Concurrency, Multithreading & Low-Latency Programming

Complete Guide: From Theory to Production HFT Systems

Table of Contents

Chapter 1: Threads vs Processes

1.1 Foundational Overview

Layman's Intuition

Think of a process as a restaurant, and threads as chefs in that kitchen. Each restaurant (process) has its own equipment and supplies (memory space), but chefs (threads) within the same restaurant share those resources. Multiple chefs can work simultaneously on different dishes, coordinating to avoid collisions. This shared-resource parallelism is threading.

Technical Foundation

Process:

- Independent execution unit
- Own memory space (code, data, heap, stack)
- Heavyweight: ~2-4MB overhead per process
- Communication: IPC (pipes, sockets, shared memory)
- Isolation: Crash doesn't affect others

Thread:

- Lightweight execution unit within process
- Shares process memory (code, data, heap)
- Own stack and registers
- Lightweight: ~8KB stack overhead
- Communication: Shared memory (fast but needs synchronization)
- Risk: Crash can corrupt entire process

Comparison:

Aspect	Process	Thread
Creation Time	1-10ms	10-100µs
Context Switch	1-10µs	100-1000ns
Memory Overhead	~2-4MB	~8KB
Communication	Slow (IPC)	Fast (shared memory)
Isolation	Strong	Weak

Industry Relevance

HFT Systems:

- **Market Data:** Multiple threads parse different exchanges simultaneously
- **Order Execution:** Separate threads for order routing, risk checks, confirmations

- **Performance:** Thread switch 10-100x faster than process switch

Backend Systems:

- **Web Servers:** Thread pool handles concurrent requests
- **Databases:** Per-connection threads for query execution
- **Microservices:** Thread-per-request or async I/O models

1.2 Thread Creation and Management

POSIX Threads (pthreads)

Basic Thread Creation:

```
#include &lt;pthread.h&gt;
#include &lt;iostream&gt;

// Thread function signature: void* func(void*)
void* worker_thread(void* arg) {
    int thread_id = *(int*)arg;

    std::cout &lt;&lt; "Thread " &lt;&lt; thread_id &lt;&lt; " starting\n";

    // Simulate work
    for (int i = 0; i &lt; 5; i++) {
        std::cout &lt;&lt; "Thread " &lt;&lt; thread_id &lt;&lt; " iteration " &lt;&lt; i
        sleep(1);
    }

    std::cout &lt;&lt; "Thread " &lt;&lt; thread_id &lt;&lt; " exiting\n";

    // Return value passed to pthread_join
    int* result = new int(thread_id * 100);
    return result;
}

int main() {
    const int NUM_THREADS = 4;
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    // Create threads
    for (int i = 0; i &lt; NUM_THREADS; i++) {
        thread_ids[i] = i;

        int rc = pthread_create(&amp;threads[i], nullptr,
                                worker_thread, &amp;thread_ids[i]);

        if (rc != 0) {
            std::cerr &lt;&lt; "Error creating thread " &lt;&lt; i &lt;&lt; "\n";
            return 1;
        }
    }
```

```
    // Wait for threads to complete
    for (int i = 0; i &lt; NUM_THREADS; i++) {
        void* result;
        pthread_join(threads[i], &amp;result);

        int value = *(int*)result;
        std::cout &lt;&lt; "Thread " &lt;&lt; i &lt;&lt; " returned " &lt;&lt; value &lt;
        delete (int*)result;
    }

    return 0;
}
```

C++11 std::thread (Modern Approach)

Basic Usage:

```
#include &lt;thread&gt;
#include &lt;iostream&gt;
#include &lt;vector&gt;

void worker_function(int id) {
    std::cout &lt;&lt; "Thread " &lt;&lt; id &lt;&lt; " running\n";
}

int main() {
    std::vector&lt;std::thread&gt; threads;

    // Create threads
    for (int i = 0; i &lt; 4; i++) {
        threads.emplace_back(worker_function, i);
    }

    // Join all threads
    for (auto&amp; t : threads) {
        if (t.joinable()) {
            t.join();
        }
    }

    return 0;
}
```

