国嵌

www.enjoylinux.cn

LINUX
字符设备驱动程序

Contents

Contents

Linux驱动程序学习

www.enjoylinux.cn

知识结构:

1. Linux驱动程序设计模式(40%)
2. 内核相关知识(30%)
3. 硬件相关知识(30%)

学习方法:

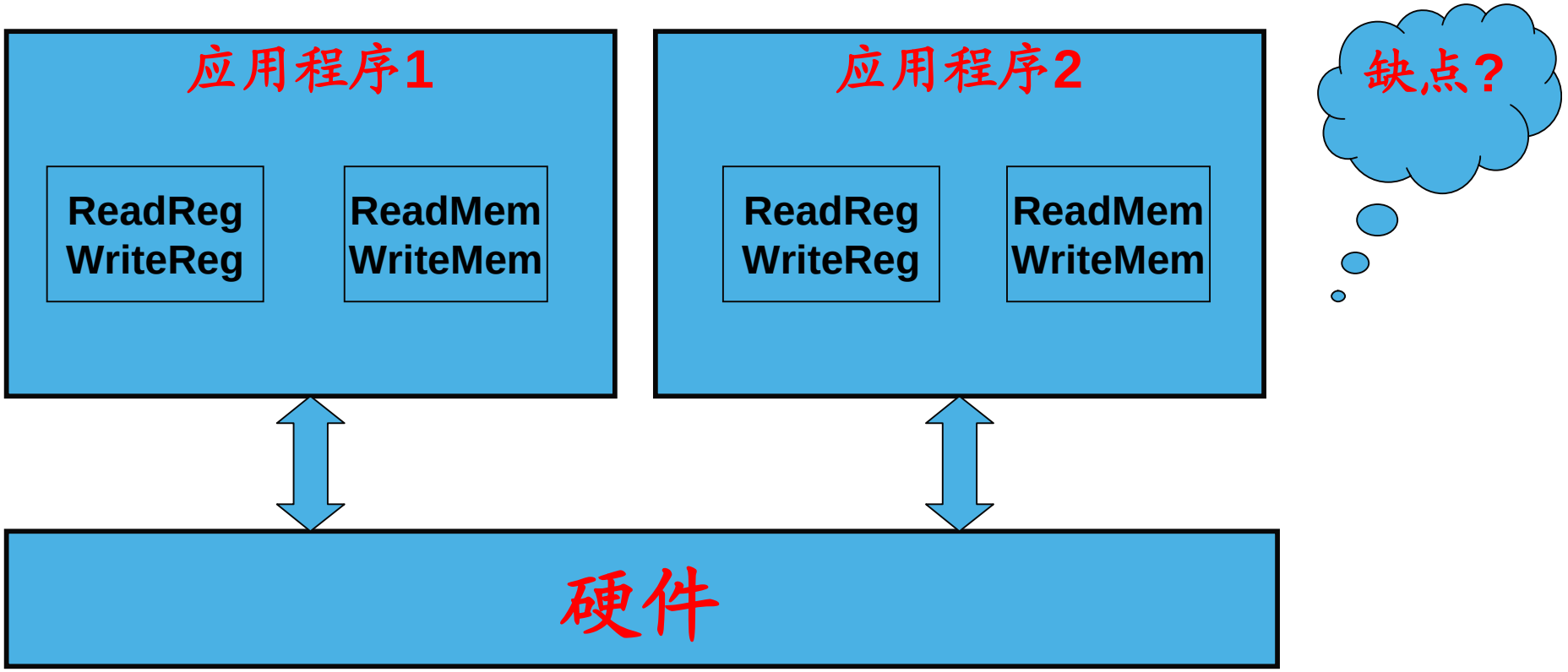
理论-->实验(疑问)-->理论-->实验-->…

定义

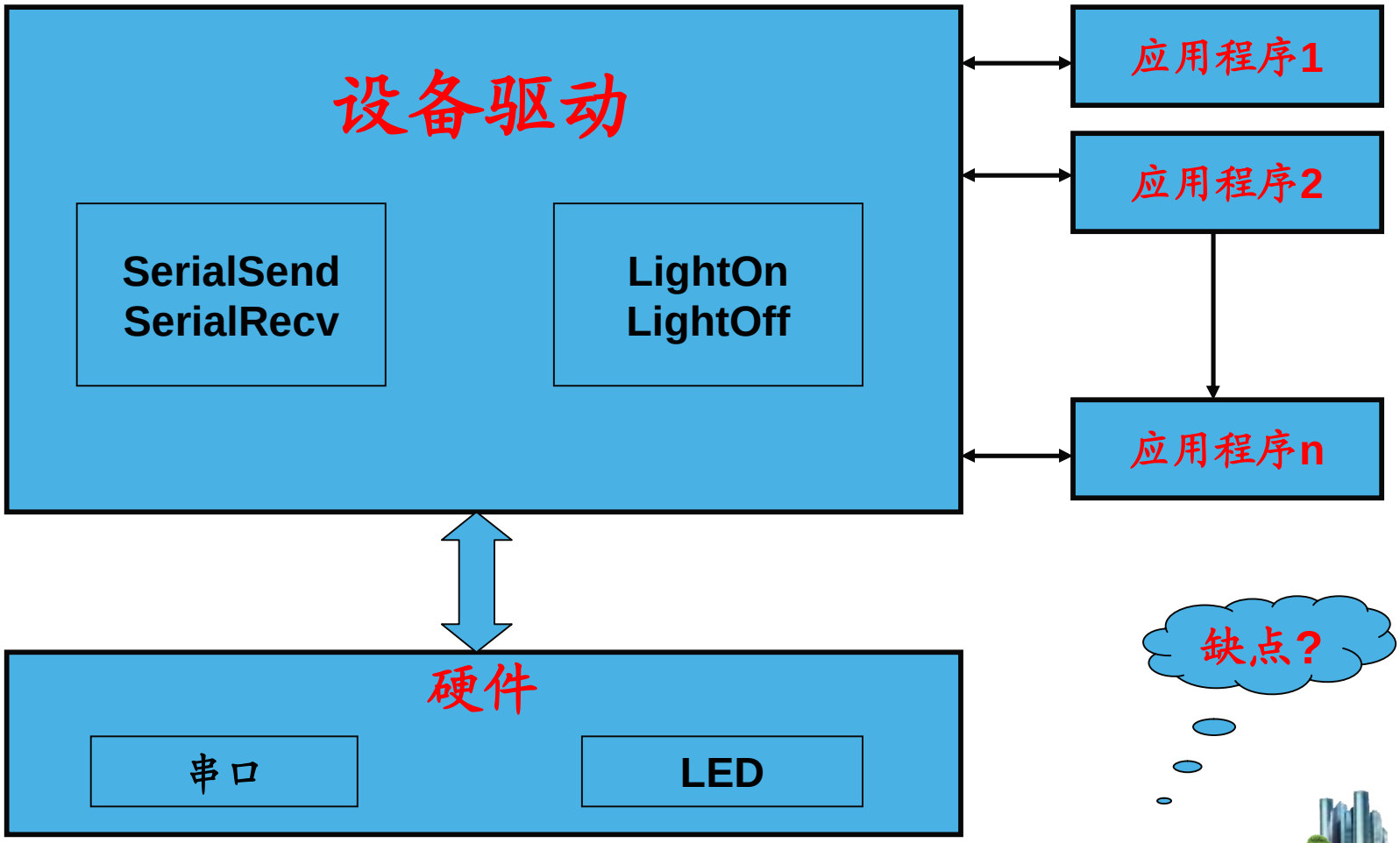
驱动程序？

使硬件工作的软件

早期驱动-模式一



早期驱动-模式二



驱动分类

- 字符设备驱动 （重点）
- 网络接口驱动（重点）
- 块设备驱动

字符设备

