

Linux设备驱动

杨少华

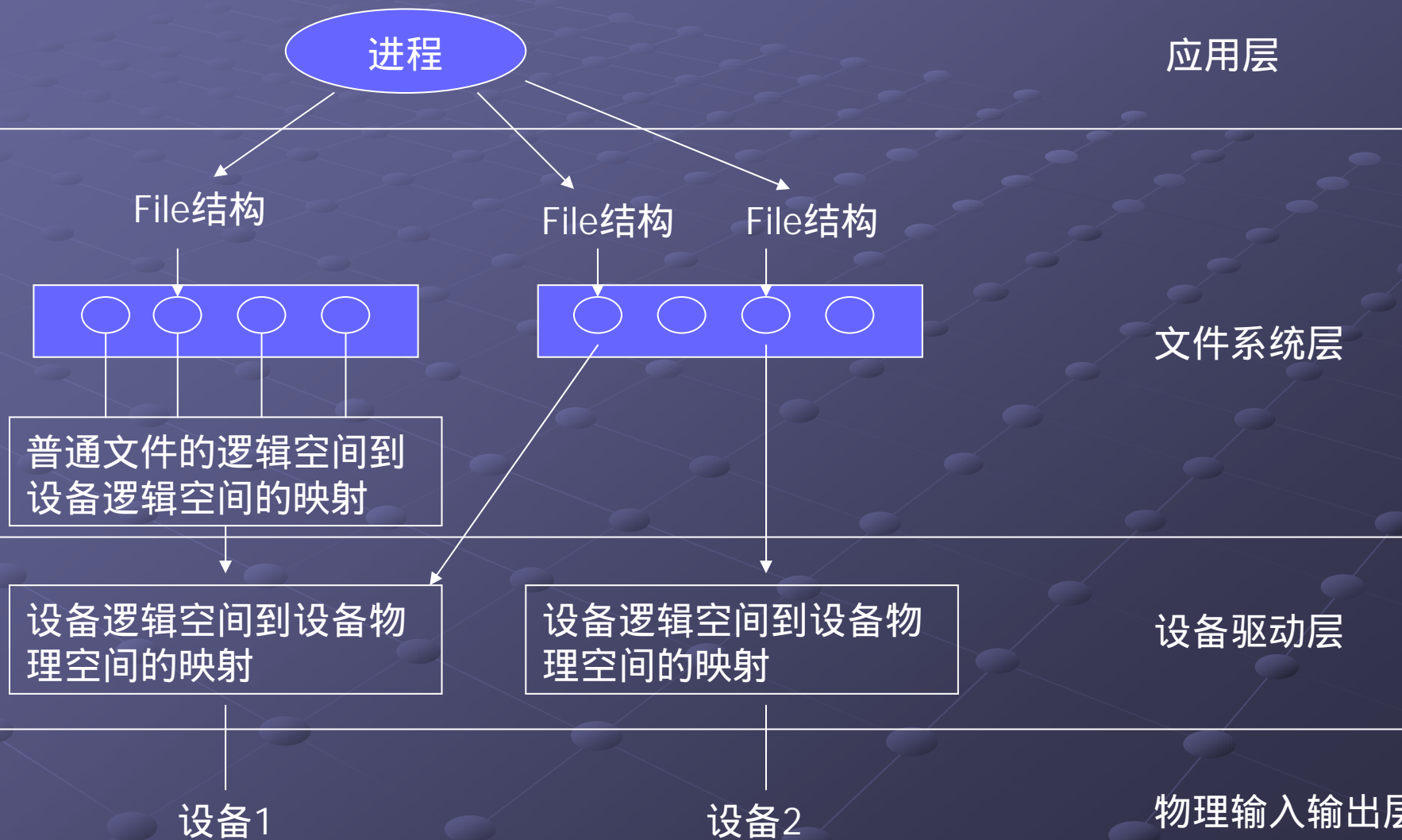
报告主要内容

- 基本原理
- 一个典型设备驱动的实现
- 驱动程序的安装
- 驱动程序的使用

基本原理

- 设备驱动分层结构
- 可安装模块 (module)
- 数据结构
 - 文件系统、module、字符设备、块设备
- 几个系统调用
 - mknod、create_module、init_module
query_module、delete_module

设备驱动分层结构



可安装模块

● module

- 经过编译但尚未连接的目标代码（.o）文件，可在系统运行时动态地“安装”到内核中
- 模块可以使用内核“移出”的一些符号，如子程序入口、全局变量
- module如果引用内核移出符号，声明为extern外部变量

