中国人民大学 信息学院
RENMIN UNIVERSITY OF CHINA SCHOOL OF INFORMATION

操作系统内核分析与安全

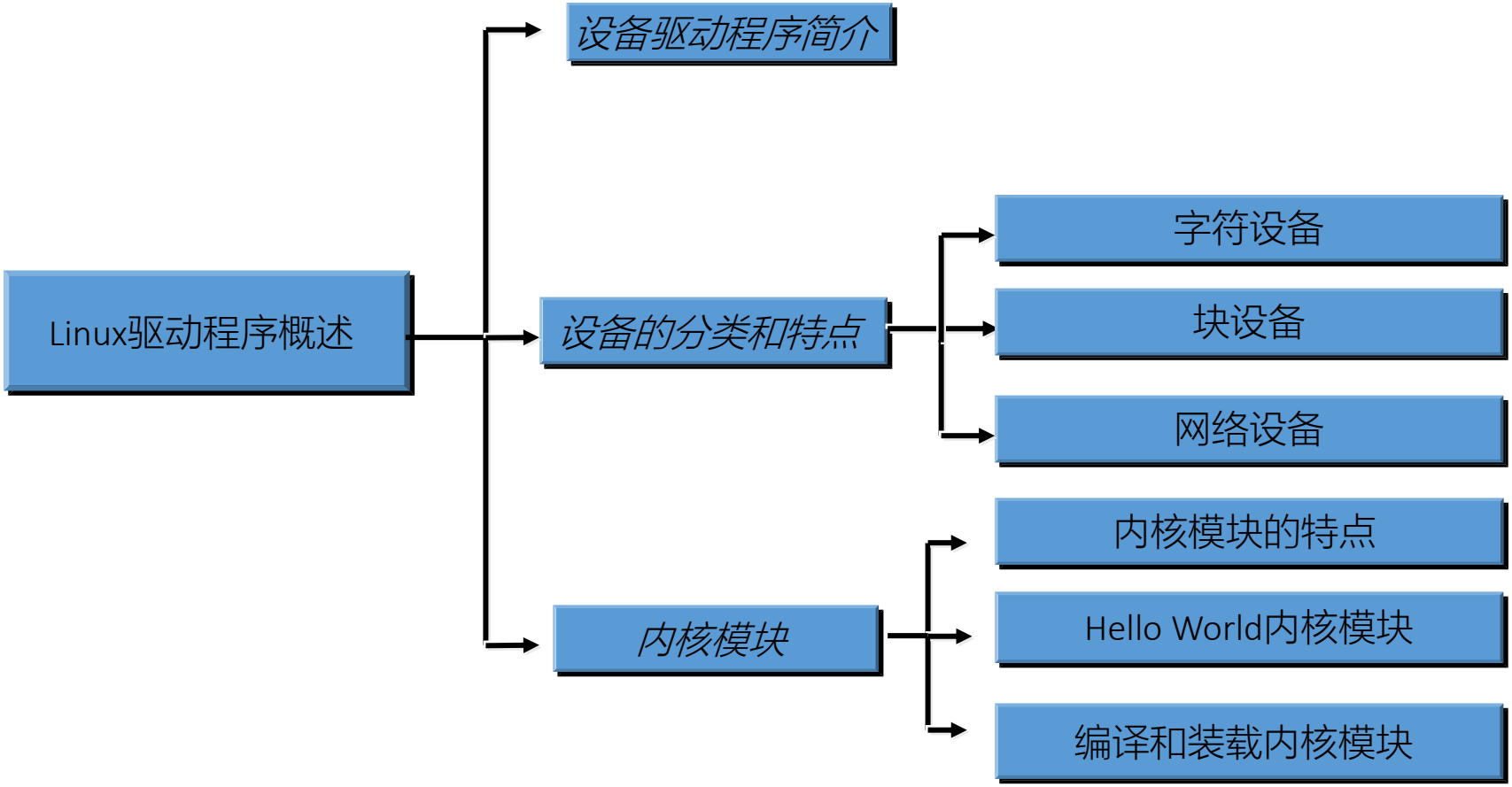
9. 内核模块与设备驱动

授课教师：游伟 副教授

授课时间：周五14:00 – 16:30（立德楼807）

课程主页：https://www.youwei.site/course/kernel

本章内容


Linux驱动程序概述
设备驱动程序简介
设备的分类和特点
字符设备
块设备
网络设备
内核模块
内核模块的特点
Hello World内核模块
编译和装载内核模块

为什么要学习嵌入式Linux驱动程序开发？

- 高需求
  - 内核代码的大部分
  - 新芯片、新设备
  - 内核新版本
- 高门槛
  - 需要具有硬件知识
  - 了解操作系统的实现
  - 需要了解内核基础知识
  - 需要了解内核中的并发控制和同步
  - 复杂的软件结构框架
- 高回报

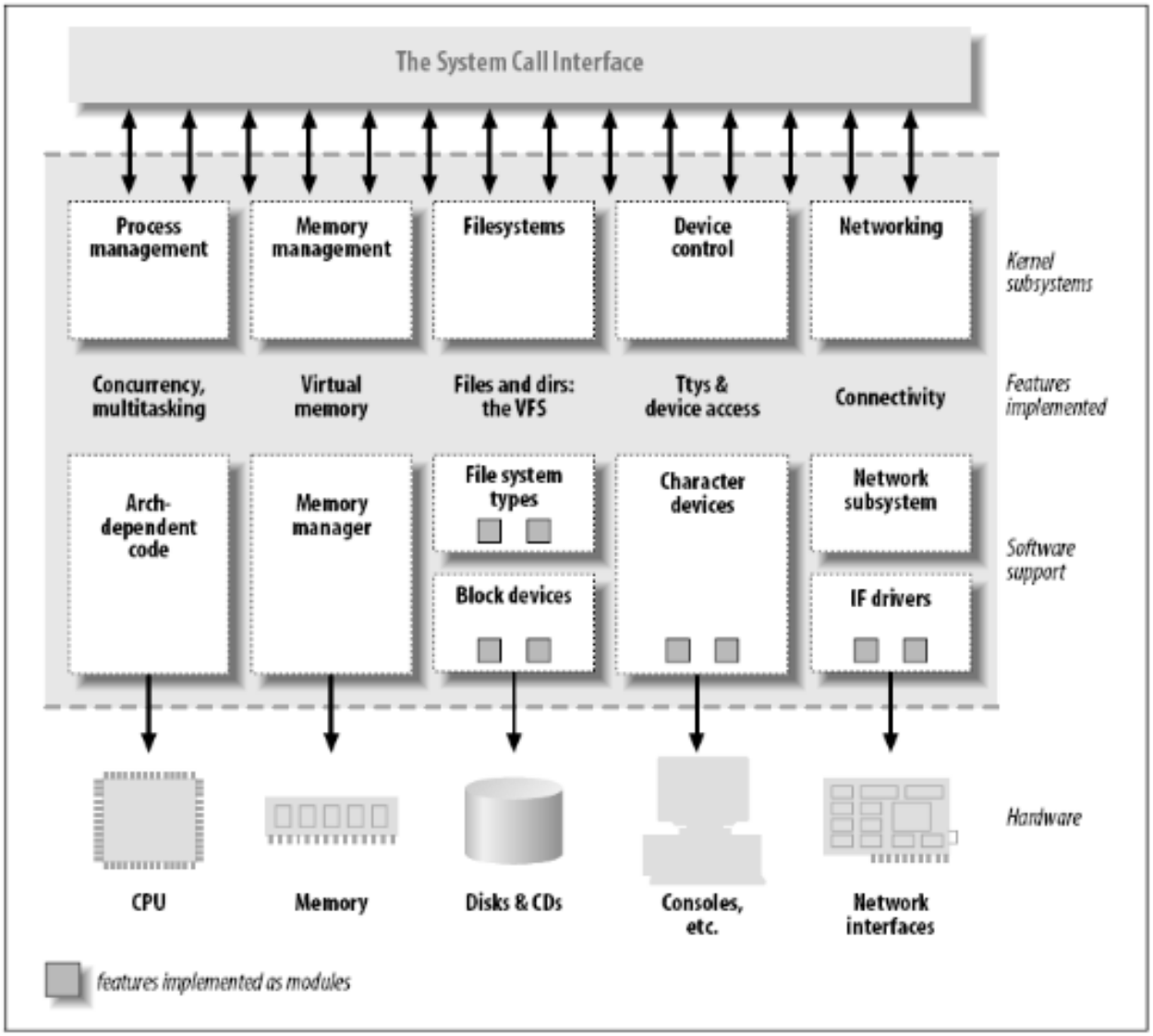
设备驱动程序简介

- **驱动程序的特点**
- 操控硬件，是应用程序和硬件设备之间的一个接口
  - 隐藏硬件细节，提高应用软件的可移植性
  - 提供安全性
  - 开发模式
    - 内核态驱动
    - 用户态驱动
- 提供机制，而不是提供策略
  - 机制：驱动程序能实现什么功能
  - 策略：用户如何使用这些功能

设备的分类和特点

- 设备分类
  - 字符设备(char device)
  - 块设备(block device)
  - 网络设备(network device)

