“菜鸟玩转嵌入式”视频培训讲座
— Linux驱动开发基础班

主办：上海申嵌信息科技有限公司
承办：嵌入式家园
协办：上海嵌入式家园-开发板商城
广州友善之臂计算机科技有限公司
主讲：贺光辉（嵌入式系统工程师）

嵌入式家园 www.embedclub.com

上海嵌入式家园-开发板商城 http://embedclub.taobao.com/

上章回顾

- 字符驱动程序开发的基本结构
  - 主设备号和次设备号，以及如何分配和释放设备号
  - cdev结构
  - file_operations结构
  - file结构
  - inode结构
- 如何把驱动添加到内核中？
- 应用程序如何使用驱动？

第3章

并发和竞态控制

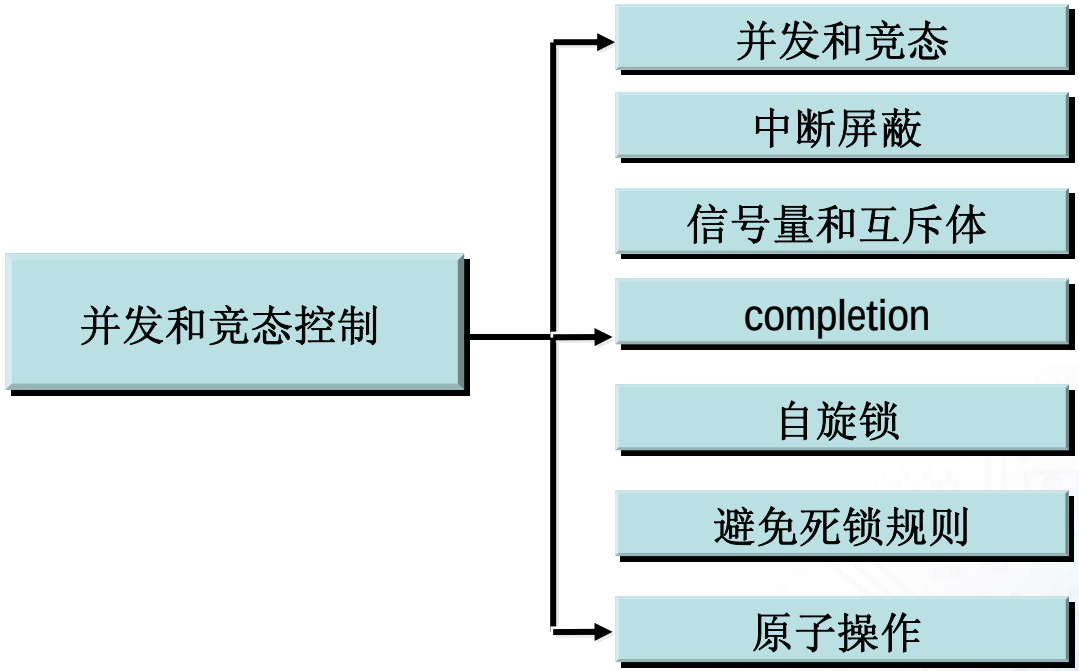
预习检查

- 防止竞态的机制有哪些？
  - semaphore, spinlock, completion,原子操作 等
- 原子操作的意义是什么？
  - 操作的不可分割性

本章目标

- 掌握信号量、completion、自旋锁和原子操作等并发控制机制

本章结构



3-1 并发和竞态

- 什么是并发？
  - 多个执行单元同时、并行被执行。
  - 竞态(race conditions)
    - 并发的执行单元对共享资源的访问则很容易导致竞态。
    - 共享资源:
      - 硬件资源，软件上的全局变量、静态变量等

- Linux内核中，什么情况会发生竞态？
  - 对称多处理器(SMP)的多个CPU之间的竞态
  - 单CPU内进程间的竞态
  - 中断(硬中断、软中断、Tasklet、底半部)与进程之间的竞态

- **如何解决竞态问题？**
  - 保证对共享资源的互斥访问，
    - 指一个执行单元在访问共享资源的时候，其他的执行单元被禁止访问。
  - Linux设备驱动中可采用的互斥途径
    - 中断屏蔽
    - 原子操作
    - 自旋锁
    - 信号量
    - completion

