# 一、Linux系统软件体系架构

应用程序、库、内核、驱动程序的关系：

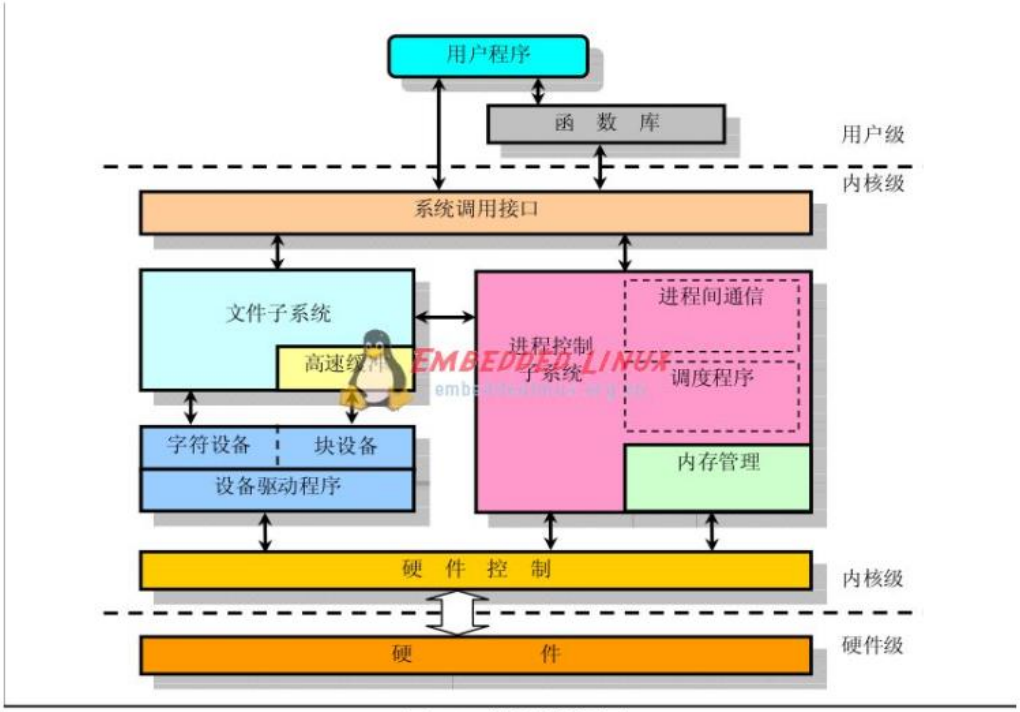
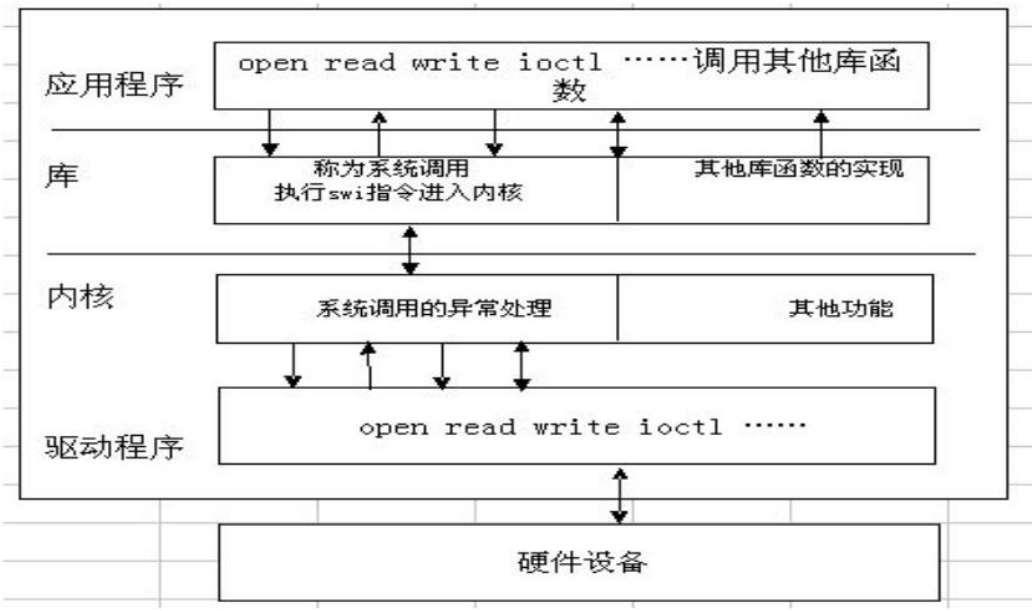