设备的分类和特点


The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

设备的分类和特点

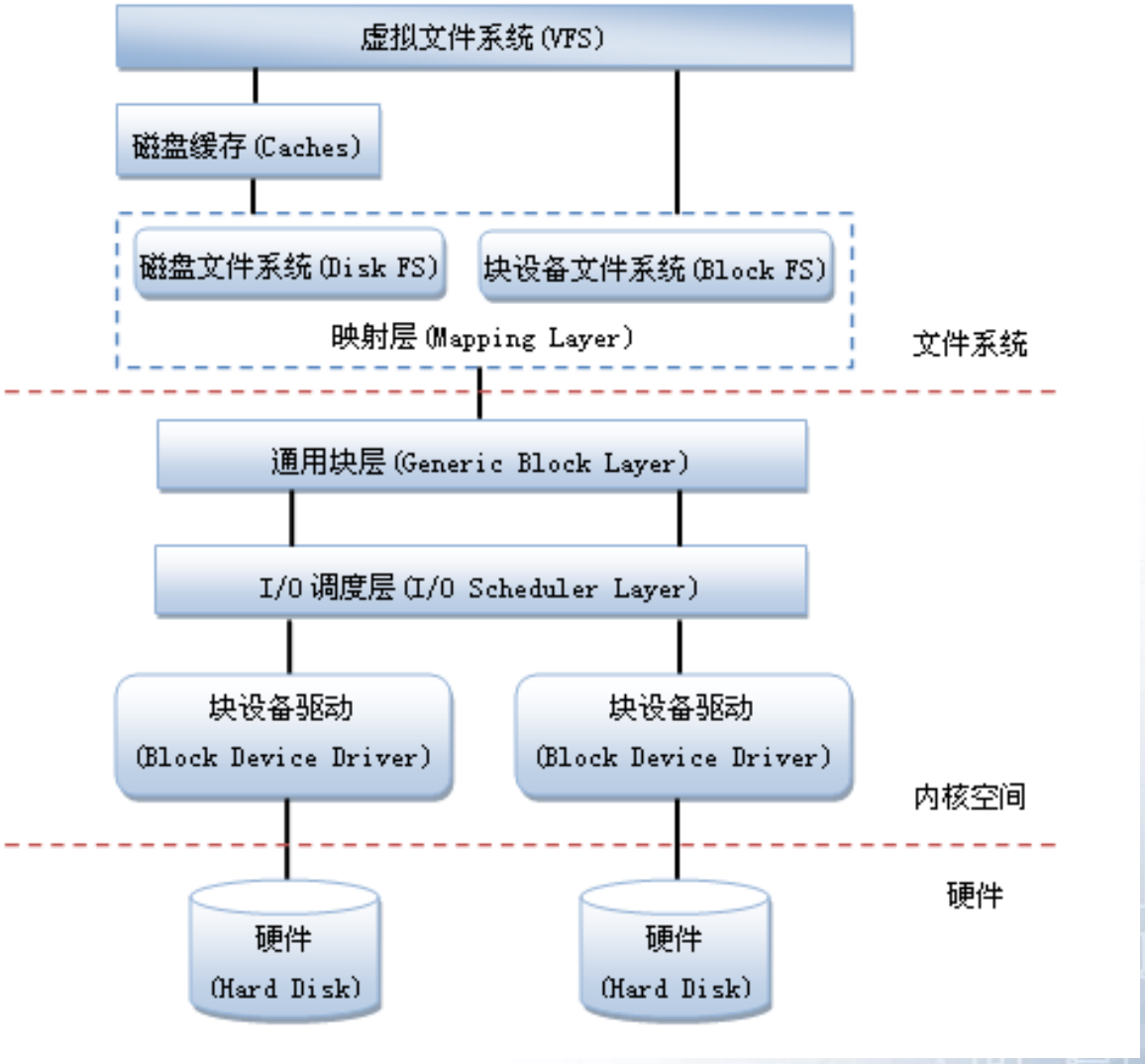
- **字符设备特点**
  - 像字节流一样来存取的设备(如同文件)
  - 通过/dev下的文件系统结点来访问。
  - 通常至少需要实现 open, close, read, 和 write 等系统调用
  - 只能顺序访问的数据通道，不能前后移动访问指针。
    - 特例:比如framebuffer设备就是这样的设备，应用程序可以使用mmap或lseek访问图像的各个区域

设备的分类和特点

块设备特点

- 块设备通过位于 /dev 目录的文件系统结点来存取
- 块设备和字符设备的区别仅仅在于内核内部管理数据的方式
- 块设备有专门的接口，块设备的接口必须支持挂装（mount）文件系统。
- 应用程序一般通过文件系统来访问块设备上的内容

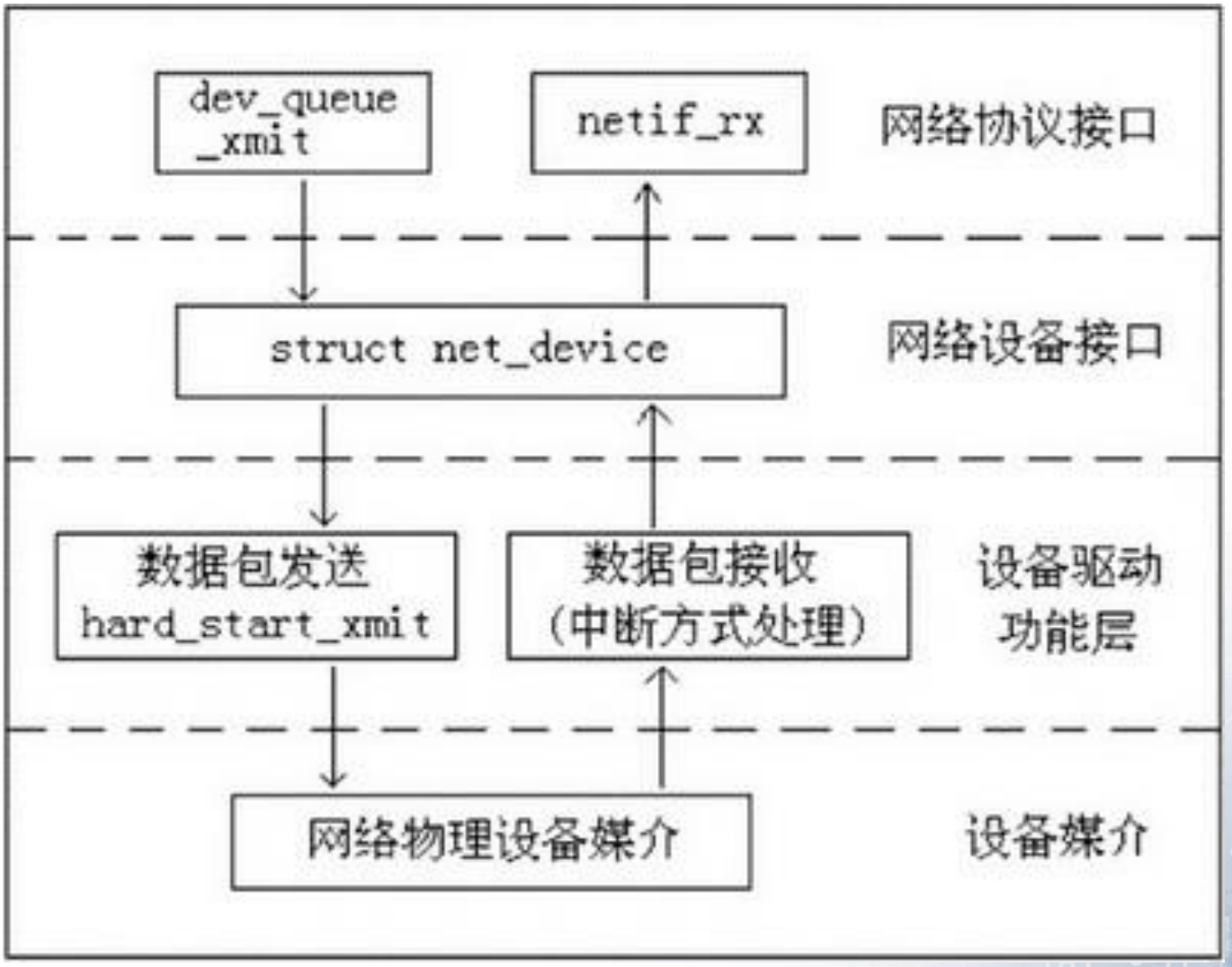
块设备


虚拟文件系统(VFS)
磁盘缓存(Caches)
磁盘文件系统(Disk FS)
块设备文件系统(Block FS)
映射层(Mapping Layer)
文件系统
通用块层(Generic Block Layer)
I/O调度层(I/O Scheduler Layer)
块设备驱动
(Block Device Driver)
块设备驱动
(Block Device Driver)
内核空间
硬件
硬件
(Hard Disk)
硬件
(Hard Disk)

网络设备

- 处理报文的发送和接收
- 上面是网络协议栈

网络设备驱动程序


dev_queue
_xmit
netif_rx
网络协议接口
struct net_device
网络设备接口
数据包发送
hard_start_xmit
数据包接收
（中断方式处理）
设备驱动
功能层
网络物理设备媒介
设备媒介

构造和运行模块

- **驱动程序加入内核的方法**
  - **把所有需要的功能都编译到内核中**
    - 生成的内核镜像（Image）文件会很大
    - 如果我们要在现有的内核中新增或删除功能，将不得不重新编译和装载内核。

驱动程序加入内核的方法

- Linux提供了机制被称为模块（Module）的机制
  - 提供了对许多模块支持, 包括但不限于, 设备驱动
  - 每个模块由目标代码组成(没有连接成一个完整可执行程序)
    - insmod 将模块动态加载到正在运行内核
    - rmmod 程序移除模块

