8. 内核并发与同步

授课教师：游伟 副教授

授课时间：周二14:00 – 15:30（立德楼807）

课程主页：https://www.youwei.site/course/kernel

目录

8.1 基本概念

- 临界区
- 竞争状态
- 同步
- 共享队列、加锁
- 待保护对象
- 死锁

临界区和竞争状态

- 什么是临界区（*critical regions*）？
  - 就是访问和操作共享数据的代码段，这段代码必须被**原子地**执行
- 什么是竞争状态?
  - 多个内核任务同时访问同一临界区
- 什么是同步?
  - 避免并发和防止竞争状态称为*同步*（*synchronization*）

临界区举例

- 考虑一个非常简单的共享资源的例子：一个全局整型变量和一个简单的临界区，其中的操作仅仅是将整型变量的值增加1：

```
i++
```

- 该操作可以转化成下面三条机器指令序列：
  - (1) 得到当前变量i的值并拷贝到一个寄存器中
  - (2)将寄存器中的值加1
  - (3) 把i的新值写回到内存中

临界区举例

可能的实际执行结果：

内核任务1	内核任务2
获得 i(1)	---
---	获得 i(1)
增加 i(1->2)	---
---	增加 i(1->2)
写回 i(2)	---
---	写回 i(2)

期望的结果

内核任务1	内核任务2
获得i(1)	---
增加 i(1->2)	---
写回 i(2)	---
	获得 i(2)
	增加 i(2->3)
	写回 i(3)

共享队列和加锁

- 当共享资源是一个复杂的数据结构时，竞争状态往往会使该数据结构遭到破坏。
- 对于这种情况，锁机制可以避免竞争状态正如门锁和门一样，门后的房间可想象成一个临界区。
- 在一个指定时间内，房间里只能有个一个内核任务存在，当一个任务进入房间后，它会锁住身后的房门；当它结束对共享数据的操作后，就会走出房间，打开门锁。如果另一个任务在房门上锁时来了,那么它就必须等待房间内的任务出来并打开门锁后，才能进入房间。

共享队列和加锁

- 任何要访问队列的代码首先都需要占住相应的锁，这样该锁就能阻止来自其它内核任务的并发访问：

任务 1

试图锁定队列

成功：获得锁

访问队列...

为队列解除锁

...

任务2

试图锁定队列

失败：等待...

等待...

等待...

成功：获得锁

访问队列...

为队列解除锁

确定保护对象

- 找出哪些数据需要保护是关键所在
- 内核任务的局部数据仅仅被它本身访问，显然不需要保护
- 如果数据只会被特定的进程访问，也不需加锁
- **大多数内核数据结构都需要加锁**：若有其它内核任务可以访问这些数据，那么就给这些数据加上某种形式的锁；若任何其它东西能看到它，那么就要锁住它

死　锁

- 死锁产生的条件：有一个或多个并发执行的内核任务和一个或多个资源，每个任务都在等待其中的一个资源，但所有的资源都已经被占用。所有任务都在相互等待，但它们永远不会释放已经占有的资源，于是任何任务都无法继续
- 典型的死锁：
  - 四路交通堵塞
  - 自死锁：一个执行任务试图去获得一个自己已经持有的锁

死锁的避免

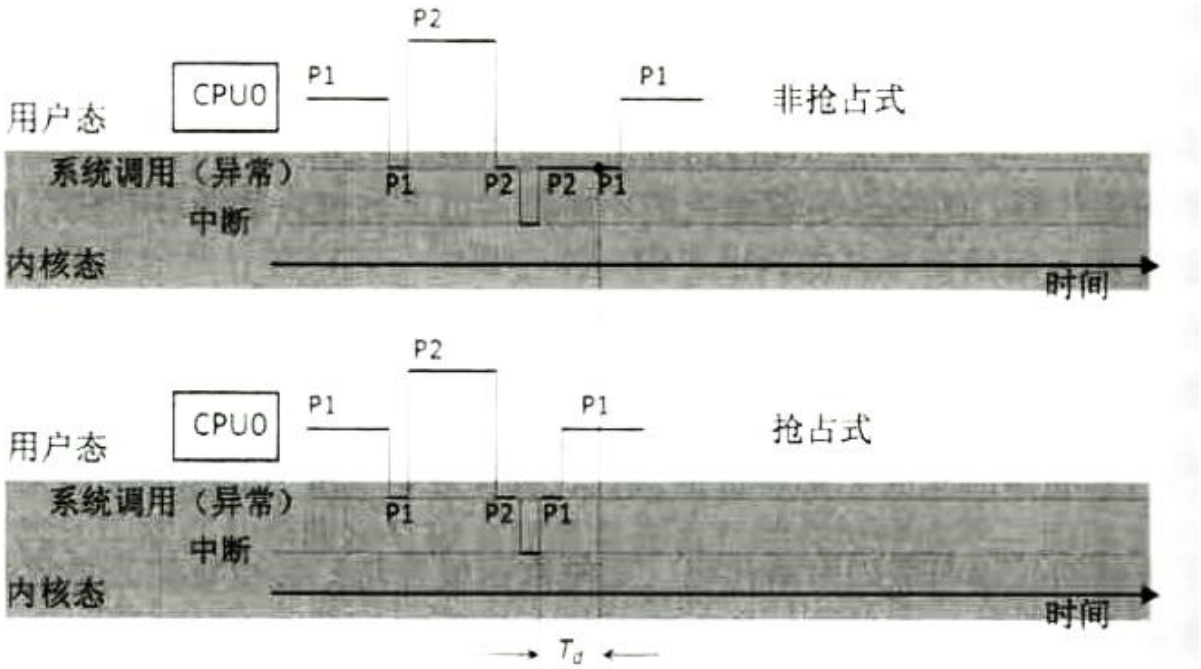
- 加锁的顺序是关键。使用嵌套的锁时必须保证以相同的顺序获取锁，这样可以阻止致命拥抱类型的死锁
- 防止发生饥饿
- 不要重复请求同一个锁。
- 越复杂的加锁方案越有可能造成死锁，因此设计应力求简单

并发执行的原因

- 中断——中断几乎可以在任何时刻异步发生，也可能随时打断正在执行的代码。
- 内核抢占——若内核具有抢占性，内核中的任务就可能会被另一任务抢占
- 睡眠及与用户空间的同步——在内核执行的进程可能会睡眠，这将唤醒调度程序，导致调度一个新的用户进程执行
- 对称多处理——两个或多个处理器可以同时执行代码

8.2 内核同步的需求

- 内核抢占：处于内核态的进程，是否允许被切换
  - 非抢占式内核：除非主动放弃CPU或返回用户态，高优先级进程才可能被调度
  - 抢占式内核：高优先级进程可以抢占处于内核态的低优先级进程
  - 抢占式内核让进程调度切换更及时，降低紧急任务被延迟的程度，提升实时性
  - 内核编译选项CONFIG_PREEMPT，控制内核抢占功能是否开启



