课程内容

日期	知识模块	知识点
2月20日	课程介绍	Linux 内核基本结构、Linux的历史、开源基础知识简介、驱动程序介绍
2月27日	实验课一	内核编译，内核补丁
3月5日	内核编程基础知识概述	内核调试技术、模块编程、源代码阅读工具、linux启动过程、git简介
3月12日	实验课2	内核调试
3月19日 3月26日	进程管理与调度	Linux进程基本概念、进程的生命周期、进程上下文切换、Linux 进程调度策略、调度算法、调度相关的调用
4月2日	实验课3	提取进程信息
4月9日	系统调用、中断处理	系统调用内核支持机制、系统调用实现、Linux中断处理、下半部
4月16日	实验课4	添加系统调用、显示系统缺页次数
4月23日	内核同步	原子操作、自旋锁、RCU、内存屏障等linux内核同步机制
4月30日	内存管理1	内存寻址、Linux物理内存和虚拟内存的组织、伙伴系统、vmalloc
5月14日	内存管理2	Slab分配器、进程地址空间
5月21日	实验课5	观察内存映射、逻辑地址与物理地址的对应
5月28日	文件系统	Linux虚拟文件系统、Ext2/Ext3/Ext4文件系统结构与特性
6月4日	Linux设备驱动基础字符设备驱动程序设计	Linux设备驱动基础、字符设备创建和加载、字符设备的操作，IOCTL、阻塞IO、异步事件等
6月11日	报告课	期末课程报告