字符设备是一种按字节来访问的设备，字符驱动则负责驱动字符设备，这样的驱动通常实现 **open, close, read和 write** 系统调用。

块设备

- 在大部分的 Unix 系统, 块设备不能按字节处理数据，只能一次传送一个或多个长度是512字节(或一个更大的 2 次幂的数)的整块数据。

- 而Linux则允许块设备传送任意数目的字节。因此, 块和字符设备的区别仅仅是驱动的与内核的接口不同。

网络接口

任何网络事务都通过一个接口来进行，一个接口通常是一个硬件设备(eth0)，但是它也可以是一个纯粹的软件设备，比如回环接口（lo）。一个网络接口负责发送和接收数据报文。

驱动程序安装

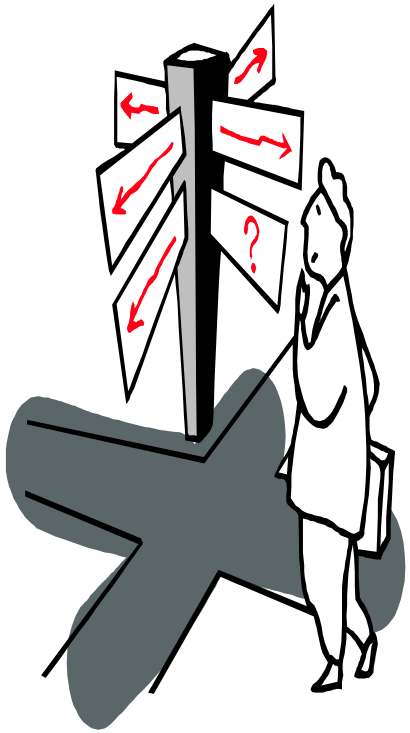
- 模块方式(已知 J)
- 直接编译进内核

直接编译进内核

- Kconfig ?
- Makefile ?