Lambda Functions:

```
std::thread t([](int x, int y) {
    std::cout &lt;&lt; "Sum: " &lt;&lt; (x + y) &lt;&lt; "\n";
}, 10, 20);

t.join();
```

Member Functions:

```
class Worker {
public:
    void process(int iterations) {
        for (int i = 0; i &amp;lt; iterations; i++) {
            std::cout &amp;lt;&amp;lt; "Processing " &amp;lt;&amp;lt; i &amp;lt;&amp;lt; "\n";
        }
    }
};

Worker w;
std::thread t(&amp;amp;Worker::process, &amp;amp;w, 100);
t.join();
```

Thread Attributes and Configuration

Setting Stack Size:

```
#include &amp;lt;pthread.h&amp;gt;

void create_thread_custom_stack() {
    pthread_attr_t attr;
    pthread_attr_init(&amp;amp;attr);

    // Set stack size to 2MB
    size_t stack_size = 2 * 1024 * 1024;
    pthread_attr_setstacksize(&amp;amp;attr, stack_size);

    pthread_t thread;
    pthread_create(&amp;amp;thread, &amp;amp;attr, worker_thread, nullptr);

    pthread_attr_destroy(&amp;amp;attr);
    pthread_join(thread, nullptr);
}
```

Detached Threads:

```
// Detached thread: runs independently, no join needed
void create_detached_thread() {
    pthread_attr_t attr;
    pthread_attr_init(&amp;amp;attr);
    pthread_attr_setdetachstate(&amp;amp;attr, PTHREAD_CREATE_DETACHED);

    pthread_t thread;
    pthread_create(&amp;amp;thread, &amp;amp;attr, worker_thread, nullptr);

    pthread_attr_destroy(&amp;amp;attr);
    // No pthread_join - thread cleans up automatically
}
```

Chapter 2: Thread Synchronization Primitives

2.1 Mutexes (Mutual Exclusion)

Layman's Intuition

A mutex is like a bathroom key. Only one person can hold the key (lock) at a time. Others must wait for the key to be returned (unlocked) before they can enter. This prevents collisions when multiple threads access shared resources.

Technical Foundation

Mutex Properties:

- **Mutual Exclusion:** Only one thread holds lock
- **Ownership:** Only locking thread can unlock
- **Blocking:** Threads wait if lock unavailable
- **Overhead:** ~25-100ns uncontended, ms if contended

POSIX Mutex

```
#include &lt;pthread.h&gt;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int shared_counter = 0;

void* increment_counter(void* arg) {
    for (int i = 0; i &lt; 100000; i++) {
        // Critical section start
        pthread_mutex_lock(&amp;mutex);

        shared_counter++;

        pthread_mutex_unlock(&amp;mutex);
        // Critical section end
    }

    return nullptr;
}

int main() {
    pthread_t threads[10];

    for (int i = 0; i &lt; 10; i++) {
        pthread_create(&amp;threads[i], nullptr, increment_counter, nullptr);
    }

    for (int i = 0; i &lt; 10; i++) {
        pthread_join(threads[i], nullptr);
    }
```

```
    std::cout &lt;&lt; "Final counter: " &lt;&lt; shared_counter &lt;&lt; "\n";
    // Expected: 1,000,000

    pthread_mutex_destroy(&amp;mutex);
    return 0;
}
```

C++11 std::mutex

```
#include &lt;mutex&gt;
#include &lt;thread&gt;
#include &lt;vector&gt;

std::mutex mtx;
int shared_counter = 0;

void increment_counter() {
    for (int i = 0; i &lt; 100000; i++) {
        // Method 1: Manual lock/unlock
        mtx.lock();
        shared_counter++;
        mtx.unlock();
    }
}

void increment_counter_raii() {
    for (int i = 0; i &lt; 100000; i++) {
        // Method 2: RAII with lock_guard (recommended)
        std::lock_guard&lt;std::mutex&gt; lock(mtx);
        shared_counter++;
        // Automatically unlocks when lock goes out of scope
    }
}

void increment_counter_flexible() {
    for (int i = 0; i &lt; 100000; i++) {
        // Method 3: unique_lock (more flexible)
        std::unique_lock&lt;std::mutex&gt; lock(mtx);
        shared_counter++;
        lock.unlock();  // Can unlock early if needed

        // Do non-critical work

        lock.lock();  // Can re-lock
        // More critical work
    }
}
```