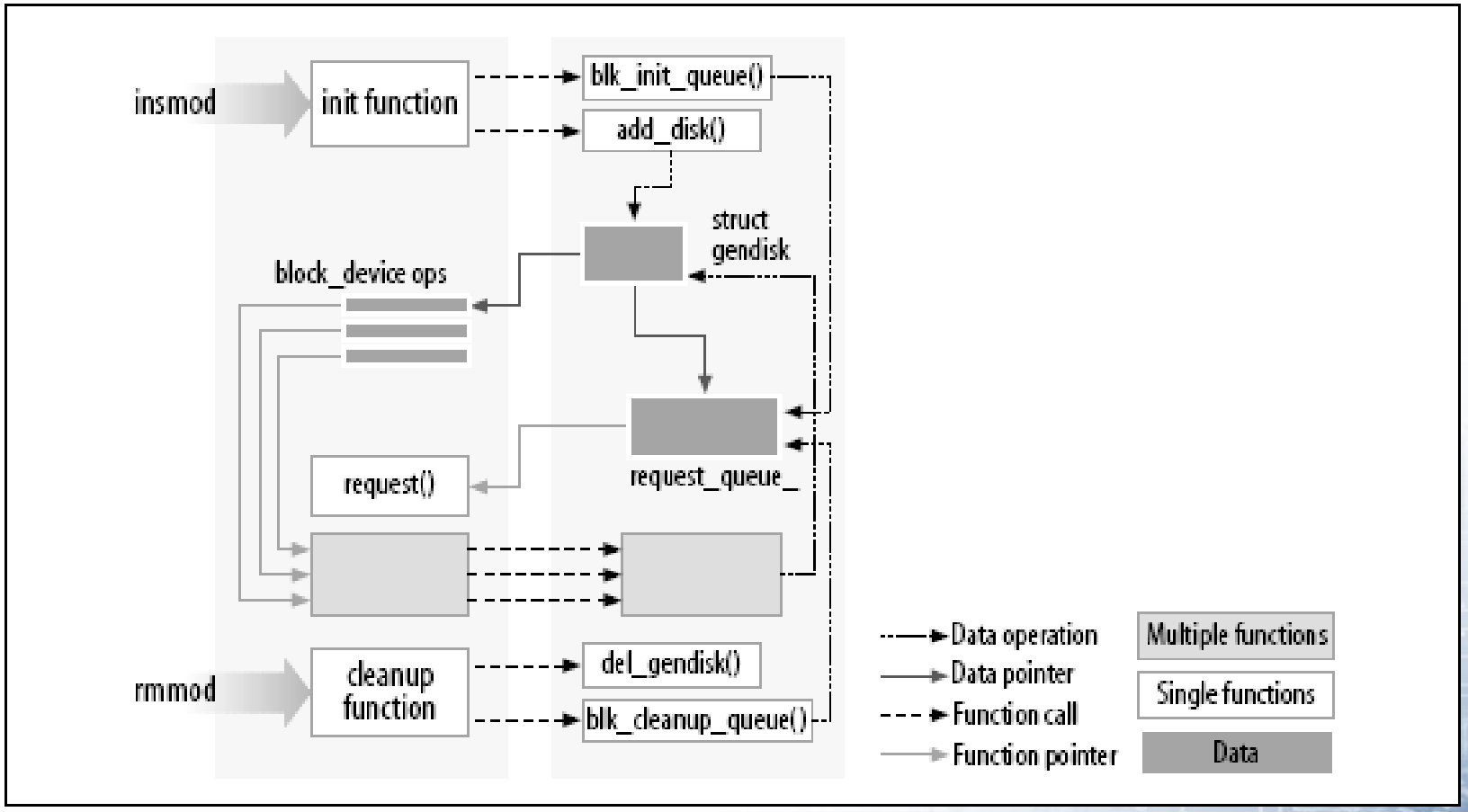
内核模块的定义

- **什么是内核模块？**
  - 具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间运行
- **内核模块的特点**
  - 需要提供入口和出口函数等等
  - 不能独立运行，这些函数由内核在需要的时候来调用
  - 使用Linux内核里定义的函数，不能使用glibc的库
  - 这些函数可以用来完成硬件访问等操作

内核模块与应用程序对比

- **应用程序是一个进程**
  - 编程从主函数main（）开始
  - 主函数main返回即是进程结束
  - 使用glibc的库
- **驱动程序是一系列内核函数**
  - 函数入口和出口不一样
  - 使用Linux内核的函数
  - 这些函数由内核在适当的时候来调用
  - 这些函数可以用来完成硬件访问等操作

内核模块是如何被调用的



内核模块必须的函数

- **Linux内核模块的程序结构**
  - **module_init()---模块加载函数**
    - 通过insmod或modprobe命令加载内核模块时，模块的加载函数会自动被内核执行，完成模块的相关初始化工作
  - **module_exit()---模块卸载函数**
    - 当通过rmmod命令卸载某模块时，模块的卸载函数会自动被内核执行，完成与模块装载函数相反的功能
  - **MODULE_LICENSE()---模块许可证声明**
    - 模块许可证（LICENSE）声明描述内核模块的许可权限
    - 如果不声明LICENSE,模块被加载时，将收到内核被污染（kernel tainted）的警告

动手写一个内核模块

- #include <linux/init.h>
- …

设备驱动的Hello World模块

- **模块加载函数**

```
static int __init initialization_function(void)
{
    /* 初始化代码 */
}
module_init(initialization_function);
```

应当声明成静态的（static），因为它们不会在特定文件之外可见

表明该函数只是在初始化时使用。模块加载器在模块加载后会丢掉这个初始化函数，这样可将与用的内存释放出来，以作他用。

原型：#define __init __attribute__ ((

定义会在模块目标代码中增加一个特殊的段，用于说明内核模块初始化函数所在的位置。没有这个定义，初始化函数不会被调用。

设备驱动的Hello World模块

- **模块卸载函数**

```
static void __exit cleanup_function(void)
{
    /* 释放资源 */
}
module_exit(cleanup_function);
```

在模块被移除前注销接口并释放所有所占用的系统资源

标识这个代码是只用于模块卸载(通过使编译器把它放在特殊的 ELF 段)

原型：#define __exit __attribute__ ((__section__(“.exit.text”)))

编译和装载驱动模块

编译模块

```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build PWD := $(shell pwd)
default:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

如果我们想由两个源文件(比如file1.c和file2.c)构造出一个名称为module.ko的模块，则正确的makefile可如下编写:

```
obj-m := module.o
module-objs := file1.o file2.o
```

Makefile

- 文件名字必须为Makefile
- KERNELRELEASE 是在内核源码的顶层Makefile中定义的一个变量，内核发行的版本号
- $(obj-m)表示对象文件（object files）编译成可加载的内核模块。

Makefile

- 第一次进入Makefile，KERNELRELEASE没有被定义
- 执行else后面的语句，给KERNEL，PWD赋值
- 执行default，编译
- Make -C选项进入内核源代码目录，找到顶层的Makefile
- M=PWD，返回当前目录执行Makefile文件
- 这就是第二次进入这个Makefile，在这次，由于KERNELRELEASE变量已经定义，因此不需要进入else语言，在这里，obj-m:=hello.o，在这里内核会帮你处理一切，这句话是告诉内核，需要从hello.o创建一个驱动模型（module）

装载驱动模块

装载模块

- Insmod和modprobe可以用来装载模块

Insmod和modprobe主要区别

modprobe会考虑要装载的模块是否引用了一些当前内核不存在的符号。如果有这类引用，modprobe会在当前模块路径中搜索定义了这些符号的其他模块，并同时将这些模块也装载到内核。如果在这种情况下使用insmod，该命令则会失败，并在系统日志文件中记录“unresolved symbols（未解析的符号）”消息。

查看已加载模块

- lsmod
- cat /proc/modules.

卸载驱动模块

卸载模块

- 从内核中卸载模块可以用rmmod工具.

> 注意，如果内核认为该模块任然在使用状态，或者内核被禁止移除该模块，则无法移除该模块。

内核打印函数

- printk（fmt, args …)
- 级别
  - KERN_EMERG 用于紧急消息, 常常是那些崩溃前的消息.
  - KERN_ALERT 需要立刻动作的情形.
  - KERN_CRIT 严重情况, 常常与严重的硬件或者软件失效有关.
  - KERN_ERR 用来报告错误情况; 设备驱动常常使用 KERN_ERR 来报告硬件故障.
  - KERN_WARNING 有问题的情况的警告, 这些情况自己不会引起系统的严重问题.
  - KERN_NOTICE 正常情况, 但是仍然值得注意. 在这个级别一些安全相关的情况会报告.
  - KERN_INFO 信息型消息. 在这个级别, 很多驱动在启动时打印它们发现的硬件的信息.
  - KERN_DEBUG 用作调试消息.
- 不能打印浮点数

运行内核模块

- 编译
- 上传模块，rz
- 加载模块 insmod
- 查看模块 lsmod、cat /proc/modules
- 卸载模块 rmmod

设备驱动的Hello World模块(hello.c)

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

MODULE_LICENSE("Dual BSD/GPL");

static int __init hello_init(void)
{
    printk(KERN_ALERT "Hello world\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_ALERT " Hello world exit\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

自由许可证

用法类似于printf，但它有优先级(比如KERN_ALERT)

宏，告诉内核这两个函数只会在加载和卸载模块时使用

模块初始化宏

模块卸载宏

内核模块参数

- **模块也可以拥有参数数组**
  - 形式为“module_param_array（数组名，数组类型，数组长，参数读/写权限）”。
  - 运行insmod或modprobe命令时，应使用逗号分隔输入的数组元素
- **装载模块时改变参数：**
  - **可通过insmod或modprobe**
  - insmod hello_ext.ko howmany=5 whom="Students"
  - modprobe也可以从它的配置文件(/etc/modprobe.conf)读取参数的值

内核模块参数

module_param（参数名，参数类型，参数读/写权限）

```
static char *whom = "world";
 static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

内核支持的模块参数类型包括：

byte、short、ushort、int、uint、long、ulong、charp(字符指针)、bool以‘u’开头的为无符号值。

导出符号

- **模块导出符号**
  - EXPORT_SYMBOL(name);
  - EXPORT_SYMBOL_GPL(name);