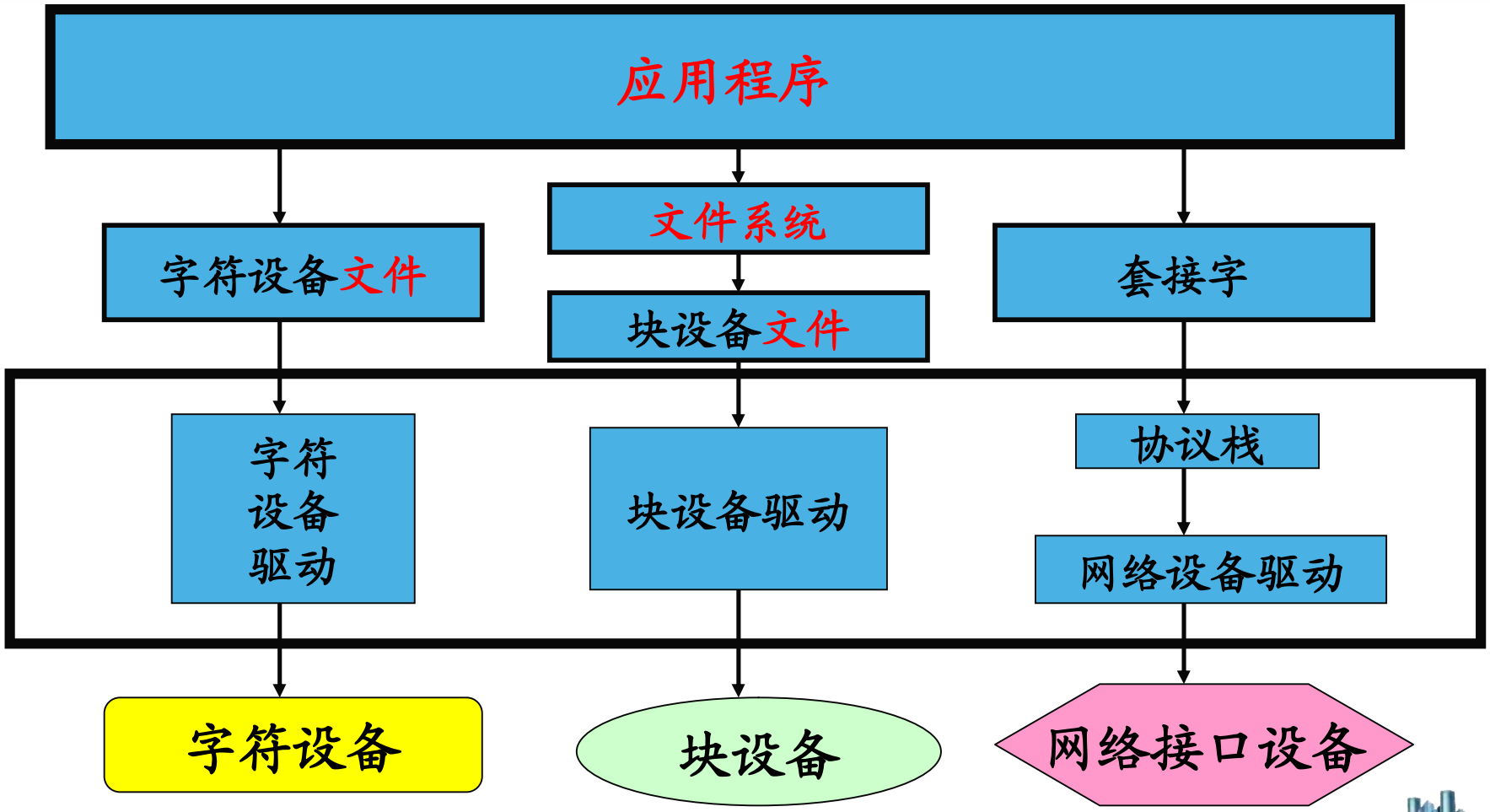
例：将helloWorld编译进内核

驱动程序使用



Linux用户如何使用驱动程序？

使用驱动程序



驱动程序使用

A: Linux用户程序通过设备文件（又名：设备节点）来使用驱动程序操作字符设备和块设备

Q: 设备（字符、块）文件在何处？

Contents

知识点

- 设备号
- 创建设备文件
- 设备注册
- 重要数据结构
- 设备操作

