一.嵌入式linux设备驱动开发相关内容
<<Linux设备驱动程序>>第三版
<<Linux内核设计与实现>>第三版
<<跟我一起写Makefile>>电子档

关于SecureCRT远程登录linux系统的配置过程：
打开快速连接->协议：ssh2
主机名：192.168.1.8
用户名：tarena
->保存->输入密码->配置SecureCRT->会话选项->仿真->终端：ANSI
使用颜色方案选中
->外观：设置自己喜欢的字体Courier New
字符编码：UTF-8
保存
->重新SecureCRT即可

面试题：如何开发一个linux硬件设备驱动?
友情提示：嵌入式linux系统一旦运行起来以后,要花更多的时间
和精力放在开发板上的外设硬件的设备驱动程序上,
如果这个外设有了驱动,需要进行测试,测试的前提是
你要看得懂;
如果这个外设没有驱动,要进行这个外设硬件的设备驱动开发

1. 设备驱动概念
一个驱动的关键两个内容：
1.将硬件的整个操作过程进行封装
2.必须能够给用户提供访问操作硬件的接口(函数)
将来用户调用函数能够随便访问硬件

2.linux系统的两个空间(两种状态)：用户空间和内核空间(了解即可)
用户空间：
又称用户态
包含的软件就是各种命令,各种应用程序,各种库,各种配置服务等
用户空间的软件在运行的时候,CPU的工作模式为USER模式
用户空间的软件不能访问硬件设备的物理地址,如果要访问硬件物理地址
必须将硬件外设的物理地址映射到用户空间的虚拟地址上
用户空间的软件不能直接访问内核空间的代码,地址和数据
用户空间的软件如果进行非法的内存访问,不会导致操作系统崩溃
但是应用软件会被操作系统干掉(例如：*(int *)0=0)
用户空间的软件类似论坛的普通用户
用户空间的虚拟地址空间大小为3G(0x00000000~0xBFFFFFFF)
内核空间：
又称内核态
内核空间的软件就是内核源码(zImage)
内核代码运行时,CPU的工作模式为SVC模式
内核空间代码同样不能访问硬件外设的物理地址,必须将物理地址
映射到内核空间的虚拟地址上
内核代码如果进行非法的内存访问,操作系统会直接崩溃(吐核)
(例如：*(int *)0=0)
内核空间的软件类似论坛的管理员
内核空间的虚拟地址空间大小为1G(0xC0000000~0xFFFFFFFF)

3.linux系统设备驱动分类
字符设备驱动
字符设备访问时按照字节流形式访问
例如：LED,按键,UART接口设备(BT,GPS,GPRS),触摸屏

LCD,声卡,摄像头,各种传感器

块设备驱动

块设备访问时按照一定的数据块进行访问,数据块比如为512字节,1KB字节

例如：硬盘,U盘,SD卡,TF卡,Nor,Nand等

网络设备驱动

网络设备驱动访问时需要配合TCP/IP协议栈进行访问

驱动一般由芯片厂家提供,驱动开发者需要进行移植

例如：DM9000网卡基地址

4.linux字符设备驱动开发相关内容

4.0.明确不管什么驱动,它们都是内核程序

4.1.明确linux内核程序编程基本框架

回忆：应用程序编程框架

vim helloworld.c

```
#include <stdio.h> //标准C头文件

//main：程序的入口函数
int main(int argc, char *argv[])
{
	//标准C库函数
	printf("hello,world\n");

	//程序的出口
	return 0;
}
```

内核程序参考代码：

```
mkdir /opt/drivers/day01/1.0 -p
cd /opt/drivers/day01/1.0
vim helloworld.c 添加第一个内核程序
#include <linux/init.h>
#include <linux/module.h>
static int helloworld_init(void)
{
	printk("hello,world\n");
	return 0;
}
static void helloworld_exit(void)
{
	printk("good bye world!\n");
}
module_init(helloworld_init);
module_exit(helloworld_exit);
MODULE_LICENSE("GPL");
```

说明：

1.内核程序使用的头文件位于linux内核源码中(/opt/kernel)

2.内核程序的入口函数需要使用module_init宏进行修饰,例如
helloworld_init函数就是此内核程序的入口函数,将来加载
安装驱动时(insmod),内核会调用此函数;
此函数的返回值必须为int型,执行成功返回0,执行失败返回负值

3.内核程序的出口函数需要使用module_exit宏进行修饰,例如
helloworld_exit函数就是此内核程序的出口函数,将来卸载
驱动(rmmod)时,内核会调用此函数

4.任何一个内核程序源码(.c结尾)必须添加MODULE_LICENSE("GPL")

这句话,就是告诉内核,此内核程序同样遵循GPL协议,否则后果很严重

5.内核打印函数使用printk,此函数定义不再C库中,而是在内核源码中

6.结论：编译内核程序肯定需要关联内核源码

4.2.内核程序的编译

回顾：应用程序的编译

```
gcc -o helloworld helloworld.c
编写Makefile,make编译即可
```

内核程序编译：

回顾led_drv.c编译步骤：

1.静态编译

```
拷贝内核程序到内核源码中
修改Kconfig
修改Makfile
make menuconfig //选择为*
make zImage (led_drv.c包含在zImage里面)
```

2.模块化编译

```
拷贝内核程序到内核源码中
修改Kconfig
修改Makfile
make menuconfig //选择为M
make zImage
make modules //将led_drv.c->led_drv.ko
insmod
rmmod
```

3.模块化编译方法2：

```
思想就是无需把内核程序拷贝到内核源码中
无需修改Kconfig和Makefile
无需make menuconfig
无需make zImage
只需一个小小的Makeifle即可搞定：
死记一下参考代码：
cd /opt/drivers/day01/1.0
vim Makefile 添加如下内容：
obj-m += helloworld.o  #采用模块化编译,helloworld.c->helloworld.ko
#执行命令make all或者make,执行对应的命令make -C ...
all:
    make -C /opt/kernel SUBDIRS=$(PWD) modules
    #make -C /opt/kernel=cd /opt/kernel && make
    #SUBDIRS=/opt/drivers/day01/1.0,告诉内核源码,在内核源码之外还有一个目录作为子目录
    #modules：对1.0这个子目录下的内核程序采用模块化编译
clean：
    make -C /opt/kernel SUBDIRS=$(PWD) clean
    #将子目录1.0的程序进行make clean操作
保存退出
make //编译
ls
  helloworld.ko //编译成果
cp helloworld.ko /opt/rootfs/

开发板测试：
重启开发板,进入uboot,让内核加载采用tftp,让内核启动采用nfs
setenv bootcmd tftp 20008000 zImage \; bootm 20008000
```

```
setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/opt/rootfs ...
savenv
boot //启动
启动以后：
insmod helloworld.ko //安装内核程序,内核执行入口函数helloworld_init
lsmod //查看内核程序的安装信息
rmmod helloworld //卸载内核程序,内核执行出口函数helloworld_exit
```

4.2.linux内核程序编程之命令行传参

1.回忆应用程序的命令行传参

```
vim helloworld.c
#include <stdio.h>
int main(int argc, char *argv[])
{
      int a;
      int b;

      if (argc != 3) {
         printf("用法: %s num1 num2\n", argv[0]);
         return -1;
      }

      //"100"->100
      a = strtoul(argv[1], NULL, 0);
      b = strtoul(argv[2], NULL, 0);

      printf("a = %d, b = %d\n", a, b);
      return 0;
}
gcc -o helloworld helloworld.c
./helloworld 100 200
分析：
argc = 3
argv[0] = "./helloworld"
argv[1] = "100"
argv[2] = "200"
缺点：一旦程序启动,后序就没法再次传递新的参数

内核程序的命令行传参实现过程：
1.内核程序的命令行传参时,接收参数的内核程序变量必须是
  全局变量
2.变量的数据类型必须是基本的数据类型,结构体不行
3.如果要给内核程序的某个全局变量传递参数,需要内核程序
  显式的进行传参声明,传参声明的宏：
  module_param(name, type, perm)
  name：接收参数的内核全局变量名
  type: 变量的数据类型：
        bool invbool
        short ushort
        int uint
        long ulong
        charp(=char *)
        切记：内核不允许处理浮点数(float,double)
        例如：2.3*3.2
              23*32/100
              浮点数的运算放在用户空间的应用程序来进行
```

perm：变量的访问权限(rwx)
例如：0664
注意：不允许有可执行权限(x=1)

案例：编写内核程序,实现内核程序的命令行传参
实施步骤：
虚拟机执行：
1.mkdir /opt/drivers/day01/2.0 -p
cd /opt/drivers/day01/2.0
3.vim helloworld.c 添加如下内容
4.vim Makefile 添加如下内容
5.make
helloworld.ko
6.cp helloworld.ko /opt/rootfs/

linux系统调试宏：
__FILE__
__LINE__
__FUNCITION__/__func__
__DATE__
__TIME__

ARM板执行：
1.不传递参数
insmod helloworld.ko
lsmod
rmmod helloworld

2.加载安装内核程序时传递参数
insmod helloworld.ko irq=100 pstring=china
lsmod
rmmod helloworld

3.加载安装内核程序以后传递参数
insmod helloworld.ko irq=100 pstring=china
//读取文件irq的内容
cat /sys/module/helloworld/parameters/irq
ls /sys/module/helloworld/parameters/pstring //没有此文件
//向文件irq重新写入一个新内容
echo 20000 > /sys/module/helloworld/parameters/irq
rmmod helloworld

结论：
1.如果传参声明时,权限为非0,那么在/sys/...../parameters
会生成一个跟变量名同名的文件,文件内容就是变量的值
2.通过修改文件的内容就可以间接修改变量的值
3.如果权限为0,那么在/sys/.../parameters下就不会
生成同名的文件,这个变量的传参只能在程序加载时进行
4.注意：/sys/目录下所有的内容都是内核创建,存在于
内存中,将来如果没有内核程序加载以后传递参数的需求
权限必须一律给0,目的为了节省内存资源!

4.3.linux内核程序编程之内核符号导出
回忆：应用程序多文件之间的调用
参考代码：
mkdir /opt/drivers/day01/3.0

```
cd /opt/drivers/day01/3.0
vim test.h //声明
#ifndef __TEST_H
#define __TEST_H

extern void test(void);

#endif

vim test.c //定义
#include <stdio.h>
void test(void)
{
        printf("%s\n", __func__);
}

vim main.c //调用
#include <stdio.h>
#include "test.h"

int main(void)
{
        test();//调用
        return 0;
}

编译：
arm-linux-gcc -fpic -shared -o libtest.so test.c
arm-linux-gcc -o main main.c -L. -ltest
mkdir /opt/rootfs/home/applib
cp libtest.so /opt/rootfs/home/applib
cp main /opt/rootfs/home/applib

开发板测试：
export LD_LIBRARY_PATH=/home/applib:$LD_LIBRARY_PATH
/home/applib/main

内核程序多文件的调用实现过程：
1.内核程序多文件的调用实现过程和应用程序多文件的调用实现
  过程一模一样:该声明的声明,该定义的定义,该调用的调用
2.还需要显式的进行符号(函数名或者变量名)的导出
  导出符号的宏：
  EXPORT_SYMBOL(函数名或者变量名);
  或者
  EXPORT_SYMBOL_GPL(函数名或者变量名);
  前者导出的变量和函数,不管其他内核程序是否添加：MODULE_LICENSE("GPL")
  都能访问调用;
  后者导出的变量和函数,只能给那些添加了MODULE_LICENSE("GPL")的内核程序访问

案例：编写内核程序,掌握内核的符号导出知识点
实施步骤：
虚拟机执行：
mkdir /opt/drivers/day01/4.0
cd /opt/drivers/day01/4.0
vim test.h 添加如下内容
#ifndef __TEST_H
```

```
#define __TEST_H

//函数声明
extern void test(void);

#endif
```

保存退出

vim test.c 添加如下内容

```
#include <linux/init.h>
#include <linux/module.h>

//函数定义
void test(void)
{
        printk("%s\n");
}

//显式的进行导出
EXPORT_SYMBOL(test);

EXPORT_SYMBOL_GPL(test);

//添加遵循GPL协议的信息
MODULE_LICENSE("GPL");
```

保存退出

vim helloworld.c 添加如下内容

```
#include <linux/init>
#include <linux/module.h>

static int helloworld_init(void)
{
        test(); //调用
        printk("%s\n", __func__);
        return 0;
}

static void helloworld_exit(void)
{
        test(); //调用
        printk("%s\n", __func__);
}

module_init(helloworld_init);
module_exit(helloworld_exit);

MODULE_LICENSE("GPL");
```

保存退出

修改Makefile,添加对test.c的编译支持

```
obj-m += helloworld.o test.o
```

或者

```
obj-m += helloworld.o
```

```
obj-m += test.o

make //开始编译
test.ko helloworld.ko

mkdir /opt/rootfs/home/drivers/ //创建驱动目标文件的存放目录
cp *.ko /opt/rootfs/home/drivers/
```

开发板测试：
1.insmod /home/drivers/?
2.insmod /home/drivers/?
3.rmmod ?
4.rmmod ?

