

Lab8 - Synchronization

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

实验代码

拉取实验代码：

```
$ git clone https://github.com/sdww0/sustech-os-lab.git && cd sustech-os-lab/lab8-sync
```

或更新实验代码：

```
# In sustech-os-lab
$ git pull && cd lab8-sync
```

本次代码没有对内核进行更新，在lab8-sync下更新了几个本节课要用到的用户态程序

1. 数据竞争

数据竞争（Data Race）是并发编程中常见的问题，它发生在多个线程同时访问同一个共享内存位置，且至少有一个访问是写入操作，且这些访问没有适当的同步机制时。

首先来看一个普通的C程序：

```
// test.c

#include <threads.h>
#include <stdio.h>

unsigned long counter = 0;
const unsigned long N = 1000000;

void thread_func() {
    while (counter < N) {
        counter++;
    }
}

int main() {
    thrd_t thread[20];

    for (int i = 0; i < 20; i++) {
        thrd_create(&thread[i], (thrd_start_t)thread_func, NULL);
    }

    for (int i = 0; i < 20; i++) {
        thrd_join(thread[i], NULL);
    }
}
```

```

    }

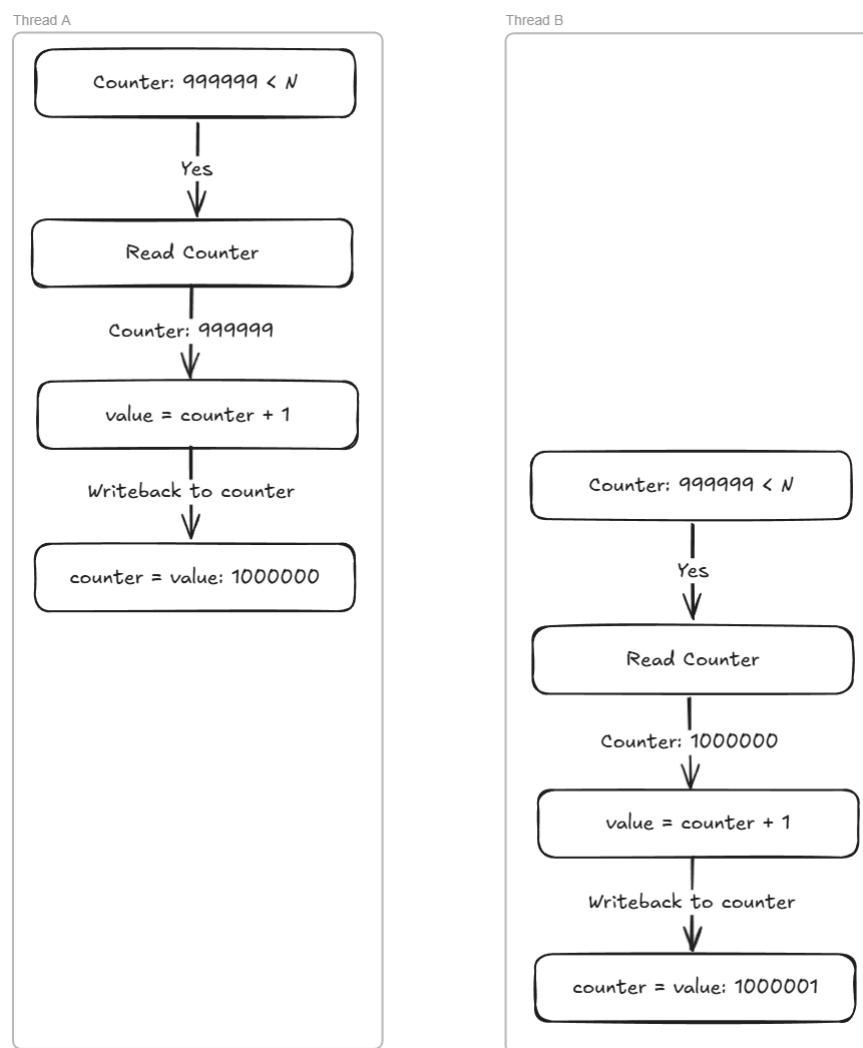
    printf("Final counter value: %lu\n", counter);

    return 0;
}