8.2 内核同步的需求

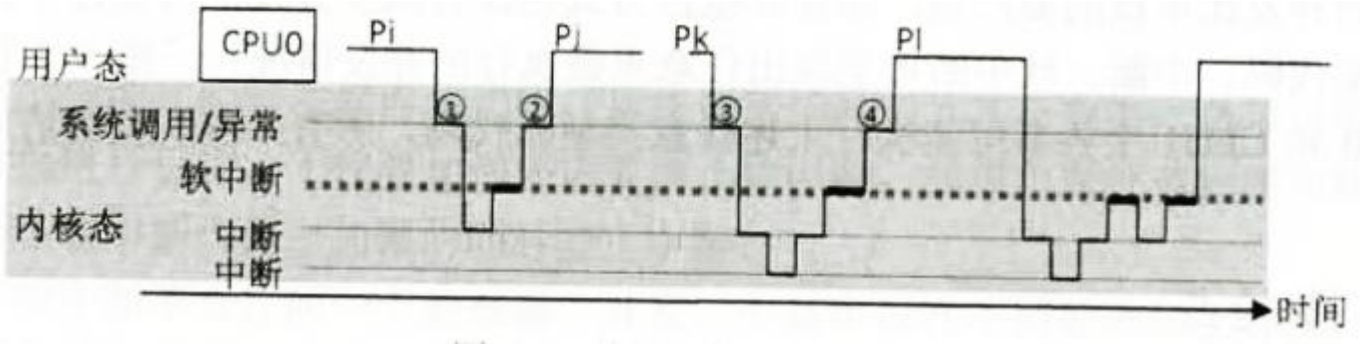
- 对称多处理器（SMP）调度
  - 目标：将进程迁移到合适的处理器上，并且保持各个处理器的负载均衡
  - 公平共享：CPU负载需要公平共享，不能出现某个CPU空闲，造成资源浪费
  - 可设置进程与CPU亲和性：可为某些类型的进程与指定的处理器设置亲和性
  - 进程迁移：Linux内核可以将进程在不同的CPU处理器之间进行迁移

- 内核并发情形

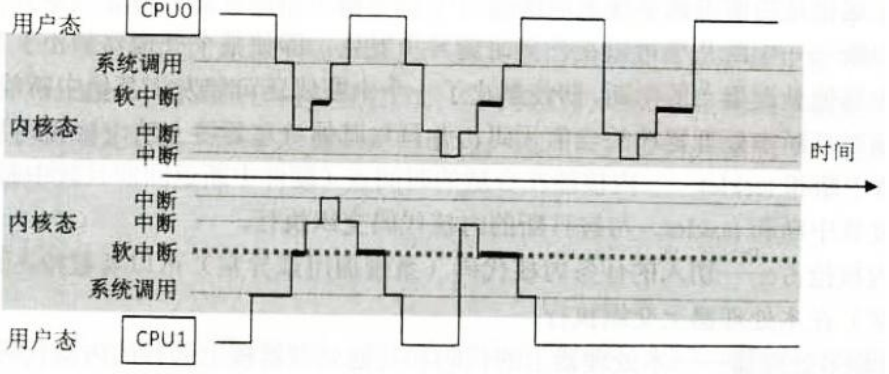
处理器 / 内核	单核	多核
非抢占	单核-非抢占	多核-非抢占
抢占	单核-抢占	多核-抢占

单核环境下的并发

- 非抢占式
  - 系统调用不会交织运行，无需考虑不同系统调用之间的并发同步
  - 异常处理不会交织运行，无需考虑不同异常处理之间的并发同步
  - 中断处理/可延迟函数与系统调用/异常处理会交织运行，需考虑并发同步
- 抢占式
  - 系统调用/异常处理会交织运行，需要考虑并发同步
  - 其余并发情况与单核非抢占式场景相同



多核环境下的并发



- 在同一个CPU上的抢占关系
  - 中断处理：可以抢占其它中断处理/系统调用/异常处理/可延迟函数，在中断退出前触发可延迟函数
  - 可延迟函数：不能抢占中断处理/其它可延迟函数
  - 异常处理：不能抢占中断处理/可延迟函数/其它异常处理
  - 系统调用：若允许内核抢占，则切入进程的系统调用可抢占当前进程的系统调用；否则，系统调用不能抢占以上任何一种内核代码
- 在多个CPU上的并发关系
  - 中断处理：不同CPU的中断处理可并行执行，但共享硬件资源时需同步（如设备寄存器访问）
  - 可延迟函数：不同CPU的可延迟函数可并行执行，通过每CPU变量（per-CPU data）减少竞争
  - 异常处理：不同CPU的异常处理可并行执行，但访问共享内存时需同步
  - 系统调用：不同CPU的系统调用可并发执行，通过RCU/原子操作/自旋锁等机制保护共享资源。

8.3 内核共享变量的保护

- 被保护的对象：实现临界区并不是为了保护代码执行本身，而是要保护被代码操作的数据对象。因此内核同步问题其实就是:
  1. 首先是找出哪些数据对象在什么情况下会被哪些内核代码共享访问;
  2. 其次就是采用哪种方法来实现对它们的互斥访问;
  3. 最后要在保证互斥访问的前提下如何避免死锁的产生。
- 那么到底什么数据对象需要加锁保护呢?实际上大多数内核数据结构都需要加锁。几乎所有的内核全局变量和共享数据都需要某种形式的同步方法，除非可以确定该数据的访不会有其他执行代码访问，例如，执行线程的局部自动变量(内核函数中的、属于各进程内核栈。
- 具体的判断方法如下---如果有其他的执行路径可以访问该数据结构，则需要加上某钟锁。编写内核代码时需要考虑下面的问题。
  1. 是否全局变量，其他执行路径能否访问到它?
  2. 该数据是否会在进程上下文和中断上下文中共享(例如系统调用和中断处理函数之间共享)?是否会在两个不同的中断处理函数之间共享?
  3. 当前内核代码是否会被抢占、抢占它的代码是否会访问该数据?如果内核配置时使用CONFIG_SMP 选项支持 SMP，那么内核代码中将加入许多锁,否则代码会简洁很多。同样编译内核时如果用CONFIG_PREEMPT允许内核抢占的话也会如入很多锁相关的代码。
- 加锁是为了保护数据，而不是为了代码串行执行。

8.3 内核共享变量的保护

- 根据并发访问情况的不同，相应的共享数据对象所需要的内核保护手段也不一样。下面将共享数据根据并发代码的不同而划分不同情况，并给出应该完成的保护措施。以下情况考虑的都是允许内核抢占的系统。本节提到的自旋锁还包括读写锁和顺序锁等。

1. 仅系统调用和异常
   - 1)单处理器：此时的 CPU运行在内核态为用户进程提供服务。此竞争条件可通过信号量避免，无须关闭中断等更高级别的措施。
   - 2)多处理器：与单处理器环境时相同,使用信号量即可。但是在允许内核抢占的系统上访问 per-CPU变量时，还要禁用抢占，以免被抢占后的内核代码也访问这个每 CPU变量，也可以防止本进程被迁移到其他处理器上。
2. 仅中断
   - 1)单处理器
     - (1)只有一种中断的访问时，中断都相对自己串行地执行，无须同步。
     - (2)多种中断间的并发访问时，要关中断。
   - 2)多处理器
     - (1)只有一种中断的访问时，中断都相对自己串行地执行，无须保护。
     - (2)多种中断可访问时，要关中断阻断本处理器上其他中断代码的执行，并加上自旋锁阻止其他处理器上的中断代码的访问。
3. 仅可延迟函数(软中断和 tasklet)
   - 1)单处理器（UP）：无需保护。软中断在单核上串行执行；tasklet在单核上独占运行。
   - 2)多处理器（SMP）：
     - 软中断：加自旋锁（防止跨CPU软中断竞争）。
     - 单一tasklet：无需保护（同一tasklet仅在单CPU运行）。
     - 多tasklet：加自旋锁（防止跨CPU不同tasklet并发）。

8.3 内核共享变量的保护

4. 系统调用/异常和中断

- 1)单处理器（UP）：
  - 系统调用/异常侧：关闭中断。
  - 中断侧：无需保护（中断不可被系统调用抢占）。
- 2)多处理器（SMP）：
  - 系统调用/异常侧：关闭本地中断 + 自旋锁。
  - 中断侧：自旋锁（或原子操作/down_trylock 替代）。

5. 系统调用/异常和可延迟函数

- 1)单处理器（UP）：
  - 系统调用/异常侧：禁用可延迟函数（或关闭中断）。
  - 可延迟函数侧：无需保护（不可被抢占）。
- 2）多处理器（SMP）：
  - 系统调用/异常侧：禁用可延迟函数 + 自旋锁。
  - 可延迟函数侧：加自旋锁。