案例：利用EXPORT_SYMBOL_GPL进行符号导出,做对比测试

回顾：
1.linux内核设备驱动开发相关基础
  1.1.设备驱动两大核心思想
  1.2.linux内核设备驱动的分类
  1.3.linux系统包含的两个空间：用户空间和内核空间
  1.4.linux内核程序的编程框架
  1.5.linux内核程序的命令行传参
  1.6.linux内核程序的符号导出

2.linux内核程序的打印函数printk
  2.1.printk VS printf
  相同点：
        都是用于打印信息
        用法完全一致
  不同点：
        前者只能用于内核空间
        后者只能用于用户空间
        前者能够指定打印输出级别

  2.2.printk的打印输出级别
  级别共8级:0~7,数字越大,输出级别越小

```
  #define  KERN_EMERG   "<0>" //系统崩溃时需要打印
  #define  KERN_ALERT   "<1>" //立即需要处理
  #define  KERN_CRIT    "<2>" //严重问题
  #define  KERN_ERR     "<3>" //错误信息
  #define  KERN_WARNING "<4>" //警告
  #define  KERN_NOTICE  "<5>" //正常但还需要引起注意
  #define  KERN_INFO    "<6>" //信息
  #define  KERN_DEBUG   "<7>" //额外的调试信息
```

  用法：

```
  printk(KERN_ERR "this is a error msg!\n");
```

  或者

```
  printk("<3>" "this is a error msg!\n");
```

  2.3.问：实际产品在发布的时候,有些信息是没有必要输出的,
      有些信息可能需要进行打印输出,只需设置一个默认的
      打印输出级别(类似水位的警戒线)进行控制信息是否输出

如果printk指定的输出级别大于默认的打印输出级别,此信息
输出,否则不输出,如何设置默认的打印输出级别呢？
答：通过两种方法进行配置

方法1：通过修改printk打印输出级别的配置文件
/proc/sys/kernel/printk
案例：练习方法1
实施步骤：
虚拟机执行：
1.mkdir /opt/drivers/day02/1.0 -p
2.cd /opt/drivers/day02/1.0
3.vim printk_all.c
4.vim Makefile
5.make
cp printk_all.ko /opt/rootfs/home/drivers

ARM板执行：
0.cd /home/drivers
1.insmod printk_all.ko //查看打印信息
2.rmmod printk_all.ko //查看打印信息
3.查看当前内核printk的默认打印输出级别
cat /proc/sys/kernel/printk
7(串口终端设备对应的输出级别)　　4　　1　　7
4.修改配置文件来修改默认打印输出级别
echo 8 > /proc/sys/kernel/printk
cat /proc/sys/kernel/printk
5.insmod printk_all.ko
rmmod printk_all

6.结论：方法1不能解决内核启动时候的打印信息,例如：不能
完全将内核信息进行屏蔽

方法2：通过修改内核的启动参数,来设置默认的打印输出级别
案例：练习方法2
实施步骤：
1.重启开发板,进入uboot命令行模式,执行：
setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/opt/rootfs ip=192.168.1.110:192.168.1.8:192.168.1.1:255.255.255.0::eth0:on init=/linuxrc console=ttySAC0,115200 debug
boot
2.系统启动
cd /home/drivers/
insmod printk_all.ko
rmmod printk_all
cat /proc/sys/kernel/printk
结论：debug对应的级别为10

3.重启开发板,进入uboot命令行模式,执行：
setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/opt/rootfs ip=192.168.1.110:192.168.1.8:192.168.1.1:255.255.255.0::eth0:on init=/linuxrc console=ttySAC0,115200 quiet
boot
4.系统启动
cd /home/drivers/
insmod printk_all.ko
rmmod printk_all

cat /proc/sys/kernel/printk
结论：quiet对应的级别为4

5.重启开发板,进入uboot命令行模式,执行：
setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/opt/rootfs ip=192.168.1.110:192.168.1.8:192.168.1.1:255.255.255.0::eth0:on init=/linuxrc console=ttySAC0,115200 loglevel=0
boot
6.系统启动
cd /home/drivers/
insmod printk_all.ko
rmmod printk_all
cat /proc/sys/kernel/printk
结论：loglevel=数字

3.linux内核GPIO操作库函数
3.1.明确”GPIO操作“：配置GPIO为输出或者输入
如果是输出口,输出1或者0
如果是输入口,获取GPIO的状态1或者0

3.2.ARM裸板GPIO操作软件编程
例如：配置为输出口,输出1
```
*gpiocon &= ~(0xf << xxx);
*gpiocon |= (1 < xxx);
*gpiodata|= (1 << xxx);
```

3.3.linux内核便于驱动开发者进行GPIO操作,提供了相关的GPIO操作库函数
1.int gpio_request(int gpio, char *label)
功能：明确CPU的任何硬件信息,比如GPIO管脚硬件信息对于内核来说都是一种宝贵的资源(像内存),程序要想访问必须先向内核去申请硬件资源,类似malloc
参数：
gpio:表示硬件GPIO管脚对应的内核软件编号(类似身份证号)
具有唯一性
内核软件编号由内核已经定义好,例如：

硬件GPIO名	内核软件编号
GPC0_3	S5PV210_GPC0(3) //宏
GPC0_4	S5PV210_GPC0(4)
GPF1_3	S5PV210_GPF1(3)
...	...

label：随意指定一个名称即可

内核函数的返回值一律不允许记忆,只需要利用SI打开内核源码,看大神如何写,照猫画虎;
置于头文件的添加,将大神的代码的头文件一股脑全部拷贝即可

2.void gpio_free(int gpio)
功能：硬件GPIO资源不再使用了,记得要释放资源,类似free

3.int gpio_direction_output(int gpio, int value)
功能：配置GPIO为输出口,同时输出一个value值(1/0)

4.int gpio_direction_input(int gpio)
功能：配置GPIO为输入口

5.int gpio_set_value(int gpio, int value)
功能：仅仅设置GPIO的输出状态为value值(1/0)
前提是GPIO必须配置为输出口

6.int gpio_get_value(int gpio)
功能：获取GPIO的状态,返回值保存状态(1/0)
不管是输入还是输出口都可以使用

涉及头文件：
```
#include <asm/gpio.h>
#include <plat/gpio-cfg.h>
```

案例：加载驱动开灯;卸载驱动关灯
实施步骤：
虚拟机执行：
```
mkdir /opt/drivers/day02/2.0
cd /opt/drivers/day02/2.0
vim led_drv.c
vim Makefile
make
cp led_drv.ko /opt/rootfs/home/drivers/
```

开发板执行：
```
cd /home/drivers
insmod led_drv.ko //开灯
rmmod led_drv //关灯
```

结构体的标记初始化方式：
```
struct std {
	int a;
	int b;
	int c;
	int d;
	int e;
};

//一般定义初始化
struct std info = {100, 200, 300, 400, 500};

//标记初始化
struct std info = {
	.e = 100,
	.a = 200,
	.c = 300
}; //不用全部初始化,还可以不用按照顺序
```

4.linux系统的系统调用实现原理
面试题：谈谈linux系统调用
回忆：学过系统调用函数：open/read/write/close/mmap/lseek/fork/exit/sbrk等
系统调用函数作用：它是用户空间和内核空间数据交互的唯一的通道
应用程序利用系统调用函数能够向内核发起一个业务处理的请求
内核最终帮你完成业务,并且将处理的结果给应用程序
系统调用实现的基本过程：
这里以write系统调用函数为例：
1.首先应用程序调用write系统调用函数
2.会调用到C库的write函数的定义

3.C库的write函数将做两件事：

3.1.保存write函数对应的系统调用号到R7寄存器

系统调用号：linux系统调用函数都有唯一的一个软件编号(类似身份证号)

系统调用号定义在内核源码的arch/arm/include/asm/unistd.h

```
#define __NR_restart_syscal (0+  0)
#define __NR_exit                   (0+  1)
#define __NR_fork                   (0+  2)
#define __NR_read                   (0+  3)
#define __NR_write                  (0+  4)
...                                  ...
#define __NR_函数名                   数字
```

3.2.调用svc软中断指令,触发软中断异常

3.3.一旦触发软中断异常,CPU开启了软中断异常的处理

最终CPU跳转到软中断处理的入口地址,至此应用程序

由用户空间“陷入”内核空间,CPU的工作模式有USER切换到

SVC管理模式

3.4.进入软中断的处理入口地址以后,同样做两件事：

1.从R7寄存器中取出之前保存的系统调用号

2.然后以取出的系统调用号为下标在内核事先已经

定义好的系统调用表(表=大数组)中找到一个函数

此函数为sys_write,找到以后执行此函数

系统调用表：本质就是一个大数组,数组中每一个元素

保存的是一个函数地址

定义在内核源码的：arch/arm/kernel/calls.S

3.5.执行完毕,最终原路返回到用户空间

提示：必须会画图

****

4.linux内核字符设备驱动开发相关内容

面试题：如何编写一个字符设备驱动程序

4.1.linux的理念：一切皆文件

“一切”：就是指硬件资源

将来只要访问某个文件,本质上就是在访问硬件本身!

4.2.设备文件特性

字符设备对应的文件又称字符设备文件

块设备对应的文件又称块设备文件

网络设备没有对应的文件,通过socket套接字进行访问

结论：设备文件包括字符设备文件和块设备文件

设备文件存在于根文件系统rootfs的dev目录中,以TPAD的4个UART为例,4个UART对应的设备文件：

```
ls /dev/s3c2410_serial* -lh
crw-rw----  204,  64  /dev/s3c2410_serial0
crw-rw----  204,  65  /dev/s3c2410_serial1
crw-rw----  204,  66  /dev/s3c2410_serial2
crw-rw----  204,  67  /dev/s3c2410_serial3
```

说明：

"c":表示此设备文件对应的硬件是字符设备硬件

块设备用"b"

"204":表示设备文件的主设备号
"64~67":表示设备文件的次设备号
s3c2410_serial0:第一个UART的设备文件名
s3c2410_serial1:第二个UART的设备文件名
s3c2410_serial2:第三个UART的设备文件名
s3c2410_serial3:第四个UART的设备文件名

设备文件的访问：一定要利用系统调用函数进行：

```
//打开第一个串口
int fd = open("/dev/s3c2410_serial0", O_RDWR);

//从串口读取数据
read(fd, buf, size);

//向串口写入数据
write(fd, "hello,world", 12);

//关闭串口
close(fd);
```

设备文件的创建方法：2种
1.手动创建,利用命令mknod
  mknod /dev/设备文件名 c 主设备号 次设备号
  例如：
  mknod /dev/zhangsan c 250 0

2.自动创建

4.3.设备号
设备号包括主设备号和次设备号
设备号的数据类型：dev_t,本质unsigned int
设备号的高12位保存的主设备号
设备号的低20位保存的次设备号
设备号相关操作宏：
MKDEV:已知主，次设备号,合并一个设备号
      dev_t dev = MKDEV(主设备号, 次设备号);
MAJOR:已知设备号,提取主设备号
      int major = MAJOR(设备号);
MINOR：已知设备号，提取次设备号
      int minor = MINOR(设备号);
主设备号：应用程序根据设备文件的主设备号在茫茫的内核源码中
          找到自己匹配的设备驱动程序,一个设备驱动仅有一个
          主设备号
次设备号：如果多个硬件共享一个主设备号,也就共享一个设备驱动
          将来设备驱动根据次设备号来区分用户到底想操作哪个
          硬件个体
结论：设备号对于内核来说是一种宝贵的资源,驱动必须首先
      向内核去申请设备号资源

向内核申请和释放设备号的两个函数：

```
int alloc_chrdev_region(dev_t *dev,
                        unsigned baseminor,
                        unsigned count,
                        const char *name);
```

功能：向内核去申请设备号
dev:保存内核给你分配的设备号

baseminor:希望起始的次设备号,一般给0
count：分配次设备号的个数
name：设备名称而不是设备文件名,通过cat /proc/devices查看

```
void unregister_chrdev_region(dev_t dev, int count);
```
功能：释放申请的设备号
dev:申请好的设备号
count:次设备号的个数

5.自行设计linux内核字符设备驱动
声明一个描述字符设备驱动的数据结构
```
struct char_device {
	char *name; //字符设备的名称
	dev_t dev; //字符设备对应的设备号
	int count; //硬件设备的个数
	int (*open)(...) //打开设备接口
	int (*close)(...) //关闭设备接口
	int (*read)(...) //读设备接口
	int (*write)(...) //写设备接口
};
```
大胆设想：
应用程序调用open->软中断->内核的sys_open->驱动的open接口
应用程序调用close->软中断->内核的sys_close->驱动的close接口
应用程序调用read->软中断->内核的sys_read->驱动的read接口
应用程序调用write->软中断->内核的sys_write->驱动的write接口