```

编译并运行：`gcc test.c && ./a.out`，虽然多数情况下其输出是 `Final counter value: 1000000`，但仍然会在某些运行中得到输出：`Final counter value: 1000001`。这个结果明显不符合预期，因为我们期望最终计数器的值正好等于1000000。这种现象就是典型的数据竞争（Data Race）问题。

在上面的例子中，`counter++` 操作实际上不是原子性的，它可能被拆分为多个步骤 [读入寄存器，寄存器值加1，写回内存]。当多个线程同时执行这个操作时，可能会发生以下执行顺序：



这里数据竞争的原因是在进行某些操作时进行了线程切换，第一种避免该问题的方法是关闭中断，但这在用户态程序中无法进行，因此我们可以采用第二个策略，即**原子操作**，原子操作是不可中断的，能够确保在操作期间不会发生线程切换，修改如下：

```

// test2.c

#include <threads.h>
#include <stdio.h>
#include <stdatomic.h>

atomic_ulong counter = 0;

```

```

const unsigned long N = 1000000;

void thread_func() {
    unsigned long current;
    while (1) {
        current = atomic_load(&counter);
        if (current >= N)
            break;
        atomic_compare_exchange_weak(&counter, &current, current + 1);
    }
}

int main(){ ... }

```

编译并运行：`gcc test2.c && ./a.out`，此时所有的输出均是 `Final counter value: 1000000`。

这里，我们通过引入了 `atomic_load` 和 `atomic_compare_exchange_weak` 两个函数。`atomic_load` 用于原子地加载 `counter` 的值，而 `atomic_compare_exchange_weak` 用于原子地比较并交换：如果 `counter` 的当前值等于预期的值 `current`，则将 `counter` 设置为 `current + 1`；否则，不进行写入操作。这样确保了在更新 `counter` 时不会发生数据竞争。

这只是一种简单的情况，我们需要设计一套更通用的方法来应对数据竞争，实现互斥，这便引入了两种基础的锁：自旋锁与互斥锁，他们能保证同一时刻，只有一个线程能进行操作。

互斥 (Mutual Exclusion) 与同步 (Synchronization)

互斥 (Mutual Exclusion) 与同步 (Synchronization) 是解决并发编程中数据竞争与协调问题的两种核心机制。互斥用于保证对共享资源的独占访问，即同一时刻最多只有一个访问者能对某一资源进行访问。同步则是在互斥的基础上，进一步管理多个线程之间的执行顺序，确保它们能够按照预期的逻辑顺序协作，例如一个线程需要等待另一个线程完成特定任务后才能继续执行。

Rust中的原子类型

Rust 在 `core::sync::atomic` 模块中提供了基础数据类型的原子版本，如 `AtomicBool`、`AtomicUsize`、`AtomicIsize` 等。这些原子类型在正确使用时可以实现线程间的同步更新。在自旋锁实现中，我们使用了 `AtomicBool::compare_exchange` 和 `AtomicBool::store` 操作，这些操作涉及内存序 (Memory Ordering)