3-2 中断屏蔽

- **中断屏蔽：**
  - 可解决中断与进程之间的并发；
  - 也可解决内核抢占进程之间的并发

注意：不要长时间屏蔽中断，由于Linux系统的异步I/O、进程调度等很多重要操作都依赖于中断，在屏蔽中断期间所有的中断都无法得到处理，因此长时间屏蔽中断是很危险的，有可能造成数据丢失甚至系统崩溃。

```
local_irq_disable()//屏蔽中断
…
critical section //临界区
…
local_irq_enable()//打开中断
```

3-2 中断屏蔽实例分析

- linux-2.6.32.2\arch\arm\plat-s3c24xx\gpio.c

```
00139: void s3c2410_gpio_setpin(unsigned int pin, unsigned int to)
00140: {
00141:     void __iomem *base = S3C24XX_GPIO_BASE(pin);
00142:     unsigned long offs = S3C2410_GPIO_OFFSET(pin);
00143:     unsigned long flags;
00144:     unsigned long dat;
00145:
00146:     local_irq_save(flags);
00147:
00148:     dat = __raw_readl(base + 0x04);
00149:     dat &= ~(1 << offs);
00150:     dat |= to << offs;
00151:     __raw_writel(dat, base + 0x04);
00152:
00153:     local_irq_restore(flags);
00154: }
```

3-3 信号量和互斥体

- 信号量（ semaphore ）
  - 信号量本质上是一个整数值，
  - 一对操作函数通常称为P和V。
  - 进入临界区
    - 相关信号量上调用P；
    - 如果信号量>零，则该值会减小一，而进程可以继续。
    - 如果信号量的值=零(或更小)，进程必须等待直到其他人释放该信号量。
  - 退出临界区
    - 信号量的解锁通过调用V完成；
    - 该函数增加信号量的值，
    - 并在必要时唤醒等待的进程。

- 信号量用于互斥
  - 避免多个进程同时在一个临界区中运行
  - 信号量的值应初始化为1。
    - 只能由单个进程或线程拥有。
    - 一个信号量有时也称为一个“互斥体(mutex)”，
    - 它是互斥(mutual exclusion)的简称。
    - Linux内核中几乎所有的信号量均用于互斥。
    - 互斥体的取值只能为：1、0

初始化信号量

- 信号量类型为struct semaphore，定义在<linux/semaphore.h>；
- 信号量可通过几种途径来声明和初始化：
- **动态的初始化信号量**

信号量的初始值

```
void sema_init(struct semaphore *sem，int val);
```

- **静态的声明互斥信号量：**

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

声明互斥信号量“name”，并初始化为1

声明互斥信号量“name”，并初始化为0

- **动态的初始化互斥信号量：**

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

3-3 信号量和互斥体

- 获得信号量

减小信号量的值，如果不能获得信号量就一直等待

```
void down(struct semaphore *sem);
```

完成和down相同的工作，但操作是可中断的

```
int down_interruptible(struct semaphore *sem);
```

```
int down_trylock(struct semaphore *sem);
```

永远不会休眠，如果信号量在调用时不可获得， 就会立即返回一个非零值。

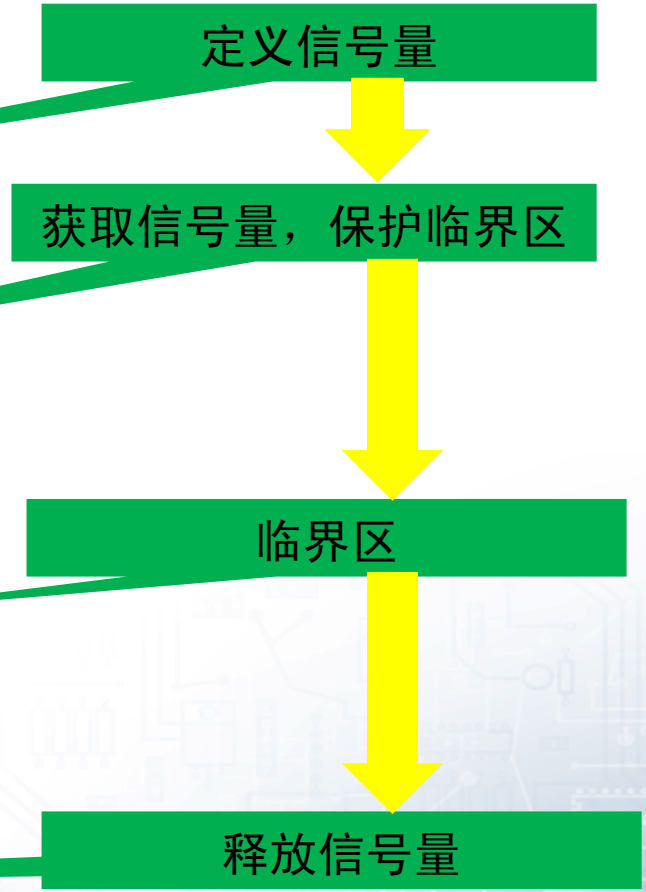
- 释放信号量

```
void up(struct semaphore *sem);
```

