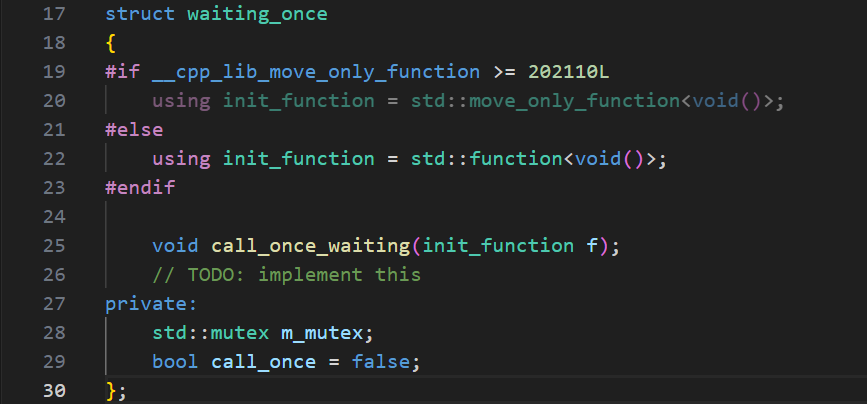
HW11 多线程同步练习

**一、简述优化方法**

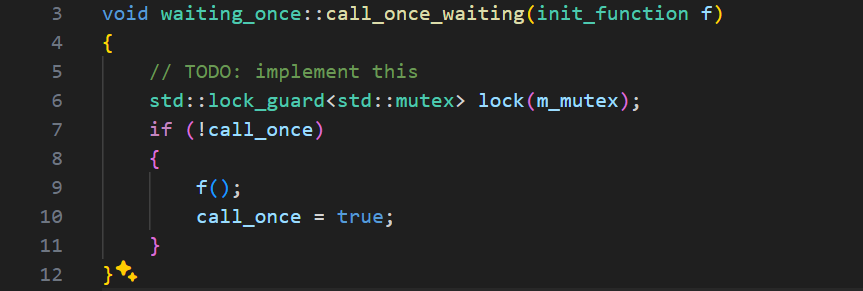
**1、朴素实现**

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：



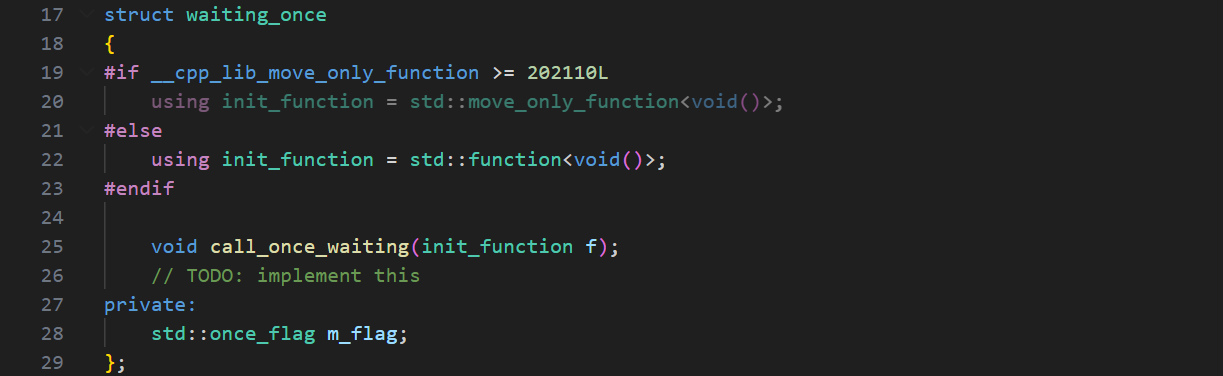
接着、在call\_once\_waiting函数中、创建了一个 lock\_guard 对象，它会自动锁定lock方法给定的m\_mutex。而当 lock\_guard 对象被销毁时，它会自动解锁互斥体。用此方法利用互斥锁保证了线程安全的基础上、再用call\_once变量去判断是否执行给定的f函数，便能实现朴素要求。具体代码如下图所示：



