# 一、同步概述

## 1、临界区和竞争条件

访问和操作共享数据的代码段称为临界区。如果两个执行线程在同一个临界区中同时执行称为竞争条件。同步就是避免并发和防止这样的竞争条件。

之所以需要同步，是因为进程会被调度程序抢占和重新调度。由于进程可以在任何时刻被抢占，而调度程序完全可能选择另外一个高优先级的进程到处理器上执行，所以就会使得一个程序正处于临界区时被非自愿的抢占了。内核通过原子操作和加锁等方式进行处理，锁有多种多样，而且加锁的粒度范围也各不相同，但主要区别在于：锁被其他进程持有时的行为表现（一些锁会简单的执行忙等待，而另外一些锁会使当前任务睡眠直到锁可用为止）。

## 2、内核并发情况

内核可能造成并发执行的情况：

A．中断：中断几乎可以在任何时刻异步发生，也就可能随时打断当前正在执行的代码

B．软中断和tasklet：内核能在任何时刻唤醒或调度软中断和tasklet，打断当前正在执行的代码

C．内核抢占：内核具有抢占性时内核的任务可能会被另外一个任务抢占

D．睡眠及与用户空间的同步：在内核执行的进程可能会睡眠，这就会唤醒调度程序，从而导致调度一个新的用户进程执行

1. 对称多处理：两个或多个处理器可以同时执行相同代码。

## 3、加锁保护

加锁是给数据而不是代码加锁，经验判断方法：如果有其他执行线程可以访问这些数据，那么就给这些数据加上某种形式的锁；如果任何其他什么东西都可以看见它，那么就要锁住它。在编写内核代码时，需要确认以下问题：

A．这个数据是不是全局的？除了当前线程外，其他线程能不能访问它？

B．这个数据会不会在进程上下文和中断上下文中共享？它是不是要在两个不同的中断处理程序中共享？

C．进程在访问数据时可不可能被抢占？被调度的新程序会不会访问同一数据？

D．当前进程是不是会睡眠（阻塞）在某些资源上，如果是，它会让共享数据处于何种状态？

E．怎样防止数据失控？

F．如果这个函数又在另一个处理器上被调度将会发生什么呢？

G．如何确定代码远离并发威胁呢？

## 4、死锁

产生的条件：要有一个或多个执行线程和一个或多个资源，每个线程都在等待其中的一个资源，但所有的资源都已经被占用。所有线程都互相等待，但它们永远不会释放已经占有的资源，于是任何线程都无法继续。

避免死锁的一些简单规则：

A．按顺序加锁：使用嵌套的锁时必须保证以相同的顺序获取锁，可以阻止致命拥抱类型的死锁。

B．防止发生饥饿：试问该代码执行是否一定会结束

C．不要重复请求同一个锁

D．设计应该力求简单：越复杂的加锁方案越有可能造成死锁

# 二、同步方法

最好的同步技术就是把设计不需要同步的内核放在首位，因为任何一种显式的同步方法都有不容忽视的性能开销。常用内核使用同步方法：

|  |  |  |
| --- | --- | --- |
| 方法 | 描述 | 使用范围 |
| per-CPU变量 | 在CPU之间复制数据结构 | 所有CPU |
| 原子操作 | 原子地“读—修改—写”的指令 | 所有CPU |
| 内存屏障 | 避免指令重新排序 | 本地CPU或所有CPU |
| 自旋锁 | 加锁时忙等 | 所有CPU |
| 信号量 | 加锁时阻塞等待 | 所有CPU |
| 顺序锁 | 基于访问计数器的锁 | 所有CPU |
| RCU | 通过指针而不是锁来访问共享数据结构 | 所有CPU |
| 本地中断的禁止 | 禁止单个CPU上的中断处理 | 本地CPU |
| 本地软中断的禁止 | 禁止单个CPU上的可延迟函数处理 | 本地CPU |

## 1、per CPU变量

Per-CPU变量主要是数据结构的数组，系统的每个CPU对应数组的一个元素。一个CPU不应该访问与其他CPU对应的数组元素，但它可以随意读或修改它自己的元素而不用担心出现竞争条件。