使用信号量模板

```
DECLARE_MUTEX(sem);

if(down_interruptible(&sem))
{
    return –ERESTARTSYS;
}
…
critical section
…
up(&sem);
```



3-3 互斥体实例讲解

- 互斥体使用实例-mouse driver
- linux-2.6.32.2\drivers\input\mousedev.c

```
static void mousedev_remove_chrdev(struct mousedev *mousedev)
{
    mutex_lock(&mousedev_table_mutex);
    mousedev_table[mousedev->minor] = NULL;
    mutex_unlock(&mousedev_table_mutex);
}
```

读取者/写入者信号量

- 读操作并发，写操作互斥。即一个rwsem可允许一个写入者或无限多个读取者拥有该信号量。
- 写入者优先级别更高，当有大量写入者竞争该信号量时，会导致读取者“饿死”，即长时间拒绝读者的访问。
- 但是驱动程序很少使用该机制。一般在 很少需要写访问且写入者只会短期拥有信号量的时候使用rwsem。
- 数据类型：struct rw_semaphore；

- 读取者/写入者信号量
  - 初始化：

```
void init_rwsem(struct rw_semaphore *sem);
```

  - 主要函数：

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);

void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
```

读取者/写入者信号量实例

```
rw_semaphore_t rw_sem;
init_rwsem(&rw_sem); //
//读时获取信号量
down_read(&rw_sem);
… //临界资源
up_read(&rw_sem);
 //写时获取信号量
down_write(&rw_sem);
… //临界资源
up_write(&rw_sem);
```

定义读写信号量

初始化读写信号量

获取临界资源

临界区

释放临界资源

3-4 completion

- 一种轻量级的机制
- 它允许一个线程告诉另一线程某个工作已经完成。
- 初始化

```
DECLARE_COMPLETION(xxx_completion);  // 静态创建

struct completion xxx_completion;  // 动态创建
init_completion(&xxx_completion);  // 初始化
```

- 等待completion

```
void wait_for_completion(struct completion *c);
```

3-4 completion

- 触发完成

```
void complete(struct completion *c)；//唤醒一个等待线程
void complete_all(struct completion *c)；// 唤醒所有等待线程
```

如果使用complete_all并想重复使用completion结构，则必须在重复使用该结构之前重新初始化它。下面这个宏可用来快速执行重新初始化：

INIT_COMPLETION(struct completion c)；

3-4 completion

completion使用示例

```
DECLARE_COMPLETION(xxx_comp);
ssize_t complete_read(struct file *filp,
                      char __user *buf, size_t count,
                      loff_t *pos)
{
    wait_for_completion(&xxx_comp);
    return 0;
}

ssize_t complete_write(struct file *filp,
                       const char __user *buf, size_t count,
                       loff_t *pos)
{
    complete(&xxx_comp);
    return count;
}
```

定义并初始化completion

等待某个事件完成

表示某个事件完成

3-5 自旋锁

概念

- 一个自旋锁是一个互斥设备，
- 只有两个值：“锁定”和“解锁”。
  - 通常实现为一个整数值中的单个位，希望获得某特定锁的代码,则测试相关的位。
  - 如果锁可用，则“锁定”位被设置，而代码继续进入临界区
  - 如果锁被其他人获得，则代码进入忙循环并重复检查这个锁，直到该锁可用为止。这个循环就是自旋锁的“自旋”

初始化自旋锁

编译时初始化：