8.3 内核共享变量的保护

6. 中断和可延迟函数

- 1）单处理器（UP）：
  - 可延迟函数侧：关闭中断。
  - 中断侧：无需保护（不可被可延迟函数抢占）。
- 2）多处理器（SMP）：
  - 可延迟函数侧：关闭中断 + 自旋锁。
  - 中断侧：加自旋锁。

7. 系统调用/异常、中断和可延迟函数

- 1）单处理器（UP）：
  - 系统调用/异常侧：关闭中断。
  - 可延迟函数侧：关闭中断。
  - 中断侧：无需保护。
- 2）多处理器（SMP）：
  - 系统调用/异常侧：关闭中断 + 自旋锁。
  - 可延迟函数侧：关闭中断 + 自旋锁。
  - 中断侧：加自旋锁。

8.3 内核共享变量的保护

8. 无须同步的情况，内核通过以下规则降低同步需求：

① 中断串行化：中断处理期间禁止同类型中断（通过PIC禁用IRQ线）。

② 不可抢占性：中断、软中断和tasklet不可被抢占或阻塞。

③ 执行隔离性：

- 中断不会被软中断/tasklet打断。
- 软中断和tasklet在同一CPU上不交错执行。
- Tasklet仅在单一CPU运行，不会跨CPU并发。

8.4 内核同步方法

- 原子操作
- 自旋锁、读写锁、顺序锁
- RCU机制
- 信号量、读写信号量、互斥量
- 等待队列
- 每CPU变量

8.4 内核同步方法——实践场景

- 场景设计：虚拟网络设备驱动的并发控制模块
  - 假设我们为一块虚拟多队列网卡（如 vnic）编写内核驱动，该设备需处理高并发数据包收发、动态配置更新、统计信息维护等任务。

场景	同步方法	选择理由
中断处理队列状态	自旋锁	中断上下文不可睡眠
统计信息读多写少	读写锁	减少读者竞争
实时链路状态高频读取	顺序锁	写者优先，避免读者阻塞写者
用户配置互斥访问	互斥量	简单互斥，自动处理睡眠
批量任务并发限制	信号量	控制资源池大小
历史记录无锁读取	RCU	读者零开销，写者延迟释放
固件升级与版本读取	读写信号量	细粒度读写控制，支持独占升级与并发读版本

原子操作

- 原子操作可以保证指令以原子的方式被执行，也就是过程不被打断。
  - atomic_t实际上就是一个int类型的counter。
  - 之所以定义为一个特殊类型，是为了让编译器进行严格的类型检查，防止原子类型变量被当作普通类型误操作。

```
typedef struct {
        int counter;
} atomic_t;
```

原子操作

- 操作API

函数	描述
ATOMIC_INIT(int i)	定义原子变量的时候对其初始化。
int atomic_read(atomic_t *v)	读取 v 的值，并且返回。
void atomic_set(atomic_t *v, int i)	向 v 写入 i 值。
void atomic_add(int i, atomic_t *v)	给 v 加上 i 值。
void atomic_sub(int i, atomic_t *v)	从 v 减去 i 值。
void atomic_inc(atomic_t *v)	给 v 加 1，也就是自增。
void atomic_dec(atomic_t *v)	从 v 减 1，也就是自减
int atomic_dec_return(atomic_t *v)	从 v 减 1，并且返回 v 的值。
int atomic_inc_return(atomic_t *v)	给 v 加 1，并且返回 v 的值。
int atomic_sub_and_test(int i, atomic_t *v)	从 v 减 i，如果结果为 0 就返回真，否则返回假
int atomic_dec_and_test(atomic_t *v)	从 v 减 1，如果结果为 0 就返回真，否则返回假
int atomic_inc_and_test(atomic_t *v)	给 v 加 1，如果结果为 0 就返回真，否则返回假
int atomic_add_negative(int i, atomic_t *v)	给 v 加 i，如果结果为负就返回真，否则返回假

表 47.2.2.1 原子整形操作 API 函数表

- 适用场景：整数原子操作最常见的用途就是实现计数器，使用复杂的锁机制来保护一个单纯的计数器是很笨拙的。

原子操作——实践场景

- 场景：网卡全局使能标志位（vnic_enabled）的快速开关。
- 理由：原子操作适用于单变量无竞争更新，避免锁开销。

```
static atomic_t vnic_enabled = ATOMIC_INIT(1);  // 初始化为启用状态

// 读标志：无锁快速访问，适用于高频检查
int is_vnic_enabled(void) {
    return atomic_read(&vnic_enabled);  // 直接读取原子变量
}

// 写标志：原子性关闭设备，避免中间状态暴露
void disable_vnic(void) {
    atomic_set(&vnic_enabled, 0);       // 原子写操作
    smp_mb();                           // 内存屏障确保修改全局可见
}
```

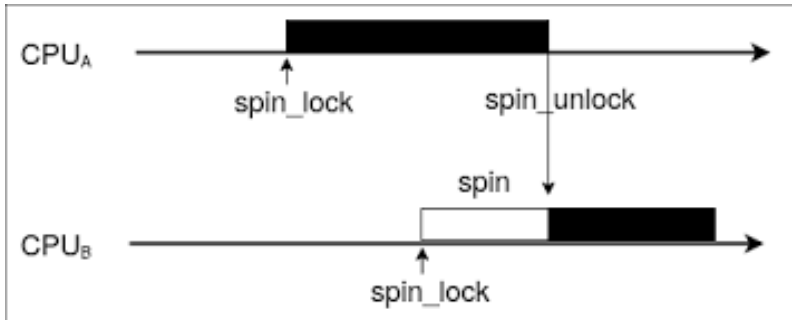
自旋锁

- 概念
  - 自旋锁是为防止多处理器并发而引入的一种锁，在内核中被大量应用于中断处理等部分
  - 设计自旋锁的初衷是在短时间内进行轻量级的锁定，所以自旋锁不应该被持有时间过长

```c
/* Non PREEMPT_RT kernels map spinlock to raw_spinlock */
typedef struct spinlock {
        union {
                struct raw_spinlock rlock;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
                struct {
                        u8 __padding[LOCK_PADSIZE];
                        struct lockdep_map dep_map;
                };
#endif
        };
} spinlock_t;
```

