hash Table

Implement a hash table that supports concurrent insertions, deletions, and lookups using atomic operations for synchronization.

Exercise 25: Transactional Memory Simulation

Create a simple software transactional memory system using atomic operations to detect conflicts and retry transactions.

Performance Analysis Exercises

Exercise 26: Memory Ordering Benchmarks

Benchmark the performance difference between sequential consistency, acquire-release, and relaxed ordering in various scenarios.

Exercise 27: Cache Line Contention

Design experiments to demonstrate false sharing and show how to avoid it using proper alignment and padding.

Exercise 28: Scalability Testing

Test how your lock-free data structures scale with increasing numbers of threads compared to mutex-based alternatives.

Exercise 29: Memory Ordering on Different Architectures

If possible, test your atomic code on different CPU architectures (x86, ARM) and observe any performance differences.

Exercise 30: Atomic vs Mutex Performance

Create detailed benchmarks comparing atomic operations vs mutex-protected operations under different contention levels.

Tips for Success:

- Start with simple examples and gradually increase complexity
- Use tools like thread sanitizer to detect data races
- Test with different numbers of threads and workloads
- Pay attention to memory ordering - start with sequential consistency, then optimize
- Profile your solutions to understand performance characteristics
- Read the generated assembly to understand what the compiler produces