Linux Device Driver Programming

Introduction and Char Drivers

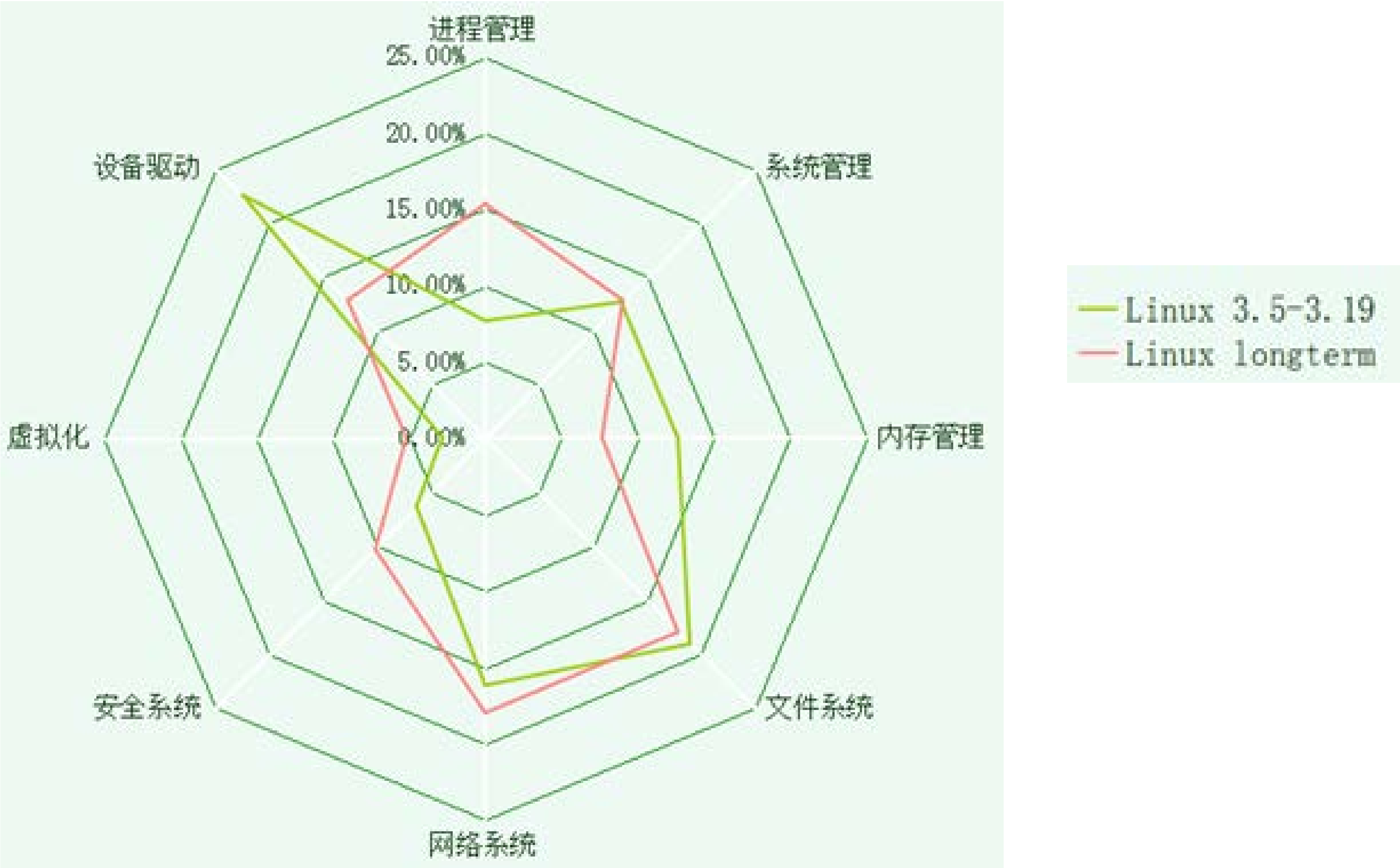
荆琦
jingqi@pku.edu.cn
北京大学软件与微电子学院

Agenda

1. An Introduction to Device Drivers
2. The Linux Device Model
3. Linux Device I/O
4. Char Drivers

1. An Introduction to Device Drivers

Linux 3.5到3.19和longterm版本变更**特性数**占总特性数比例对比图



为什么学习驱动程序开发？

- 设备驱动开发是**Linux**开发的热门领域之一
- 驱动程序开发门槛较高，驱动程序开发人员相对较少
- 帮助你更好的了解整个**Linux**系统的工作过程

如何学好驱动开发？

- 编写驱动程序应掌握以下方面知识
  - **Linux**驱动框架及规范
  - **Linux**内核知识
    - 需要掌握内核中自旋锁、信号量、完成变量、中断顶／底半部、定时器、内存和**I／O**映射以及异步通知、阻塞／非阻塞、**I／O**等大量理论知识
  - 硬件知识

为什么需要编写驱动程序？

- 单片机无操作系统
  - 用户程序中直接通过指令访问、操作硬件
- 单片机有简单操作系统（**μc/os**）
  - 将驱动程序封装为独立的模块
  - 用户程序直接调用相应的函数
- **Linux**操作系统
  - **Linux**分为内核空间与用户空间
  - 用户如果想访问硬件必须通过内核函数来完成

