C++ Concurrency Programming Guide

A comprehensive guide to mastering concurrency in modern C++

Table of Contents

1. Introduction to Concurrency
2. Threads in C++
3. Synchronization Primitives
4. Atomic Operations
5. Futures and Promises
6. Higher-Level Concurrency Patterns
7. Best Practices and Pitfalls
8. Exercises and Exams
9. Final Project - Concurrent Web Server Simulation

C++ Concurrency Guide

Section 1: Introduction to Concurrency

1.1 What is Concurrency? (vs. Parallelism)

Concurrency is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome. This allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems. In simpler terms, concurrency is about *dealing* with lots of things at once.

Parallelism is about *doing* lots of things at once. It means that multiple tasks or parts of a task literally run at the same time, typically on different processors or cores.

Key Difference:

- **Concurrency** is a property of the program structure. It's about managing multiple tasks that are *in progress* over the same period. These tasks might be interleaved on a single core or run in parallel on multiple cores.
- **Parallelism** is a property of the execution. It requires hardware with multiple processing units and involves tasks running simultaneously.

Think of a chef cooking multiple dishes. Concurrency is the chef managing the different steps for each dish (chopping vegetables for one, checking the oven for another, stirring a pot for a third). Parallelism would be having multiple chefs (or multiple hands) working on different tasks simultaneously.

C++ provides tools to write concurrent code. Whether that code executes in parallel depends on the hardware and the operating system's scheduler.

1.2 Why Use Concurrency?

There are two main reasons to use concurrency in your applications:

1. **Performance (Separation of Concerns & Speedup):**
    - **Utilizing Multiple Cores:** Modern CPUs have multiple cores. Concurrent programming allows you to utilize these cores to perform computations in parallel, potentially speeding up CPU-bound tasks significantly.
    - **Handling I/O Latency:** Many applications wait for Input/Output operations (like reading from a disk, network requests, or user input). Concurrency allows the program to perform other tasks while waiting for I/O, making better use of CPU time.
    - **Task Decomposition:** Complex tasks can often be broken down into smaller, independent sub-tasks that can be executed concurrently.
2. **Responsiveness:**

- **User Interfaces (UI):** In applications with a graphical user interface (GUI), performing long-running tasks on the main UI thread can make the application unresponsive (freeze). By running these tasks on separate threads, the UI thread remains free to handle user input and updates, keeping the application responsive.
- **Servers:** Web servers and other network servers need to handle multiple client connections simultaneously. Concurrency allows a server to manage many clients without making one client wait for another to finish.

1.3 Challenges of Concurrency

While powerful, concurrency introduces significant challenges:

1. **Race Conditions:** Occur when multiple threads access shared data concurrently, and at least one access is a write. The final state of the data depends on the unpredictable order in which threads execute. This leads to non-deterministic behavior and bugs that are hard to reproduce.
2. **Deadlocks:** A situation where two or more threads are blocked forever, each waiting for a resource held by another thread in the group. For example, Thread A holds Lock 1 and waits for Lock 2, while Thread B holds Lock 2 and waits for Lock 1.
3. **Livelocks:** Threads are not blocked, but they are unable to make progress because they are continuously changing state in response to each other without doing useful work. Imagine two people trying to pass each other in a narrow corridor, repeatedly stepping aside in the same direction.
4. **Starvation:** A thread is perpetually denied necessary resources to process its work, often because other threads (e.g., higher priority or just unlucky timing) consistently get the resources first.
5. **Complexity:** Designing, implementing, and debugging concurrent code is inherently more complex than single-threaded code. Issues can be subtle and depend on timing, making them difficult to find and fix.

1.4 C++ Concurrency History

Before C++11, C++ had no standard support for concurrency. Programmers relied on platform-specific APIs (like POSIX threads or Windows threads) or third-party libraries.

- **C++11:** Introduced the foundational concurrency library, including:
  - `std::thread` : For creating and managing threads.
  - `std::mutex` , `std::lock_guard` , `std::unique_lock` : For basic mutual exclusion.
  - `std::condition_variable` : For thread synchronization based on conditions.
  - `std::future` , `std::promise` , `std::packaged_task` , `std::async` : For managing asynchronous tasks and their results.
  - `std::atomic` : For low-level atomic operations.
  - A standardized memory model defining how memory operations behave in concurrent contexts.
- **C++14:** Added `std::shared_timed_mutex` and related lock types for reader-writer locking scenarios.
- **C++17:** Introduced `std::scoped_lock` for easier handling of multiple mutexes, `std::shared_mutex` (standardizing the C++14 feature), and enhanced parallel algorithms in the Standard Library (e.g., `std::for_each` with execution policies like `std::execution::par`).
- **C++20:** Brought significant additions:
  - `std::jthread` : An improved thread class with automatic joining and support for cooperative interruption (`std::stop_token`).
  - `std::counting_semaphore` , `std::binary_semaphore` : Standard semaphore implementations.
  - `std::latch` , `std::barrier` : Synchronization primitives for coordinating groups of threads.
  - Coroutines: A language feature enabling cooperative multitasking and simplifying asynchronous code (though library support is still evolving).

- Enhancements to atomics, including `std::atomic_ref` and floating-point atomics.

Understanding these features and how they build upon each other is crucial for writing modern, efficient, and safe concurrent C++ code.

C++ Concurrency Guide

Section 2: Threads (`std::thread`)

2.1 Creating and Launching Threads

In C++, threads are represented by the `std::thread` class, which was introduced in C++11. A thread is a sequence of instructions that can be executed concurrently with other threads. The C++ Standard Library provides a platform-independent way to create and manage threads.

Basic Thread Creation

To create a thread, you need to:

1. Include the `<thread>` header
2. Create a `std::thread` object with a callable (function, lambda, functor) as its argument

Plain Text

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

// A simple function to be executed in a thread
void hello() {
    std::cout << "Hello from thread!" << std::endl;
}

int main() {
    // Create a thread that executes the hello function
    std::thread t(hello);

    // Main thread continues execution here concurrently with t
    std::cout << "Hello from main thread!" << std::endl;
```

```
    // Wait for the thread to finish
    t.join();

    return 0;
}
```

In this example:

- We define a simple function `hello()` that prints a message
- We create a thread `t` that executes this function
- The main thread continues execution concurrently
- We call `join()` to wait for the thread to complete before exiting the program

Using Lambdas with Threads

You can also use lambda expressions to define the code to be executed in a thread:

Plain Text

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

int main() {
    // Create a thread with a lambda expression
    std::thread t([]() {
        std::cout << "Hello from lambda thread!" << std::endl;
    });

    std::cout << "Hello from main thread!" << std::endl;

    t.join();
    return 0;
}
```

Using Member Functions as Thread Entry Points

You can also use member functions of classes as thread entry points:

Plain Text

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

class Worker {
public:
    void doWork(int id) {
        std::cout << "Worker " << id << " is working" << std::endl;
    }
};

int main() {
    Worker w;
    // Create a thread that calls the doWork member function on object w
    std::thread t(&Worker::doWork, &w, 1);

    t.join();
    return 0;
}
```

In this example:

- We create a `Worker` class with a `doWork` method
- We create a thread that calls `w.doWork(1)`
- The first argument is the member function pointer
- The second argument is the object pointer (or reference)
- Additional arguments are passed to the member function

2.2 Joining and Detaching Threads

Once you've created a thread, you have two options for handling its lifetime:

Joining Threads

Calling `join()` on a thread object makes the calling thread wait until the thread represented by the object has finished executing:

```
Plain Text
```

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

void work() {
    std::cout << "Thread starting work..." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(2)); // Simulate work
    std::cout << "Thread finished work!" << std::endl;
}

int main() {
    std::thread t(work);

    std::cout << "Main thread waiting for worker to finish..." << std::endl;
    t.join(); // Main thread waits here until t finishes
    std::cout << "Worker thread has finished!" << std::endl;

    return 0;
}
```

Important points about `join()`:

- It blocks the calling thread until the thread it's called on completes
- It can only be called once on a thread object
- After joining, the thread object is no longer associated with an actual thread of execution
- If an exception is thrown before `join()` is called, the thread might continue running after `main()` exits, which can lead to undefined behavior

Detaching Threads

Calling `detach()` disconnects the thread object from the actual thread of execution, allowing the thread to run independently:

Plain Text

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

```
void background_task() {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "Background task completed!" << std::endl;
}

int main() {
    std::thread t(background_task);

    t.detach(); // Detach the thread - it will run independently

    std::cout << "Main thread continues execution..." << std::endl;
    // Main might finish before the background task completes

    // Sleep to give the background thread time to complete
    // (In real code, you'd use a different synchronization mechanism)
    std::this_thread::sleep_for(std::chrono::seconds(3));

    return 0;
}
```

Important points about `detach()`:

- After detaching, the thread runs independently of the thread object
- You can no longer communicate with or wait for a detached thread
- When the program exits, any detached threads still running are terminated
- Use detached threads for "fire and forget" background tasks
- Be careful with detached threads accessing local variables that might go out of scope

Checking Thread Status

You can check if a thread is joinable (i.e., it represents an active thread of execution):

Plain Text

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

void task() {
    std::cout << "Task executing..." << std::endl;
}
```

```
int main() {
    std::thread t(task);

    if (t.joinable()) {
        std::cout << "Thread is joinable" << std::endl;
        t.join();
    }

    // After joining, the thread is no longer joinable
    if (!t.joinable()) {
        std::cout << "Thread is not joinable anymore" << std::endl;
    }

    return 0;
}
```

2.3 Passing Arguments to Threads

You can pass arguments to the function executed by a thread by providing them after the function in the thread constructor:

Plain Text

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

void greeting(std::string name, int age) {
    std::cout << "Hello, " << name << "! You are " << age << " years old." <<
std::endl;
}

int main() {
    // Pass arguments to the thread function
    std::thread t(greeting, "Alice", 30);

    t.join();
    return 0;
}
```

Avoiding Common Pitfalls with Arguments

Pitfall 1: Reference Arguments

When passing references to a thread, you need to use `std::ref` or `std::cref`:

Plain Text

```
#include <iostream>
#include <thread>
#include <string>
#include <functional> // For std::ref

void modify(std::string& str) {
    str = "Modified by thread";
}

int main() {
    std::string data = "Original";

    // WRONG: This would pass a copy of data
    // std::thread t(modify, data);

    // CORRECT: Use std::ref to pass by reference
    std::thread t(modify, std::ref(data));

    t.join();
    std::cout << "Data: " << data << std::endl; // Prints "Modified by
thread"

    return 0;
}
```

Pitfall 2: Temporary Objects and Dangling References

Be careful when passing references to local variables that might go out of scope:

Plain Text

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

void process(const std::string& str) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    // By this time, the string might be destroyed if not handled properly
    std::cout << "Processing: " << str << std::endl;
}

void dangerous() {
```

```
    std::string local_str = "Temporary string";

    // DANGEROUS: local_str might be destroyed before the thread uses it
    std::thread t(process, std::ref(local_str));
    t.detach(); // This makes it even worse

    // Function exits, local_str is destroyed
}

void safe() {
    std::string local_str = "Temporary string";

    // SAFE: We join the thread before local_str goes out of scope
    std::thread t(process, std::ref(local_str));
    t.join();

    // Function exits, local_str is destroyed
}

int main() {
    // Uncomment to see the dangerous behavior (might crash or show undefined
behavior)
    // dangerous();

    safe();

    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 0;
}
```

Pitfall 3: Moving Arguments

Sometimes you want to transfer ownership of an object to a thread. Use `std::move` for this:

Plain Text

```
#include <iostream>
#include <thread>
#include <vector>
#include <utility> // For std::move

void process_data(std::vector<int> data) {
    std::cout << "Processing " << data.size() << " elements in thread" <<
std::endl;
    // Process the data...
}
```

```
int main() {
    std::vector<int> big_data(10000, 1); // A large vector

    // Move the data to the thread instead of copying it
    std::thread t(process_data, std::move(big_data));

    // big_data is now in a valid but unspecified state
    std::cout << "big_data size after move: " << big_data.size() <<
std::endl;

    t.join();
    return 0;
}
```

2.4 Thread Management (RAII, `std::jthread` in C++20)

RAII for Thread Management

RAII (Resource Acquisition Is Initialization) is a programming idiom where resource management is tied to object lifetime. For threads, this means ensuring that `join()` is called when the thread object goes out of scope:

Plain Text

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

class thread_guard {
private:
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_) : t(t_) {}

    ~thread_guard() {
        if (t.joinable()) {
            t.join();
        }
    }

    // Prevent copying and assignment
    thread_guard(const thread_guard&) = delete;
    thread_guard& operator=(const thread_guard&) = delete;
};

void some_function() {
```

```
    std::cout << "Doing some work..." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

void function_that_might_throw() {
    std::thread t(some_function);
    thread_guard g(t); // RAII guard ensures t is joined

    // If an exception is thrown here, g's destructor will join t
    std::cout << "Function that might throw an exception" << std::endl;

    // g goes out of scope here, joining t
}

int main() {
    function_that_might_throw();
    return 0;
}
```

`std::jthread` (C++20)

C++20 introduced `std::jthread`, which automatically joins in its destructor and supports cooperative cancellation:

Plain Text

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

// Note: This requires C++20 support
void long_running_operation(std::stop_token stoken) {
    for (int i = 0; i < 10 && !stoken.stop_requested(); ++i) {
        std::cout << "Working... " << i << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
    }

    if (stoken.stop_requested()) {
        std::cout << "Operation was interrupted" << std::endl;
    } else {
        std::cout << "Operation completed normally" << std::endl;
    }
}

int main() {
    {
```

```
        // Create a jthread that automatically joins in its destructor
        std::jthread worker(long_running_operation);

        // Do some work in the main thread
        std::this_thread::sleep_for(std::chrono::seconds(2));

        // Request the worker to stop
        std::cout << "Requesting worker to stop" << std::endl;
        worker.request_stop();

        // worker will be joined automatically when it goes out of scope
    }

    std::cout << "Worker has been joined" << std::endl;

    return 0;
}
```

Benefits of `std::jthread`:

- Automatically joins in its destructor
- Provides a cancellation mechanism via `stop_token`
- Simplifies thread management and makes code safer

2.5 Thread Identification (`std::this_thread::get_id()`)

Each thread has a unique identifier that can be accessed using `std::this_thread::get_id()` or `thread::get_id()`:

Plain Text

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

void print_id() {
    std::ostringstream oss;
    oss << "Thread ID: " << std::this_thread::get_id() << std::endl;
    std::cout << oss.str();
}

int main() {
    std::thread t1(print_id);
```

```
    std::thread t2(print_id);

    std::cout << "Main thread ID: " << std::this_thread::get_id() <<
std::endl;
    std::cout << "t1's ID: " << t1.get_id() << std::endl;
    std::cout << "t2's ID: " << t2.get_id() << std::endl;

    t1.join();
    t2.join();

    return 0;
}
```

Thread IDs are useful for:

- Debugging and logging
- Identifying which thread is executing a particular piece of code
- Thread-specific behavior or data structures

2.6 Thread-Local Storage (`thread_local`)

`thread_local` is a storage class specifier that indicates a variable has thread storage duration, meaning each thread has its own copy of the variable:

Plain Text

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

// Each thread will have its own copy of this variable
thread_local std::string thread_name = "Unnamed";

void set_thread_name(const std::string& name) {
    thread_name = name;
}

void print_thread_name() {
    std::cout << "Thread [" << std::this_thread::get_id()
              << "] name: " << thread_name << std::endl;
}

void thread_function(const std::string& name) {
```

```
    set_thread_name(name);
    print_thread_name();
}

int main() {
    set_thread_name("Main Thread");
    print_thread_name();

    std::thread t1(thread_function, "Worker 1");
    std::thread t2(thread_function, "Worker 2");

    t1.join();
    t2.join();

    // The main thread's copy is unchanged
    print_thread_name();

    return 0;
}
```

Benefits of `thread_local`:

- Thread-safe access to thread-specific data without synchronization
- Useful for thread-specific caches, counters, or state
- Can improve performance by reducing contention

2.7 Practical Example: Parallel Sum

Let's put together what we've learned to create a practical example that calculates the sum of a large vector by dividing the work among multiple threads:

Plain Text

```
#include <iostream>
#include <thread>
#include <vector>
#include <numeric>
#include <functional>
#include <chrono>

// Function to sum a range of elements
void sum_range(const std::vector<int>& data, size_t start, size_t end, int&
result) {
```

```
    result = std::accumulate(data.begin() + start, data.begin() + end, 0);
}

int main() {
    // Create a large vector with 100 million elements
    const size_t size = 100'000'000;
    std::vector<int> data(size, 1); // All elements are 1

    // Get the number of supported threads
    unsigned int num_threads = std::thread::hardware_concurrency();
    // Ensure at least 2 threads
    num_threads = (num_threads > 0) ? num_threads : 2;

    std::cout << "Using " << num_threads << " threads" << std::endl;

    // Vector to store partial sums
    std::vector<int> partial_sums(num_threads);

    // Vector to store thread objects
    std::vector<std::thread> threads;

    // Record start time
    auto start_time = std::chrono::high_resolution_clock::now();

    // Divide the work among threads
    size_t chunk_size = size / num_threads;

    for (unsigned int i = 0; i < num_threads; ++i) {
        size_t start = i * chunk_size;
        size_t end = (i == num_threads - 1) ? size : (i + 1) * chunk_size;

        threads.emplace_back(sum_range, std::ref(data), start, end,
std::ref(partial_sums[i]));
    }

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

    // Calculate final sum
    int total_sum = std::accumulate(partial_sums.begin(), partial_sums.end(),
0);

    // Record end time
    auto end_time = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>
(end_time - start_time);
```

```
    std::cout << "Sum: " << total_sum << std::endl;
    std::cout << "Time taken: " << duration.count() << " ms" << std::endl;

    // Verify with single-threaded calculation
    start_time = std::chrono::high_resolution_clock::now();
    int control_sum = std::accumulate(data.begin(), data.end(), 0);
    end_time = std::chrono::high_resolution_clock::now();
    duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time
- start_time);

    std::cout << "Control sum: " << control_sum << std::endl;
    std::cout << "Single-threaded time: " << duration.count() << " ms" <<
std::endl;

    return 0;
}
```

This example demonstrates:

- Creating multiple threads to divide work
- Passing references to thread functions
- Joining threads to wait for completion
- Measuring performance improvements from parallelism

2.8 Common Pitfalls and Best Practices

Pitfall: Oversubscription

Creating too many threads can lead to oversubscription, where there are more active threads than CPU cores, causing excessive context switching:

Plain Text

```
#include <iostream>
#include <thread>
#include <vector>
#include <chrono>

void do_work(int id) {
    std::cout << "Thread " << id << " starting" << std::endl;
```

```
    // Simulate CPU-intensive work
    for (int i = 0; i < 100'000'000; ++i) {
        // Just burn CPU cycles
        double result = std::sin(i) * std::cos(i);
        if (i % 10'000'000 == 0) {
            // Prevent optimization
            std::cout << ".";
        }
    }
    std::cout << "\nThread " << id << " finished" << std::endl;
}

int main() {
    unsigned int num_cores = std::thread::hardware_concurrency();
    std::cout << "This system has " << num_cores << " hardware threads" <<
std::endl;

    // Good practice: Create threads based on available cores
    std::vector<std::thread> optimal_threads;
    auto start = std::chrono::high_resolution_clock::now();

    for (unsigned int i = 0; i < num_cores; ++i) {
        optimal_threads.emplace_back(do_work, i);
    }

    for (auto& t : optimal_threads) {
        t.join();
    }

    auto end = std::chrono::high_resolution_clock::now();
    auto optimal_duration =
std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "Optimal threads time: " << optimal_duration.count() << "
ms" << std::endl;

