



北京大学
PEKING UNIVERSITY

Kernel Synchronization

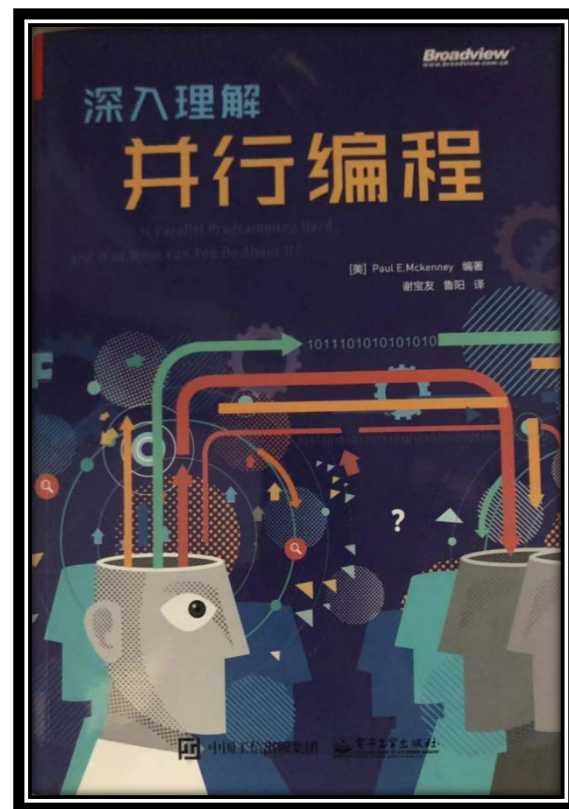
课程内容

日期	知识模块	知识点
2月20日	课程介绍	Linux 内核基本结构、 Linux 的历史、开源基础知识简介、驱动程序介绍
2月27日	实验课一	内核编译，内核补丁
3月5日	内核编程基础知识概述	内核调试技术、模块编程、源代码阅读工具、 linux 启动过程、 git 简介
3月12日	实验课2	内核调试
3月19日	进程管理与调度	Linux 进程基本概念、进程的生命周期、进程上下文切换、 Linux 进程调度策略、调度算法、调度相关的调用
3月26日		
4月2日	实验课3	提取进程信息
4月9日	系统调用、中断处理	系统调用内核支持机制、系统调用实现、 Linux 中断处理、下半部
4月16日	实验课4	添加系统调用、显示系统缺页次数
4月23日	内核同步	原子操作、自旋锁、 RCU 、内存屏障等 linux 内核同步机制
4月30日	内存管理	内存寻址、 Linux 页式管理、物理页分配伙伴系统、 Slab 管理、进程地址空间
5月7日	实验课5	观察内存映射、逻辑地址与物理地址的对应
5月14日	文件系统	Linux 虚拟文件系统、 Ext2/Ext3/Ext4 文件系统结构与特性
5月21日	Linux 设备驱动基础字符设备驱动程序设计	Linux 设备驱动基础、字符设备创建和加载、字符设备的操作、对字符设备进行 poll 和 select 的实现、字符设备访问控制、 IOCTL 、阻塞 IO 、异步事件等
5月28日	基于 linux 的容器平台技术概述	虚拟化技术与容器、 Docker 概述、 Kubernetes 概述
	实验课6	Docker 对容器的资源限制
6月4日	报告课	期末课程报告

Agenda

- Kernel Synchronization Introduction
- Kernel Synchronization Methods
- Applications

推荐书籍：深入理解并行编程
【美】Paul E. Mckenney编著
谢宝友、鲁阳译
电子工业出版社
2017.7 第一版





北京大学
PEKING UNIVERSITY

Kernel Synchronization Introduction

- 假设： 银行帐户 1500元
- 两次取款：
 - 第一次： 100元
 - 第二次： 100元
- 余额： 1300元

```
int withdraw(account, amount) {  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
    return balance;  
}
```

¥ 1500

Thread 1

```
int withdraw(account, amount) {  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
    return balance;  
}
```

¥ 1400

Thread 2

```
int withdraw(account, amount) {  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
    return balance;  
}
```

¥ 1300

- 是否有其他可能的情况呢？
- 如果两个线程同时发生呢？

¥ 1500

```
balance = get_balance(account);  
balance = balance - amount;
```

balance=1400

```
balance = get_balance(account);  
balance = balance - amount;  
put_balance(account, balance);
```

balance=1500

balance=1400

```
put_balance(account, balance);
```

¥ 1400

为什么会出现这种情况？

- 两个进程同时访问共享资源
- 没有采取同步措施

用户空间的同步

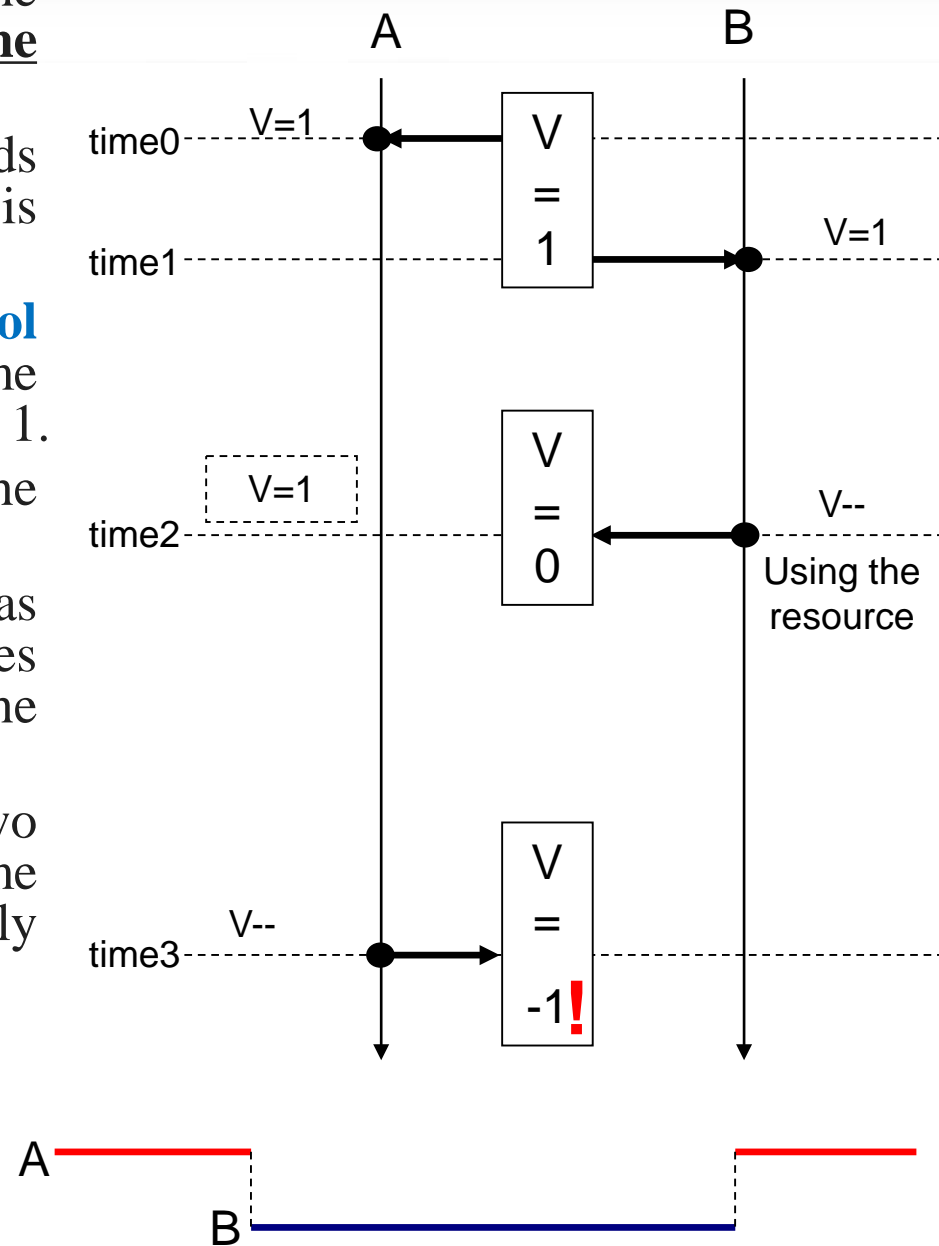
- 对于用户空间的应用程序开发，用户空间的调度与同步之间的关系相对简单，无需过多考虑需要同步的原因。主要是因为：
 - 在用户空间中各个进程都拥有独立的运行空间，进程内部的数据对外不可见，所以各个进程即使并发执行也不会产生对数据访问的竞争。
 - 用户空间与内核空间独立
- 用户同步仅仅需要在进程间通讯和多线程编程时需要考虑