_GPL 版本的宏定义的导出符号只能对 GPL 许可的模块可用

符号必须在模块文件的全局部分导出, 不能在函数中导出

模块计数

- **如果模块在使用被卸载？怎么办**
- **模块的使用计数**
  - **Linux2.4内核**
    - MOD_INC_USE_COUNT（加一计数）
    - MOD_DEC_USE_COUNT（减一计数）
  - **Linux2.6内核中**
    - int try_module_get(struct module *module);
    - void module_put(struct module *module);

在Linux2.6内核下，对于设备驱动工程师而言，很少需要亲自调用try_module_get()和module_put()，因为模块的计数管理由内核里更底层的代码（如总线驱动或是此类设备共用的核心模块）来实现，从而简化了设备驱动的开发

补充信息

- **模块声明与描述**

MODULE_AUTHOR(author); ---声明模块的作者
MODULE_DESCRIPTION(description); ---声明模块的描述
MODULE_VERSION(version_string); ---声明模块的版本
MODULE_DEVICE_TABLE(table_info); ---声明模块的设备表
MODULE_ALIAS(alternate_name); ---声明模块的别名

带参数的hello world内核模块程序

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static char *whom = "world";
static int howmany = 1;
static int hello_init(void)
{
	int i;
	for(i=0;i<howmany;i++){
		printk(KERN_ALERT "Hello %s\n",whom);
	}
	return 0;
}
static void hello_exit(void)
{
	printk(KERN_ALERT " Hello world exit\n");
}
module_init(hello_init);
module_exit(hello_exit);
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

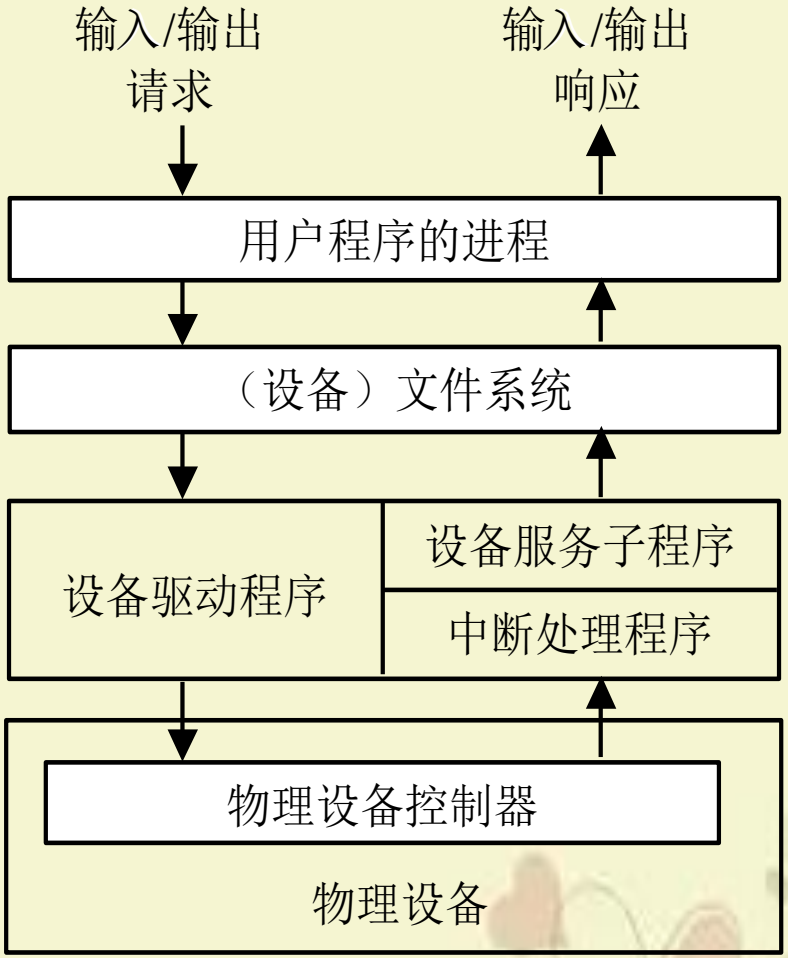
module_param（参数名，参数类型，参数读/写权限）

设备号

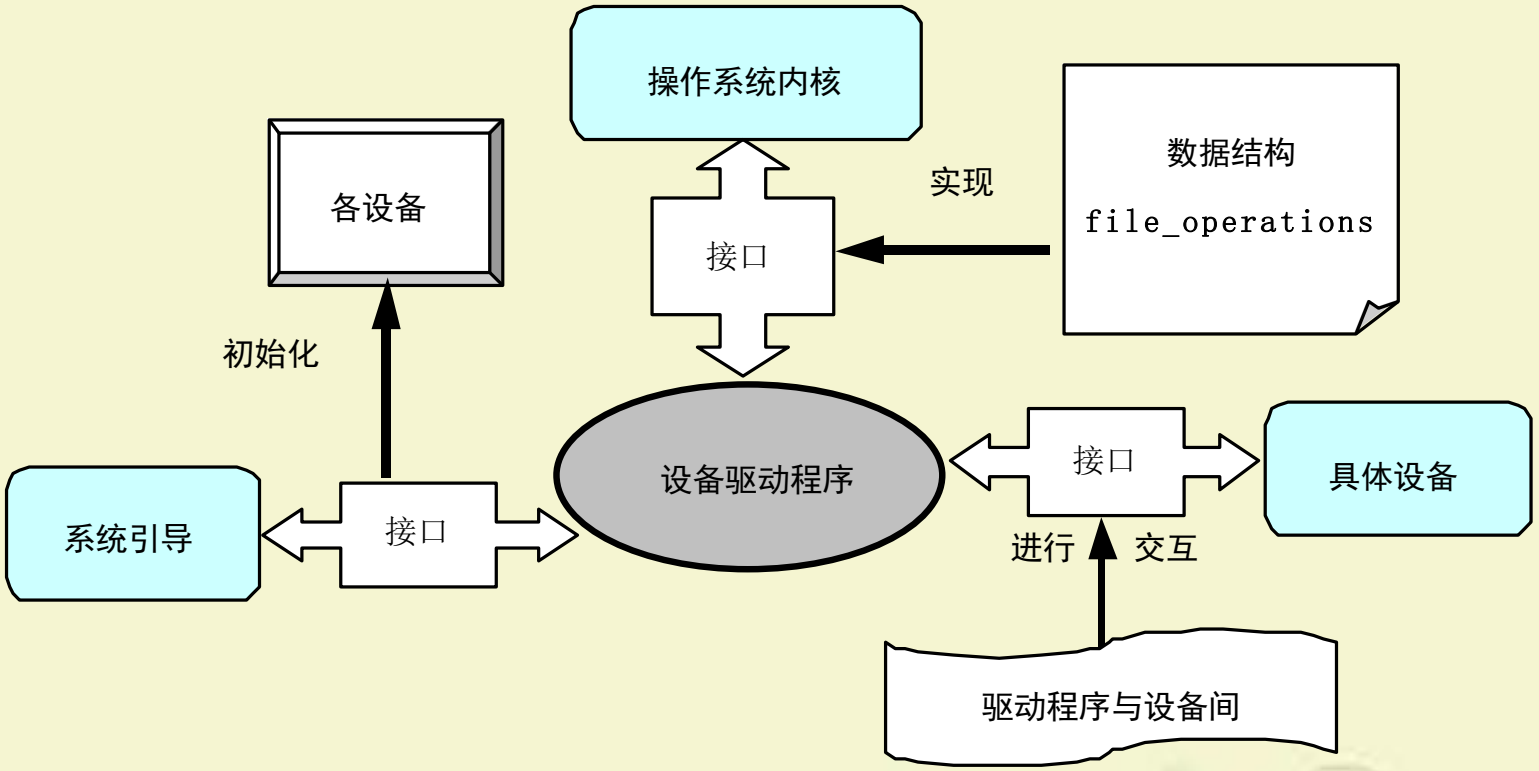
- 设备号是一个数字,它是设备的标志。就如前面所述,一个设备文件〔也就是设备节点〕可以通过mknod命令来创立,其中指定了主设备号和次设备号。主设备号说明设备的类型〔例如串口设备、SCSI硬盘〕,与一个确定的驱动程序对应;次设备号通常是用于标明不同的属性,例如不同的使用方法,不同的位置,不同的操作等,它标志着某个具体的物理设备。高字节为主设备号,底字节为次设备号。
- 例如,在系统中的块设备IDE硬盘的主设备号是3,而多个IDE硬盘及其各个分区分别赋予次设备号1、2、3……

```
$ ls –l /dev
crw-rw----  1 root  uucp 4,  64 08-30 22:58 ttyS0 /* 主设备号4,此设备号64 */
```

驱动层次结构


输入/输出
请求
输入/输出
响应
用户程序的进程
（设备）文件系统
设备驱动程序
设备服务子程序
中断处理程序
物理设备控制器
物理设备

设备驱动程序与外界的接口



设备驱动程序的特点〔1〕

〔1〕内核代码：设备驱动程序是内核的一局部,如果驱动程序出错,则可能导致系统崩溃。

〔2〕内核接口：设备驱动程序必须为内核或者其子系统提供一个标准接口。比方,一个终端驱动程序必须为内核提供一个文件I/O接口；一个SCSI设备驱动程序应该为SCSI子系统提供一个SCSI设备接口,同时SCSI子系统也必须为内核提供文件的I/O接口及缓冲区。

