Multithreading, Concurrency, Asynchronous Programming in Modern C++

Prepared by Ayman Alheraki

https://simplifycpp.org

Multithreading, Concurrency, Asynchronous Programming in Modern C++

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Contents **2**

Preface **4**

1 Foundations of Concurrency in Modern Systems **5**

1.1 Threads: A Precise Definition 5

1.2 Concurrency vs Parallelism 6

1.3 Asynchrony 6

2 Advanced Threading with `std::thread` **7**

2.1 Thread Creation Overhead 7

2.2 Example: Deterministic Worker Thread 7

2.3 `detach()` vs Lifetime Safety 8

3 Synchronization and Memory Correctness **9**

3.1 The Data Race Problem 9

3.2 Mutex: Correct Use 9

3.3 Advanced: `unique_lock` 10

3.4 Real Producer–Consumer Pattern 10

4 Atomics and Lock-Free Techniques **12**
4.1 Atomic Operations 12
4.2 Lock-Free Counter 12
4.3 Acquire–Release Relationship 13

5 Futures, Promises, and Packaged Tasks **14**
5.1 Returning Data from Threads 14
5.2 Transforming Functions into Future-Producing Tasks 14

6 `std::async`: High-Level Parallel Tasks **16**
6.1 async Launch Policies 16
6.2 Example: Parallel Row Reduction 16

7 C++20 Coroutines: The async/await Model **18**
7.1 Coroutines Are Not Threads 18
7.2 Minimal Awaitable Example 18
7.3 Coroutine Returning a Value 19

8 A Production-Grade Thread Pool **20**
8.1 Why Thread Pools? 20
8.2 Modern Thread Pool Implementation 20
8.3 Submitting Work 22

9 Design Principles and Best Practices **23**
9.1 Modern C++ Concurrency Principles 23
9.2 Debugging Concurrency 23

Conclusion **25**

Preface

Modern hardware is parallel by design. Multi-core architectures, deep execution pipelines, shared caches, NUMA topologies, and hardware threads (SMT) shape the real performance landscape.
Modern C++ gives developers direct access to extremely powerful concurrency tools:

- std::thread, std::jthread
- mutexes, shared mutexes, condition variables
- atomics + memory ordering
- futures, promises, packaged_task
- thread pools
- C++20 coroutines (async/await model)

This booklet provides an `advanced`, `practical`, and `professional-level` guide to concurrency in Modern C++.

Chapter 1

Foundations of Concurrency in Modern Systems

1.1 Threads: A Precise Definition

A thread is not simply “a function that runs concurrently.” It has:

- a private call stack,
- a program counter,
- register state,
- OS scheduler metadata,
- thread-local storage (TLS),
- a scheduling quantum.

Modern CPUs use:

- register renaming,
- out-of-order execution,
- branch prediction + speculation,
- micro-op fusion,
- SMT (simultaneous multi-threading).

Your multithreaded C++ code interacts with all of this.

1.2 Concurrency vs Parallelism

Concurrency = multiple tasks making progress. **Parallelism** = physically executing simultaneously across CPU cores.
C++ supports both.

1.3 Asynchrony

Asynchronous programming means:

- the caller is not blocked;
- the task continues later;
- no requirement to create threads;
- handles I/O naturally and efficiently.

Coroutines shine here.

Chapter 2

Advanced Threading with `std::thread`

2.1 Thread Creation Overhead

A new thread costs:

- memory for stack (1–8 MB),
- kernel object allocation,
- scheduler registration,
- TLS initialization.

Thus, thread pools are essential.

2.2 Example: Deterministic Worker Thread

```
#include <thread>
#include <iostream>
```

```cpp
#include <chrono>

void worker(int id) {
    using namespace std::chrono_literals;
    for(int i = 0; i < 5; ++i) {
        std::cout << "T" << id << " step " << i << "\n";
        std::this_thread::sleep_for(40ms);
    }
}

int main() {
    std::thread t1(worker, 1);
    std::thread t2(worker, 2);
    t1.join();
    t2.join();
}
```

2.3 `detach()` vs Lifetime Safety

Detached threads are dangerous: they can outlive key resources. Prefer:

- std::jthread,
- thread pools,
- structured concurrency,
- coroutines.

Chapter 3

Synchronization and Memory Correctness

3.1 The Data Race Problem

A data race exists when:

- two threads access the same memory,
- at least one writes,
- without synchronization.

This causes undefined behavior.