    // Bad practice: Create too many threads
    std::vector<std::thread> too_many_threads;
    start = std::chrono::high_resolution_clock::now();

    for (unsigned int i = 0; i < num_cores * 4; ++i) {
        too_many_threads.emplace_back(do_work, i);
    }

    for (auto& t : too_many_threads) {
        t.join();
    }

    end = std::chrono::high_resolution_clock::now();
```

```
    auto excessive_duration =
std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "Too many threads time: " << excessive_duration.count() << "
ms" << std::endl;

    return 0;
}
```

Best Practices

1. **Use the Right Number of Threads**:
   - For CPU-bound tasks, use `std::thread::hardware_concurrency()` to determine the optimal number of threads
   - For I/O-bound tasks, you might use more threads than cores
2. **Always Join or Detach Threads**:
   - A thread must be either joined or detached before its destructor is called
   - Use RAII or `std::jthread` to ensure proper cleanup
3. **Be Careful with Shared Data**:
   - Use proper synchronization mechanisms (mutexes, atomic operations) when sharing data between threads
   - Minimize shared mutable state
4. **Avoid Data Races**:
   - Ensure that when multiple threads access the same data, all accesses are properly synchronized
   - Use thread-local storage when appropriate
5. **Consider Higher-Level Abstractions**:
   - Thread pools for managing multiple tasks
   - Task-based programming with `std::async` and futures

- Parallel algorithms from the standard library (C++17 and later)

6. **Profile and Measure**:
   - Don't assume threading always improves performance
   - Measure and compare different approaches
   - Be aware of overhead from thread creation and synchronization

C++ Concurrency Guide

Section 3: Synchronization Primitives

3.1 Mutexes

Mutexes (mutual exclusion objects) are synchronization primitives that prevent multiple threads from simultaneously accessing shared resources. They are fundamental building blocks for thread synchronization in C++.

3.1.1 Basic Mutex Usage (`std::mutex`)

The most basic mutex in C++ is `std::mutex`, which provides exclusive, non-recursive ownership semantics:

Plain Text

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

std::mutex print_mutex; // Mutex to protect console output
std::mutex data_mutex;  // Mutex to protect shared data

// Shared data
std::vector<int> shared_data;

void print_safe(const std::string& message) {
    // Lock the mutex before printing
    print_mutex.lock();
```

```
    std::cout << message << std::endl;
    // Unlock the mutex after printing
    print_mutex.unlock();
}

void add_to_vector(int value) {
    // Lock the data mutex before modifying shared data
    data_mutex.lock();
    shared_data.push_back(value);
    print_safe("Added value: " + std::to_string(value) +
               ", vector size: " + std::to_string(shared_data.size()));
    // Unlock the data mutex after modifying
    data_mutex.unlock();
}

int main() {
    std::vector<std::thread> threads;

    // Create 10 threads, each adding its ID to the vector
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(add_to_vector, i);
    }

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

    // Print the final vector contents
    print_safe("Final vector contents:");
    data_mutex.lock();
    for (int value : shared_data) {
        std::cout << value << " ";
    }
    std::cout << std::endl;
    data_mutex.unlock();

    return 0;
}
```

In this example:

- We use two mutexes: one to protect console output and another to protect a shared vector

- Each thread locks the mutex before accessing the shared resource and unlocks it afterward
- This ensures that only one thread can modify the shared data at a time, preventing race conditions

However, this approach has several issues:

1. If an exception is thrown between `lock()` and `unlock()`, the mutex remains locked
2. It's easy to forget to unlock the mutex
3. The code is verbose and error-prone

3.1.2 RAII Lock Guards (`std::lock_guard`, `std::unique_lock`)

C++ provides RAII (Resource Acquisition Is Initialization) wrappers for mutexes that automatically lock the mutex when constructed and unlock it when destroyed:

Plain Text

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

std::mutex print_mutex;
std::mutex data_mutex;
std::vector<int> shared_data;

void print_safe(const std::string& message) {
    // Use lock_guard for RAII-style locking
    std::lock_guard<std::mutex> guard(print_mutex);
    std::cout << message << std::endl;
    // Mutex is automatically unlocked when guard goes out of scope
}

void add_to_vector(int value) {
    // Use lock_guard for RAII-style locking
    std::lock_guard<std::mutex> guard(data_mutex);
    shared_data.push_back(value);
    print_safe("Added value: " + std::to_string(value) +
               ", vector size: " + std::to_string(shared_data.size()));
    // Mutex is automatically unlocked when guard goes out of scope
}
```

```
int main() {
    std::vector<std::thread> threads;

    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(add_to_vector, i);
    }

    for (auto& t : threads) {
        t.join();
    }

    print_safe("Final vector contents:");
    {
        std::lock_guard<std::mutex> guard(data_mutex);
        for (int value : shared_data) {
            std::cout << value << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

Benefits of using `std::lock_guard`:

- Automatically unlocks the mutex when it goes out of scope, even if an exception is thrown
- Prevents forgetting to unlock the mutex
- Makes the code cleaner and less error-prone

3.1.3 More Flexible Locking with `std::unique_lock`

`std::unique_lock` provides more flexibility than `std::lock_guard`:

Plain Text

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

std::mutex resource_mutex;
```

```
void worker(int id) {
    std::cout << "Worker " << id << " trying to acquire lock" << std::endl;

    // Create a unique_lock without immediately locking the mutex
    std::unique_lock<std::mutex> lock(resource_mutex, std::defer_lock);

    // Do some preparation work without holding the lock
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    // Now lock the mutex
    std::cout << "Worker " << id << " attempting to lock" << std::endl;
    lock.lock();

    std::cout << "Worker " << id << " acquired lock" << std::endl;

    // Simulate working with the protected resource
    std::this_thread::sleep_for(std::chrono::milliseconds(500));

    // Temporarily release the lock to allow other threads to work
    std::cout << "Worker " << id << " temporarily releasing lock" <<
std::endl;
    lock.unlock();

    // Do some more work without holding the lock
    std::this_thread::sleep_for(std::chrono::milliseconds(200));

    // Reacquire the lock
    std::cout << "Worker " << id << " reacquiring lock" << std::endl;
    lock.lock();

    std::cout << "Worker " << id << " finishing work with lock held" <<
std::endl;

    // Lock is automatically released when lock goes out of scope
}

int main() {
    std::thread t1(worker, 1);
    std::thread t2(worker, 2);

    t1.join();
    t2.join();

    return 0;
}
```

Features of `std::unique_lock`:

- Can be constructed without immediately locking the mutex (`std::defer_lock`)
- Allows explicit `lock()` and `unlock()` operations
- Can be used with condition variables (covered later)
- Can be moved but not copied
- Slightly more overhead than `std::lock_guard`

3.1.4 Types of Mutexes

C++ provides several types of mutexes for different use cases:

1. `std::recursive_mutex`

Allows the same thread to lock the mutex multiple times:

Plain Text

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

std::recursive_mutex rm;

void recursive_function(int depth) {
    std::lock_guard<std::recursive_mutex> lock(rm);
    std::cout << "Recursion depth: " << depth << std::endl;

    if (depth > 0) {
        // This would deadlock with a regular mutex
        recursive_function(depth - 1);
    }
}

int main() {
    recursive_function(5);
    return 0;
}
```

2. `std::timed_mutex`

Allows attempts to acquire the lock with a timeout:

Plain Text

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

std::timed_mutex tm;

void worker(int id) {
    // Try to acquire the lock for 500ms
    if (tm.try_lock_for(std::chrono::milliseconds(500))) {
        std::cout << "Worker " << id << " acquired lock" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        tm.unlock();
    } else {
        std::cout << "Worker " << id << " couldn't acquire lock within
timeout" << std::endl;
    }
}

int main() {
    // First thread acquires the lock
    tm.lock();
    std::cout << "Main thread acquired lock" << std::endl;

    // Start worker thread that will try to acquire the lock
    std::thread t(worker, 1);

    // Sleep for 2 seconds, keeping the lock
    std::this_thread::sleep_for(std::chrono::seconds(2));

    // Release the lock
    std::cout << "Main thread releasing lock" << std::endl;
    tm.unlock();

    t.join();
    return 0;
}
```

3. `std::recursive_timed_mutex`

Combines the features of `std::recursive_mutex` and `std::timed_mutex`.

3.1.5 Avoiding Deadlocks

A deadlock occurs when two or more threads are blocked forever, each waiting for a resource held by another thread. Here's a classic example:

Plain Text

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

std::mutex mutex1;
std::mutex mutex2;

void thread1_function() {
    std::cout << "Thread 1: Trying to lock mutex1..." << std::endl;
    std::lock_guard<std::mutex> lock1(mutex1);
    std::cout << "Thread 1: Locked mutex1" << std::endl;

    // Sleep to increase the chance of a deadlock
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    std::cout << "Thread 1: Trying to lock mutex2..." << std::endl;
    std::lock_guard<std::mutex> lock2(mutex2);
    std::cout << "Thread 1: Locked mutex2" << std::endl;

    // Do work with both mutexes locked
    std::cout << "Thread 1: Working with both mutexes" << std::endl;
}

void thread2_function() {
    std::cout << "Thread 2: Trying to lock mutex2..." << std::endl;
    std::lock_guard<std::mutex> lock2(mutex2);
    std::cout << "Thread 2: Locked mutex2" << std::endl;

    // Sleep to increase the chance of a deadlock
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    std::cout << "Thread 2: Trying to lock mutex1..." << std::endl;
    std::lock_guard<std::mutex> lock1(mutex1);
    std::cout << "Thread 2: Locked mutex1" << std::endl;

    // Do work with both mutexes locked
    std::cout << "Thread 2: Working with both mutexes" << std::endl;
}
```

```
int main() {
    std::thread t1(thread1_function);
    std::thread t2(thread2_function);

    t1.join();
    t2.join();

    return 0;
}
```

This code will likely deadlock because:

1. Thread 1 locks mutex1, then tries to lock mutex2
2. Thread 2 locks mutex2, then tries to lock mutex1
3. Both threads are waiting for a mutex held by the other

Solution 1: Lock Ordering

Always lock mutexes in the same order:

Plain Text

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

std::mutex mutex1;
std::mutex mutex2;

void thread_function(int id) {
    // Always lock mutex1 first, then mutex2
    std::lock_guard<std::mutex> lock1(mutex1);
    std::cout << "Thread " << id << ": Locked mutex1" << std::endl;

    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    std::lock_guard<std::mutex> lock2(mutex2);
    std::cout << "Thread " << id << ": Locked mutex2" << std::endl;

    // Do work with both mutexes locked
    std::cout << "Thread " << id << ": Working with both mutexes" <<
std::endl;
}
```

```
int main() {
    std::thread t1(thread_function, 1);
    std::thread t2(thread_function, 2);

    t1.join();
    t2.join();

    return 0;
}
```

Solution 2: `std::lock` and `std::scoped_lock`

C++ provides functions to lock multiple mutexes atomically, avoiding the possibility of deadlock:

Plain Text

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

std::mutex mutex1;
std::mutex mutex2;

void thread_function_with_lock(int id) {
    std::unique_lock<std::mutex> lock1(mutex1, std::defer_lock);
    std::unique_lock<std::mutex> lock2(mutex2, std::defer_lock);

    // Lock both mutexes atomically
    std::lock(lock1, lock2);

    std::cout << "Thread " << id << ": Locked both mutexes" << std::endl;

    // Do work with both mutexes locked
    std::this_thread::sleep_for(std::chrono::milliseconds(500));

    std::cout << "Thread " << id << ": Releasing both mutexes" << std::endl;
    // Mutexes are automatically released when lock1 and lock2 go out of
scope
}

// C++17 version with scoped_lock
void thread_function_with_scoped_lock(int id) {
    // Lock both mutexes atomically with scoped_lock (C++17)
```

```
    std::scoped_lock lock(mutex1, mutex2);

    std::cout << "Thread " << id << ": Locked both mutexes with scoped_lock"
<< std::endl;

    // Do work with both mutexes locked
    std::this_thread::sleep_for(std::chrono::milliseconds(500));

    std::cout << "Thread " << id << ": Releasing both mutexes" << std::endl;
    // Mutexes are automatically released when lock goes out of scope
}

int main() {
    // Using std::lock
    {
        std::thread t1(thread_function_with_lock, 1);
        std::thread t2(thread_function_with_lock, 2);

        t1.join();
        t2.join();
    }

    std::cout << "\nNow using std::scoped_lock (C++17):\n" << std::endl;

    // Using std::scoped_lock (C++17)
    {
        std::thread t1(thread_function_with_scoped_lock, 1);
        std::thread t2(thread_function_with_scoped_lock, 2);

        t1.join();
        t2.join();
    }

    return 0;
}
```

3.2 Condition Variables

Condition variables allow threads to wait until a specific condition is met, typically used for signaling between threads.

3.2.1 Basic Usage of `std::condition_variable`

Plain Text

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <chrono>

std::mutex mtx;
std::condition_variable cv;
std::queue<int> data_queue;
bool finished = false;

void producer() {
    for (int i = 0; i < 10; ++i) {
        // Simulate work
        std::this_thread::sleep_for(std::chrono::milliseconds(500));

        // Add data to the queue
        {
            std::lock_guard<std::mutex> lock(mtx);
            data_queue.push(i);
            std::cout << "Produced: " << i << std::endl;
        }

        // Notify one waiting thread
        cv.notify_one();
    }

    // Signal that production is finished
    {
        std::lock_guard<std::mutex> lock(mtx);
        finished = true;
    }

    // Notify all waiting threads that we're done
    cv.notify_all();
}

void consumer() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);

        // Wait until there's data or we're finished
        cv.wait(lock, [] { return !data_queue.empty() || finished; });

        // If queue is empty and production is finished, exit
        if (data_queue.empty() && finished) {
```

```
            std::cout << "Consumer exiting" << std::endl;
            break;
        }

        // Process data
        int value = data_queue.front();
        data_queue.pop();

        std::cout << "Consumed: " << value << std::endl;

        // Release the lock while processing
        lock.unlock();

        // Simulate processing time
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    }
}

int main() {
    std::thread prod(producer);
    std::thread cons(consumer);

    prod.join();
    cons.join();

    return 0;
}
```

Key points about condition variables:

- They are always used with a mutex
- `wait()` atomically releases the mutex and puts the thread to sleep
- When notified, the thread wakes up, reacquires the mutex, and checks the condition
- The condition is checked in a loop to handle spurious wakeups

3.2.2 Spurious Wakeups

Condition variables may wake up even when not notified (spurious wakeups). Always use a predicate with `wait()`:

```
Plain Text
```

```
// BAD: No predicate check
cv.wait(lock);  // Might wake up spuriously

// GOOD: With predicate check
cv.wait(lock, [] { return !data_queue.empty() || finished; });
```

3.2.3 Timed Waits

Condition variables support timed waits:

Plain Text

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void worker() {
    std::unique_lock<std::mutex> lock(mtx);

    std::cout << "Worker: Waiting for signal or timeout..." << std::endl;

    // Wait for up to 2 seconds
    auto status = cv.wait_for(lock, std::chrono::seconds(2), [] { return
ready; });

    if (status) {
        std::cout << "Worker: Condition met!" << std::endl;
    } else {
        std::cout << "Worker: Timeout occurred!" << std::endl;
    }
}

void signaler(bool signal) {
    std::this_thread::sleep_for(std::chrono::seconds(1));

    if (signal) {
        std::lock_guard<std::mutex> lock(mtx);
        ready = true;
        std::cout << "Signaler: Sending signal" << std::endl;
        cv.notify_one();
```

```
    } else {
        std::cout << "Signaler: Not sending any signal" << std::endl;
    }
}

int main() {
    // Test with signal
    {
        ready = false;
        std::thread w(worker);
        std::thread s(signaler, true);
        w.join();
        s.join();
    }

    std::cout << "\n";

    // Test with timeout
    {
        ready = false;
        std::thread w(worker);
        std::thread s(signaler, false);
        w.join();
        s.join();
    }

    return 0;
}
```

3.2.4 Producer-Consumer Pattern

A complete example of the producer-consumer pattern using a thread-safe queue:

Plain Text

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <vector>
#include <chrono>
#include <random>

// Thread-safe queue
template<typename T>
```

```
class ThreadSafeQueue {
private:
    std::queue<T> queue;
    mutable std::mutex mtx;
    std::condition_variable cv;
    bool finished = false;

public:
    // Push an item to the queue
    void push(T item) {
        std::lock_guard<std::mutex> lock(mtx);
        queue.push(std::move(item));
        cv.notify_one();
    }

    // Try to pop an item from the queue
    bool try_pop(T& item) {
        std::lock_guard<std::mutex> lock(mtx);
        if (queue.empty()) {
            return false;
        }

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

    // Wait and pop an item from the queue
    bool wait_and_pop(T& item) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [this] { return !queue.empty() || finished; });

        if (queue.empty() && finished) {
            return false;
        }

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

    // Check if the queue is empty
    bool empty() const {
        std::lock_guard<std::mutex> lock(mtx);
        return queue.empty();
    }

    // Signal that no more items will be pushed
```

```
    void finish() {
        std::lock_guard<std::mutex> lock(mtx);
        finished = true;
        cv.notify_all();
    }

    // Check if the queue is finished and empty
    bool is_finished() const {
        std::lock_guard<std::mutex> lock(mtx);
        return finished && queue.empty();
    }
};

// Producer function
void producer(ThreadSafeQueue<int>& queue, int id, int num_items) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(100, 1000);

    for (int i = 0; i < num_items; ++i) {
        // Simulate variable work time
        std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));

        int item = id * 1000 + i;
        queue.push(item);
        std::cout << "Producer " << id << " produced: " << item << std::endl;
    }

    std::cout << "Producer " << id << " finished" << std::endl;
}

// Consumer function
void consumer(ThreadSafeQueue<int>& queue, int id) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(200, 500);

    while (true) {
        int item;
        if (queue.wait_and_pop(item)) {
            std::cout << "Consumer " << id << " consumed: " << item <<
std::endl;

            // Simulate processing time
            std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));
        } else {
            // Queue is finished and empty
            break;
```

```
        }
    }

    std::cout << "Consumer " << id << " finished" << std::endl;
}

int main() {
    ThreadSafeQueue<int> queue;

    // Create producers
    const int num_producers = 3;
    const int items_per_producer = 5;
    std::vector<std::thread> producers;

    for (int i = 0; i < num_producers; ++i) {
        producers.emplace_back(producer, std::ref(queue), i + 1,
items_per_producer);
    }

    // Create consumers
    const int num_consumers = 2;
    std::vector<std::thread> consumers;

    for (int i = 0; i < num_consumers; ++i) {
        consumers.emplace_back(consumer, std::ref(queue), i + 1);
    }

    // Wait for producers to finish
    for (auto& p : producers) {
        p.join();
    }

    // Signal that production is finished
    queue.finish();

    // Wait for consumers to finish
    for (auto& c : consumers) {
        c.join();
    }

    return 0;
}
```

3.3 Semaphores (C++20)

Semaphores are synchronization primitives that allow a specified number of threads to access a resource simultaneously.

3.3.1 `std::counting_semaphore` and `std::binary_semaphore`

Plain Text

```
#include <iostream>
#include <thread>
#include <semaphore>  // C++20
#include <vector>
#include <chrono>

// Limit concurrent access to 3 threads
std::counting_semaphore<3> resource_semaphore(3);  // Initial count is 3

void worker(int id) {
    std::cout << "Worker " << id << " waiting for resource..." << std::endl;

    // Acquire the semaphore
    resource_semaphore.acquire();

    std::cout << "Worker " << id << " acquired the resource" << std::endl;

    // Simulate using the resource
    std::this_thread::sleep_for(std::chrono::seconds(2));

    std::cout << "Worker " << id << " releasing the resource" << std::endl;

    // Release the semaphore
    resource_semaphore.release();
}

int main() {
    std::vector<std::thread> threads;

    // Create 5 worker threads
    for (int i = 0; i < 5; ++i) {
        threads.emplace_back(worker, i + 1);
    }

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

```
    return 0;
}
```

In this example:

- We create a counting semaphore with a maximum count of 3
- Only 3 threads can acquire the semaphore at the same time
- Other threads must wait until the semaphore is released

3.3.2 Binary Semaphore (Mutex Alternative)

A binary semaphore is a counting semaphore with a maximum count of 1, which can be used as a lightweight mutex:

Plain Text

```
#include <iostream>
#include <thread>
#include <semaphore>  // C++20
#include <vector>
#include <chrono>

// Binary semaphore (max count = 1)
std::binary_semaphore mutex_sem(1);  // Initial count is 1

int shared_counter = 0;

void increment_counter(int iterations) {
    for (int i = 0; i < iterations; ++i) {
        // Acquire the semaphore
        mutex_sem.acquire();

        // Critical section
        ++shared_counter;

        // Release the semaphore
        mutex_sem.release();
    }
}

int main() {
    std::vector<std::thread> threads;
    const int num_threads = 10;
```

```
    const int iterations_per_thread = 1000;

    // Create threads
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(increment_counter, iterations_per_thread);
    }

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

    std::cout << "Expected counter value: " << num_threads *
iterations_per_thread << std::endl;
    std::cout << "Actual counter value: " << shared_counter << std::endl;

    return 0;
}
```

3.4 Latches and Barriers (C++20)

Latches and barriers are synchronization primitives that allow multiple threads to wait until a specified number of operations are completed.