图 2-4 内核结构框图




例：应用程序调用内核的流程

1. 应用程序使用系统调用提供的 open 函数打开代表 LED 的设备文件。
2. open 函数最后会执行“swi”指令，这条指令会引起 CPU 异常，进入内核。
3. 内核的异常处理函数根据 open 函数的参数找到相应的驱动程序，返回一个文件句柄给应用程序。
4. 应用程序得到文件句柄后， 使用系统调用提供的 write 或 ioctl 函数发出控制命令。
5. write 或 ioctl 函数最后也会执行“swi”指令，这条指令会引起异常，再次进入内核。
6. 内核的异常处理函数根据 write、ioctl 参数调用驱动程序的相关函数，点亮 LED。

应用层通过操作系统调用接口函数(open、read、write、ioctl、mmap 等)来调用文件系统里的设备节点，从而操作硬件资源；

文件系统里的设备节点是驱动程序注册产生的（在/dev 目录下)。

它们都是设置好相关寄存器后，执行某条指令引发异常进入内核。对于 ARM 架构的 CPU,这条指令为 swi。应用程序调用接口函数时，将会调用驱动程序中的 open、read、write、ioctl、mmap 等函数来进行操作（初始化、读、写等)。与应用程序不同，驱动程序从不主动运行，它根据应用程序的要求进行初始化和读写。驱动程序加载进内核时，只是告诉内核“我在这里，我能做这些工作”，至于这些“工作”何时开始，取决于应用程序。

# 二、驱动原理

## 1、 设备分类

a、字符设备(c)： 像普通文件或字节流一样，以字节为单位，遵循单个字符读写操作。字符设备可以通过设备文件节点访问。 （串口、 LCD 屏、 pwd、led、 key 等)；

b、块设备(b)： 以块为单位进行随机读写。

块设备也是通过文件节点来访问。 (flash、 sd 卡、IDE 硬盘、 SCSI 硬盘等)；

c、网络设备(s)：它和字符设备，块设备区别在于没有设备节点，通过内部的 buffer 缓存实现它的数据传输，其实就是把要发送的原始数据加上一些协议栈的头，最终发送出去。 遵循套接字编程。 (网卡)；

相对于字符设备，网络设备是有结构的、成块的；但是同样相对于块设备，它的块大小是不确定的，网络报文大到几千字节小到几字节。内核给网络设备提供了一整套数据包传输相关的函数，不是 open write 而是 socket 套接字。linux 对网络的提供很强大的支持。

## 2、设备节点

设备节点是驱动程序给用户空间提供访问接口的文件，不同于普通的文件，存放/dev/目录下（网络设备 dev 下没有设备文件)，每一个设备节点都有一个设备号， 由 32 位的数来描述， 设备号由主设备号和次设备号组成。

主设备号：同一类的设备具有相同的主设备号 --- 高 12 位；

次设备号：用来区别同一类设备下不同的个体 --- 低 20 位；

## 3、地址操作

Linux 使用的地址全部是虚拟地址。内核中不能直接访问物理地址；

意味着驱动程序要把物理地址转换为虚拟地址。

void __iomem * ioremap (unsigned long phys_addr(物理地址), unsigned long size(大小))