〔3〕内核机制和效劳：设备驱动程序使用一些标准的内核效劳,如内存分配等。

设备驱动程序的特点〔2〕

〔4〕可装载：大多数的Linux操作系统设备驱动程序都可以在需要时装载进内核,在不需要时从内核中卸载。

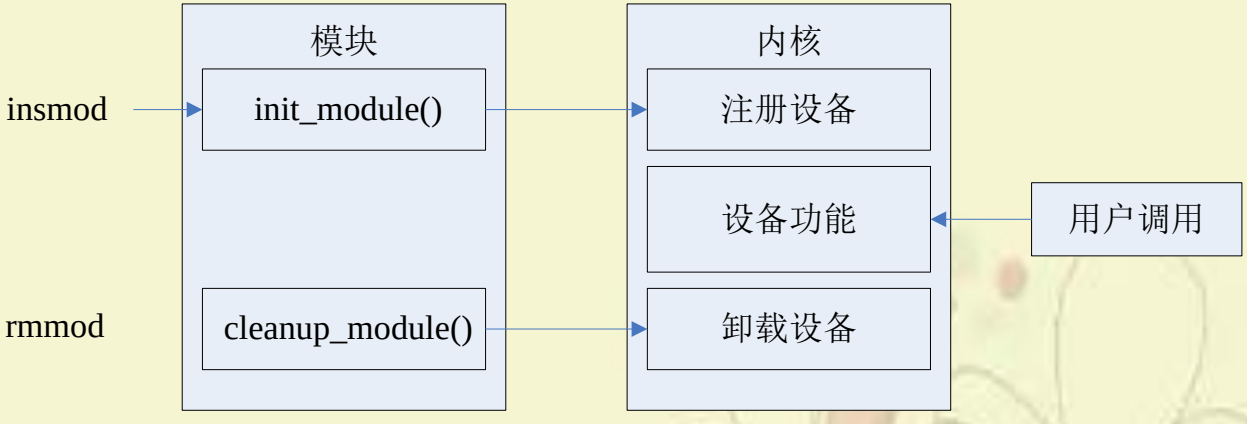
〔5〕可设置：Linux操作系统设备驱动程序可以集成为内核的一局部,并可以根据需要把其中的某一局部集成到内核中,这只需要在系统编译时进行相应的设置即可。

〔6〕动态性：在系统启动且各个设备驱动程序初始化后,驱动程序将维护其控制的设备。如果该设备驱动程序控制的设备不存在也不影响系统的运行,那么此时的设备驱动程序只是多占用了一点系统内存罢了。

字符设备驱动编程

设备驱动程序工作原理

模块在调用insmod命令时被加载,此时的入口点是init_module()函数,通常在该函数中完成设备的注册。同样,模块在调用rmmod命令时被卸载,此时的入口点是cleanup_module()函数,在该函数中完成设备的卸载。在设备完成注册加载之后,用户的应用程序就可以对该设备进行一定的操作,如open()、read()、write()等,而驱动程序就是用于实现这些操作,在用户应用程序调用相应入口函数时执行相关的操作。



重要数据结构- file_operaions结构

```
struct file_operations
{
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *filp, char *buff, size_t count, loff_t *offp);
    ssize_t (*write) (struct file *filp, const char *buff, size_t count, loff_t
*offp);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *);
    int (*fasync) (int, struct file *, int);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
    int (*lock) (struct file *, int, struct file_lock *);
};
```

重要数据结构- inode结构

```
struct file
{
    mode_t f_mode;/*标识文件是否可读或可写,FMODE_READ或FMODE_WRITE*/
    dev_t f_rdev; /* 用于/dev/tty */
    off_t f_pos; /* 当前文件位移 */
    unsigned short f_flags; /* 文件标志,如O_RDONLY、O_NONBLOCK和O_SYNC */
    unsigned short f_count; /* 翻开的文件数目 */
    unsigned short f_reada;
    struct inode *f_inode; /*指向inode的结构指针 */
    struct file_operations *f_op;/* 文件索引指针 */
};
```

早期版本的字符设备注册〔1〕

- 早期版本的设备注册使用函数register_chrdev(),调用该函数后就可以向系统申请主设备号,如果register_chrdev()操作成功,设备名就会出现在/proc/devices文件里。在关闭设备时,通常需要解除原先的设备注册,此时可使用函数unregister_chrdev(),此后该设备就会从/proc/devices里消失。其中主设备号和次设备号不能大于255。

所需头文件	#include <linux/fs.h>
函数原型	int register_chrdev(unsigned int major, const char *name,struct file_operations *fops)
函数传入值	major：设备驱动程序向系统申请的主设备号，如果为 0 则系统为此驱动程序动态地分配一个主设备号
	name：设备名
	fops：对各个调用的入口点
函数返回值	成功：如果是动态分配主设备号，此返回所分配的主设备号。且设备名就会出现在/proc/devices 文件里
	出错：-1

早期版本的字符设备注册〔2〕

所需头文件	#include <linux/fs.h>
函数原型	int unregister_chrdev(unsigned int major, const char *name)
函数传入值	major：设备的主设备号，必须和注册时的主设备号相同。
	name：设备名
函数返回值	成功：0，且设备名从/proc/devices 文件里消失。
	出错：-1

设备号相关函数〔1〕

- 在linux2.6的版本中,用dev_t类型来描述设备号〔dev_t是32位数值类型,其中高12位表示主设备号,低20位表示次设备号〕。用两个宏MAJOR和MINOR分别获得dev_t设备号的主设备号和次设备号,而且用MKDEV宏来实现逆过程,即组合主设备号和次设备号而获得dev_t类型设备号。
- 分配设备号有静态和动态的两种方法。静态分配(register_chrdev_region()函数)是指在事先知道设备主设备号的情况下,通过参数函数指定第一个设备号〔它的次设备号通常为0〕而向系统申请分配一定数目的设备号。动态分配〔alloc_chrdev_region()〕是指通过参数仅设置第一个次设备号〔通常为0,事先不会知道主设备号〕和要分配的设备数目而系统动态分配所需的设备号。
- 通过unregister_chrdev_region()函数释放已分配的〔无论是静态的还是动态的〕设备号。

设备号相关函数〔2〕

所需头文件	#include <linux/fs.h>
函数原型	int register_chrdev_region (dev_t first, unsigned int count, char *name) int alloc_chrdev_region (dev_t *dev, unsigned int firstminor, unsigned int count, char *name) void unregister_chrdev_region (dev_t first, unsigned int count)
函数传入值	first：要分配的设备号的初始值。 count：要分配（释放）的设备号数目 name：要申请设备号的设备名称（在/proc/devices 和 sysfs 中显示） dev：动态分配的第一个设备号
函数返回值	成功：0（只限于两种注册函数）
	出错：-1（只限于两种注册函数）

字符设备注册〔1〕

- 在Linux内核中使用struct cdev结构来描述字符设备,我们在驱动程序中必须将已分配到的设备号以及设备操作接口〔即为struct file_operations结构〕赋予struct cdev结构变量。首先使用cdev_alloc()函数向系统申请分配struct cdev结构,再用cdev_init()函数初始化已分配到的结构并与file_operations结构关联起来。最后调用cdev_add()函数将设备号与struct cdev结构进行关联并向内核正式报告新设备的注册,这样新设备可以被用起来了！。
- 如果要从系统中删除一个设备,则要调用cdev_del()函数。

字符设备注册〔2〕

所需头文件	#include <linux/cdev.h>
函数原型	sturct cdev *cdev_alloc(void) void cdev_init(struct cdev *cdev, struct file_operations *fops) int cdev_add (struct cdev *cdev, dev_t num, unsigned int count) void cdev_del(struct cdev *dev)
函数传入值	cdev：需要初始化/注册/删除的 struct cdev 结构 fops：该字符设备的 file_operations 结构 num：系统给该设备分配的第一个设备号 count：该设备对应的设备号数量
函数返回值	成功： cdev_alloc：返回分配到的 struct cdev 结构指针 cdev_add：返回 0
	出错： cdev_alloc：返回 NULL cdev_add：返回 -1

翻开设备

- 翻开设备的函数接口是open,根据设备的不同,open函数接口完成的功能也有所不同,但通常情况下在open函数接口中要完成如下工作。
- 递增计数器,检查错误。
- 如果未初始化,则进行初始化。
- 识别次设备号,如果必要,更新f_op指针。
- 分配并填写被置于filp->private_data的数据结构。
- 其中递增计数器是用于设备计数的。由于设备在使用时通常会翻开屡次,也可以由不同的进程所使用,所以若有一进程想要删除该设备,则必须保证其他设备没有使用该设备。因此使用计数器就可以很好地完成这项功能。

释放设备

- 释放设备的函数接口是release()。要注意释放设备和关闭设备是完全不同的。当一个进程释放设备时,其他进程还能继续使用该设备,只是该进程暂时停止对该设备的使用；而当一个进程关闭设备时,其他进程必须重新翻开此设备才能使用它。
- 释放设备时要完成的工作如下。
- 递减计数器MOD_DEC_USE_COUNT〔最新版本已经不再使用〕。
- 释放翻开设备时系统所分配的内存空间〔包括filp->private_data指向的内存空间〕。
- 在最后一次释放设备操作时关闭设备。

读写设备

- 读写设备的主要任务就是把内核空间的数据复制到用户空间,或者从用户空间复制到内核空间,也就是将内核空间缓冲区里的数据复制到用户空间的缓冲区中或者相反。

所需头文件	#include <linux/fs.h>
函数原型	ssize_t (*read) (struct file *filp, char *buff, size_t count, loff_t *offp) ssize_t (*write) (struct file *filp, const char *buff, size_t count, loff_t *offp)
函数传入值	filp：文件指针
	buff：指向用户缓冲区
	count：传入的数据长度
	offp：用户在文件中的位置
函数返回值	成功：写入的数据长度