- 操作API



自旋锁

- spin_lock函数实现

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
        preempt_disable();
        spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
        LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}
```

- 通过原子操作获取锁，并在多核/高并发场景下确保临界区互斥访问。

具体流程如下：

- 禁用内核抢占（preempt_disable()）：禁止当前进程被其他高优先级进程抢占，避免持有锁期间发生调度，导致其他进程因无法获取锁而长时间空转（尤其在单核可抢占内核中至关重要）。
- 锁调试跟踪（spin_acquire(&lock->dep_map, ...)）：在内核配置锁依赖检测（CONFIG_LOCKDEP）时，记录锁的获取位置（_RET_IP_为返回地址），用于死锁分析和调试。
- 实际锁获取操作（LOCK_CONTENDED宏）：
  - 先尝试快速获取锁（do_raw_spin_trylock），若成功则直接进入临界区。
  - 若失败则进入忙等待（do_raw_spin_lock），通过原子指令（如x86的LOCK前缀或ARM的LDREX/STREX）循环检测锁状态，直到锁被释放。

自旋锁

- 自旋锁不会导致睡眠，自旋锁是一种忙等锁，不会导致睡眠，所以可以用在中断上下文中
- 持有自旋锁不能睡眠
  - ① 如果是在中断上下文中持有自旋锁，不能睡眠是应有之义②如果是在进程上下文中持有自旋锁，也是不能睡眠的。因为自旋锁会关抢占，该进程一旦睡眠，该CPU上就无法再调度其他任务执。行，只能响应中断
- 在实际使用中，在关抢占的情况下调用schedule，内核运行会报bug
- 自旋锁变种分析：自旋锁变种使用的原则就是谁抢CPU就关谁
  - ① 何时关中断：如果除了当前的内核任务，还有中断处理程序会访问临界区，则需要关中断。其中需要注意的是，即使是在单核CPU中也需要关中断因为如果在进程上下文中持有自旋锁但是不关中断，如果当前CPU核被中断抢占并目该中断也要占有该自旋锁，则会导致死锁
  - ② 何时关底半部：同样地，如果还有中断底半部会访问临界区，则关底半部，比如tasklet和timer。而且由于这些底半部机制是在软中断环境中运行的也存在和上面关中断一样的问题，即使是单核CPU也要关底半部说明:其实上面2种场景中，即使是SMP也需要关中断/底半部，因为只要是在当前持有自旋锁的CPU上会发生中断或中断底半部的抢占，且该中断或中断底半部也需要获取该自旋锁，就会导致死锁如果只是两个线程之前互斥则无需关中断，因为互斥锁会先关闭抢占

自旋锁

- 自旋锁不同版本的使用说明
  - ① spin_lock用于阻止在不同CPU上的执行单元对共享资源的同时访问，以及不同进程上下文互相抢占导致的对共享资源的非同步访河(这是前一种问题在单CPU环境中的场景)
  - ② spin_lock_irg和spin_lock_bh是为了阻止在同一CPU上中断或软中断对共享资源的非同步访问
- 使用场景辨析
  - ① 如果被保护的共享资源只在进程上下文和软中断上下文中被访问，那么当在进程上下文访问共享资源时，可能被软中断打断，从而可能进入软中断上下文来访问被保护的共享资源。因此对于这种情况，对共享资源最好使用spin_lock_bh进行保护。
    - 说明1:在多CPU场景中，软中断上下文不一定和进程在同一CPU上运行(因为spin_lock_bh只能关闭本CPU的底半部)，所以软中断中也应该调用spin lock进行互斥
    - 说明2:软中断上下文包括了tasklet、timer等
  - ②如果被保护的共享资源只在两个或多个tasklet或timer上下文访问，那么仅需要使用spin_lock进行保护，不必使用 bh版本因为当taskle或timer运行时，不可能有其他tasklet或timer在当前CPU上运行说明:根据之前对软中断的实现分析，在某个CPU上的软中断的执行是顺序的，即软中断的触发者只是记录请求，实际执行时遍历请求列表
  - ③如果被保护的共享资源只在一个软中断上下文访问，但是是多CPU环境，那么需要使用spin_lock进行保护，因为同样的软中断可以同时在不同的CPU上运行
  - ④ 如果被保护的共享资源在软中断或进程上下文或中断顶半部上下文访问，那么在软中断或进程上下文访问期间，可能被中断顶半部打断。因此，在进程或软中断上下文需要使用spin_lock__irg进行保护。
    - 说明:使用spin_lock_irq还是spin_lock_irgsave需要视情况而定，二者差异在于 irgsave版本会保留当前的中断关闭状态，该特性可用于中断的嵌套关闭。
    - 如果可以确定在对共享资源访问前中断是使能的，那么使用spin_lock_irg会比spin_lock_irgsave更快一些。

自旋锁——实践场景

- 场景：中断处理函数中更新硬件队列状态。
- 理由：中断上下文禁止睡眠，自旋锁通过忙等待保证原子性。

```
DEFINE_SPINLOCK(rx_queue_lock);  // 定义自旋锁

// 中断处理函数（非睡眠上下文）
irqreturn_t vnic_rx_isr(int irq, void *dev) {
    unsigned long flags;
    spin_lock_irqsave(&rx_queue_lock, flags);  // 关中断+获取锁
    // 更新硬件描述符队列（临界区）
    update_hw_rx_descriptors();
    spin_unlock_irqrestore(&rx_queue_lock, flags);  // 恢复中断+释放锁
    return IRQ_HANDLED;
}
```

读写锁

- 概念
  - ① 读写锁是一种特殊的自旋锁，将对共享资源的访问分为读者和写者② 读写锁的概念与读写信号量类似，只是读写锁基于自旋锁，因此不会导致睡眠
  - ③) 在读写锁保持期间也是抢占失效的
  - ④如果读写锁当前没有读者也没有写者，那么写者可以立刻获得读写锁;否则写者必须自旋，直到没有任何读者和写者⑤ 如果读写锁没有写者，那么读者可以立刻获得读写锁;否则读者必须自旋，直到写者释放该读写锁说明:读写锁相对于自旋锁能提高并发性，因为在多处理器系统中，他同时允许多个读者访问共享资源，最大可能的读者数为实际的逻辑CPU个数

```
typedef struct {
        arch_rwlock_t raw_lock;
#ifdef CONFIG_DEBUG_SPINLOCK
        unsigned int magic, owner_cpu;
        void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map dep_map;
#endif
} rwlock_t;
```

读写锁

1	rwlock_init(lock)	静态声明并初始化读写锁
2	read_lock (lock)	获得读者锁
3	read_unlock(lock)	获得写者锁
4	read_trylock(lock)	读者尝试获得读者锁，不能获得则立刻返回
5	write_lock(lock)	获得写者锁
6	write_unlock(lock)	释放写者锁
7	write_trylock（lock）	尝试获得写者锁，不能获得则立刻返回

8	read_lock_irqsave(lock， flags) /read_unlock_irqrestore(lock， flags)	获得读者锁并关中断，将标志寄存器内容写入 flags，解锁反之
9	read_lock_irq/read_unlock_irq	获得读者锁并关中断， 解锁反之
10	read_lock_bh/read_unlock_bh	获得读者锁并关软中断，解锁反之
11	write_lock_irqsave(lock， flags) /write_unlock_irqrestore(lock， flags)	获得写者锁并关中断，将标志寄存器内容写入 flags，解锁反之
12	write_lock_irq/write_unlock_irq	获得写者锁并关中断， 解锁反之
13	write_lock_bh/write_unlock_bh	获得写者锁并关软中断， 解锁反之
14	write_trylock_irqsave	将标志寄存器内容写入 flags 并尝试获得写者锁，成功则关中断，失败则恢复寄存器内容立刻返回

读写锁——实践场景

- 场景：统计信息（如 struct vnic_stats）的读多写少场景。
- 理由：读写锁减少读者竞争。

```
rwlock_t stats_lock = __RW_LOCK_UNLOCKED(stats_lock);  // 初始化读写锁
struct vnic_stats stats;                               // 统计数据结构

// 写者：更新统计信息（低频）
void update_stats(struct packet_metadata *pkt) {
    write_lock(&stats_lock);          // 获取写锁（独占）
    stats.tx_bytes += pkt->len;
    stats.tx_packets++;
    write_unlock(&stats_lock);
}

// 读者：用户态查询统计（高频）
void read_stats_user(void) {
    read_lock(&stats_lock);           // 获取读锁（共享）
    copy_to_user(user_buf, &stats, sizeof(stats));
    read_unlock(&stats_lock);
}
```

顺序锁

- 概念
  - ① 顺序锁是对读写锁的一种优化，在顺序锁中，读者绝不会被写者阻塞，也就是说，读者可以在写者对顺序锁保护的共享资源进行写操作时进行读操作，而不必等待写者完成写操作
  - ② 写者也不需要等待所有读者完成读操作才进行写操作
  - ③ 写者与写者之间仍然是互斥的
  - 说明1:由于读写之间不互斥，所以如果读者在读操作期间，写者已经发生了写操作，那么读者必须重新读取数据，以确保得到的数据是完整的
  - 说明2:如果读写同时进行的概率很小，顺序锁的性能是非常好的。而且顺序锁允许读写同时进行，更大地提高了并发性顺序锁的一个典型应用在于时钟系统
  - 说明3:顺序锁有一个限制，他所保护的共享资源不能包含指针。因为写者可能使得指针失效，如果读者正要访问该指针，将导致OOPS