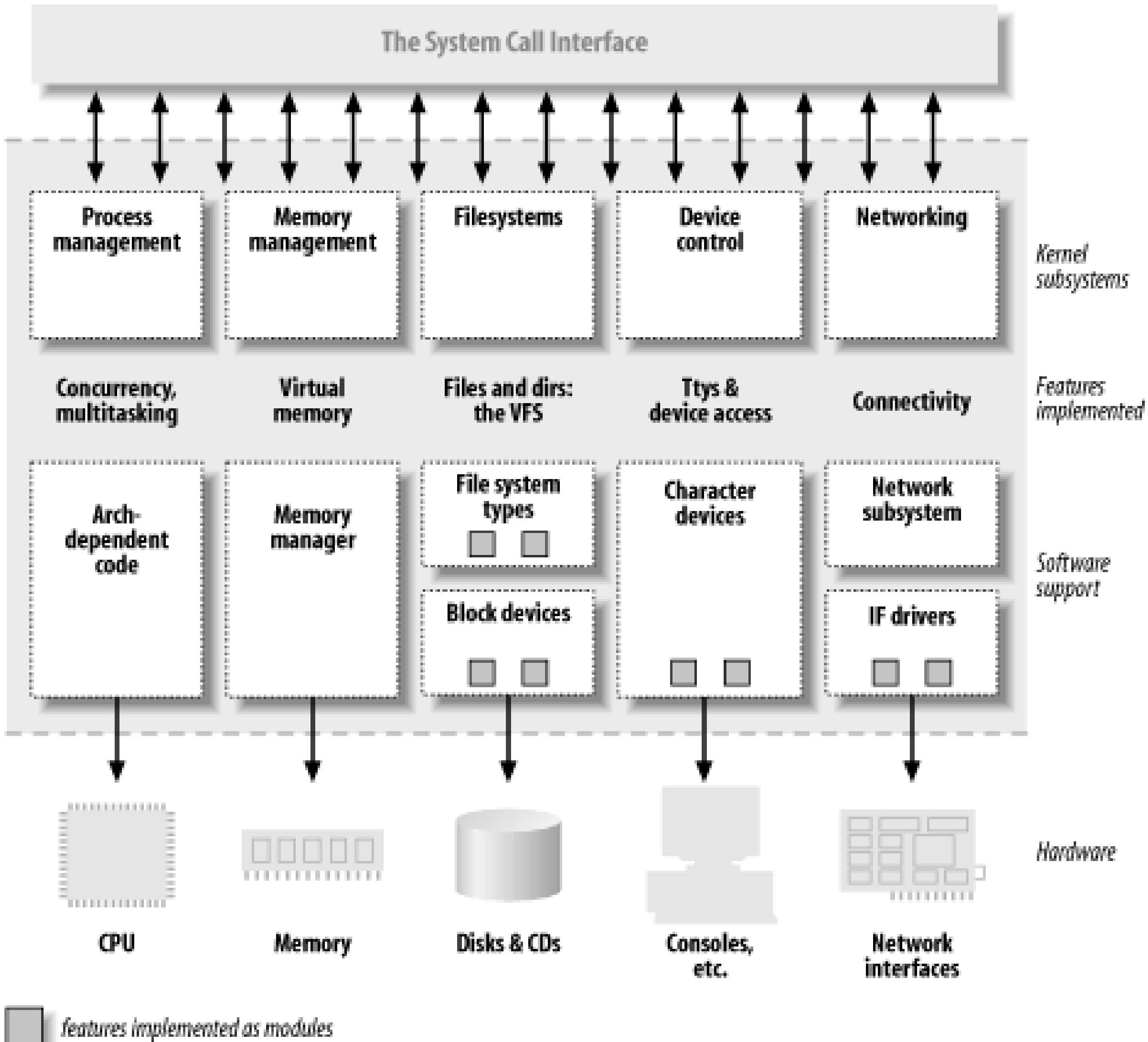
驱动程序介绍

- 驱动程序是硬件设备与应用程序之间的一个软件层
  - 用户通过一组**与具体设备无关**的标准化的系统调用来完成相应的操作，完全隐蔽了设备的工作细节
  - 驱动程序将这些系统调用**映射**到具体设备对于实际硬件的特定操作上
- 驱动程序是内核的一部分，可以采用**静态编译**或**动态加载**模块的方式来安装驱动程序进内核

驱动程序的角色

- 只是提供机制，不是策略
  - 机制： “what capabilities are to be provided” (the mechanism) 相对稳定
  - 策略：“how those capabilities can be used” (the policy). 相对多变
  - The driver should deal with making the hardware available, leaving all the issues about how to use the hardware to the applications.

Figure 1-1. A split view of the kernel



设备分类(1)

- 字符设备（Character devices）
  - 这类设备可以像一个文件一样被打开、读写、关闭
  - 例如：文本控制台设备 */dev/console* & 串口设备 */dev/ttyS0*，键盘、打印机、调制解调器等
  - 和普通文件的区别：普通文件可以使用lseek等操作“来回”读数据，而“大部分”字符设备只是一个内核和用户空间的数据通道，这样的设备一般情况下只能顺序读取数据。当然也有某些字符设备看起来是一个数据区域，在这个区域内， mmap 和lseek可以来回移动读数据，这样的设备一般是对“数据帧”或成块的内存进行处理，比如视频数据
  - 在/dev目录下有文件与之对应

设备分类(2)

- 块设备(Block devices)
  - 块设备是能“容纳”文件系统的设备，比如磁盘、Flash等
  - 通常块设备只能处理以块为单位的数据操作，通常块的大小是512字节或者其整数倍。Linux块设备有点特殊之处在于可以对其以字节单位读取，这样的话，块设备＆字符设备的在驱动程序的角度最大的区别是其在内核中的内部数据管理方式和接口的不同
  - 在/dev目录下有文件与之对应

设备分类(3)