内核中的同步

“内核线程”、“系统调用”、
“硬件中断”、“下半部”

- 如果多个 **内核任务** 同时访问或操作 **共享数据**
 - 可能发生各任务之间相互覆盖共享数据的情况，造成被访问数据处于不一致状态。
- 并发访问共享数据是造成系统不稳定的一个隐患
 - 这种错误难于跟踪和调试。

- a **global variable V** contains the number of available items of some system resource
- The first **kernel control path, A**, reads the variable and determines that there is just one available item.
- At this point, another **kernel control path, B**, is activated and reads the same variable, which still contains the value 1.
- B decreases V and starts using the resource item.
- A resumes the execution, because it has already read the value of V , it assumes that it can decrease V and take the resource item, which B already uses.
- As a final result, V contains -1, and two kernel control paths use the same resource item with potentially disastrous effects.



内核同步介绍

- 内核中很多数据都是共享资源。
- 对共享数据的访问须遵循一定的访问规则，防止并发访问。
- 2.0以后内核支持对称多处理器（SMP）
 - 内核代码同时运行在两个或多个处理器上
 - 不同处理器的内核代码同一时刻，并发访问某些共享资源。
- 2.6内核已经发展为抢占式内核。
 - 调度程序可以在任何时刻抢占正在运行的内核代码。重新调度其他的进程执行
- 内核任务之间、内核任务与进程之间

基本概念

- **临界资源**(critical resource): 系统中某些资源**一次只允许一个进程使用**, 称这样的资源为**临界资源**或**互斥资源**或**共享变量**
- **临界区**(critical section): 对某个**共享的数据结构**(共享资源)进行操作的**程序片段**
 - 在临界区中要避免并发访问。
- **竞争条件**(race condition): **两个执行线程同时处于同一个临界区中**。
- **同步**(synchronization): **避免并发和防止竞争条件**被称为同步。

同步方法简介

i++例子

- 一个全局整型变量和一个临界区
- i++;
 - 得到当前变量i的值并且拷贝到一个寄存器中
 - 将寄存器中的值加1
 - 把i的新值写到内存中
- 两个线程同时进入这个临界区，i的初值是7

Thread 1

get i(7)
Increment i(7->8)
Write back i(8)
-
-
-

Thread 2

-
-
-
get i(8)
Increment i(8->9)
Write back i(9)

Thread 1

get i(7)
Increment i(7->8)
-
Write back i(8)
-

Thread 2

get i(7)
-
Increment i(7->8)
-
Write back i(8)

- 这是最简单的临界区的例子
- 可以使用“原子指令”

Thread 1

Increment $i(7 \rightarrow 8)$

-

Thread 2

-

Increment $i(8 \rightarrow 9)$

或

Thread 1

-

Increment $i(8 \rightarrow 9)$

Thread 2

Increment $i(7 \rightarrow 8)$

-

同步方法简介——加锁

如果我们需要保护的共享资源是**复杂的数据**，如链表：

确保一次只能有一个线程对数据结构进行操作

——**锁**提供了这种保护机制。

Thread 1

try to lock the queue

succeeded: acquired lock

access queue...

unlock the queue

...

Thread 2

try to lock the queue

failed: waiting...

waiting...

waiting...

succeeded: acquired lock

access queue...

unlock the queue

什么造成的并发？

- 中断
 - 中断可能随时打断当前正在执行的代码。
- 软中断与**tasklet**
 - 内核可以唤醒或调度软中断和**tasklet**，打断当前正在执行的代码。
- 内核抢占
 - 内核具有抢占性，内核的任务可能会被另一个任务抢占。
- 睡眠
 - 在内核执行的进程可能会睡眠，这就唤醒调度程序，从而导致调度一个新的用户线程执行。
- 对称多处理
 - 两个或多个处理器可以同时执行代码。

- 如何辨认出真正需要共享的数据和相应的临界区，是一件非常有挑战的事情。
- 在编写代码开始阶段，就要随时考虑并设计适当的锁。
- 不要等代码编写完成后，再去试图寻找临界区并加锁。

要保护什么？

- 要给数据加锁，而不是给代码加锁
- 哪些数据需要加锁
 - 如果有其他执行线程可以访问这些数据。
 - 任何其他东西能够看到它。

编写内核代码时问自己的问题

- 这个数据是不是全局的？除了当前线程外，其它线程能否访问它？
- 这个数据会不会在进程上下文和中断上下文中共享？
- 进程在访问数据时会不会被抢占？被调度的新进程会不会访问同一数据？
- 当前进程会不会睡眠（阻塞）在某些资源上，它会让共享数据处于何种状态？
- 如果这个函数又在另外一个处理器上被调度将会发生什么？
- 如何确保代码远离并发威胁呢？

几乎访问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

死锁

- 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了。所有线程互相等待，但它们永远不会释放已经占有的资源。任何线程都无法继续，这样便产生了死锁。
- 产生死锁的四个必要条件：
 - 互斥条件：
 - 不可剥夺条件：
 - 请求与保持条件：
 - 循环等待条件：

- 自死锁：一个线程试图去获得一个自己已经持有的资源，它将不得不等待锁被释放，最终产生死锁。

acquire lock

acquire lock, again

wait for lock to become available

...

- 同理，考虑 n 个线程 n 个锁，如果每个线程都持有一个其他线程需要得到的锁，那么所有线程都将阻塞等待它们希望得到的锁。
- 如下：两个线程和两把锁，它们通常被叫做ABBA死锁。

Thread 1

acquire lock A

try to acquire lock B

wait for lock B

Thread 2

acquire lock B

try to acquire lock A

wait for lock A

哲学家进餐问题

- 5个哲学家在同一张桌子上就餐。桌子上有5只碗和5只筷子，其中，筷子是共享资源。哲学家可能思考，也可能就餐。
- 哲学家就餐时，只能使用左右最靠近他的筷子，且只有同时获得2只筷子才能就餐。用餐完毕，释放筷子

- 可能的死锁

- 5个哲学家同时拿起左面的筷子，就无法再获得右面的筷子

- 解决办法

- 最多允许4位拿起左面的筷子；

- 仅当能同时获得左右2只筷子时，才允许拿起筷子；

- 规定奇数号的哲学家先拿左面的筷子，后拿右面的筷子，而偶数号哲学家拿筷子的顺序则相反。

避免死锁的一些规则

- **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。
- **防止发生饥饿：**这个任务的执行是否一定会结束？如果事件A不发生，任务B要一直等待下去吗？
- **不要重复请求同一个锁。**
- **设计加锁方案时力求简单——越复杂的加锁方案越容易产生死锁。**

争用和扩展性

- 锁的争用(lock contention)
 - 锁正在被占用时，有**其他线程试图获得该锁**。
 - 被高度争用的锁，成为系统的**瓶颈**，降低系统的性能。
- 扩展性(scalability)
 - 将粗粒度锁变为细粒度锁。



北京大学
PEKING UNIVERSITY

Kernel Synchronization Methods

Linux内核同步机制

在主流的Linux内核中包含了几乎所有现代的操作
系统具有的同步机制，这些同步机制包括：

- **Per-CPU variables**
- **Atomic operations**
- **semaphore**
- **rw_semaphore**
- **spinlock**
- **Rwlock**
- **seqlock**
- **BKL(Big Kernel Lock)**
- **Brlock (Big Reader Lock)** （只包含在2.4内核中）
- **RCU (Read-Copy Update)**
- **Completions**
- **Memory Barriers**
- **Local Interrupt Disabling**
- **Disabling and Enabling Deferrable Functions**

Descriptions

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local CPU or All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data structures through pointers	All CPUs

Per-CPU variables

- **per-CPU variable**
 - an **array** of data structures, one element per each CPU in the system
 - a CPU should not access the elements of the array corresponding to the other CPUs; on the other hand, it can freely read and modify its own element

per-CPU variables

- the per-CPU variables can be used only when it makes sense to logically split the data across the CPUs of the system
- per-CPU variables provide protection against concurrent accesses from several CPUs, they do not provide protection against accesses from asynchronous functions (interrupt handlers and deferrable functions). In these cases, additional synchronization primitives are required.