//内存映射，返回一个对应的虚拟地址；

void iounmap (volatile void __iomem *addr) //解除内存映射；

# 三、杂项设备注册方式

（主设备号默认全部为 10 ）

头文件： #include <linux/miscdevice.h>

函数：int misc_register(struct miscdevice * misc)； //杂项设备注册函数

参数：

```
struct miscdevice
{
  int minor;        //次设备号， 255 是代表自动分配次设备号；
```

```
    const char *name;        //设备节点名；  /dev 目录下
    const struct file_operations *fops;          //关联文件操作函数集；
  /* 下面的成员是供内核使用 ，驱动编写不需要理会 */
    struct list_head list;          //内核自动填充
    struct device *parent;
    struct device *this_device;
    const char *nodename;
    umode_t mode;
  };
  struct file_operations          //文件操作函数集；
  {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, \
                                                unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,\
                                                                size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, \
                                                                size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,loff_t len);
  };    //操作设备节点的函数接口
```

卸载设备函数：misc_deregister(struct miscdevice * misc)

补充：

教科书上讲 C 语言结构体初始化是按照顺序方式来讲的，没有涉及到乱序的方式。

顺序初始化：必须要按照成员的顺序进行，缺一不可，如果结构体比较大，很容易出现错误，而且表现形式不直观，不能一眼看出各个 struct 各个数据成员的值。

乱序初始化：是 C99 标准新加的，比较直观的一种初始化方式。乱序初始化成员可以不按照顺序初始化，而且可以只初始化部分成员，扩展性较好。

linux 内核中采用乱序初始化方式初始化结构体。用点（.）符号，它是 C99 标准。

例：

```
38 static struct file_operations led_fops = {
39     .owner = THIS_MODULE,
40     .open  = leds_open,
41     .release = leds_close,
42 };
```

系统调用号： /usr/include/asm/unistd.h
系统调用表： /root/linux-3.5/arch/arm/kernel/calls.S

# 四、经典设备注册方式

注册了一个设备， 有一个主设备号以及关联了一个file_operations。

头文件：#include <linux/fs.h>

早期版本的设备注册使用函数 register_chrdev()，操作成功返回值为 0。在关闭设备时，通常需要解除原先的设备注册，使用函数 unregister_chrdev()。其中主设备号和次设备号不能大于 255。

函数：register_chrdev(unsigned int major,const char * name,const struct file_operations * fops)

参数 1： 大于 0 --- 直接指定主设备号，返回 0；
等于 0 --- 自动分配主设备号，并返回；

参数 2： 名字 ； //和 dev 下面名字无关；

参数 3： 关联的文件指针函数集；

函数：void unregister_chrdev(unsigned int major, const char *name)

参数：major： 主设备号
name： 设备名，使用 register_chrdev 注册的设备名

这种方式不会在 dev 下面创建设备节点， 调用一个 register_chrdev 注册后， 256 个次设备号就都被占用完了。也就是说一个主设备号只能使用 register_chrdev 函数注册一次。

有两种方式创建设备节点：

a、手动创建：

mknod /dev/xxx c 主设备号 次设备号

b、通过代码实现自动创建：（多两个函数）

register_chrdev(int mojor,char *name,struct file_operations *fops);

static struct class * adb_dev_class = class_create(THIS_MODULE,"adb");

device_create(adb_dev_class, NULL, MKDEV(ADB_MAJOR, 0),NULL, "adb");

先创建设备类：class_create(THIS_MODULE,类名); //返回了一个类指针

创建设备节点：device_create(类);

device_create创建一个节点(物理介质文件),将设备和节点相关联；

应用层 open("/dev/led",O_RDWR);
内核层这个设备指定的open函数。

```
struct   device *device_create(
                struct class *cls, // 设备的类 类指针   让那个类来管理
                struct device *parent,     // NULL
                dev_t devt,       // 主设备号和次设备号的组合 32 位，主 12 位 次 20 位
                void *drvdata,         // NULL,设备私有数据
                const char *fmt, ...    /*可以格式化的 fmt:/dev/下的设备名*/
);
```

dev_t　　//unsigned int ，代表完整设备号；

MKDEV(主设备号，次设备号);　　//已知主设备号和次设备号合成完整设备号；

MAJOR(dev_t dev)；　　//获取主设备号；

MINOR(dev_t dev)；　　//获取次设备号；

例：

```
major=register_chrdev(0,"hello",&myfops);
my_class=class_create(THIS_MODULE,"XYD");
device_create(my_class,NULL,MKDEV(major,0), NULL, "HELLO");
```