优化数据结构：
```
//描述字符设备驱动的数据结构
struct char_device {
	char *name; //字符设备的名称
	dev_t dev; //字符设备对应的设备号
	int count; //硬件设备的个数
	struct file_operations *ops;
};

//描述字符设备驱动接口的数据结构
struct file_operations {
	int (*open)(...) //打开设备接口
	int (*close)(...) //关闭设备接口
	int (*read)(...) //读设备接口
	int (*write)(...) //写设备接口
	... //为所欲为添加任何一个接口
};
```

6.内核描述字符设备驱动的数据结构和硬件操作接口的数据结构
```
  //描述字符设备驱动的数据结构
struct cdev {
	dev_t dev; //字符设备对应的设备号
	int count; //硬件设备的个数
	struct file_operations *ops;//字符设备驱动具有的硬件操作接口
	... //内核使用
};

//描述字符设备驱动接口的数据结构
struct file_operations {
	int (*open)(struct inode *, struct file *) //打开设备接口
```

```
        int (*close)(struct inode *, struct file *) //关闭设备接口
        ...
};
```

回顾：
笔试题：自己编写一个strcmp或者stcpy函数
内核源码/lib/string.c提供相关的参考代码

1.linux内核设备驱动开发相关基础内容
  1.1.用户空间和内核空间
  1.2.linux内核程序编程基本框架
  1.3.linux内核程序编译
  1.4.linux内核程序操作
  1.5.linux内核程序命令行传参
  1.6.linux内核程序符号导出
  1.7.linux内核程序打印函数printk
  1.8.linux内核GPIO操作库函数
  1.9.linux内核系统调用实现过程

2.linux内核字符设备驱动开发相关内容
  2.1.linux系统理念
  2.2.linux设备驱动分类
  2.3.设备文件
      字符设备文件
      块设备文件
      /dev/
      包含主，次设备号
      mknod
      利用系统调用函数进行访问
  2.4.设备号
      包括主，次设备号
      dev_t
      12
      20
      MKDEV
      MAJOR
      MINOR
      主设备号：应用根据主设备号找驱动
      次设备号：驱动根据次设备号区分硬件个体
      设备号是一种宝贵的资源
      申请：alloc_chrdev_region
      释放：unregister_chrdev_region

  2.5.linux内核描述字符设备驱动的数据结构

```
      struct cdev {
        dev_t dev; //保存申请的设备号
        int count; //保存硬件设备的个数
        const struct file_operations *ops;//保存字符设备驱动具有硬件操作接口
        ...
      };

      linux内核字符设备驱动的硬件操作接口的数据结构:
      struct file_operations {
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
```

```
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ...
    };
```

2.6.问：如何实现一个字符设备驱动程序呢？
答：编程步骤是定死的！
具体编程步骤：
1.定义初始化一个struct file_operations硬件操作接口对象
2.定义初始化一个struct cdev字符设备对象
cdev_init用来初始化字符设备对象
3.调用cdev_add将字符设备对象注册到内核中
至此内核就有一个新的字符设备驱动
4.调用cdev_del将字符设备对象从内核中卸载
一旦卸载,内核就不存在这个字符设备驱动

2.7.切记应用程序的系统调用函数和底层驱动硬件操作接口之间的联系：
应用程序调用open->C库open->软中断->内核sys_open->驱动open接口
应用程序调用read->C库read->软中断->内核sys_read->驱动read接口
应用程序调用write->C库write->软中断->内核sys_write->驱动write接口
应用程序调用close->C库close->软中断->内核sys_close->驱动release

案例：应用程序打开LED设备时,开灯
应用程序关闭LED设备时,关灯
分析：
0.将两个LED灯作为一个硬件设备
1.应用程序使用open/close
驱动提供open/release接口
2.两个接口只需要利用GPIO操作库函数分别操作LED即可
3.前提是需要将file_operations和cdev进行定义初始化和注册

实施步骤：
虚拟机执行：

```
mkdir /opt/drivers/day03/1.0 -p
cd /opt/drivers/day03/1.0
vim led_drv.c
vim Makfile
make
cp led_drv.ko /opt/rootfs/home/drivers
vim led_test.c //测试的应用程序
arm-linux-gcc -o led_test led_test.c
cp led_test /opt/rootfs/home/drivers
```

ARM板执行：

```
insmod /home/drivers/led_drv.ko
cat /proc/devices //查看申请的主设备号
Character devices: //当前系统的字符设备
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  ...
  250 tarena
  ...
```

第一列：申请的主设备号
第二列：设备名称

```
mknod /dev/myled c 250 0
/home/drivers/led_test
```

3.编写一个字符设备驱动的步骤：
1.写头文件
2.写入口和出口函数
3.先声明和定义初始化硬件信息
4.然后定义初始化软件信息
  file_operations
  cdev
  dev
5.填充入口和出口函数
  先写注释流程
  再写代码
6.最后将给用户提供的接口函数完善

4.linux内核字符设备驱动硬件操作接口之write
  回忆：应用write系统调用函数的使用
  char msg[1024] = {....};
  write(fd, msg, 1024); //向设备写入数据

  对应的底层驱动的write接口：
```
struct file_operations {
      ssize_t (*write) (struct file *file,
                        const char __user *buf,
                        size_t count,
                        loff_t *ppos);
};
```
  调用关系：
  应用调write->C库write->软中断->内核sys_write->驱动write接口
  功能：此接口用于将数据写入硬件设备
  参数：
  file:文件指针
  buf：此指针变量用__user修饰,表明此指针变量保存的地址位于
       用户空间(0~3G),也就是保存用户缓冲区的首地址(例如msg)
       虽然底层驱动可以访问buf获取用户的数据,但是内核不允许驱动
       直接操作这个用户缓冲区(例如：int data = *(int *)buf),
       这么操作时相当危险的,如果要从buf缓冲区读取数据到驱动中
       必须要利用内核提供的内存拷贝函数copy_from_user将用户缓冲区的
       数据拷贝到内核缓冲区(3G~4G)中
  count:要写入的字节数
  ppos：记录了上一次的写位置,参考代码：
        loff_t pos = *ppos //获取上一次的写位置
        这一次写了512字节,此接口返回之前,要记得更新写位置：
        *ppos = pos + 512

        友情提示：如果写操作只进行一次,无需记录写位置！
        所以此参数用于多次写入操作

  内存拷贝函数copy_from_user
```
unsigned long copy_from_user(void *to,
                             const void __user *from,
                             unsigned long n)
```
  函数功能：将用户空间缓冲区的数据拷贝到内核空间的缓冲区中
  参数：
  to:目标,内核缓冲区的首地址

from:源,用户缓冲区的首地址
n:要拷贝的字节数
切记：此函数仅仅做了将用户内存的数据拷贝到内核的内存
此时还没有进行硬件操作,硬件操作还需要进行额外的操作

案例：应用向设备写1,开灯
应用写设备写0,关灯
实施步骤：
mkdir /opt/drivers/day03/2.0
cd /opt/drivers/day03/2.0
vim led_drv.c
vim Makfile
make
cp led_drv.ko /opt/rootfs/home/drivers
vim led_test.c //测试的应用程序
arm-linux-gcc -o led_test led_test.c
cp led_test /opt/rootfs/home/drivers

ARM板执行：
insmod /home/drivers/led_drv.ko
cat /proc/devices //查看申请的主设备号
Character devices: //当前系统的字符设备
1 mem
2 pty
3 ttyp
4 /dev/vc/0
...
250 tarena
...
第一列：申请的主设备号
第二列：设备名称

mknod /dev/myled c 250 0
/home/drivers/led_test on
/home/dirvers/led_test off

总结：底层驱动write编写三步曲：
1.定义内核缓冲区
2.从用户缓冲区拷贝数据到内核缓冲区
3.根据用户的需求进行硬件数据写入操作

5. linux内核字符设备驱动硬件操作接口之read
回忆：应用read系统调用函数的使用
```
char msg[1024];
read(fd, msg, 1024); //从硬件设备读取数据到msg缓冲区
```

对应的底层驱动的read接口：
```
struct file_operations {
      ssize_t (*read) (struct file *file,
                       char __user *buf,
                       size_t count,
                       loff_t *ppos);
};
```
调用关系：
应用调read->C库read->软中断->内核sys_read->驱动read接口
功能：从硬件设备读取数据到用户缓冲区

参数：
file:文件指针
buf：此指针变量用__user修饰,表明此指针变量保存的地址位于
用户空间(0~3G),也就是保存用户缓冲区的首地址(例如msg)
虽然底层驱动可以访问buf向用户缓冲区写入数据,但是内核不允许驱动
直接操作这个用户缓冲区(例如：*(int *)buf=0x5555),
这么操作时相当危险的,如果驱动要想用户缓冲区写入数据
必须要利用内核提供的内存拷贝函数copy_to_user将内核缓冲区的
数据拷贝到用户缓冲区中
count:要读取的字节数
ppos：记录了上一次的读位置,参考代码：
loff_t pos = *ppos //获取上一次的读位置
这一次读了512字节,此接口返回之前,要记得更新读位置：
*ppos = pos + 512

友情提示：如果读操作只进行一次,无需记录读位置！
所以此参数用于多次读操作

内存拷贝函数copy_to_user
unsigned long copy_to_user(void __user *to,
const void *from,
unsigned long n)
函数功能：将内核空间缓冲区的数据拷贝到用户空间的缓冲区中
参数：
to:目标,用户缓冲区的首地址
from:源,内核缓冲区的首地址
n:要拷贝的字节数
切记：此函数仅仅做了将内核内存的数据拷贝到用户的内存
此时还没有进行硬件操作,硬件操作还需要进行额外的操作

总结：数据流走向
对于write：
用户数据->经过一次拷贝到内核空间->经过二次拷贝到硬件
对于read：
硬件经过一次数据拷贝->内核,经过二次拷贝->用户

案例：获取灯的开关状态
总结：底层驱动read编写三步曲：
1.定义内核缓冲区
2.从硬件获取数据拷贝到内核缓冲区
3.将内核缓冲区数据最终拷贝到用户缓冲区

案例：应用能够任意开关某个灯
提示：应用应该向设备写入两个信息：开或者关和灯的编号(1或者2)
应用只需向驱动写入一个结构体即可：

```
struct led_cmd {
        int cmd; //开：1;关：0
        int index;//第一个灯：1；第二个灯：2
};
write(fd, 结构体,结构体大小);
```

底层驱动write接口用结构体来接收数据，
接收完解析判断结构体成员

回顾：

1.linux内核字符设备驱动相关内容
  1.1.理念
  1.2.字符设备文件
      c
      主设备号
      次设备号
      设备文件名
      mknod
      仅仅在open时使用
  1.3.设备号
      dev_t
      12
      20
      MKDEV
      MAJOR
      MINOR
      宝贵资源
      alloc_chrdev_region
      unregister_chrdev_region
      主设备号：应用找到驱动
      次设备号：驱动分区硬件个体
  1.4.linux内核描述字符设备驱动数据结构
      struct cdev
        .dev
        .count
        .ops
      配套函数：
      cdev_init
      cdev_add
      cdev_del

  1.5.字符设备驱动描述硬件操作接口的数据结构
      struct file_operations {
        .open
        .release
        .read
        .write
      };
      open/release:可以根据用户的实际需求不用初始化
                   应用程序open/close永远返回成功
      read:
        1.分配内核缓冲区
        2.获取硬件信息将信息拷贝到内核缓冲区
        3.拷贝内核缓冲区数据到用户缓冲区
      write:
        1.分配内核缓冲区
        2.拷贝用户缓冲区数据到内核缓冲区
        3.拷贝内核缓冲区到硬件

      注意：第二个形参buf:保存用户缓冲区的首地址,驱动不能直接
                         访问,需要利用内存拷贝函数
      copy_from_user/copy_to_user

2.linux内核字符设备驱动硬件操作接口之ioctl
  2.1.掌握ioctl系统调用函数：
  函数原型：

```
int ioctl(int fd, int request, ...);
```

函数功能：
1.不仅仅能够向设备发送控制命令(例如开关灯命令)
2.还能够跟硬件设备进行数据的读写操作

参数：
fd：设备文件描述符
request:向设备发送的控制命令,命令需要自己定义

```
        例如：
        #define LED_ON   (0x100001)
        #define LED_OFF  (0x100002)
```