- 网络接口(network interface)
  - 网络接口可以是硬件接口，也可以是纯软件接口
  - 通过内核网络部分相关代码驱动网络接口发送和接收数据包
  - 网络接口不在/dev目录下出现，而是以某个系统中唯一的名字出现，比如eth0

设备文件

- 在**/dev** 目录下除了字符设备和块设备节点之外通常还会存在：**FIFO**管道、**Socket**、软**/**硬连接、目录，每个设备在**/dev** 目录下都有一个对应的文件**(节点)**，这个文件称为**设备文件**

- ls -l

```
crw-rw-rw-    1 root     root       1,   3 Apr 11  2002 null
crw-------    1 root     root      10,   1 Apr 11  2002 psaux
crw-------    1 root     root       4,   1 Oct 28 03:04 tty1
crw-rw-rw-    1 root     tty        4,  64 Apr 11  2002 ttys0
crw-rw----    1 root     uucp       4,  65 Apr 11  2002 ttyS1
crw--w----    1 vcsa     tty        7,   1 Apr 11  2002 vcs1
crw--w----    1 vcsa     tty        7, 129 Apr 11  2002 vcsa1
crw-rw-rw-    1 root     root       1,   5 Apr 11  2002 zero
```

设备号

- 每个字符设备和块设备都必须有主、次设备号
- 主设备号相同的设备使用**相同的驱动程序**，次设备号用于**区分具体设备的实例**
- 主设备号和次设备号能够唯一地标识一个设备
- 可以通过**cat /proc/devices**命令查看当前已经加载的设备驱动程序的**主设备号**
- 在**/dev**目录下的**FIFO**管道、**Socket**、软**/**硬连接、目录，**没有主/次设备号**

/proc/devices *的例子*

Character devices:
1 mem
2 pty
3 ttyp
4 ttyS
6 lp
7 vcs
10 misc
13 input
14 sound
21 sg
180 usb
Block devices:
2 fd
8 sd
11 sr
65 sd
66 sd

列出**字符和块设备**的**主设备号**，以及分配到这些设备号的**设备名称**

This file displays the various character and block devices currently configured for use with the kernel. **It does not include modules that are available but not loaded into the kernel**.

For more information about devices see **Documentation/devices.txt**, especially about the management of device number

动态加载模块

```
# Makefile for hello.c

obj-m:=hello.o
KDIR:=/lib/modules/2.6.15-1.2054_FC5/build
PWD:=$(shell pwd)

default:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

.PHONY:clean
clean:
	rm -f *.o *~ *.ko
```

可以使用**insmod**或**modprobe**命令来加载驱动

例如：　**#insmod hello.ko**

　或　**#modprobe hello**

静态编译

- 将驱动程序增加到内核源码中，通过**Kconfig**对驱动进行配置。
- 内核源码树的目录下都有两个文档**Kconfig**和**Makefile**。
  - 分布到各目录的**Kconfig**构成了一个分布式的内核配置数据库，每个**Kconfig**分别描述了所属目录源文档相关的内核配置菜单。
  - 在内核配置**make menuconfig(**或**xconfig**等**)**时，从**Kconfig**中读出菜单，用户选择后保存到**.config**的内核配置文档中。

简单的HelloWorld.c程序

- 修改相应目录下的**config**文件和**Makefile**文件
  - **#config**文件

config HELLOWORLD

bool “helloworld”

help

....