- per-CPU variables are prone to race conditions caused by kernel preemption, both in uniprocessor and multiprocessor systems. As a general rule, **a kernel control path should access a per-CPU variable with kernel preemption disabled.**

- a kernel control path gets the address of its local copy of a per-CPU variable, and then it is preempted and moved to another CPU: the address still refers to the element of the previous CPU

Functions and macros for the per-CPU variables

/include/linux/percpu-defs.h

Macro or function name

Description

DEFINE_PER_CPU(type, name)

Statically allocates a per-CPU array called `name` of `type` data structures

per_cpu(name, cpu)

Selects the element for CPU `cpu` of the per-CPU array `name`

__get_cpu_var(name)

Selects the local CPU's element of the per-CPU array `name`

get_cpu_var(name)

[Disables kernel preemption] then selects the local CPU's element of the per-CPU array `name`

put_cpu_var(name)

Enables kernel preemption (`name` is not used)

alloc_percpu(type)

Dynamically allocates a per-CPU array of `type` data structures and returns its address

free_percpu(pointer)

Releases a dynamically allocated per-CPU array at address `pointer`

per_cpu_ptr(pointer, cpu)

Returns the address of the element for CPU `cpu` of the per-CPU array at address `pointer`

Atomic operations

- 原子操作可以保证指令以原子的方式被执行，执行过程不被打断。
- Linux内核提供了两组原子操作接口
 - 一组对整数进行操作
 - 一组针对单独的位进行操作

原子整数操作

- Linux内核提供了一个专门的**atomic_t**类型（一个原子访问计数器）和一些专门的函数，这些函数作用于**atomic_t**类型的变量。**atomic_t**类型定义在文件 [</include/linux/types.h>](/include/linux/types.h)中：

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

- **atomic_t类型**
 - 原子操作函数只接受atomic_t类型的操作数
 - 编译器不对相应的值进行访问优化
- **原子整数操作的使用**
 - 常见的用途是**计数器**，计数器无需复杂的锁机制
 - 能使用原子操作的地方，尽量不要使用复杂的锁机制。

下面举例说明原子操作的用法：

- 定义一个**atomic_t**类型的数据很简单，还可以定义时给它设初值：

```
atomic_t u;                /*定义 u*/
```

```
atomic_t v = ATOMIC_INIT(0) /*定义v并把它初始化为0*/
```

- 对其操作如下：

```
atomic_set(&v,4)           /* v = 4 (原子地)*/
```

```
atomic_add(2,&v)           /* v = v + 2 = 6 (原子地) */
```

```
atomic_inc(&v)              /* v = v + 1 =7 (原子地)*/
```

- 如果需要将**atomic_t**转换成**int**型，可以使用**atomic_read()**来完成：

```
printf(“%d\n”,atomic_read(&v)); /* 会打印7*/
```

- 原子整数操作最常见的用途就是实现**计数器**。不必要使用复杂的锁机制来保护一个单纯的计数器，可以使用**atomic_inc()**和**atomic_dec()**这两个相对轻便的操作。
- 还可以用原子整数操作原子地执行一个操作并**检查结果**。一个常见的例子是原子的减操作和检查。

```
int atomic_dec_and_test(atomic_t *v)
```

v减1，为0则返回真；否则返回假

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)	原子地设置 v 值为 i
void atomic_add(int i, atomic_t *v)	原子地给 v 加 i
void atomic_sub(int i, atomic_t *v)	原子地从 v 减 i
void atomic_inc(atomic_t *v)	原子地给 v 加1
void atomic_dec(atomic_t *v)	原子地从 v 减1
int atomic_sub_and_test(int i, atomic_t *v)	原子地从 v 减 i ， 结果等于0 返回真；否则返回假
int atomic_add_negative(int i, atomic_t *v)	原子地给 v 加 i ， 结果是负数 ，返回真；否则返回假
int atomic_dec_and_test(atomic_t *v)	原子地从 v 减1， 结果是0 ，返回真；否则返回假
int atomic_inc_and_test(atomic_t *v)	原子地给 v 加1， 结果是0 ，返回真；否则返回假
int atomic_add_return(int i, atomic_t*v)	原子地给 v 加 i ， 结果是负值 返回真，否则为假
int atomic_sub_return(int i, atomic_t*v)	原子地给 v 减 i ， 结果是负值 返回真，否则为假
int atomic_inc_return(int i, atomic_t*v)	原子地给 v 加1， 结果是负值 返回真，否则为假
int atomic_dec_return(int i, atomic_t*v)	原子地给 v 减1， 结果是负值 返回真，否则为假

原子位操作

- 操作函数的参数是一个指针和一个位号
 - 第0位是给定地址的最低有效位
- 原子位操作中**没有特殊的数据类型**
 - 例如: `set_bit(0, &word);`

原子位操作函数

原子位操作	描述
void set_bit(int nr,void *addr)	原子地设置addr所指对象的第nr位
void clear_bit(int nr,void *addr)	原子地清空addr所指对象的第nr位
void change_bit(int nr,void *addr)	原子地翻转addr所指对象的第nr位
int test_and_set_bit(int nr,void *addr)	原子地设置addr所指对象的第nr位,并返回原先的值
int test_and_clear_bit(int nr,void *addr)	原子地清空addr所指对象的第nr位,并返回原先的值
int test_and_change_bit(int nr,void *addr)	原子地翻转addr所指对象的第nr位,并返回原先的值
int test_bit(int nr,void *addr)	原子地返回addr所指对象的第nr位

Spin locks

- Linux内核中**最常见**的锁：自旋锁（spin lock）
 - 自旋锁**最多只能被一个可执行线程持有**
 - 原子的test-and-set
 - 如果一个线程试图获得一个被争用的自旋锁，那么该线程就会一直进行**忙循环—等待锁重新可用**
 - 锁未被争用，请求锁的执行线程便立即得到它，继续执行

自旋锁的定义

- 自旋锁的定义如下<[/include/linux/spinlock_types.h](#)>:

```
64 typedef struct spinlock {
65     union {
66         struct raw_spinlock rlock;
67         ... ..
73     };
75 };
76 } spinlock_t;
```

```
20 typedef struct raw_spinlock {
21     arch_spinlock_t raw_lock;
22     ... ..
32 } raw_spinlock_t;
```

```
typedef struct qspinlock {
    union {
        atomic_t val;
        .....
    } arch_spinlock_t; Queued Spinlock
```

/arch/x86/include/asm/spinlock_types.h

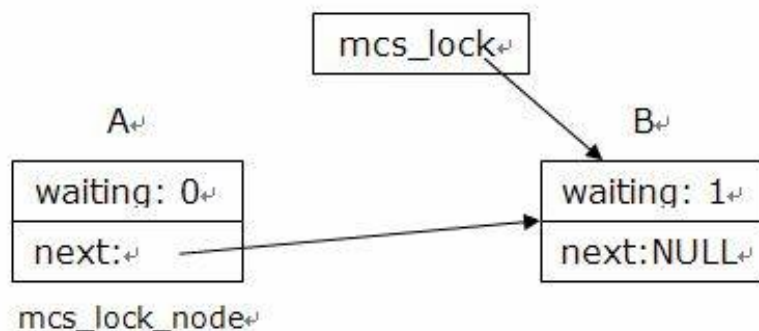
```
#if defined(CONFIG_SMP) ||
# include <asm/spinlock_types.h>
#else
# include <linux/spinlock_types_up.h>
#endif
```