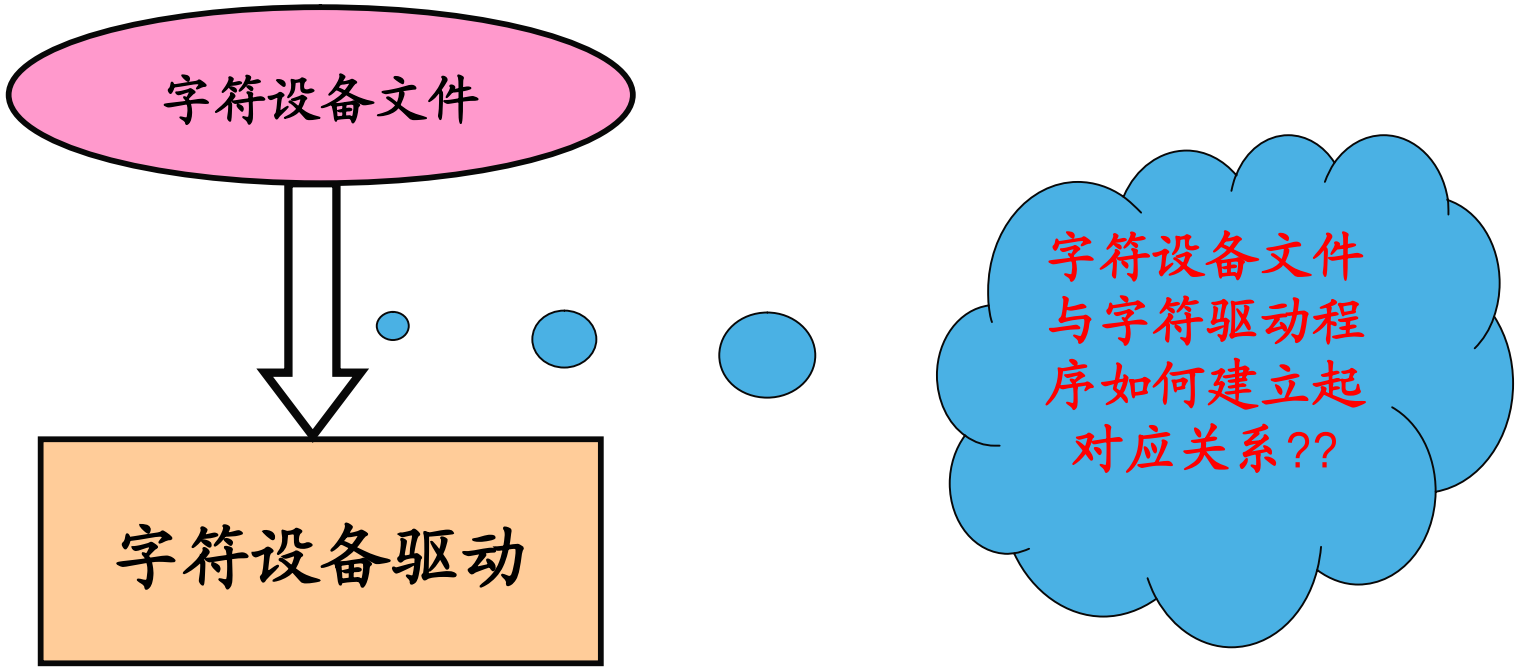
主次设备号

字符设备通过字符设备文件来存取。字符设备文件由使用 ls -l 的输出的第一列的“c”标识。如果使用 ls -l 命令, 会看到在设备文件项中有 2 个数(由一个逗号分隔) 这些数字就是设备文件的主次设备编号。（举例察看/dev）

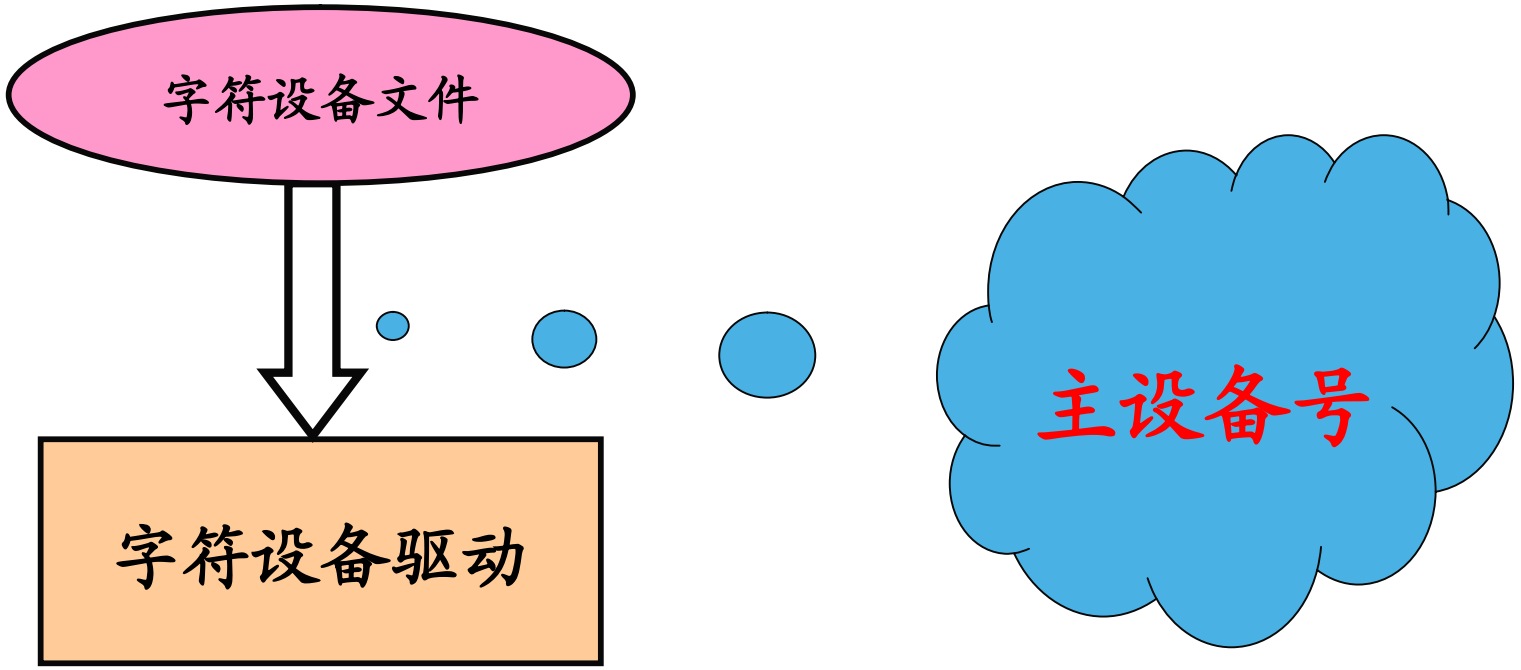
设备号

设备号用来做什么??