3.4.1 std::latch

A latch is a single-use counter that allows threads to wait until the counter reaches zero:

Plain Text

```
#include <iostream>
#include <thread>
#include <latch>  // C++20
#include <vector>
#include <chrono>
#include <random>

void worker(std::latch& start_latch, std::latch& finish_latch, int id) {
    // Simulate preparation
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(100, 1000);
    std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));

    std::cout << "Worker " << id << " ready" << std::endl;
```

```
    // Signal that this thread is ready and wait for all threads to be ready
    start_latch.count_down_and_wait();

    std::cout << "Worker " << id << " starting main task" << std::endl;

    // Simulate main task
    std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));

    std::cout << "Worker " << id << " completed main task" << std::endl;

    // Signal that this thread has completed its task
    finish_latch.count_down();
}

int main() {
    const int num_threads = 5;

    // Latch for synchronizing the start of the main task
    std::latch start_latch(num_threads);

    // Latch for tracking completion of the main task
    std::latch finish_latch(num_threads);

    std::vector<std::thread> threads;

    std::cout << "Starting " << num_threads << " workers..." << std::endl;

    // Create worker threads
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(worker, std::ref(start_latch),
std::ref(finish_latch), i + 1);
    }

    // Wait for all workers to complete their tasks
    finish_latch.wait();
    std::cout << "All workers have completed their tasks" << std::endl;

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

    return 0;
}
```

In this example:

- `start_latch` ensures all threads start their main task at the same time
- `finish_latch` allows the main thread to wait until all worker threads have completed their tasks
- A latch can only be used once; once its count reaches zero, it cannot be reset

3.4.2 `std::barrier`

A barrier is similar to a latch but can be reused:

Plain Text

```
#include <iostream>
#include <thread>
#include <barrier>  // C++20
#include <vector>
#include <chrono>
#include <random>

void worker(std::barrier<>& sync_point, int id, int iterations) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(100, 500);

    for (int i = 0; i < iterations; ++i) {
        // Simulate work for phase 1
        std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));
        std::cout << "Worker " << id << " completed phase " << i + 1 << "a"
<< std::endl;

        // Wait for all threads to complete phase 1
        sync_point.arrive_and_wait();

        // Simulate work for phase 2
        std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));
        std::cout << "Worker " << id << " completed phase " << i + 1 << "b"
<< std::endl;

        // Wait for all threads to complete phase 2
        sync_point.arrive_and_wait();
    }
}

int main() {
    const int num_threads = 4;
```

```
    const int iterations = 3;

    // Create a barrier with a completion function
    std::barrier sync_point(num_threads, [] {
        std::cout << "All threads have reached the barrier, starting next
phase\n";
    });

    std::vector<std::thread> threads;

    // Create worker threads
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(worker, std::ref(sync_point), i + 1,
iterations);
    }

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

    return 0;
}
```

In this example:

- The barrier synchronizes threads at multiple points
- Each thread calls `arrive_and_wait()` to signal it has reached the barrier and wait for other threads
- When all threads have arrived, the completion function is executed and threads are released
- The barrier is reused for multiple synchronization points

3.4.3 `std::barrier` with Thread Dropout

Threads can permanently leave a barrier using `arrive_and_drop()`:

Plain Text

```
#include <iostream>
#include <thread>
#include <barrier>  // C++20
```

```
#include <vector>
#include <chrono>
#include <random>

void worker(std::barrier<>& sync_point, int id, int max_phases) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(100, 500);

    for (int phase = 1; phase <= max_phases; ++phase) {
        // Simulate work
        std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));

        std::cout << "Worker " << id << " completed phase " << phase <<
std::endl;

        if (phase == id) {
            // This worker has completed all its required phases
            std::cout << "Worker " << id << " is dropping out" << std::endl;
            sync_point.arrive_and_drop();
            break;
        } else {
            // Continue to the next phase
            sync_point.arrive_and_wait();
        }
    }
}

int main() {
    const int num_threads = 4;
    const int max_phases = 4;

    // Create a barrier
    std::barrier sync_point(num_threads, [] {
        std::cout << "Phase complete, moving to next phase\n";
    });

    std::vector<std::thread> threads;

    // Create worker threads
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(worker, std::ref(sync_point), i + 1,
max_phases);
    }

    // Join all threads
    for (auto& t : threads) {
        t.join();
```

```
    }

    return 0;
}
```

In this example:

- Each worker drops out after completing a specific number of phases
- `arrive_and_drop()` decrements the expected count for future synchronizations
- This is useful for algorithms where the number of participating threads decreases over time

3.5 Read-Write Locks (`std::shared_mutex` in C++17)

Read-write locks allow multiple readers or a single writer to access a resource:

Plain Text

```
#include <iostream>
#include <thread>
#include <shared_mutex>  // C++17
#include <vector>
#include <chrono>
#include <random>
#include <string>

class ThreadSafeCounter {
private:
    mutable std::shared_mutex mutex;
    int value = 0;

public:
    // Multiple threads can read the counter's value at the same time
    int get() const {
        std::shared_lock<std::shared_mutex> lock(mutex);
        return value;
    }

    // Only one thread can increment the counter at a time
    void increment() {
        std::unique_lock<std::shared_mutex> lock(mutex);
        ++value;
    }
```

```
    // Only one thread can reset the counter at a time
    void reset() {
        std::unique_lock<std::shared_mutex> lock(mutex);
        value = 0;
    }
};

void reader(ThreadSafeCounter& counter, int id) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(100, 500);

    for (int i = 0; i < 5; ++i) {
        // Read the counter value
        int value = counter.get();
        std::cout << "Reader " << id << " read value: " << value <<
std::endl;

        // Simulate some work
        std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));
    }
}

void writer(ThreadSafeCounter& counter, int id) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(200, 1000);

    for (int i = 0; i < 3; ++i) {
        // Simulate some work before writing
        std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));

        // Increment the counter
        counter.increment();
        std::cout << "Writer " << id << " incremented counter" << std::endl;
    }
}

int main() {
    ThreadSafeCounter counter;

    std::vector<std::thread> threads;

    // Create reader threads
    for (int i = 0; i < 5; ++i) {
        threads.emplace_back(reader, std::ref(counter), i + 1);
    }
```

```
    // Create writer threads
    for (int i = 0; i < 2; ++i) {
        threads.emplace_back(writer, std::ref(counter), i + 1);
    }

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

    std::cout << "Final counter value: " << counter.get() << std::endl;

    return 0;
}
```

Key points about `std::shared_mutex`:

- `std::shared_lock` allows multiple threads to acquire the lock for reading
- `std::unique_lock` provides exclusive access for writing
- This improves performance when reads are more frequent than writes
- C++14 provided `std::shared_timed_mutex`, and C++17 added the more efficient `std::shared_mutex`

3.6 Practical Example: Thread-Safe Lookup Table

Let's combine multiple synchronization primitives to create a thread-safe lookup table:

Plain Text

```
#include <iostream>
#include <thread>
#include <shared_mutex>
#include <unordered_map>
#include <string>
#include <vector>
#include <optional>
#include <chrono>
#include <random>

template<typename Key, typename Value>
```

```
class ThreadSafeLookupTable {
private:
    std::unordered_map<Key, Value> data;
    mutable std::shared_mutex mutex;

public:
    // Add or update a key-value pair
    void insert_or_update(const Key& key, const Value& value) {
        std::unique_lock<std::shared_mutex> lock(mutex);
        data[key] = value;
    }

    // Remove a key-value pair
    bool remove(const Key& key) {
        std::unique_lock<std::shared_mutex> lock(mutex);
        return data.erase(key) > 0;
    }

    // Look up a value by key
    std::optional<Value> find(const Key& key) const {
        std::shared_lock<std::shared_mutex> lock(mutex);
        auto it = data.find(key);
        if (it != data.end()) {
            return it->second;
        }
        return std::nullopt;
    }

    // Check if the table contains a key
    bool contains(const Key& key) const {
        std::shared_lock<std::shared_mutex> lock(mutex);
        return data.find(key) != data.end();
    }

    // Get the size of the table
    size_t size() const {
        std::shared_lock<std::shared_mutex> lock(mutex);
        return data.size();
    }

    // Clear the table
    void clear() {
        std::unique_lock<std::shared_mutex> lock(mutex);
        data.clear();
    }
};

void reader(ThreadSafeLookupTable<std::string, int>& table, const
```

```
std::vector<std::string>& keys, int id) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, keys.size() - 1);
    std::uniform_int_distribution<> delay(50, 200);

    for (int i = 0; i < 20; ++i) {
        std::string key = keys[dis(gen)];
        auto value = table.find(key);

        if (value) {
            std::cout << "Reader " << id << ": Found key '" << key << "' with
value " << *value << std::endl;
        } else {
            std::cout << "Reader " << id << ": Key '" << key << "' not found"
<< std::endl;
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(delay(gen)));
    }
}

void writer(ThreadSafeLookupTable<std::string, int>& table, const
std::vector<std::string>& keys, int id) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> key_dis(0, keys.size() - 1);
    std::uniform_int_distribution<> value_dis(1, 1000);
    std::uniform_int_distribution<> delay(100, 500);
    std::uniform_int_distribution<> action(0, 10);

    for (int i = 0; i < 10; ++i) {
        std::string key = keys[key_dis(gen)];
        int value = value_dis(gen);

        // 80% insert, 20% remove
        if (action(gen) < 8) {
            table.insert_or_update(key, value);
            std::cout << "Writer " << id << ": Inserted/updated key '" << key
<< "' with value " << value << std::endl;
        } else {
            bool removed = table.remove(key);
            std::cout << "Writer " << id << ": " << (removed ? "Removed" :
"Failed to remove") << " key '" << key << "'" << std::endl;
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(delay(gen)));
    }
```

```
}

int main() {
    ThreadSafeLookupTable<std::string, int> table;

    // Initialize with some data
    std::vector<std::string> keys = {"apple", "banana", "cherry", "date",
"elderberry",
                                     "fig", "grape", "honeydew", "kiwi",
"lemon"};

    for (int i = 0; i < keys.size(); ++i) {
        table.insert_or_update(keys[i], (i + 1) * 10);
    }

    std::cout << "Initial table size: " << table.size() << std::endl;

    std::vector<std::thread> threads;

    // Create reader threads
    for (int i = 0; i < 3; ++i) {
        threads.emplace_back(reader, std::ref(table), std::ref(keys), i + 1);
    }

    // Create writer threads
    for (int i = 0; i < 2; ++i) {
        threads.emplace_back(writer, std::ref(table), std::ref(keys), i + 1);
    }

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

    std::cout << "Final table size: " << table.size() << std::endl;

    // Print final table contents
    std::cout << "Final table contents:" << std::endl;
    for (const auto& key : keys) {
        auto value = table.find(key);
        if (value) {
            std::cout << "  " << key << ": " << *value << std::endl;
        }
    }

    return 0;
}
```

This example demonstrates:

- Using `std::shared_mutex` to allow multiple concurrent readers
- Using exclusive locks for writers
- Combining multiple synchronization techniques in a practical data structure
- Safe concurrent access to a shared resource

C++ Concurrency Guide

Section 4: Atomic Operations (`std::atomic`)

4.1 Introduction to Atomicity and Lock-Free Programming

Atomic operations are operations that complete in a single step relative to other threads. They provide a way to perform thread-safe operations without using mutexes, which can lead to more efficient and scalable code.

Key Concepts:

1. **Atomicity**: An operation is atomic if it appears to happen instantaneously from the perspective of other threads. There's no intermediate state visible to other threads.
2. **Lock-Free Programming**: A programming technique that allows threads to make progress without being blocked by other threads. It typically uses atomic operations instead of locks.
3. **Wait-Free Programming**: A stronger guarantee than lock-free, ensuring that every thread makes progress in a finite number of steps, regardless of the behavior of other threads.

The C++ Standard Library provides atomic types through the `<atomic>` header, which offers a set of operations that are guaranteed to be atomic.

4.2 Atomic Types (`std::atomic<T>` , `std::atomic_flag`)

4.2.1 std::atomic<T>

`std::atomic<T>` is a template class that wraps a value of type `T` and provides atomic operations on that value:

Plain Text

```
#include <iostream>
#include <atomic>
#include <thread>
#include <vector>

std::atomic<int> counter(0);  // Atomic counter initialized to 0

void increment(int iterations) {
    for (int i = 0; i < iterations; ++i) {
        // Atomic increment
        counter++;
    }
}

int main() {
    const int num_threads = 10;
    const int iterations_per_thread = 10000;

    std::vector<std::thread> threads;

    // Create threads
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(increment, iterations_per_thread);
    }

    // Join threads
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Expected counter value: " << num_threads *
iterations_per_thread << std::endl;
    std::cout << "Actual counter value: " << counter << std::endl;

    return 0;
}
```

In this example:

- We create an atomic counter initialized to 0
- Multiple threads increment the counter concurrently
- The atomic increment operation ensures that there are no race conditions
- The final value is exactly what we expect

4.2.2 `std::atomic_flag`

`std::atomic_flag` is the simplest atomic type, representing a boolean flag that can be either set or clear:

Plain Text

```
#include <iostream>
#include <atomic>
#include <thread>
#include <vector>

std::atomic_flag lock = ATOMIC_FLAG_INIT;  // Initialize to clear state

void critical_section(int id) {
    // Spin until we acquire the lock
    while (lock.test_and_set(std::memory_order_acquire)) {
        // Busy-wait (spin)
    }

    // Critical section
    std::cout << "Thread " << id << " is in the critical section" <<
std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    // Release the lock
    lock.clear(std::memory_order_release);

    std::cout << "Thread " << id << " has left the critical section" <<
std::endl;
}

int main() {
    std::vector<std::thread> threads;

    // Create threads
    for (int i = 0; i < 5; ++i) {
        threads.emplace_back(critical_section, i + 1);
```

```
    }

    // Join threads
    for (auto& t : threads) {
        t.join();
    }

    return 0;
}
```

In this example:

- We use `std::atomic_flag` to implement a simple spinlock
- `test_and_set()` atomically sets the flag and returns its previous value
- If the flag was already set, we spin (busy-wait) until it becomes clear
- `clear()` releases the lock

`std::atomic_flag` is the only atomic type guaranteed to be lock-free on all platforms.

4.2.3 Supported Atomic Types

C++ supports atomic operations for the following types:

- All integral types (bool, char, short, int, long, etc.)
- Standard integer types (std::int8_t, std::uint64_t, etc.)
- Character types (char, wchar_t, char8_t, etc.)
- Pointers (T*)
- C++20 adds support for floating-point types (float, double)
- User-defined types that are trivially copyable (with limitations)

4.3 Standard Atomic Operations

4.3.1 Load and Store Operations

```
Plain Text
```

```
#include <iostream>
#include <atomic>
#include <thread>
#include <vector>

std::atomic<bool> ready(false);
std::atomic<int> data(0);

void producer() {
    // Prepare data
    data.store(42, std::memory_order_relaxed);

    // Signal that data is ready
    ready.store(true, std::memory_order_release);
}

void consumer() {
    // Wait for data to be ready
    while (!ready.load(std::memory_order_acquire)) {
        std::this_thread::yield();  // Hint to reschedule to another thread
    }

    // Read the data
    int value = data.load(std::memory_order_relaxed);
    std::cout << "Consumed value: " << value << std::endl;
}

int main() {
    std::thread prod(producer);
    std::thread cons(consumer);

    prod.join();
    cons.join();

    return 0;
}
```

In this example:

- `store()` writes a value to the atomic object
- `load()` reads the current value of the atomic object
- Both operations take a memory ordering parameter (explained later)

4.3.2 Exchange Operation

The `exchange()` operation atomically replaces the value and returns the previous value:

Plain Text

```
#include <iostream>
#include <atomic>
#include <thread>
#include <vector>

std::atomic<int> token(0);

void worker(int id) {
    int expected_token = 0;

    // Try to take the token
    int old_token = token.exchange(id, std::memory_order_acq_rel);

    if (old_token == 0) {
        // Successfully acquired the token
        std::cout << "Thread " << id << " acquired the token" << std::endl;

        // Simulate work
        std::this_thread::sleep_for(std::chrono::milliseconds(100));

        // Release the token
        token.store(0, std::memory_order_release);
        std::cout << "Thread " << id << " released the token" << std::endl;
    } else {
        // Failed to acquire the token
        std::cout << "Thread " << id << " could not acquire the token (held
by " << old_token << ")" << std::endl;
    }
}

int main() {
    std::vector<std::thread> threads;

    // Create threads
    for (int i = 0; i < 5; ++i) {
        threads.emplace_back(worker, i + 1);
    }

    // Join threads
    for (auto& t : threads) {
        t.join();
```

```
    }

    return 0;
}
```

In this example:

- `exchange()` atomically replaces the value with a new one and returns the old value
- We use it to implement a simple token-passing mechanism

4.3.3 Compare-Exchange Operations

Compare-exchange operations are the foundation of many lock-free algorithms. They attempt to update a value only if it currently equals an expected value:

Plain Text

```
#include <iostream>
#include <atomic>
#include <thread>
#include <vector>

std::atomic<int> shared_data(0);

void update_if_equal(int expected, int desired) {
    bool success = shared_data.compare_exchange_strong(expected, desired);

    if (success) {
        std::cout << "Successfully updated from " << expected << " to " <<
desired << std::endl;
    } else {
        std::cout << "Update failed. Expected " << expected << " but got " <<
expected << std::endl;
    }
}

int main() {
    // Initial value is 0

    // This should succeed (0 -> 10)
    update_if_equal(0, 10);

    // This should fail (expected 0, but actual is now 10)
    update_if_equal(0, 20);
```

```
    // This should succeed (10 -> 30)
    update_if_equal(10, 30);

    std::cout << "Final value: " << shared_data.load() << std::endl;

    return 0;
}
```

C++ provides two compare-exchange operations:

1. `compare_exchange_strong` : Guarantees to succeed if the current value equals the expected value.
2. `compare_exchange_weak` : May fail spuriously (return false even when the current value equals the expected value), but can be more efficient on some platforms.