第一个名字(“hello”)：　主设备号，命令：cat /proc/device 可以查看到；

第二个名字(“XYD”)：管理设备的类名，命令：ls /sys/class/　可以查看到类名；

第三个名字(“HELLO”)：创建/dev 下面的设备节点名；

# 五、linux 系统 2.6 版本的注册方式

## 1、这个方式定义了一个核心结构体（在头文件 include/linux/cdev.h 里)。

使用 struct cdev 结构体来描述一个设备，只需填充这个结构体就可以完成设备注册。（写设备驱动只要填充 owner， ops， dev，count 这几个就可以了）

```
struct cdev
{
    struct kobject kobj;
    struct module *owner;                    //拥有者指针
    const struct file_operations *ops;       //文件操作集
    struct list_head list;                   //链表头
    dev_t dev;                               //完整的设备号
    unsigned int count;                      //次设备号
};
```

ops：void cdev_init(struct cdev *cdev, const structfile_operations *fops)指定关联文件操作指针集；

dev：　起始设备号（ 包含主设备号和次设备号)，dev_t：实际上是一个 u32 类型；

count：次设备号的数量（连续的)，从 dev 号开始；

设备号：使用 dev_t 来表示一个设备号。其中高 12 位是主设备号(2^12=4K，其中 10 是给杂项设备使用)，低 20 位是次设备号（2^20=1M)。

合成设备号：MKDEV(ma,mi)：　ma：主设备号，　mi：　次设备号；

分解设备号：MAJOR(dev) ：从设备号 dev 中分解出主设备号；

MINOR(dev) ：从设备号 dev 中分解出次设备号；

特征：

a. 安装后，不会自动创建/dev/设备文件节点，　需要手动使用 mknod 命令创建。

b. 调用一个 cdev_add 注册后，指定数量的次设备号被占用完了。数量可以自己指定，一个主设备可以使用 cdev_add 函数注册多次。

c. 设备号使用前需要先申请：　register_chrdev_region 或 alloc_chrdev_region 函数申请。

## 2、申请设备号

A、静态申请：(已知主设备号)

函数：int register_chrdev_region(dev_t from, unsigned count, const char*name)

参数：from：设备号的指定值（起始值)；

count：连续申请的次设备号的数量；

name：设备名；

返回值：　0：成功；负数：失败

B、动态申请：（自动分配一个设备号）

函数：int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,const char*name)

参数：dev ： 存放分配成功的第一个设备号的指针。

baseminor: 起始次设备号

count：连续申请的次设备号的数量；

name：设备名；

返回值： 0：成功， 负数：失败

C、注销设备号：

函数：void unregister_chrdev_region(dev_t from, unsigned count);

参数：from：设备号的起始值；

count: 连续注销次设备号的数量；

### 3、 将设备添加到内核

函数：int cdev_add(struct cdev *cdev, dev_t dev, unsigned count)

参数：cdev :struct cdev 字符设备结构指针

dev:设备号，包含主和次设备号。

count: 连续次设备号的个数；

返回值：成功返回 0，失败返回负数；

注销函数：void cdev_del(struct cdev *cdev)

