回顾：
1.linux内核中断编程
面试题：谈谈对中断的理解
1.为什么有中断机制
举例子
2.中断的硬件连接
画图
3.中断的处理过程
画图
4.中断的软件编程
四部曲
5.linux内核中断编程
目标：只需向内核注册硬件中断的中断处理函数即可
大名鼎鼎函数：requeset_irq/free_irq
int request_irq(中断号,中断处理函数,中断标志,中断名称,给中断处理函数传递的参数)
void free_irq(中断号,传递参数)
6.linux内核中断编程的注意事项
中断不隶属于任何进程,不参与进程调度
硬件中断>软中断>进程
要求中断处理函数执行的速度越快越好,目的提高系统的相应能力和并发能力
更不能进行休眠操作
问：有些场合无法满足这种要求,此时可以考虑使用内核提供的顶半部和底半部机制来优化
7.中断编程顶半部和底半部机制
这种机制的本质目的就是及时释放CPU资源,让其他高优先级的
任务及时获取到CPU资源,投入运行;
底半部机制本质就是延后执行,就是将不重要的事情往后拖延,以后去执行
底半部并不一定非要和顶半部配合使用

顶半部本质就是中断处理函数,紧急,耗时较短的内容,不可被中断
底半部执行不紧急,耗时较长的内容,可以中断
画图

底半部实现方法：
tasklet
工作队列
软中断
8.底半部实现方法之tasklet
特点：
中断上下文：就是中断的整个处理的过程(跳转到异常向量表,保护现场,执行中断处理函数,恢复现场)
进程上下文：就是进程的整个处理的过程(进程的创建,调度,抢占,进程休眠,进程的唤醒,进程的退出)
基于软中断实现,优先级高于进程低于硬件中断,延后处理函数不能进行休眠操作
tasklet延后处理函数工作在中断上下文
本质是延后执行的一种手段

数据结构：
```
struct tasklet_struct
    .function //延后处理函数,一旦登记完成,内核会在适当的时候去执行
              不能进行休眠操作
    .data //给延后处理函数传递的参数
```

配套函数：
DECLARE_TASKLET(name,func,data)//定义初始化对象

```
tasklet_schedule(&name);//登记延后处理函数
```

9.底半部实现方法之工作队列
1.特点：
工作队列的延后处理函数工作在进程上下文,所以此函数可以进行
休眠操作,参与进程之间的调度
优先级低于中断
工作队列的本质也是延后执行的一种手段
总结：
延后执行的内容中如果有休眠操作必须使用工作队列
延后执行的内容中没有休眠,三种方法都可以使用,但是如果考虑
到效率问题,使用tasklet或者软中断

2.数据结构

```
struct work_struct {
    void (*function)(struct work_struct *work);
    ...
};
```

function:工作队列的延后处理函数,工作在进程上下文,所以可以进行
休眠操作
形参work指针指向驱动自己定义初始化的工作对象
问：工作队列如何给延后处理函数传递参数呢？
答：认真自行研究内核大名鼎鼎的宏：container_of

配套函数：

```
//给工作对象添加一个延后处理(初始化对象)
INIT_WORK(&工作对象名,延后处理函数);
schedule_work(&工作对象名); //向内核登记工作的延后处理函数
```

一旦登记完成,内核会在适当的时候去执行此函数

编程步骤：
1.定义工作对象

```
struct work_struct work;
```

2.初始化工作对象,添加延后处理函数

```
INIT_WORK(&work, xxx_work_function);
```

3.在适当的位置进行登记延后处理函数

```
schedule_work(&work);
```

总结：
如果有顶半部,一般在顶半部登记
如果没有顶半部,何时何地登记随意

案例：利用工作队列,"优化"按键驱动
打印按键信息放在底半部工作队列的延后处理函数中执行

10.底半部实现方法之软中断(了解)
特点：
1.软中断的延后处理函数工作在中断上下文中,不能进行休眠操作
2.tasklet基于软中断实现
3.软中断的延后处理函数可以同时运行在多个CPU上,但是tasklet
的延后处理函数只能运行在一个CPU上;
所以软中断的延后处理函数必须具备可重入性(可重入函数)

```
int g_data
swap1(int *a, int *b)
{
    g_data = *b;
    *b = *a;
```

```
    *a = g_data;
} //不具有可重入性

swap2(int *a, int *b)
{
    int data;

    data = *b;
    *b = *a;
    *a = data;
} //具有可重入性
```

