老师用了两个环境变量，一个互斥锁实现读写锁，我们现在要用一个环境变量，一个互斥锁实现读写锁。

读写锁大概是，多个线程可以同时读取共享资源，但当有线程在进行写操作时，其他线程（无论是读还是写）都必须等待，并且同一时间只能有一个线程读取文件。

我会从互斥锁和单一的环境变量详细说一下：

读操作：

- **多个线程可以同时执行读操作**，只要没有线程在进行写操作。
- **读线程不会阻塞其他读线程**，但会被写线程阻塞，直到所有读操作完成。
- 所以，在读请求处理函数处理请求的时候，首先一个读写锁同一时间只能运行（临界区），所以不管是处理请求还是进行读写操作都要挂锁，等待的时候再释放。

锁说完了，再说一下环境变量：

- 一个请求到达的时候，当前有读者或者写者，环境变量就会让他等待。
- 如果条件满足就开始写，写完了就先看有没有写者排队，有的话就唤醒环境变量，没有的话看有没有读者排队，同理。

写操作（Writer）：

- **当有写线程在执行时，所有的读线程和其他写线程都需要等待**，直到写操作完成。
- **写线程是独占的**，即写操作必须在没有任何读或写操作时才能执行。

锁上面已经说了，这里只说环境变量。

- 一个请求到达的时候，当前有写者或者排队的写者，环境变量就会让他等待。
- 如果条件满足就开始读，读完了就先看当前有没有读者以及有没有写者排队，无读者以及有写者排队话就唤醒写者的环境变量。

在进程里，我们会分配不同的进程，分别是读进程和写进程，这两个进程都是死循环。

读进程：

1. 首先调用读写锁对象的 `start_read` 方法，如果当前没有写者正在写入，或者如果允许多个读者同时读取，则允许该读者继续。否则，读者线程会被阻塞，直到可以安全地读取。
2. 通过之后，打印读取信息，让线程暂停一段随机时间，范围在 0.1 到 0.3 秒之间，模拟了实际读取操作所需的时间。
3. 再然后调用读写锁对象的 `done_read` 方法，表示该读者完成了读取操作。
4. 最后让线程暂停一段随机时间，模拟读者在两次读取操作之间的等待时间。

写进程：

- 写进程也一样，和读进程差不多。

以下是我代码执行的开头一段的输出：

[读者 1] 等待读取

[读者 1] 开始读取

[读者 1] 正在读取: 0

[读者 2] 等待读取

[读者 2] 开始读取

[读者 2] 正在读取: 0

[读者 3] 等待读取

[读者 3] 开始读取

[读者 3] 正在读取: 0

[写者 1] 等待写入

[写者 1] 等待读者和写者释放

[写者 2] 等待写入

[写者 2] 等待读者和写者释放

[读者 1] 读取完成

[读者 3] 读取完成

[读者 2] 读取完成

[读者 2] 通知写者

[写者 1] 开始写入

[写者 1] 正在写入: 1

[写者 2] 等待读者和写者释放

[读者 1] 等待读取

[读者 1] 等待写者释放

[读者 2] 等待读取

[读者 2] 等待写者释放

[写者 1] 写入完成

[写者 1] 通知写者

[写者 2] 开始写入

[写者 2] 正在写入: 2

[读者 2] 等待写者释放

[读者 1] 等待写者释放

[读者 3] 等待读取

[读者 3] 等待写者释放

[写者 1] 等待写入

[写者 1] 等待读者和写者释放

[写者 2] 写入完成

[写者 2] 通知写者

[读者 1] 等待写者释放

[写者 1] 开始写入

[写者 1] 正在写入: 3

[读者 3] 等待写者释放

[读者 2] 等待写者释放

[写者 2] 等待写入

[写者 2] 等待读者和写者释放

[写者 1] 写入完成

日志解读

多个读者同时读取：

1. `[读者 1]` 和 `[读者 2]` 同时开始读取，显示 `currentReaders` 能够增加到多个，允许并发读取。
2. `[读者 3]` 也开始读取，进一步验证了多读并发性。

写者等待：

1. `[写者 1]` 和 `[写者 2]` 等待写入，直到所有读者完成读取。

写者顺序执行：

1. 写者按照请求顺序执行写入，确保写者之间互斥。

通知机制：

1. 完成读取和写入后，适时通知其他等待的线程，确保写者和读者能够继续执行。

关键点确认

- **并发读取**：
  - 多个读者能够同时读取共享资源，符合读写锁的预期行为。
- **写者独占写入**：
  - 写者在有读者正在读取或其他写者正在写入时，会进入等待状态，确保写入的独占性。

easy：

见cpp文件

medium：

Python标准库中并没有直接提供CAS操作，尤其是在解释器层面，我用lock模拟了一下。

```
def _compare_and_swap(self, address, expected, new_value):
    """模拟CAS操作，比较并交换。"""
    with self.lock:
        if address == expected:
            return new_value
        return address
```

with self.lock: 语句确保在进入代码块时获取锁，离开代码块时自动释放锁。这样，整个 if判断和可能的赋值操作是在同一时间内由单个线程执行的，保证了操作的原子性。

- `if address == expected:`：比较当前值与预期值是否相同。
- `return new_value` 或 `return address`：根据比较结果返回新值或保持原值。

如果CAS操作能够通过硬件支持实现，效果会更好。