内核空间和用户空间的数据交换

- 内核空间地址和用户空间地址是有很大区别的,其中一个区别是用户空间的内存是可以被换出的,因此可能会出现页面失效等情况。所以不能使用诸如memcpy()之类的函数来完成这样的操作。在这里要使用copy_to_user()或copy_from_user()等函数,它们是用来实现用户空间和内核空间的数据交换的。

所需头文件	#include <asm/uaccess.h>
函数原型	unsigned long copy_to_user(void *to, const void *from, unsigned long count) unsigned long copy_from_user(void *to, const void *from, unsigned long count)
函数传入值	to：数据目的缓冲区
	from：数据源缓冲区
	count：数据长度
函数返回值	成功：写入的数据长度 失败：-EFAULT

ioctl

- 大局部设备除了读写操作,还需要硬件配置和控制〔例如,设置串口设备的波特率〕等很多其他操作。在字符设备驱动中ioctl函数接口给用户提供对设备的非读写操作机制。

所需头文件	#include <linux/fs.h>
函数原型	int(*ioctl)(struct inode* inode, struct file* filp, unsigned int cmd, unsigned long arg)
函数传入值	inode：文件的内核内部结构指针
	filp：被打开的文件描述符
	cmd：命令类型
	arg：命令相关参数

获取内存〔1〕

- 在应用程序中获取内存通常使用函数malloc(),但在设备驱动程序中动态开辟内存可以以字节或页面为单位。其中,以字节为单位分配内存的函数有kmalloc(),注意的是,kmalloc()函数返回的是物理地址,而malloc()等返回的是线性虚拟地址,因此在驱动程序中不能使用malloc()函数。与malloc()不同,kmalloc()申请空间有大小限制。长度是2的整次方,并且不会对所获取的内存空间清零。
- 以页为单位分配内存的函数如下所示:
- get_zeroed_page()：获得一个已清零页面。
- get_free_page()：获得一个或几个连续页面。
- get_dma_pages()：获得用于DMA传输的页面。
- 与之相对应的释放内存用也有kfree()或free_page函数族。

获取内存〔2〕

所需头文件	#include <linux/malloc.h>	
函数原型	void *kmalloc(unsigned int len,int flags)	
函数传入值	len：希望申请的字节数	
	flags	GFP_KERNEL：内核内存的通常分配方法，可能引起睡眠
		GFP_BUFFER：用于管理缓冲区高速缓存
		GFP_ATOMIC：为中断处理程序或其他运行于进程上下文之外的代码分配内存，且不会引起睡眠
		GFP_USER：用户分配内存，可能引起睡眠
		GFP_HIGHUSER：优先高端内存分配
		__GFP_DMA：DMA 数据传输请求内存
		__GFP_HIGHMEN：请求高端内存
函数返回值	成功：写入的数据长度 失败：-EFAULT	

获取内存〔3〕

所需头文件	#include <linux/malloc.h>
函数原型	void kfree(void * obj)
函数传入值	obj：要释放的内存指针
函数返回值	成功：写入的数据长度 失败：-EFAULT

所需头文件	#include <linux/malloc.h>
函数原型	unsigned long get_zeroed_page(int flags) unsigned long __get_free_page(int flags) unsigned long __get_free_page(int flags,unsigned long order) unsigned long __get_dma_page(int flags,unsigned long order)
函数传入值	flags：同 kmalloc()
	order：要请求的页面数，以 2 为底的对数
函数返回值	成功：返回指向新分配的页面的指针 失败：-EFAULT

获取内存〔4〕

所需头文件	#include <linux/malloc.h>
函数原型	unsigned long free_page(unsigned long addr) unsigned long free_pages(unsigned long addr, unsigned long order)
函数传入值	addr：要释放的内存起始地址
	order：要请求的页面数，以 2 为底的对数
函数返回值	成功：写入的数据长度 失败：-EFAULT

打印信息

- 在内核空间要用函数printk()而不能用平常的函数printf()。printk()还可以定义打印消息的优先级。

所需头文件	#include <linux/kernel>	
函数原型	int printk(const char * fmt, …)	
函数传入值	fmt： 日志级别	KERN_EMERG：紧急时间消息
		KERN_ALERT：需要立即采取动作的情况
		KERN_CRIT：临界状态，通常涉及严重的硬件或软件操作失败
		KERN_ERR：错误报告
		KERN_WARNING：对可能出现的问题提出警告
		KERN_NOTICE：有必要进行提示的正常情况
		KERN_INFO：提示性信息
		KERN_DEBUG：调试信息
	…：与 printf()相同	
函数返回值	成功：0 失败：-1	

proc文件系统〔1〕

/proc文件系统是一个伪文件系统,它是一种内核和内核模块用来向进程发送信息的机制。这个伪文件系统让用户可以和内核内部数据结构进行交互,获取有关系统和进程的有用信息,在运行时通过改变内核参数来改变设置。与其他文件系统不同,/proc存在于内存之中而不是在硬盘上。读者可以通过“ls”查看/proc文件系统的内容。

proc文件系统〔2〕

目录名称	目录内容	目录名称	目录内容
apm	高级电源管理信息	locks	内核锁
cmdline	内核命令行	meminfo	内存信息
cpuinfo	CPU 相关信息	misc	杂项
devices	设备信息（块设备/字符设备）	modules	加载模块列表
dma	使用的 DMA 通道信息	mounts	加载的文件系统
filesystems	支持的文件系统信息	partitions	系统识别的分区表
interrupts	中断的使用信息	rtc	实时时钟
ioports	I/O 端口的使用信息	stat	全面统计状态表
kcore	内核映像	swaps	对换空间的利用情况
kmsg	内核消息	version	内核版本
ksyms	内核符号表	uptime	系统正常运行时间
loadavg	负载均衡	……	……

目录名称	目录内容	目录名称	目录内容
cmdline	命令行参数	cwd	当前工作目录的链接
environ	环境变量值	exe	指向该进程的执行命令文件
fd	一个包含所有文件描述符的目录	maps	内存映像
mem	进程的内存被利用情况	statm	进程内存状态信息
stat	进程状态	root	链接此进程的 root 目录
status	进程当前状态，以可读的方式显示出来	……	……

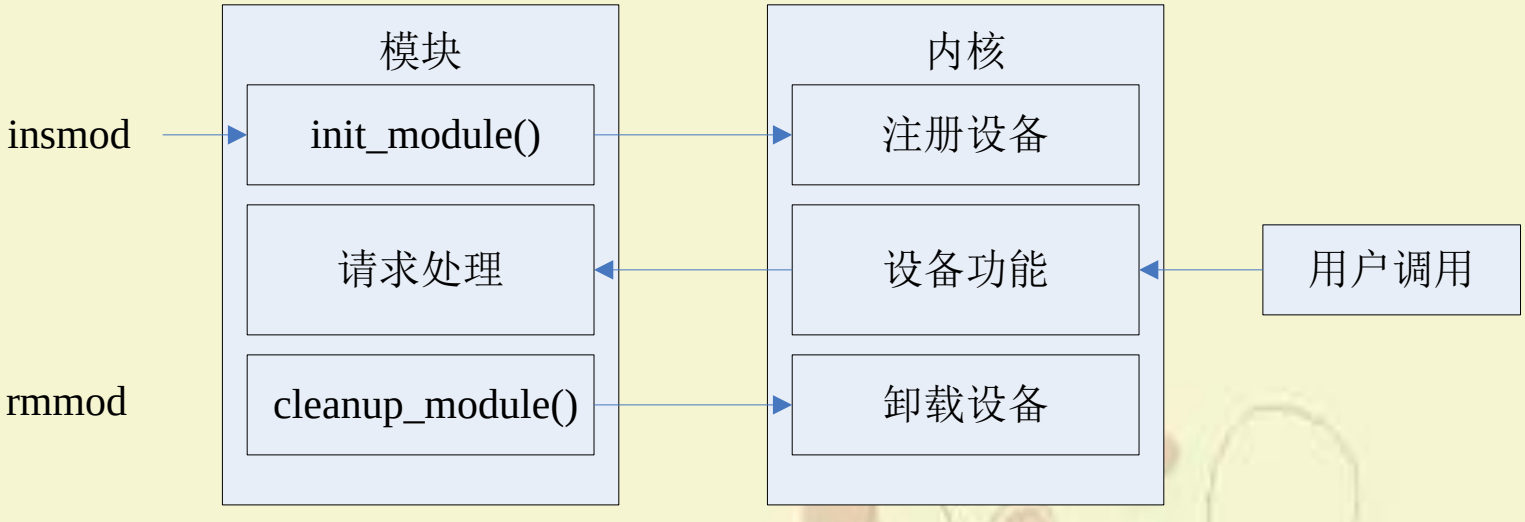
块设备驱动编程

块设备驱动

- 块设备通常指一些需要以块〔如512字节〕的方式写入的设备,如IDE硬盘、SCSI硬盘、光驱等。它的驱动程序的编写过程与字符型设备驱动程序的编写有很大的区别。
- 块设备驱动编程接口相对复杂,不如字符设备明晰易用。块设备驱动程序对整个系统的性能影响较大,速度和效率是设计块设备驱动程要重点考虑的问题。系统中使用缓冲区与访问请求的优化管理〔合并与重新排序〕来提高系统性能。

块设备驱开工作流程

- 块设备驱动程序的编写流程同字符设备驱动程序的编写流程很类似,也包括了注册和使用两局部。但与字符驱动设备所不同的是,块设备驱动程序包括一个request请求队列。它是当内核安排一次数据传输时在列表中的一个请求队列,以最大化系统性能为原则进行排序。