总结：
1.尽量避免访问全局变量
2.如果要访问全局变量,要注意互斥访问,但是代码的执行效率降低

4.软中断代码的实现过程不能采用insmod/rmmod动态加载和卸载只能静态编译到内核中(zImage,在一起),不便于代码的维护
5.软中断的本质就是延后执行

2.linux内核软件定时器
2.1.了解计算机的硬件定时器
特点：一旦上电,硬件定时器周期性的按照一定的频率给CPU发送中断信号
这个中断又称时钟中断,或者定时器中断;
中断触发的频率可以软件编程设置!

2.2.了解linux内核对应的定时器中断的中断处理函数
1.此函数由内核已经实现好
cd /opt/kernel
vim arch/arm/mach-s5pv210/mach-cw210.c
.init_timer=sys_timer //跟踪进入找到对应的中断处理函数
2.此函数会周期性,按照一定的频率被内核调用
3.此函数将做一下几个事情：
1.更新系统的运行时间
2.更新系统的实际时间(又称wall time)
3.检查进程的时间片是否用尽,决定是否启动调度
4.检查内核是否有超时的软件定时器,如果有超时的软件定时器内核调用超时的软件定时器的超时处理函数
5.统计的CPU利用率,内存的使用率等等系统资源

2.3.linux内核中时间相关概念
1.HZ
内核常量,ARM架构HZ=100,X86架构HZ=1000
例如HZ=100,表示硬件定时器1秒钟给CPU发送100个定时器中断信号
每发生一次中断的时间间隔为10ms
2.jiffies_64,jiffies
jiffies_64是内核全局变量,64位,unsigned long long,
记录自开机以来发生了多少次的定时器中断,每发生一次,"自动"加1

jiffies也是内核全局变量,32位,unsigned long,值取得jiffies_64的低32位，每发生一次定时器中断,同样"自动"加1,一般用来记录流失时间(时间间隔)
注意：只要在内核代码中看到jiffies,就等于此刻为当前时间

参考代码：

```
unsigned long timeout = jiffies + 5*HZ;
```

说明：

jiffies:表示当前时间
5*HZ:时间间隔为5秒
timeout:表示5秒以后的时间,也就是超时时间

参考代码：判断是否超时
```
unsigned long timeout = jiffies + 5*HZ;
...
...
... //若干条代码,代码执行需要时间
if (jiffies > timeout)
  超时
else
  没有超时
```

解决方法：
```
if(time_after(jiffies, timeout)) {
  超时
} else {
  没有超时
}//此函数无需记忆,只需看大神如何判断即可
```

2.4.linux内核软件定时器
特点：
1.内核软件定时器基于软中断实现
2.内核软件定时器对应的超时处理函数不能进行休眠操作
3.内核软件定时器到期以后,内核会调用对应的超时处理函数完成某个用户的业务

数据结构:
```
struct timer_list {
      unsigned long expirs; //超时时候的时间,例如超时时间间隔为5s
                            expires = jiffies + 5*HZ;
      void (*function)(unsigned long data) //超时处理函数,不能进行休眠操作
      unsigned long data; //给超时处理函数传递的参数,一般传递指针
      ...
};
```

配套函数：
```
init_timer(&定时器对象);//初始化定时器对象
add_timer(&定时器对象);//向内核注册添加一个定时器对象
                        一旦添加完毕,内核开始对此定时器进行倒计时
                        超时时间到期,内核调用对应的超时处理函数,并且
                        内核将定时器对象从内核中删除
del_timer(&定时器对象); //删除定时器
mod_timer(&定时器对象, 新的超时时候的时间);//修改定时器
```
此函数等价于一下三步骤：
1.先删除之前的定时器del_timer
2.在修改新的超时时间expires = ...
3.重新添加定时器add_timer