```
typedef struct {
        /*
         * Make sure that readers don'
         * seqcount_spinlock_t instead
         */
        seqcount_spinlock_t seqcount;
        spinlock_t lock;
} seqlock_t;
```

- 数据类型
  - ① 从结构上看，顺序锁也依赖自旋锁
  - ② seqcount用于同步写者访问的顺序以更新读者访问(即被读者用于判断从此读取过程中是否发生过数据被写者访问)
  - ③ 自旋锁用于实现写操作之间的互斥，读者访问不受限制

顺序锁

1	seqlock_init(lock) /DEFINE_SEQLOCK	声明并初始化顺序锁
2	write_seqlock	写者获得顺序锁
3	write_sequnlock	写者释放顺序锁
4	read_seqbegin	读取 seqlock 当前的写操作发生次数
5	read_seqretry	判断是否与当前 seqlock 的 seqcount 一致
6	read_seqlock_excl /read_sequnlock_excl	一般读者无需获得锁，但内核也提供了读者获得锁/释放锁的接口。

顺序锁

- read_seqcount_begin返回当前seqlock的segcount值，在读取完成后，需要调用read_seqcount_retry判断读者读完后的segcount值是否与读之前一致。一致则结束;不一致则说明有写操作正在或已经执行，需要重新读取一次以更新数据

```
void ktime_get_ts64(struct timespec64 *ts)
{
        struct timekeeper *tk = &tk_core.timekeeper;
        struct timespec64 tomono;
        unsigned int seq;
        u64 nsec;

        WARN_ON(timekeeping_suspended);

        do {
                seq = read_seqcount_begin(&tk_core.seq);
                ts->tv_sec = tk->xtime_sec;
                nsec = timekeeping_get_ns(&tk->tkr_mono);
                tomono = tk->wall_to_monotonic;

        } while (read_seqcount_retry(&tk_core.seq, seq));

        ts->tv_sec += tomono.tv_sec;
        ts->tv_nsec = 0;
        timespec64_add_ns(ts, nsec + tomono.tv_nsec);
}
EXPORT_SYMBOL_GPL(ktime_get_ts64);
```

顺序锁——实践场景

- 场景：实时链路状态（如 link_speed）的高频读取、低频更新。
- 理由：顺序锁避免写者饥饿。

```
seqlock_t link_lock = __SEQLOCK_UNLOCKED(link_lock);  // 初始化顺序锁
u32 link_speed;                                       // 链路速率
 (Mbps)

// 写者：更新链路速率（低频）
void update_link_speed(u32 speed) {
    write_seqlock(&link_lock);  // 获取顺序锁写版本
    link_speed = speed;
    write_sequnlock(&link_lock);
}

// 读者：无锁读取（高频）
u32 get_current_speed(void) {
    unsigned seq;
    u32 speed;
    do {
        seq = read_seqbegin(&link_lock);  // 读取序列号
        speed = link_speed;               // 无锁读数据
    } while (read_seqretry(&link_lock, seq));  // 检查序列号是否变化
    return speed;
}
```

信号量

- 概念
  - ① Linux中的信号量是一种睡眠锁
  - ② 若有一个任务试图获得一个已被持有的信号量，信号量会将其推入等待队列，然后让其睡眠，这时处理器获得自由去执行其他代码。当持有信号量的进程将信号量释放后，在等待队列中的一个任务将被唤醒，从而可以获得这个信号量
  - 说明:信号量与互斥量(mutex)都是基于spinlock实现的，都建立在spinlock实现的执行同步上

```
struct semaphore {
        raw_spinlock_t          lock;
        unsigned int            count;
        struct list_head        wait_list;
};
```

信号量

- 操作API

函数	描述
down(struct semaphore *);	获取信号量，如果不可获取，则进入不可中断睡眠状态（目前已经不建议使用）。
down_interruptible(struct semaphore *);	获取信号量，如果不可获取，则进入可中断睡眠状态
down_killable(struct semaphore *);	获取信号量，如果不可获取，则进入可被致命信号中断的睡眠状态。
down_trylock(struct semaphore *);	尝试获取信号量，如果不能获取，则立刻返回
down_timeout(struct semaphore *, long jiffies);	在给定时间（jiffies）内获取信号量，如果不能够获取，则返回。
up(struct semaphore *);	释放信号量。

Monitor Implementation – Condition Variables

For each condition variable **c**, we have:

```
semaphore c_sem; // (initially = 0)
int c_count = 0;
```

The operation wait (c) can be implemented as:

```
c_count++;
if (next_count > 0) up(next);
else up(mutex);
down(c_sem);
c_count--;
```

The operation signal (c) can be implemented as:

```
if (c_count > 0) {
   next_count++; up(c_sem);
   down(next); next_count--;
}
```

Mentimeter Signal Q7: 36 33 16

信号量

- down 函数是 Linux 内核中用于获取信号量的遗留接口，其核心逻辑是在信号量不可用时将调用进程挂起（睡眠）直到资源释放，但现已不推荐使用（建议改用可响应中断的 down_interruptible 或 down_killable）。具体实现分为三步：
  - 睡眠检查：通过 might_sleep() 确保当前上下文允许进程睡眠（如在原子上下文中调用会触发警告）。
  - 原子操作信号量：关中断并持自旋锁（raw_spin_lock_irqsave）保护临界区：
    - 若信号量计数 count > 0，直接减1并释放锁。
    - 若资源耗尽（count ≤ 0），调用 __down(sem) 将进程加入等待队列并触发调度切换。
  - 恢复中断与锁状态：通过 raw_spin_unlock_irqrestore 释放自旋锁并恢复中断。

```
void __sched down(struct semaphore *sem)
{
        unsigned long flags;

        might_sleep();
        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                __down(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(down);
```

信号量

- up 函数是 Linux 内核中用于释放信号量的标准接口，其核心逻辑是通过原子操作唤醒等待进程或直接增加信号量计数，支持任意上下文调用（包括中断处理程序），且不要求调用者必须持有信号量。具体实现分为两步：
  - 原子操作信号量：关中断并持自旋锁（raw_spin_lock_irqsave）保护临界区：
    - 若等待队列为空（list_empty(&sem->wait_list)），直接增加信号量计数 count++。
    - 若存在等待任务，调用 __up(sem) 唤醒队列中的第一个任务（通过调度器切换使其获取信号量）。
  - 恢复中断与锁状态：通过 raw_spin_unlock_irqrestore 释放自旋锁并恢复中断。

```
void __sched up(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(list_empty(&sem->wait_list)))
                sem->count++;
        else
                __up(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(up);
```

信号量

- 因为信号量会导致睡眠，所以只能用在进程上下文中，适用于锁会被长时间持有的情况
  - 说明1：Linux内核中并没有提供在中断上下文中可以长期持有的锁，因为在Linux内核的设计意图中，中断处理就必须尽快，所以没有中断中可以长时间持有锁的机制
  - 说明2：信号量与自旋锁的比较

需求	建议的加锁方法
低开销加锁	优先使用自旋锁
短期锁定	优先使用自旋锁
长期加锁	优先使用信号量
中断上下文中加锁	使用自旋锁
持有锁时需要睡眠、调度	使用信号量

互斥量

- 互斥锁的出现时间晚于信号量，互斥锁可以看作是对互斥信号量(count为1)的改进，互斥锁的接口更简单、性能更好,并且互斥锁有着更严格的约束和使用场景:
  - 任意时刻，互斥锁最多只能有一个持有者。
  - 谁上的锁，谁就负责解锁。不能在一个上下文中加锁，在另一个上下文中解锁。
  - 一个进程持有互斥锁时不能退出
  - mutex_lock会使任务进入睡眠，所以不能在中断上下文或者下半部(这里的下半部应该是不包括工作队列的)中调用
  - 互斥锁和信号量之间选择时，能用互斥锁尽量用互斥锁，用不了互斥锁再考虑信号量。
  - 互斥锁使用struct mutex_t类型来表示，下面是内核提供的互斥锁操作函数:

操作函数	功能描述
mutex_lock(struct mutex *)	Locks the given mutex; sleeps if the lock is unavailable
mutex_unlock(struct mutex *)	Unlocks the given mutex
mutex_trylock(struct mutex *)	Tries to acquire the given mutex; returns one if successful and the lock is acquired and zero otherwise
mutex_is_locked (struct mutex *)	Returns one if the lock is locked and zero otherwise

互斥量

下面是一个use case