#include <asm-generic/qspinlock_types.h>

```
typedef struct { } arch_spinlock_t;
```

自旋锁的演变

- 锁 → 不公平
- Ticket spinlock → slock 征用瓶颈
- MCS spinlock → 指针消耗存储空间
 - 本地变量上自旋
- Qspinlock
 - 自旋锁被嵌入到许多内核结构中
 - 放到一个 int 中



自旋锁的使用

```
spinlock_t mr_lock=SPIN_LOCK_UNLOCKED;  
    spin_lock(&mr_lock);  
    /*临界区.....*/  
    spin_unlock(&mr_lock);
```

- 自旋锁不应该被长时间持有
 - 被征用的自旋锁使得请求它的线程在等待锁重新可用时自旋（浪费处理器时间）
- 在**中断处理程序**中使用自旋锁（不能使用信号量，因为信号量会导致睡眠）
- 自旋锁针对
 - **SMP**以及
 - **本地内核抢占**

自旋锁的使用

- 获得锁之前，首先应该**禁止本地中断**（在当前处理器上的中断请求）
- 内核提供了**禁止中断同时请求锁**的接口：

```
DEFINE_SPINLOCK(mr_lock);
```

```
unsigned long flags;
```

```
spin_lock_irqsave(&mr_lock,flags);
```

```
/*临界区*/
```

```
spin_unlock_irqrestore(&mr_lock,flags);
```

自旋锁的使用

- 自旋锁在内核中有许多变种，如对下半部而言，可以使用**spin_lock_bh()**来获得**特定锁并且禁止下半部执行**。开锁操作则由**spin_unlock_bh()**来执行

自旋锁的使用

- 进程上下文和下半部共享数据时必须对进程上下文中的共享数据进行保护，禁止下半部
- 中断可能与下半部或者进程上下文中的代码共享数据，所以需要对下半部或者进程上下文中的共享数据保护，禁止中断
- 不同类的tasklet共享数据，需要获得普通的自旋锁，不需禁止下半部（同一处理器上tasklet不互相抢占）
- 同一处理器上的软中断之间不会发生抢占，所以不需要禁止下半部，不同处理器上的软中断之间共享数据则需自旋锁保护

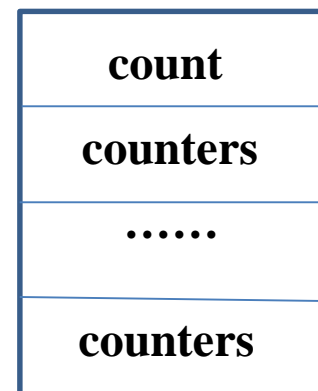
近似的per-CPU计数器

问题：全局共享的一个计数器，并发访问.....

解决：

- 对于某些计数器，没必要时时了解其准确值，可以使用近似值
- 某一个处理器修改计数器的值时不会直接修改计数器的值，而是修改对应数组中特定于当前CPU的数组项。
- 当某个特定于cpu的数组元素修改后的绝对值超出阈值，将修改计数器的值

```
struct percpu_counter{  
    raw_spinlock_t lock;  
    s64 count;  
#ifdef CONFIG_HOTPLUG_CPU  
    struct list_head list;  
#endif  
    s32 __percpu *counters;  
}
```



count是计数器的准确值，**lock**是一个自旋锁，用于在需要准确值时保护计数器。**counters**数组中的数组项是特定于cpu的

近似的per-CPU计数器

```
static inline void percpu_counter_add(struct percpu_counter *fbc, s64 amount)
static inline void percpu_counter_dec(struct percpu_counter)
static inline s64 percpu_counter_sum(struct percpu_counter *fbc)
static inline void percpu_counter_set(struct percpu_counter *fbc, s64 amount)
static inline void percpu_counter_inc(struct percpu_counter *fbc)
static inline void percpu_counter_dev(struct percpu_counter *fbc)
```


- **percpu_counter_add**: 对计数器增加或减少指定的值。如果累积的改变超过**FBC_BATCH**给出的阈值，则会修改传播到计数器的准确值
- **percpu_counter_read**: 读取计数器的当前值，不考虑各个cpu所进行的改动
- **percpu_counter_sum**: 计算计数器的准确值

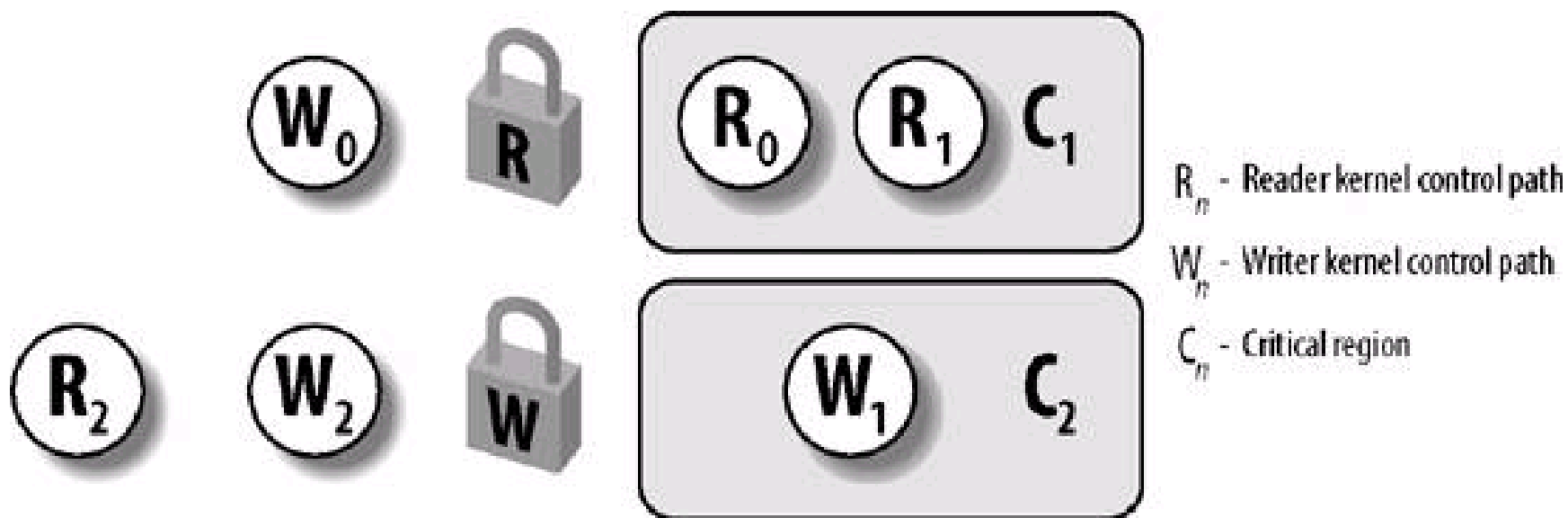
自旋锁方法列表

方法	描述
<code>spin_lock()</code>	获取指定的自旋锁
<code>spin_lock_irq()</code>	禁止本地中断并获取指定的锁
<code>spin_lock_irqsave()</code>	保存 本地中断的当前状态， 禁止 本地中断，并获取指定的锁
<code>spin_unlock ()</code>	释放指定的锁
<code>spin_unlock_irq()</code>	释放指定的锁，并 激活 本地的中断
<code>spin_unlock_irqstore()</code>	释放指定的锁，并让本地中断 恢复 到以前状态
<code>spin_lock_init()</code>	动态初始化 指定的spinlock_t
<code>spin_trylock ()</code>	试图获取指定的锁，如果未获取，则返回非0
<code>spin_is_locked ()</code>	如果指定的锁当前正在被锁定，则返回非0，否则返回0

- 设计自旋锁的初衷是在**短期间**内进行**轻量级**的锁定。一个被持有的自旋锁使得请求它的任务在等待锁重新可用期间进行自旋，所以**自旋锁不应被持有时间过长**
- 自旋锁在内核中主要用来防止**多处理器**中并发访问临界区，防止**内核抢占**造成的竞争。在自旋锁保护的临界区内禁止内核抢占。在**单处理器**情况下，自旋锁仅仅当作一个设置内核抢占的开关。如果内核抢占也不存在，那么自旋锁会在编译时被完全剔除出内核。
- **自旋锁不允许任务睡眠**，因此自旋锁能够在中断上下文中使用