虽然per-CPU变量为来自不同CPU的并发访问提供保护，但对来自异步函数（中断处理程序和可延迟函数）的访问不提供保护，在这种情况下需要另外的同步原语

此外，在单处理器和多处理器系统中，内核抢占都可能是per-CPU变量产生竞争条件。这种情况原则就是：内核控制路径应该在禁止抢占的情况下访问per-CPU变量。一个简单的例子：一个内核控制路径获得了它的per-CPU变量本地副本的地址，然后它因抢占而转移到另外一个CPU上，但仍然引用原来的CPU元素的地址。

## 2、原子操作

若干汇编语言指令通常有如下操作序列：

读一个位于memory中的变量的值到寄存器中

修改该变量的值（也就是修改寄存器中的值）

将寄存器中的数值写回memory中的变量值

在这个过程中运行在两个CPU上的两个内核控制路径试图通过执行非原子操作来同时“读—修改—写”同一存储器单元，就可能造成最终结果不对。避免由于“读—修改—写”指令引起的竞争条件的最容易的办法就是确保这样的操作在芯片级是原子的：任何一个这样的操作都必须以单个指令执行，中间不能中断，且避免其他的CPU访问同一存储单元。

原子操作是其他同步方法的基石，内核提供两组原子操作接口：一组针对整数进行操作，另一组针对单独的位进行操作。Linux支持的所有体系结构都实现了两组接口，基本用汇编实现，不同体系结构实现方式不一样，但大部分体系结构会提供支持原子操作的简单算术指令。整数操作接口API：

|  |  |
| --- | --- |
| 原子整数操作 | 描述 |
| ATOMIC\_INIT(int i) | 在声明一个atomic\_t变量时，将它初始化为i |
| int atomic\_read(atomic\_t \*v) | 原子地读取整数变量v |
| void atomic\_set(atomic\_t \*v, int i) | 原子的设置v值为i |
| void atomic\_add(int i, atomic\_t \*v) | 原子的给v加i |
| void atomic\_sub(int i, atomic\_t \*v) | 原子的从v减i |
| void atomic\_inc(atomic\_t \*v) | 原子的给v加1 |
| void atomic\_dec(atomic\_t \*v) | 原子的从v减1 |

## 3、优化和内存屏障

屏障有两种：优化屏障和内存屏障。

编译器编译源代码时，根据优化级别将源代码进行优化，使源代码的指令进行重新排序，以适合于CPU的并行执行。优化屏障原语保证编译程序不会混淆放在原语操作之前的汇编语言指令和放在原语操作之后的汇编语言指令，这些汇编语言指令在C中都有对应的语句。

#define barrier() \_\_asm\_\_ \_\_volatile\_\_("": : :"memory")

\_\_asm\_\_表示插入了汇编语言程序，\_\_volatile\_\_表示阻止编译器对该值进行优化，确保变量使用了用户定义的精确地址，而不是装有同一信息的一些别名。

内存屏障原语确保在原语之后的操作开始执行前，原语之前的操作已经完成，保证内存访问按程序的顺序完成。

|  |  |
| --- | --- |
| 屏障 | 描述 |
| rmb() | 阻止跨越屏障的载入动作发生重排序 |
| read\_barrier\_depends() | 阻止跨越屏障的具有数据依赖关系的载入动作重排序 |
| wmb() | 阻止跨越屏障的存储动作发生重排序 |
| mb() | 阻止跨越屏障的载入和存储动作重新排序 |
| smp\_rmb() | 在SMP上提供rmb()功能，在UP上提供barrier()功能 |
| smp\_read\_barrier\_depends() | 在SMP上提供read\_barrier\_depends()功能，在UP上提供barrier()功能 |
| smp\_wmb() | 在SMP上提供wmb()功能，在UP上提供barrier()功能 |
| smp\_mb() | 在SMP上提供mb()功能，在UP上提供barrier()功能 |
| barrier | 阻止编译器跨屏障对载入或存储操作进行优化 |