When using `compare_exchange_weak` , it's common to use it in a loop:

Plain Text

```
bool expected = false;
while (!atomic_flag.compare_exchange_weak(expected, true) && !expected) {
    // If we get here, either:
    // 1. The exchange failed because the flag was already set (expected is
now true)
    // 2. The exchange failed spuriously (expected is still false)
    expected = false;  // Reset for next attempt
}
```

4.4 Fetch-and-Modify Operations

Atomic types provide several fetch-and-modify operations that perform an operation and return the previous value:

Plain Text

```
#include <iostream>
#include <atomic>
#include <thread>
#include <vector>

std::atomic<int> counter(0);
```

```
void worker(int id) {
    // Increment and get previous value
    int prev_value = counter.fetch_add(1);
    std::cout << "Thread " << id << " incremented counter from " <<
prev_value << " to " << prev_value + 1 << std::endl;

    // Add 10 and get previous value
    prev_value = counter.fetch_add(10);
    std::cout << "Thread " << id << " added 10 to counter from " <<
prev_value << " to " << prev_value + 10 << std::endl;

    // Subtract 5 and get previous value
    prev_value = counter.fetch_sub(5);
    std::cout << "Thread " << id << " subtracted 5 from counter from " <<
prev_value << " to " << prev_value - 5 << std::endl;

    // Bitwise AND with 0xFFFF and get previous value
    prev_value = counter.fetch_and(0xFFFF);
    std::cout << "Thread " << id << " performed AND operation on counter" <<
std::endl;

    // Bitwise OR with 0x1000 and get previous value
    prev_value = counter.fetch_or(0x1000);
    std::cout << "Thread " << id << " performed OR operation on counter" <<
std::endl;

    // Bitwise XOR with 0xFF and get previous value
    prev_value = counter.fetch_xor(0xFF);
    std::cout << "Thread " << id << " performed XOR operation on counter" <<
std::endl;
}

int main() {
    std::vector<std::thread> threads;

    // Create threads
    for (int i = 0; i < 3; ++i) {
        threads.emplace_back(worker, i + 1);
    }

    // Join threads
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Final counter value: " << counter << std::endl;
```

```
    return 0;
}
```

Available fetch-and-modify operations:

- `fetch_add()` : Add a value and return the previous value
- `fetch_sub()` : Subtract a value and return the previous value
- `fetch_and()` : Perform bitwise AND and return the previous value
- `fetch_or()` : Perform bitwise OR and return the previous value
- `fetch_xor()` : Perform bitwise XOR and return the previous value

4.5 Memory Ordering Models

Memory ordering specifies how memory accesses (including regular, non-atomic memory accesses) are ordered around an atomic operation. C++ provides several memory ordering options:

4.5.1 Sequential Consistency (`memory_order_seq_cst`)

This is the default memory ordering and provides the strongest guarantees:

Plain Text

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

std::atomic<bool> x(false);
std::atomic<bool> y(false);
std::atomic<int> z(0);

void write_x() {
    x.store(true, std::memory_order_seq_cst);
}

void write_y() {
    y.store(true, std::memory_order_seq_cst);
}
```

```
void read_x_then_y() {
    while (!x.load(std::memory_order_seq_cst));
    if (y.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

void read_y_then_x() {
    while (!y.load(std::memory_order_seq_cst));
    if (x.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

int main() {
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);

    a.join();
    b.join();
    c.join();
    d.join();

    std::cout << "z = " << z << std::endl;
    // With sequential consistency, z will always be 2

    return 0;
}
```

With sequential consistency:

- All threads see the same order of operations
- Operations cannot be reordered
- This is the most intuitive but also the most expensive in terms of performance

4.5.2 Acquire-Release Semantics (`memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`)

These provide synchronization between threads:

Plain Text

```
#include <iostream>
#include <atomic>
#include <thread>
#include <cassert>

std::atomic<bool> ready(false);
int data = 0;  // Non-atomic data

void producer() {
    // Prepare data
    data = 42;

    // Release fence ensures all previous writes are visible to other threads
    // that acquire this atomic variable
    ready.store(true, std::memory_order_release);
}

void consumer() {
    // Acquire fence ensures all subsequent reads see writes from the
producer
    while (!ready.load(std::memory_order_acquire)) {
        std::this_thread::yield();
    }

    // At this point, we're guaranteed to see data = 42
    assert(data == 42);  // This will never fail
    std::cout << "Data: " << data << std::endl;
}

int main() {
    std::thread prod(producer);
    std::thread cons(consumer);

    prod.join();
    cons.join();

    return 0;
}
```

- `memory_order_release`: All memory operations before the store are visible to other threads that acquire the same atomic variable
- `memory_order_acquire`: All memory operations after the load will see the effects of memory operations that happened before the release

- `memory_order_acq_rel` : Combines acquire and release semantics for read-modify-write operations

4.5.3 Relaxed Ordering (`memory_order_relaxed`)

This provides the weakest guarantees but the best performance:

Plain Text

```
#include <iostream>
#include <atomic>
#include <thread>
#include <vector>

std::atomic<int> counter(0);

void increment(int iterations) {
    for (int i = 0; i < iterations; ++i) {
        // Relaxed ordering - only guarantees atomicity, not ordering
        counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    const int num_threads = 10;
    const int iterations_per_thread = 10000;

    std::vector<std::thread> threads;

    // Create threads
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(increment, iterations_per_thread);
    }

    // Join threads
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Final counter value: " << counter << std::endl;

    return 0;
}
```

With relaxed ordering:

- Only the atomicity of the operation is guaranteed
- No synchronization or ordering constraints are imposed
- This is useful when you only need atomicity but not ordering, such as for a simple counter

4.5.4 Consume Ordering (`memory_order_consume`)

This is a specialized form of acquire semantics that's rarely used directly:

Plain Text

```
#include <iostream>
#include <atomic>
#include <thread>
#include <string>

struct X {
    int a;
    std::string b;
};

std::atomic<X*> p;
int value;

void producer() {
    X* x = new X{42, "Hello"};
    value = 24;  // This may or may not be visible when p is read
    p.store(x, std::memory_order_release);
}

void consumer() {
    X* x;
    while (!(x = p.load(std::memory_order_consume))) {
        std::this_thread::yield();
    }

    // x->a and x->b are guaranteed to be visible correctly
    // because they are data-dependent on the loaded value of p
    std::cout << "x->a: " << x->a << std::endl;
    std::cout << "x->b: " << x->b << std::endl;

    // value may or may not be visible correctly
    // because it's not data-dependent on the loaded value of p
    std::cout << "value: " << value << std::endl;
```

```
    delete x;
}

int main() {
    std::thread prod(producer);
    std::thread cons(consumer);

    prod.join();
    cons.join();

    return 0;
}
```

Note: `memory_order_consume` is rarely used directly because its implementation is complex and not well-supported across all platforms. The C++ Standards Committee has recommended against using it directly.

4.6 Practical Example: Lock-Free Stack

Let's implement a simple lock-free stack using atomic operations:

Plain Text

```
#include <iostream>
#include <atomic>
#include <memory>
#include <thread>
#include <vector>

template<typename T>
class LockFreeStack {
private:
    struct Node {
        T data;
        Node* next;

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

    std::atomic<Node*> head;

public:
    LockFreeStack() : head(nullptr) {}
```

```
    ~LockFreeStack() {
        // Clean up remaining nodes
        Node* current = head.load();
        while (current) {
            Node* next = current->next;
            delete current;
            current = next;
        }
    }

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

        // Set new_node->next to current head
        new_node->next = head.load(std::memory_order_relaxed);

        // Try to set head to new_node
        while (!head.compare_exchange_weak(new_node->next, new_node,
                                           std::memory_order_release,
                                           std::memory_order_relaxed)) {
            // If CAS fails, new_node->next is updated to the current head
            // So we just retry
        }
    }

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

        // If stack is empty
        if (!old_head) {
            return false;
        }

        // Try to update head to next node
        while (!head.compare_exchange_weak(old_head, old_head->next,
                                           std::memory_order_acquire,
                                           std::memory_order_relaxed)) {
            // If CAS fails, old_head is updated to the current head
            // If the stack became empty, return false
            if (!old_head) {
                return false;
            }
        }

        // Successfully popped the node
        result = old_head->data;

        // In a production implementation, we would use a memory reclamation
```

```
        // strategy like hazard pointers or reference counting instead of
        // deleting immediately
        delete old_head;

        return true;
    }

    bool empty() const {
        return head.load(std::memory_order_relaxed) == nullptr;
    }
};

void producer(LockFreeStack<int>& stack, int start, int count) {
    for (int i = 0; i < count; ++i) {
        stack.push(start + i);
        std::cout << "Pushed: " << start + i << std::endl;
    }
}

void consumer(LockFreeStack<int>& stack, int count) {
    int value;
    int popped = 0;

    while (popped < count) {
        if (stack.pop(value)) {
            std::cout << "Popped: " << value << std::endl;
            popped++;
        } else {
            std::this_thread::yield();
        }
    }
}

int main() {
    LockFreeStack<int> stack;

    // Test with single-threaded push and pop
    for (int i = 0; i < 5; ++i) {
        stack.push(i);
    }

    int value;
    while (stack.pop(value)) {
        std::cout << "Popped: " << value << std::endl;
    }

    // Test with multiple threads
    const int items_per_producer = 1000;
```

```
    const int num_producers = 3;
    const int num_consumers = 2;
    const int total_items = items_per_producer * num_producers;
    const int items_per_consumer = total_items / num_consumers;

    std::vector<std::thread> threads;

    // Create producer threads
    for (int i = 0; i < num_producers; ++i) {
        threads.emplace_back(producer, std::ref(stack), i *
items_per_producer, items_per_producer);
    }

    // Create consumer threads
    for (int i = 0; i < num_consumers; ++i) {
        threads.emplace_back(consumer, std::ref(stack), items_per_consumer);
    }

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

    // Check if stack is empty
    std::cout << "Stack is " << (stack.empty() ? "empty" : "not empty") <<
std::endl;

    return 0;
}
```

This example demonstrates:

- Using atomic operations to implement a lock-free data structure
- Using compare-exchange operations for safe updates
- Proper memory ordering for thread synchronization

Note: This is a simplified example. A production-ready lock-free stack would need to address the ABA problem and use a proper memory reclamation strategy.

4.7 Atomic Smart Pointers (C++20)

C++20 introduced atomic smart pointers, which provide atomic operations on `std::shared_ptr` and `std::weak_ptr`:

Plain Text

```
#include <iostream>
#include <atomic>
#include <memory>
#include <thread>
#include <vector>

struct Data {
    int value;

    Data(int v) : value(v) {}
    ~Data() {
        std::cout << "Destroying Data with value " << value << std::endl;
    }
};

std::atomic<std::shared_ptr<Data>> global_data;

void producer() {
    // Create a new shared_ptr
    auto new_data = std::make_shared<Data>(42);

    // Store it atomically
    global_data.store(new_data);

    std::cout << "Producer stored new data" << std::endl;
}

void consumer() {
    // Load the shared_ptr atomically
    std::shared_ptr<Data> local_data = global_data.load();

    if (local_data) {
        std::cout << "Consumer read value: " << local_data->value <<
std::endl;
    } else {
        std::cout << "Consumer found null pointer" << std::endl;
    }
}

int main() {
    // Initialize with nullptr
    global_data.store(nullptr);

    // First consumer should find nullptr
    std::thread cons1(consumer);
    cons1.join();
```

```
    // Producer creates and stores data
    std::thread prod(producer);
    prod.join();

    // Second consumer should find the data
    std::thread cons2(consumer);
    cons2.join();

    // Clear the global data
    global_data.store(nullptr);

    return 0;
}
```

Atomic smart pointers provide:

- Thread-safe access to shared resources
- Automatic memory management
- Atomic operations like load, store, exchange, and compare-exchange

4.8 Atomic Reference Wrapper (C++20)

C++20 also introduced `std::atomic_ref`, which provides atomic operations on non-atomic objects:

Plain Text

```
#include <iostream>
#include <atomic>
#include <thread>
#include <vector>

int main() {
    int value = 0;

    // Create an atomic reference to value
    std::atomic_ref<int> atomic_value(value);

    std::vector<std::thread> threads;

    // Create threads that increment the value
    for (int i = 0; i < 10; ++i) {
```

```
        threads.emplace_back([&atomic_value]() {
            for (int j = 0; j < 1000; ++j) {
                atomic_value.fetch_add(1, std::memory_order_relaxed);
            }
        });
    }

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

    std::cout << "Final value: " << value << std::endl;
    std::cout << "Atomic value: " << atomic_value.load() << std::endl;

    return 0;
}
```

`std::atomic_ref` is useful when:

- You want to perform atomic operations on existing objects
- You want to avoid the overhead of creating atomic objects
- You need to switch between atomic and non-atomic access to the same object

4.9 Best Practices for Atomic Operations

1. **Use the Right Memory Ordering**:
    - Use `memory_order_seq_cst` when in doubt (it's the default)
    - Use `memory_order_relaxed` for simple counters and flags when no synchronization is needed
    - Use `memory_order_acquire` / `memory_order_release` pairs for producer-consumer patterns
2. **Avoid the ABA Problem**:
    - The ABA problem occurs when a value changes from A to B and back to A, which can cause issues in lock-free algorithms

- Use techniques like hazard pointers, reference counting, or tagged pointers to avoid it

3. **Be Careful with User-Defined Types**:
   - `std::atomic<T>` works best with simple types
   - For complex types, consider using atomic operations on pointers to those types
4. **Test Thoroughly**:
   - Lock-free code is notoriously difficult to get right
   - Use tools like thread sanitizers to detect race conditions
   - Test on different platforms and architectures
5. **Consider Performance Implications**:
   - Atomic operations can be expensive, especially with stronger memory ordering
   - Measure and compare different approaches
   - Sometimes a simple mutex can be more efficient than complex lock-free code
6. **Document Memory Ordering Requirements**:
   - Clearly document the memory ordering requirements of your functions
   - Explain the synchronization guarantees provided by your code

C++ Concurrency Guide

Section 5: Futures and Promises

5.1 Introduction to Asynchronous Operations

Futures and promises provide a high-level mechanism for asynchronous programming in C++. They allow you to retrieve results from operations that might not have completed yet, without blocking the calling thread until the result is needed.

Key Components:

1. **Promise (`std::promise`)**: An object that "promises" to provide a result at some point in the future.
2. **Future (`std::future`)**: An object that refers to a result that will be available later.
3. **Packaged Task (`std::packaged_task`)**: A wrapper for a callable that can be executed asynchronously, with its result stored in a future.
4. **Async (`std::async`)**: A high-level function to launch asynchronous tasks and get futures for their results.

These components work together to provide a clean and safe way to handle asynchronous operations without dealing directly with threads and synchronization primitives.

5.2 Retrieving Results from Asynchronous Tasks

5.2.1 Basic Usage of `std::future`

Plain Text

```
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

// A function that performs a computation and returns a result
int compute_value() {
    std::cout << "Computing value..." << std::endl;
    // Simulate a time-consuming operation
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "Computation complete!" << std::endl;
    return 42;
}

int main() {
    // Launch the computation asynchronously
    std::future<int> result_future = std::async(compute_value);

    std::cout << "Main thread continues execution..." << std::endl;

    // Do other work in the main thread
    std::this_thread::sleep_for(std::chrono::seconds(1));
```

```
    std::cout << "Main thread did some work" << std::endl;

    // Get the result - this will block if the computation isn't finished yet
    int result = result_future.get();
    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

In this example:

- `std::async` launches `compute_value()` asynchronously and returns a future
- The main thread continues execution while the computation runs in the background
- When we call `get()` on the future, it blocks until the result is available
- Once the result is retrieved, the future becomes invalid (you can only call `get()` once)

5.2.2 Waiting for Futures

`std::future` provides several ways to wait for a result:

Plain Text

```
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

int compute_value(int input) {
    std::cout << "Computing value for input " << input << "..." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(input));
    return input * 10;
}

int main() {
    // Launch three computations with different durations
    std::future<int> future1 = std::async(compute_value, 1);
    std::future<int> future2 = std::async(compute_value, 2);
    std::future<int> future3 = std::async(compute_value, 3);

    // Wait for future1 to be ready (blocks until the result is available)
    future1.wait();
    std::cout << "Future 1 result: " << future1.get() << std::endl;
```

```
    // Wait for future2 with a timeout
    if (future2.wait_for(std::chrono::milliseconds(500)) ==
std::future_status::ready) {
        std::cout << "Future 2 is ready" << std::endl;
    } else {
        std::cout << "Future 2 is not ready yet, but we'll wait for it" <<
std::endl;
        future2.wait();
    }
    std::cout << "Future 2 result: " << future2.get() << std::endl;

    // Wait for future3 until a specific time point
    auto deadline = std::chrono::system_clock::now() +
std::chrono::seconds(2);
    if (future3.wait_until(deadline) == std::future_status::ready) {
        std::cout << "Future 3 is ready" << std::endl;
    } else {
        std::cout << "Future 3 is not ready yet, but we'll wait for it" <<
std::endl;
        future3.wait();
    }
    std::cout << "Future 3 result: " << future3.get() << std::endl;

    return 0;
}
```

This example demonstrates three ways to wait for futures:

- `wait()` : Blocks until the result is available
- `wait_for(duration)` : Waits for a specified duration, returns a status
- `wait_until(time_point)` : Waits until a specified time point, returns a status

The status can be:

- `std::future_status::ready` : The result is ready
- `std::future_status::timeout` : The wait timed out
- `std::future_status::deferred` : The task hasn't started yet (for deferred execution)

5.3 Setting Results or Exceptions (`std::promise`)

`std::promise` allows you to set a value or exception that will be retrieved through an associated future:

Plain Text

```
#include <iostream>
#include <future>
#include <thread>
#include <stdexcept>

void producer(std::promise<int> promise) {
    try {
        // Simulate work
        std::cout << "Producer thread working..." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));

        // Set the promised value
        promise.set_value(42);
    } catch (...) {
        // If an exception occurs, store it in the promise
        promise.set_exception(std::current_exception());
    }
}

void producer_with_error(std::promise<int> promise) {
    try {
        // Simulate work
        std::cout << "Error producer thread working..." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));

        // Simulate an error
        throw std::runtime_error("Something went wrong!");

        // This line will never be reached
        promise.set_value(0);
    } catch (...) {
        // Store the exception in the promise
        promise.set_exception(std::current_exception());
    }
}

int main() {
    // Create a promise and get its future
    std::promise<int> promise1;
    std::future<int> future1 = promise1.get_future();

    // Launch a thread that will fulfill the promise
```

```
    std::thread t1(producer, std::move(promise1));

    // Create another promise for the error case
    std::promise<int> promise2;
    std::future<int> future2 = promise2.get_future();

    // Launch a thread that will set an exception
    std::thread t2(producer_with_error, std::move(promise2));

    // Wait for and get the result from the first future
    try {
        int result = future1.get();
        std::cout << "Got result: " << result << std::endl;
    } catch (const std::exception& e) {
        std::cout << "Exception from future1: " << e.what() << std::endl;
    }

    // Wait for and try to get the result from the second future
    try {
        int result = future2.get();
        std::cout << "Got result: " << result << std::endl;
    } catch (const std::exception& e) {
        std::cout << "Exception from future2: " << e.what() << std::endl;
    }

    t1.join();
    t2.join();

    return 0;
}
```