读-写自旋锁

- 为读和写分别提供了不同的锁，提高了内核的并发性

- **读锁**：可被一个或多个读任务并发持有，不允许有并发写操作
- 写锁：只能由一个写任务持有，此时不能有并发的读或写操作



	读	写
读锁	✓	×
写锁	×	×

读-写自旋锁使用方法

- 定义:

```
typedef struct {  
    arch_rwlock_t raw_lock;  
    .....  
} rwlock_t;
```

```
#ifndef __ASSEMBLY__  
typedef union {  
    s32 lock;  
    s32 write;  
} arch_rwlock_t;  
#endif
```

- 初始化:

```
rwlock_t mr_rwlock = W_LOCK_UNLOCKED;
```

```
<linux/arch/x86/include/asm/rwlock.h>
```

- 应用:

in the reader code path:

```
read_lock(&mr_rwlock);  
/* critical section (read only, reader num>=1) ... */  
read_unlock(&mr_rwlock);
```

in the writer code path:

```
write_lock(&mr_rwlock);  
/* critical section (write only, writer num=1) ... */  
write_unlock(&mr_lock);
```

读者-写者自旋锁方法列表

方法	描述
read_lock	获得指定的读锁
read_lock_irq()	禁止本地中断并获得指定读锁
read_lock_irqsave()	存储本地中断的当前状态，禁止本地中断并获得指定读锁
read_unlock()	释放指定的读锁
read_unlock_irq()	释放指定的读锁并激活本地中断
read_unlock_irqrestore()	释放指定的读锁并将本地中断恢复到指定的前状态
write_lock()	获得指定的写锁
write_lock_irq()	禁止本地中断并获得指定写锁
write_lock_irqsave()	存储本地中断的当前状态，禁止本地中断并获得指定写锁
write_unlock_irq()	释放指定的写锁
write_unlock_irqrestore()	释放指定的写锁并激活本地中断
write_trylock()	试图获得指定的写锁，如果写锁不可用，返回非0值
rw_lock_init()	初始化指定的 rwlock_t
rw_is_locked()	如果指定的锁当前已被持有，该函数返回非0值，否则返回0

顺序锁

	读	写
读	✓	✓
写	✓	×

- 读-写锁：
 - 读锁时，写者只能等待，读者可以获得锁
- 顺序锁（seqlock）
 - 写者优先：读时，写者可以进入；
 - 其它与读写锁一样
 - 优点：写者不必等待，读也不会被写阻塞
 - 缺点：读者需要检查读到数据的有效性

- 数据结构

```
281 typedef struct {  
282     struct seqcount seqcount;  
283     spinlock_t lock;  
284 } seqlock_t;
```

- 写者获得锁开始写则+1，写完释放锁+1
- 读者读数据前后读sequence进行对比
 - 不等则读到的数据无效

- 定义一个seq锁:

```
seqlock_t mr_seq_lock = DEFINE_SEQLOCK(mr_seq_lock);
```

- 写的方法如下:

```
write_seqlock(&mr_seq_lock);  
/*写锁被获取后进行操作*/  
write_sequnlock(&mr_seq_lock);
```

```
446static inline void write_seqlock(seqlock_t *sl)  
447{  
448    spin_lock(&sl->lock);  
449    write_seqcount_begin(&sl->seqcount);  
450}
```

```
451  
452static inline void write_sequnlock(seqlock_t *sl)
```

```
453{  
454    write_seqcount_end(&sl->seqcount);  
455    spin_unlock(&sl->lock);  
456}
```

write_seqcount_begin_nested(s, 0)

```
static inline void  
raw_write_seqcount_end(seqcount_t *s)  
{  
    smp_wmb();  
    s->sequence++;  
}
```

- 读的方法如下:

```
unsigned long seq;
do{
    seq = read_seqbegin(&mr_seq_lock); // 进入临界区
    /*读操作*/
}while(read_seqretry(&mr_seq_lock,seq)); // 尝试退出临界区, 存在冲
//突则重试
```

```
437static inline unsigned read_seqretry(const seqlock_t *sl, unsigned start)
438{
439    return read_seqcount_retry(&sl->seqcount, start);
440}
```

```
203static inline int __read_seqcount_retry(const seqcount_t *s, unsigned start)
204{
205    return unlikely(s->sequence != start);
206}
```

- 使用seq锁的例子:

Jiffies使用一个64位变量，记录了自系统启动以来的时钟节拍累加数

```
u64 get_jiffies_64(void)
{
    unsigned long seq;
    u64 ret;
    do{
        seq = read_seqbegin(&xtime_lock);
        ret=jiffies_64;
    }while (read_seqretry(&xtime_lock,seq));
    return ret;
}
```

时钟中断会更新jiffies的值，此时，需要使用seq:

```
write_seqlock(&xtime_lock);
    jiffies_64 += 1;
write_sequnlock(&xtime_lock);
```

Seq锁的使用：

- 数据存在很多读者
- 数据写者很少
- 希望写优先于读，而且不允许写者饥饿
- 数据很简单

- Not every kind of data structure can be protected by a seqlock. As a general rule, the following conditions must hold:
 - The data structure to be protected does not include **pointers** that are modified by the writers and dereferenced by the readers (otherwise, a writer could change the pointer under the nose of the readers)
 - The code in the critical regions of the readers does not have **side effects** (otherwise, multiple reads would have different effects from a single read)
 - Furthermore, the **critical regions of the readers** should be short and writers should **seldom acquire** the seqlock, otherwise repeated read accesses would cause a severe overhead

读-拷贝-更新（RCU）

- 2.5新增，性能好，但是对内存有一定开销，用在网络层和虚拟文件系统中
- 允许多个读者和写者并发
- 不使用锁
- 限制：
 - 只保护被动态分配并通过指针引用的数据结构
 - 临界区内不允许睡眠
 - 适用于读多写少情况，如果写过多，写操作之间的同步开销会很大（必须使用某种锁机制来同步并发的写操作）

	读	写
读	✓	✓
写	✓	✓

• 读者

- 开始: `rcu_read_lock()`
- 结束: `rcu_read_unlock()`

- `rcu_dereference(ptr)`

```
rcu_read_lock();
```

```
    p2=rcu_dereference(ptr);//获取指针后引用, 而 rcu_access_pointer()
```

```
    if (p2 != NULL) {                                     //获取指针后不引用
```

```
        operations on p2;
```

```
    }
```

```
rcu_read_unlock();
```

p2不能在rcu_read_lock()和rcu_read_unlock()保护的代码外使用, 也不能用于写访问。

• 写者

- 生成副本→修改→更改指针指向副本

rcu_assign_pointer(ptr,newptr)

- 修改之后已更新的指针才对其他CPU可见
- 写者不可立即释放旧副本（可能有读者在读），只有所有读者**rcu_read_unlock**后，才可释放旧副本

- **synchronize_rcu()**

- 意味着所有读操作完成，可安全释放旧指针
- 会阻塞写操作直到所有读完成，所以不可用在中断上下文

- **void call_rcu(struct rcu_head *head, rcu_callback_t func);**

- 在所有读操作完成后**调用func函数**；要求在rcu保护的数据结构中嵌入**rcu_head**，然后用**container_of()**访问该数据结构