...:如果应用程序要传递第三个参数,第三个参数要传递用户缓冲区
    的首地址,将来底层驱动可以访问这个用户缓冲区的首地址
    同样底层驱动不能直接访问,需要利用内存拷贝函数
返回值：成功返回0，失败返回-1
参考代码：

```
//传递两个参数：
//开灯
ioctl(fd, LED_ON);
//关灯
ioctl(fd, LED_OFF);
```

说明：仅仅发送命令

```
//传递三个参数：
//开第一个灯：
int uindex = 1;
ioctl(fd, LED_ON, &uindex);

//关第一个灯：
  int uindex = 1;
ioctl(fd, LED_OFF, &uindex);
```

说明：不仅仅发送命令,还传递用户缓冲区的首地址,完成和设备的读或者写

2.2.ioctl对应的底层驱动的接口
回忆C编程：

```
int a = 0x12345678;
int *p = &a;
printf("a = %#x\n", *p);
```

等价于：

```
unsigned long p = &a;
printf("a = %#x\n", *(int *)p);
```

ioctl对应的底层驱动的接口

```
struct file_operations {
      long (*unlocked_ioctl)(struct file *file,
                             unsigned int cmd,
                             unsigned long arg)
};
```

调用关系：
应用ioctl->C库ioctl->软中断->内核sys_ioctl->驱动unlocked_ioctl接口
接口功能：
1.不仅仅向设备发送控制命令
2.还能够和设备进行数据的读或者写操作
参数：
file：文件指针
cmd:保存用户传递过来的参数,保存应用ioctl的第二个参数

arg：如果应用ioctl传递第三个参数,arg保存用户缓冲区的
首地址,内核不允许直接直接访问(int kindex=*(int *)arg)
需要利用内存拷贝函数,使用时注意数据类型的转换

案例：利用ioctl实现开关任意一个灯
实施步骤：
虚拟机执行：
```
mkdir /opt/drivers/day04/1.0 -p
cd /opt/drivers/day04/1.0
vim led_drv.c
vim Makefile
vim led_test.c
make
arm-linux-gcc -o led_test led_test.c
cp led_drv.ko led_test /opt/rootfs/home/drivers
```

ARM执行：
```
insmod /home/drivers/led_drv.ko
cat /proc/devices
mknod /dev/myled c 主设备号 0
/home/drivers/led_test on 1
/home/drivers/led_test on 2
/home/drivers/led_test off 1
/home/drivers/led_test off 2
```

3.linux内核字符设备驱动之设备文件的自动创建
3.1.设备文件手动创建
mknod /dev/设备文件名 c 主设备号 次设备号
3.2.设备文件自动创建实施步骤：
1.保证根文件系统rootfs具有mdev可执行程序
mdev可执行程序将来会帮你自动创建设备文件
```
which is mdev
/sbin/mdev
```
2.保证根文件系统rootfs的启动脚本etc/init.d/rcS必须有
以下两句话：
```
/bin/mount -a
echo /sbin/mdev > /proc/sys/kernel/hotplug
```
说明：
/bin/mount -a:将来系统会自动解析etc/fstab文件，进行
一系列的挂接动作
echo /sbin/mdev > /proc/sys/kernel/hotplug：将来驱动
创建设备文件时,会解析hotplug文件,驱动最终启动mdev来帮
驱动创建设备文件
3.保证根文件系统rootfs必须有etc/fstab文件,文件内容如下：
```
proc            /proc        proc   defaults 0 0
sysfs           /sys         sysfs  defaults 0 0
```
将/proc,/sys目录分别作为procfs,sysfs两种虚拟文件系统
的入口,这两个目录下的内容都是内核自己创建,创建的内存
存在于内存中
4.驱动程序只需调用以下四个函数即可完成设备文件的自动
创建和自动删除
```
struct class *cls; //定义一个设备类指针
//定义一个设备类,设备类名为tarena(类似长树枝)
cls = class_create(THIS_MODULE, "tarena");
//创建设备文件(类似长苹果)
device_create(cls, NULL, 设备号, NULL, 设备文件名);
```

例如：
```
//自动在/dev/创建一个名为myled的设备文件
device_create(cls, NULL, dev, NULL, "myled");

//删除设备文件(摘苹果)
device_destroy(cls, dev);

//删除设备类(砍树枝)
class_destroy(cls);
```

案例：在ioctl实现的设备驱动中添加设备文件自动创建功能

4.linux内核字符设备驱动之通过次设备号区分硬件个体

4.1.明确：多个硬件设备个体可以作为一个硬件看待,驱动也能够
通过软件进行区分

4.2.实现思路：

1.驱动管理的硬件特性相似
四个UART,Nand的多个分区
不能把LED,按键放在一起

2.主设备号为一个,驱动为一个,驱动共享
cdev共享
file_operations共享
.open
.read
.write
.release
.unlocked_ioctl
都共享

3.多个硬件个体都有对应的设备文件

4.驱动通过次设备号区分,次设备号的个数和硬件个体的数据一致

4.3.了解两个数据结构：struct inode, struct file

```
struct inode {
        dev_t    i_rdev; //保存设备文件的设备号信息
        struct cdev *i_cdev; //指向驱动定义初始化的字符设备对象led_cdev
        ...
};
```

作用：描述一个文件的物理上的信息(文件UID,GID,时间信息,大小等)
生命周期：文件一旦被创建(mknod),内核就会创建一个文件对应的inode对象
文件一旦被销毁(rm),内核就会删除文件对应的inode对象
注意：struct file_operations中的open,release接口的第一个
形参struct inode *inode,此指针就是指向内核创建的inode对象
所以,驱动可以通过inode指针获取到设备号信息：inode->i_rdev

```
struct file {
        const struct file_operations *f_op;//指向驱动定义初始化的硬件操作接口对象led_fops
        ...
};
```

作用：描述一个文件被打开以后的信息
生命周期：一旦文件被成功打开(open),内核就会创建一个file对象来描述
一个文件被打开以后的状态属性
一旦文件被关闭(close),内核就会销毁对应的file对象

总结：一个文件只有一个inode对象,但是可以有多个file对象
切记：通过file对象指针获取inode对象指针的方法：

内核源码：fbmem.c
struct inode *inode = file->f_path.dentry->d_inode;
提取次设备号：
int minor = MINOR(inode->i_rdev);

案例：编写设备驱动,实现通过次设备号来区分两个LED
mknod /dev/myled1 c 250 0
mknod /dev/myled2 c 250 1

****
二.linux内核混杂设备驱动开发相关内容
1.概念
混杂设备本质还是字符设备,只是混杂设备的主设备号由内核已经定义为,为10,将来各个混杂设备个体通过次设备号来进行区分
2.混杂设备驱动的数据结构

```
struct miscdevice {
        int minor;
        const char *name;
        const struct file_operations *fops;
        ...
};
```

minor:混杂设备对应的次设备号,一般初始化时指定
MISC_DYNAMIC_MINOR,表明让内核帮你分配一个次设备号
name:设备文件名,并且设备文件由内核帮你自动创建
fops：混杂设备具有的硬件操作接口

配套函数：
misc_register(&混杂设备对象);//注册混杂设备到内核
misc_deregister(&混杂设备对象);//卸载混杂设备

案例：利用混杂设备实现LED驱动,给用户提供的接口为ioctl
同上
vim分屏显示：
进入vim的命令行模式输入：
vs 文件名 //左右分屏
sp 文件名 //上下分屏
屏幕切换：ctrl+ww

回顾：
1.linux内核字符设备驱动开发相关内容
1.1.设备文件
/dev/
mknod
三个保证+四个函数
系统调用函数
1.2.设备号
dev_t
MKDEV
MAJOR
MINOR
主设备号
次设备号
资源
申请
释放

1.3.字符设备驱动数据类型
    struct cdev
    配套函数
    cdev_init
    cdev_add
    cdev_del
1.4.硬件操作接口数据类型
    struct file_operations
    open
    release
    read
    write
      注意：buf
    unlocke_ioctl
      注意：arg
1.5.内存拷贝函数
    copy_from_user
    copy_to_user
1.6.设备文件自动创建
    三个保证
    四个函数
1.7.两个额外数据结构
    struct inode
      .i_rdev
    struct file
    inode和file关系：fbmem.c

2.linux内核混杂设备驱动开发相关内容
本质还是字符设备
主设备号为10
通过次设备号区分
数据结构：
struct miscdevice
      .minor = MISC_DYNAMIC_MINOR->让内核分配次设备号
      .name = 内核帮你创建设备文件
      .fops = 硬件操作接口
配套函数：
misc_register
misc_deregister

3.linux内核中断编程
面试题：谈谈对中断的理解
3.1.计算机为什么有中断机制
由于计算机硬件层面来讲由CPU和外设组成,CPU需要跟外设进行不断的数据通信,又由于外设的处理速度远远慢于CPU的处理速度,CPU为了保证访问外设时的数据正常,一般会想到采用轮询方式(死等,CPU不能干别的其他事情),最终降低CPU的利用率,让功耗提高;对于这种情况可以考虑使用中断机制,这里以CPU读取UART数据为例：
CPU读取UART数据时(接收缓冲区寄存器),当发现数据没有准备就绪,CPU可以去干其他别的事情(处理某个进程),一旦UART接收缓冲区有数据,UART控制器会给CPU发送一个中断信号(嗨,我这有数了),CPU一旦接收到这个中断信号,CPU停止手头的工作,转去读取UART数据读取完毕,CPU继续接着执行原先被打断的工作,提高了CPU的利用率

3.2.中断的硬件触发过程和硬件连接
明确：外设产生的中断信号不会直接送达CPU,而是要经过

中断控制器的处理以后再由中断控制器决定是否送达
给CPU一个中断信号(IRQ/FIQ)
画出一个简要的中断硬件连接图
外设和中断控制的中断线分两类：
外设中断：中断线可以有原理图来调整(肉眼能看到)
内部中断：中断线不可修改(集成处理器内部,肉眼看不到)

中断控制器的功能：
1.能够使能或者禁止某个外设中断信号
2.能够配置外设中断信号将来以IRQ还是FIQ发给CPU
3.能够设置外设中断的优先级
4.能够设置外设中断信号的有效触发方式
  高电平触发
  低电平触发
  上升沿触发
  下降沿触发
  双边沿触发

以按键为例,谈谈中断电信号的触发流程：
按键没有操作,中断线为高电平,按键按下,中断线上有一个下降沿
电信号,此电信号自动跑到中断控制器,中断控制经过一番的判断
如果合适,中断控制器最终给CPU发送一个中断信号,这个过程都是
硬件自动完成,一旦CPU接收到中断电信号,CPU开启中断异常的处理
流程：
ARM核硬件将完成：
1.备份CPSR到SPSR_IRQ/FIQ
2.设置CPSR
  MODE
  T
  I
  F
3.保存返回地址lr_irq/fiq=pc-4
4.设置PC为中断异常处理入口
  pc=0x18
  或者
  pc=0x1c
5.进入了异常向量表的中断处理入口,开启软件的中断异常处理
6.中断的软件处理过程：
  实现编写好异常向量表的代码
  保护现场
  处理中断处理函数
  恢复现场

3.3.画图展示中断的处理流程
具体参见int.bmp

3.4.中断编程步骤
明确：不管是ARM裸板程序还是带操作系统的程序,中断编程必须一致
中断编程四步骤：
1.编写异常向量表的代码
2.编写保存现场的代码
3.编写中断处理函数
  具体内容如何实现,严格按照用户的需求来定
4.编写恢复现场的代码

明确：不管是ARM裸板还是在linux系统下,1,2,4三步骤都是由

ARM公司或者linux内核已经帮你实现！第3步必须由程序员
根据用户的需求来完成！

问：linux内核驱动如何添加一个外设的中断处理函数呢？
一旦中断处理函数添加完成,将来外设产生中断,内核
就会最终调用此中断处理函数,完成用户的业务需求！
答：利用大名鼎鼎的request_irq函数即可完成向内核添加注册
一个外设的中断处理函数！

3.5.大名鼎鼎的request_irq函数详解
函数原型：

```
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev_id)
```

功能：
1.向内核申请硬件中断资源
2.向内核注册硬件中断对应的中断处理函数

参数：
irq:硬件中断对应的内核软件编号(类似身份证号),又称中断号
以宏的形式表示
例如：

硬件中断	中断号
XEINT0	IRQ_EINT(0)
XEINT1	IRQ_EINT(1)
.....	....
XEINT10	IRQ_EINT(10)

handler:中断处理函数,只需将中断处理函数地址传递过来即可
一旦注册中断处理函数到内核,将来硬件触发中断
内核就会调用此函数
中断处理函数的原型：
irqreturn_t 中断处理函数名(int irq, void *dev_id)
irq:当前触发的硬件中断对应的中断号
dev_id:给中断处理函数传递的参数信息
建议参数使用时,数据类型进行强转

flags:中断标志
对于外部中断,中断标志：
IRQ_TRIGGER_FALLING:
IRQ_TRIGGER_RISING:
IRQ_TRIGGER_HIGH:
IRQ_TRIGGER_LOW:
设置有效的触发方式
可以做位或操作

对于内部中断,给0即可,通过配置控制器内部寄存器
实现有效的中断触发方式配置
name：中断名称
通过cat /proc/interrupts查看此名称

dev_id:给中断处理函数传递的参数
想想pthread_create

```
void *thread_func(void *arg)
{
        int *p = (int *)arg;

        printf("g_data = %#x\n", *p);
}

int g_data = 0x55;
pthread_create(&id, NULL, thread_func, &g_data);
```

中断不再使用时,记得要删除中断处理函数和释放资源
free_irq(int irq, void *dev_id)
irq:中断号
dev_id:给中断处理函数传递的参数,切记注册中断处理函数
        传递的参数务必要和释放传递的参数要一致!