设备号



主设备号



设备号作用

- 主设备号用来标识与设备文件相连的驱动程序。次编号被驱动程序用来辨别操作的是哪个设备。

** 主设备号用来反映设备类型 **

次设备号用来区分同类型的设备

主次设备号

Q: 内核中如何描述设备号?

A: dev_t

**其实质为unsigned int 32位整数，其中高12位为主设备号，低20位为次设备号。

Q: 如何从dev_t中分解出主设备号?

A: MAJOR(dev_t dev)

Q: 如何从dev_t中分解出次设备号?

A: MINOR(dev_t dev)

分配主设备号

Linux内核如何给设备分配主设备号?

可以采用静态申请，动态分配两种方法

静态申请

- 方法：

1、根据Documentation/devices.txt，确定一个没有使用的主设备号

2、使用 register_chrdev_region 函数注册设备号

- 优点：

简单

- 缺点：

一旦驱动被广泛使用，这个随机选定的主设备号可能会导致设备号冲突，而使驱动程序无法注册。

静态申请


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

功能:

申请使用从 from 开始的 count 个设备号(主设备号不变，次设备号增加)

参数:

from: 希望申请使用的设备号

count: 希望申请使用设备号数目

name: 设备名(体现在/proc/devices)

动态分配

- 方法:

  使用 alloc_chrdev_region 分配设备号

- 优点:

  简单，易于驱动推广

- 缺点:

  无法在安装驱动前创建设备文件（因为安装前还没有分配到主设备号）。

- 解决办法:

  安装驱动后，从 /proc/devices 中查询设备号

动态分配

www.enjoylinux.cn

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

功能:

请求内核动态分配 count 个设备号，且次设备号从baseminor开始。

参数:

dev: 分配到的设备号

baseminor: 起始次设备号

count: 需要分配的设备号数目

name: 设备名(体现在/proc/devices)

注销设备号

不论使用何种方法分配设备号，都应该在不再使用它们时释放这些设备号。

```
void unregister_chrdev_region(dev_t from,
    unsigned count)
```

功能:

释放从from开始的count个设备号

创建设备文件

2种方法:

1. 使用mknod 命令手工创建

2. 自动创建

手工创建

mknod 用法:

mknod filename type major minor

filename:设备文件名

type: 设备文件类型

major: 主设备号

minor: 次设备号

例： mknod serial0 c 100 0

自动创建

www.enjoylinux.cn

后面课程介绍 J

重要结构

在Linux字符设备驱动程序设计中，有3种非常重要的数据结构:

Struct file

Struct inode

Struct file_operations

Struct File

代表一个打开的文件。系统中每个打开的文件在内核空间都有一个关联的 **struct file**。它由内核在打开文件时创建，在文件关闭后释放。

- 重要成员：

loff_t f_pos /*文件读写位置*/

struct file_operations *f_op

Struct Inode

用来记录文件的物理上的信息。因此，它和代表打开文件的file结构是不同的。一个文件可以对应多个file结构，但只有一个inode 结构。

- 重要成员：

dev_t i_rdev: 设备号

Struct file_operations

一个函数指针的集合，定义能在设备上进行的操作。结构中的成员指向驱动中的函数,这些函数实现一个特别的操作,对于不支持的操作保留为NULL。

例： mem_fops


```
struct file_operations mem_fops = {
.owner = THIS_MODULE,
.llseek = mem_seek,
.read = mem_read,
.write = mem_write,
.ioctl = mem_ioctl,
.open = mem_open,
.release = mem_release,
};
```


应用-驱动模型

内核代码导读

应用程序如何访问驱动程序？

（Read_write.c ）

设备注册

在linux 2.6内核中，字符设备使用 **struct cdev** 来描述。

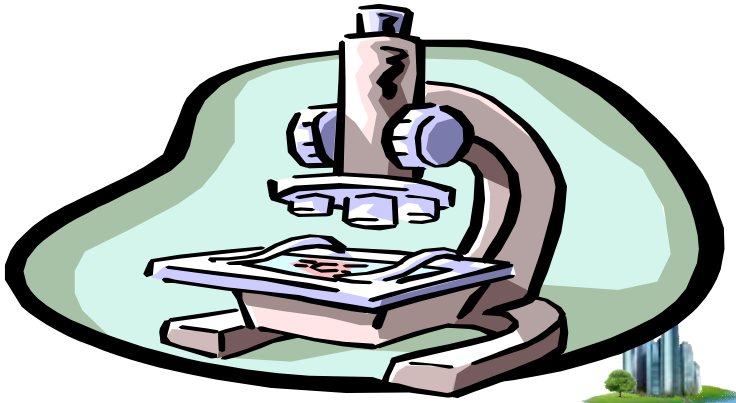
字符设备的注册可分为如下3个步骤:

1. 分配cdev
2. 初始化cdev
3. 添加cdev

设备注册（分配）

Struct cdev的分配可使用cdev_alloc函数来完成。

```
struct cdev *cdev_alloc(void)
```



设备注册（初始化）

Struct cdev的初始化使用cdev_init函数来完成。

```
void cdev_init(struct cdev *cdev, const
    struct file_operations *fops)
```

参数:

cdev: 待初始化的cdev结构

fops: 设备对应的操作函数集

设备注册（添加）

struct cdev的注册使用cdev_add函数来完成。

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

参数:

p: 待添加到内核的字符设备结构

dev: 设备号

count: 添加的设备个数

设备操作实现

完成了驱动程序的
注册，下一步该做什么呢？？
实现设备所支持的操作

设备操作

- int (*open)(struct inode *, struct file *)

  在设备文件上的第一个操作，并不要求驱动程序一定要实现这个方法。如果该项为NULL，设备的打开操作永远成功。

- void (*release)(struct inode *, struct file *)

  当设备文件被关闭时调用这个操作。与open相仿，release也可以没有。

设备操作

- ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)

  从设备中读取数据。

- ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)

  向设备发送数据。

- unsigned int (*poll) (struct file *, struct poll_table_struct *)

  对应select系统调用

- int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long)

  控制设备

设备操作

- int (*mmap) (struct file *, struct vm_area_struct *)

  将设备映射到进程虚拟地址空间中。

- off_t (*llseek) (struct file *, loff_t, int)

  修改文件的当前读写位置，并将新位置作为返回值。

Open方法

Open方法是驱动程序用来为以后的操作完成初始化准备工作的。在大部分驱动程序中，open完成如下工作:

- 初始化设备。
- 标明次设备号。

Release方法

Release方法的作用正好与open相反。这个设备方法有时也称为close，它应该：

- 关闭设备。

读和写

读和写方法都完成类似的工作：从设备中读取数据到用户空间；将数据传递给驱动程序。它们的原型也相当相似：

```
ssize_t xxx_read(struct file * filp, char __user * buff, size_t count, loff_t *
    offp);
```

```
ssize_t xxx_write(struct file *filp, char __user * buff, size_t count, loff_t
    *offp);
```