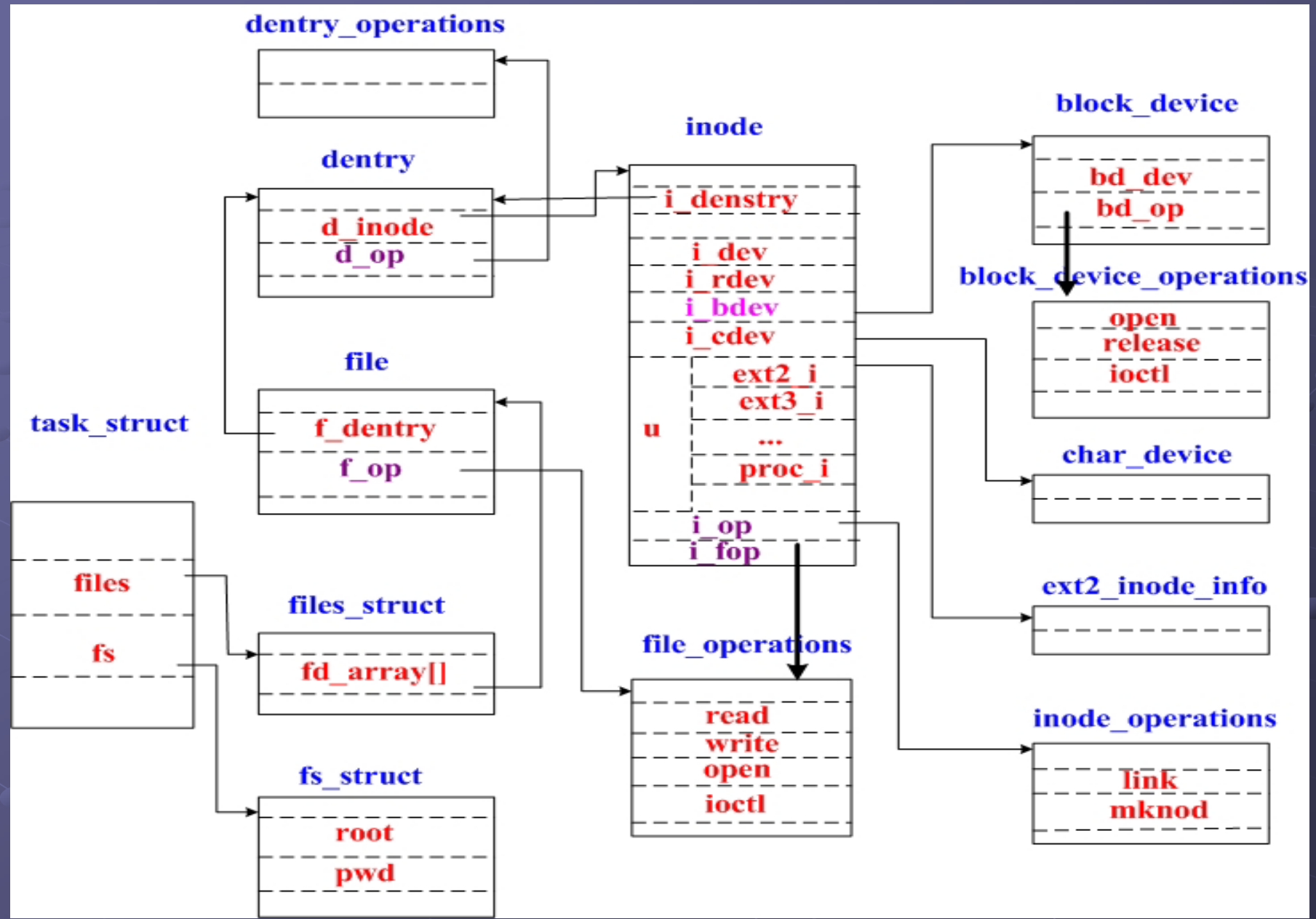
● /sbin/insmod

- 负责模块的装入和与内核的连接

● mknod

- 建立设备文件节点，之后通过open、read、write、ioctl对设备进行操作

纳入设备驱动的FS



字符设备驱动

- #define MAX_CHRDEV 255
- struct device_struct {
- const char * name;
- struct file_operations * fops;
- };
- static struct device_struct chrdevs[MAX_CHRDEV];

内核提供注册、注销字符设备操作函数的方法（在模块的init函数调用）：

devfs_register_chrdev
devfs_unregister_chrdev


输入：

主设备号
模块名
操作函数入口表

```
struct file_operations {  
    struct module *owner;  
    llseek;  
    read;  
    write;  
    readdir;  
    ioctl;  
    ... ..  
}
```

块设备驱动

```
● #define MAX_BLKDEV    255
● static struct {
●     const char *name;
●     struct block_device_operations *bdops;
● } blkdevs[MAX_BLKDEV];
```