Key points about `std::promise`:

- It's the "producer" end of the promise-future pair
- You can set a value with `set_value()` or an exception with `set_exception()`
- Each promise is associated with exactly one future
- A promise can only be fulfilled once
- Promises are typically moved into threads, as they're not copyable

5.4 Wrapping Callables (`std::packaged_task`)

`std::packaged_task` wraps a callable object (function, lambda, etc.) so it can be executed asynchronously with its result stored in a future:

Plain Text

```
#include <iostream>
#include <future>
#include <thread>
#include <vector>
#include <cmath>

// Function to check if a number is prime
bool is_prime(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;

    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) return false;
    }

    return true;
}

int count_primes(int start, int end) {
    int count = 0;
    for (int i = start; i <= end; ++i) {
        if (is_prime(i)) {
            count++;
        }
    }
    return count;
}

int main() {
    const int num_tasks = 4;
    const int range_size = 250000;

    std::vector<std::packaged_task<int(int, int)>> tasks;
    std::vector<std::future<int>> futures;

    // Create tasks
    for (int i = 0; i < num_tasks; ++i) {
        int start = i * range_size + 1;
        int end = (i + 1) * range_size;

        // Create a packaged task that will count primes in a range
```

```
        std::packaged_task<int(int, int)> task(count_primes);

        // Get the future associated with the task
        futures.push_back(task.get_future());

        // Move the task into our vector
        tasks.push_back(std::move(task));
    }

    // Launch tasks on separate threads
    std::vector<std::thread> threads;
    for (int i = 0; i < num_tasks; ++i) {
        int start = i * range_size + 1;
        int end = (i + 1) * range_size;

        threads.emplace_back([&tasks, i, start, end]() {
            tasks[i](start, end);
        });
    }

    // Wait for all threads to complete
    for (auto& thread : threads) {
        thread.join();
    }

    // Collect and sum the results
    int total_primes = 0;
    for (auto& future : futures) {
        total_primes += future.get();
    }

    std::cout << "Total primes found: " << total_primes << std::endl;

    return 0;
}
```

Key points about `std::packaged_task`:

- It wraps a callable object and allows you to get a future for its result
- It doesn't execute the callable automatically; you need to call it explicitly
- It's useful when you want to control when and where a task is executed
- It's typically used with thread pools or task systems

5.5 High-Level Asynchronous Launch (`std::async`)

`std::async` is a high-level function that launches a function asynchronously and returns a future for its result:

Plain Text

```
#include <iostream>
#include <future>
#include <vector>
#include <string>
#include <algorithm>

// Function to count occurrences of a character in a string
int count_char(const std::string& str, char ch) {
    return std::count(str.begin(), str.end(), ch);
}

int main() {
    std::string text = "The quick brown fox jumps over the lazy dog";
    std::vector<char> chars_to_count = {'a', 'e', 'i', 'o', 'u'};

    // Launch tasks with different launch policies
    std::vector<std::future<int>> futures;

    // Default launch policy (implementation-defined)
    futures.push_back(std::async(count_char, text, 'a'));

    // Deferred execution (lazy evaluation)
    futures.push_back(std::async(std::launch::deferred, count_char, text,
'e'));

    // Asynchronous execution (eager evaluation)
    futures.push_back(std::async(std::launch::async, count_char, text, 'i'));

    // Either deferred or asynchronous (implementation's choice)
    futures.push_back(std::async(std::launch::deferred | std::launch::async,
count_char, text, 'o'));

    // One more with default policy
    futures.push_back(std::async(count_char, text, 'u'));

    // Collect and print results
    int total_vowels = 0;
    for (size_t i = 0; i < futures.size(); ++i) {
        int count = futures[i].get();
        std::cout << "Count of '" << chars_to_count[i] << "': " << count <<
```

```
std::endl;
        total_vowels += count;
    }

    std::cout << "Total vowels: " << total_vowels << std::endl;

    return 0;
}
```

`std::async` supports different launch policies:

- `std::launch::async` : The task is executed on a separate thread
- `std::launch::deferred` : The task is executed when `get()` or `wait()` is called on the future
- `std::launch::async | std::launch::deferred` : The implementation chooses (this is the default)

Benefits of `std::async` :

- Simple and concise way to run tasks asynchronously
- Automatic thread management
- Exception propagation from the task to the future
- Flexible launch policies

5.6 Shared Futures (`std::shared_future`)

A `std::future` can only be accessed by a single thread, and `get()` can only be called once. `std::shared_future` allows multiple threads to wait for and access the same result:

Plain Text

```
#include <iostream>
#include <future>
#include <thread>
#include <vector>

void reader(std::shared_future<int> future, int id) {
    try {
        // Wait for the result
        int result = future.get();  // Multiple threads can call get()
        std::cout << "Reader " << id << " got result: " << result <<
```

```
std::endl;
    } catch (const std::exception& e) {
        std::cout << "Reader " << id << " caught exception: " << e.what() <<
std::endl;
    }
}

int main() {
    // Create a promise
    std::promise<int> promise;

    // Get a future and convert it to a shared_future
    std::shared_future<int> shared_future = promise.get_future().share();

    // Create multiple reader threads
    std::vector<std::thread> threads;
    for (int i = 0; i < 5; ++i) {
        threads.emplace_back(reader, shared_future, i + 1);
    }

    // Simulate work
    std::this_thread::sleep_for(std::chrono::seconds(1));

    // Set the promised value
    std::cout << "Setting value..." << std::endl;
    promise.set_value(42);

    // Wait for all readers to finish
    for (auto& thread : threads) {
        thread.join();
    }

    return 0;
}
```

Key points about `std::shared_future`:

- Multiple threads can access the same future
- `get()` can be called multiple times (it doesn't invalidate the future)
- You can create a `shared_future` from a `future` using the `share()` method
- `shared_future` is copyable (unlike `future`, which is move-only)

5.7 Practical Example: Parallel Image Processing

Let's use futures and promises to implement a parallel image processing pipeline:

Plain Text

```
#include <iostream>
#include <future>
#include <vector>
#include <string>
#include <chrono>
#include <random>
#include <algorithm>

// Simulated image class
class Image {
private:
    std::vector<int> pixels;
    int width;
    int height;

public:
    Image(int w, int h) : width(w), height(h), pixels(w * h, 0) {}

    void randomize() {
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<> dis(0, 255);

        for (auto& pixel : pixels) {
            pixel = dis(gen);
        }
    }

    int get_pixel(int x, int y) const {
        return pixels[y * width + x];
    }

    void set_pixel(int x, int y, int value) {
        pixels[y * width + x] = value;
    }

    int get_width() const { return width; }
    int get_height() const { return height; }

    // Simulated image loading
    static std::future<Image> load_async(const std::string& filename) {
        return std::async(std::launch::async, [filename]() {
            std::cout << "Loading image: " << filename << std::endl;
```

```
            // Simulate loading time
            std::this_thread::sleep_for(std::chrono::milliseconds(500));

            // Create a random image
            Image img(800, 600);
            img.randomize();

            std::cout << "Image loaded: " << filename << std::endl;
            return img;
        });
    }
};

// Image processing functions
Image apply_blur(const Image& input) {
    std::cout << "Applying blur filter..." << std::endl;

    // Simulate processing time
    std::this_thread::sleep_for(std::chrono::milliseconds(300));

    // Create output image with same dimensions
    Image output(input.get_width(), input.get_height());

    // Simulate blur by averaging pixels (in a real application, we'd do
actual blurring)
    for (int y = 1; y < input.get_height() - 1; ++y) {
        for (int x = 1; x < input.get_width() - 1; ++x) {
            int sum = 0;
            for (int dy = -1; dy <= 1; ++dy) {
                for (int dx = -1; dx <= 1; ++dx) {
                    sum += input.get_pixel(x + dx, y + dy);
                }
            }
            output.set_pixel(x, y, sum / 9);
        }
    }

    std::cout << "Blur filter applied" << std::endl;
    return output;
}

Image apply_threshold(const Image& input, int threshold) {
    std::cout << "Applying threshold filter..." << std::endl;

    // Simulate processing time
    std::this_thread::sleep_for(std::chrono::milliseconds(200));

    // Create output image with same dimensions
```

```
    Image output(input.get_width(), input.get_height());

    // Apply threshold
    for (int y = 0; y < input.get_height(); ++y) {
        for (int x = 0; x < input.get_width(); ++x) {
            int pixel = input.get_pixel(x, y);
            output.set_pixel(x, y, pixel > threshold ? 255 : 0);
        }
    }

    std::cout << "Threshold filter applied" << std::endl;
    return output;
}

// Simulated image saving
std::future<void> save_async(const Image& image, const std::string& filename)
{
    return std::async(std::launch::async, [&image, filename]() {
        std::cout << "Saving image: " << filename << std::endl;

        // Simulate saving time
        std::this_thread::sleep_for(std::chrono::milliseconds(400));

        std::cout << "Image saved: " << filename << std::endl;
    });
}

int main() {
    auto start_time = std::chrono::high_resolution_clock::now();

    // Start loading images asynchronously
    auto future_image1 = Image::load_async("image1.jpg");
    auto future_image2 = Image::load_async("image2.jpg");

    std::cout << "Main thread: Images loading in background..." << std::endl;

    // Do some other work in the main thread
    std::cout << "Main thread: Doing other work..." << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));

    // Process the first image
    std::cout << "Main thread: Waiting for image1..." << std::endl;
    Image image1 = future_image1.get();

    // Apply filters to image1
    auto future_blurred = std::async(std::launch::async, apply_blur,
std::ref(image1));
```

```
    // Process the second image while the first one is being blurred
    std::cout << "Main thread: Waiting for image2..." << std::endl;
    Image image2 = future_image2.get();

    // Apply threshold to image2
    auto future_thresholded = std::async(std::launch::async, apply_threshold,
std::ref(image2), 128);

    // Wait for both processing tasks to complete
    Image blurred_image = future_blurred.get();
    Image thresholded_image = future_thresholded.get();

    // Save the processed images asynchronously
    auto save_future1 = save_async(blurred_image, "blurred.jpg");
    auto save_future2 = save_async(thresholded_image, "thresholded.jpg");

    // Wait for both saves to complete
    save_future1.get();
    save_future2.get();

    auto end_time = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>
(end_time - start_time);

    std::cout << "All operations completed in " << duration.count() << " ms"
<< std::endl;

    return 0;
}
```

This example demonstrates:

- Using `std::async` to load images asynchronously
- Processing images in parallel
- Chaining asynchronous operations
- Coordinating multiple asynchronous tasks
- Measuring the performance benefits of parallelism

5.8 Best Practices for Futures and Promises

1. **Use `std::async` for Simple Tasks**:

- For straightforward asynchronous operations, `std::async` provides a clean and simple interface
- Be aware of the launch policy and its implications

2. **Handle Exceptions**:
   - Exceptions thrown in asynchronous tasks are stored in the future
   - Always use try-catch when calling `get()` on a future
3. **Be Careful with References**:
   - Ensure that referenced objects outlive the asynchronous task
   - Use `std::ref` or `std::cref` to pass references to async functions
4. **Avoid Blocking When Possible**:
   - Use `wait_for()` or `wait_until()` with timeouts instead of blocking indefinitely
   - Consider polling futures periodically instead of blocking
5. **Use `std::shared_future` for Multiple Observers**:
   - When multiple threads need to access the same result
   - Remember that `shared_future` is copyable, while `future` is move-only
6. **Be Aware of Thread Creation Overhead**:
   - `std::async` with `std::launch::async` creates a new thread, which has overhead
   - For many small tasks, consider using a thread pool instead
7. **Understand the Lifetime of Futures**:
   - A future returned by `std::async` will block in its destructor if the task hasn't completed
   - This can lead to unexpected blocking if you don't store the future

C++ Concurrency Guide

Section 6: Higher-Level Concurrency Patterns

While the fundamental primitives (threads, mutexes, atomics, futures) are essential, building complex concurrent applications often involves using higher-level patterns that structure the interaction between threads and tasks.

6.1 Thread Pools

A thread pool is a collection of pre-started worker threads that can execute tasks submitted to the pool. This avoids the overhead of creating and destroying threads for each task and limits the number of concurrent threads.

6.1.1 Concept and Benefits

- **Resource Management**: Limits the number of active threads, preventing system overload.
- **Performance**: Reuses existing threads, avoiding the cost of thread creation and destruction.
- **Responsiveness**: Tasks can be submitted quickly without waiting for thread creation.

6.1.2 Simple Thread Pool Implementation

Plain Text

```
#include <iostream>
#include <vector>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <future>
#include <stdexcept>

class ThreadPool {
public:
```

```
    ThreadPool(size_t num_threads);
    ~ThreadPool();

    // Add a new task to the pool
    template<class F, class... Args>
    auto enqueue(F&& f, Args&&... args)
        -> std::future<typename std::invoke_result<F, Args...>::type>;

private:
    // Worker threads
    std::vector<std::thread> workers;
    // Task queue
    std::queue<std::function<void()>> tasks;

    // Synchronization
    std::mutex queue_mutex;
    std::condition_variable condition;
    bool stop = false;
};

// Constructor: create and start worker threads
ThreadPool::ThreadPool(size_t num_threads)
    : stop(false)
{
    for(size_t i = 0; i < num_threads; ++i)
        workers.emplace_back(
            [this] {
                while(true) {
                    std::function<void()> task;
                    {
                        std::unique_lock<std::mutex> lock(this->queue_mutex);
                        // Wait until there is a task or the pool is stopped
                        this->condition.wait(lock,
                            [this]{ return this->stop || !this-
>tasks.empty(); });

                        // If stopped and no tasks left, exit the thread
                        if(this->stop && this->tasks.empty())
                            return;

                        // Get the next task
                        task = std::move(this->tasks.front());
                        this->tasks.pop();
                    }
                    // Execute the task
                    task();
                }
            }
```

```
        );
}

// Destructor: stop and join all worker threads
ThreadPool::~ThreadPool() {
    {
        std::unique_lock<std::mutex> lock(queue_mutex);
        stop = true;
    }
    // Notify all waiting threads
    condition.notify_all();
    // Join all worker threads
    for(std::thread &worker: workers)
        worker.join();
}

// Enqueue a task and return a future for its result
template<class F, class... Args>
auto ThreadPool::enqueue(F&& f, Args&&... args)
    -> std::future<typename std::invoke_result<F, Args...>::type>
{
    using return_type = typename std::invoke_result<F, Args...>::type;

    // Create a packaged_task to wrap the function and its arguments
    auto task = std::make_shared< std::packaged_task<return_type()> >(
            std::bind(std::forward<F>(f), std::forward<Args>(args)...)
        );

    // Get the future associated with the task
    std::future<return_type> res = task->get_future();
    {
        std::unique_lock<std::mutex> lock(queue_mutex);

        // Don't allow enqueueing after stopping the pool
        if(stop)
            throw std::runtime_error("enqueue on stopped ThreadPool");

        // Add the task to the queue
        tasks.emplace([task](){ (*task)(); });
    }
    // Notify one waiting worker thread
    condition.notify_one();
    return res;
}

// Example usage
int multiply(int a, int b) {
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
```

```
    return a * b;
}

void print_message(const std::string& msg) {
    std::cout << msg << std::endl;
}

int main() {
    // Create a thread pool with 4 worker threads
    ThreadPool pool(4);

    // Enqueue tasks
    std::vector< std::future<int> > results;
    for(int i = 0; i < 8; ++i) {
        results.emplace_back(
            pool.enqueue(multiply, i, i + 1)
        );
    }

    // Enqueue a void task
    auto void_future = pool.enqueue(print_message, "Task with no return value
executed");

    // Retrieve results
    for(auto && result: results) {
        std::cout << "Result: " << result.get() << std::endl;
    }

    // Wait for the void task (optional)
    void_future.get();

    std::cout << "All tasks completed" << std::endl;
    // ThreadPool destructor will automatically stop and join threads
    return 0;
}
```

This implementation demonstrates:

- A fixed number of worker threads waiting on a condition variable
- A queue to hold tasks (`std::function<void()>`).
- An `enqueue` method that takes a function and arguments, wraps them in a `packaged_task` , adds the task to the queue, and returns a future.
- Proper synchronization using mutexes and condition variables.

- Graceful shutdown in the destructor.

6.2 Task Queues

Task queues are fundamental components of many concurrent systems, including thread pools and active objects. They allow threads to communicate by passing tasks or messages.

6.2.1 Blocking vs. Non-Blocking Queues

- **Blocking Queue**: If a thread tries to pop an item from an empty queue, it blocks until an item becomes available. If it tries to push an item to a full queue (if bounded), it blocks until space becomes available. Condition variables are typically used for blocking.
- **Non-Blocking Queue**: Operations like `try_pop` or `try_push` return immediately, indicating success or failure (e.g., queue empty or full). These often use atomic operations for lock-free implementations.

The `ThreadSafeQueue` implemented in Section 3.2.4 is an example of a blocking queue (`wait_and_pop`) with a non-blocking option (`try_pop`).

6.2.2 Usage Scenarios

- **Producer-Consumer**: Decoupling threads that produce work from threads that consume it.
- **Work Distribution**: Distributing tasks among worker threads in a thread pool.
- **Event Handling**: Queuing events from different sources to be processed by a dedicated handler thread.
- **Message Passing**: Implementing communication between different components or threads (as in the Active Object pattern).

6.3 Pipelines

A pipeline is a pattern where data flows through a series of processing stages. Each stage runs concurrently, often implemented as a separate thread or group of threads, and passes its output to the next stage via a queue.

6.3.1 Concept

Imagine an assembly line:

- Stage 1: Generates raw data.
- Stage 2: Processes the raw data.
- Stage 3: Aggregates or filters the processed data.
- Stage 4: Outputs the final result.

Each stage works concurrently. Stage 2 can process item N while Stage 1 is generating item N+1.