对于 2 个方法，filp是文件指针，count是请求传输的数据量。buff 参数指向数据缓存。最后，offp 指出文件当前的访问位置。

读和写

Read 和 Write 方法的 buff 参数是用户空间指针。因此，它不能被内核代码直接引用，理由如下：

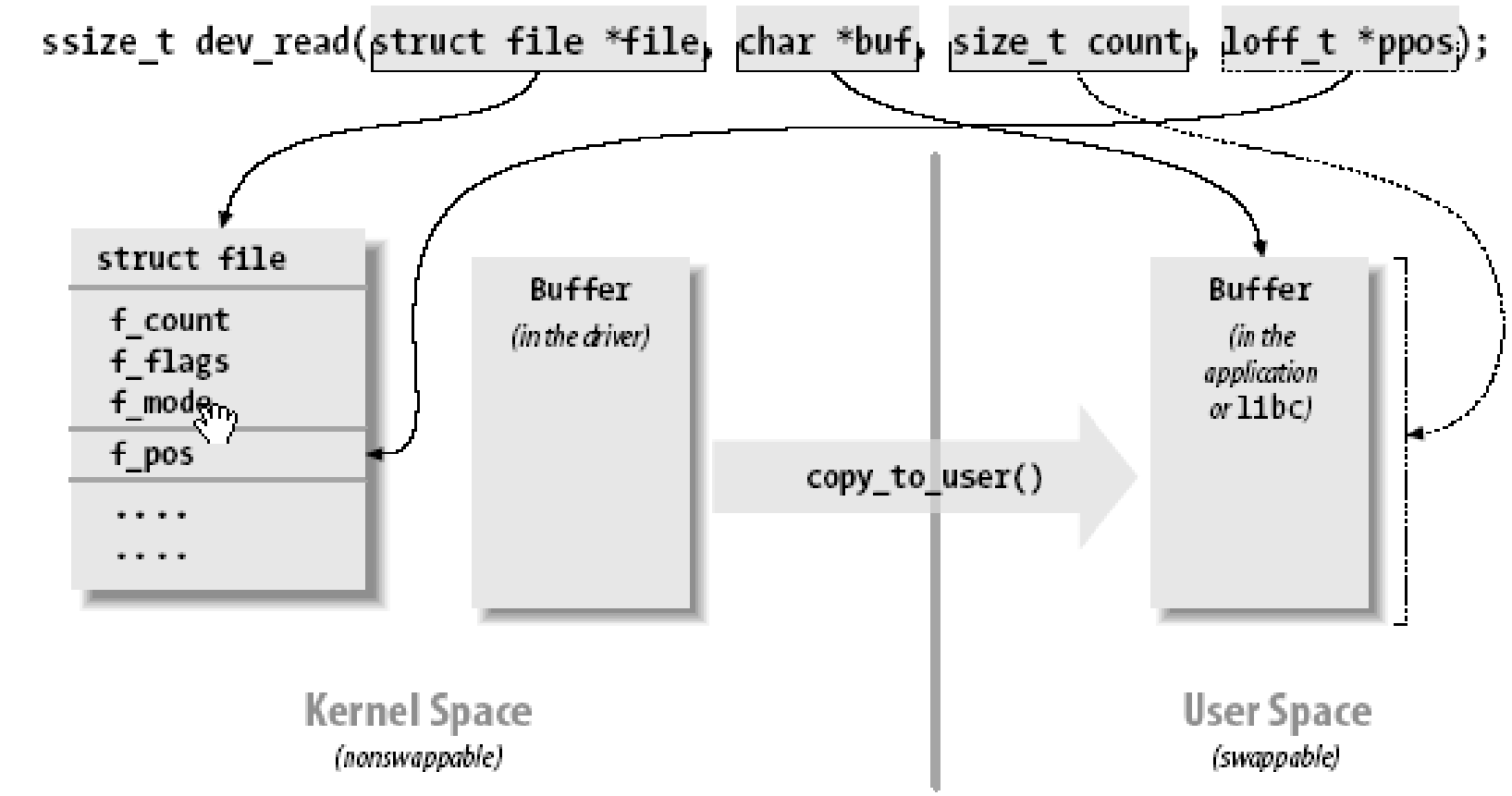
用户空间指针在内核空间时可能根本是无效的---没有那个地址的映射。

读和写

内核提供了专门的函数用于访问用户空间的指针，例如:

- int copy_from_user(void *to, const void __user *from, int n)
- int copy_to_user(void __user *to, const void *from, int n)

数据模型-读



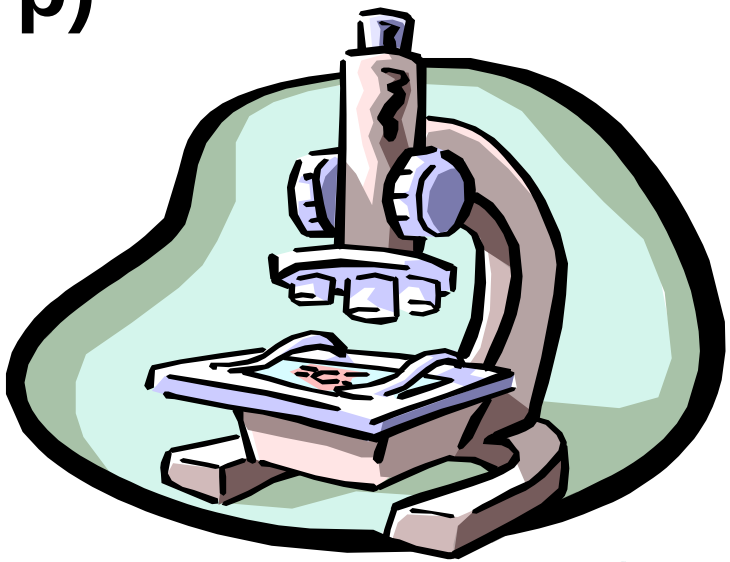
设备注消

字符设备的注销使用cdev_del函数来完成。

int cdev_del(struct cdev *p)

参数:

p: 要注销的字符设备结构



实例分析

字符设备驱动程序

memdev.c

实验

www.enjoylinux.cn

编写简单字符设备驱动程序

1、编写驱动，实现读、写、定位

2、编写应用程序，访问设备

（在mini2440平台实现）

Contents

调试技术分类

对于驱动程序设计来说，核心问题之一就是如何完成调试。当前常用的驱动调试技术可分为：

- 打印调试
- 调试器调试
- 查询调试

打印调试

在调试应用程序时，最常用的调试技术是打印，就是在应用程序中合适的点调用printf。当调试内核代码的时候，可以用printk完成类似任务。

合理使用Printk

在驱动开发时，printk 非常有助于调试。但当正式发行驱动程序时，应当去掉这些打印语句。但你有可能很快又发现，你又需要在驱动程序中实现一个新功能(或者修复一个bug)，这时你又要用到那些被删除的打印语句。这里介绍一种使用 printk 的合理方法,可以全局地打开或关闭它们，而不是简单地删除。

合理使用Printk


```
#ifdef PDEBUG
#define PLOG(fmt,args...) printk(KERN_DEBUG
    "scull:"fmt,##args)
#else
#define PLOG(fmt,args...)      /*do nothing */
#endif
```

合理使用Printk

Makefile作如下修改:

```
DEBUG =y
ifeq ($(DEBUG),y)
DEBFLAGS =-O2 -g -DPDEBUG
else
DEBFLAGS =-O2
endif
CFLAGS +=$(DEBFLAGS)
```

