回顾：
1.linux内核中断编程
  面试题：谈谈对中断理解
  1.1.为什么有中断
      举例子
  1.2.中断硬件连接和触发过程
      画图
  1.3.中断的处理流程
      画图
  1.4.中断编程步骤
      四步骤
  1.5.linux内核中断编程
      request_irq/free_irq
  1.6.linux内核对中断处理函数的要求
  1.7.linux内核中断顶半部和底半部
  1.8.linux内核底半部实现方法
      tasklet
      工作队列
      软中断

2.linux内核软件定时器
  硬件定时器
  HZ
  jiffies
  数据结构：
  struct timer_list
  基于软中断
  不能休眠操作

3.linux内核延时方法
  忙延时
        ndelay/udelay/mdelay
  休眠延时
        msleep/ssleep/schedule/schedule_timeout

4.linux内核并发和竞态
  案例：一个设备要求只能被打开一次
  思路：
  方法1：在应用层实现
  方法2：在驱动层实现

  4.1.概念
  并发：多个执行单元(中断和进程)同时发生
  竞态：多个执行单元对共享资源的同时访问,形成的竞争状态
        三个条件：
        1.要有共享资源
        2.要有多个执行单元
        3.必须对共享资源同时访问

  共享资源：软件上的全局变量和硬件资源(硬件寄存器)
            例如：int open_cnt = 1; //全局变量
                  GPCOCON //硬件寄存器

  临界区：访问共享资源的代码区域
          例如：
          static int open_cnt = 1; //共享资源

```
static int led_open(...)
{
        //临界区
        if (--open_cnt != 0) {
                ...
        }
        //临界区结尾
}
```

互斥访问：当一个执行单元在访问临界区时,其他执行单元禁止
访问临界区,直到前一个执行单元访问完毕

执行路径具有原子性：当一个执行单元在访问临界区时,不允许
发生CPU资源的切换,保证这个执行单元
踏踏实实访问临界区

4.2.linux内核中形成竞态的4种情形：
1.多核(SMP),由于多核共享内存,闪存,IO资源
2.单CPU的进程与进程之前的抢占(高优先级的进程抢占低优先级进程的CPU资源)
3.中断和进程
  硬件和进程
  软中断和进程
4.中断和中断
  硬件和软中断
  软中断和软中断

4.3.linux内核解决竞态的方法
中断屏蔽
自旋锁
信号量
原子操作

4.4.linux内核解决竞态方法之中断屏蔽
特点：
1.中断屏蔽能够解决以下竞态问题：
  中断和进程
  中断和中断
  进程与进程的抢占(切记进程与进程的抢占基于软中断)

2.中断屏蔽无非保护的是临界区,当CPU执行临界区时,不允许中断
  进行来抢占CPU资源,但是由于是屏蔽了中断,而操作系统很多机制
  又跟中断密切相关,所以中断屏蔽保护的临界区的代码执行速度
  要快,更不能进行休眠操作

编程使用步骤：
1.明确驱动代码中哪些是共享资源
2.明确驱动代码中哪些是临界区
3.明确驱动代码中的临界区是否有休眠操作
  如果有,势必不能使用中断屏蔽此方法
  如果没有,可以考虑使用中断屏蔽
4.访问临界区之前屏蔽中断
  unsigned long flags
  local_irq_save(flags); //屏蔽中断,保存中断状态到flags

5.接下来可以踏踏实实的访问临界区,此时也不会发生CPU资源的切换

6.访问临界区完毕,一定要记得恢复中断
```
local_irq_restore(flags);
```

7.屏蔽中断和恢复中断一定要逻辑上配对使用!

参考代码:
底层驱动的led_open参考代码:
```
static int open_cnt = 1; //共享资源
static int led_open(struct inode *inode,
                            struct file *file)
{
  unsigned long flags;
  //屏蔽中断
  local_irq_save(flags);

  //临界区
  if (--open_cnt !=0 ){
          printk("设备已被打开!\n");
          open_cnt++;
          //恢复中断
          local_irq_restore(flags);
          return -EBUSY;//设备忙
  }

  //恢复中断
  local_irq_restore(flags);
  printk("设备打开成功!\n");
  return 0;
}
```