内核提供注册、注销块设备操作函数的方法（在模块的init函数调用）：

devfs_register_blkdev
devfs_unregister_blkdev

输入：

主设备号

模块名

操作函数入口表

```
struct block_device_operations {
    open;
    release;
    ioctl;
    check_media_change;
    revalidate;
    struct module *owner;
};
```


module

```
● struct module
● {
●     unsigned long size_of_struct;      // module结构体大小
●     struct module *next;               // 链表指针
●     const char *name;                  // 模块名称
●     unsigned long size;                 // 模块映像大小
●     atomic_t usecount;                  // 使用计数
●     unsigned long flags;                // 标志，指示模块状态等信息
●     ... ..
● }
```

模块状态有：MOD_UNINITIALIZED、MOD_INITIALIZING、MOD_RUNNING、
MOD_DELETED、MOD_JUST_FREED

其他标志位：MOD_AUTOCLEAN、MOD_VISITED、MOD_USED_ONCE

module

• struct module

• {

•

• unsigned nsyms; //符号计数，指明符号表syms大小

• unsigned ndeps; //依赖模块计数，指明deps大小

• struct module_symbol *syms; //模块符号表

• struct module_ref *deps; //依赖模块数组

• struct module_ref *refs; //引用模块链表

• int (*init)(void); //指向模块的init函数

• void (*cleanup)(void); //指向模块的cleanup函数

• //异常入口表开始与截止地址

• const struct exception_table_entry *ex_table_start;

• const struct exception_table_entry *ex_table_end;

•

• }

```
struct module_symbol  
{  
    unsigned long value;  
    const char *name;  
};
```

```
struct module_ref  
{  
    struct module *dep;  
    struct module *ref;  
    struct module_ref *next_ref;  
};
```

module

- struct module

- {

-

- // 持久性数据

- const struct module_persist *persist_start;

- const struct module_persist *persist_end;

- int (*can_unload)(void);

- int runsize;

- const char *kallsyms_start;

- const char *kallsyms_end;

- const char *archdata_start;

- const char *archdata_end;

- const char *kernel_data;

- }

模块队列

module_list

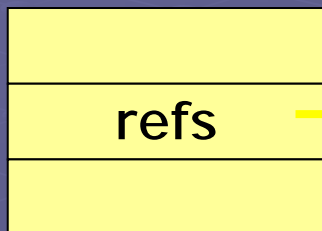


模块的依赖

```
struct module_ref  
{  
    struct module *dep;  
    struct module *ref;  
    struct module_ref *next_ref;  
};
```