6.3.2 Simple Pipeline Example

Plain Text

```
#include <iostream>
#include <thread>
#include <vector>
#include <chrono>
#include <random>
#include "section3_synchronization.md" // Assuming ThreadSafeQueue is defined
here

// Stage 1: Generate random numbers
void generate_data(ThreadSafeQueue<int>& output_queue, int count) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 100);

    for (int i = 0; i < count; ++i) {
        int data = dis(gen);
        std::cout << "Stage 1: Generated " << data << std::endl;
        output_queue.push(data);
        std::this_thread::sleep_for(std::chrono::milliseconds(50)); //
Simulate work
    }
    output_queue.finish(); // Signal end of data
    std::cout << "Stage 1: Finished generation" << std::endl;
}

// Stage 2: Square the numbers
void process_data(ThreadSafeQueue<int>& input_queue, ThreadSafeQueue<int>&
```

```
output_queue) {
    int data;
    while (input_queue.wait_and_pop(data)) {
        int processed_data = data * data;
        std::cout << "Stage 2: Processed " << data << " -> " <<
processed_data << std::endl;
        output_queue.push(processed_data);
        std::this_thread::sleep_for(std::chrono::milliseconds(100)); //
Simulate work
    }
    output_queue.finish(); // Signal end of data
    std::cout << "Stage 2: Finished processing" << std::endl;
}

// Stage 3: Print the results
void output_data(ThreadSafeQueue<int>& input_queue) {
    int data;
    int sum = 0;
    int count = 0;
    while (input_queue.wait_and_pop(data)) {
        std::cout << "Stage 3: Received " << data << std::endl;
        sum += data;
        count++;
        std::this_thread::sleep_for(std::chrono::milliseconds(20)); //
Simulate work
    }
    std::cout << "Stage 3: Finished output. Total sum: " << sum << ", Count:
" << count << std::endl;
}

int main() {
    ThreadSafeQueue<int> queue1; // Output of Stage 1, Input of Stage 2
    ThreadSafeQueue<int> queue2; // Output of Stage 2, Input of Stage 3

    const int data_count = 20;

    // Start pipeline stages in reverse order to ensure consumers are ready
    std::thread stage3(output_data, std::ref(queue2));
    std::thread stage2(process_data, std::ref(queue1), std::ref(queue2));
    std::thread stage1(generate_data, std::ref(queue1), data_count);

    // Join threads
    stage1.join();
    stage2.join();
    stage3.join();

    std::cout << "Pipeline finished" << std::endl;
```

```
    return 0;
}
```

This example shows:

- Three stages connected by thread-safe queues.
- Each stage runs in its own thread.
- Data flows from `generate_data` -> `process_data` -> `output_data`.
- The `finish()` method on the queue signals the end of data for downstream stages.

6.4 Active Objects

The Active Object pattern decouples method execution from method invocation. When a client calls a method on an active object, the call is transformed into a request message and placed on a queue. A dedicated thread owned by the active object processes requests from the queue sequentially.

6.4.1 Concept

- **Proxy**: Provides the public interface, intercepts method calls, creates request messages, and adds them to the queue.
- **Scheduler**: Manages the request queue and determines the order of execution.
- **Activation Queue**: Holds pending request messages.
- **Servant**: Implements the actual logic of the methods.
- **Execution Thread**: A dedicated thread that dequeues requests and dispatches them to the servant.

6.4.2 Simple Active Object Example (Logging Service)

Plain Text

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

```
#include <condition_variable>
#include <queue>
#include <functional>
#include <string>
#include <memory>
#include <future>

// Message type for the queue
using LogTask = std::function<void()>;

// The Active Object (Logger)
class ActiveLogger {
private:
    // Servant: Actual logging implementation
    void write_log(const std::string& message) {
        // In a real logger, this would write to a file, console, etc.
        std::lock_guard<std::mutex> lock(cout_mutex); // Protect console
output
        std::cout << "[Log] " << message << std::endl;
    }

    // Activation Queue
    std::queue<LogTask> task_queue;
    std::mutex queue_mutex;
    std::condition_variable condition;
    bool done = false;

    // Execution Thread
    std::thread worker_thread;

    // Mutex for console output (for demonstration)
    std::mutex cout_mutex;

    // Worker function executed by the thread
    void run() {
        while (!done) {
            LogTask task;
            {
                std::unique_lock<std::mutex> lock(queue_mutex);
                condition.wait(lock, [this]{ return !task_queue.empty() ||
done; });

                if (done && task_queue.empty()) {
                    return;
                }

                task = std::move(task_queue.front());
                task_queue.pop();
```

```
            }
            task(); // Execute the logging task
        }
    }

public:
    ActiveLogger() {
        // Start the worker thread upon construction
        worker_thread = std::thread(&ActiveLogger::run, this);
    }

    ~ActiveLogger() {
        // Signal the worker thread to stop
        {
            std::unique_lock<std::mutex> lock(queue_mutex);
            done = true;
        }
        condition.notify_one();

        // Wait for the worker thread to finish
        if (worker_thread.joinable()) {
            worker_thread.join();
        }
    }

    // Proxy method: Enqueues a log request
    void log(const std::string& message) {
        // Create a task that calls the servant's method
        LogTask task = [this, message]() {
            write_log(message);
        };

        // Add the task to the queue
        {
            std::lock_guard<std::mutex> lock(queue_mutex);
            task_queue.push(std::move(task));
        }
        condition.notify_one();
    }

    // Example method that returns a value via future
    std::future<size_t> get_queue_size() {
        auto promise = std::make_shared<std::promise<size_t>>();
        std::future<size_t> future = promise->get_future();

        LogTask task = [this, promise]() {
            // This runs on the worker thread
            size_t size = 0;
```

```
            {
                // Need to lock queue_mutex briefly to get size
                // Note: This is just an example; accessing queue size
                // from the worker thread itself might be simpler in
practice.
                std::lock_guard<std::mutex> lock(queue_mutex);
                size = task_queue.size();
            }
            promise->set_value(size);
        };

        {
            std::lock_guard<std::mutex> lock(queue_mutex);
            task_queue.push(std::move(task));
        }
        condition.notify_one();

        return future;
    }
};

// Client code
void client_thread(ActiveLogger& logger, int id) {
    for (int i = 0; i < 5; ++i) {
        std::string msg = "Message from client " + std::to_string(id) + ",
iteration " + std::to_string(i);
        logger.log(msg);
        std::this_thread::sleep_for(std::chrono::milliseconds(10 * id)); //
Vary delay
    }
}

int main() {
    ActiveLogger logger;

    std::vector<std::thread> clients;
    for (int i = 0; i < 3; ++i) {
        clients.emplace_back(client_thread, std::ref(logger), i + 1);
    }

    // Get queue size asynchronously
    auto future_size = logger.get_queue_size();

    // Wait for clients to finish submitting logs
    for (auto& client : clients) {
        client.join();
    }
```

```
    // Wait for the queue size result
    try {
        size_t q_size = future_size.get();
        std::cout << "Approximate queue size after client submission: " <<
q_size << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Error getting queue size: " << e.what() << std::endl;
    }

    // Allow logger time to process remaining messages before destruction
    std::this_thread::sleep_for(std::chrono::seconds(1));

    std::cout << "Logger shutting down..." << std::endl;
    // Logger destructor will handle shutdown

    return 0;
}
```

Benefits of Active Object:

- **Concurrency**: Method invocations from multiple clients run concurrently without blocking each other.
- **Synchronization**: Simplifies synchronization, as the servant's state is only accessed by the single execution thread.
- **Scheduling**: Allows control over the order of request execution.
- **Decoupling**: Separates method invocation from execution.

C++ Concurrency Guide

Section 7: Concurrency Best Practices and Pitfalls

Concurrent programming introduces unique challenges that can lead to subtle bugs and performance issues. This section covers best practices and common pitfalls to help you write robust, efficient concurrent code.

7.1 Data Race Detection and Prevention

A data race occurs when two or more threads access the same memory location concurrently, and at least one of the accesses is a write. Data races lead to undefined behavior and are one of the most common sources of bugs in concurrent programs.

7.1.1 Identifying Data Races

Data races can be difficult to detect because they often manifest intermittently and may depend on specific timing conditions. Here are some signs that might indicate a data race:

- Program behavior changes when adding or removing debugging statements
- Inconsistent results between runs
- Crashes or corruption that only occur under load
- Failures that only appear on specific hardware or operating systems

7.1.2 Preventing Data Races

1. **Use Proper Synchronization**:
    - Protect shared data with mutexes or other synchronization primitives
    - Use atomic operations for simple shared variables
    - Consider using higher-level abstractions like thread-safe containers

Plain Text

```
// BAD: Data race
int counter = 0;
void increment() { counter++; }

// GOOD: Using mutex
std::mutex mtx;
int counter = 0;
void increment() {
    std::lock_guard<std::mutex> lock(mtx);
    counter++;
}

// GOOD: Using atomic
```

```
std::atomic<int> counter(0);
void increment() { counter++; }
```

1. **Minimize Shared Mutable State**:
   - Design your code to minimize the amount of data shared between threads
   - Make shared data immutable when possible
   - Use thread-local storage for thread-specific data

Plain Text

```
// BAD: Shared mutable state
std::vector<int> results;
void process(int id) {
    int result = compute(id);
    results.push_back(result); // Data race!
}

// GOOD: Thread-local results, combined at the end
void process(int id, std::vector<int>& local_results) {
    int result = compute(id);
    local_results.push_back(result); // Thread-local, no race
}

// In main:
std::vector<std::vector<int>> thread_local_results(num_threads);
// ... launch threads with their own result vectors ...
// ... then combine results after joining ...
```

1. **Use Thread-Safe Data Structures**:
   - Use containers designed for concurrent access
   - Implement your own thread-safe wrappers when needed

Plain Text

```
// Thread-safe queue example (simplified)
template<typename T>
class ThreadSafeQueue {
    std::queue<T> queue;
    mutable std::mutex mtx;
```

```
public:
    void push(T value) {
        std::lock_guard<std::mutex> lock(mtx);
        queue.push(std::move(value));
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lock(mtx);
        if (queue.empty()) return false;
        value = std::move(queue.front());
        queue.pop();
        return true;
    }
};
```

1. **Use Tools for Detection**:
   - Thread sanitizers (e.g., `-fsanitize=thread` in GCC/Clang)
   - Static analysis tools
   - Runtime verification tools

7.2 Deadlock Avoidance Strategies

A deadlock occurs when two or more threads are blocked forever, each waiting for resources held by the other threads.

7.2.1 Conditions for Deadlock

Four conditions must be present for a deadlock to occur:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode
2. **Hold and Wait**: A thread holds at least one resource while waiting for additional resources
3. **No Preemption**: Resources cannot be forcibly taken from a thread
4. **Circular Wait**: A circular chain of threads exists, each waiting for a resource held by the next thread

7.2.2 Deadlock Prevention Techniques

1. **Lock Ordering**:
   - Always acquire locks in a consistent order
   - Define a global ordering of locks and follow it everywhere

Plain Text

```
// BAD: Inconsistent lock order can cause deadlock
void transfer(Account& from, Account& to, double amount) {
    std::lock_guard<std::mutex> lock_from(from.mutex);
    std::lock_guard<std::mutex> lock_to(to.mutex);
    // Transfer logic...
}

// GOOD: Consistent lock order based on account ID
void transfer(Account& from, Account& to, double amount) {
    if (from.id < to.id) {
        std::lock_guard<std::mutex> lock_from(from.mutex);
        std::lock_guard<std::mutex> lock_to(to.mutex);
        // Transfer logic...
    } else {
        std::lock_guard<std::mutex> lock_to(to.mutex);
        std::lock_guard<std::mutex> lock_from(from.mutex);
        // Transfer logic...
    }
}
```

1. **Use `std::lock` or `std::scoped_lock`**:
   - These functions can lock multiple mutexes atomically, preventing deadlock

Plain Text

```
// BETTER: Using std::lock to acquire multiple locks atomically
void transfer(Account& from, Account& to, double amount) {
    std::unique_lock<std::mutex> lock_from(from.mutex, std::defer_lock);
    std::unique_lock<std::mutex> lock_to(to.mutex, std::defer_lock);
    std::lock(lock_from, lock_to);
    // Transfer logic...
}

// BEST (C++17): Using std::scoped_lock
void transfer(Account& from, Account& to, double amount) {
    std::scoped_lock lock(from.mutex, to.mutex);
```

```
    // Transfer logic...
}
```

1. **Avoid Nested Locks**:
   - Minimize the number of locks held simultaneously
   - Release locks as soon as possible
2. **Use Timeouts**:
   - Use `try_lock_for` or `try_lock_until` with a timeout
   - Implement a back-off strategy if locks can't be acquired

Plain Text

```
bool try_transfer(Account& from, Account& to, double amount) {
    // Try to acquire locks with timeout
    std::unique_lock<std::timed_mutex> lock_from(from.timed_mutex,
std::defer_lock);
    if (!lock_from.try_lock_for(std::chrono::milliseconds(100)))
        return false;

    std::unique_lock<std::timed_mutex> lock_to(to.timed_mutex,
std::defer_lock);
    if (!lock_to.try_lock_for(std::chrono::milliseconds(100))) {
        // Release first lock and return
        return false;
    }

    // Transfer logic...
    return true;
}
```

1. **Deadlock Detection**:
   - Implement runtime detection of potential deadlocks
   - Use tools like thread sanitizers or deadlock detectors

7.3 Livelock Avoidance

A livelock occurs when threads are actively performing operations but not making progress. This often happens when threads repeatedly respond to each other's actions.

7.3.1 Example of Livelock

Plain Text

```
// Livelock example: Two threads trying to avoid collision
void thread1_function(std::atomic<bool>& thread1_resource, std::atomic<bool>&
thread2_resource) {
    while (true) {
        // Try to acquire resource
        thread1_resource = true;

        // If other thread has its resource, back off
        if (thread2_resource) {
            thread1_resource = false; // Release our resource
            // Wait a bit before retrying
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            continue;
        }

        // If we get here, we have the resource and can proceed
        break;
    }
}

void thread2_function(std::atomic<bool>& thread1_resource, std::atomic<bool>&
thread2_resource) {
    while (true) {
        // Try to acquire resource
        thread2_resource = true;

        // If other thread has its resource, back off
        if (thread1_resource) {
            thread2_resource = false; // Release our resource
            // Wait a bit before retrying
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            continue;
        }

        // If we get here, we have the resource and can proceed
        break;
    }
}
```

In this example, both threads might keep acquiring and releasing their resources in response to each other, never making progress.

7.3.2 Livelock Prevention

1. **Introduce Randomness**:
   - Add random delays or backoff times
   - This breaks the symmetry that can cause livelocks

Plain Text

```
// Adding randomness to avoid livelock
void thread_function(std::atomic<bool>& my_resource, std::atomic<bool>&
other_resource, int id) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 100);

    while (true) {
        // Try to acquire resource
        my_resource = true;

        // If other thread has its resource, back off
        if (other_resource) {
            my_resource = false; // Release our resource
            // Wait a random time before retrying
            std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));
            continue;
        }

        // If we get here, we have the resource and can proceed
        break;
    }
}
```

1. **Prioritize Threads**:
   - Assign priorities to threads
   - In conflict situations, the higher-priority thread proceeds

Plain Text

```
// Using priorities to avoid livelock
void thread_function(std::atomic<bool>& my_resource, std::atomic<bool>&
other_resource, int priority) {
    while (true) {
        // Try to acquire resource
        my_resource = true;

        // If other thread has its resource, use priority to decide
        if (other_resource) {
            if (priority > other_priority) {
                // Higher priority, keep our resource and wait
                std::this_thread::sleep_for(std::chrono::milliseconds(10));
                continue;
            } else {
                // Lower priority, back off
                my_resource = false;
                std::this_thread::sleep_for(std::chrono::milliseconds(10));
                continue;
            }
        }

        // If we get here, we have the resource and can proceed
        break;
    }
}
```

1. **Use Higher-Level Synchronization**:
   - Replace low-level resource management with higher-level constructs
   - Use established algorithms for resource allocation

7.4 Performance Considerations

Concurrent programming introduces performance trade-offs that need to be carefully considered.

7.4.1 Contention

Contention occurs when multiple threads compete for the same resource, leading to serialization and reduced performance.

Strategies to Reduce Contention:

1. **Fine-Grained Locking**:
    - Use multiple locks for different parts of data structures
    - This allows more concurrent access

Plain Text

```
// Coarse-grained locking (high contention)
class CoarseList {
    std::mutex mtx;
    std::list<int> data;
public:
    void add(int value) {
        std::lock_guard<std::mutex> lock(mtx);
        data.push_back(value);
    }

    bool contains(int value) {
        std::lock_guard<std::mutex> lock(mtx);
        return std::find(data.begin(), data.end(), value) != data.end();
    }
};

// Fine-grained locking (lower contention)
class FineList {
    struct Node {
        int value;
        std::mutex mtx;
        Node* next;
        Node(int v) : value(v), next(nullptr) {}
    };

    Node head;

public:
    FineList() : head(0) {} // Dummy head node

    void add(int value) {
        Node* new_node = new Node(value);
        Node* current = &head;

        std::lock_guard<std::mutex> head_lock(head.mtx);
        if (!head.next) {
            head.next = new_node;
            return;
        }
```

```
        Node* next = head.next;
        std::lock_guard<std::mutex> next_lock(next->mtx);
        current = next;

        // Continue traversing with hand-over-hand locking...
    }

    // Similar implementation for contains()...
};
```

1. **Lock-Free Data Structures**:
   - Use atomic operations instead of locks
   - This eliminates blocking and can improve scalability
2. **Partitioning**:
   - Divide data into partitions that can be processed independently
   - Each thread works on its own partition

Plain Text

```
// Partitioned processing
void process_partitioned(const std::vector<int>& data, int num_threads) {
    std::vector<std::thread> threads;
    std::vector<int> results(num_threads, 0);

    int chunk_size = data.size() / num_threads;

    for (int i = 0; i < num_threads; ++i) {
        int start = i * chunk_size;
        int end = (i == num_threads - 1) ? data.size() : (i + 1) *
chunk_size;

        threads.emplace_back([&data, &results, start, end, i]() {
            int local_sum = 0;
            for (int j = start; j < end; ++j) {
                local_sum += process(data[j]);
            }
            results[i] = local_sum;
        });
    }
```

```
    for (auto& t : threads) {
        t.join();
    }

    // Combine results
    int total = 0;
    for (int result : results) {
        total += result;
    }
}
```

7.4.2 False Sharing

False sharing occurs when threads on different cores write to variables that are on the same cache line, causing cache invalidation and reduced performance.

Strategies to Avoid False Sharing:

1. **Padding**:
   - Add padding between variables used by different threads
   - Ensure variables are on different cache lines

Plain Text

```
// Potential false sharing
struct ThreadData {
    int counter;  // Used by thread 1
    int flag;     // Used by thread 2
};

// Avoiding false sharing with padding
struct ThreadData {
    int counter;  // Used by thread 1
    char padding[64]; // Typical cache line size
    int flag;     // Used by thread 2
};

// C++17 solution using std::hardware_destructive_interference_size
struct ThreadData {
    int counter;  // Used by thread 1
    alignas(std::hardware_destructive_interference_size) int flag; // Used by
```

```
thread 2
};
```

1. **Thread-Local Storage**:
   - Use thread-local variables for intermediate results
   - Combine results only when necessary

Plain Text

```
void count_elements(const std::vector<int>& data, int value,
std::atomic<int>& total) {
    // Thread-local counter to avoid atomic operations for every element
    int local_count = 0;

    for (int elem : data) {
        if (elem == value) {
            local_count++;
        }
    }

    // Update shared counter only once
    total.fetch_add(local_count);
}
```

7.4.3 Cache Coherency

Cache coherency protocols ensure that changes to shared data are visible to all cores, but they can introduce overhead.

Strategies for Better Cache Usage:

1. **Locality of Reference**:
   - Process data in a way that maximizes cache hits
   - Keep related data together in memory

Plain Text

```
// Bad for cache: processing elements in random order
void process_random(std::vector<int>& data) {
```

```
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, data.size() - 1);

    for (size_t i = 0; i < data.size(); ++i) {
        int idx = dis(gen);
        process(data[idx]);
    }
}

// Good for cache: processing elements sequentially
void process_sequential(std::vector<int>& data) {
    for (int& elem : data) {
        process(elem);
    }
}
```

1. **Minimize Sharing**:
   - Reduce the amount of data shared between threads
   - Use techniques like thread-local storage and partitioning
2. **Batch Updates**:
   - Accumulate changes locally before updating shared data
   - This reduces the frequency of cache coherency traffic

7.5 Exception Safety in Concurrent Code

Exception handling in concurrent code requires special attention to avoid resource leaks and inconsistent states.