## 4、自旋锁

Linux内核中最常见的锁是自旋锁。自旋锁最多只能被一个可执行线程持有：如果一个执行线程试图获得一个被已经持有的自旋锁，那么该线程就会一直进行忙循环（自旋浪费CPU时间）等待锁重新可用；要是锁未被争用，请求锁的执行线程便能立刻得到它，继续执行。

因为自旋锁在同一时刻至多被一个执行线程持有，所以一个时刻只能有一个线程位于临界区内，这就为多处理器提供了防止并发访问所需的保护机制。在单处理器上，编译的时候并不会加入自旋锁，仅仅被当作一个设置内核抢占机制是否被启用的开关，如果禁止内核抢占，那么在编译时自旋锁会被完全剔除出内核。

自旋锁可以使用在中断处理程序：在中断处理程序使用自旋锁时，一定要在获取之前首先禁止本地中断，否则中断处理程序会打断正持有锁的内核代码，有可能会试图争用这个已经被持有的自旋锁，引发双重请求导致死锁。

注意：自旋锁不可以递归使用。内核可以打开配置CONFIG\_DEBUG\_SPINLOCK调试自旋锁。

自旋锁与下半部的几种情况：

情况1：当下半部和进程上下文共享数据时，必须对进程上下文中的共享数据进行保护，因为下半部可以抢占进程上下文的代码，因此需要加锁的同时还需要禁止下半部执行。

情况2：当中断处理程序和下半部共享数据时，因为中断处理程序可以抢占下半部，因此在获取恰当锁的同时还要禁止中断。

情况3：同类的tasklet不可能同时运行，所以对于同类tasklet中的共享数据不需要保护。当两个不同类的tasklet共享数据时，需要访问下半部中的数据前先获得一个普通的自旋锁，因为不同类型tasklet可以同时运行在多个处理器。但是不需要禁止下半部，因为同一处理器上不会有tasklet互相抢占。

情况4：软中断不管是同类型还是不同类型，共享数据时都必须先得到锁的保护，因为同种类型的连个软中断可以同时运行在多个处理器上。但是不需要禁止下半部，因为同一处理器一个软中断不会去抢占另一个软中断。

自旋锁操作接口列表：

|  |  |
| --- | --- |
| 接口 | 描述 |
| spin\_lock | 获取指定的自旋锁 |
| spin\_unlock | 释放指定的锁 |
| spin\_lock\_irq | 禁止本地中断并获取指定锁 |
| spin\_unlock\_irq | 释放指定锁并开启本地中断 |
| spin\_lock\_irqsave | 保存本地中断状态，禁止本地中断，获取指定锁 |
| spin\_unlock\_irqrestore | 释放指定锁，让本地中断恢复到以前的状态并开启 |
| spin\_lock\_init | 动态初始化指定的spinlock\_t |
| spin\_trylock | 试图获取指定的锁，如果未获取，则返回非0 |
| spin\_is\_locked | 如果指定的锁当前正在被获取，则返回非0，否则返回0 |

## 5、读写自旋锁

读写自旋锁的引入是为了增加内核的并发能力，为读和写分别提供了不同的锁。一个或多个读任务可以并发的持有读者锁；相反，用于写的锁最多只能被一个写任务持有，而且此时不能有并发的读操作。允许对数据结构并发读提高了系统性能。

读写自旋锁机制照顾读比照顾写要多一点：当读锁被持有时，写操作为了互斥访问只能等待，但是读者却可以继续成功的占用锁；而自旋锁等待的写者在所有读者释放锁之前是无法获得锁的。

读写锁接口操作列表：

|  |  |
| --- | --- |
| 接口 | 描述 |
| read\_lock | 获得指定的读锁 |
| read\_unlock | 释放指定读锁 |
| read\_lock\_irq | 禁止本地中断并获得指定读锁 |
| read\_unlock\_irq | 释放指定的读锁并激活本地中断 |
| read\_lock\_irqsave | 保存本地中断状态，禁止本地中断并获得指定读锁 |
| read\_unlock\_irqrestore | 释放指定读锁，将本地中断恢复到保存的状态并开启 |
| write\_lock | 获取指定写锁 |
| write\_unlock | 释放指定写锁 |
| write\_lock\_irq | 禁止本地中断并获得指定写锁 |
| write\_unlock\_irq | 释放指定写锁并开启本地中断 |
| write\_lock\_irqsave | 保存本地中断状态，禁止本地中断并获得指定写锁 |
| write\_unlock\_irqrestore | 释放指定写锁，将本地中断恢复到保存状态并开启 |
| write\_trylock | 试图获得指定写锁，如果写锁不可用，返回非0值 |
| rwlock\_init | 初始化指定的rwlock\_t |