struct module

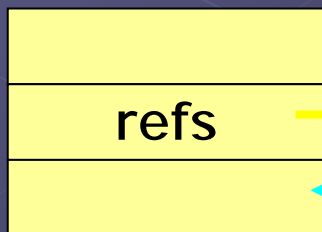
Parent 1



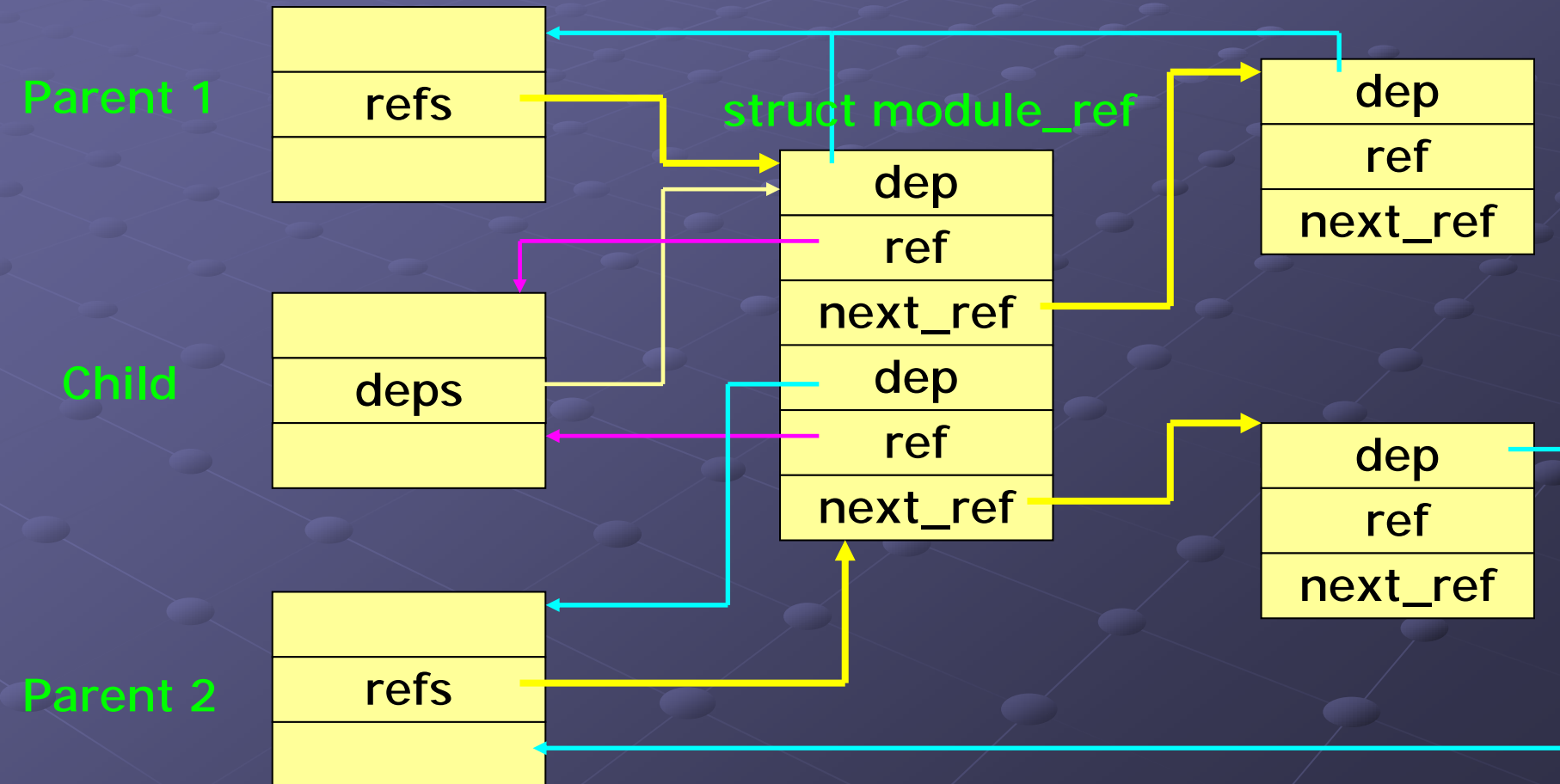
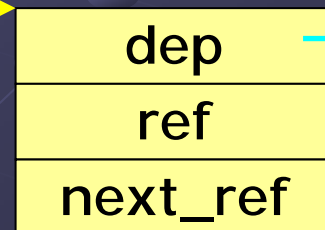
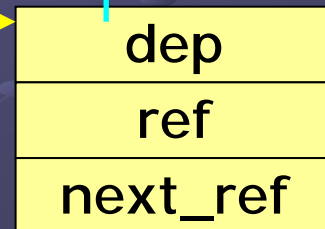
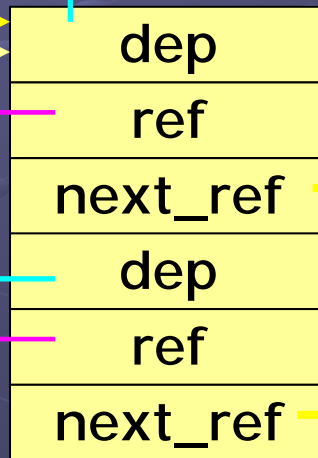
Child



Parent 2



struct module_ref



mknod



`sys_mknod(filename, mode, dev) mknod(inode, dentry, mode, rdev)`
`filename: 父节点 inode: 设备节点 inode`
`mode: 权限及标志 dentry: 欲创建设备节点之 dentry`
`dev: 高8位主设备号, 低8位次设备号`
`ext2_mknod(dir, dentry, mode, dev)`
`dir: 父节点 inode: 设备节点 inode`
`mode: 权限及标志 dentry: 欲创建设备节点之 dentry`
`dev: 高8位主设备号, 低8位次设备号`
`init_special_inode(inode, mode, rdev)`
`inode: 设备节点 inode`
`mode: 权限及标志 dentry: 欲创建设备节点之 dentry`
`dev: 高8位主设备号, 低8位次设备号`
`def_chrfops = {`
`open: chrdev_open,`
`};`

```

... ..
return ret;
} else
... ..
}
  
```

create_module

- `sys_create_module(const char *name_user, size_t size)`
- {
- 参数及权限检查；
- 为**module映象**申请空间；
- //包括module结构体和其余部分（节）如
- //指令、数据、符号表等
- 初始化module的名称及大小；
- //大小有size_of_struct和**size**（**module映象**）
- 链入module_list队列；
- }

query_module

- 模块查询系统调用

- `sys_query_module(const char *name_user, int which, char *buf, size_t bufsize, size_t *ret)`