3.2 Mutex: Correct Use

```
std::mutex m;
int counter = 0;

void inc() {
```

```
    std::lock_guard<std::mutex> lock(m);
    ++counter;
}
```

3.3 Advanced: `unique_lock`

```
std::unique_lock<std::mutex> lock(m, std::try_to_lock);
if (lock.owns_lock()) {
    // Safe access
}
```

3.4 Real Producer–Consumer Pattern

```
std::mutex m;
std::condition_variable cv;
bool ready = false;
int payload = 0;

void producer() {
    {
        std::lock_guard<std::mutex> lock(m);
        payload = 42;
        ready = true;
    }
    cv.notify_one();
}

void consumer() {
    std::unique_lock<std::mutex> lock(m);
    cv.wait(lock, []{ return ready; });
    std::cout << "Data: " << payload << "\n";
```

```
}
```

Chapter 4

Atomics and Lock-Free Techniques

4.1 Atomic Operations

Atomics allow:

- lock-free synchronization,
- reduced latency,
- precise memory ordering.

4.2 Lock-Free Counter

```
std::atomic<int> counter{0};

void work() {
    for(int i = 0; i < 1'000'000; ++i)
        counter.fetch_add(1, std::memory_order_relaxed);
}
```

4.3 Acquire–Release Relationship

```
std::atomic<bool> ready = false;
int data;

void writer() {
    data = 123;
    ready.store(true, std::memory_order_release);
}

void reader() {
    while(!ready.load(std::memory_order_acquire));
    std::cout << data;
}
```

This guarantees visibility order.

Chapter 5

Futures, Promises, and Packaged Tasks

5.1 Returning Data from Threads

```
std::promise<std::string> p;
auto f = p.get_future();

std::thread t([&]{
    p.set_value("Hello");
});

t.join();
std::cout << f.get();
```

5.2 Transforming Functions into Future-Producing Tasks

```
std::packaged_task<int(int)> task([](int x){
    return x * x;
});
```

```
auto f = task.get_future();
std::thread(std::move(task), 7).detach();

std::cout << f.get();
```

Chapter 6

`std::async`: High-Level Parallel Tasks

6.1 async Launch Policies

- **async** → always creates a thread.
- **deferred** → lazy execution on .get().

6.2 Example: Parallel Row Reduction

```
auto sum_row = [&](auto& r) {
    return std::accumulate(r.begin(), r.end(), 0);
};

std::vector<std::future<int>> futs;
for(auto& row : matrix) {
    futs.push_back(std::async(std::launch::async,
                              sum_row, std::ref(row)));
}
```

```
int total = 0;
for(auto& f : futs)
    total += f.get();
```

Chapter 7

C++20 Coroutines: The async/await Model

7.1 Coroutines Are Not Threads

They:

- do not allocate stacks,
- suspend and resume,
- are extremely cheap,
- work perfectly for async I/O.

7.2 Minimal Awaitable Example

```
struct SuspendAlways {
    bool await_ready() const noexcept { return false; }
```

```
    void await_suspend(std::coroutine_handle<>) const noexcept {}
    void await_resume() const noexcept {}
};
```

7.3 Coroutine Returning a Value

```
task<int> compute() {
    co_await suspend_io();
    co_return 99;
}
```

Chapter 8

A Production-Grade Thread Pool

8.1 Why Thread Pools?

Creating new threads repeatedly is expensive. Thread pools solve this by reusing workers.

8.2 Modern Thread Pool Implementation

```
class ThreadPool {
public:
    ThreadPool(size_t n) : stop(false) {
        for(size_t i = 0; i < n; ++i)
            workers.emplace_back([this]{ worker_loop(); });
    }

    ~ThreadPool() {
        {
            std::unique_lock lock(m);
            stop = true;
```

```
        }
        cv.notify_all();
        for(auto& t : workers) t.join();
    }

    template<typename F>
    void submit(F&& f) {
        {
            std::unique_lock lock(m);
            tasks.push(std::function<void()>(f));
        }
        cv.notify_one();
    }

private:
    void worker_loop() {
        while(true) {
            std::function<void()> task;
            {
                std::unique_lock lock(m);
                cv.wait(lock,
                    [&]{ return stop  !tasks.empty(); });
                if(stop && tasks.empty()) return;
                task = std::move(tasks.front());
                tasks.pop();
            }
            task();
        }
    }

    std::mutex m;
    std::condition_variable cv;
    bool stop;
```

```
    std::queue<std::function<void()>> tasks;
    std::vector<std::thread> workers;
};
```

8.3 Submitting Work

```
ThreadPool pool(8);

pool.submit([]{ heavy_compute(); });
pool.submit([]{ render_tile(); });
pool.submit([]{ simulate_physics(); });
```

Chapter 9

Design Principles and Best Practices

9.1 Modern C++ Concurrency Principles

- Prefer std::jthread over std::thread.
- Avoid shared mutable state.
- Prefer message passing and immutability.
- Use lock-free only when necessary.
- Use coroutines for async I/O.
- Use thread pools for CPU-bound workloads.

9.2 Debugging Concurrency

- Thread Sanitizer (TSan)
- Valgrind Helgrind

- perf / VTune

Conclusion

Modern C++ offers a rich concurrency model, blending low-level control with high-level abstractions. Mastering these tools enables the creation of fast, scalable, and robust software for the multi-core era.