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
	...
};
```