回顾：
1.linux内核并发和竞态
1.1.概念
并发
竞态
共享资源
临界区
互斥访问
执行路径具有原子性

1.2.形成竞态的4中情形
SMP
单CPU,进程与进程的抢占
中断和进程
中断和中断
画图

1.3.解决竞态问题的方法
中断屏蔽
自旋锁
衍生自旋锁
信号量

场景：设置某个GPIO的高低电平的时间为严格的500us
此时要考虑到竞态问题,一般来说中断最会捣鬼!

```
spin_lock_irqsave(&lock ,flags);
gpio_direction_output(., 1);
udelay(500);
gpio_direction_output(., 0);
udelay(500);
spin_unlock_irqrestore(&lock, flags);
```

1.4.原子操作
特点：
原子操作能够解决所有的竞态问题
原子操作分两类：位原子操作和整形原子操作

位原子操作=位操作具有原子性,位操作期间不允许发生CPU资源切换

使用位原子操作的场景：如果驱动对共享资源进行位操作,并且考虑
到竞态问题,此时可以考虑使用内核提供的位原子操作来避免竞态问题
但是对共享资源的位操作必须使用内核提供的相关函数：

```
void set_bit(int nr, void *addr)
void clear_bit(int nr, void *addr)
void change_bit(int nr, void *addr)
int test_bit(int nr, void *addr)
...
```

addr:共享资源的首地址
nr:第几个bit位(从0开始)

参考代码：

```
static int open_cnt = 1; //共享资源

//临界区
open_cnt &= ~(1 << 1); //不具有原子性,会发生CPU资源的切换
```

改造,考虑竞态问题：
方案1：
```
local_irq_save(flags);
open_cnt &= ~(1 << 1);
local_irq_restore(flags);
```

方案2：
```
spin_lock_irqsave(&lock, flags);
open_cnt &= ~(1 << 1);
spin_unlock_irqrestore(&lock, flags);
```

方案3：
```
down(&sema);
open_cnt &= ~(1 << 1);
up(&sema);
```

方案4：
```
clear_bit(&open_cnt, 1);
```

案例：加载驱动,在驱动入口函数将0x5555数据变成0xaaaa
不允许使用change_bit函数

整形原子操作 = 整形操作具有原子性
使用场景：如果驱动中对共享资源进行整型数的操作,并且考虑到
竞态问题,可以考虑使用整形原子操作

整形原子变量数据类型：atomic_t(类比成int)

编程步骤：
1.定义初始化整型原子变量
```
  atomic_t tv = ATOMIC_INIT(1);
```
2.对整形原子变量操作,内核提供了相关的配套函数
```
  atomic_add
  atomic_sub
  atomic_inc
  atomic_dec
  atomic_set
  atomic_read
  atomic_dec_and_test(&tv) //整形原子变量tv自减1,然后判断tv的值
  是否为0,如果为0,返回真;否则返回假

  参考代码：
  static int open_cnt = 1;

  //临界区
  --open_cnt; //不具有原子性
```

改造,添加方法：
方案1：
```
local_irq_save(flags);
 --open_cnt
local_irq_restore(flags);
```

方案2：
```
spin_lock_irqsave(&lock, flags);
```

```
 --open_cnt
spin_unlock_irqrestore(&lock, flags);

方案3：
down(&sema);
--open_cnt
up(&sema);

方案4：整形原子操作
static atomic_t open_cnt = ATOMIC_INIT(1);
atomic_dec(&open_cnt);
```

****

2.linux内核等待队列机制

2.1.等待分两种

忙等待：CPU原地空转,等待时间比较短的场合

休眠等待：专指进程,进程等待某个事件发生进入休眠状态

等待队列机制研究休眠等待!

2.2.等待的本质

由于外设的处理速度慢于CPU,当某个进程要操作访问硬件外设

当硬件外设没有准备就绪,那么此进程将进入休眠等待,那么进程

会释放CPU资源给其他任务使用,直到外设准备好数据(外设会给CPU发送中断信号),唤醒之前

休眠等待的进程,进程被唤醒以后即可操作访问硬件外设

以CPU读取UART数据为例,理理数据的整个操作流程：

1.应用程序调用read,最终进程通过软中断由用户空间
  陷入内核空间的底层驱动read接口

2.进程进入底层UART的read接口发现接收缓冲区数据没有准备
  就绪,此进程释放CPU资源进入休眠等待状态,此时代码停止不前
  等待UART缓冲区来数据

3.如果在某个时刻,UART接收到数据,最终势必给CPU发送一个中断信号
  内核调用其中断处理函数,只需在中断处理函数中唤醒之前
  休眠的进程

4.休眠的进程一旦被唤醒,进程继续执行底层驱动的read接口
  read接口将接收缓冲区的数据最终上报给用户空间

问：如何让进程在内核空间休眠呢？

答：

利用已学的休眠函数：msleep/ssleep/schedule/schedule_timeout

这些函数的缺点都需要指定一个休眠超时时间,不能够随时随地休眠

随时随地被唤醒!

问：如何让进程在内核空间随时随地休眠,随时随地被唤醒呢？

答：利用等待队列机制

msleep/ssleep/信号量这些休眠机制都是利用等待队列实现!

