0. 前言

实验概述

锁，中断禁用与抢占式调度禁用。了解自旋锁和互斥锁在OSTD的实现，以及Rust提供的原子操作

实验内容

1. 了解OSTD中自旋锁
2. 了解OSTD中的禁用中断和抢占式调度
3. 了解OSTD中的互斥锁

1. 自旋锁

1.1 自旋锁的实现

自旋锁会在锁被他人占用时进行等待，下面是OSTD中自旋锁`SpinLock`的实现，代码均经过简化：

```
// Ref: asterinas: ostd/src/sync/spin.rs

pub struct SpinLock<T: ?Sized, G = PreemptDisabled> {
    phantom: PhantomData<G>,
    inner: SpinLockInner<T>,
}

struct SpinLockInner<T: ?Sized> {
    lock: AtomicBool,
    val: UnsafeCell<T>,
}

impl<T: ?Sized, G: Guardian> SpinLock<T, G> {
    pub fn lock(&self) -> SpinLockGuard<T, G> {
        let inner_guard = G::guard();
        self.acquire_lock();
        SpinLockGuard_ {
            lock: self,
            guard: inner_guard,
        }
    }

    fn acquire_lock(&self) {
        while !self.try_acquire_lock() {
            core::hint::spin_loop();
        }
    }

    fn try_acquire_lock(&self) -> bool {
        self.inner
            .lock
            .compare_exchange(false, true, Ordering::Acquire,
Ordering::Relaxed)
```

```
            .is_ok()
    }

    fn release_lock(&self) {
        self.inner.lock.store(false, Ordering::Release);
    }
}

```

OSTD的锁会与存放资源进行绑定，在使用纯safe Rust的前提下，使用者不能通过获取锁以外的功能访问内部存放的数据。当使用者希望访问时，需要进行锁操作，在其内部不断地通过调用`acquire_lock`尝试获取锁，如果没有成功则会遗址进行自旋等待操作。这里判断锁是否被他人占用是通过使用`AtomicBool`，一个布尔的原子类型。

> **Rust中的原子类型**
>
> Rust在`std::sync::atomic`或`core::sync::atomic`中提供了基础数据类型所对应的原子类型如`AtomicBool`，`AtomicUsize`，`AtomicIsize`。这些原子类型可在正确使用时可以实现线程间的同步更新。在自旋锁实现的时候，我们使用到了`AtomicBool::compare_exchange`和`AtomicBool::store`的操作，其中会涉及到内存序（Memory Ordering），在感兴趣的同学可以查看Rust对于所有内存序的介绍。
>
> 在这里，使用Acquire和Release的解释为：尝试获取锁的操作总会在别人正在释放锁后进行。

同时可以注意到在获得锁后，返回的是一个`SpinLockGuard<T, G>`，而不是直接返回内部存放数据的可变或不可变引用。原因在于为了方便以及强制使用者进行释放锁操作，需要使用到Assignment1提到的Guard-like结构，即当`SpinLockGuard<T, G>`被释放时，会自动调用`SpinLock::release_lock`以释放锁。

1.2 自旋锁的性能损失和死锁

自旋锁带来的问题是当获取锁失败时，系统会不断地进行等待操作，导致浪费CPU时间片，降低系统性能，甚至在未拥有抢占式调度的情况下，直接导致系统死锁。因此，OSTD为了减轻这种情况的发生频率，提供了禁用中断和禁用抢占式调度两种功能。这两种功能可以消除其他线程拥有运行的可能性，减少因等待锁导致的性能损失（在有抢占式调度的前提下）。

禁用抢占式调度和中断代表两种不同级别的禁用，下面是对于他们的解释:

- 禁用抢占式调度会阻止其他线程进行运行，但不会阻止中断的产生，因此一些代码中如果涉及到了中断上下文，该禁用不会产生效果，也会造成性能损失和死锁。
- 禁用中断是比禁用抢占式调度更高一级的禁用，在之前的课程我们学习到，抢占式调度是需要基于时钟中断的，因此禁用掉会使得抢占式调度同时也会间接被禁用。