Contents

并发与竞态

- 并发：多个执行单元同时被执行。
- 竞态：并发的执行单元对共享资源（硬件资源和软件上的全局变量等）的访问导致的竞争状态

并发与竞态

例：

```
if (copy_from_user(&(dev->data[pos]), buf, count))
    ret = -EFAULT;
    goto out;
```

假设有 2 个进程试图同时向一个设备的相同位置写入数据，就会造成数据混乱。

并发与竞态

处理并发的常用技术是加锁或者互斥，即确保在任何时间只有一个执行单元可以操作共享资源。在Linux内核中主要通过semaphore机制和spin_lock机制实现。

信号量

Linux内核的信号量在概念和原理上与用户态的信号量是一样的，但是它不能在内核之外使用，它是一种睡眠锁。如果有一个任务想要获得已经被占用的信号量时，信号量会将这个进程放入一个等待队列，然后让其睡眠。当持有信号量的进程将信号释放后，处于等待队列中的任务将被唤醒，并让其获得信号量。

信号量

- 信号量在创建时需要设置一个初始值，表示允许有几个任务同时访问该信号量保护的共享资源，初始值为1就变成互斥锁（Mutex），即同时只能有一个任务可以访问信号量保护的共享资源。

- 当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把信号量的值加1实现，如果释放后信号量的值为非正数，表明有任务等待当前信号量，因此要唤醒等待该信号量的任务。

信号量

信号量的实现也是与体系结构相关的，定义在<asm/semaphore.h>中，struct semaphore类型用来表示信号量。

1. 定义信号量

```
struct semaphore sem;
```

2. 初始化信号量

```
void sema_init (struct semaphore *sem, int val)
```

该函用于数初始化设置信号量的初值，它设置信号量sem的值为val。

信号量

void init_MUTEX (struct semaphore *sem)

该函数用于初始化一个互斥锁，即它把信号量sem的值设置为1。

void init_MUTEX_LOCKED (struct semaphore *sem)

该函数也用于初始化一个互斥锁，但它把信号量sem的值设置为0，即一开始就处在已锁状态。

信号量

定义与初始化的工作可由如下宏一步完成:

DECLARE_MUTEX(name)

定义一个信号量**name**，并初始化它的值为**1**。

DECLARE_MUTEX_LOCKED(name)

定义一个信号量**name**，但把它的初始值设置为**0**，即锁在创建时就处在已锁状态。

信号量

3. 获取信号量

void down(struct semaphore * sem)

获取信号量sem，可能会导致进程睡眠，因此不能在中断上下文使用该函数。该函数将把sem的值减1，如果信号量sem的值非负，就直接返回，否则调用者将被挂起，直到别的任务释放该信号量才能继续运行。

信号量

- int down_interruptible(struct semaphore * sem)

  获取信号量sem。如果信号量不可用，进程将被置为TASK_INTERRUPTIBLE类型的睡眠状态。该函数由返回值来区分是正常返回还是被信号中断返回，如果返回0，表示获得信号量正常返回，如果被信号打断，返回-EINTR。

信号量

- down_killable(struct semaphore *sem)

  获取信号量sem。如果信号量不可用，进程将被置为TASK_KILLABLE类型的睡眠状态。

注：

down()函数现已不建议继续使用。建议使用down_killable() 或 down_interruptible() 函数。

信号量

4. 释放信号量

void up(struct semaphore * sem)

该函数释放信号量sem，即把sem的值加1，如果sem的值为非正数，表明有任务等待该信号量，因此唤醒这些等待者。

自旋锁

自旋锁最多只能被一个可执行单元持有。自旋锁不会引起调用者睡眠，如果一个执行线程试图获得一个已经被持有的自旋锁，那么线程就会一直进行忙循环，一直等待下去，在那里看是否该自旋锁的保持者已经释放了锁，“自旋”就是这个意思。

自旋锁

- **spin_lock_init(x)**

  该宏用于初始化自旋锁x，自旋锁在使用前必须先初始化。

- **spin_lock(lock)**

  获取自旋锁lock，如果成功，立即获得锁，并马上返回，否则它将一直自旋在那里，直到该自旋锁的保持者释放。

自旋锁

- **spin_trylock(lock)**

  试图获取自旋锁lock，如果能立即获得锁，并返回真，否则立即返回假。它不会一直等待被释放。

- **spin_unlock(lock)**

  释放自旋锁lock，它与spin_trylock或spin_lock配对使用。

信号量PK自旋锁

- 信号量可能允许有多个持有者，而自旋锁在任何时候只能允许一个持有者。当然也有信号量叫互斥信号量(只能一个持有者)，允许有多个持有者的信号量叫计数信号量。

- 信号量适合于保持时间较长的情况；而自旋锁适合于保持时间非常短的情况，在实际应用中自旋锁控制的代码只有几行，而持有自旋锁的时间也一般不会超过两次上下文切换的时间，因为线程一旦要进行切换，就至少花费切出切入两次，自旋锁的占用时间如果远远长于两次上下文切换，我们就应该选择信号量。

实验

www.enjoylinux.cn

编写带并发控制的字符设备驱动程序

1、编写驱动

2、编写应用程序，访问设备

（在mini2440平台实现）