## 6、顺序锁

顺序锁和读写自旋锁非常相似，只是它为写者赋予了较高的优先级：即使读者正在读的时候也允许写着继续运行。这种策略好处是写者永远不会等待（除非另外一个写者正在写），缺点是导致读者不得不反复多次读相同的数据直到它获得有效的副本。

使用场景：

共享数据存在很多读者

共享数据写者很少

希望写者优先于读者，而且不允许读者让写者饥饿

共享数据简单，但无法使用原子操作的

最有说服力的一个例子：jiffies。

使用方法如下：

定义一个seq锁：

seqlock\_t mr\_seq\_lock = DEFINE\_SEQLOCK(mr\_seq\_lock)

写锁的方法：

write\_seqlock(&mr\_seq\_lock)

/\* 操作共享数据 \*/

write\_sequnlock(&mr\_seq\_lock)

这个和普通自旋锁类似，不同的情况发生在读时，并且与自旋锁有很大不同：

unsigned log seq;

do

{

Seq=read\_readbegin(&mr\_seq\_lock);

/\* 读共享数据 \*/

}while(read\_seqretry(&mr\_seq\_lock, seq))

## 7、RCU（读-拷贝-更新）

RCU是为了保护在多数情况下被多个CPU读的数据结构而设计的另一种同步技术。RCU允许多个读者和写者并发执行，它不使用被所有CPU共享的锁或计数器。

同步技术：读者执行rcu\_read\_lock间接引用数据结构指针所对应的内存单元并开始读这个数据结构；写者要跟新数据结构时，它间接引用指针并生成整个数据结构的副本，然后写者修改这个副本，一旦修改完毕，写者改变指向数据结构的指针，以使它指向被修改后的副本。由于修改指针操作是一个原子操作，所以旧副本和新副本对每个读者或写者都是可见的，在数据结构中不会出现数据崩溃。困难之处在于：写者修改指针不能立即释放数据结构的旧副本，因为写者修改时读者可能还在读旧副本；只有在CPU上所有读者都执行完宏rcu\_read\_unlock之后才可以释放旧副本。

它的主要使用场景：

A．RCU只能保护动态分配的数据机构，并且必须是通过指针访问该数据机构

B．受RCU保护的临界区内不能sleep

C．读写不对称，对写者的性能没有特别要求，但对读者要求极高

D．读者端对新旧数据不敏感。

## 8、信号量

信号量是一种睡眠锁：如果有一个任务试图获得一个不可用的信号量时，信号量会将其推进一个等待队列，然后让其睡眠。这时处理器能重获自由，从而去执行其他代码。当持有的信号量可用后，处于等待队列中的那个任务将被唤醒，并获得该信号量。

信号量可以睡眠的特性决定了一些使用场景：

进程等待信号量重新可用时会睡眠，所以信号量适用于锁会长时间持有的情况

信号量不适用锁短时间持有的情况（短时间更适合自旋锁），因为睡眠、维护队列、唤醒等开销不小

信号量会睡眠不能用于中断上下文，适合使用进程上下文

可以在持有信号量时睡眠，其他进程试图获得同一信号量时不会因此而死锁

占用信号的同时不能占用自旋锁，信号量可以睡眠，而自旋锁不允许睡眠

信号量分为：计数信号量和二值信号量

信号量接口操作列表：

|  |  |
| --- | --- |
| 接口 | 描述 |
| sema\_init() | 指定的计数值初始化动态创建的信号量 |
| down | 试图获得信号量，如果不可用则进入不可中断睡眠状态 |
| down\_interruptible | 试图获得信号量，如果不可用则进入可中断睡眠状态 |
| down\_trylock | 试图获得信号量，如果不可用返回非0值 |
| up | 释放信号量，如果睡眠队列不空则唤醒其中一个任务 |