参数：cdev :struct cdev 字符设备结构指针

### 4、备注

MAJOR(dev_t dev)； //获取主设备号

MINOR(dev_t dev)； //获取次设备号

MKDEV(主设备号,次设备号) //合成完整设备号

unsigned int iminor(struct inode *inode); //次设备号

unsigned int imajor(struct inode *inode); //主设备号

注：通过提取次设备号， 可以判断打开的设备节点。

### 5、编写一个 Linux 2.6 字符设备驱动步骤

模块的初始化函数：

1). 申请设备号；

2). 分配 cdev 结构空间；

3). 初始化 cdev 结构；

4). 注册已经初始化好的 cdev 结构；

模块卸载函数：

1). 注销 cdev 结构；

2). 释放 cdev 结构空间；

3). 释放设备号；

# 六、补充知识点

### 1、设备号：主设备号，次设备号

设备号是一个数字，它是设备的标志。linux 下的设备，一般都会在/dev 目录下生成一个设备文件（也就是设备节点），用户程序通过设备文件对设备进行操作。

设备文件通过使用 mknod 命令来创建，其中指定了主设备号和次设备号。主设备号表明设备的类型（例如串口设备，硬盘等），与一个确定的驱动程序对应；次设备号通常是用于标明不同的属性，例如不同的使用方法、不同的位置、不同的操作等，它标志着某个具体

的物理设备。

对于 linux 的驱动程序来说，一般一个驱动代码针对一类设备，驱动代码中再根据次设备号来区分要执行哪一部分，/dev 下 c 开头的文件表示字符设备，b 开头的是块设备，l 表示是连接文件，d 开头是目录；

例：mknod /dev/ttySAC0 c 204 64

在 dev 目录下创建一个设备节点,指向一个主设备号为 204,次设备号为 64 的设备。

## 2、杂项设备驱动模型

杂项设备是在嵌入式系统中用得比较多的一种设备驱动。在 Linux 内 核 的 include/linux/miscdevice.h 文件里，要把自己定义的 miscdevice 从设备定义在这里。

一般因为这些字符设备不符合预先确定的字符设备范畴，或者不方便归类，所以把这些设备归类为杂项设备，这些设备采用主设备号 10，一起归于 miscdevice，其实 misc_register 就是用主设备号 10 调用 register_chrdev()的。misc 设备其实也是特殊的字符设备。

**以下是在 insmod 驱动的时候执行的：**

struct file_operations 当注册设备的时候，需要指定设备的（文件）操作函数。

```
static struct file_operationsled_fops =
{
   .owner = THIS_MODULE,   //所有者;
   .open = leds_open,     //指定 open 函数的实现;
   .release = leds_close,   //指定 close 函数的实现;
};
int (*open) (struct inode *, struct file *);     //指定打开设备文件的时候执行的工作;
```

在操作设备前必须先调用 open 函数打开文件,可以干一些需要的初始化操作。当然，如果不实现这个函数的话，驱动会默认设备的打开永远成功。打开成功时 open 返回 0。

```
int (*release) (struct inode *, struct file *);   //关闭设备文件的时候执行的工作;
```

当设备文件被关闭时内核会调用这个操作，当然这也可以不实现，函数默认为 NULL。关闭设备永远成功。

.owner：它是一个指向拥有这个结构的模块的指针. 这个成员用来在它的操作还在被使用时阻止模块被卸载.几乎所有时间中，它被简单初始化为 THIS_MODULE，一个在<linux/module.h>中定义的宏。

## 3、 struct file 结构体中提取次设备号

应用程序匹配驱动是通过设备号来匹配，而不是通过设备名。所以/dev 下文件名是什么不重要，重要是这个文件和设备号。获得次设备号：MINOR(file->f_dentry->d_inode->i_rdev)

## 4. 杂项设备驱动模型和标准字符设备驱动模型比较

1)、 杂项设备驱动可以自动创建设备节点，标准字符设备驱动不能自动创建节点；

2)、 杂项设备驱动和经典字符设备驱动模型次设备号范围是 0-255；
Linux2.6 的设备驱动模型次设备号范围 0-(1M-1)；

3)、 杂项设备驱动主设备号为 10；
标准字符设备驱动模型主设备号可以指定或者由系统分配；

4)、 标准设备驱动模型想要实现自动创建节点，需要创建调用创建设备类（class_create）和创建节点(device_create)的函数；

5)、 杂项设备驱动模型是对标准字符设备驱动模型的封装；