7.5.1 RAII for Resource Management

RAII (Resource Acquisition Is Initialization) is essential for exception safety in concurrent code:

Plain Text

```
// BAD: Manual lock/unlock can leak if an exception is thrown
void unsafe_function() {
    mutex.lock();
```

```
    // If an exception occurs here, mutex remains locked
    process_data();
    mutex.unlock();
}

// GOOD: RAII ensures mutex is unlocked even if an exception occurs
void safe_function() {
    std::lock_guard<std::mutex> lock(mutex);
    // If an exception occurs here, lock's destructor will unlock the mutex
    process_data();
}
```

7.5.2 Exception Propagation Across Threads

Exceptions don't automatically propagate across thread boundaries:

Plain Text

```
// BAD: Exception in thread is not caught
void start_thread() {
    std::thread t([]() {
        throw std::runtime_error("Error in thread");
    });
    t.join(); // Program will terminate with uncaught exception
}

// GOOD: Capture and handle exceptions within the thread
void start_thread() {
    std::thread t([]() {
        try {
            throw std::runtime_error("Error in thread");
        } catch (const std::exception& e) {
            std::cerr << "Caught exception: " << e.what() << std::endl;
        }
    });
    t.join();
}

// BETTER: Use std::promise to propagate exceptions
void start_thread() {
    std::promise<void> promise;
    std::future<void> future = promise.get_future();

    std::thread t([&promise]() {
        try {
            // Do work...
```

```
            throw std::runtime_error("Error in thread");
            promise.set_value();
        } catch (...) {
            promise.set_exception(std::current_exception());
        }
    });

    t.join();

    try {
        future.get(); // Will rethrow the exception
    } catch (const std::exception& e) {
        std::cerr << "Thread threw exception: " << e.what() << std::endl;
    }
}
```

7.5.3 Maintaining Invariants

Ensure that shared data structures maintain their invariants even in the presence of exceptions:

Plain Text

```
class ThreadSafeQueue {
    std::queue<int> queue;
    mutable std::mutex mtx;

public:
    void push(int value) {
        std::lock_guard<std::mutex> lock(mtx);
        queue.push(value); // No exception expected here
    }

    void push_multiple(const std::vector<int>& values) {
        std::lock_guard<std::mutex> lock(mtx);
        // Create a copy of the queue in case of exceptions
        std::queue<int> temp_queue = queue;

        try {
            for (int value : values) {
                temp_queue.push(value);
            }
            // Only update the real queue if all operations succeeded
            queue = std::move(temp_queue);
        } catch (...) {
            // Original queue remains unchanged
```

```
            throw;
        }
    }
};
```

7.6 Debugging Concurrent Applications

Debugging concurrent code presents unique challenges due to non-determinism and timing-dependent behavior.

7.6.1 Tools and Techniques

1. **Thread Sanitizers**:
   - Use compiler options like `-fsanitize=thread` (GCC/Clang)
   - These can detect data races, deadlocks, and other concurrency issues
2. **Logging**:
   - Use thread-safe logging to record events and state changes
   - Include thread IDs in log messages

Plain Text

```
void thread_safe_log(const std::string& message) {
    static std::mutex log_mutex;
    std::lock_guard<std::mutex> lock(log_mutex);
    std::cout << "[Thread " << std::this_thread::get_id() << "] "
              << message << std::endl;
}
```

1. **Assertions**:
   - Use assertions to verify invariants and preconditions
   - This helps catch issues early

Plain Text

```

```

```
void process_data(const std::vector<int>& data) {
    std::lock_guard<std::mutex> lock(data_mutex);
    assert(!data.empty() && "Data should not be empty");
    // Process data...
}
```

1. **Reproducible Tests**:
   - Create tests that can reliably reproduce concurrency issues
   - Use controlled timing and synchronization to trigger race conditions

Plain Text

```
void test_race_condition() {
    std::atomic<bool> thread1_ready(false);
    std::atomic<bool> thread2_ready(false);

    int shared_data = 0;

    std::thread t1([&]() {
        // Signal ready
        thread1_ready = true;

        // Wait for other thread
        while (!thread2_ready) {
            std::this_thread::yield();
        }

        // Try to trigger race condition
        shared_data = 1;
    });

    std::thread t2([&]() {
        // Signal ready
        thread2_ready = true;

        // Wait for other thread
        while (!thread1_ready) {
            std::this_thread::yield();
        }

        // Try to trigger race condition
        shared_data = 2;
    });
```

```
    t1.join();
    t2.join();

    // Check result
    std::cout << "Final value: " << shared_data << std::endl;
}
```

1. **Debugger Support**:
   - Use debuggers with thread support (e.g., GDB, LLDB)
   - Set breakpoints in specific threads
   - Examine thread stacks and shared variables

7.6.2 Common Debugging Strategies

1. **Simplify**:
   - Reduce the number of threads
   - Simplify the synchronization logic
   - Isolate the problematic component
2. **Visualize**:
   - Create diagrams of thread interactions
   - Use tools to visualize lock acquisition and thread states
3. **Stress Test**:
   - Run tests with many threads and high load
   - Vary timing and scheduling to expose race conditions
4. **Add Delays**:
   - Strategically add delays to change timing
   - This can help expose or avoid race conditions

```
```

Plain Text

```
void test_with_delays() {
    std::thread t1([]() {
        // Add delay to change timing
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        // Access shared resource
    });

    std::thread t2([]() {
        // Access shared resource
    });

    t1.join();
    t2.join();
}
```

7.7 Summary of Best Practices

1. **Design for Concurrency**:
   - Minimize shared mutable state
   - Design clear ownership and access patterns
   - Use message passing when appropriate
2. **Use Appropriate Synchronization**:
   - Choose the right primitives for your needs
   - Use the highest-level abstraction that works
   - Be consistent in synchronization strategies
3. **Follow Established Patterns**:
   - Use well-tested concurrency patterns
   - Don't reinvent synchronization mechanisms
   - Learn from existing concurrent libraries
4. **Test Thoroughly**:

- Test with different thread counts and loads
- Use tools to detect concurrency issues
- Create stress tests that target concurrency

5. **Document Thread Safety**:
   - Clearly document thread safety guarantees
   - Specify synchronization requirements
   - Document any assumptions about thread interaction
6. **Measure Performance**:
   - Don't assume concurrency always improves performance
   - Measure and compare different approaches
   - Be aware of diminishing returns with too many threads

C++ Concurrency Guide

Section 8: Exercises and Exams

This section provides exercises to test your understanding of the concepts covered in this guide. Solutions are provided at the end of the section.

8.1 Conceptual Questions (Multiple Choice)

1. What is the primary difference between concurrency and parallelism?
   a) Concurrency is doing many things at once; parallelism is dealing with many things at once.
   b) Concurrency is dealing with many things at once; parallelism is doing many things at once.
   c) Concurrency requires multiple cores; parallelism does not.
   d) They are the same concept.

2. Which of the following is NOT one of the four conditions required for a deadlock?
   a) Mutual Exclusion
   b) Hold and Wait
   c) Preemption
   d) Circular Wait

3. What is the purpose of `std::thread::join()`?
   a) To start a thread.
   b) To detach a thread so it runs independently.
   c) To wait for a thread to complete its execution.
   d) To get the ID of a thread.

4. Which RAII wrapper is generally preferred for simple mutex locking within a scope?
   a) `std::unique_lock`
   b) `std::lock_guard`
   c) `std::scoped_lock`
   d) `std::shared_lock`

5. Why is it necessary to check the predicate in a loop when using `std::condition_variable::wait()`?
   a) To handle spurious wakeups.
   b) To ensure the mutex is locked correctly.
   c) To prevent deadlocks.
   d) To improve performance.

6. What does `std::memory_order_release` guarantee?
   a) All memory operations after the store are visible to other threads.
   b) All memory operations before the store are visible to other threads that acquire the same atomic variable.
   c) Only the atomicity of the store operation is guaranteed.
   d) It provides sequential consistency.

7. What happens when you call `get()` on a `std::future` that was created with `std::launch::deferred`?
   a) It returns immediately with a default value.
   b) It throws an exception.

c) It executes the associated task synchronously and then returns the result.
d) It waits for another thread to execute the task.

8. Which C++20 feature provides automatic joining in its destructor and support for cooperative interruption?
   a) `std::thread`
   b) `std::jthread`
   c) `std::packaged_task`
   d) `std::async`

9. What is false sharing?
   a) Threads sharing data that they shouldn't.
   b) Multiple threads writing to different variables that happen to reside on the same cache line.
   c) Using locks incorrectly, leading to shared access.
   d) A deadlock caused by sharing resources incorrectly.

10. Which synchronization primitive is suitable for allowing multiple readers OR one writer?
    a) `std::mutex`
    b) `std::counting_semaphore`
    c) `std::shared_mutex`
    d) `std::latch`

8.2 Short Coding Problems

1. **Parallel Sum**: Write a function `parallel_sum(const std::vector<int>& data, size_t num_threads)` that calculates the sum of elements in the input vector `data` using `num_threads` threads. Use `std::thread` and basic synchronization (e.g., mutex for accumulating the final sum). Return the total sum.

2. **Thread-Safe Counter**: Implement a `ThreadSafeCounter` class with `increment()`, `decrement()`, and `get()` methods. Use `std::mutex` and `std::lock_guard` for synchronization.

3. **Producer-Consumer with Condition Variable**: Implement a simple producer-consumer scenario using `std::mutex`, `std::condition_variable`, and a `std::queue<int>`. The producer thread should push numbers 0-9 into the queue, and the consumer thread should pop and print them. Ensure proper signaling and handling of the empty/finished state.

4. **Atomic Flag Spinlock**: Implement a simple spinlock class `Spinlock` using `std::atomic_flag`. It should have `lock()` and `unlock()` methods.

5. **Using `std::async`**: Write a function that takes a `std::vector<std::string>` and uses `std::async` to asynchronously calculate the length of each string. Collect the results using futures and return a `std::vector<size_t>` containing the lengths.

6. **Deadlock Creation (for understanding)**: Write a small program that intentionally creates a deadlock between two threads using two `std::mutex` objects. Explain why the deadlock occurs.

7. **Using `std::packaged_task`**: Create a `std::packaged_task` that wraps a function calculating the factorial of a number. Execute this task on a separate thread and retrieve the result using the associated future.

8. **Barrier Synchronization (C++20)**: Write a program where 3 threads perform work in two phases. Use `std::barrier` to ensure all threads complete phase 1 before any thread starts phase 2.

8.3 Debugging Exercises

For each code snippet below, identify the concurrency issue(s) and explain how to fix them.

1. **Data Race on Vector**:

2. **Incorrect Reference Passing**:

3. **Potential Deadlock**:

4. **Unprotected Read**:

5. **Detached Thread with Local Reference**:

8.4 Solutions

8.4.1 Solutions to Conceptual Questions

1. **b)** Concurrency is dealing with many things at once; parallelism is doing many things at once.
2. **c)** Preemption (No Preemption is the condition).
3. **c)** To wait for a thread to complete its execution.
4. **b)** `std::lock_guard` (simplest RAII wrapper for single mutex).
5. **a)** To handle spurious wakeups.
6. **b)** All memory operations before the store are visible to other threads that acquire the same atomic variable.
7. **c)** It executes the associated task synchronously and then returns the result.
8. **b)** `std::jthread` .
9. **b)** Multiple threads writing to different variables that happen to reside on the same cache line.
10. **c)** `std::shared_mutex` .

8.4.2 Solutions to Short Coding Problems

1. **Parallel Sum**:
2. **Thread-Safe Counter**:
3. **Producer-Consumer with Condition Variable**:
4. **Atomic Flag Spinlock**:
5. **Using** `std::async` :
6. **Deadlock Creation**:
7. **Using** `std::packaged_task` :

8. **Barrier Synchronization (C++20)**:

8.4.3 Solutions to Debugging Exercises

1. **Data Race on Vector**:
   - **Issue**: Multiple threads call `push_back` on the same `std::vector` (`shared_vec`) without synchronization. `std::vector` is not thread-safe for concurrent modifications.
   - **Fix**: Protect access to `shared_vec` with a `std::mutex`.
2. **Incorrect Reference Passing**:
   - **Issue**: `my_string` is passed by value (copied) to the thread constructor because `std::thread`'s constructor copies its arguments by default. The `modify_string` function receives a reference to this *copy*, not the original `my_string`.
   - **Fix**: Use `std::ref` to explicitly pass `my_string` by reference.
3. **Potential Deadlock**:
   - **Issue**: Classic deadlock scenario due to inconsistent lock ordering. `func1` locks `m1` then `m2`, while `func2` locks `m2` then `m1`.
   - **Fix**: Ensure a consistent lock order (e.g., always lock `m1` before `m2`) or use `std::lock` / `std::scoped_lock`.
4. **Unprotected Read**:
   - **Issue**: The `reader` thread reads `shared_value` without acquiring the mutex `mtx`, while the `writer` thread modifies it. This is a data race.
   - **Fix**: Protect the read operation in `reader` with the same mutex.
5. **Detached Thread with Local Reference**:
   - **Issue**: The thread `t` is detached and captures a reference (`std::ref(local_msg)`) to a local variable `local_msg` in `create_detached`. When `create_detached` exits, `local_msg` is destroyed, but the detached thread `t` might still be running or about to access the (now dangling) reference, leading to undefined behavior.

- **Fix 1 (Pass by Value)**: Pass the string by value (copy) to the thread.
- **Fix 2 (Join the Thread)**: Don't detach the thread; join it before `local_msg` goes out of scope.

C++ Concurrency Guide

Section 9: Final Project - Concurrent Web Server Simulation

9.1 Project Overview

In this final project, you will implement a simulated concurrent web server that can handle multiple client requests simultaneously. This project will test your understanding of various C++ concurrency concepts and patterns covered in this guide.

The server will:

- Accept and process multiple client requests concurrently
- Implement a thread pool for efficient thread management
- Use a request queue with producer-consumer pattern
- Handle different types of requests with varying processing times
- Implement proper synchronization and resource management
- Provide statistics on server performance

9.2 Project Requirements

9.2.1 Core Components

1. **Client Request Simulator**:
   - Generate simulated HTTP requests (GET, POST, PUT, DELETE)
   - Assign different processing times to different request types
   - Send requests to the server at configurable rates

2. **Request Queue**:
    - Thread-safe queue for incoming requests
    - Support for prioritization (optional)
    - Proper signaling between producers and consumers
3. **Thread Pool**:
    - Fixed number of worker threads
    - Efficient task distribution
    - Graceful shutdown mechanism
4. **Request Handler**:
    - Process different request types
    - Simulate I/O operations and CPU-bound work
    - Return appropriate responses
5. **Response Handler**:
    - Collect and process responses
    - Maintain statistics (response times, throughput, etc.)
6. **Logger**:
    - Thread-safe logging mechanism
    - Different log levels (INFO, WARNING, ERROR)
    - Option to log to console or file

9.2.2 Technical Requirements

Your implementation must use:

- `std::thread` for thread management

- At least three different synchronization primitives (e.g., mutex, condition variable, atomic)
- At least one higher-level concurrency pattern (e.g., thread pool, active object)
- Proper exception handling in a concurrent context
- RAII principles for resource management

9.2.3 Performance Requirements

Your server should:

- Handle at least 1000 requests per second
- Maintain reasonable response times under load
- Avoid thread starvation
- Properly utilize available CPU cores
- Avoid excessive memory usage

9.3 Project Structure

Here's a suggested structure for your project:

Plain Text

```
concurrent_web_server/
├── include/
│   ├── client_simulator.h
│   ├── request_queue.h
│   ├── thread_pool.h
│   ├── request_handler.h
│   ├── response_handler.h
│   ├── logger.h
│   └── common.h
├── src/
│   ├── client_simulator.cpp
│   ├── request_queue.cpp
│   ├── thread_pool.cpp
│   ├── request_handler.cpp
│   ├── response_handler.cpp
```

```
│   ├── logger.cpp
│   └── main.cpp
├── CMakeLists.txt
└── README.md
```

9.4 Implementation Guide

9.4.1 Step 1: Define the Request and Response Types

Start by defining the basic data structures:

Plain Text

```
// common.h
#pragma once

#include <string>
#include <chrono>

enum class RequestType {
    GET,
    POST,
    PUT,
    DELETE
};

struct Request {
    int id;
    RequestType type;
    std::string resource;
    std::string body;
    std::chrono::steady_clock::time_point arrival_time;

    Request(int id, RequestType type, std::string resource, std::string body
= "")
        : id(id), type(type), resource(resource), body(body),
          arrival_time(std::chrono::steady_clock::now()) {}
};

enum class StatusCode {
    OK = 200,
    CREATED = 201,
    BAD_REQUEST = 400,
    NOT_FOUND = 404,
    SERVER_ERROR = 500
};
```

```
struct Response {
    int request_id;
    StatusCode status;
    std::string body;
    std::chrono::steady_clock::time_point completion_time;

    Response(int request_id, StatusCode status, std::string body = "")
        : request_id(request_id), status(status), body(body),
          completion_time(std::chrono::steady_clock::now()) {}
};
```

9.4.2 Step 2: Implement the Thread-Safe Request Queue

Plain Text

```
// request_queue.h
#pragma once

#include "common.h"
#include <queue>
#include <mutex>
#include <condition_variable>

class RequestQueue {
private:
    std::queue<Request> queue_;
    mutable std::mutex mutex_;
    std::condition_variable not_empty_;
    std::condition_variable not_full_;
    size_t capacity_;
    bool closed_ = false;

public:
    explicit RequestQueue(size_t capacity = 100);

    // Add a request to the queue
    bool enqueue(const Request& request);

    // Get a request from the queue
    bool dequeue(Request& request);

    // Check if the queue is empty
    bool empty() const;

    // Get the current size of the queue
    size_t size() const;
```

```
    // Close the queue (no more enqueuing)
    void close();

    // Check if the queue is closed
    bool is_closed() const;
};
```

Plain Text

```
// request_queue.cpp
#include "request_queue.h"

RequestQueue::RequestQueue(size_t capacity)
    : capacity_(capacity) {}

bool RequestQueue::enqueue(const Request& request) {
    std::unique_lock<std::mutex> lock(mutex_);

    // Wait until there's space or the queue is closed
    not_full_.wait(lock, [this] {
        return queue_.size() < capacity_ || closed_;
    });

    // Don't enqueue if the queue is closed
    if (closed_) {
        return false;
    }

    queue_.push(request);

    // Notify that the queue is not empty
    not_empty_.notify_one();

    return true;
}

bool RequestQueue::dequeue(Request& request) {
    std::unique_lock<std::mutex> lock(mutex_);

    // Wait until there's an item or the queue is closed and empty
    not_empty_.wait(lock, [this] {
        return !queue_.empty() || (closed_ && queue_.empty());
    });

    // If the queue is empty and closed, there's nothing to dequeue
    if (queue_.empty()) {
```

```
        return false;
    }

    request = queue_.front();
    queue_.pop();

    // Notify that the queue is not full
    not_full_.notify_one();

    return true;
}

bool RequestQueue::empty() const {
    std::lock_guard<std::mutex> lock(mutex_);
    return queue_.empty();
}

size_t RequestQueue::size() const {
    std::lock_guard<std::mutex> lock(mutex_);
    return queue_.size();
}

void RequestQueue::close() {
    std::lock_guard<std::mutex> lock(mutex_);
    closed_ = true;
    not_empty_.notify_all();
    not_full_.notify_all();
}

bool RequestQueue::is_closed() const {
    std::lock_guard<std::mutex> lock(mutex_);
    return closed_;
}
```

