HW11

522031910213 朱涵

May 26, 2024

1 优化方法

1.1 朴素实现

代码如下图。

```
void waiting_once::call_once_waiting(init_function f) // 朴素版 每次调用都需要获取互斥锁 浪费资源
{
    std::lock_guard<std::mutex> lock(mtx); // 获取互斥锁
    if (!initialized)                      // 如果未初始化
    {
        f();
        initialized = true;
    }
}
```

Figure 1: 朴素版代码

即使用一个布尔变量initialized来判断此对象是否已经初始化。因为是共享变量，因此访问需要用一个互斥锁进行保护。每次调用初始化函数时，先获取锁，然后判断是否初始化，如果未初始化则调用函数f，否则不进行任何操作。

1.2 进阶实现

代码如下图。

```
void waiting_once::call_once_waiting_plus(init_function f)
// 进阶版 无法解决刚开始时多个线程同时初始化的问题 但是在初始化完成后 后续的线程不需要获取互斥锁
{
    if (!initialized) // 如果已经初始化 不做任何事
    {
        std::lock_guard<std::mutex> lock(mtx); // 未初始化 需要进行第一次初始化 则获取互斥锁
        if (!initialized)                      // 防止多个线程同时初始化
        {
            f();
            initialized = true;
        }
    }
}
```

Figure 2: 进阶版代码

实现思路也很简单。朴素版的实现中，在进行初始化之后，如果继续调用callonce函数，仍需要获取锁。实际上不需要每次都获取锁，我们只要先判断一下是否已经初始化，如果已经初始化，就不获取锁，否则再去尝试获取锁进行初始化。虽然这个版本解决不了最开始多线程进行初始化时仍会产生多次获取锁的问题，但是解决了后续调用的资源浪费。

注意，为了方便，我没有修改main程序的测试代码，而是在waiting_once类里编写了两个函数，分别是朴素版和进阶版。如果需要进行测试，记得对测试代码进行相应的修改，或者把朴素版的函数替换成进阶版的函数代码。

2 测试结果

2.1 朴素实现

如下图。

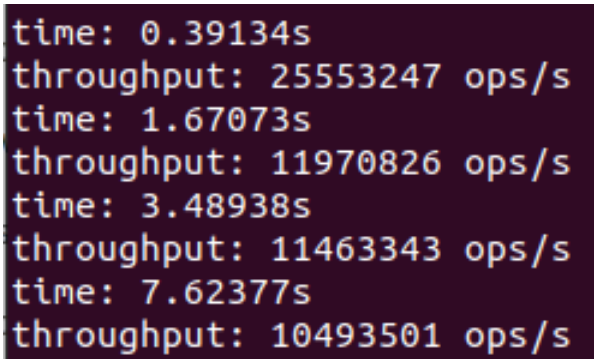

Figure 3: 朴素版测试结果

从上到下一次是线程数为1、2、4、8的耗时以及吞吐量，测试调用次数为10000000。可以发现随着线程数增多，耗时变大，吞吐量也不增反减。这是因为每个线程都要获取锁资源，造成了极大的**无用开销**。

2.2 进阶实现

如下图。

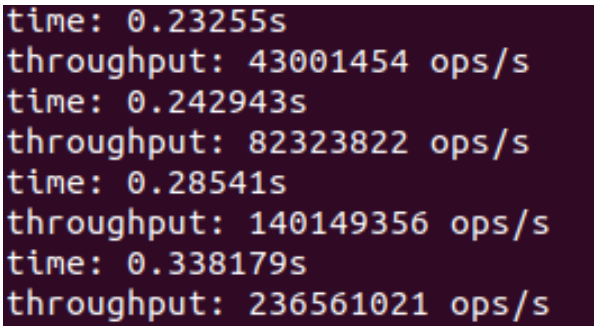

Figure 4: 进阶版测试结果

测试参数同朴素版。可以发现不管是耗时还是吞吐量都显著增优，而且随着线程数变多吞吐量成倍增长，这是符合预期的。**主要原因**就是后续大部分的线程都无需再获取锁，只是简单的进行判断和返回。至于执行耗时为什么会有略微增长，我也不是很明白原因，可能与系统底层相关。