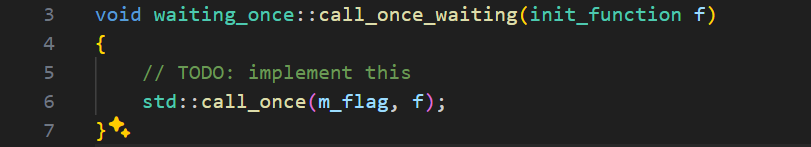
**2.1、进阶实现方法1**

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：

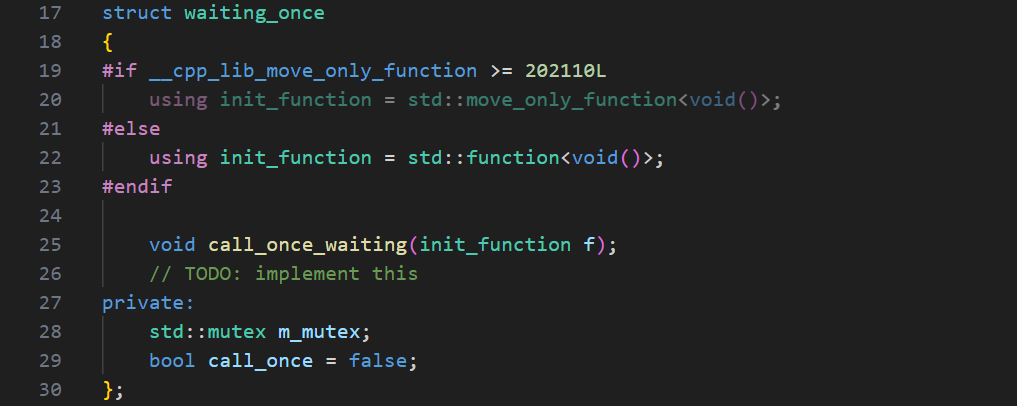


接着、在call\_once\_waiting函数中、通过调用std::call\_once方法、传入m\_flag参数和对应的一个可调用的对象f。m\_flag对象用于记录call\_once\_waiting函数是否已经被调用过。如果已经调用过，std::call\_once 就不会再次调用它。对于进阶实现的问题，std::call\_once 和 std::once\_flag 的组合已经考虑了这个问题。在函数被调用之后，std::call\_once 不会再锁定互斥量，因此没有额外的开销。具体代码如下图所示：

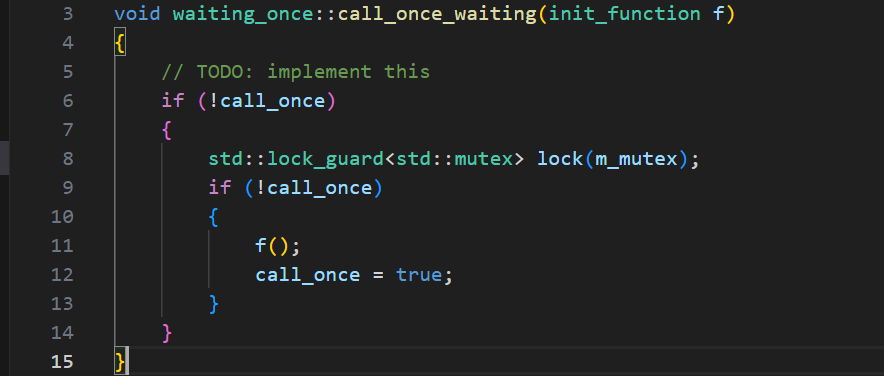


**2.2、进阶实现方法2**

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：

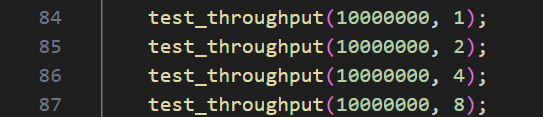
****

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。

****

**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。

****

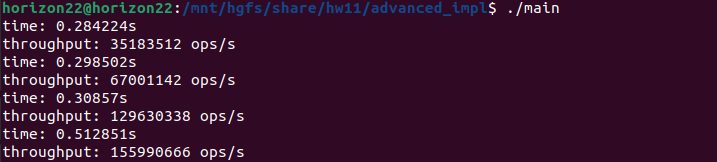
**1、朴素实现**

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



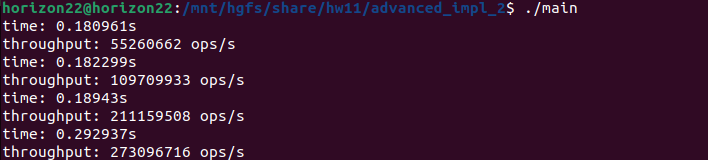
**2.1、进阶实现方法1**

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：

****

**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：

****

**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。**