```
//创建互斥锁，并初始化
struct mutex mx
mutex_init(&mx)

//上锁
mutext_lock(&mx);

/* 临界区 Critical Region */

//解锁
mutext_unlock(&mx);
```

信号量+互斥量——实践场景

```c
DEFINE_SEMAPHORE(bulk_config_sem);  // 信号量（初始值=1，此处限制并发为4）
DEFINE_MUTEX(user_config_mutex);    // 互斥量

// 批量配置任务（可能阻塞）
int bulk_config_task(void *arg) {
    if (down_interruptible(&bulk_config_sem))  // 获取信号量（可被信号中断）
        return -ERESTARTSYS;

    perform_config_update(arg);  // 可能涉及IO或睡眠操作
    up(&bulk_config_sem);        // 释放信号量
    return 0;
}

// 用户配置IOCTL（互斥访问）
long vnic_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {
    mutex_lock(&user_config_mutex);      // 阻塞获取互斥量
    switch (cmd) {
        case SET_VNIC_PARAM:
            // 解析并应用用户配置
            break;
        default:
            return -ENOTTY;
    }
    mutex_unlock(&user_config_mutex);
    return 0;
}
```

信号量+互斥量——实践场景

- 场景：
  - 信号量：限制批量配置任务的并发数（如最多同时执行 4 个）。
  - 互斥量：保护用户态配置接口（如 ioctl）的互斥访问。
- 理由：信号量管理资源池，互斥量简化互斥逻辑。

读写信号量

- 概念
  - ① 读写信号量将访问者分为读者与写者
  - ② 读写信号量可能引起进程阻塞，但是他允许N个读执行单元同时访问共享资源，而最多只允许有一个写执行单元访问共享资源
  - ③ 若读写信号量未被写者持有或者等待，读者就可以获得读写信号量，否则必须等待直到写者释放读写信号量为止
  - ④)若读写信号量未被读者或写者持有，也没有写者等待，写者可以获得读写信号量，否则必须等待直到信号量全部释放(没有其他访问者)为止
  - 说明1:根据上述分析，在有读者持有读写信号量的情况下，后续的写者请求将被阻塞;而在这个被阻塞的写者请求之后的所有读者 & 写者请求又都会被阻塞
  - 说明2:读写操作靠进程自觉，进程在获取读锁后，应该只进行读操作，但是在实现层面，操作系统无法约束进程的读写行为
  - 说明3:与信号量的比较
    - ① 信号量不允许任何操作之间有并发，即读-读/读-写1写-写操作之间都不允许并发
    - ② 读写信号量只允许读-读操作并发，但不允许其他操作并发
    - 因此读写信号量更适合读多写少的场景，可更好地利用读者并发的特性

读写信号量

- 数据结构

```
struct rw_semaphore {
        atomic_long_t count;
        /*
         * Write owner or one of the read
         * the current state of the rwsem
         * check to see if the write owne
         */
        atomic_long_t owner;
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER
        struct optimistic_spin_queue osq;
#endif
        raw_spinlock_t wait_lock;
        struct list_head wait_list;
#ifdef CONFIG_DEBUG_RWSEMS
        void *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map      dep_map;
#endif
};
```

读写信号量

- 操作API

1	DECLARE_RWSEM(name)	静态声明并初始化读写信号量
2	Init_rwsem (sem)	动态声明并初始化读写信号量
3	down_read(sem)	读者获得读写信号量
4	down_read_trylock(sem)	读者尝试获得读写信号量，不能获得则立刻返回
5	down_write(sem)	写者获得读写信号量
6	down_write_trylock(sem)	写者尝试获得读写信号量，不能获得则立刻返回
7	up_write(sem)	写者释放读写信号量
8	up_read(sem)	读者释放读写信号量
9	downgrade_write(sem)	写者身份降级为读者

读写信号量

- 说明:在使用downgrade_write将写者身份降级为读者后，再释放读写信号量时应以读者身份释放在内核中downgrade_write的使用较少，下面给出一个典型示例，

```
/* snap trace */
if (rinfo->snapblob_len) {
        down_write(&mdsc->snap_rwsem);
        ceph_update_snap_trace(mdsc, rinfo->snapblob,
                               rinfo->snapblob + rinfo->snapblob_len,
                               le32_to_cpu(head->op) == CEPH_MDS_OP_RMSNAP);
        downgrade_write(&mdsc->snap_rwsem);
} else {
        down_read(&mdsc->snap_rwsem);
}

/* insert trace into our cache */
mutex_lock(&req->r_fill_mutex);
err = ceph_fill_trace(mdsc->client->sb, req, req->r_session);
if (err == 0) {
        if (result == 0 && rinfo->dir_nr)
                ceph_readdir_prepopulate(req, req->r_session);
        ceph_unreserve_caps(&req->r_caps_reservation);
}
mutex_unlock(&req->r_fill_mutex);

up_read(&mdsc->snap_rwsem);
```

以写者身份获取锁

在完成写入操作后，降级为读者

另一个分支中，以读者身份获取锁

统一以读者身份释放锁

读写信号量——实践场景

- 场景：固件升级时独占设备访问，允许并发读取固件版本。
- 理由：读写信号量细化读写权限，避免升级时版本读取冲突。

```
DECLARE_RWSEM(fw_rwsem);

// 升级固件（写者独占）
int upgrade_firmware(void) {
    down_write(&fw_rwsem);
    flash_firmware();
    up_write(&fw_rwsem);
}