案例：采用中断方式,编写按键驱动,实现按键按下或者松开打印
      按键的信息
实施步骤：
先卸载官方的按键驱动：

```
cd /opt/kernel
make menuconfig
  Device Drivers->
        Input devices supports->
                Keyboards->
                   <*> S3c gpio keypads support... //去掉
```

保存退出

```
make zImage
cp arch/arm/boot/zImage /tftpboot
```

用新zImage重启开发板

```
mkdir /opt/drivers/day05/1.0 -p
cd /opt/drivers/day05/1.0
vim btn_drv.c
vim Makefile
make
cp btn_drv.ko /opt/rootfs/home/drivers/
```

ARM测试：

```
insmod /home/drivers/btn_drv.ko
cat /proc/interrupts //查看中断注册的信息
           CPU0
16:         63    s3c-uart  s5pv210-uart
18:         98    s3c-uart  s5pv210-uart
32:          0  s5p_vic_eint  KEY_UP
33:          0  s5p_vic_eint  KEY_DOWN
...         ...    ...          ...
```

第一列：中断号,例如IRQ_EINT(0)=32
第二列：中断触发次数
第三列：中断类型
第四列：中断名称

按下或者松开按键查看打印信息

```
cat /proc/interrupts //查看中断的触发次数
```

4.linux内核中断编程之顶半部和底半部机制

4.1.明确相关概念

linux系统,CPU软件层面一天到晚做两类事：进程和中断,哪个要想运行,必须先获取到CPU资源

“任务”：包括进程和中断

“休眠”：仅存在进程的世界里,进程休眠只是当前进程会释放所占用的CPU资源给其他任务使用,中断是不允许休眠操作

“优先级”：衡量一个任务获取CPU的一种能力,优先级越高,获取CPU资源的能力就越强

中断分两类：硬件中断和软中断

中断不隶属于任何进程,不参与进程之间的调度

前提是在linux系统,任务优先级的划分：

硬件中断优先级大于软中断

软中断的优先级大于进程

软中断有优先级之分

进程有优先级之分

硬件中断无优先级之分

4.2.切记：linux内核要求中断处理函数的执行速度越快越好,其他任务就能够及时获取到CPU资源

注意：中断处理函数更不能做休眠操作

如果中断处理函数长时间占用CPU资源,会影响系统的并发能力和响应能力!

问：有些场合,中断处理函数势必会长时间占用CPU资源也会势必影响系统的并发和响应能力,怎么办?

答：通过中断编程之顶半部和底半部机制进行优化

将原先的中断处理函数一分为二,分别是顶半部和底半部

4.3.顶半部特点

本质上就是中断处理函数,也就是一旦硬件产生中断,内核首先执行顶半部的内容(内核首先调用中断处理函数);

此时的中断处理函数和原中断处理函数不一样,此时的中断处理函数会做原先中断处理函数中比较紧急,耗时较短的内容,一旦执行完毕快速释放CPU资源给其他任务使用!

顶半部执行期间不允许被打断,不允许发生CPU资源的切换!

4.4.底半部特点

底半部要执行原先中断处理函数中不紧急,耗时较长的内容;

CPU会在"适当的时候"会去执行底半部的内容;

由于它不紧急,所以如果来了高优先级的任务同样可以打断底半部的执行过程,允许CPU资源发生切换

4.5.底半部实现方法：三种

tasklet

工作队列

软中断

4.6.底半部机制之tasklet特点

1.本质就是延后执行的一种手段

2.tasklet对应的延后处理函数,此函数中原先中断处理函数中不紧急,耗时较长的内容

3.tasklet是基于软中断实现,优先级高于进程,低于硬件中断所以tasklet延后处理函数不能进行休眠操作

4.tasklet的数据结构

```
struct tasklet_struct {
    void (*function)(unsigned long data);
    unsigned long data;
    ...
};
```

成员：
function:tasklet的延后处理函数,做不紧急,耗时较长的内容
不能进行休眠操作
形参data保存是给延后处理函数传递的参数,一般传递
参数的指针,注意数据类型的转换
data：就是给延后处理函数传递的参数

配套函数：
DECLARE_TASKLET(name,func,data);
功能：定义初始化tasklet对象
name:tasklet对象名
func:tasklet延后处理函数地址
data:给延后处理函数传递的参数

tasklet_schedule(&tasklet对象);
功能：向内核登记tasklet延后处理函数,一旦登记完成,内核
会在"适当的时候"去执行对应的延后处理函数
如果驱动中有顶半部(中断处理函数),一般在顶半部的代码中
调用登记即可;
如果驱动中没有顶半部,依据实际的硬件操作需求进行调用登记
即可!
一旦登记完成,内核就会在适当的时候去执行!

最后编写好延后处理函数即可

切记：tasklet基于软中断实现,所以延后处理函数执行速度尽量快
更不能做休眠操作

案例：利用tasklet"优化"按键驱动

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

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)
```

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

回顾：
1.linux内核并发和竞态之原子操作
  特点：
  能够解决所有的竞态问题
  位原子操作：对共享资源进行位操作,考虑竞态问题
  整形原子操作：对共享资源进行整型操作,考虑竞态问题
  数据类型：atomic_t(类比成int)
  总结：操作务必要使用内核提供的相关函数

2.linux内核等待队列机制
  产生根本原因：外设的处理速度远远慢速CPU
  能够让进程随时随地休眠,随时随地被唤醒
  编程步骤：
  1.定义初始化等待队列头(全局变量)
  2.定义初始化装载休眠进程的容器(局部变量)
    current：内核全局变量,指向"当前进程"
  3.添加休眠进程到等待队列中
  4.设置当前进程的休眠状态
  5.进入真正的休眠状态,释放CPU资源,等待被唤醒
    注意：唤醒的方法
  6.一旦进程被唤醒,设置进程的状态为运行,并且将进程从
    等待队列中移除
  7.一般要判断进程唤醒的原因
  8.驱动主动唤醒的方法
    wake_up/wake_up_interruptible
  9.务必掌握按键驱动包括去抖动

3.linux内核等待队列编程方法2：
  编程步骤：
  1.定义初始化等待队列头(构造鸡妈妈)
    wait_queue_head_t wq;
    init_waitqueue_head(&wq);
  2.调用以下两个方法即可实现进程的休眠
    wait_event(wq, condition);//切记：此乃宏
    说明：
    wq：等待队列头
    condition:如果condition为真,进程不会休眠,立即返回,即硬件设备可用(可读或者可写)
              如果condition为假,进程将进入不可中断的休眠状态,等待被唤醒,即硬件设备不可用

    或者
    wait_event_interruptible(wq, condition);
    说明：
    wq：等待队列头
    condition:如果condition为真,进程不会休眠,立即返回,即硬件设备可用(可读或者可写)
              如果condition为假,进程将进入可中断的休眠状态,等待被唤醒,即硬件设备不可用
    总结：
    以上两个宏等价于编程方法1的第2步~第7步

3.唤醒的方法
  wake_up/wake_up_interruptible

方法2的编程框架：
```
//休眠的地方
xxx(...) {
      //起初condition为假
      wait_event_interruptible(wq, condition);
      //一旦被唤醒
      condition设置为假
}

//唤醒的地方
yyy(...) {
      condition设置为真
      //唤醒休眠的进程
      wake_up_interruptible(&wq);
}
```

案例：利用等待队列编程方法2实现按键驱动

****

3.linux内核内存分配相关内容

3.1.linux内核内存的划分(了解)

明确：不管是在用户空间还是在内核空间,软件一律不允许访问硬件外设的物理地址,要想软件访问硬件外设的物理地址必须将硬件外设的物理地址映射到用户虚拟地址或者内核虚拟地址,将来软件只要访问用户虚拟地址或者内核虚拟地址就是在访问对应的物理地址

特例：uclinux操作系统软件访问的地址都是物理地址

回顾用户空间3G虚拟内存的划分

总结：用户空间3G虚拟内存地址和物理地址之间的映射属于动态映射(用到时进行映射,不用时将映射关系解除)

4G虚拟地址划分为用户虚拟地址和内核虚拟地址

用户虚拟地址空间范围：0~0xbfffffff

内核虚拟地址空间范围：0xc0000000~0xffffffff

linux内核1G虚拟内存地址和物理内存地址的映射属于一一映射,即内核在启动的时候就已经将物理内存地址和内核1G虚拟内存地址建立好映射关系,将来软件无需再次建立映射,直接访问即可,内存的访问效率最高：

物理内存地址	内核虚拟地址
0x0	0xC0000000
0x1	0xC0000001
0x2	0xC0000002
...	...
1G	1G

一一映射存在线性关系！

问题：如果采用一一映射,linux内核最多只能访问1G的物理内存如何让内核访问到所有的物理内存地址呢？又要兼顾内存的访问效率

答：linux内核将1G虚拟内存划分若干个区域

X86划分：

直接内存映射区：

大小为896M
内核在启动的时候,将直接内存映射区的内核虚拟内存地址
和物理内存地址进行一一映射,这块内存区域的效率最高
又称低端内存

动态内存映射区：
默认大小为120M
内核代码需要访问某块物理内存时,内核动态建立动态内存映射区
的虚拟内存和物理内存的映射,使用完毕,一定要记得解除
地址映射,否则造成内存泄漏

永久内存映射区：
固定内存映射区：
固定=永久
大小都为4M
如果频繁的访问某块物理内存,考虑到效率,可以将
物理内存映射到永久或者固定内存的虚拟内存上
前者映射时会导致休眠,不能用于中断上下文
后者不会导致休眠

高端内存=动态内存映射区+永久+固定

TPAD开发板,linux内核1G虚拟内存的划分：
启动开发板,观察内核打印信息,找到1G内核虚拟内存的划分：

```
Virtual kernel memory layout:
 区域名     内核起始地址   内核结束地址   区域大小

 异常向量表
 vector  : 0xffff0000 - 0xffff1000   (   4 kB)

 固定内存映射区
 fixmap  : 0xfff00000 - 0xfffe0000   ( 896 kB)

 DMA内存映射区
 DMA     : 0xff000000 - 0xffe00000   (  14 MB)

 动态内存映射区
 vmalloc : 0xec800000 - 0xfc000000   ( 248 MB)

 直接内存映射区
 lowmem  : 0xc0000000 - 0xec600000   ( 710 MB)

 //模块加载区域
 modules : 0xbf000000 - 0xc0000000   (  16 MB)

 //初始化段
   .init : 0xc0008000 - 0xc0037000   ( 188 kB)
 //代码段
   .text : 0xc0037000 - 0xc0832000   (8172 kB)
 //数据段
   .data : 0xc0832000 - 0xc0886960   ( 339 kB)
```

切记：一个物理地址可以有多个虚拟地址(用户的和内核的)
但是一个虚拟地址不能对应多个物理地址

3.2.linux内核内存分配的函数
回忆应用内存分配：

int a; //局部,全局的(初始化和没初始化)
malloc/free //堆

内核分配函数一：
kmalloc/kfree
函数原型：
void *kmalloc(int size, gfp_t flags)
功能：
1.从直接内存映射区分配内存
  访问效率高
2.分配内存大小最小32字节,最大4MB
3.分配的内核虚拟内存和物理内存都是连续的

参数：
size:指定分配内存的大小,单位为字节
flags:指定分配内存时的行为标志：
      GFP_KERNEL:告诉内核,请努力将这次内存分配搞定
                 如果内存不足,会导致休眠,所以不能用在
                 中断上下文
      GFP_ATOMIC:如果内存不足,不会进行休眠,而是立即返回
                 可以用在中断上下文
返回值：返回分配内核虚拟内存的首地址
例如：
```
void *addr;
addr = kmalloc(100, GFP_KERNEL);
if (addr == NULL)
     return -ENOMEM;

memcpy(addr, "hello,world", 12);
```

不再使用时,记得要释放内存：
void kfree(void *addr)

内核内存分配函数二：
__get_free_pages/free_pages
函数原型：
unsigned long __get_free_pages(gfp_t flags, int order)
功能：
1.从直接内存映射区分区
2.物理和虚拟内存上都是连续的
3.最大4MB

参数：
flags：表示分配内存时的行为
       GFP_KERNEL:告诉内核,请努力将这次内存分配搞定
                  如果内存不足,会导致休眠,所以不能用在
                  中断上下文
       GFP_ATOMIC:如果内存不足,不会进行休眠,而是立即返回
                  可以用在中断上下文
order:order=0,分配1页
      order=1,分配2页
      order=2,分配4页
      order=3,分配8页
      ...
返回值：返回分配内存的首地址,注意数据类型的转换