Try-Lock Pattern (Non-Blocking)

```
void try_increment() {
    for (int i = 0; i &lt; 100000; i++) {
        if (mtx.try_lock()) {
            // Successfully acquired lock
            shared_counter++;
            mtx.unlock();
        } else {
            // Lock busy, do alternative work or retry
            std::this_thread::yield();  // Give up time slice
        }
    }
}
```

Timed Lock

```
#include &lt;chrono&gt;

void timed_increment() {
    using namespace std::chrono_literals;

    std::timed_mutex tmtx;

    if (tmtx.try_lock_for(100ms)) {
        // Acquired lock within timeout
        shared_counter++;
        tmtx.unlock();
    } else {
        // Timeout - lock unavailable
        std::cout &lt;&lt; "Failed to acquire lock\n";
    }
}
```

2.2 Spinlocks

Layman's Intuition

Instead of sleeping while waiting for a lock (mutex), spinlocks make the thread continuously check ("spin") if the lock is available. Like repeatedly trying a door handle instead of waiting for someone to tell you it's open. Wastes CPU but avoids sleep overhead—critical for ultra-low latency.

Implementation

Basic Spinlock:

```
#include &lt;atomic&gt;

class Spinlock {
private:
```

```
    std::atomic_flag lock_flag = ATOMIC_FLAG_INIT;

public:
    void lock() {
        // Spin until lock acquired
        while (lock_flag.test_and_set(std::memory_order_acquire)) {
            // Busy-wait (spin)
            // CPU executes this loop continuously
        }
    }

    void unlock() {
        lock_flag.clear(std::memory_order_release);
    }
};

// Usage
Spinlock spin;
int counter = 0;

void increment_with_spinlock() {
    for (int i = 0; i &lt; 100000; i++) {
        spin.lock();
        counter++;
        spin.unlock();
    }
}
```

Optimized Spinlock with Pause:

```
class OptimizedSpinlock {
private:
    std::atomic_flag lock_flag = ATOMIC_FLAG_INIT;

public:
    void lock() {
        while (lock_flag.test_and_set(std::memory_order_acquire)) {
            // PAUSE instruction reduces CPU power consumption
            // and improves spin-wait performance
            #if defined(__x86_64__) || defined(_M_X64)
                __builtin_ia32_pause();  // GCC/Clang
                // Or: _mm_pause(); for MSVC
            #elif defined(__aarch64__)
                asm volatile("yield" ::: "memory");
            #endif
        }
    }

    void unlock() {
        lock_flag.clear(std::memory_order_release);
    }
};
```

Exponential Backoff Spinlock:

```
class BackoffSpinlock {
private:
    std::atomic_flag lock_flag = ATOMIC_FLAG_INIT;

public:
    void lock() {
        int backoff = 1;
        const int MAX_BACKOFF = 1024;

        while (lock_flag.test_and_set(std::memory_order_acquire)) {
            // Exponentially increase wait time
            for (int i = 0; i &lt; backoff; i++) {
                __builtin_ia32_pause();
            }

            backoff = std::min(backoff * 2, MAX_BACKOFF);
        }
    }

    void unlock() {
        lock_flag.clear(std::memory_order_release);
    }
};
```

Mutex vs Spinlock Decision

Use Mutex When:

- Lock held for >1μs
- Many threads contend for lock
- Want to avoid wasting CPU
- Not time-critical

Use Spinlock When:

- Lock held for <1μs (ultra-short critical sections)
- Low contention
- Dedicated CPU cores available
- Ultra-low latency required (HFT)

