522031910747+李若彬+hw11

实现性能要求

朴素实现

> 一个显然的实现是使用mutex保护一个记录是否已经执行过 `call_once` 操作的变量，请实现。

- 使用 `std::mutex` 和一个布尔标志 `initialized` 来确保初始化函数只被调用一次。
- `call_once_waiting` 函数中，使用 `std::lock_guard` 锁住互斥锁，检查 `initialized` 标志，如果未初始化则调用传入的函数并设置标志。

```
struct waiting_once
{
    // 如果想遵从C++23，可以用 std::move_only_function<void()>
    using init_function = std::function<void()>;

    void call_once_waiting(init_function f);
private:
    // 添加你需要的成员或函数
    std::mutex mtx;
    bool initialized = false;
};
```

```
void waiting_once::call_once_waiting(init_function f)
{
    // TODO: implement this
     std::lock_guard<std::mutex> lock(mtx);
     if (!initialized) {
        f();
        initialized = true;
     }
}

```

进阶实现（优化）

> 由于初始化只有1次，当程序运行一定进度后，再每次都上互斥锁开销太大。请考虑如何优化。

为了优化 `call_once_waiting` 的性能，我们可以使用双重检查锁定（Double-Checked Locking）模式。这种模式可以减少在初始化完成后每次调用时的锁开销。

```
struct waiting_once
{
    // 如果想遵从C++23，可以用 std::move_only_function<void()>
    using init_function = std::function<void()>;

    void call_once_waiting(init_function f);
private:
    // 添加你需要的成员或函数
    std::atomic<bool> initialized = false;
    std::mutex mtx;
    std::condition_variable cv;
};
```

```
void waiting_once::call_once_waiting(init_function f)
{
    // TODO: implement this
    if (!initialized.load(std::memory_order_acquire)) {
        std::unique_lock<std::mutex> lock(mtx);
        if (!initialized.load(std::memory_order_relaxed)) {
            f();
            initialized.store(true, std::memory_order_release);
            cv.notify_all();
        } else {
            cv.wait(lock, [this]() { return
initialized.load(std::memory_order_relaxed); });
        }
    }
}
```

1. **双重检查锁定**：
   - 首先使用 `initialized.load(std::memory_order_acquire)` 检查是否已经初始化。如果已经初始化，则直接返回，避免了锁的开销。
   - 如果未初始化，进入临界区，使用 `std::unique_lock` 锁定互斥锁。
   - 再次检查 `initialized` 标志，确保在获取锁后仍未初始化。
   - 调用初始化函数 `f()`，然后设置 `initialized` 标志，并通知所有等待的线程。
   - 如果在获取锁后发现已经初始化，则等待条件变量，直到初始化完成。
2. **内存顺序**：
   - 使用 `memory_order_acquire` 和 `memory_order_release` 确保内存操作的顺序，避免编译器和 CPU 重排序导致的竞态条件。

这种优化方法可以减少在初始化完成后每次调用时的锁开销，从而提高性能。

测试两种实现的性能

> 请测试你两种实现的性能，至少包括已经完成初始化后 `call_once_waiting` 的吞吐量（即单位时间内能执行几次 `call_once_waiting` 函数调用）。

朴素实现

```
time: 0.588386s
throughput: 16995642 ops/s
time: 2.55781s
throughput: 7819182 ops/s
time: 3.58448s
throughput: 11159224 ops/s
time: 8.63553s
throughput: 9264055 ops/s
```

进阶实现（优化）

```
time: 0.275915s
throughput: 36243103 ops/s
time: 0.28781s
throughput: 69490329 ops/s
time: 0.316513s
throughput: 126377172 ops/s
time: 0.65696s
throughput: 121773031 ops/s
```

分析

1. **时间**：
   - 进阶实现的时间明显短于朴素实现。例如，在单线程情况下，进阶实现的时间为 `0.275915s`，而朴素实现为 `0.588386s`。
   - 随着线程数量的增加，进阶实现的时间增长较慢，而朴素实现的时间增长较快。
2. **吞吐量**：
   - 进阶实现的吞吐量显著高于朴素实现。例如，在单线程情况下，进阶实现的吞吐量为 `36243103 ops/s`，而朴素实现为 `16995642 ops/s`。
   - 随着线程数量的增加，进阶实现的吞吐量增长较快，而朴素实现的吞吐量增长较慢。

原因

1. **锁开销**：
   - 朴素实现每次调用 `call_once_waiting` 都需要获取互斥锁，这在多线程情况下会导致较高的锁开销和线程争用。
   - 进阶实现使用了双重检查锁定，减少了在初始化完成后每次调用时的锁开销，从而提高了性能。
2. **内存顺序**：
   - 进阶实现使用了 `memory_order_acquire` 和 `memory_order_release` 来确保内存操作的顺序，避免了不必要的内存屏障和同步开销。

3. **条件变量**：
    - 进阶实现使用了条件变量来等待初始化完成，这比简单的互斥锁更高效，尤其是在高并发情况下。

结论

进阶实现通过双重检查锁定和条件变量的使用，显著减少了锁开销和线程争用，从而在多线程环境中表现出更高的性能和吞吐量。测试结果表明，进阶实现的优化是有效的，特别是在高并发情况下，性能提升尤为显著。