4.5.linux内核解决竞态方法之自旋锁
特点:
1.除了中断引起的竞态问题都可以进行解决
2.自旋锁必须附加在某个共享资源上
3.想访问临界区而没有获取自旋锁的任务将原地空转,原地忙等待
4.持有自旋锁的任务访问临界区时,执行速度要快,更不能做休眠操作
总结: 自旋锁保护的临界区不能进行休眠操作

数据类型: spinlock_t

编程操作步骤:
1.明确驱动代码中哪些是共享资源
2.明确驱动代码中哪些是临界区
3.明确临界区中是否有休眠
  如果没有,可以考虑使用自旋锁
  当然还要考虑是否有中断引起的竞态,如果有,同样不能
  使用自旋锁
4.定义自旋锁对象
```
spinlock_t lock;
```
5.初始化自旋锁对象
```
spinlock_init(&lock);
```
6.访问临界区之前先获取自旋锁
```
spin_lock(&lock); //任务获取自旋锁,立马返回
                  //如果没有获取自旋锁,任务在此忙等待
```
7.一旦获取自旋锁,踏踏实实的访问临界区
  注意: 临界区不能进行休眠操作

8.访问临界区之后,记得要释放自旋锁
  spin_unlock(&lock);
9.获取锁和释放锁在逻辑上要配对使用
参考代码：
//定义自旋锁对象
static spinlock_t lock;
入口函数调用：
     spin_lock_init(&lock);
底层驱动的led_open参考代码：
```
   static int open_cnt = 1; //共享资源
   static int led_open(struct inode *inode,
                               struct file *file)
   {
     unsigned long flags;
     //获取自旋锁
     spin_lock(&lock);

     //临界区
     if (--open_cnt !=0 ){
             printk("设备已被打开!\n");
             open_cnt++;
             //释放自旋锁
             spin_unlock(&lock);
             return -EBUSY;//设备忙
     }

     //释放自旋锁
     spin_unlock(&lock);
     printk("设备打开成功!\n");
     return 0;
   }
```

4.6.linux内核解决竞态方法之自旋锁扩展,又称衍生自旋锁
特点：
1.所有的竞态问题都能够解决
2.衍生自旋锁必须附加在某个共享资源上
3.想访问临界区而没有获取衍生自旋锁的任务将原地空转,原地忙等待
4.持有衍生自旋锁的任务访问临界区时,执行速度要快,更不能做休眠操作
总结：衍生自旋锁保护的临界区不能进行休眠操作

数据类型：spinlock_t

编程操作步骤：
1.明确驱动代码中哪些是共享资源
2.明确驱动代码中哪些是临界区
3.明确临界区中是否有休眠
  如果没有,可以考虑使用衍生自旋锁
4.定义自旋锁对象
  spinlock_t lock;
5.初始化自旋锁对象
  spinlock_init(&lock);
6.访问临界区之前先获取衍生自旋锁
```
  unsigned long flags
  spin_lock_irqsave(&lock,flags); //先屏蔽中断然后获取自旋锁
                                  //任务获取自旋锁,立马返回
                                  //如果没有获取自旋锁,任务在此忙等待
```

7.一旦获取自旋锁,踏踏实实的访问临界区
  注意：临界区不能进行休眠操作
8.访问临界区之后,记得要释放自旋锁,再恢复中断
  spin_unlock_irqrestore(&lock,flags);
9.获取锁和释放锁在逻辑上要配对使用
参考代码：
//定义自旋锁对象
static spinlock_t lock;
入口函数调用：
     spin_lock_init(&lock);
底层驱动的led_open参考代码：
```
   static int open_cnt = 1; //共享资源
   static int led_open(struct inode *inode,
                               struct file *file)
   {
     unsigned long flags;
     //获取自旋锁
     spin_lock_irqsave(&lock, flags);

     //临界区
     if (--open_cnt !=0 ){
             printk("设备已被打开!\n");
             open_cnt++;
             //释放自旋锁
             spin_unlock_irqrestore(&lock, flags);
             return -EBUSY;//设备忙
     }

     //释放自旋锁
     spin_unlock_irqrestore(&lock, flags);
     printk("设备打开成功!\n");
     return 0;
   }
```
  案例：利用混杂设备驱动,实现一个LED设备只能被打开一次
  ARM测试步骤：
  insmod /home/drivers/led_drv.ko
  ls /dev/myled -lh
  /home/led_test & //启动A进程
  ps //查看A进程的PID
  /home/led_test //启动B进程