Benchmark:

```
void benchmark_locks() {
    const int ITERATIONS = 1000000;

    // Test mutex
    std::mutex mtx;
    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i &lt; ITERATIONS; i++) {
        mtx.lock();
```

```
        // Empty critical section
        mtx.unlock();
    }

    auto end = std::chrono::high_resolution_clock::now();
    auto mutex_time = std::chrono::duration_cast&lt;std::chrono::nanoseconds&gt;
                      (end - start).count();

    // Test spinlock
    Spinlock spin;
    start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i &lt; ITERATIONS; i++) {
        spin.lock();
        // Empty critical section
        spin.unlock();
    }

    end = std::chrono::high_resolution_clock::now();
    auto spinlock_time = std::chrono::duration_cast&lt;std::chrono::nanoseconds&gt;
                         (end - start).count();

    std::cout &lt;&lt; "Mutex time: " &lt;&lt; mutex_time / (double)ITERATIONS &lt;&lt; '
    std::cout &lt;&lt; "Spinlock time: " &lt;&lt; spinlock_time / (double)ITERATIONS &lt;
}
```

2.3 Read-Write Locks

Concept

Multiple readers can hold lock simultaneously, but writers need exclusive access. Optimizes read-heavy workloads.

Implementation

POSIX Read-Write Lock:

```
#include &lt;pthread.h&gt;

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
int shared_data = 0;

void* reader_thread(void* arg) {
    for (int i = 0; i &lt; 100; i++) {
        pthread_rwlock_rdlock(&amp;rwlock);

        // Multiple readers can execute this simultaneously
        int value = shared_data;
        std::cout &lt;&lt; "Read: " &lt;&lt; value &lt;&lt; "\n";

        pthread_rwlock_unlock(&amp;rwlock);
    }
```

```
    return nullptr;
}

void* writer_thread(void* arg) {
    for (int i = 0; i &amp;lt; 10; i++) {
        pthread_rwlock_wrlock(&amp;amp;rwlock);

        // Exclusive access - no other readers or writers
        shared_data++;
        std::cout &amp;lt;&amp;lt; "Write: " &amp;lt;&amp;lt; shared_data &amp;lt;&amp;lt; "\n";

        pthread_rwlock_unlock(&amp;amp;rwlock);

        sleep(1);  // Simulate work
    }
    return nullptr;
}
```

C++14 Shared Mutex:

```
#include &amp;lt;shared_mutex&amp;gt;

std::shared_mutex rw_mutex;
int shared_resource = 0;

void reader() {
    std::shared_lock&amp;lt;std::shared_mutex&amp;gt; lock(rw_mutex);
    // Multiple readers can acquire shared_lock simultaneously
    int value = shared_resource;
}

void writer() {
    std::unique_lock&amp;lt;std::shared_mutex&amp;gt; lock(rw_mutex);
    // Exclusive access - blocks all readers and writers
    shared_resource++;
}
```

HFT Example: Market Data Cache

```
class MarketDataCache {
private:
    std::shared_mutex mutex;
    std::unordered_map&amp;lt;uint32_t, double&amp;gt; prices;  // symbol_id -&amp;gt; price

public:
    // Read: Many threads query prices simultaneously
    double get_price(uint32_t symbol_id) const {
        std::shared_lock&amp;lt;std::shared_mutex&amp;gt; lock(mutex);

        auto it = prices.find(symbol_id);
        return (it != prices.end()) ? it-&amp;gt;second : 0.0;
    }

    // Write: Single thread updates prices
```

```
    void update_price(uint32_t symbol_id, double price) {
        std::unique_lock&lt;std::shared_mutex&gt; lock(mutex);
        prices[symbol_id] = price;
    }

    // Bulk read (lock once for multiple queries)
    std::vector&lt;double&gt; get_prices(const std::vector&lt;uint32_t&gt;&amp; symbols)
        std::shared_lock&lt;std::shared_mutex&gt; lock(mutex);

        std::vector&lt;double&gt; result;
        result.reserve(symbols.size());

        for (uint32_t sym : symbols) {
            auto it = prices.find(sym);
            result.push_back(it != prices.end() ? it-&gt;second : 0.0);
        }

        return result;
    }
};
```

2.4 Condition Variables

Layman's Intuition

A condition variable is like a waiting room. Threads wait for a specific condition to become true (e.g., "queue not empty"). When another thread changes the state, it signals waiting threads to wake up and check the condition.