```
unsigned long addr;
addr = __get_free_pages(GPF_KERNEL, 2);
memcpy((void *)addr, "hello,world", 12);
```

内存不再使用时,记得要释放内存:

```
void free_pages(unsigned long addr, int order);
```

内核内存分配函数三:

```
vmalloc/vfree
void *vmalloc(int size);
```

函数功能:

1.从动态内存映射区分配内存
2.内核虚拟地址是连续的,但是对应的物理地址不一定连续
  动态内存映射区的内存访问效率低
3.理论默认最大分配120M
4.同样会导致休眠,所以不能用在中断上下文

释放内存:

```
void vfree(void *addr)
```

内核内存分配另类方法:
在内核启动参数中,添加vmalloc=? (例如vmalloc=250M)表示
内核启动时,将动态内存映射区的大小由原先的120M扩展到
250M

案例:跳转动态内存映射区的大小
1.启动开发板的linux系统,观察内核打印信息,首先确认
  直接内存映射区和动态内存映射区的大小(默认832M和120M)

2.重启开发板,进入uboot,执行:

```
setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/opt/rootfs
ip=192.168.1.110:192.168.1.8:192.168.1.1:255.255.255.0::eth0:on init=/linuxrc
console=ttySAC0,115200 vmalloc=250M
saveenv
boot
cat /proc/cmdline
```

继续观察内核的启动信息,再次查看直接内存映射区和动态内存映射区的大小是否有变化

内核内存分配另类方法:
在内核启动参数中,添加mem=? (例如mem=8M)表示
内核启动时,将物理内存的最后8M预留出来,将来给驱动单独使用
但是驱动使用时,必须利用ioremap函数进行地址映射,将最后的
8M物理内存和内核的动态内存映射区的虚拟内存做映射,一旦完成
映射,将来驱动访问映射的内核虚拟内存就是在访问最后的8M物理
内存

****

4.大名鼎鼎的ioremap函数
  4.1.明确
  嵌入式系统,CPU访问某个外设,必须要先获取到这个外设的基地址
  只要有了这个基地址,将来就可以以地址指针的形式访问:

```
*(unsigned long *)0x20000000 = 0x55;
```

  4.2.明确
  linux系统不管是用户空间还是内核空间,一律不允许访问外设的物理基地址
  ,要想访问,必须将设备的物理地址映射到用户虚拟地址上或者

内核虚拟地址上,将来访问映射的用户或者内核虚拟地址就是在
访问对应的物理地址

4.3.问：如何将外设的物理地址映射到内核的虚拟地址上呢？
　　答：利用大名鼎鼎的ioremap函数

4.4.ioremap使用
函数原型：
void *ioremap(unsigned long phys_addr, int size)
函数功能：
1.将外设的物理地址映射到内核的虚拟地址上
2.映射到内核1G的动态内存映射区

phys_addr:硬件外设的起始物理地址
size:映射的"物理内存"的大小

返回值：返回的映射的内核起始虚拟地址

参考代码：
LED1,LED2硬件寄存器信息：
GPC0CON:起始物理地址0xE0200060,大小4字节
GPC0DATA:起始物理地址0xE0200064,大小4字节

映射如下：
```
unsigned long *gpiocon *gpiodata;
gpiocon = ioremap(0xE0200060, 4);
gpiodata = ioremap(0xE0200064, 4);
```

或者：
由于发现两个硬件寄存器的内存空间都是连续的,物理上连续：
```
gpiocon = ioremap(0xE0200060, 8);
gpiodata = gpiocon + 1;

*gpiocon &= ~(0xf << 12);
*gpiocon |= (1 << 12);
...
```

内存不再使用时,记得要释放内存,解除地址映射
```
iounmap(void *addr);
```

例如：
```
iounmap(gpiocon);
```

案例：不再使用GPIO库函数,实现LED驱动,接口采用ioctl

小项目：寄存器编辑器
用户需求：随意能够查看和修改CPU的任何一个寄存器
思路：
应用测试思路：
```
./regeditor w regaddr regdata
./regeditor r regaddr //打印寄存器的值
```
例如：
```
./regeditor w 0xE0200080 0x11000 //将0x11000写入寄存器0xE0200080
./regeditor r 0xE0200080 //打印寄存器0xE0200080的值
```

应用程序编程思路：

```
struct reg_info {
      unsigned long regaddr;
      unsigned long regadata;
};

#define REG_WRITE     0x100001 //写寄存器
#define REG_READ      0x100002 //读寄存器

struct reg_info reg;

if (!strcmp) { //写寄存器
      reg.regaddr = strtoul(...);
      reg.regdata = strtoul(...);
      ioctl(fd, REG_WRITE, &reg);
} else { //读
      reg.regaddr = strtoul(...);
      //reg.regdata = ?
      ioctl(fd, REG_READ, &reg); //读寄存器
      printf("寄存器值=%#x\n", reg.regdata);
}
```

底层驱动ioctl接口：

```
 struct reg_info {
      unsigned long regaddr;
      unsigned long regadata;
};

#define REG_WRITE     0x100001 //写寄存器
#define REG_READ      0x100002 //读寄存器

reg_ioctl(...) {
      unsigned long *regbase;
      struct reg_info reg;
      copy_from_user(&reg, (struct reg_info*)arg, 8);
      regbase = ioremap(reg.regaddr, 4);
      switch(cmd) {
              case REG_WRITE:
              *regbase = reg.regdata;
              break;
              case REG_READ:
              reg.regdata = *regbase;
              copy_to_user((struct reg_info *)arg, &reg, 8);
              break;
      }
      iounmap(regbase);
}
```

检验：利用寄存器编辑器软件开关蜂鸣器

回顾：

1.linux内核等待队列编程方法2

工作队列：底半部一种实现方法,延后执行的一种手段
延后处理函数工作在进程上下文,可以休眠操作

等待队列：实现进程在内核中随时随地休眠,随时随地唤醒

编程实现两种形式：

方式1：
1.定义初始化等待队列头
2.定义初始化装载休眠进程的容器
3.添加
4.设置为休眠状态
5.进入真正的休眠，等待被唤醒
6.一旦被唤醒,设置为运行,移除
7.判断唤醒的原因
8.在某个地方进行唤醒,一般在中断处理函数中唤醒
方式2：
1.定义初始化等待队列头
2.直接进入休眠
3.在某个地方进行唤醒,一般在中断处理函数中唤醒
4.注意,编程口诀：唤醒前设置为真;唤醒以后设置为假

2.linux内核内存分配相关内容
2.1.linux内核1G虚拟内存的划分
划分的本质目的：一方面让内核访问到所有的物理地址
另一方面让内核的内存访问效果提高

2.2.linux内核内存分配的方法
kmalloc/kfree
__get_free_pages/free_pages
注意：GFP_KERNEL/GFP_ATOMIC
vmalloc/vfree
启动参数添加vmalloc=?
启动参数添加mem=?

2.3.linux内核地址映射的函数:ioremap
明确：软件一律不允许直接访问硬件外设的物理地址
问：如何将外设的物理地址映射到内核的虚拟地址上？
答：ioremap

3.问：如何将外设的物理地址映射到用户空间的虚拟地址上呢？
如果一旦完成这种映射,将来用户应用程序只需要在用户
空间访问映射的用户虚拟地址就是在访问实际的硬件物理地址？
也就是原先驱动中的这些代码：
```
*gpiocon &= ~((0xff << 12)); //gpiocon此时就是用户虚拟地址
...
```
这些代码都应该在应用程序中完成！
答：利用mmap

3.1.回忆：应用程序mmap系统调用函数的使用：
```
void *addr;
int fd = open("a.txt", O_RDWR);

addr = mmap(0, 0x1000, PROT_READ|PROT_WRITE,
            MAP_SHARED, fd ,0);
```
说明：
功能：将文件a.txt映射到当前进程的3G虚拟内存的MMAP内存映射区中的
某块虚拟内存上,一旦完成映射,将来访问映射的用户虚拟内存就是
在访问实际的文件；
明确：
"文件"：太抽象,文件实际代表的是一个硬件设备(硬盘),通常说
访问文件,其他就是在说访问硬盘的物理存储地址空间,由于linux系统
不允许直接访问硬盘的物理地址,需要进行映射,利用mmap就能够

将硬盘的物理地址映射到用户的虚拟地址上,将来访问映射的用户
虚拟地址就是在访问实际的硬盘物理地址

第一个参数0:让内核帮你在MMAP内存映射区中找一块空闲的虚拟内存
用来映射外设的物理地址

第二个参数0x1000:要映射的用户虚拟内存区域的大小
一般为页面大小的整数倍
第三个参数PROT_READ|PROT_WRITE:将来给这块虚拟内存设置一个访问权限
...
返回值addr：保存的就是内存映射区域的首地址

3.2.mmap系统调用的过程做了哪些工作：
1.当应用程序调用mmap系统调用函数
2.首先调用到C库的mmap函数的定义
3.C库的mmap函数将做两件事：
1.保存mmap系统调用号到R7寄存器
2.调用svc指令触发软中断异常
至此进程由用户态陷入内核态
4.进入内核的软中断处理的入口,做两件事：
1.从R7寄存器取出之前保存的系统调用号
2.以系统调用号为下标在内核的系统调用表中
找到对应的内核函数sys_mmap
调用此函数;
5.内核的sys_mmap同样做三件事：
1.内核会在当前进程的3G的MMAP内存映射区中找一块
空闲的虚拟内存区域来映射硬件的物理地址
2.一旦内核找到了空闲内存区域,内核用struct vm_area_struct
结构体帮你创建一个对象来描述这块空闲内存区域的属性(起始地址,大小,访问权限等)

```
struct vm_area_struct {
  unsigned long vm_start;//空闲虚拟内存区域的起始地址
  unsigned long vm_end;//结束地址
  pgprot_t vm_page_prot;//访问权限
  unsigned long vm_pgoff; //偏移量
  ...
};
```

总结：应用程序的mmap的返回值addr就是vm_start
此数据结构对应的对象由内核创建
3.最后内核的sys_mmap调用底层驱动的mmap接口,切记并且
内核将创建的vm_area_struct结构体对象指针传递给底层驱动
的mmap接口,底层驱动的mmap接口通过参数获取空闲虚拟内存
区域的属性!

6.底层驱动的mmap接口函数只做仅作一件事,将已知的空闲虚拟内存
和外设的物理内存建议映射即可,一旦建议映射,将来对硬件的访问
过程都是在应用程序中完成!
底层驱动mmap接口的角色类似“媒婆”,只负责映射(牵线),
具体的硬件访问操作跟mmap没有任何关系!

7.底层驱动mmap接口

```
struct file_operations {
     int (*mmap)(struct file *file,
                 struct vm_area_struct *vma)
};
```

功能：

此接口函数只做仅作一件事：将已知的用户虚拟地址和已知的
物理地址建立映射

参数：
file:文件指针
vma:指向内核描述空闲虚拟内存区域的属性对象
vma指向的对象由内核创建
对象描述内核找到的空闲用户虚拟内存区域的属性

底层驱动mmap接口只需调用一下函数即可完成最终的映射：

```
int remap_pfn_range(struct vm_area_struct *vma,
                    unsigned long from,
                    unsigned long to,
                    unsigned long size,
                    pgprot_t prot);
```

功能：将已知的用户虚拟地址和已知的物理地址进行映射
参数：
vma: 指向内核描述空闲虚拟内存区域的属性对象
from:已知要映射的用户虚拟的起始地址vm_start
to:已知要映射的物理地址,注意要将物理地址>>12
size:映射的用户虚拟内存的大小
prot：用户虚拟内存区域的属性

切记切记：mmap地址映射时,地址(不管是用户虚拟地址还是物理地址)
必须是页面大小的整数倍！
例如：
GPC1CON:0xE0200080
GPC1DATA:0xE0200084
注意：不能拿0xE0200080或者0xE0200084去做地址映射,这两个地址
不是页面大小的整数倍！所以拿0xE0200000做地址映射即可：

物理地址	用户虚拟地址
0xE0200000	vm_start
0xE0200080	vm_start + 0x80
0xE0200084	vm_start + 0x84

案例：利用mmap实现LED驱动

切记：利用mmap进行GPIO操作(输入和输出操作时),把cache关闭