4.7.linux内核解决竞态方法之信号量
特点：
1.本质就是解决自旋锁保护的临界区不能休眠的问题
2.信号量又称睡眠锁,本身基于自旋锁扩展而来
3.信号量保护的临界区可以进行休眠操作
4.要想访问临界区的任务,而没有获取到信号量,任务将进入休眠
  状态等待信号量被释放
5.信号量一般应用于进程

数据类型：struct semaphore

编程操作步骤：
1.明确驱动中哪些是共享资源
2.明确驱动中哪些是临界区
3.明确临界区中是否有休眠操作

如果有,使用信号量
如果没有,可以考虑使用信号量
4.定义信号量对象
struct semaphore sema;
5.初始化信号量为互斥信号量
sema_init(&sema, 1);
6.访问临界区之前获取信号量
down(&sema);//获取信号量,如果正常获取信号量,此函数立即返回
//如果没有获取信号量,进程将进入不可中断的休眠状态
//代码停止不动,进程等待被唤醒,唤醒的方法是正在获取
信号量的任务释放信号量,同时也会唤醒这个休眠的进程
//A获取信号量,B进程在此进入不可中断的休眠状态(睡眠期间不会立即响应和处理信号)
//A释放信号量,同时唤醒B,B进程被唤醒以后,需要处理之前接受到的信号
或者
down_interruptible(&sema);//获取信号量,如果正常获取信号量,此函数立即返回
//如果没有获取信号量,进程将进入可中断的休眠状态
//代码停止不动,进程等待被唤醒,唤醒的方法是有两种:
1.获取信号量的任务进行唤醒
2.接收到了信号进行唤醒
//A获取信号量,B进程在此进入可中断的休眠状态(睡眠期间会立即响应和处理信号)
//A释放信号量,同时唤醒B

7.一旦获取信号量成功,踏踏实实的访问临界区
8.访问完毕,记得要释放信号量
up(&sema); //不仅仅会释放信号量,还要唤醒之前休眠的进程

案例:利用信号量,实现一个LED设备只能被打开一次
采用down来获取信号量
强调:
1.如果没有获取信号量,进程进入不可中断的休眠状态(休眠期间不会立即响应和处理信号)
2.此休眠进程被唤醒的方法只有1个:
只能由A进程释放信号量时唤醒此休眠的进程
此进程一旦被唤醒,还要处理之前接受到的信号
ARM实验步骤:
insmod /home/drivers/led_drv.ko
/home/drivers/led_test & //启动A进程
/home/drivers/led_test & //启动B进程
ps //查看A,B的PID
top //查看A,B进程的状态,按Q键退出top命令
S:可中断的休眠状态
D:不可中断的休眠状态
R:运行状态

kill B进程的PID //向休眠中的B进程发送信号
ps //查看B是否被干掉
kill A进程的PID //杀死A进程,A进程释放信号量,唤醒B进程

/home/drivers/led_test & //启动A进程
/home/drivers/led_test & //启动B进程
ps //查看A,B的PID
top //查看A,B进程的状态,按Q键退出top命令
kill A进程的PID //杀死A进程,A进程释放信号量,唤醒B进程
top //查看B进程的状态信息

kill B进程PID
    结果是:D->S(应用调用sleep)

采用down_interruptible来获取信号量
强调:
1.没有获取信号量,进程进入可中断的休眠状态
2.可中断的休眠状态表示休眠期间可以响应处理接收到的信号
3.可中断休眠进程被唤醒的方法:
  1.通过信号唤醒
  2.通过A进程释放信号量唤醒

insmod /home/drivers/led_drv.ko
/home/drivers/led_test & //启动A进程
/home/drivers/led_test & //启动B进程
ps //查看A,B的PID
top //查看A,B进程的状态,按Q键退出top命令
    S:可中断的休眠状态
    D:不可中断的休眠状态
    R:运行状态

kill B进程的PID //向休眠中的B进程发送信号
ps //查看B是否被干掉
kill A进程的PID //杀死A进程,A进程释放信号量,唤醒B进程

/home/drivers/led_test & //启动A进程
/home/drivers/led_test & //启动B进程
ps //查看A,B的PID
top //查看A,B进程的状态,按Q键退出top命令
kill A进程的PID //杀死A进程,A进程释放信号量,唤醒B进程
top //查看B进程的状态信息
kill B进程PID
    结果是:D->S(应用调用sleep)