2.4.等待队列和工作队列对比

工作队列是底半部的一个实现方法,本质让事情延后执行

等待队列是让进程在内核空间进行休眠唤醒

2.5.利用等待队列实现进程在驱动中休眠的编程步骤：

老鹰<------->进程的调度器(给进程分配CPU资源,时间片,切换,抢占),此代码有内核已经实现

鸡妈妈<------>等待队列头,所代表的等待队列中每一个节点表示的是要休眠的进程

只要进程休眠,只需把休眠的进程放到鸡妈妈所对应的等待队列中
小鸡<----->每一个休眠的进程,一个休眠的进程对应的是一个小鸡

linux内核进程状态的宏：
进程的运行状态：TASK_RUNNING
进程的休眠状态：
不可中断的休眠状态：TASK_UNINTERRUPTIBLE
可中断的休眠状态：TASK_INTERRUPTIBLE
进程的准备就绪状态：TASK_READY

编程操作步骤：
1.定义初始化等待队列头对象(构造一个鸡妈妈)
```
wait_queue_head_t wq; //定义
init_waitqueue_head(&wq); //初始化
```

2.定义初始化装载休眠进程的容器(构造一个小鸡)
```
wait_queue_t wait; //定义一个装载休眠进程的容器
init_waitqueue_entry(&wait, current);//将当前进程添加到wait容器中
                                     //此时当前进程还么以后休眠
```
“当前进程”：正在获取CPU资源执行的进程,当前进程是一个动态变化的
current：内核全局指针变量,对应的数据类型：
```
struct task_struct {
    pid_t pid;//进程号
    char comm[TASK_COMM_LEN];//进程的名称
    ...
};//此数据结构就是描述linux系统进程
```
只要创建一个进程,内核就会帮你创建一个task_struct
对象来描述你创建的这个进程信息
current指针就是指向当前进程对应的task_struct对象
打印当前进程的PID和名称：
```
printk("当前进程[%s]PID[%d]\n",
        current->comm, current->pid);
```

注意：一个休眠的进程要有一个对应的容器wait!

3.将休眠的进程添加到等待队列中去(将小鸡添加到鸡妈妈的后面)
```
add_wait_queue(&wq, &wait);
```

4.设置进程的休眠状态
```
set_current_state(TASK_INTERRUPTIBLE);//可中断的休眠状态
```
或者
```
set_current_state(TASK_UNINTERRUPTIBLE); //不可中断的休眠状态
```
//此时进程还没有休眠

5.当前进程进入真正的休眠状态,一旦进入休眠状态,代码
停止不前,等待被唤醒
```
schedule(); //休眠然后等待被唤醒
```
对于可中断的休眠状态,唤醒的方法有两种：
1.接收到了信号引起唤醒
2.驱动主动唤醒(数据到来,中断处理函数中进行唤醒)

对于不可中断的休眠状态，唤醒的方法有一种：
1.驱动主动唤醒

6. 一旦进程被唤醒,设置进程的状态为运行,并且将当前进程
从等待队列中移除

```
set_current_state(TASK_RUNNING);
remove_wait_queue(&wq, &wait);
```

7.一旦被唤醒,一般还要判断唤醒的原因

```
if(signal_pending(current)) {
    printk("进程由于接收到了信号引起的唤醒!\n");
    return -ERESTARTSYS;
} else {
    printk("驱动主动唤醒!\n");
    //说明硬件数据准备就绪
    //进程继续操作硬件
    copy_to_user//将数据上报给用户空间
}
```

8.驱动主动唤醒的方法：

```
wake_up(&wq); //唤醒wq所对应的等待队列中所有的进程
或者
wake_up_interruptible(&wq); //只唤醒休眠类型为可中断的进程
```

案例：写进程唤醒读进程

ARM测试步骤：

```
insmod /home/drivers/btn_drv.ko
/home/drivers/btn_test r & //启动读进程
ps  //查看PID
top //查看休眠类型
/home/drivers/btn_test w   //启动写进程
```

案例：编写按键驱动,给用户应用程序上报按键状态和按键值

分析：

1.应用程序获取按键的状态和按键值
  应用程序调用read或者ioctl获取这些信息
2.如果按键没有操作,应用程序应该进入休眠等待按键有操作
3.一旦按键有操作,势必给CPU发送中断信号,此时唤醒休眠的
  进程,进程再去读取按键的信息上报给用户

测试步骤：

```
insmod /home/drivers/btn_drv.ko
ls /dev/mybtn -lh
cat /proc/interrupts //查看中断的注册信息
```

3.按键去除抖动

由于按键的机械结构,按键的质量问题造成按键存在抖动
按键抖动实际的波形
去除抖动的方法：
硬件去抖动
软件去抖动：宗旨延时
每次上升沿和下降沿的时间间隔经验值5~10ms
单片机裸板开发,采用忙延时,浪费CPU资源
linux内核同样采用延时,延时采用定时器进行延时

编写驱动步骤：

1.先头文件
2.该声明的声明,该定义的定义,该初始化的初始化
  先搞硬件再弄软件

```
  struct btn_event
  struct btn_resource

  struct btn_event g_data;
  struct btn_resource btn_info...

  struct file_operations ...
  struct miscdevice ...
```

3.填充入口和出口
  但凡初始化工作都在入口
  出口跟入口对着干
4.最后完成各个接口函数
  .open
  .release
  .read
  .write
  .unlocked_ioctl
  .mmap
  编写接口函数一定要根据实际的用户需求来完成

  如果有中断,编写中断处理函数
  注意：中断处理函数和各个接口函数之间的关系(数据传递)

按键程序的执行流程：
应用read->驱动read,睡眠->按键按下->产生中断->内核调中断处理函数
->中断处理函数准备上报的数据,唤醒进程->read进程唤醒,继续执行-》
把中断准备好的数据给用户