重要数据结构〔1〕

- 每个块设备物理实体由一个gendisk结构体来表示,每个gendisk可以支持多个分区。每个gendisk中包含了本物理实体的全部信息以及操作函数接口。整个块设备的注册过程是围绕gendisk来展开的。

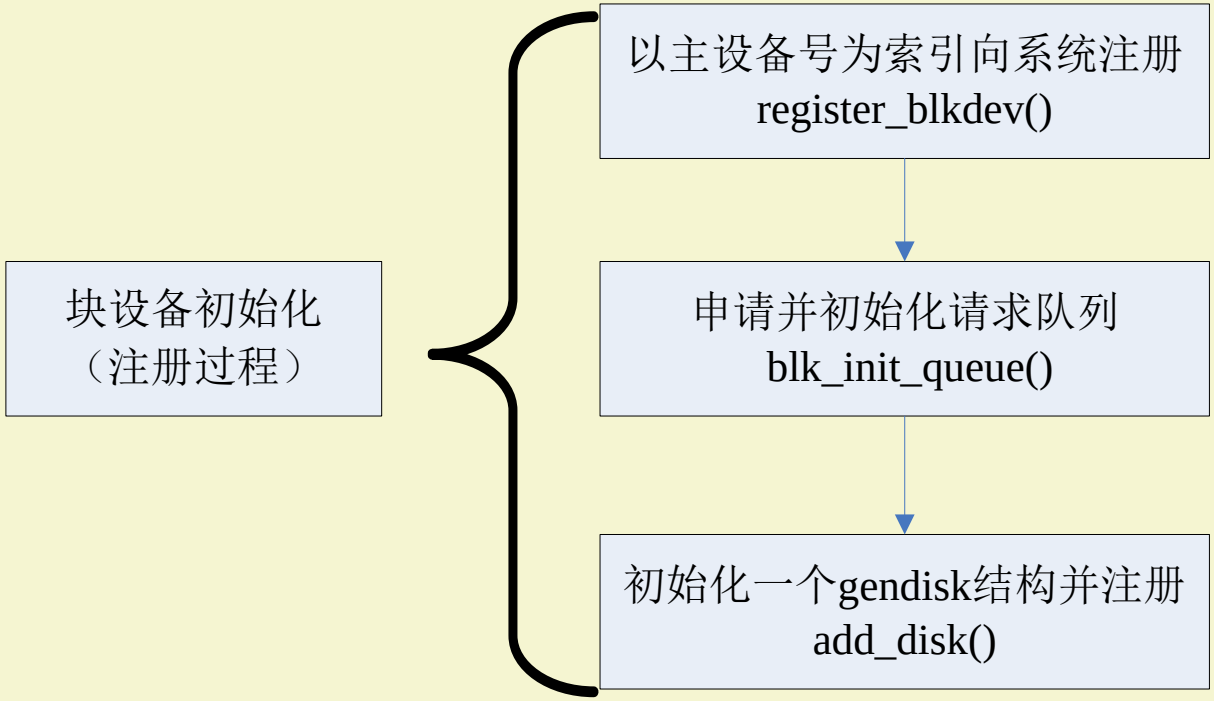
```
struct gendisk
{
    int major;          /* 主设备号 */
    int first_minor;    /* 第一个次设备号 */
    int minors;         /* 次设备号个数,一个块设备至少需要使用一个次设备号,而且块设
                备的每个分区都需要一个次设备号,因此这个成员等于1,则说明该块
                设备是不可被分区的,否则可以包含minors – 1    个分区。*/
    char disk_name[32];       /* 块设备名称,在/proc/partions中显示 */
    struct hd_struct **part;    /* 分区表 */
    struct block_device_operations *fops;       /* 块设备操作接口,与字符设备的
                                    file_operations结构对应*/
    struct request_queue *queue;    /* I/O请求队列 */
    void *private_data;       /* 指向驱动程序私有数据 */
    sector_t capacity;    /* 块设备可包含的扇区数 */
    …… /* 其他省略 */
};
```

重要数据结构〔2〕

块设备驱动程序也包含一个在<linux/fs.h>中定义的block_device_operations结构块设备并不提供read()、write()等函数接口。对块设备的读写请求都是以异步方式发送到设备相关的request 队列之中。

```
struct block_device_operations
{
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned, unsigned long);
    int (*direct_access) (struct block_device *, sector_t, unsigned long *);
    int (*media_changed) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    int (*getgeo)(struct block_device *, struct hd_geometry *);
    struct module *owner;
};
```

块设备注册和初始化〔1〕


块设备初始化
（注册过程）
以主设备号为索引向系统注册
register_blkdev()
申请并初始化请求队列
blk_init_queue()
初始化一个gendisk结构并注册
add_disk()

块设备注册和初始化〔2〕

〔1〕向内核注册

使用register_blkdev() 函数对设备进行注册。

```
int register_blkdev(unsigned int major, const char *name);
```

其中参数major为要注册的块设备的主设备号,如果其值等于0,则系统动态分配并返回主设备号。参数name为设备名,在/proc/devices中显示。如果出错,则该函数返回负值。

与其对应的块设备的注销函数为unregister_blkdev() 。

```
int unregister_blkdev(unsigned int major, const char *name);
```

其参数必须与注册函数中的参数相同。如果出错则返回负值。

〔2〕申请并初始化请求队列

这一步要调用blk_init_queue()函数来申请并初始化请求队列。

```
struct request_queue *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
```

其中参数rfn是请求队列的处理函数指针,它负责执行块设备的读、写请求。参数lock为自旋锁,用于控制对所分配的队列的访问。

块设备注册和初始化〔3〕

〔3〕初始化并注册gendisk结构

函数格式	说明
struct gendisk *alloc_disk(int minors)	动态分配gendisk结构，参数为次设备号的个数。
void add_disk(struct gendisk *disk)	向系统注册 gendisk 结构
void del_gendisk(struct gendisk *disk)	从系统注销 gendisk 结构

首先使用alloc_disk()函数动态分配gendisk结构,接下来,对gendisk结构的主设备号〔major〕、次设备号相关成员〔first_minor和minors〕、块设备操作函数〔fops〕、请求队列〔queue〕、可包含的扇区数〔capacity〕以及设备名称〔disk_name〕等成员进行初始化。

在完成对gendisk的分配和初始化之后,调用add_disk〔 〕函数向系统注册块设备。在卸载gendisk结构的时候,要调用del_gendisk()函数。

块设备请求处理

块设备驱动中一般要实现一个请求队列处理函数来处理队列中的请求。从块设备的运行流程,可知请求处理是块设备的根本处理单位,也是最核心的局部。对块设备的读写操作被封装到了每一个请求中。

函数格式	说明
request_queue_t *blk_alloc_queue(int gfp_mask)	分配请求队列
request_queue_t *blk_init_queue (request_fn_proc *rfn, spinlock_t *lock)	分配并初始化请求队列
struct request *blk_get_request (request_queue_t *q, int rw, int gfp_mask)	从队列中获取一个请求
void blk_requeue_request(request_queue_t *q, struct request *rq)	将请求再次加入队列
void blk_queue_max_sectors (request_queue_t *q, unsigned short max_sectors)	设置最大访问扇区数
void blk_queue_max_phys_segments (request_queue_t *q, unsigned short max_segments)	设置最大物理段数
void end_request(struct request *req, int uptodate)	结束本次请求处理
void blk_queue_hardsect_size (request_queue_t *q, unsigned short size)	设置物理扇区大小

中断编程

中断编程接口〔1〕

- 实际上,有很多Linux的驱动都是通过中断的方式来进行内核和硬件的交互。中断机制提供了硬件和软件之间异步传递信息的方式。硬件设备在发生某个事件时通过中断通知软件进行处理。中断实现了硬件设备按需获得处理器关注的机制,与查询方式相比可以大大节省CPU资源的开销。
- 申请中断使用request_irq()调用,释放中断使用free_irq()调用。

```
int request_irq(unsigned int irq,
        void (*handler)(int irq, void *dev_id, struct pt_regs *regs),
            unsigned long irqflags, const char * devname, oid *dev_id);
```

```
void free_irq(unsigned int irq, void *dev_id);
```

中断编程接口〔2〕

- 其中irq是要申请的硬件中断号。在Intel平台,范围是0～15。
- 参数handler为将要向系统注册的中断处理函数。这是一个回调函数,中断发生时,系统调用这个函数,传入的参数包括硬件中断号、设备id以及存放器值。设备id就是在调用request_irq()时传递给系统的参数dev_id。
- 参数irqflags是中断处理的一些属性,其中比较重要的有SA_INTERRUPT。这个参数用于标明中断处理程序是快速处理程序〔设置SA_INTERRUPT〕还是慢速处理程序〔不设置SA_INTERRUPT〕。快速处理程序被调用时屏蔽所有中断。慢速处理程序只屏蔽正在处理的中断。还有一个SA_SHIRQ属性,设置了以后运行多个设备共享中断,在中断处理程序中根据dev_id区分不同设备产生的中断。
- 参数devname为设备名,会在/dev/interrupts中显示。
- 参数dev_id在中断共享时会用到。一般设置为这个设备的device结构本身或者NULL。中断处理程序可以用dev_id找到相应的控制这个中断的设备,或者用irq2dev_map()找到中断对应的设备。

实践任务：申请一段内存空间作为设备进行读写操作

- **1.** 首先创建头文件**driver_insmod.c**源文件
- 第一步：声明文件包含、宏定义及变量定义。

其中**MEM_MAJOR**是一个主设备号，这里我们采用的静态获取主设备号，如果在当前的**Linux**系统下，主设备号**246**已经被某一个设备占用，需要重新选择一个其它的主设备号。

```
MODULE_LICENSE("GPL");                  //声明模块的许可证
#define MEM_MALLOC_SIZE    4096       //定义申请内存字节数
#define MEM_MAJOR          246        //定义主设备号
#define MEM_MINOR          0          //定义次设备号
char *mem_spvm;          //定义内存指针 mem_spvm
struct cdev *mem_cdev;       //定义设备对象 mem_cdev
struct class *mem_class;       //定义设备类 mem_class
```