- `name_user` : 查询对象所在模块, 0则为内核
- `which` : 查询内容, QM_MODULES、QM_DEPS、QM_REFS、QM_SYMBOLS、QM_INFO
- `buf` : 查询结果缓冲区
- `bufsize` : `buf`大小
- `ret` : 写入`buf`的实际大小

init_module

- 完成module映象从用户空间向系统空间的拷贝，最后调用模块中用户编写的init_module函数（与同名系统调用相区别）对模块进行初始化
- `sys_init_module(const char *name_user, struct module *mod_user)`
 - name_user：模块名
 - mod_user：模块映象：module结构 + 各种“section”（详见下页）
- 调用init_module前需由应用程序（insmod）在用户空间完成模块与内核符号的连接，即mod_user指向已连接好的模块映象

.o文件与module映象

.o文件

ELF头	
section头表	
sections	.strtab
	.symtab
	.data.init
	.dynamic
	.text.init

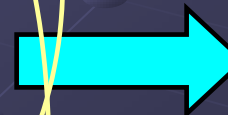
insmod



module映象

module 结构	syms
	deps
	init
	kallsyms_start
sections
	.strtab
	__kallsyms
	.kmodtab
	.symtab
	.data.init
	.text.init

init_module
系统调用




init_module

```
sys_init_module(const char *name_user, struct module
*mod_user)
{ // name_user模块名，mod_user指向module映象
  权限检查；
  参数检查；
  从用户空间拷贝module结构到系统空间；
  module映象中的指针检查（指针落在映象范围内）；
  拷贝映象的其余部分（各种“section”）；
  检查依赖和设置引用关系；
  调用模块的init函数（mod->init()）；
}
```

由此可见一个模块映象装入内核空间的过程非常简单（只是拷贝的过程），模块映象转入后即可顺利运行，有点不可思议！这些都是insmod做了大量的工作。在后面会较详细的说明insmod的过程。

delete_module

- `sys_delete_module(const char *name_user)`
- { // name_user非0：拆除一个特定模块，为0：拆除所有可拆除模块
 - `if(name_user) { // 拆除一个特定模块`
 - `mod = find_module(name);`
 - `free_module(mod);`
 - }
 - `else { // 拆除所有可拆除模块`
 - `循环拆除可拆除的模块；`
 - }
- }



`free_module`首先调用目标模块的`cleanup_module()`函数，通常，在这个函数中要将模块在系统中的“登记”撤销，检查该模块所依赖的模块，看是否可以置成可拆除状态。

一个典型设备驱动的实现

- 这里以声卡驱动程序sparcaudio为例介绍一个典型的模块
- 一个典型模块包括：
 - 一组文件操作函数的实现
 - 一个file_operations结构存放文件操作函数指针（如果是块设备还得准备一个block_device_operations结构）
 - init_module()函数
 - cleanup_module()函数
 - 移出符号（函数或变量），使用EXPORT_SYMBOL宏，允许其他模块访问
 - 内核符号声明，使用extern关键字
 - 其他自定义的一些函数或变量

sparcaudio实现

- 一个file_operations结构sparcaudio_fops存放文件操作函数指针

- static struct file_operations sparcaudio_fops = {
- owner: THIS_MODULE,
- llseek: no_llseek,
- read: sparcaudio_read,
- write: sparcaudio_write,
- poll: sparcaudio_poll,
- ioctl: sparcaudio_ioctl,
- open: sparcaudio_open,
- release: sparcaudio_release,
- };

sparcaudio实现

● init_module函数

- static int __init sparcaudio_init(void)
- {
- /* Register our character device driver with the **VFS**. */
- if (devfs_register_chrdev(SOUND_MAJOR,
"sparcaudio", &sparcaudio_fops))
- return -EIO;
- devfs_handle = devfs_mk_dir (NULL, "sound", NULL);
- return 0;
- }

SOUND_MAJOR 主设备号
“sparcaudio” 驱动程序名
Sparcaudio_fops文件操作
函数

注：sparcaudio并非直接操作声卡，所以只是新建一个devfs的目录

sparcaudio实现

cleanup_module函数

- static void __exit sparcaudio_exit(void)
- {
- devfs_unregister_chrdev(SOUND_MAJOR, "sparcaudio");
- devfs_unregister (devfs_handle);
- }