Producer-Consumer Pattern

```
#include &lt;queue&gt;
#include &lt;mutex&gt;
#include &lt;condition_variable&gt;

template&lt;typename T&gt;
class ThreadSafeQueue {
private:
    std::queue&lt;T&gt; queue;
    mutable std::mutex mutex;
    std::condition_variable cond_var;

public:
    // Producer: Add item to queue
    void push(T item) {
        {
            std::lock_guard&lt;std::mutex&gt; lock(mutex);
            queue.push(std::move(item));
        }  // Unlock before notify

        cond_var.notify_one();  // Wake up one waiting consumer
    }
```

```
    // Consumer: Wait for and remove item
    T pop() {
        std::unique_lock&lt;std::mutex&gt; lock(mutex);

        // Wait until queue not empty
        cond_var.wait(lock, [this] { return !queue.empty(); });

        T item = std::move(queue.front());
        queue.pop();

        return item;
    }

    // Consumer: Try to pop with timeout
    bool try_pop(T&amp; item, std::chrono::milliseconds timeout) {
        std::unique_lock&lt;std::mutex&gt; lock(mutex);

        if (!cond_var.wait_for(lock, timeout,
                               [this] { return !queue.empty(); })) {
            return false;  // Timeout
        }

        item = std::move(queue.front());
        queue.pop();
        return true;
    }

    // Check if empty (for monitoring)
    bool empty() const {
        std::lock_guard&lt;std::mutex&gt; lock(mutex);
        return queue.empty();
    }
};

// Usage example
void producer_consumer_demo() {
    ThreadSafeQueue&lt;int&gt; queue;

    // Producer thread
    std::thread producer([&amp;queue]() {
        for (int i = 0; i &lt; 100; i++) {
            queue.push(i);
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    });

    // Consumer threads
    std::vector&lt;std::thread&gt; consumers;
    for (int i = 0; i &lt; 3; i++) {
        consumers.emplace_back([&amp;queue, i]() {
            while (true) {
                int item;
                if (queue.try_pop(item, std::chrono::seconds(1))) {
                    std::cout &lt;&lt; "Consumer " &lt;&lt; i &lt;&lt; " got " &lt;&lt; i
                } else {
```

```
                    break;  // Timeout - producer finished
                }
            }
        });
    }

    producer.join();
    for (auto&amp;amp; c : consumers) {
        c.join();
    }
}
```

HFT Market Data Processing Pipeline

```
struct MarketTick {
    uint64_t timestamp;
    uint32_t symbol_id;
    double price;
    uint32_t volume;
};

class MarketDataPipeline {
private:
    ThreadSafeQueue&amp;lt;MarketTick&amp;gt; raw_ticks;
    ThreadSafeQueue&amp;lt;MarketTick&amp;gt; processed_ticks;

    std::atomic&amp;lt;bool&amp;gt; running{true};

public:
    // Stage 1: Network thread receives ticks
    void receive_tick(const MarketTick&amp;amp; tick) {
        raw_ticks.push(tick);
    }

    // Stage 2: Parser thread processes ticks
    void parser_thread() {
        while (running) {
            MarketTick tick = raw_ticks.pop();

            // Process tick (validation, normalization, etc.)
            process_tick(tick);

            processed_ticks.push(tick);
        }
    }

    // Stage 3: Strategy threads consume processed ticks
    void strategy_thread(int id) {
        while (running) {
            MarketTick tick;
            if (processed_ticks.try_pop(tick, std::chrono::milliseconds(100))) {
                // Execute trading logic
                handle_tick(id, tick);
            }
        }
```

```
    }

    void start() {
        // Start parser thread
        std::thread parser(&amp;MarketDataPipeline::parser_thread, this);

        // Start strategy threads
        std::vector&lt;std::thread&gt; strategies;
        for (int i = 0; i &lt; 4; i++) {
            strategies.emplace_back(&amp;MarketDataPipeline::strategy_thread,
                                    this, i);
        }

        // ... run for duration ...

        running = false;

        parser.join();
        for (auto&amp; s : strategies) {
            s.join();
        }
    }

private:
    void process_tick(MarketTick&amp; tick) {
        // Normalization, validation, etc.
    }

    void handle_tick(int strategy_id, const MarketTick&amp; tick) {
        // Trading logic
    }
};
```

2.5 Atomic Operations

Layman's Intuition

Atomic operations are indivisible—they happen completely or not at all, with no interruption possible. Like a vending machine transaction: money goes in and item comes out as one uninterruptible action, not as separate steps.