2. 自旋锁

2.1 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：

```

// In: 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的锁会与内部资源绑定。在使用纯safe Rust的前提下，用户只能通过获取锁来访问内部数据，在`acquire_lock`函数中，其不断调用`try_acquire_lock`尝试获取锁，如果失败则持续自旋等待。锁的状态判断使用`AtomicBool`原子布尔类型实现。

需要注意的是，获取锁后返回的是`SpinLockGuard<T, G>`，而不是直接返回内部数据的引用。这样做是为了通过Assignment1中提到的Guard模式，在`SpinLockGuard<T, G>`被释放时自动调用`SpinLock::release_lock`来释放锁，从而简化使用并强制释放操作。

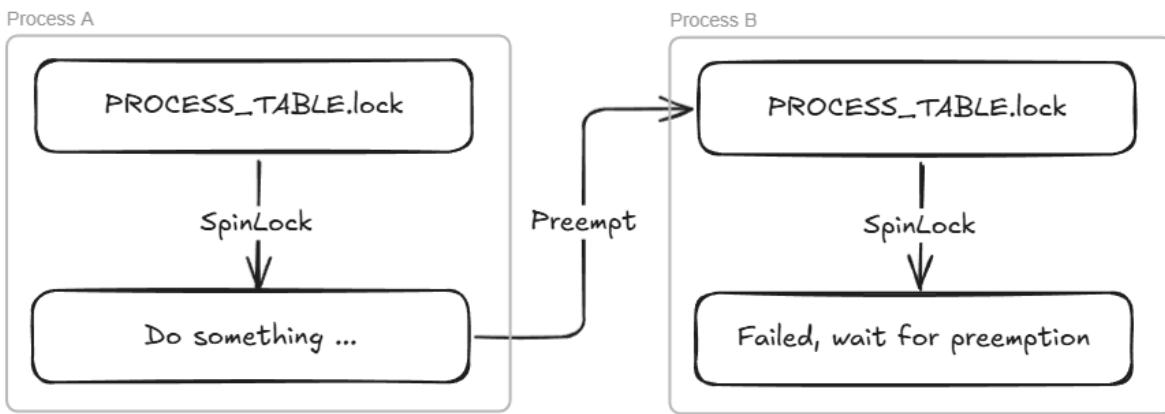
内存序

在多线程编程里，编译器和处理器会为了提升性能对代码进行指令重排。指令重排虽然在单线程环境下不会影响程序的正确性，但在多线程环境中，不同线程对共享内存的访问顺序可能会被打乱，从而导致数据竞争和不一致的问题。内存序列允许开发者对原子操作的内存顺序进行精确控制，从而避免这类问题的发生。

在自旋锁的实现中，使用Acquire和Release内存序的含义是：获取锁的操作总是在其他线程释放锁之后执行。感兴趣的同学可以查阅Rust关于[内存序的详细介绍](#)。

2.2 自旋锁的性能损失和死锁

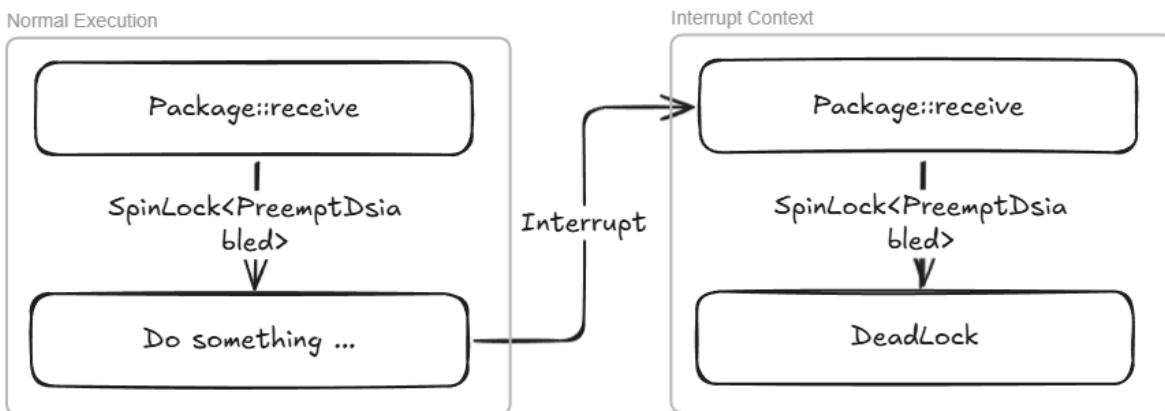
自旋锁的一个主要问题是，当获取锁失败时，系统会持续进行等待操作，导致CPU时间片浪费，降低系统性能。在未启用抢占式调度的情况下，甚至可能导致系统死锁。默认情况下，存在以下导致CPU资源浪费的场景：



在此场景中，进程A在获取全局进程表锁时被抢占式调度打断，切换到进程B。进程B尝试获取同一全局进程表锁但失败，只能持续等待，直到下一次抢占式调度切换回进程A，系统才能继续正常运行。

为避免这种情况，OSTD提供了禁用抢占式调度的 `PreemptDisabled` 机制。该机制在自旋锁获取成功后禁用抢占式调度，确保当前进程完成处理后再重新启用抢占式调度。

然而，禁用抢占式调度仍存在一个重要问题：它不会阻止中断的产生。因此，在涉及中断上下文的代码中，该机制可能失效，仍会导致性能损失和死锁。例如：



在此场景中，相同函数使用同一把锁，但在不同上下文中执行会导致死锁。正常执行流程获取锁后被中断打断，进入中断上下文，中断处理程序再次尝试获取同一把锁，导致中断上下文持续自旋等待。与之前示例不同，在中断上下文中系统会禁用中断，使得没有任何外部事件能够打断执行，系统因此进入死锁状态。

同样，为避免这种情况，OSTD提供了禁用中断的 `LocalIRQDisabled` 机制。该机制在获取锁成功后禁用中断（同时也会禁用抢占式调度，因为抢占式调度依赖于时钟中断）。

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。以下是相关代码：