sparcaudio实现

● 移出符号

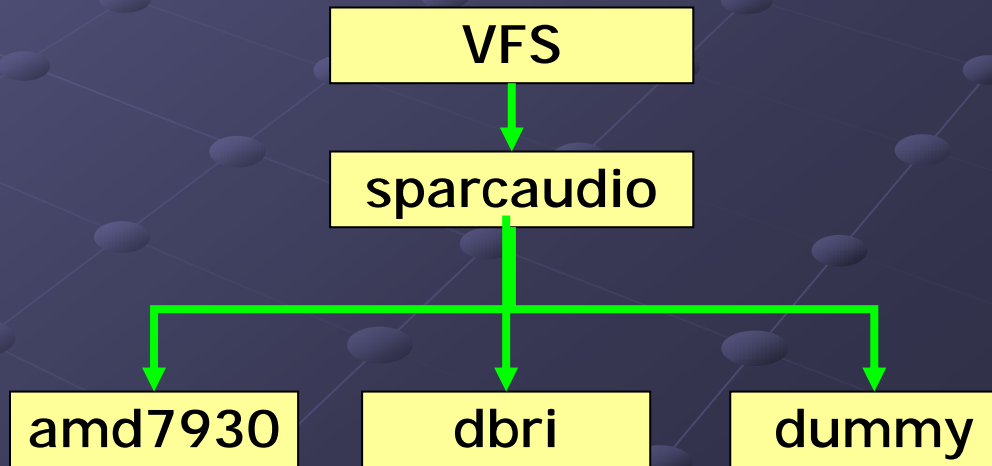
- EXPORT_SYMBOL(register_sparcaudio_driver);
- EXPORT_SYMBOL(unregister_sparcaudio_driver);
- EXPORT_SYMBOL(sparcaudio_output_done);
- EXPORT_SYMBOL(sparcaudio_input_done);

提供给下层模块的注册/
撤销函数，相当于
devfs_register_chrdev

提供给下层模块在完成
输入/输出时使用的
回调函数

注：

模块也可以类似文件系统一样分为不同层次，形成一个“**模块堆栈**”，sparcaudio并不直接操作硬件，而是提供一个**声卡驱动的虚拟层**，层次结构如有图



sparcaudio实现

● static struct sparcaudio_driver *drivers[3];

```
struct sparcaudio_driver
```

```
{
```

```
    const char * name;
```

```
    struct sparcaudio_operations *ops;
```

```
    void *private;
```

```
    ... ..
```

```
};
```

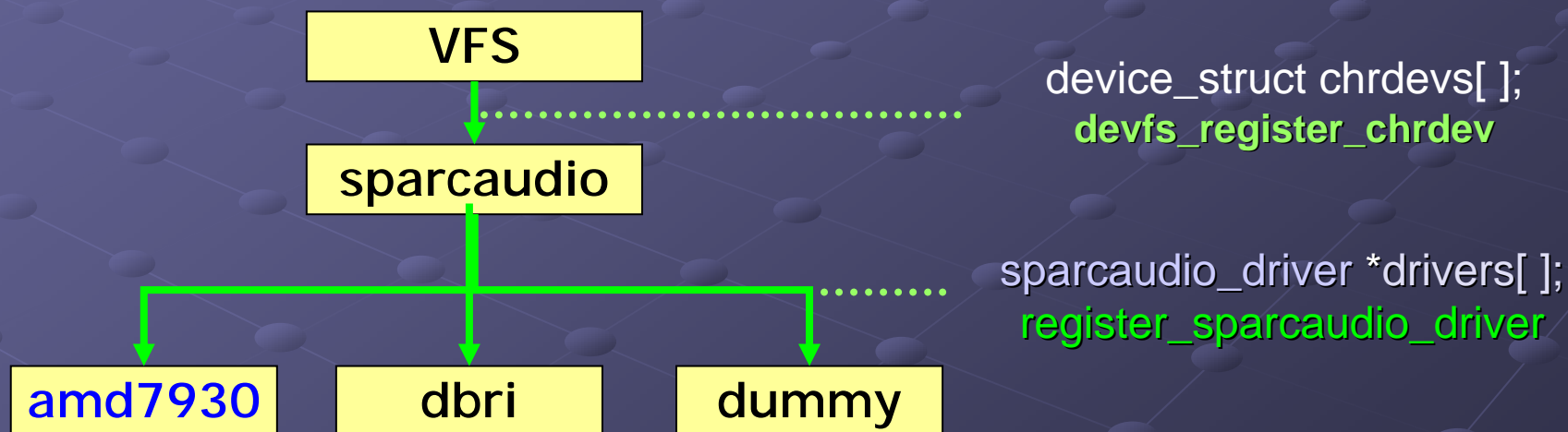
register_sparcaudio_driver
unregister_sparcaudio_driver



open
release
ioctl
start_output
stop_output
start_input
stop_input
... ..