编程步骤：
1.定义定时器对象
```
  struct timer_list mytimer;
```
2.初始化定时器对象
```
  init_timer(&mytimer);
  //需要额外自己初始化以下三个字段：
  expires = jiffies + 5*HZ;
```

```
  function = mytimer_function;
  data = (unsigned long)&...;
3.注册定时器对象到内核
  add_timer(&mytimer);
4.不再使用定时器删除
  del_timer(&mytimer);
5.mod_timer(&mytimer, jiffies + 20*HZ);

案例1：利用定时器,实现每隔2000ms打印一句话
案例2：利用定时器,实现每隔2000ms开关灯
案例3：利用定时器,实现能够动态修改灯的闪烁频率,例如100ms
       500ms，1000ms，2000ms,提示不允许使用字符设备驱动和
       混杂设备驱动编程框架,只需采用内核程序的命令行传参实现即可
       驱动加载以后,灯的闪烁频率可以修改
       提示：module_param(name,type,权限);
       提示：毫秒ms转中断触发的次数
       msecs_to_jiffies(毫秒数);
       注意：不允许使用if...else判断

3.linux内核延时方法
  linux内核延时分两种：忙延时和休眠延时
  "忙延时"：CPU原地空转,打转,应用在等待延时极短的场合
            中断和进程都可以使用忙延时
  "休眠延时"：当前进程释放所占用的CPU资源给其他任务使用
              仅适用于进程,延时时间比较长的场合

  忙延时的函数：
  ndelay(纳秒数);//纳秒级延时
  例如：ndelay(100);//忙延时100纳秒

  udelay(微秒数);
  例如：udelay(100); //忙延时100微秒

  mdelay(毫秒数);
  例如：mdelay(5); //忙延时5毫米
  注意：如果忙延时的时间超过10ms,建议采用休眠延时

  休眠延时的函数：
  msleep(毫秒数);
  例如：msleep(20); //休眠延时20毫秒

  ssleep(秒数);
  例如：ssleep(20);

  schedule(); //永久性休眠

  scheudle_timeout(5*HZ); //休眠延时5秒

**********************************************************
4.linux内核并发和竞态相关内容
面试题：谈谈进程间通信的方法
案例：要求一个LED设备只能被打开一次
分析实现过程：
方案1:应用层实现
      采用进程间通信机制,实现多个进程之间通信决定是否打开设备
      A打开,B问A,C问A...
```

缺点：不太灵活

方案2:驱动层实现
明确：不管应用层有多少进程,访问设备永远先open
它们最终都会调用底层驱动的led_open,只需在底层驱动的
led_open代码中做相关的限定即可
”一夫当关万夫莫开“

底层驱动的led_open参考代码：

```
static int open_cnt = 1;
static int led_open(struct inode *inode,
                            struct file *file)
{
  if (--open_cnt !=0 ){
          printk("设备已被打开!\n");
          open_cnt++;
          return -EBUSY;//设备忙
  }
  printk("设备打开成功!\n");
  return 0;
}
static int led_close(struct inode *inode,
                            struct file *file)
{
  open_cnt++;
  return 0;
}
```

代码分析：研究代码片段if (--open_cnt !=0)
汇编访问变量：
ldr 加载
sub 运算
str 存储
正常情况：
A进程打开设备：
读取：open_cnt=1
修改,写回：open_cnt=0
结果：打开设备成功

B进程打开设备：
读取：open_cnt=0
修改，写回：open_cnt=-1
结果：打开设备失败

异常情况：
A进程先打开：
读取:open_cnt=1
就在此时此刻,并且由于linux系统允许进程之前进行抢占(进程有优先级之分)
高优先级的B进程开始投入运行,并且此时抢占A进程的CPU资源,B进程开始执行：

B进程开始执行：
读取：open_cnt=1
修改,写回：open_cnt=0
结果：打开设备成功
B进程执行完毕,B进程释放CPU资源给A进程,A进程继续执行

A进程继续执行：
    修改,写回：open_cnt=0
    结果：打开设备成功

结论：
1.发生这种抢占的概率极其之低
2.产生这种异常的根本原因是linux系统进程与进程的抢占
  linux内核允许进程进行抢占而提高系统的实时性
  高优先级的进程抢占低优先级的进程的CPU资源
  本质：还是优先级问题
3.问：linux内核中产生类似以上异常的情况还有哪些？
  答：
  1.进程与进程之间的抢占
  2.中断和进程
  3.中断和中断
  4.SMP