第二步：声明模块安装初始化和退出函数并定义设备驱动文件结构体。

```
/*声明模块安装初始化和退出函数*/
static int    __init    driver_init_module (void);
static void    __exit    driver_exit_module (void);
module_init(driver_init_module);
module_exit(driver_exit_module);

/*声明文件结构体中所用的域函数*/
static int mem_open(struct inode *ind, struct file *filp);
static int mem_release(struct inode *ind, struct file *filp);
static ssize_t mem_read(struct file *filp, char __user *buf, size_t size, loff_t *fpos);
static ssize_t mem_write(struct file *filp, const char __user *buf, size_t size, loff_t *fpos);

/*定义设备驱动文件结构体*/
struct file_operations mem_fops =
{
        .open = mem_open,
        .release = mem_release,
        .read = mem_read,
        .write = mem_write,
};
```

- 第三步： 编写模块的安装操作函数，其中主要包括将**mem_spvm**指向内核虚拟内存空间，并将其加载进内核系统设备中。

```
int __init driver_init_module (void)
{
    int res;
    int devno = MKDEV(MEM_MAJOR, 0);
    mem_spvm = (char *)vmalloc(MEM_MALLOC_SIZE);
    if (mem_spvm == NULL)
        printk(KERN_INFO"vmalloc failed!\n");
    else
        printk(KERN_INFO"vmalloc successfully! addr=0x%x\n", (unsigned int)mem_spvm);
```

```
/*动态分配一个新的字符设备对象 mem_cdev */
    mem_cdev = cdev_alloc();
    if (mem_cdev == NULL)
    {
        printk(KERN_INFO"cdev_alloc failed!\n");
        return 0;
    }
```

```
/*初始化字符设备对象 mem_cdev */
    cdev_init(mem_cdev, &mem_fops);
    mem_cdev->owner = THIS_MODULE;
    mem_cdev->ops = &mem_fops;

    /*向内核系统中添加一个新的字符设备 mem_cdev */
    res = cdev_add(mem_cdev, devno, 1);
    if (res)
    {
        cdev_del(mem_cdev);
        mem_cdev = NULL;
        printk(KERN_INFO"cdev_add error\n");
    }
    else
    {
        printk(KERN_INFO"cdev_add ok\n");
    }
```

```
/*  建立一个系统设备类  mem_class */
    mem_class = class_create(THIS_MODULE, "myalloc");
    if(IS_ERR(mem_class)) {
          printk("Err: failed in creating class.\n");
        return -1;
    }

/*  注册设备文件系统，  并建立设备节点  */
    device_create(mem_class, NULL, MKDEV(MEM_MAJOR,0), NULL, "myalloc");
    return 0;
}
```

- 第四步：编写模块的退出操作函数，其中主要包括删除设备结构、设备文件及设备节点，将**mem_spvm**所指向内核虚拟内存空间释放掉。

```
void __exit driver_exit_module (void)
{
	if (mem_cdev != NULL)
		cdev_del(mem_cdev);    //从内核中将设备删除
	printk(KERN_INFO"cdev_del ok\n");
	device_destroy(mem_class, MKDEV(MEM_MAJOR, 0)); //删除设备节点及目录
	class_destroy(mem_class);            //删除设备类
	if (mem_spvm != NULL)
		vfree(mem_spvm);
	printk(KERN_INFO"vfree ok!\n");
}
```

- 第五步：编写设备文件结构中的打开设备操作函数。

```
int mem_open(struct inode *ind, struct file *filp)
{
    printk(KERN_INFO"open vmalloc space\n");
    try_module_get(THIS_MODULE);      //模块使用计数加 1
    return 0;
}
```

- 第六步：编写设备文件结构中的读设备操作函数，读取内核存储空间，然后将内核空间的内存内容复制到用户空间。

```
ssize_t mem_read(struct file *filp, char *buf, size_t size, loff_t *lofp)
{
	int res = -1;
	char *tmp;
	struct inode *inodep;
	inodep = filp->f_dentry->d_inode;
	tmp = mem_spvm;
	if (size > MEM_MALLOC_SIZE)
		size = MEM_MALLOC_SIZE;
	if (tmp != NULL)
		res = copy_to_user(buf, tmp, size);//将内核空间的内存内容复制到用户空间
	if (res == 0)
		return size;
	else
		return 0;
}
```

- 第七步：编写设备文件结构中的写设备操作函数，将用户空间的内存内容复制到内核空间。

```
ssize_t mem_write(struct file *filp, const char *buf, size_t size, loff_t *lofp)
{
	int res = -1;
	char *tmp;
	struct inode *inodep;
	inodep = filp->f_dentry->d_inode;
		tmp = mem_spvm;
	if (size > MEM_MALLOC_SIZE)
		size = MEM_MALLOC_SIZE;
	if (tmp != NULL)
		res = copy_from_user(tmp, buf, size);   //将用户空间的内存内容复制到内核空间
	if (res == 0)
		return size;
	else
		return 0;
}
```

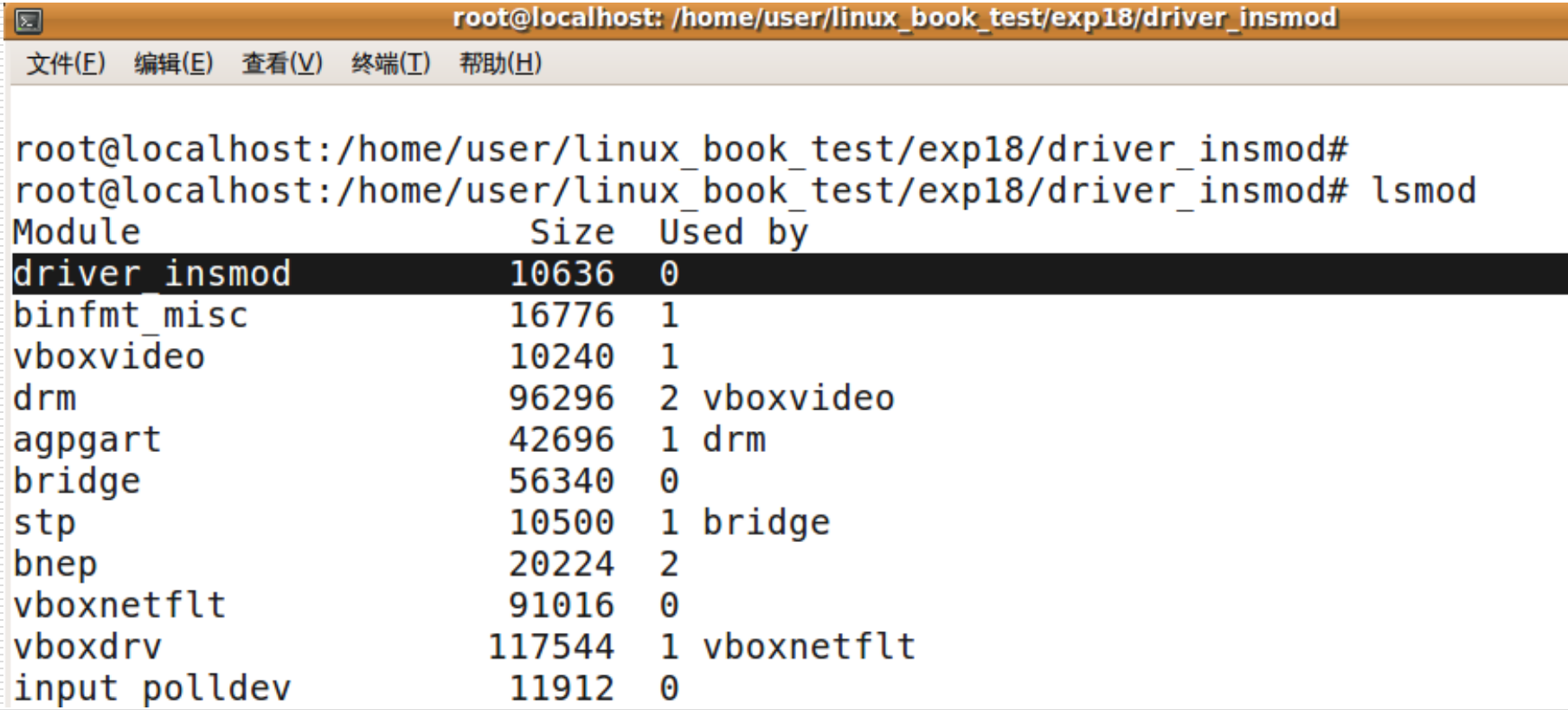
- 第八步：编写设备文件结构中的释放设备操作函数，给出提示信息，使模块使用计数减1。

```
int mem_release(struct inode *ind, struct file *filp)
{
    printk(KERN_INFO"close vmalloc space\n");
    module_put(THIS_MODULE);    //模块计数减 1
    return 0;
}
```

2. 编写Makefile如下：

```
ifneq ($(KERNELRELEASE),)
obj-m += driver_insmod.o
else
PWD := $(shell pwd)
KVER := $(shell uname -r)
KDIR := /lib/modules/$(KVER)/build
all:
	$(MAKE) -C $(KDIR) M=$(PWD)
clean:
	rm -rf *.o *.mod.c *.ko *.symvers *.order *.markers *~
endif
```

3. 编译并安装模块，安装后用lsmod 命令查看安装的模块



测试驱动程序

- 编写测试函数对内存设备进行测试。