```
// In: 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>,
}
```

```

// In: ostd/src/sync/guard.rs

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

pub enum PreemptDisabled {}

impl SpinGuardian for PreemptDisabled {
    type Guard = DisabledPreemptGuard;
    type ReadGuard = DisabledPreemptGuard;

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

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

pub enum LocalIrqDisabled {}

impl SpinGuardian for LocalIrqDisabled {
    type Guard = DisabledLocalIrqGuard;
    type ReadGuard = DisabledLocalIrqGuard;

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

```

在这两种禁用机制中，分别使用 `disable_preempt` 和 `disable_local` 函数来禁用抢占式调度和中断。这两个函数会记录禁用次数，以防止在多次嵌套锁操作过程中，抢占式调度或中断被意外启用。

然而，我们不应在所有场景中都使用禁用中断，因为这会导致系统响应速度下降。例如，在执行时间较长的逻辑中禁用中断，可能导致网络驱动无法正常收包，或输入设备驱动无法接收用户输入，使系统在“观感”上卡死。

RISC-V中的禁用中断

在RISC-V架构中，通过 `sstatus.SIE` 位控制中断使能状态：设置为 `1` 时使能中断，设置为 `0` 时禁用中断。`disable_local` 函数内部正是通过操作这一寄存器位来屏蔽中断的。

3. 互斥锁

互斥锁在锁被其他线程占用时会阻塞当前线程，使其挂起。内核可以调度其他线程执行，并在适当时机唤醒挂起的线程。以下是OSTD中互斥锁 `Mutex` 的实现：

```

// In: ostd/src/sync/mutex.rs

pub struct Mutex<T: ?Sized> {

```

```

    lock: AtomicBool,
    queue: WaitQueue,
    val: UnsafeCell<T>,
}

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);
    }
}

```

与自旋锁相比，互斥锁内部多了一个等待队列，用于维护等待唤醒的线程。当锁被释放时，会通知等待队列唤醒其中一个线程。

读写锁

OSTD除了提供 `Mutex` 与 `SpinLock`，还提供了读写锁 `RwMutex` 与 `RwLock`。由于它们的实现比 `Mutex` 或 `SpinLock` 复杂，本节课未作详细讲解。完成上手练习后，可查看 `ostd/src/sync` 目录下的 `rwmutex.rs` 与 `rwlock.rs` 文件进一步学习。

4. 上手练习

信号量是并发编程中的一种重要同步机制，其维护一个计数器，用于控制对共享资源的访问：

- **计数器值**：表示当前可用的资源数量
- **P操作（acquire）**：当计数器值大于0时，线程可以获取资源，计数器减1；否则线程阻塞等待
- **V操作（release）**：释放资源，计数器加1，并唤醒一个等待的线程

与互斥锁只能有一个线程访问不同，信号量允许多个线程同时访问资源（当计数器值大于1时），可以用于解决生产者-消费者等经典并发问题。

请参考Mutex与MutexGuard的实现，利用spinLock<usize, LocalIrqDisabled>完成信号量的实现
(文件路径: `src/sem.rs` 即可) , 包含Semaphore与SemaphoreGuard, 其中SemaphoreGuard用于
自动回收资源并唤醒线程, Semaphore对外提供三个接口:

- `new(count:usize) -> Semaphore`
- `acquire(&self) -> SemaphoreGuard`
- `try_acquire(&self) -> Option<SemaphoreGuard>`

测试脚本: `cargo osdk test --target-arch=riscv64`, 以下是简单的测试程序:

```
#[cfg(ktest)]
pub mod test {
    use ostd::prelude::ktest;

#[ktest]
pub fn basic_test() {
    use super::Semaphore;

    let sem = Semaphore::new(2);

    {
        let _g1 = sem.lock();
        let _g2 = sem.lock();
        assert!(sem.try_lock().is_none());
    }

    {
        let _g3 = sem.lock();
        assert!(sem.try_lock().is_some());
    }
}
```