## 9、读写信号量

类似读写自旋锁

## 10、互斥锁

mutex也是一种可睡眠的锁，其行为和使用计数为1的信号量类似，但操作接口更简单，实现也更高效，而且使用限制更强。

使用场景：

任何时刻中只有一个任务可以持有mutex（mutex的使用计数永远是1）

给mutex上锁者必须负责给其再解锁，即常用方式：在同一上下文中上锁和解锁

递归的上锁和解锁是不允许的

当持有一个mutex时，进程不可以退出

mutex不能在中断或者下半部中使用，mutex\_tryloc也不行

mutex操作接口列表：

|  |  |
| --- | --- |
| 接口 | 描述 |
| mutex\_lock | 为指定的mutex上锁，如果锁不可用则睡眠 |
| mutex\_unlock | 为指定的mutex解锁 |
| mutex\_trylock | 试图获取指定的mutex，如果不可用则返回0 |
| mutex\_is\_locked |  |

互斥锁和自旋锁使用比较：

|  |  |
| --- | --- |
| 需求 | 建议加锁方法 |
| 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 | 优先使用互斥体 |
| 中断上下文中加锁 | 使用自旋锁 |
| 持有锁需要睡眠 | 使用互斥体 |

## 11、完成变量

如果内核中一个任务需要发出信号通知另外一任务发生某个特定事件，完成变量是使两个任务得以同步的简单方法。

如果一个任务要执行一些工作时，另一个任务就会在完成变量上等待；当这个任务完成工作后，会使用完成变量去唤醒在等待的任务。

完成变量操作接口列表：

|  |  |
| --- | --- |
| 接口 | 描述 |
| init\_completion | 初始化指定的动态创建的完成变量 |
| wait\_for\_completion | 等待指定的完成变量接收信号 |
| complete | 发送信号唤醒任何等待任务 |

## 12、禁止本地中断

操作函数：local\_irq\_enable和local\_irq\_disable

## 13、禁止和激活可延迟函数

操作函数：local\_bh\_enable和local\_bh\_disable

# 三、同步选择

使用前面所述的同步原语保护共享数据结构避免竞争条件。系统性能也可能随所选择同步原语种类的不同而有很大变化。内核开发者采用下述由经验得到的法则：把系统中的并发度保持在尽可能高的程度。系统中的并发度又取决于两个主要因素：

同时运转的I/O设备数

进行有效工作的CPU数

为了使I/O吞吐量最大化，应该使中断禁止保持在很短的时间。因为当中断禁止时，由I/O设备产生的IRQ被暂时忽略。

为了有效的利用CPU，应该尽可能避免使用基于自旋锁的同步原语。当一个CPU执行紧指令循环等待自旋锁打开时，是在浪费宝贵的机器周期，更糟糕的是：由于自旋锁对硬件高速缓存的影响而使其对系统的整体性能产生不利影响。

在自旋锁、信号量及中断禁止之间选择，一般来说同步原语的选取取决于访问数据结构的内核控制路径种类。但是只要内核控制路径获得自旋锁，就禁用本地中断或本地软中断，自动禁用内核抢占。

|  |  |  |
| --- | --- | --- |
| 访问共享数据的内核控制路径 | 单处理器保护 | 多处理器进一步保护 |
| 异常 | 信号量 | 无 |
| 中断 | 本地中断禁止 | 自旋锁 |
| 可延迟函数 | 无 | 无或自旋锁 |
| 异常与中断访问共享数据 | 本地中断禁止 | 自旋锁 |
| 异常与可延迟函数访问共享数据 | 本地软中断禁止 | 自旋锁 |
| 中断与可延迟函数访问共享数据 | 本地中断禁止 | 自旋锁 |
| 异常、中断与可延迟函数访问共享数据 | 本地中断禁止 | 自旋锁 |