// 读取固件版本（多个读者）
u32 get_firmware_version(void) {
    down_read(&fw_rwsem);
    u32 ver = read_fw_version();
    up_read(&fw_rwsem);
    return ver;
}
```

RCU

- RCU是读写锁的高性能版本，既允许多个读者同时访问被保护的数据，又允许多个读者和多个写者同时访问被保护的数据(注意:是否可以有多个写者并行访问取决于写者之间使用的同步机制)读者没有任何同步开销，而写者的同步开销则取决于使用的写者间同步机制。

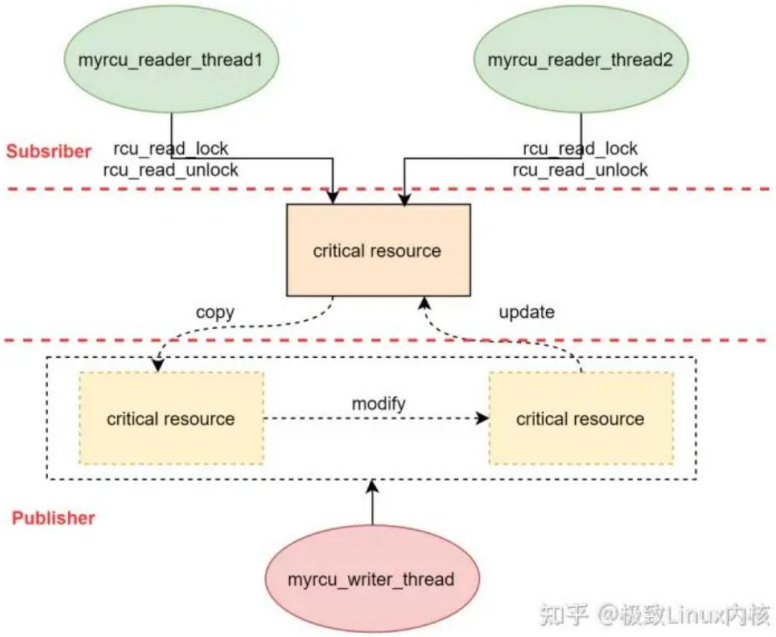
- 对于被RCU保护的共享数据结构，读者不需要获得任何锁就可以访问它，但写者在访问它时首先拷贝一个副本，然后对副本进行修改，最后使用一个回调(callback)机制在适当的时机(所有引用该数据的CPU都退出对共享数据的操作时)把指向原来数据的指针重新指向新的被修改的数据。有一个专门的垃圾收集器探测读者的信号，一旦所有读者都已发送信号告知它们不在使用被RCU保护的数据结构，垃圾收集器就调用回调函数完成最后的数据释放或修改操作。

- RCU不能替代读写锁，当写比较多时，对读者的性能提高不能弥补写者导致的损失，但是大多数情况下RCU的表现高于读写锁。

RCU

- 可以用下面一张图来总结，当写线程 myrcu_writer_thread 写完后，会更新到另外两个读线程 myrcu_reader_thread1 和 myrcu_reader_thread2。读线程像是订阅者，一旦写线程对临界区有更新，写线程就像发布者一样通知到订阅者那里

- 写者在拷贝副本修改后进行 update 时，首先把旧的临界资源数据移除（Removal）；然后把旧的数据进行回收（Reclamation）。结合 API 实现就是，首先使用 rcu_assign_pointer 来移除旧的指针指向，指向更新后的临界资源；然后使用 synchronize_rcu 或 call_rcu 来启动 Reclaimer，对旧的临界资源进行回收（其中 synchronize_rcu 表示同步等待回收，call_rcu 表示异步回收）

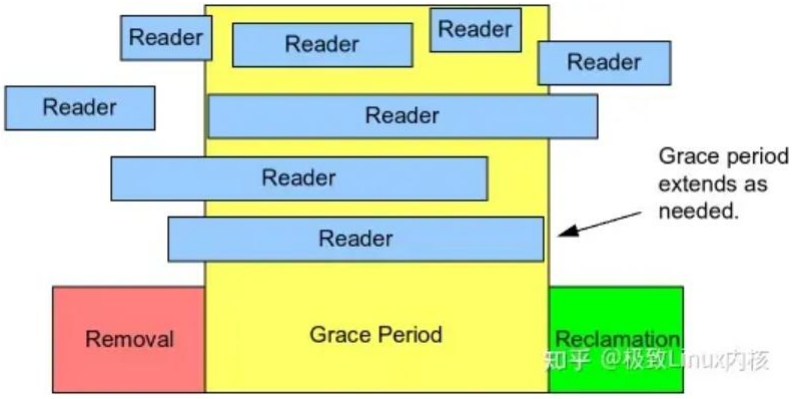


RCU

1	rcu_read_lock	读者在读取由 RCU 保护的共享数据时使用该函数标记它进入读端临界区。
2	rcu_read_unlock	与 rcu_read_lock 配套使用，以标记读者退出临界区。
3	synchronize_rcu	同步 RCU，即所有的读者已经完成读端临界区，写者才可以继续下一步操作。由于该函数将阻塞写者，只能在进程上下文中使用。
4	call_rcu	该函数将把回调函数 func 注册到 RCU 回调函数链上，然后立即返回。。
5	rcu_assign_pointer	用于 RCU 指针赋值
6	rcu_dereference	用于 RCU 指针取值
7	list_add_rcu	向 RCU 注册一个链表结构
8	list_del_rcu	从 RCU 移除一个链表结构

RCU

- 为了确保没有读者正在访问要回收的临界资源，Reclaimer 需要等待所有的读者退出临界区，这个等待的时间叫做宽限期（Grace Period）。
- Grace Period，中文叫做宽限期，从 Removal 到 Reclamation，中间就隔了一个宽限期，只有当宽限期结束后，才会触发回收的工作。宽限期的结束代表着 Reader 都已经退出了临界区，因此回收工作也就是安全的操作了



RCU

- RCU 临界区管理
  - 之前的同步机制中，均是利用锁或原子操作实现的，一个锁管理一个临界区，并通过加锁解锁控制进程进入或者离开临界区。一个程序中可以存在若干的临界区，因此可以对应存在若干把锁分别管理，这是之前所有锁机制的基础。
  - 然而RCU并不基于锁机制实现，RCU字段是耦合在进程描述符和CPU变量中的，是一种与系统强耦合的同步机制，RCU负责管理进程内所有的临界区，进程通过调用rcuread_lock与rcu_read_unlock标记读者临界区，通过rcu_assign_pointer、list_add_rcu将数据纳入保护区，当写者copy出新数据时在读者全部退出临界区后，将新数据指针更新，旧数据将在垃圾收集器的检查中被释放，但存在延迟、
- RCU 限制条件
  - RCU只保护动态分配并通过指针引用的数据结构
  - 在被RCU保护的临界区中，任何内核路径都不能睡眠(经典实现中)

RCU

- rcu_head 是RCU回调函数的关键结构。此外，回调机制主要涉及两个基本函数 call_rcu(用于注册)，rcu_reclaim(用于调用)。

```
struct callback_head {
        struct callback_head *next;
        void (*func)(struct callback_head *head);
} __attribute__((aligned(sizeof(void *))));
#define rcu_head callback_head
```

```
__call_rcu_common(struct rcu_head *head, rcu_callback_t func, boo
{
        static atomic_t doublefrees;
        unsigned long flags;
        bool lazy;
        struct rcu_data *rdp;

        /* Misaligned rcu_head! */
        WARN_ON_ONCE((unsigned long)head & (sizeof(void *) - 1));

        if (debug_rcu_head_queue(head)) {
                /*
                 * Probable double call_rcu(), so leak the callba
                 * Use rcu:rcu_callback trace event to find the p
                 * time callback was passed to call_rcu().
                 */
                if (atomic_inc_return(&doublefrees) < 4) {
                        pr_err("%s(): Double-freed CB %p->%pS()!!
                        mem_dump_obj(head);
                }
                WRITE_ONCE(head->func, rcu_leak_callback);
                return;
        }
```

```
static inline bool rcu_reclaim_tiny(struct rcu_head *head)
{
        rcu_callback_t f;
        unsigned long offset = (unsigned long)head->func;

        rcu_lock_acquire(&rcu_callback_map);
        if (__is_kvfree_rcu_offset(offset)) {
                trace_rcu_invoke_kvfree_callback("", head, offset);
                kvfree((void *)head - offset);
                rcu_lock_release(&rcu_callback_map);
                return true;
        }

        trace_rcu_invoke_callback("", head);
        f = head->func;
        debug_rcu_head_callback(head);
        WRITE_ONCE(head->func, (rcu_callback_t)0L);
        f(head);
        rcu_lock_release(&rcu_callback_map);
        return false;
}
```

RCU

- call_rcu 核心功能是将需要延迟释放的内存或数据结构（如 rcu_head）安全地提交到当前 CPU 的 RCU 回调链表中。具体流程包括：通过双重释放检测（debug_rcu_head_queue）避免重复提交导致的内存泄漏风险，绑定用户指定的清理函数（如 kvfree），并根据运行模式（如无回调批处理 No-CB 或普通模式）选择异步或本地处理逻辑。全程禁用中断（local_irq_save）确保链表操作的原子性，同时支持离线 CPU 和早期启动阶段的特殊初始化。

- RCU 的回收逻辑（如 rcu_reclaim_tiny）由内核自动触发，在宽限期结束后安全执行回调函数。当所有读者退出临界区时，RCU 子系统遍历回调链表，若回调为内存释放（通过 __is_kvfree_rcu_offset 判断），则直接调用 kvfree；若为用户自定义逻辑，则执行注册的清理函数。执行后清除回调指针并记录跟踪信息，同时通过锁标记（rcu_lock_acquire）确保上下文一致性。

RCU