然而，我们不能在每个地方均使用禁用中断，因为会导致系统的响应速度变慢，比如在一个需要执行很久的逻辑中，禁用中断会导致网络驱动无法正常收包，或者导致输入设备驱动无法接受用户输入，使得系统在"观感"上卡死。

OSTD将禁用抢占式调度和禁用中断分别进行了声明，并与`SpinLock`绑定在一起，下面是这三种声明所对应的代码：

```
// Ref: asterinas: ostd/src/sync/spin.rs

pub struct SpinLock<T: ?Sized, G = PreemptDisabled> {
    phantom: PhantomData<G>,
    inner: SpinLockInner<T>,
```

```rust
}

struct SpinLockInner<T: ?Sized> {
    lock: AtomicBool,
    val: UnsafeCell<T>,
}

// ========== Preemption Disabled ==========

pub struct PreemptDisabled;

impl Guardian for PreemptDisabled {
    type Guard = DisabledPreemptGuard;

    fn guard() -> Self::Guard {
        disable_preempt()
    }
}

// ========== Interrupt Disabled ==========

pub struct LocalIrqDisabled;

impl Guardian for LocalIrqDisabled {
    type Guard = DisabledLocalIrqGuard;

    fn guard() -> Self::Guard {
        disable_local()
    }
}
```

在这两种禁用中，分别会使用到`disable_preempt`和`disable_local`来进行禁用抢占式调度和禁用中断的行为。除此之外，这两个函数会记录禁用的次数，以防止在进行多次嵌套锁操作时，中途抢占式调度或中断被启用。

> **RISC-V中的禁用中断**
>
> 禁用中断方面，RISC-V使用`sstatus.SIE`编码中断使能（Enable）状态，当其被设置为`1`时，表示中断被使能，当其被设置为`0`时，表示中断被禁用。`disable_local`内部正是通过这一机制屏蔽中断的。

2. 互斥锁

互斥锁会在锁被他人占用时进行阻塞操作，使得当前线程被挂起。内核可以调度执行其他线程，并在合适的时机唤醒挂起的线程。下面是OSTD中互斥锁`Mutex`的实现：

```rust
// Ref: asterinas: ostd/src/sync/mutex.rs

pub struct Mutex<T: ?Sized> {
    lock: AtomicBool,
    queue: WaitQueue,
    val: UnsafeCell<T>,
}

```

```rust
impl<T: ?Sized> Mutex<T> {
    pub fn lock(&self) -> MutexGuard<T> {
        self.queue.wait_until(|| self.try_lock())
    }

    pub fn try_lock(&self) -> Option<MutexGuard<T>> {
        self.acquire_lock()
            .then(|| unsafe { MutexGuard::new(self) })
    }

    fn unlock(&self) {
        self.release_lock();
        self.queue.wake_one();
    }

    fn acquire_lock(&self) -> bool {
        self.lock
            .compare_exchange(false, true, Ordering::Acquire,
Ordering::Relaxed)
            .is_ok()
    }

    fn release_lock(&self) {
        self.lock.store(false, Ordering::Release);
    }
}

```

相比自旋锁，互斥锁内部会多出一个等待队列，其内部维护了等待唤醒的线程，当释放锁时通知等待队列唤醒其中的线程。

3. 上手练习

在并发中另一个比较重要的概念便是信号量，但其并没有在OSTD中进行实现，原因是信号量机制在系统内部不太常用。请你参考Mutex与MutexGuard的实现，实现一个信号量，并对外提供两个接口：

- `acquire(&self) -> SemaphoreGuard`
- `try_acquire(&self) -> Option<SemaphoreGuard>`

实现不需要考虑过多，可以直接使用`SpinLock`或者`Mutex`来对信号进行保护，需要注意的是，要使用到等待队列`WaitQueue`，当释放一个`SemaphoreGuard`的时候要去唤醒队列中的一个任务。

> OSTD除了提供`Mutex`与`SpinLock`，还提供了读写锁`RwMutex`与`RwLock`，由于它们内部实现会比`Mutex`或者`SpinLock`复杂的多，因此在本节课并没有进行讲述，实现好信号量的同学可以查看`ostd/src/sync`目录下的`rwmutex.rs`与`rwlock.rs`文件