```
spinlock_t xxx_lock=SPIN_LOCK_UNLOCKED;
```

运行时初始化：

```
void spin_lock_init(spinlock_t *lock);
```

3-5 自旋锁

锁定函数

```
void spin_lock(spinlock_t *lock);

//获得自旋锁之前禁止中断（包括软件中断和硬件中断）
void spin_lock_irq(spinlock_t *lock);

//获得自旋锁之前禁止中断（包括软件中断和硬件中断），中断状态保存在状态字flags中。
void spin_lock_irqsave(spinlock_t *lock，unsigned long flags);

// 获得自旋锁之前禁止软件中断，硬件中断保持打开
void spin_lock_bh(spinlock_t *lock);
```

释放自旋锁

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock，unsigned long flags);
void spin_unlock_bh(spinlock_t *lock);
```

3-5 自旋锁

自旋锁使用模板

```
spinlock_t lock;

spin_lock_init(&lock);

spin_lock(&lock);

critical section

spin_unlock(&lock);
```

定义一个自旋锁

初始化自旋锁

获取自旋锁，保护临界区

临界区

释放自旋锁

3-5 自旋锁实例讲解

- 自旋锁使用实例-mouse driver
- linux-2.6.32.2\drivers\input\mousedev.c

```
00819: static void mousedev_hangup(struct mousedev *mousedev)
00820: {
00821:         struct mousedev_client *client;
00822:
00823:         spin_lock(&mousedev->client_lock);
00824:         list_for_each_entry(client, &mousedev->client_list, node)
00825:                 kill_fasync(&client->fasync, SIGIO, POLL_HUP);
00826:         spin_unlock(&mousedev->client_lock);
00827:
00828:         wake_up_interruptible(&mousedev->wait);
00829: }
```

3-5 自旋锁

- 读写自旋锁
  - 任意数量的读取者可以同时进入临界区
  - 写入者必须互斥访问
  - 变量类型: **rwlock_t**

- 初始化

```
#include <linux/spinlock.h>
 rwlock_t xxx_rwlock = RW_LOCK_UNLOCKED；

或：
 rwlock_t xxx_rwlock；
 rwlock_init(&xxx_rwlock)；
```

- 读取者获得锁

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock，unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
```

- 读取者释放锁

```
void read unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock，unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

3-5 自旋锁

- 写入者获得锁

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock，unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);
```

- 写入者释放锁

```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock，unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

3-5 自旋锁

读写自旋锁实例

```
unsigned long flags;
rwlock_t rwlock;
rwlock_init(&rwlock);

read_lock(&rwlock);
... //临界资源
read_unlock(&rwlock);
//写时获取锁
write_lock_irqsave(&rwlock，flags);
... //临界资源
write_unlock_irqrestore(&rwlock，flags);
```

定义rwlock

初始化rwlock

获取锁

临界区

释放锁

- 自旋锁 VS信号量
  - 开销成本
    - 使用信号量的开销是进程上下文切换时间
    - 自旋锁的开销是忙等待获取自旋锁
  - 等待机制不同
    - 信号量可能导致阻塞，所以在不允许阻塞的代码中不能用可能引起阻塞的信号量处理方式
    - 自旋锁是忙等待

3-6 避免死锁规则

- 自旋锁的使用规则
  - 任何拥有自旋锁的代码都必须是原子的；
  - 如果中断处理函数中也要获得自旋锁，那么驱动程序需要在拥有自旋锁时禁止中断；
  - 自旋锁必须在可能的最短时间内拥有；

其他规则

- 避免某个获得锁的函数调用其他同样试图获取这个锁的函数，否则代码就会死锁；
- 不论是信号量还是自旋锁，都不允许锁拥有者第二次获得这个锁，如果试图这么做，系统将挂起；

3-6 避免死锁规则

- **锁的顺序规则**
  - 在必须获取多个锁时，应该始终以相同的顺序获得。这样能有效避免死锁。
  - 如果必须获得一个局部锁和一个属于内核更中心位置的锁，则应该首先获取自己的局部锁。
  - 如果我们拥有信号量和自旋锁的组合，则必须首先获得信号量；在拥有自旋锁时调用down(可导致休眠)是个严重的错误的 。