9.4.3 Step 3: Implement the Thread Pool

Plain Text

```
// thread_pool.h
#pragma once

#include <vector>
#include <thread>
#include <functional>
#include <atomic>
#include "request_queue.h"
#include "request_handler.h"
```

```
#include "response_handler.h"

class ThreadPool {
private:
    std::vector<std::thread> workers_;
    RequestQueue& request_queue_;
    RequestHandler& request_handler_;
    ResponseHandler& response_handler_;
    std::atomic<bool> stop_ = false;

    // Worker thread function
    void worker_function();

public:
    ThreadPool(RequestQueue& request_queue,
               RequestHandler& request_handler,
               ResponseHandler& response_handler,
               size_t num_threads);

    ~ThreadPool();

    // Start the thread pool
    void start();

    // Stop the thread pool
    void stop();

    // Get the number of worker threads
    size_t size() const;
};
```

Plain Text

```
// thread_pool.cpp
#include "thread_pool.h"
#include "logger.h"

ThreadPool::ThreadPool(RequestQueue& request_queue,
                       RequestHandler& request_handler,
                       ResponseHandler& response_handler,
                       size_t num_threads)
    : request_queue_(request_queue),
      request_handler_(request_handler),
      response_handler_(response_handler) {

    workers_.reserve(num_threads);
}
```

```
void ThreadPool::start() {
    for (size_t i = 0; i < workers_.capacity(); ++i) {
        workers_.emplace_back(&ThreadPool::worker_function, this);
    }
    Logger::log(Logger::Level::INFO, "Thread pool started with " +
                std::to_string(workers_.size()) + " workers");
}

void ThreadPool::stop() {
    stop_ = true;
    request_queue_.close();

    for (auto& worker : workers_) {
        if (worker.joinable()) {
            worker.join();
        }
    }

    Logger::log(Logger::Level::INFO, "Thread pool stopped");
}

void ThreadPool::worker_function() {
    while (!stop_) {
        Request request;
        if (request_queue_.dequeue(request)) {
            try {
                Response response = request_handler_.handle_request(request);
                response_handler_.handle_response(request, response);
            } catch (const std::exception& e) {
                Logger::log(Logger::Level::ERROR, "Exception in worker: " +
                            std::string(e.what()));

                Response error_response(request.id, StatusCode::SERVER_ERROR,
                                        "Internal server error");
                response_handler_.handle_response(request, error_response);
            }
        }
    }
}

size_t ThreadPool::size() const {
    return workers_.size();
}

ThreadPool::~ThreadPool() {
    if (!stop_) {
        stop();
```

```
    }
}
```

9.4.4 Step 4: Implement the Request Handler

Plain Text

```
// request_handler.h
#pragma once

#include "common.h"
#include <unordered_map>
#include <functional>
#include <random>

class RequestHandler {
private:
    std::unordered_map<std::string, std::function<Response(const Request&)>>
resource_handlers_;
    std::mt19937 rng_; // Random number generator for simulating processing
time

    // Simulate processing time based on request type
    void simulate_processing(const Request& request);

public:
    RequestHandler();

    // Handle a request and return a response
    Response handle_request(const Request& request);

    // Register a handler for a specific resource
    void register_handler(const std::string& resource,
                          std::function<Response(const Request&)> handler);
};
```

Plain Text

```
// request_handler.cpp
#include "request_handler.h"
#include "logger.h"
#include <thread>

RequestHandler::RequestHandler() : rng_(std::random_device()()) {
    // Register default handlers for common resources
```

```
    register_handler("/index.html", [](const Request& req) {
        return Response(req.id, StatusCode::OK, "<html><body>Hello
World</body></html>");
    });

    register_handler("/api/data", [](const Request& req) {
        if (req.type == RequestType::GET) {
            return Response(req.id, StatusCode::OK, "{\"data\":
\"sample\"}");
        } else if (req.type == RequestType::POST) {
            return Response(req.id, StatusCode::CREATED, "{\"status\":
\"created\"}");
        } else {
            return Response(req.id, StatusCode::BAD_REQUEST, "{\"error\":
\"Method not allowed\"}");
        }
    });
}

void RequestHandler::simulate_processing(const Request& request) {
    // Simulate different processing times for different request types
    std::uniform_int_distribution<> dist;

    switch (request.type) {
        case RequestType::GET:
            dist = std::uniform_int_distribution<>(10, 50); // Fast
            break;
        case RequestType::POST:
            dist = std::uniform_int_distribution<>(50, 200); // Medium
            break;
        case RequestType::PUT:
            dist = std::uniform_int_distribution<>(100, 300); // Slow
            break;
        case RequestType::DELETE:
            dist = std::uniform_int_distribution<>(30, 100); // Medium
            break;
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(dist(rng_)));
}

Response RequestHandler::handle_request(const Request& request) {
    Logger::log(Logger::Level::INFO, "Handling request " +
std::to_string(request.id) +
                " for resource " + request.resource);

    // Simulate processing time
    simulate_processing(request);
```

```
    // Find handler for the requested resource
    auto it = resource_handlers_.find(request.resource);
    if (it != resource_handlers_.end()) {
        return it->second(request);
    }

    // Resource not found
    return Response(request.id, StatusCode::NOT_FOUND, "Resource not found");
}

void RequestHandler::register_handler(const std::string& resource,
                                      std::function<Response(const Request&)>
handler) {
    resource_handlers_[resource] = handler;
}
```

9.4.5 Step 5: Implement the Response Handler

Plain Text

```
// response_handler.h
#pragma once

#include "common.h"
#include <mutex>
#include <atomic>
#include <vector>
#include <chrono>

class ResponseHandler {
private:
    std::mutex stats_mutex_;
    std::atomic<int> total_requests_ = 0;
    std::atomic<int> successful_requests_ = 0;
    std::atomic<int> failed_requests_ = 0;

    // For calculating average response time
    std::vector<double> response_times_;

    // For calculating throughput
    std::chrono::steady_clock::time_point start_time_;

public:
    ResponseHandler();

    // Handle a response
```

```
    void handle_response(const Request& request, const Response& response);

    // Get statistics
    struct Stats {
        int total_requests;
        int successful_requests;
        int failed_requests;
        double average_response_time_ms;
        double throughput_rps; // Requests per second
    };

    Stats get_stats() const;
};
```

Plain Text

```
// response_handler.cpp
#include "response_handler.h"
#include "logger.h"

ResponseHandler::ResponseHandler()
    : start_time_(std::chrono::steady_clock::now()) {}

void ResponseHandler::handle_response(const Request& request, const Response&
response) {
    // Calculate response time
    auto response_time =
std::chrono::duration_cast<std::chrono::milliseconds>(
        response.completion_time - request.arrival_time).count();

    // Update statistics
    total_requests_++;

    if (static_cast<int>(response.status) < 400) {
        successful_requests_++;
    } else {
        failed_requests_++;
    }

    {
        std::lock_guard<std::mutex> lock(stats_mutex_);
        response_times_.push_back(response_time);
    }

    // Log response
    Logger::log(Logger::Level::INFO,
                "Request " + std::to_string(request.id) +
```

```
                " completed with status " + std::to_string(static_cast<int>
(response.status)) +
                " in " + std::to_string(response_time) + "ms");
}

ResponseHandler::Stats ResponseHandler::get_stats() const {
    Stats stats;
    stats.total_requests = total_requests_;
    stats.successful_requests = successful_requests_;
    stats.failed_requests = failed_requests_;

    // Calculate average response time
    {
        std::lock_guard<std::mutex> lock(stats_mutex_);
        if (!response_times_.empty()) {
            double sum = 0.0;
            for (double time : response_times_) {
                sum += time;
            }
            stats.average_response_time_ms = sum / response_times_.size();
        } else {
            stats.average_response_time_ms = 0.0;
        }
    }

    // Calculate throughput
    auto elapsed_seconds = std::chrono::duration_cast<std::chrono::seconds>(
        std::chrono::steady_clock::now() - start_time_).count();

    if (elapsed_seconds > 0) {
        stats.throughput_rps = static_cast<double>(total_requests_) /
elapsed_seconds;
    } else {
        stats.throughput_rps = 0.0;
    }

    return stats;
}
```

9.4.6 Step 6: Implement the Client Simulator

Plain Text

```
// client_simulator.h
#pragma once

#include "common.h"
```

```
#include "request_queue.h"
#include <thread>
#include <atomic>
#include <random>

class ClientSimulator {
private:
    RequestQueue& request_queue_;
    std::thread simulator_thread_;
    std::atomic<bool> stop_ = false;
    std::atomic<int> next_request_id_ = 0;

    // Random number generators
    std::mt19937 rng_;
    std::uniform_int_distribution<> request_type_dist_;
    std::uniform_int_distribution<> resource_dist_;

    // Available resources to request
    std::vector<std::string> resources_ = {
        "/index.html",
        "/api/data",
        "/images/logo.png",
        "/css/style.css",
        "/js/script.js"
    };

    // Simulator thread function
    void simulator_function(int requests_per_second);

    // Generate a random request
    Request generate_request();

public:
    explicit ClientSimulator(RequestQueue& request_queue);

    // Start the simulator with a specified request rate
    void start(int requests_per_second);

    // Stop the simulator
    void stop();

    ~ClientSimulator();
};
```

Plain Text

```
// client_simulator.cpp
#include "client_simulator.h"
#include "logger.h"
#include <chrono>

ClientSimulator::ClientSimulator(RequestQueue& request_queue)
    : request_queue_(request_queue),
      rng_(std::random_device()()),
      request_type_dist_(0, 3),
      resource_dist_(0, resources_.size() - 1) {}

Request ClientSimulator::generate_request() {
    int id = next_request_id_++;
    RequestType type = static_cast<RequestType>(request_type_dist_(rng_));
    std::string resource = resources_[resource_dist_(rng_)];

    std::string body;
    if (type == RequestType::POST || type == RequestType::PUT) {
        body = "{\"data\": \"sample" + std::to_string(id) + "\"}";
    }

    return Request(id, type, resource, body);
}

void ClientSimulator::simulator_function(int requests_per_second) {
    // Calculate delay between requests
    auto delay = std::chrono::microseconds(1000000 / requests_per_second);

    Logger::log(Logger::Level::INFO,
                "Client simulator started with " +
std::to_string(requests_per_second) +
                " requests per second");

    while (!stop_) {
        auto start = std::chrono::steady_clock::now();

        // Generate and enqueue a request
        Request request = generate_request();
        if (!request_queue_.enqueue(request)) {
            // Queue is closed or full
            break;
        }

        // Sleep for the remaining time to maintain the request rate
        auto elapsed = std::chrono::steady_clock::now() - start;
        if (elapsed < delay) {
            std::this_thread::sleep_for(delay - elapsed);
```

```
        }
    }

    Logger::log(Logger::Level::INFO, "Client simulator stopped");
}

void ClientSimulator::start(int requests_per_second) {
    simulator_thread_ = std::thread(&ClientSimulator::simulator_function,
                                    this, requests_per_second);
}

void ClientSimulator::stop() {
    stop_ = true;
    if (simulator_thread_.joinable()) {
        simulator_thread_.join();
    }
}

ClientSimulator::~ClientSimulator() {
    if (!stop_) {
        stop();
    }
}
```

9.4.7 Step 7: Implement the Logger

Plain Text

```
// logger.h
#pragma once

#include <string>
#include <mutex>
#include <fstream>
#include <iostream>

class Logger {
public:
    enum class Level {
        DEBUG,
        INFO,
        WARNING,
        ERROR
    };

    static void init(const std::string& log_file = "", Level min_level =
Level::INFO);
```

```
    static void log(Level level, const std::string& message);
    static void close();

private:
    static std::mutex mutex_;
    static std::ofstream log_file_;
    static Level min_level_;
    static bool initialized_;

    static std::string level_to_string(Level level);
    static std::string get_timestamp();
};
```

Plain Text

```
// logger.cpp
#include "logger.h"
#include <chrono>
#include <iomanip>
#include <sstream>

std::mutex Logger::mutex_;
std::ofstream Logger::log_file_;
Logger::Level Logger::min_level_ = Logger::Level::INFO;
bool Logger::initialized_ = false;

void Logger::init(const std::string& log_file, Level min_level) {
    std::lock_guard<std::mutex> lock(mutex_);

    if (!log_file.empty()) {
        log_file_.open(log_file, std::ios::out | std::ios::app);
    }

    min_level_ = min_level;
    initialized_ = true;
}

void Logger::log(Level level, const std::string& message) {
    if (level < min_level_) {
        return;
    }

    std::lock_guard<std::mutex> lock(mutex_);

    std::string timestamp = get_timestamp();
    std::string level_str = level_to_string(level);
    std::string thread_id = std::to_string(
```

```
        std::hash<std::thread::id>{}(std::this_thread::get_id()));

    std::string log_message = timestamp + " [" + level_str + "] [" +
thread_id + "] " + message;

    // Output to console
    std::cout << log_message << std::endl;

    // Output to file if opened
    if (log_file_.is_open()) {
        log_file_ << log_message << std::endl;
        log_file_.flush();
    }
}

void Logger::close() {
    std::lock_guard<std::mutex> lock(mutex_);

    if (log_file_.is_open()) {
        log_file_.close();
    }

    initialized_ = false;
}

std::string Logger::level_to_string(Level level) {
    switch (level) {
        case Level::DEBUG:   return "DEBUG";
        case Level::INFO:    return "INFO";
        case Level::WARNING: return "WARNING";
        case Level::ERROR:   return "ERROR";
        default:             return "UNKNOWN";
    }
}

std::string Logger::get_timestamp() {
    auto now = std::chrono::system_clock::now();
    auto now_c = std::chrono::system_clock::to_time_t(now);
    auto now_ms = std::chrono::duration_cast<std::chrono::milliseconds>(
        now.time_since_epoch()) % 1000;

    std::stringstream ss;
    ss << std::put_time(std::localtime(&now_c), "%Y-%m-%d %H:%M:%S");
    ss << '.' << std::setfill('0') << std::setw(3) << now_ms.count();

    return ss.str();
}
```

9.4.8 Step 8: Implement the Main Function

Plain Text

```
// main.cpp
#include "client_simulator.h"
#include "request_queue.h"
#include "thread_pool.h"
#include "request_handler.h"
#include "response_handler.h"
#include "logger.h"
#include <iostream>
#include <thread>
#include <chrono>
#include <iomanip>

void print_stats(const ResponseHandler::Stats& stats) {
    std::cout << "=== Server Statistics ===" << std::endl;
    std::cout << "Total Requests: " << stats.total_requests << std::endl;
    std::cout << "Successful Requests: " << stats.successful_requests <<
std::endl;
    std::cout << "Failed Requests: " << stats.failed_requests << std::endl;
    std::cout << "Average Response Time: " << std::fixed <<
std::setprecision(2)
              << stats.average_response_time_ms << " ms" << std::endl;
    std::cout << "Throughput: " << std::fixed << std::setprecision(2)
              << stats.throughput_rps << " requests/second" << std::endl;
    std::cout << "==========================" << std::endl;
}

int main(int argc, char* argv[]) {
    // Initialize logger
    Logger::init("server.log", Logger::Level::INFO);

    // Parse command line arguments
    int num_threads = std::thread::hardware_concurrency();
    int requests_per_second = 100;
    int run_seconds = 10;

    if (argc > 1) num_threads = std::stoi(argv[1]);
    if (argc > 2) requests_per_second = std::stoi(argv[2]);
    if (argc > 3) run_seconds = std::stoi(argv[3]);

    // Create components
    RequestQueue request_queue(1000);
    RequestHandler request_handler;
    ResponseHandler response_handler;
```

```
    // Create and start thread pool
    ThreadPool thread_pool(request_queue, request_handler, response_handler,
num_threads);
    thread_pool.start();

    // Create and start client simulator
    ClientSimulator client_simulator(request_queue);
    client_simulator.start(requests_per_second);

    // Run for specified duration
    std::cout << "Server running with " << num_threads << " threads, "
              << requests_per_second << " requests/second for "
              << run_seconds << " seconds..." << std::endl;

    // Print stats periodically
    for (int i = 0; i < run_seconds; ++i) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        if (i % 2 == 0) { // Print every 2 seconds
            print_stats(response_handler.get_stats());
        }
    }

    // Stop components
    client_simulator.stop();
    std::this_thread::sleep_for(std::chrono::seconds(1)); // Allow queue to
drain
    thread_pool.stop();

    // Print final stats
    std::cout << "\nFinal Statistics:" << std::endl;
    print_stats(response_handler.get_stats());

    // Close logger
    Logger::close();

    return 0;
}
```

9.4.9 Step 9: Create CMakeLists.txt

Plain Text

```
cmake_minimum_required(VERSION 3.10)
project(concurrent_web_server)

# Set C++ standard
```

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Include directories
include_directories(include)

# Source files
set(SOURCES
    src/client_simulator.cpp
    src/request_queue.cpp
    src/thread_pool.cpp
    src/request_handler.cpp
    src/response_handler.cpp
    src/logger.cpp
    src/main.cpp
)

# Create executable
add_executable(server ${SOURCES})

# Link with pthread on Unix-like systems
if(UNIX)
    target_link_libraries(server pthread)
endif()

# Enable warnings
if(MSVC)
    target_compile_options(server PRIVATE /W4)
else()
    target_compile_options(server PRIVATE -Wall -Wextra -pedantic)
endif()
```

9.5 Building and Running the Project

To build and run the project:

1. Create the directory structure and files as described above.
2. Build the project using CMake:
3. Run the server:

9.6 Project Extensions (Optional)

If you want to challenge yourself further, consider implementing these extensions:

1. **Request Prioritization**:
   - Add priority levels to requests
   - Implement a priority queue instead of a regular queue
2. **Dynamic Thread Pool**:
   - Adjust the number of threads based on load
   - Implement a work-stealing algorithm
3. **Metrics Dashboard**:
   - Create a simple web interface to display server metrics
   - Use WebSocket for real-time updates
4. **Load Testing**:
   - Implement different load patterns (constant, burst, ramp-up)
   - Analyze server performance under different loads
5. **Persistent Storage**:
   - Add a simple key-value store for request data
   - Implement proper synchronization for database access

9.7 Evaluation Criteria

Your implementation will be evaluated based on:

1. **Correctness**:
   - Does the server handle requests correctly?
   - Are there any race conditions or deadlocks?
   - Is the code free of memory leaks and resource leaks?
2. **Performance**:

- How many requests per second can the server handle?
- How does the response time scale with load?
- Is CPU utilization efficient?

3. **Code Quality**:
   - Is the code well-structured and easy to understand?
   - Are concurrency patterns applied appropriately?
   - Is there proper error handling and logging?
4. **Concurrency Concepts**:
   - Are multiple concurrency concepts from the guide applied?
   - Is synchronization implemented correctly?
   - Are there any unnecessary synchronization points?
5. **Documentation**:
   - Is the code well-documented?
   - Is there a clear explanation of design decisions?
   - Are there instructions for building and running the project?

9.8 Submission Guidelines

Submit your project as a ZIP file containing:

1. All source code files
2. CMakeLists.txt
3. A README.md file with:
   - Project overview
   - Build instructions

- Usage instructions
- Design decisions and concurrency patterns used
- Performance analysis
- Any known limitations or issues

Good luck with your final project! This will be an excellent opportunity to apply all the concurrency concepts you've learned in a realistic scenario.