```
vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
```

****

3.linux内核分离思想
3.1.案例：将TPAD开发板的LED1,LED2对应的GPIO更换成GPF1_4,GPF1_5
并且实现其驱动程序
分析：
明确：一个完整的硬件设备驱动必然包含两个内容：
纯硬件信息和纯软件信息(软件操作硬件)
如果要实现以上驱动,只需将昨天利用ioremap实现的LED驱动改造即可
分析改动的部分;
原先：
LED1->GPC1_3
LED2->GPC1_4
0xE0200080,0xE0200084
现在
LED1->GPF1_4
LED2->GPF1_5
0xE020.... 0xE020....

总结：
1.如果是硬件发生变化,之前的LED驱动修改的部分还是蛮多
2.可以采用C的#define来优化此驱动,但是#define不能给硬件
  添加额外的属性

3.问：如何让这个LED驱动具有更好的可移植性呢(如果将来仅仅是硬件发生变化,只需修改硬件信息,软件一旦写好,就无需改动,甚至都不用去看)
  答：采用linux内核的分离思想

3.2.linux内核分离思想
linux内核分离思想就是将一个完整的硬件驱动的纯硬件信息和纯软件信息彻底分开,一旦驱动的软件部分写好,将来无需在改动,硬件变化,只需修改硬件信息即可,将来驱动开发者的重心放在硬件部分即可.

linux内核分离思想的实现基于platform机制

3.3.linux内核的platform机制
机制实现原理：参见PPT

3.4.linux设备驱动采用platform机制实现,驱动开发者只需关注两个数据结构：
struct platform_device
struct platform_driver
问：如何使用?

1.struct platform_device的使用操作步骤：
```
  struct platform_device {
      const char      * name;
      int             id;
      u32             num_resources;
      struct resource * resource;
      ...
  };
```
  功能：描述硬件外设的纯硬件信息
  成员说明：
  name:描述硬件节点的名称,将来用于匹配
  id：硬件节点的编号,如果dev链表中仅有一个硬件节点,id=-1
      如果dev链表上有同名的多个硬件节点,id=0,1,2...
  resource:装载纯硬件信息
```
           struct resource {
              unsigned long start;
              unsigned long end;
              unsigned long flags;
              ...
           };
```
           功能：描述纯硬件信息
           start:硬件的起始信息
           end:硬件的结束信息
           flags:硬件的类型：
                 IORESOURCE_MEM：地址类信息
                 IORESOURCE_IRQ: IO类信息
  num_resources:纯硬件信息对象的个数

配套函数：
int platform_device_register(&硬件节点对象);
功能：注册硬件节点到dev链表
1.添加节点到dev链表
2.内核帮你遍历drv链表,取出每一个软件节点进行匹配
3.一旦匹配成功,内核调用软件节点的probe函数
4.顺便把匹配成功的硬件节点的首地址给probe函数
  完成硬件和软件的再次结合

void platform_device_unregister(&硬件节点对象);
功能：将硬件节点从dev链表上删除
      此时内核会调用remove函数

2.struct platform_driver
```
struct platform_driver {
    int (*probe)(struct platform_device *pdev);
    int (*remove)(struct platform_device *pdev);
    struct device_driver driver;
    ...
};
```
功能：描述驱动的纯软件信息
成员：
probe:当硬件和软件匹配成功,内核调用此函数
      形参pdev指向匹配成功的硬件节点
      注意：它是否被调用至关重要！只有此函数被调用
      一个完成的驱动诞生！
remove：当卸载软件或者硬件节点时,内核调用此函数
        形参pdev指向匹配成功的硬件节点
      注意：remove和probe永远是死对头！

driver：只需关注其中的name字段,将来用于匹配

配套函数：
int platform_driver_register(&软件节点对象);
功能：注册软件节点到drv链表
1.添加节点到drv链表
2.内核帮你遍历dev链表,取出每一个硬件节点进行匹配
3.一旦匹配成功,内核调用软件节点的probe函数
4.顺便把匹配成功的硬件节点的首地址给probe函数
  完成硬件和软件的再次结合

void platform_driver_unregister(&软件节点对象);
功能：将软件节点从drv链表上删除
      此时内核会调用remove函数

案例：利用platform机制优化LED驱动
实验步骤：
```
cd /home/drivers/
insmod led_drv.ko
insmod led_dev.ko
rmmod led_dev
rmmod led_drv
...
```

项目实施：
project.doc文档

根据文档从制作文件系统开始
内容：
制作根文件系统
tslib移植
QT移植

回顾：
1.linux内核mmap机制
目的：将硬件外设的物理地址映射到用户空间的虚拟地址上

内核的sys_mmap所做的内容：
1.帮你在MMAP内存映射区找一块空闲的虚拟内存区域,用来映射物理地址
2.一旦找到,内核用struct vm_area_struct数据结构来描述这块空闲的虚拟内存区域,其对象由内核创建
3.内核sys_mmap最终调用底层驱动的mmap接口,并且将描述空闲的虚拟内存区域的对象的首地址传递给驱动的mmap接口

底层驱动的mmap仅作只做一件事：映射
```
struct file_operations {
 int (*mmap)(struct file *file, struct vm_area_struct *vma)
};//vma指针指向的对象由内核创建,用来描述空闲的虚拟内存区域
```
切记：映射时,地址必须是页面大小的整数倍！

2.linux内核platform机制
实现linux内核的分离思想
实现将硬件和软件彻底分开
一旦软件写好,无需在进行改动
将来只需要关注硬件差异即可,将来硬件有所变动,只需改动硬件部分
画图
总结：
1.probe函数是否被调用至关重要！
2.驱动开发者利用platform机制实现设备驱动,只关注两个数据结构：
```
struct platform_device
    .name 相当重要
    .id
    .resource
            .start
            .end
            .flags
              IORESOURCE_MEM
              IORESOURCE_IRQ
    .num_resources
struct platform_driver
    .driver = {
            .name 相当重要
    },
    .probe 匹配成功内核调用,形参指向匹配成功的硬件信息
    .remove 删除软件或者硬件节点内核调用
```

3.什么遍历,什么匹配,什么调用,什么传参都是由内核完成！
4.probe函数一般所做的工作：
0.调用代表一个完整的硬件设备驱动产生
1.通过形参获取硬件信息
```
  struct resource *platform_get_resource(
            struct platform_device *pdev,
```

```
            int flags;
            int index;
);
```

函数功能：通过probe函数的形参pdev或者resource描述硬件信息
pdev:指向匹配成功的硬件信息
flags：硬件信息的类型
IORESOURCE_MEM/IORESOURCE_IRQ
index:同类资源的偏移量
返回值：返回获取到的硬件信息的首地址

2.处理获取到的硬件信息
该申请的申请
该注册的注册
该映射的映射
该初始化的初始化
3.注册硬件操作接口(字符设备驱动或者混杂设备驱动)
.open
.release
.read
.write
.unlocked_ioctl
.mmap

remove跟probe对着干！

****

3.I2C总线(IIC总线)
面试题：谈谈对I2C总线理解
3.1.I2C总线的功能
计算机CPU和外设的通信方式很多种：
GPIO,例如LED,按键
总线(地址线数据线),例如内存,NorFlash,DM9000
UART,例如BT,GPS
I2C,例如重力传感器,触摸屏芯片
一线式,例如DS18B20温度传感器
SPI,例如NorFlash
等
I2C总线是CPU和外设通信的一种数据传输方式

3.2.I2C总线的定义
两线式串行总线
解释：
“两线式”：CPU跟外设的数据通信只需2根信号线,分别是SCL时钟控制信号线
和SDA数据线,画简要的连接图
SDA数据线：用来传输数据,CPU和外设都可以控制,但是不能同时
控制,例如CPU向外设写数据,数据线由CPU控制
CPU从外设读取数据,数据线由外设控制
问：由于外设的处理速度远远慢于CPU,CPU和外设如何
保证数据传输的正常呢？
答：关键靠SCL时钟线
注意：I2C总线数据传输从数据的高位开始！

SCL时钟线：同步双方的数据传输,保证数据传输正常，只能由CPU控制
例如：CPU在SCL为高电平时,将数据放到数据线上
设备就在同周期的SCL为低电平时,从数据线上读取数据
SCL为低电平时,数据线上的数据保持稳定不变！

“串行”：CPU和外设的数据通信是一个时钟周期传输1个bit位
“总线”：CPU和外设通信的两根信号线上可以挂接多个外设
画出简要的连接图
注意：SDA和SCL都会连接一个上拉电阻,默认电平都为高电平!

问：CPU如何找到总线上要访问的某个外设?
问：如果CPU找到这个外设,CPU如何通过两根信号线和外设通信呢?
问：SDA和SCL如何搭配使用呢?
答：答案在I2C总线传输协议中

3.3.I2C总线协议相关概念
CPU=主设备=master
外设=slave
MSB:高位
LSB:低位

START信号：又称起始信号,每当CPU要访问总线上某个外设,首先CPU向总线
发送一个START信号,此信号只能由CPU发起
SCL为高电平,SDA由高向低跳变产生START信号
类似“同学们,上课了”
STOP信号：又称结束信号,每当CPU要结束对某个外设的访问,CPU
只需向总线发送一个STOP信号即可,此信号同样只能由CPU发起
SCL为高电平,SDA由低向高跳变产生STOP信号
类似“同学们,下课”
画图START和STOP时序图
读写位：如果CPU要读设备,读写位=1;如果CPU要写设备,读写位=0
设备地址：同一个I2C总线上的外设都有一个唯一的设备地址(类似身份证号)
表示外设在总线上的唯一性
如果将来CPU要访问总线上某个外设,CPU只需向总线
发送这个外设对应的设备地址即可
类似“某某同学,请回答问题”
切记：设备地址不包含读写位

读设备地址=设备地址<<1|1
写设备地址=设备地址<<1|0

问：外设的设备地址如何确定?
答：
以LM77温度传感器为例P8：
确定LM77设备地址=10010A1A0(A1A0都接地)=》
1001000(高位补0)=01001000=0x48
=>
LM77的：
读设备地址=0x48<<1 | 1
写设备地址=0x48<<1 | 0

以AT24C02存储器为例P11:
确定AT24C02设备地址=1010A2A1A0(A2A1A0都接地)=》
1010000(高位补0)=01010000=0x50
读设备地址=0x50<<1 | 1
写设备地址=0x50<<1 | 0

以ADP8860背光灯控制芯片为例：
0101010x=>
设备地址=0101010(去x,高位补0)=00101010=0x2A

读设备地址=0x2A<<1 | 1
写设备地址=0x2A<<1 | 0

ACK信号:又称应答信号,表示CPU和外设的通信状态
低电平有效
类似“老师，我在”

总结：
1.CPU要想访问总线上某个外设,CPU只需向总线发送这个外设的
设备地址即可
I2C总线数据传输一周期一bit,一次一字节

问：CPU一旦通过设备地址找到某个外设,如何通过两根信号线
和外设进行数据通信呢？
答：答案都在外设的芯片手册中,重点关注其中的操作时序图
以CPU读取LM77温度传感器2字节数据为例P12：
1.CPU向总线发送START信号
2.CPU向总线发送外设的设备地址包括读写位
3.如果外设存在于总线上,外设在第九个时钟周期会给CPU发送
一个ACK信号,低电平有效
4.设备向CPU发送两字节数据的高字节
5.CPU读取数据以后,CPU同样在第九个时钟周期给外设一个有效的
ACK信号
6.设备继续向CPU发送两字节数据的低字节
7.CPU读取数据以后,没有在第九个时钟周期给外设一个有效的
ACK信号
8.数据读取完毕,CPU向总线发送STOP信号结束此次的数据读取操作
边说边画图(框框圈圈)

以AT24C02存储器的读写为例P11，字节写时序图：
AT24C02存储器特性：
存储容量256字节
内部地址编址：0x00~0xff
用户需求：将数据'A'写入到AT24C02的片内的0x10地址存储空间中
硬件操作流程：
1.CPU向总线发送START信号
2.CPU向总线发送设备地址包括读写位
3.如果设备存在于总线上,设备在第九个时钟周期给CPU发送一个
有效的ACK信号
4.CPU向设备发送要访问的片内地址0x10
5.设备接收到了CPU要访问的片内地址,设备同样在第九个时钟
周期给CPU一个有效的ACK信号
6.CPU最后向设备发送要写入的数据'A'
7.设备将数据写入到片内0x10地址以后,设备同样在第九个时钟
周期给CPU一个有效的ACK信号
8.CPU向总线发送STOP结束数据的操作
边说边画框框圈圈图

再以CPU读取AT24C02任意片内地址存储空间的数据为例P2随机读试图：
具体参见时序图

最后以CPU让HMC6352指南针传感器进入休眠模式为例：
1.CPU发送START信号
2.CPU发送设备地址<<1|0
3.设备如果存在于总线上,设备在第九个时钟周期给CPU发送一个
有效的ACK信号

4.CPU向设备发送命令'S'=0x53,让设备进入休眠模式
5.设备接收到命令以后，设备在第九个时钟周期给CPU发送一个
有效的ACK信号
6.最后CPU向总线发送STOP信号结束访问
边说边画圈圈框框图

切记：任何外设的访问必须严格按照时序图进行！！！

问：SDA和SCL如何搭配使用呢？
答：以CPU向设备写入数据为例：
CPU应该在SCL为低电平的时候将数据放到数据线上
那么设备就应该在SCL同周期的高电平从数据线上读取数据
"低放高取"
一定要画出相应的时序图,以CPU向HMC6352发送‘S’为例P4：
可以仅画START时序,写0,1时序即可

会画波形，将来也会分析示波器的波形图！！！！

回顾：
1.linux内核platform机制
目的：将硬件和软件分离
一个虚拟总线：platform_bus_type
两个链表：dev链表(硬件)和drv链表(软件)
两个数据结构：

