Chapter 14: Concurrency and the C++ Memory Model

Modern CPUs have multiple cores, and leveraging them requires writing concurrent code. The C++ standard library provides a portable, platform-independent set of tools for managing threads, protecting shared data, and reasoning about the complexities of concurrent execution.

14.1 Threads and Synchronization

`std::thread`

The `std::thread` class is the fundamental tool for launching a new thread of execution. You create an instance of `std::thread` with a function (or any callable object) that you want to run.

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

void task() {
    std::cout << "Hello from a different thread!\n";
}

int main() {
    std::thread my_thread(task);

    // Do other work in the main thread...

    my_thread.join(); // The main thread waits here for my_thread to finish.

    return 0;
}
```

Once a thread is launched, you must either `join()` it (wait for it to finish) or `detach()` it (let it run independently). If a `std::thread` object is destroyed without being joined or detached, your program will terminate.

The Problem: Data Races

When multiple threads access the same memory location and at least one of those accesses is a write, you have a **data race**. This is a form of undefined behavior and is one of the most common and difficult bugs in concurrent programming.

To prevent data races, you must use synchronization primitives to ensure that only one thread can access the shared data at a time. The most common of these is the **mutex** (`std::mutex`).

RAII Locking with `std::lock_guard` and `std::unique_lock`

Manually calling `lock()` and `unlock()` on a mutex is dangerous. If an exception is thrown while the mutex is locked, `unlock()` might never be called, leading to a permanent deadlock.

The C++ solution is to use RAII wrappers that tie the lifetime of the lock to the lifetime of a stack-allocated object.

- **`std::lock_guard`**: A simple, lightweight RAII wrapper. It locks the mutex in its constructor and unlocks it in its destructor. It cannot be unlocked manually.

  ```
  std::mutex mtx;
  int shared_counter = 0;

  void increment() {
      std::lock_guard<std::mutex> lock(mtx); // Lock is acquired here
      shared_counter++;
  } // Lock is automatically released here as `lock` goes out of scope
  ```

- **`std::unique_lock`**: A more flexible, but slightly heavier, RAII wrapper. It also locks on construction and unlocks on destruction, but it allows for more advanced operations like manually unlocking, deferred locking, and transferring ownership of the lock.

14.2 Atomic Operations and Ordering

Sometimes, a full mutex is overkill. For simple operations like incrementing a counter or setting a flag, C++ provides **atomic operations** via the `std::atomic` template.

An atomic operation is an indivisible operation. When a thread performs an atomic read or write on a shared variable, no other thread can see the variable in a half-modified state.

```
#include <atomic>

std::atomic<int> atomic_counter = 0;

void atomic_increment() {
    atomic_counter++; // This is a single, indivisible, thread-safe operation
}
```

This is the basis of **lock-free programming**: writing concurrent code that does not rely on mutexes. This can offer significant performance benefits but is notoriously difficult to get right.

Theory: The C++ Memory Model

What exactly is a data race? The C++ Memory Model provides the formal definition. It establishes a relationship called **happens-before**. If action A *happens-before* action B, then the effects of A are guaranteed to be visible to B.

A data race occurs if there are two conflicting actions (accessing the same memory, one is a write) in different threads that are not ordered by a *happens-before* relationship.

Synchronization operations (like locking a mutex or performing an atomic operation) create these *happens-before* relationships, which is why they prevent data races.

What: Memory Ordering Guarantees

Modern CPUs and compilers can reorder instructions to improve performance. They can reorder memory operations as long as it doesn't change the behavior of a single thread. However, this reordering can be disastrous for concurrent code.

Atomic operations can take an additional argument that specifies the **memory ordering** constraints the compiler and CPU must obey.

- **`std::memory_order_seq_cst` (Sequentially Consistent):** The default and strongest ordering. All threads see all sequentially consistent atomic operations in the same order. This is the easiest to reason about but can be the most expensive.
- **`std::memory_order_acquire` / `std::memory_order_release`:** A common pattern. An operation with `release` semantics (like unlocking a mutex) ensures that all memory writes that happened *before* it are visible to another thread that performs an `acquire` operation (like locking the same mutex).
- **`std::memory_order_relaxed`:** The weakest ordering. It provides no ordering guarantees with respect to other memory operations, only that the atomic operation itself is indivisible. This is the fastest but also the most difficult to use correctly.

Understanding and using memory ordering correctly is an advanced topic, but it is crucial for writing high-performance, lock-free code.

‘‘‘