```
static __always_inline void rcu_read_lock(void)
{
        __rcu_read_lock();
        __acquire(RCU);
        rcu_lock_acquire(&rcu_lock_map);
        RCU_LOCKDEP_WARN(!rcu_is_watching(),
                         "rcu_read_lock() used illegally while idle");
}
```

- RCU Read
  - rcu read lock与rcu read unlock的经典实现是不可抢占的，从代码看，这两个函数仅仅用于开关抢占。
  - RCU read之所以禁止抢占，主要是由于写者必须等待读者完全执行完退出临界区方能修改数据指针。一旦读者被抢占，那么其退出临界区的过程将会阻塞，进而阻塞写者，这对性能是一种不小的开销。但是现在的linux 内核版本中提供了可抢占的版本，只是对抢占深度做了把控。
- RCU Synchronize
  - 可是RCU是如何获知所有读者已经离开临界区?RCUread实现中并没有设置字段标记进出临界区，RCU是通过什么判断的呢?既然RCU read过程不可抢占，那么换言之，若所有 CPU 都已经过一次上下文切换，则所有前置 reader 的临界区必定全部退出。
  - 我们主要分析以下两种
    - rcu check callbacks
    - synchronize rcu

RCU

```
struct foo
{
    int a;
    struct rcu_head rcu;
};

static struct foo* g_ptr;
static int myrcu_reader_thread1(void*data) //读者线程1
{
    struct foo *p1 = NULL;
    while(1)
    {
        if (kthread_should_stop()) break;
        msleep(20);
        rcu_read_lock();
        mdelay(200);
        p1=rcu_dereference(g_ptr);
        if(p1) printk("%s: read a=%d\n",__func__,p1->a);
        rcu_read_unlock();
    }
    return 0;
}

static int myrcu_reader_thread2(void*data)//读者线程2
{
    struct foo *p2 = NULL;
    while(1)
    {
        if (kthread_should_stop()) break;
        msleep(30);
        rcu_read_lock();
        mdelay(100);
        p2=rcu_dereference(g_ptr);
        if (p2) printk("%s: read a=%d\n",__func__,p2->a);
        rcu_read_unlock();
    }
    return 0;
}

static void myrcu_del(structrcu_head*rh)//回收处理操作
{
    struct foo *p = container_of(rh, struct foo, rcu);
    printk("%s: a=%d\n",__func__,p->a);
    kfree(p);
}
```

```
static int myrcu_writer_thread(void*p)//写者线程
{
    struct foo *old;
    struct foo *new_ptr;
    int value = (unsigned long)p;
    while(1)
    {
        if (kthread_should_stop()) break;
        msleep(250);
        new_ptr = kmalloc(sizeof(structfoo),GFP_KERNEL);
        old = g_ptr;
        *new_ptr = *old;
        new_ptr->a = value;
        rcu_assign_pointer(g_ptr,new_ptr);
        call_rcu(&old->rcu, myrcu_del);
        printk("%s: write to new %d\n",__func__,value);
        value++;
    }
    return 0;
}

static struct task_struct *reader_thread1;
static struct task_struct *reader_thread2;
static struct task_struct *writer_thread;

static int __init my_test_init(void)
{
    int value = 5;
    printk("figo: my module init\n");
    g_ptr = kzalloc(sizeof(structfoo),GFP_KERNEL);
    reader_thread1 = kthread_run(myrcu_reader_thread1,NULL,"rcu_reader1");
    reader_thread2 = kthread_run(myrcu_reader_thread2,NULL,"rcu_reader2");
    writer_thread = kthread_run(myrcu_writer_thread,(void*)(unsignedlong)value,"rcu_writer");
    return0;
}

static void __exit my_test_exit(void)
{
    printk("goodbye\n");
    kthread_stop(reader_thread1);
    kthread_stop(reader_thread2);
    kthread_stop(writer_thread);
    if (g_ptr) kfree(g_ptr);
}
```

RCU——实践

```c
struct traffic_history {
    struct rcu_head rcu_head;  // RCU回调头
    u64 bytes_transferred;
    u64 packets;
};
struct traffic_history __rcu *history;  // RCU保护的指针

// 读者：无锁访问历史数据
void log_history(void) {
    struct traffic_history *h;
    rcu_read_lock();                // 进入RCU读临界区
    h = rcu_dereference(history);   // 安全获取指针
    printk(KERN_INFO "History: %llu bytes\n", h->bytes_transferred);
    rcu_read_unlock();              // 退出读临界区
}

// 写者：替换历史记录并异步释放旧数据
void update_history(void) {
    struct traffic_history *new = kmalloc(sizeof(*new), GFP_KERNEL);
    struct traffic_history *old;

    new->bytes_transferred = ...;  // 更新数据
    old = rcu_access_pointer(history);
    rcu_assign_pointer(history, new);  // 原子替换指针
    call_rcu(&old->rcu_head, free_history);  // 延迟释放旧数据
}

// RCU回调函数（宽限期结束后执行）
static void free_history(struct rcu_head *rcu) {
    struct traffic_history *h = container_of(rcu,
        struct traffic_history, rcu_head);
    kfree(h);
}
```

RCU——实践场景

- 场景：无锁读取历史流量记录（struct traffic_history），延迟释放旧数据。
- 理由：RCU 实现读者零开销，写者延迟释放确保安全。

同步机制之间的比较

机制	等待机制	优缺	适用于
原子操作	无，ldrex 与 strex 实现内存独占访问	性能相当高，场景受限	资源计数
自旋锁	忙等待，唯一持有	多处理器下性能优异，临界区时间长会浪费	中断上下文
信号量	睡眠等待（阻塞），多数持有	相对灵活，适用于复杂情况，但是时耗长	情况复杂且耗时长的场景，比如内核与用户空间的交互
互斥锁	睡眠等待（阻塞），优先自旋等待唯一持有	较信号量，高效，适用于复杂场景，但存在若干限制条件	满足使用条件下，mutex 优先于信号量
读写信号量	睡眠等待（阻塞），优先自旋等待，读者多数持有，写者唯一持有	读写优化的信号量，对于读多写少的情形性能大大提升	类似信号量，更适用于读多写少的场景，例如内存管理系统

机制	等待机制	优缺	适用于
读写锁	忙等待，读者多数持有，写者唯一持有	读写优化的自旋锁，读者阻塞写者，对于读多写少的情形性能大大提升，但写者可能受阻塞降低性能	类似自旋锁，更适用于读多写少的场景
顺序锁	忙等待	写优先读的锁，对于读写同时比较少的情况拥有高性能表现	不允许读者阻塞写者，写者大大少于读者的情形，例如时钟系统
RCU		在绝大部分为读而只有极少部分为写的情况下，它是非常高效的，但延后释放内存会造成内存开销，写者阻塞比较严重	读多写少的情况下，对内存消耗不敏感的情况下，满足 RCU 条件的情况下，优先于读写锁使用。对于动态分配数据结构这类引用计数的机制，也有高性能的表现。

实践任务

- 在不实施内核同步控制的情况下，尝试经过不同的内核路径，触发共享变量的不一致，参考：http://10.10.21.30/linux-kernel/practice_kern/-/tree/main/RaceCondition
- 编写系统调用、中断处理函数、异常处理函数、可延迟函数，在上述不同的内核代码中访问一个自建的共享变量，参考：http://10.10.21.30/linux-kernel/practice_kern/-/tree/main/VisitShared