```
struct platform_device
        .name
        .id
        .resource
        .num_resources
        .dev
                .release
struct platform_driver
        .driver
                .name
        .probe
                1.获取硬件信息
                        platform_get_resource
                2.处理硬件信息
                  4个该
                3.注册硬件操作接口
        .remove
                对着干
```

四个配套函数
platform_device_register
platform_device_unregister
platform_driver_register
platform_driver_unregister
四个什么都是由内核来完成

一个关心：probe函数是否被调用

2.I2C总线
面试题：谈谈对I2C总线的理解
2.1.I2C总线功能
2.2.I2C概念

2.3.三个问题
2.4.协议设计概念
START
STOP
设备地址
读写位
ACK
2.5.访问过程
举例子：以CPU向AT24C02片内地址0x10存储空间写入字符'A'为例
结论：一切的操作都是在芯片手册的操作时序图中
2.6.配合
画出时序图
START 设备地址=0x50 W ACK

3.linux内核I2C驱动开发
3.1.明确I2C总线实际的硬件操作
研究对象转移：
CPU访问外设
转移到CPU访问SDA和SCL
转移到CPU访问I2C控制器
转移到CPU访问I2C控制器对应的寄存器

3.2.linux内核I2C驱动的分类
I2C总线驱动：
管理的硬件是I2C控制器
此驱动操作I2C控制器最终帮你发起SCL和SDA的时序
注意：像START,ACK,读写位,STOP这些都是标准信号,I2C控制器
自动完成,但是设备地址,操作的芯片的片内地址和片内数据
控制器是不知道的,这些由I2C设备驱动来告诉
I2C总线驱动
总线驱动由CPU芯片厂家实现好,只需配置内核支持即可：

```
cd /opt/kernel
make menuconfig
  Device Drivers->
        I2C supports->
                i2c hardware bus supports...-> //在此
                        <*> s3c2410 i2c ...
```

I2C设备驱动：
管理的硬件是I2C外设本身
像设备地址,片内地址,片内数据都是跟
外设相关,I2C设备驱动需要将这些数据
信息丢给I2C总线驱动,最终完成硬件SCL
和SDA的时序传输
I2C驱动开发的重点！

问：以上两个驱动如何关联呢？

3.3.linux内核I2C驱动框架(分层思想)
例如：以CPU向AT24C02片内地址0x10存储空间写入数据'A'为例：

应用层：
作用：就是要获取到底层硬件将来要操作的数据信息(片内地址和片内数据)

```
struct at24c02_info {
        unsigned char addr;
        unsigned char data;
};
```

```
        struct at24c02_info at24c02;
        at24c02.addr = 0x10;
        at24c02.data = 'A';
        //将应用层的数据丢给I2C设备驱动
        ioctl(fd, AT24C02_WRITE, &at24c02);
```

I2C设备驱动层：

```
        从用户获取要操作的数据信息或者将数据信息丢给用户
        同时还要将数据信息利用SMBUS接口丢给I2C总线驱动
        或者利用SMBUS接口从I2C总线驱动获取数据
        at24c02_ioctl(file, cmd, arg) {
           struct at24c02_info at24c02;
           copy_from_user
               结果：
               at24c02.addr = 0x10
               at24c02.data = 'A'
           switch ....
               case AT24C02_WRITE: //写
                    I2C设备驱动利用内核提供的SMBUS接口函数
                    将这些数据信息丢给I2C总线驱动：
                    smbus_xxxx(数据信息);
               break

        }
```

SMBUS接口层：

```
        连接I2C设备驱动和I2C总线驱动
        桥梁作用
        内核已经实现！
        smbus_xxx
        0x10
        'A'
```

I2C总线驱动层：

```
        唯一的作用就是操作I2C控制器,启动硬件的最终传输
        设备地址同样由I2C设备驱动来告诉
        当然传输的数据信息来自用户0x10,'A'
```

硬件层：

```
        START->设备地址|0->ACK->0x10->ACK->'A'->ACK->STOP
```

问：linux内核I2C设备驱动如何编写呢？
答：同样采用分离思想(bus-device-drivers编程模型)
1.首先内核已经定义好了一个虚拟总线叫i2c_bus_type
  在这个总线上维护着两个链表dev链表和drv链表
2.dev链表上每一个节点描述的I2C外设的硬件信息,对应的
  数据结构为struct i2c_client, 每当添加一个I2C外设的硬件信息时,
  只需利用此数据结构定义初始化一个对象,添加到dev链表以后
  内核会帮你遍历drv链表,取出drv链表上每一个软件软件跟这个
  硬件节点进行匹配(匹配通过内核调用总线提供的match函数,比较
  i2c_client的name和i2c_driver的id_table的name)如果匹配成功
  内核调用软件节点的probe函数,并且把匹配成功的硬件节点的
  首地址给probe函数
3.drv链表上每一个节点描述的I2C外设的软件信息,对应的
  数据结构为struct i2c_driver, 每当添加一个I2C外设的软件信息时,
  只需利用此数据结构定义初始化一个对象,添加到drv链表以后

内核会帮你遍历dev链表,取出dev链表上每一个硬件节点跟这个
软件节点进行匹配(匹配通过内核调用总线提供的match函数,比较
i2c_client的name和i2c_driver的id_table的name)如果匹配成功
内核调用软件节点的probe函数,并且把匹配成功的硬件节点的
首地址给probe函数

总结：如果要实现一个I2C外设的I2C设备驱动只需关注两个数据结构：
struct i2c_client
struct i2c_driver

3.4.struct i2c_client
```
struct i2c_client {
    unsigned short addr;//设备地址,将来寻找外设
    char name[I2C_NAME_SIZE];//将来用于匹配
    int irq; //中断号
    ...
};
```
功能：描述I2C外设的硬件信息
切记：addr,name两个成员必须要进行初始化！
结论：驱动开发者不会自己去拿这个结构体去定义初始化和注册
一个硬件节点对象到内核,而是利用以下结构体间接完成
以上操作(用i2c_client定义初始化注册对象)

神秘的数据机构：
```
struct i2c_board_info {
    char type[I2C_NAME_SIZE];
    unsigned short  addr;
    int irq;
    ...
};
```
功能：将来驱动开发者用此数据结构去定义初始化注册
一个I2C外设的硬件信息到内核,将来内核会根据
你提供的硬件信息,内核会帮你定义初始化和注册
一个i2c_client硬件节点对象到内核dev链表中
成员：
type:硬件信息的名称,将来这个字段的内容会赋值给
i2c_client的name成员,用于匹配
addr:I2C外设的设备地址,将来这个字段的内容会赋值给
i2c_client的addr成员,用于找某个外设
irq:I2C外设的中断号,将来这个字段的内容会赋值给
i2c_client的irq成员
切记：type,addr必须要初始化！
内核鼓励使用I2C_BOARD_INFO宏对type,addr进行初始化！
例如：I2C_BOARD_INFO("at24c02", 0x50)

配套函数：
注册i2c_board_info描述的硬件信息到内核的函数：
```
int i2c_register_board_info(int busnum,
                    const struct i2c_board_info *info,
                                int number)
```
功能:i2c_board_info描述的硬件信息到内核
参数：
busnum:I2C外设所在的I2C总线编号(从0开始),通过原理图来获取
info:传递定义初始化的I2C外设的硬件信息对象首地址
number:硬件信息的个数
结果：将来内核根据你注册的硬件信息,内核会帮你定义初始化

和注册一个i2c_client对象到内核中
切记切记：i2c_board_info定义初始化和注册不能采用insmod
和rmmod,此代码必须放在平台代码中完成！
TPAD的平台代码：arch/arm/mach-s5pv210/mach-cw210.c

案例：向内核添加AT24C02的硬件节点到内核
实施步骤：
1.cd /opt/kernel
2.vim arch/arm/mach-s5pv210/mach-cw210.c 在头文件的后面添加如下内容：
//定义初始化AT24C02的硬件信息
```
static struct i2c_board_info at24c02[] = {
    {
        I2C_BOARD_INFO("at24c02", 0x50)
    }
};//"at24c02"将来用于匹配,0x50表示设备地址,用于寻找外设
```

3.vim arch/arm/mach-s5pv210/mach-cw210.c 在任何一个函数中调用一下函数完成对I2C外设硬件信息的注册,推荐找到：
.init_machine = smdkc110_machine_init,
找到smdkc110_machine_init,在此函数中调用：
i2c_register_board_info(0, at24c02, ARRAY_SIZE(at24c02));
保存退出
4.make zImage
cp arch/arm/boot/zImage /tftpboot
5.用新内核重启开发板
至此内核就有了AT24C02的硬件信息,内核也帮你定义初始化注册一个i2c_client,此时此刻它会静静等待着软件节点的到来！

3.5.struct i2c_driver
```
struct i2c_driver {
    int (*probe)(struct i2c_client *client,
                 const struct i2c_device_id *id);
    int (*remove)(struct i2c_client *client);
    const struct i2c_device_id *id_table;
    ...
};
```
作用：描述I2C外设的软件信息
id_table:I2C外设的标识,其中的name用于匹配
```
    struct i2c_device_id {
            char name[I2C_NAME_SIZE];//用于匹配
            unsigned long data; //给probe函数传递的参数
    };
    例如：
    static const struct i2c_device_id at24_id[] = {
            { "at24c02", 0 },
            { }
    };//"at24c02"将来用于匹配
```

probe:匹配成功,内核调用此函数
形参client指向匹配成功的硬件信息
id指向驱动自己定义初始化的i2c_device_id对象,就是at24_id
remove：卸载软件节点,内核调用此函数

配套函数：
i2c_add_driver(&软件节点对象); //注册

drv
    i2c_del_drier(&软件节点对象); //卸载

案例：编写TPAD上AT24C02存储器的软件节点代码
      注意：前提是内核已经有了AT24C02的硬件信息
实施步骤：
1.从ftp下载源码
2.mkdir /opt/drivers/day12/
  cp at24c02_1 /opt/drivers/day12
  cd /opt/drivers/day12/at24c02_1 潜心研究代码
  make
  cp at24c02_drv.ko /opt/rootfs/home/drivers
开发板测试：
insmod /home/drivers/at24c02_drv.ko
观察驱动的probe函数是否被调用

rmmod at24c02_drv //观察remove函数是否被调用

案例：编写TPAD上AT24C02存储器的软件节点代码
      注意：前提是内核已经有了AT24C02的硬件信息
实施步骤：
1.从ftp下载源码
2.mkdir /opt/drivers/day12/
  cp at24c02_2 /opt/drivers/day12
  cd /opt/drivers/day12/at24c02_2
  潜心研究at24c02_drv.c at24c02_test.c
  make
  cp at24c02_drv.ko /opt/rootfs/home/drivers
  arm-linux-gcc -o at24c02_test at24c02_test.c
  cp at24c02_test /opt/rootfs/home/drivers/

开发板测试：
insmod /home/drivers/at24c02_drv.ko
ls /dev/at24c02 -lh
./at24c02_test

SMBUS接口函数的使用操作步骤：
1.首先打开I2C外设的芯片手册,找到对应的操作时序图
2.然后打开内核关于smbus接口的说明文档：
  内核源码\Documentation\i2c\smbus-protocol
3.在文档中根据手册的时序要求找到对应的函数
  例如：
  按字节写的时序：S->设备地址<<1|0->ACK->片内地址->ACK->数据->ACK->STOP
  找到此函数：i2c_smbus_write_byte_data()
  此函数说明提示：S Addr Wr [A] Comm [A] Data [A] P //完全符合
4.赋值函数名,在sourceinsight中找到函数的定义
  在自己的驱动代码中填充参数即可