amd7930

- 比sparcaudio底层的一个模块
- “模块栈”



amd7930

- static struct sparcaudio_driver drivers[MAX_DRIVERS];
- 定义一组sparcaudio_operations
 - amd7930_open、amd7930_release、amd7930_ioctl、amd7930_start_output、.....
- Init_module函数：amd7930_init
 - 探测sbus总线上的AMD7930芯片（音频信道），给每个探测到的芯片从上面的drivers数组中分配一个数据结构，然后调用amd7930_attach

amd7930_init

amd7930_attach



amd7930

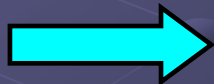
```
static int amd7930_attach(struct sparcaudio_driver *drv, int node,
                          struct sbus_bus *sbus, struct sbus_dev *sdev)
{ // 初始化amd7930和向上层sparcaudio注册
    drv->private = kmalloc(sizeof(struct amd7930_info), GFP_KERNEL);
    drv->ops = &amd7930_ops;      // 操作函数
    // 寄存器映射到内存空间
    info->regs = sbus_ioremap(resp, 0, regs.reg_size, "amd7930");
    // 初始化硬件
    sbus_writeb(AMR_DLC_EFCR, info->regs + CR);
    // 注册中断向量
    request_irq(info->irq, amd7930_interrupt, SA_INTERRUPT, "amd7930", drv)
    // 向上层注册sparcaudio_driver 结构
    err = register_sparcaudio_driver(drv, 1);
}
```

驱动程序的安装

.o文件

ELF头	
section头表	
sections	.strtab
	.symtab
	.data.init
	.dynamic
	.text.init
... ..	

insmod



module映象

module 结构	syms
	deps
	init
	kallsyms_start

sections	.strtab
	__kallsyms
	.kmodtab
	.symtab
	.data.init
	.text.init

init_module
系统调用



.o文件到module映象的转化

● 链接

- 模块中尚未定义的符号进行赋值，关键：在内核符号表中找到相应的符号

● module映象中特殊数据结构的构造

- .o文件的section并没有模块依赖数组dep等模块特有的数据结构，关键：建造数据结构并加入模块映象中

● 重定位

- 模块映象要求是连续的，最后需要重新组织各“section”的位置，传给sys_init_module时要求是已重定向的映象

Insmod的工作过程（工作于用户态）

- 1 通过系统调用query_module取得kernel中已注册的modules的信息，从内核空间拷贝至用户态，放入module_stat 结构数组中

```
▪ struct module_stat {  
▪     char *name;           // 模块名  
▪     unsigned long addr;    // 模块module结构在内核中的起始地址  
▪     unsigned long modstruct; // module结构大小  
▪     unsigned long size;    // 模块映象的大小  
▪     unsigned long flags;   // 标志  
▪     long usecount;        // 使用计数  
▪     size_t nsyms;         // 符号个数，指示syms数组大小  
▪     struct module_symbol *syms; // 符号数组，数组已拷贝至用户态  
▪     size_t nrefs;        // 应用计数  
▪     struct module_stat **refs; // 引用模块的module_stat结构的指针数组  
▪     unsigned long status;   // 用于指示是否依赖该模块  
▪ };
```

注：所有module_stat中的指针都指向用户态，所有已注册的模块信息将保存在一个module_stat结构数组中，为下一步的链接做准备

Insmod的工作过程（工作于用户态）续1

- 2 检查是否已有同名的module，只需跟module_stat数组的每一个元素中的name相比较即可判断
- 3 将.o文件读入到struct obj_file结构中，要求.o文件格式是ELF格式

```
struct obj_file
{
    ElfW(Ehdr) header;
    ElfW(Addr) baseaddr;
    struct obj_section **sections;
    struct obj_section *load_order;
    struct obj_symbol *symtab
        [HASH_BUCKETS];
}
```

```
struct obj_section
{
    ElfW(Shdr) header;
    const char *name;
    char *contents;
    struct obj_section *load_next;
    int idx;
}
```

