HW11 多线程同步练习

一、简述优化方法

1、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。

为了实现这一点、我们在 waiting_once 结构体中加入了互斥锁 m_mutex 以及记录是否执行过 call_once 操作的变量 call_once。具体代码如下图所示：

```
17  struct waiting_once
18  {
19  #if __cpp_lib_move_only_function >= 202110L
20      using init_function = std::move_only_function<void()>;
21  #else
22      using init_function = std::function<void()>;
23  #endif
24
25      void call_once_waiting(init_function f);
26      // TODO: implement this
27  private:
28      std::mutex m_mutex;
29      bool call_once = false;
30  };
```

接着、在 call_once_waiting 函数中、创建了一个 lock_guard 对象，它会自动锁定 lock 方法给定的 m_mutex。而当 lock_guard 对象被销毁时，它会自动解锁互斥体。用此方法利用互斥锁保证了线程安全的基础上、再用 call_once 变量去判断是否执行给定的 f 函数，便能实现朴素要求。具体代码如下图所示：

```
3   void waiting_once::call_once_waiting(init_function f)
4   {
5       // TODO: implement this
6       std::lock_guard<std::mutex> lock(m_mutex);
7       if (!call_once)
8       {
9           f();
10          call_once = true;
11      }
12  }
```

2.1、进阶实现方法 1

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。

考虑进行优化。这里在 waiting_once 结构体中加入了 std::once_flag 变量 m_flag 用于记录函数是否在多线程中已经被调用过。具体代码如下图所示：

```
17  struct waiting_once
18  {
19  #if __cpp_lib_move_only_function >= 202110L
20      using init_function = std::move_only_function<void()>;
21  #else
22      using init_function = std::function<void()>;
23  #endif
24
25      void call_once_waiting(init_function f);
26      // TODO: implement this
27  private:
28      std::once_flag m_flag;
29  };
```

接着、在 call_once_waiting 函数中、通过调用 std::call_once 方法、传入 m_flag 参数和对应的一个可调用的对象 f。m_flag 对象用于记录 call_once_waiting 函数是否已经被调用过。如果已经调用过，std::call_once 就不会再次调用它。对于进阶实现的问题，std::call_once 和 std::once_flag 的组合已经考虑了这个问题。在函数被调用之后，std::call_once 不会再锁定互斥量，因此没有额外的开销。具体代码如下图所示：

```cpp
void waiting_once::call_once_waiting(init_function f)
{
    // TODO: implement this
    std::call_once(m_flag, f);
}
```

2.2、进阶实现方法 2

在 waiting_once 结构体中定义和朴素方法一样的互斥锁 m_mutex 以及记录是否执行过 call_once 操作的变量 call_once。具体代码如下图所示：

```cpp
struct waiting_once
{
#if __cpp_lib_move_only_function >= 202110L
    using init_function = std::move_only_function<void()>;
#else
    using init_function = std::function<void()>;
#endif

    void call_once_waiting(init_function f);
    // TODO: implement this
private:
    std::mutex m_mutex;
    bool call_once = false;
};
```

但是在 call_once_waiting 函数中、在创建锁的外层又增加了一层简单的是否初始化的判断、如果运行过了、就不需要上锁和解锁了。同样能达到效果。

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

二、测试两种实现的性能

测试利用给定的 test_throughput 函数、测定当 times_per_thread（每个线程的执行册数）固定为 10e7，线程数分别为 1，2，4，8 个时，程序的运行时间和对应 call_once_waiting 的吞吐量。

```cpp
test_throughput(10000000, 1);
test_throughput(10000000, 2);
test_throughput(10000000, 4);
test_throughput(10000000, 8);
```

1、朴素实现

得到程序的运行时间和对应 call_once_waiting 的吞吐量如下图所示：

```
horizon22@horizon22:/mnt/hgfs/share/hw11/basic_impl$ ./main
time: 0.390837s
throughput: 25586087 ops/s
time: 1.7629s
throughput: 11344961 ops/s
time: 3.49385s
throughput: 11448698 ops/s
time: 6.67049s
throughput: 11993119 ops/s
```

2.1、进阶实现方法 1

得到程序的运行时间和对应 call_once_waiting 的吞吐量如下图所示，发现相较于朴素实现的版本、在性能上有了极大的优化和提升:

```
horizon22@horizon22:/mnt/hgfs/share/hw11/advanced_impl$ ./main
time: 0.284224s
throughput: 35183512 ops/s
time: 0.298502s
throughput: 67001142 ops/s
time: 0.30857s
throughput: 129630338 ops/s
time: 0.512851s
throughput: 155990666 ops/s
```

2.2、进阶实现方法 2

得到程序的运行时间和对应 call_once_waiting 的吞吐量如下图所示，发现相较于运用封装好的 std 方法的版本、在性能上又有了进一步的优化和提升:

```
horizon22@horizon22:/mnt/hgfs/share/hw11/advanced_impl_2$ ./main
time: 0.180961s
throughput: 55260662 ops/s
time: 0.182299s
throughput: 109709933 ops/s
time: 0.18943s
throughput: 211159508 ops/s
time: 0.292937s
throughput: 273096716 ops/s
```

分析与总结：

1、 当进行进阶优化后、节省了为多个线程上锁和解锁的开销、因此吞吐量变得更大了，多线程带来的性能提升随着线程数近似地线性增长。
2、 发现在进阶优化的实现中、当线程数为 8 时、会有明显的时间增加、分析可能原因是因为本虚拟机分配的处理器内存总数为 6 个、超过实际内存总数之后、需要靠进程切换来实现多线程的操作、因此出现了显著的耗时增加现象。
3、 进阶优化方法 2 比进阶优化方法 1 也有较大的提升、分析可能原因是因为减少了封装好的函数调用中包括的许多额外开销。
4、 朴素方法的单线程的吞吐量相较于线程数为 2、4、8 的情况并没有显著地下降、分析可能原因是因为省去了许多线程等待与切换的开销。