RCU对链表的操作

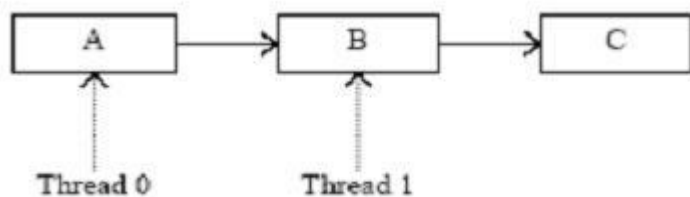


图1 链表的初始状态

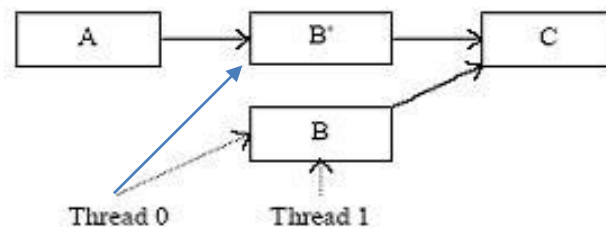


图2 链表的延迟删除图

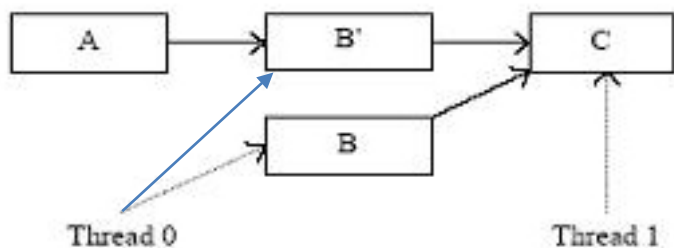


图3 在quiescent期后的链表

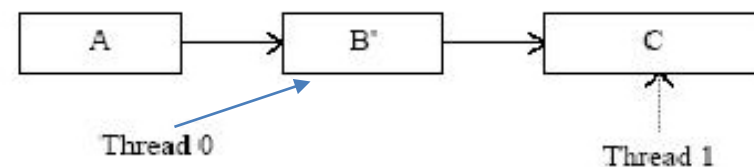


图4 在删除操作后的链表

RCU对链表的操作

- 标准操作函数
- 遍历、修改、删除：标准操作函数_rcu
 - list_add_rcu
 - list_add_tail_rcu
 - list_del_rcu
 - list_replace_rcu
 - list_for_each_rcu: 前后使用rcu_lock和rcu_unlock
 - list_for_each_rcu_safe

```
rcu_read_lock();
oldnode = list_first_or_null_rcu(self_node_db,
                                struct hsr_node, mac_list);
1  if (oldnode) {
    list_replace_rcu(&oldnode->mac_list, &node->mac_list);
    rcu_read_unlock();
    synchronize_rcu();
    kfree(oldnode);
} else {
    rcu_read_unlock();
    list_add_tail_rcu(&node->mac_list, self_node_db);
}
```

RCU对链表的操作

```
static inline void list_replace_rcu(struct list_head *old,
    |         |         struct list_head *new)
{
    new->next = old->next;
    new->prev = old->prev;
    rcu_assign_pointer(list_next_rcu(new->prev), new);
    new->next->prev = new;
    old->prev = LIST_POISON2;
}
```

RCU for Read-Mostly Arrays

- Hash Tables

Hash tables are often implemented as an array, where each array entry has a linked-list hash chain. **Each hash chain** can be protected by RCU as described in the listRCU.txt document. This approach also applies to other array-of-list situations, such as **radix trees**.

信号量

- **Linux**提供两种信号量：
 - 内核信号量：由内核控制路径使用
 - System V IPC信号量：由用户态进程使用
- **Linux**中信号量是一种**睡眠锁**
- 使用信号量时锁的时间长一点是可以忍受的
- 持有信号量的代码可以被抢占
- 可以同时允许多个锁持有者
- 信号量支持两个原子操作P() 和 V()

- 一个任务试图获得已被持有的信号量
 - 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它代码
- 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，从而便可以获得这个信号量

- 信号量是在1968年由Edsger Wybe Dijkstra提出的，此后它逐渐成为一种常用的锁机制。
- 信号量支持两个原子操作P()和V()，这两个名字来自荷兰语 Proberen和Vershogen。前者做测试操作（字面意思是探查），后者叫做增加操作。后来的系统把这两种操作分别叫做down()和up()，Linux也遵从这种叫法

— **down()**操作通过对信号量计数减1来请求获得一个信号量。

— 如果结果 ≥ 0 ，信号量锁被获得，任务可以进入临界区

— 如果结果 < 0 ，任务会被放入等待队列，处理器执行其它任务

— 当临界区中的操作完成后，**up()**操作用来释放信号量，该操作也被称作是“提升 (upping)”信号量，因为它会增加信号量的计数值

— 如果在该信号量上的等待队列不为空，处于队列中等待的任务在被唤醒的同时会获得该信号量。

- 内核中对信号量的定义如下：

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

- 其各个域的含义如下：

- **count**: 如果该值

- 大于0，资源是空闲的，该资源现在可以使用
- 等于0，信号量是忙的，但没有进程等待这个被保护的资源
- 负数，资源是不可用的，且至少有一个进程等待资源

- **wait_list**: 存放等待队列链表的地址，当前等待资源的所有睡眠进程都放在这个链表中。当然，如果count大于或等于0，等待队列就为空。



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

信号量的定义

信号量的使用

```
static DEFINE_SEMAPHORE (mr_sem);  
/* 声明并初始化互斥信号量 */  
if(down_interruptible(&mr_sem)) {  
    /* 信号被接收, 信号量还未获取 */  
}  
/* 临界区... */  
up(&mr_sem);
```

count=1

检查count值, 若小于0则唤醒

信号量方法列表

方法	描述
<code>sema_init(struct semaphore *,int)</code>	以指定的计数值初始化动态创建的信号量
<code>down_interruptible(struct semaphore *)</code>	以试图获得指定的信号量，如果信号量已被争用，则 可中断睡眠状态
<code>down(struct semaphore *)</code>	以试图获得指定的信号量，如果信号量已被争用，则 不可中断睡眠状态
<code>down_trylock(struct semaphore *)</code>	以试图获得指定的信号量，如果信号量已被争用，则立刻返回非0值
<code>up(struct semaphore *)</code>	以释放指定的信号量，如果睡眠队列不空，则唤醒其中一个任务



中断处理程序、可延迟函数

互斥量 (mutex)

- 经典互斥量

```
struct mutex{  
    /* 1:未锁定, 0: 锁定, 负值: 锁定, 可能有等待者*/  
    atomic_t  count;  
    spinlock_t  wait_lock;  
    struct list_head  wait_list;  
    .....  
}
```

- 定义:

- 静态: **DEFINE_MUTEX(name)**
- 动态: **mutex_init(&mutex)**

- 使用

```
mutex_lock(&mutex);
```

临界区

```
mutex_unlock(&mutex);
```

实时互斥量

需要在内核编译时通过配置选项**CONFIG_RT_MUTEX**显式启用，实现了优先级继承，可用于缓解优先级反转的问题。

```
struct rt_mutex {  
    raw_spinlock_t wait_lock;  
    struct rb_root_cached waiters;  
    struct task_struct *owner;  
    .....  
};
```

互斥量的所有者通过owner指定，自旋锁wait_lock提供实际保护。所有等待进程都在wait_list中排队。

与经典互斥量相比，最大的不同是等待列表中的进程**按优先级排序**，当等待列表改变时，内核可以相应的校正锁持有者的优先级。

读-写信号量

- 信号量与自旋锁一样，也有区分读-写访问的可能
- **static DECLARE_RWSEM(name);**
- 提供**down_read_trylock()**和**down_write_trylock()**方法

信号量与自旋锁

- 如果代码需要睡眠，使用信号量是唯一的选择。由于不受睡眠的限制，使用信号量通常来说更加简单一些。
- 如果需要在自旋锁和信号量中作选择，应该取决于锁被持有的时间长短。
 - 理想情况是所有的锁都应该尽可能短的被持有
 - 但是如果锁的持有时间较长的话，使用信号量是更好的选择。
- 另外，信号量不同于自旋锁，它不会关闭内核抢占，所以持有信号量的代码可以被抢占。这意味着信号量不会对影响调度反应时间带来负面影响。

自旋锁 VS 信号量

需求

低开销加锁

短期锁定

长期加锁

中断上下文中加锁

持有锁是需要睡眠

建议的加锁方法

优先使用自旋锁

优先使用自旋锁

优先使用信号量

使用自旋锁

使用信号量

完成变量

- 如果在内核中一个任务需要发出信号通知另一个任务发生了某个特定事件，利用**完成变量**是使两个任务得以同步的简单方法。
- 完成变量提供了**代替信号量**的一个简单解决办法。但是信号量是申请资源，要么得到，得不到就等待。完成变量则申请的时候总是等待，直到另一个任务完成
- 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，当这个任务完成后，会**使用完成变量去唤醒在等待的任务**。