3-7 原子操作

- 整型原子操作
  - 类型: atomic_t
  - 设置原子变量的值

```
void atomic_set(atomic_t *v，int i)；// 动态初始化
atomic_t v = ATOMIC_INIT(0)；// 静态初始化
```

  - 获取原子变量的值

```
int atomic_read(atomic_t *v)；// 返回v当前值
```

  - 原子变量加/减

```
void atomic_add(int i，atomic_t *v)；// *v累加i，无返回值
void atomic_sub(int i，atomic_t *v)；// *v递减i，无返回值
```

  - 原子变量自增/自减

```
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
```

3-7 原子操作

- 操作并测试

```
// 操作结束后，原子值为0，则返回true；否则为false
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i，atomic_t *v);
```

- 操作并返回

```
// 操作结束后，返回新值
int atomic_add_return(int i，atomic_t *v);
int atomic_sub_return(int i，atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

位原子操作

```
void set_bit(nr, void *addr);
void clear_bit(nr, void *addr);
void change_bit(nr, void *addr);
int test_bit(nr, void *addr);
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

设置第nr个bit的值

清除第nr个bit的值

改变第nr个bit的值

检测第nrbit是否被设置

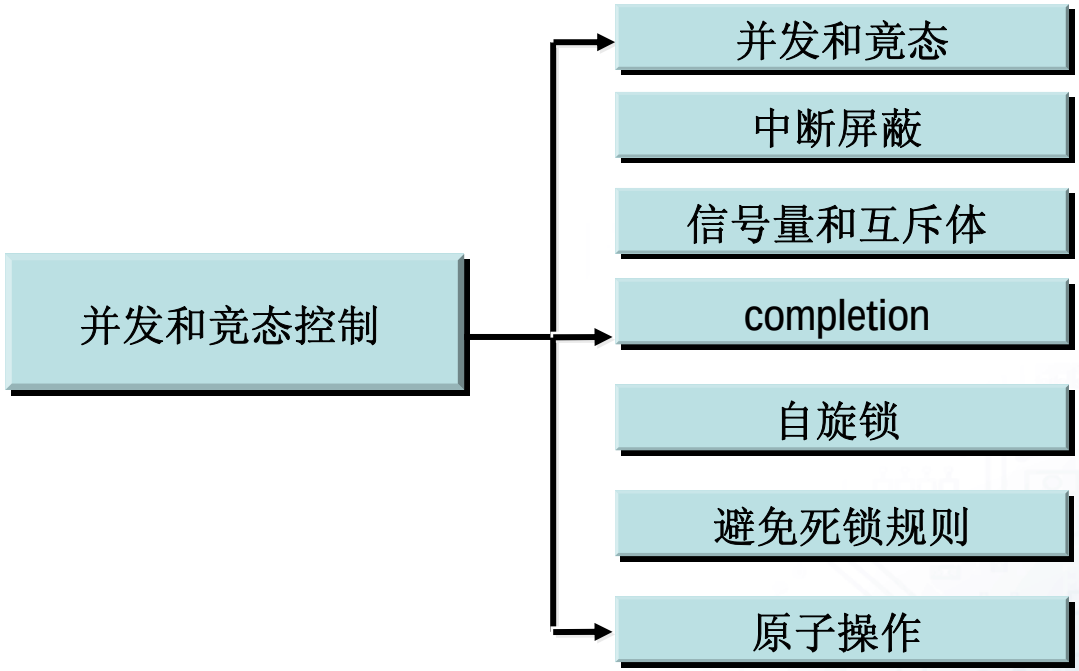
1、检测，并返回先前的值

2、再进行对应的操作

阶段总结

- 处理并发的方法：信号量、自旋锁、completion、中断屏蔽和seqlock等
- 信号量和自旋锁区别
- 如何避免死锁
- 原子操作

本章总结



实验

- 任务一、不加semaphore的多进程写试验
- 任务二、加semaphore的多进程写试验
- 任务三、修改memdev驱动，引入semaphore互斥机制