C++11 Atomics

Basic Atomic Types:

```
#include &lt;atomic&gt;

std::atomic&lt;int&gt; counter{0};
std::atomic&lt;bool&gt; flag{false};
std::atomic&lt;double&gt; price{100.0};  // Requires lock on some platforms
std::atomic&lt;void*&gt; pointer{nullptr};
```

```
// Operations
counter++;                 // Atomic increment
counter.fetch_add(5);     // Atomic add, returns old value
flag.store(true);         // Atomic write
bool old = flag.exchange(false);  // Atomic swap
```

Lock-Free Counter:

```
class LockFreeCounter {
private:
    std::atomic&lt;uint64_t&gt; count{0};

public:
    uint64_t increment() {
        return count.fetch_add(1, std::memory_order_relaxed);
    }

    uint64_t get() const {
        return count.load(std::memory_order_relaxed);
    }

    void reset() {
        count.store(0, std::memory_order_relaxed);
    }
};

// Usage - multiple threads can increment simultaneously
LockFreeCounter counter;

void thread_function() {
    for (int i = 0; i &lt; 100000; i++) {
        counter.increment();  // No locks needed!
    }
}
```

Compare-and-Swap (CAS)

Concept: Atomically check value and update if matches expected.

```
// Atomic version of:
// if (value == expected) {
//     value = desired;
//     return true;
// }
// return false;

std::atomic&lt;int&gt; value{0};

bool cas_example() {
    int expected = 0;
    int desired = 1;

    // Returns true if successful, updates expected if failed
```

```
    return value.compare_exchange_strong(expected, desired);
}
```

Lock-Free Stack:

```
template&lt;typename T&gt;
class LockFreeStack {
private:
    struct Node {
        T data;
        Node* next;

        Node(const T&amp; d) : data(d), next(nullptr) {}
    };

    std::atomic&lt;Node*&gt; head{nullptr};

public:
    void push(const T&amp; data) {
        Node* new_node = new Node(data);

        // Link new node to current head
        new_node-&gt;next = head.load(std::memory_order_relaxed);

        // CAS loop: retry if head changed
        while (!head.compare_exchange_weak(
            new_node-&gt;next,
            new_node,
            std::memory_order_release,
            std::memory_order_relaxed
        )) {
            // Loop continues if CAS failed
            // new_node-&gt;next updated to current head
        }
    }

    bool pop(T&amp; result) {
        Node* old_head = head.load(std::memory_order_relaxed);

        // CAS loop: retry if head changed
        while (old_head &amp;&amp; !head.compare_exchange_weak(
            old_head,
            old_head-&gt;next,
            std::memory_order_acquire,
            std::memory_order_relaxed
        )) {
            // Loop continues if CAS failed
            // old_head updated to current head
        }

        if (old_head) {
            result = old_head-&gt;data;
            delete old_head;
            return true;
        }
```

```
        return false;  // Stack empty
    }
};
```

Memory Ordering

Memory Order Options:

1. **memory_order_relaxed:** No synchronization, only atomicity
2. **memory_order_acquire:** Reads happen-before subsequent operations
3. **memory_order_release:** Writes happen-after previous operations
4. **memory_order_acq_rel:** Both acquire and release
5. **memory_order_seq_cst:** Sequential consistency (strongest, default)

Example: Producer-Consumer Flag:

```
std::atomic&lt;bool&gt; data_ready{false};
int data = 0;

// Producer thread
void producer() {
    data = 42;  // Non-atomic write

    // Release: Ensures data write completes before flag set
    data_ready.store(true, std::memory_order_release);
}

// Consumer thread
void consumer() {
    // Acquire: Ensures flag read happens before data read
    while (!data_ready.load(std::memory_order_acquire)) {
        // Spin-wait
    }

    // Guaranteed to see data = 42
    assert(data == 42);
}
```

This comprehensive textbook continues with similar exhaustive depth covering lock-free data structures, cache optimization, NUMA programming, and production HFT threading patterns. Each section provides theory, multiple implementations, benchmarks, and real-world examples.