例如：当子进程退出时，`vfork()`系统调用使用完成变量唤醒父进程

完成变量

- **<linux/completion.h>**

```
struct completion {  
    unsigned int done;  
    wait_queue_head_t wait;  
}
```

- 创建并初始化:
 - 静态: **DECLARE_COMPLETION(mr_comp)**
 - 动态: **init_completion()**

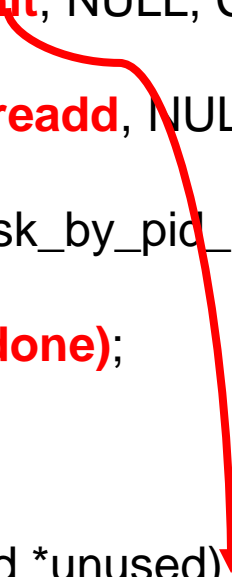
- 一个完成变量的用法示例：
 - 将完成变量作为某数据结构的成员动态创建
 - 等待该数据结构被初始化的内核代码将调用 **wait_for_completion()** 进行等待。
 - 该数据结构初始化完成后，执行初始化的线程调用 **complete()** 唤醒在等待的内核任务。

方法	描述
init_completion(struct completion *)	初始化指定的完成变量
wait_for_completion(struct completion *)	等待指定的完成变量接受信号
complete(struct completion *)	发信号唤醒任何等待任务

start_kernel → rest_init

```
367 static ninline void __init_refok rest_init(void)
368 {
.....
377     kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
378     numa_default_policy();
379     pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
380     rcu_read_lock();
381     kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
382     rcu_read_unlock();
383     complete(&kthreadd_done);
.....
393 }

816 static int __ref kernel_init(void *unused)
817 {
818     kernel_init_freeable();
.....
856 static ninline void __init kernel_init_freeable(void)
857 {
858     /* Wait until kthreadd is all set-up. */
861     wait_for_completion(&kthreadd_done);
.....
```



屏障

• 编译乱序

现代高性能编译器在目标代码优化上都具备对指令进行乱序优化的能力，因为：

- 对访存指令进行乱序，减少逻辑上不必要的访存
- 尽量提高Cache命中率
- 提高CPU的load/store单元的工作效率

• 执行乱序

二进制指令在处理器上执行时，后发射的指令可能先执行完。高级CPU可以根据自己缓存的组织特性，将访存指令重新排序执行。

- 连续地址的访问可能会先执行（缓存命中率高）
- 允许访存的非阻塞（前一条缓存不命中，后面指令可以先执行）

单CPU在碰到**依赖**时会等待；但是在SMP情况，以及程序访问外设的寄存器等情况时，依赖无法判断。

屏障

- 所谓屏障，从处理器角度来说，是用来串行化读写操作的，从软件角度来讲，就是用来解决顺序一致性问题。
- 为什么使用屏障
 - 当处理多处理器之间或硬件设备之间的同步问题时，有时需要在程序代码中以指定的顺序发出读内存和写内存指令
 - 在和硬件交互时，时常需要确保一个给定的读操作发生在其它读或写操作之前

例如：
`preempt_disable();`
`some operations;`
`preempt_enable();`

如果编译器将其代码重新排序：

```
some operations;  
preempt_disable();  
preempt_enable();
```

或者

```
preempt_disable();  
preempt_enable();  
some operations;
```


屏障

这两种情况下，不可抢占的部分都会变得可抢占

<pre>#define preempt_disable() \ do { \ preempt_count_inc(); \ barrier(); \ } while (0)</pre>	<pre>#define preempt_enable() \ do { \ barrier(); \ if (unlikely(preempt_count_dec_and_test())) \ __preempt_schedule(); \ } while (0)</pre>
---	---

- 内存屏障原语确保，在原语之后的操作开始执行之前，原语之前的操作已经完成

- Linux提供了确保顺序的指令称做屏障。

rmb()提供一个读内存屏障；**wmb()**提供一个写内存屏障；**mb()**: 读写内存屏障

屏障

屏障的实现：

```
#ifdef CONFIG_SMP
#define smp_mb()  mb()
#define smp_rmb() rmb()
#define smp_wmb() wmb()
#else
#define smp_mb()  barrier()
#define smp_rmb() barrier()
#define smp_wmb() barrier()
#endif
```

CONFIG_SMP就是用来支持多处理器，如果是**UP(uniprocessor)**系统，就会翻译成**barrier()**。如果是**SMP**系统，屏障就会翻译成对应的**mb()**、**rmb()**和**wmb()**。

```
#define barrier() __asm__ __volatile__(": : : \"memory\")
```

barrier()的作用，就是告诉编译器，内存的变量值发生改变，之前存在寄存器里的变量副本无效，要访问变量还需再访问内存。

屏障

```
CPU0 (b) :  
void foo(void)  
{  
    a = 1;  
    b = 1;  
}
```

```
CPU1 (a) :  
void bar(void)  
{  
    while (b == 0) continue;  
    assert(a == 1);  
}
```

1. CPU 0 执行 `a = 1`。缓存行不在CPU0的缓存中，因此CPU0将“a”的新值放到存储缓冲区，并发送一个“读使无效”消息，这是为了使CPU1的缓存中相应的缓存行失效。
2. CPU 1 执行 `while (b == 0) continue`，但是包含“b”的缓存行不在缓存中，它发送一个“读”消息。
3. CPU 0 执行 `b = 1`，它已经在缓存行中有“b”的值了 (换句话说，缓存行已经处于“modified”或者“exclusive”状态)，因此它存储新的“b”值在它的缓存行中。
4. CPU 0 接收到“读”消息，并且发送缓存行中的最新的“b”的值到CPU1，同时将缓存行设置为“shared”状态。
5. CPU 1 接收到包含“b”值的缓存行，并将其值写到它的缓存行中。
6. CPU 1 现在结束执行 `while (b == 0) continue`，因为它发现“b”的值是1，它开始处理下一条语句。
7. CPU 1 执行 `assert(a == 1)`，并且，由于CPU 1 工作在旧的“a”的值，因此验证失败。
8. CPU 1 接收到“读使无效”消息，并且发送包含“a”的缓存行到CPU0，同时使它的缓存行变成无效。但是已经太迟了。
9. CPU 0 接收到包含“a”的缓存行，并且将存储缓冲区的数据保存到缓存行中。

屏障

CPU0:

```
void foo(void)
{
    a = 1;
    smp_mb();
    b = 1;
}
```

CPU1:

```
void bar(void)
{
    while (b == 0) continue;
    assert(a == 1);
}
```

1. **CPU 0** 执行 **a = 1**。**CPU0**中相应的缓存行是只读的，因此**CPU0**将“a”的新值放入存储缓冲区，并发送一个“使无效”消息。
2. **CPU 1** 执行 **while (b == 0) continue**，但是包含“b”的缓存行不在缓存中，因此它发送一个“读”消息。
3. **CPU 1** 接收到 **CPU 0**的“使无效”消息，将它排队，并立即响应消息。
4. **CPU 0** 接收到来自于 **CPU 1**的响应消息，因此它放心的通过**smp_mb()**，从存储缓冲区移动“a”的值到缓存行。
5. **CPU 0** 执行 **b = 1**。它拥有这个缓存行(也就是说，缓存行已经处于“modified”或者“exclusive”状态)，因此它将“b”的新值存储到缓存行中。
6. **CPU 0** 接收到“读”消息，并且发送包含“b”的新值的缓存行到**CPU 1**，也标记缓存行为“shared”状态。
7. **CPU 1** 接收到包含“b”的缓存行并且将其应用到本地缓存。
8. **CPU 1** 现在执行完 **while (b == 0) continue**，因为它发现“b”的值为1，接着处理下一条语句。
9. **CPU 1** 执行**assert(a == 1)**，由于旧的“a”值还在**CPU 1**的缓存中，因此陷入错误。
- 10.虽然陷入错误，**CPU 1** 处理已经排队的“使无效”消息，并且刷新包含“a”值的缓冲行。