Insmod的工作过程（工作于用户态）续2

● 4 对.o文件中未定义的符号指定其符号值（链接）

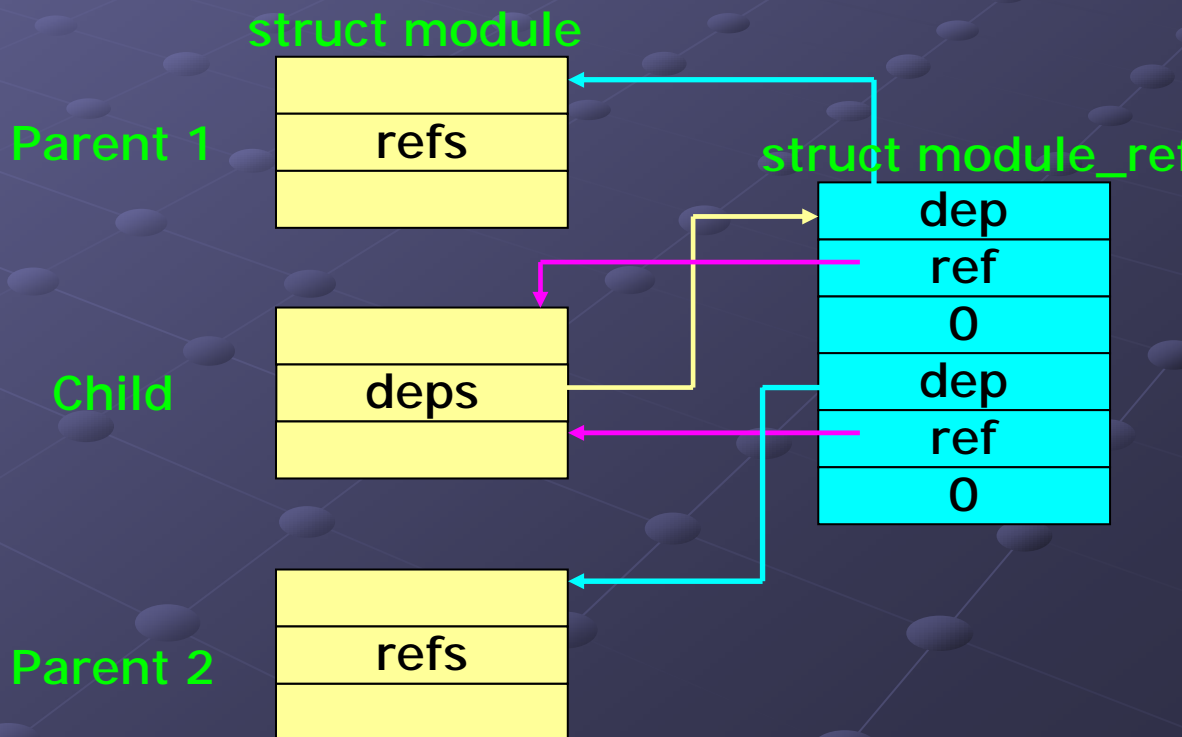
- .o文件已读入struct obj_file结构中
- 内核中各模块移出符号也已存入module_stat结构数组的符号表中
- 查找module_stat结构数组的符号表，将.o文件未定义的值填上
- 如果引用到某个模块的符号，那么该模块的module_stat结构中的status置1，表示依赖于该模块，后面构造模块依赖数组dep时有用

● 5 在用户态申请一个module结构，并将结构作为一个名为“.this”的section链入struct obj_file结构中，并增加一个符号“__this_module”，最后将作为模块映象的头

Insmod的工作过程（工作于用户态）续3

- 6 再一次检查是否还有没定义的symbol
- 7 在struct obj_file结构创建并加入“.kmodtab”节，依赖的模块已由module_stat的status域指示（为1则表示依赖），模块的地址由module_stat的addr域指示

注：.kmodtab节存放模块的依赖数组，由于引用关系refs在内核空间，在用户空间无法设置，所以next_ref域置为0，其设置推到调用系统调用sys_init_module进行设置



Insmod的工作过程（工作于用户态）续3

- 8 如需要创建并加入“__archdata”节和“__kallsyms”
- 9 至此可以计算module映象的大小m_size，即是struct obj_file结构的每一节大小的总和，不包括节的头部，调用系统调用create_module()，传入模块名和映象大小m_size，该系统调用在内核空间申请一个大小m_size的空间，并链入module_list中
- 10 非常重要的一步：重定位
 - 每一节并不是连续的，而最终传给内核的将是一个连续的模块映象
 - 修改每一节中涉及到地址的部分
 - 重定位module文件中.text中的地址

Insmod的工作过程（工作于用户态）续4

- **11** 初始化在用户空间申请的module结构，module->init和module->cleanup也是在这时设置的
- **12** 到此可以建构module映象了，^_^
 - struct obj_file结构有两个域：
 - struct obj_section **sections; // 所有的section指针数组
 - struct obj_section *load_order; // 指向第一个section：.this
 - 申请一块空间，用于存放module映象，从load_order开始遍历，将所有section的contents所指的空间拷贝至模块的映象中
 - 调用系统调用sys_init_module，传入模块名和模块映象
 - section主要有：.this、__ksymtab、.kmodtab、__ex_table、.text.init、data.init、__archdata、__kallsyms等

驱动程序的使用

- 主动方式：应用程序调用操作函数open，read，write，ioctl对设备进行操作
- 被动方式：中断方式



谢谢！