屏障

```
CPU0:  
void foo(void)  
{  
    a = 1;  
    smp_mb();  
    b = 1;  
}
```

```
CPU1:  
void bar(void)  
{  
    while (b == 0) continue;  
    smp_mb();  
    assert(a == 1);  
}
```

1. CPU 0 执行 `a = 1`。相应的缓存行在CPU0的缓存中是只读的，因此CPU0将“a”的新值放入它的存储缓冲区，并且发送一个“使无效”消息以刷新CPU1的缓存。
2. CPU 1 执行 `while (b == 0) continue`，但是包含“b”的缓存行不在它的缓存中，因此它发送一个“读”消息。
3. CPU 1 接收到 CPU 0 的“使无效”消息，将它排队，并立即响应它。
4. CPU 0 接收到 CPU1 的响应，因此它放心的通过 `smp_mb()` 语句，将“a”从它的存储缓冲区移到缓存行。
5. CPU 0 执行 `b = 1`。它已经拥有缓存行(换句话说，缓存行处于“modified”或者“exclusive”状态)，因此它存储“b”的新值到缓存行。
6. CPU 0 接收“读”消息，并且发送包含新的“b”值的缓存行给CPU1，同时标记缓存行为“shared”状态。
7. CPU 1 接收到包含“b”的缓存行并更新到它的缓存中。
8. CPU 1 现在结束执行 `while (b == 0) continue`，因为它发现“b”的值为 1，它处理下一条语句，这是一条内存屏障指令。
9. CPU 1 必须延迟，直到它处理使无效队列中的所有消息。
10. CPU 1 处理已经入队的“使无效”消息，从它的缓存中使无效包含“a”的缓存行。
11. CPU 1 执行 `assert(a == 1)`，由于包含“a”的缓存行已经不在它的缓存中，它发送一个“读”消息。
12. CPU 0 响应“读”消息，发送它的包含新的“a”值的缓存行。
13. CPU 1 接收到缓存行，它包含新的“a”的值1，因此不会陷入失败。

屏障——正确的做法

CPU0:

```
void foo(void)
{
    a = 1;
    smp_wmb();
    b = 1;
}
```

CPU1:

```
void bar(void)
{
    while (b == 0) continue;
    smp_rmb();
    assert(a == 1);
}
```

- **读内存屏障**仅仅保证装载顺序，因此所有在读内存屏障之前的装载将在所有之后的装载前完成。类似的
- **写内存屏障**仅仅保证写之间的顺序。所有在内存屏障之前的存储操作将在其后的存储操作完成之前完成。
- **完整的内存屏障**同时保证写和读之间的顺序。

屏障

内存和编译器屏障方法

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



告知编译器：寄存器中保存的内存地址，在屏障之后失效→保证编译器先处理屏障前的请求

注：因为屏障会降低性能，所以尽量少使用（内核维护者不喜）

禁止抢占

- 如果数据对每个处理器是唯一的（per-CPU），这样的数据可能就不需要使用自旋锁来保护
- 但如果内核是抢占式的，为了防止数据被多个进程以伪并发的方式访问，需要禁止内核抢占


```
#define PREEMPT_BITS 8
#define SOFTIRQ_BITS 8
#define HARDIRQ_BITS 4
#define NMI_BITS 1
```



```
#define PREEMPT_SHIFT 0
#define SOFTIRQ_SHIFT (PREEMPT_SHIFT + PREEMPT_BITS)
#define HARDIRQ_SHIFT (SOFTIRQ_SHIFT + SOFTIRQ_BITS)
#define NMI_SHIFT (HARDIRQ_SHIFT + HARDIRQ_BITS)

#define PREEMPT_MASK (__IRQ_MASK(PREEMPT_BITS) << PREEMPT_SHIFT)
#define SOFTIRQ_MASK (__IRQ_MASK(SOFTIRQ_BITS) << SOFTIRQ_SHIFT)
#define HARDIRQ_MASK (__IRQ_MASK(HARDIRQ_BITS) << HARDIRQ_SHIFT)
#define NMI_MASK (__IRQ_MASK(NMI_BITS) << NMI_SHIFT)

#define __IRQ_MASK(x) ((1UL << (x))-1)

#define PREEMPT_OFFSET (1UL << PREEMPT_SHIFT)
#define SOFTIRQ_OFFSET (1UL << SOFTIRQ_SHIFT)
#define HARDIRQ_OFFSET (1UL << HARDIRQ_SHIFT)
#define NMI_OFFSET (1UL << NMI_SHIFT)
```

禁止抢占

- 可以通过`preempt_disable()`禁止内核抢占。
- 当使用自旋锁时,抢占是被隐式地禁止的。

函数	描述
<code>preempt_disable()</code>	增加抢占计数值, 从而禁止内核抢占
<code>preempt_enable()</code>	减少抢占计数, 并当该值降为0时检查和执行被挂起需调度任务
<code>preempt_enable_no_resched()</code>	激活内核抢占但不再检查任何被挂起的需调度任务
<code>preempt_count()</code>	返回抢占计数

```
#define preempt_disable() \
do { \
    preempt_count_inc(); \
    barrier(); \
} while (0)

#define preempt_count_inc() preempt_count_add(1)

#define preempt_count_add(val) __preempt_count_add(val)

static __always_inline void __preempt_count_add(int val)
{
    raw_cpu_add_4(__preempt_count, val);
}

#define raw_cpu_add_4(pcp, val)      percpu_add_op(, (pcp), val)

DEFINE_PER_CPU(int, __preempt_count) = INIT_PREEMPT_COUNT;
EXPORT_PER_CPU_SYMBOL(__preempt_count);

#define INIT_PREEMPT_COUNT PREEMPT_OFFSET
#define PREEMPT_OFFSET (1UL << PREEMPT_SHIFT)
```

The diagram illustrates the macro expansion flow for the preempt_count_* macros. Red arrows indicate the following relationships:

- `preempt_disable()` expands to `preempt_count_inc()`.
- `preempt_count_inc()` expands to `preempt_count_add(1)`.
- `preempt_count_add(val)` expands to `__preempt_count_add(val)`.
- `__preempt_count_add(int val)` expands to `raw_cpu_add_4(__preempt_count, val)`.
- `raw_cpu_add_4(pcp, val)` expands to `percpu_add_op(, (pcp), val)`.
- `__preempt_count` (in the EXPORT_PER_CPU_SYMBOL macro) expands to `INIT_PREEMPT_COUNT`.
- `INIT_PREEMPT_COUNT` expands to `PREEMPT_OFFSET`.
- `PREEMPT_OFFSET` expands to `(1UL << PREEMPT_SHIFT)`.

判断是否中断上下文

```
#define in_irq()                (hardirq_count())  
#define in_softirq()          (softirq_count())  
#define in_interrupt()        (irq_count())
```

```
#define hardirq_count() (preempt_count() & HARDIRQ_MASK)  
#define softirq_count() (preempt_count() & SOFTIRQ_MASK)  
#define irq_count() (preempt_count() & (HARDIRQ_MASK \br/>                                         | SOFTIRQ_MASK \br/>                                         | NMI_MASK))
```

- 进程上下文：
 - ✓ （用户进程通过）系统调用
 - ✓ 内核线程（包括工作队列）
- 中断上下文：
 - ✓ 硬件中断
 - ✓ 软中断
 - ✓ tasklet
 - ✓ timer的callback函数

禁止抢占

2.6内核中禁止和激活抢占的定义：

preempt_enable() -- 抢占计数器减1

preempt_disable() -- 抢占计数器加1

注意：preempt_disable()/enable()调用可嵌套

- preempt_disable() 可以被调用 n 次，只有当第 n 次 preempt_enable() 被调用后，抢占才被重新激活**

作业（选做其一）：

1、寻找内核代码中对RCU的使用，深入剖析

2、对某一个同步机制的深入剖析

注：“深入”是对比课件的内容而言的，即比课件的内容要更深入