最常见的是**tristate**和**bool**，分别对应于配置界面中[ ]和< >选项

tristate：可取y、n、m
bool：可取y、n
string：取值为字符串
hex：取值为十六进制数据
int：取值为十进制数据

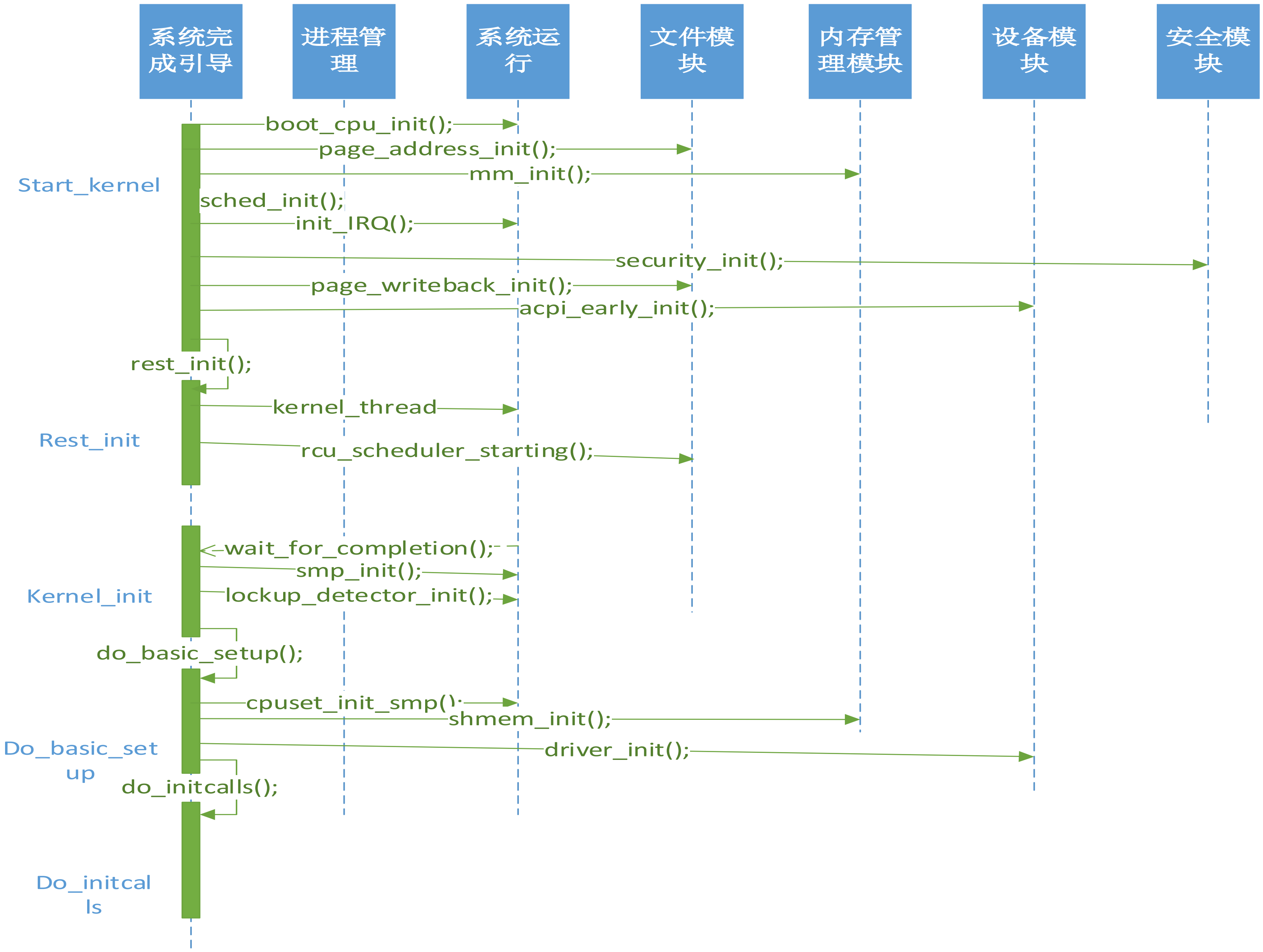
  - **#Makefile**文件

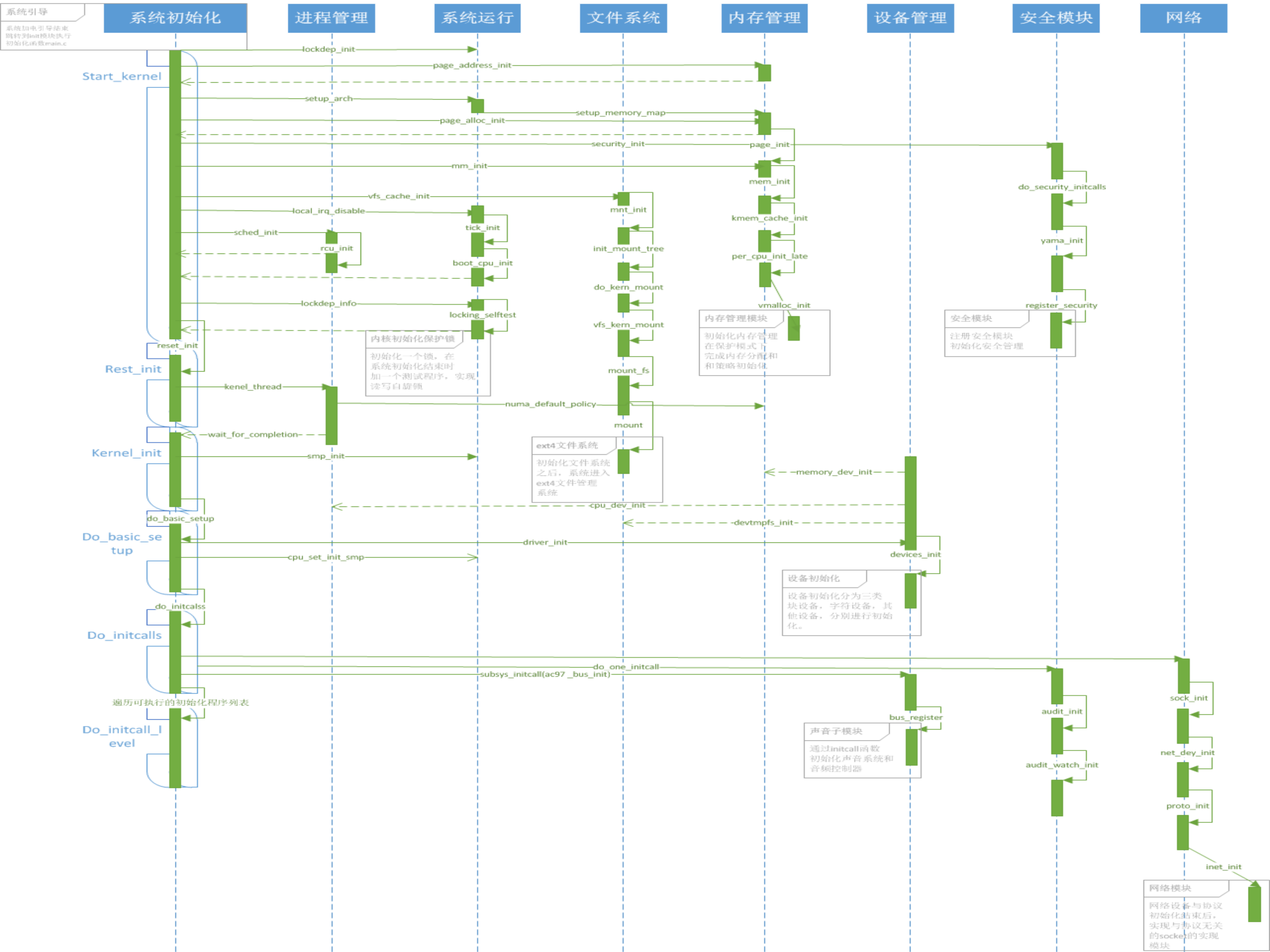
obj-$(CONFIG_HELLOWORLD)+=HelloWorld.o

驱动程序注册时机

- 如果设备驱动程序被**静态编译**进内核，则注册发生在**内核初始化**阶段
- 如果作为一个内核**模块**来编译，则在**装入模块**的时候注册（并在卸载模块时注销）

Linux 内核 x86 引导的主要函数流程




系统引导
系统加电引导结束
跳转到init模块执行
初始化函数main.c
系统初始化
进程管理
系统运行
文件系统
内存管理
设备管理
安全模块
网络
Start_kernel
lockdep_init
page_address_init
setup_arch
setup_memory_map
page_alloc_init
security_init
page_init
mm_init
mem_init
do_security_initcalls
vfs_cache_init
mnt_init
local_irq_disable
kmem_cache_init
tick_init
sched_init
rcu_init
yama_init
init_mount_tree
boot_cpu_init
per_cpu_init_late
do_kern_mount
lockdep_info
vmalloc_init
register_security
locking_selftest
内存管理模块
初始化内存管理
在保护模式下
完成内存分配和
和策略初始化
安全模块
注册安全模块
初始化安全管理
vfs_kern_mount
内核初始化保护锁
初始化一个锁，在
系统初始化结束时
加一个测试程序，实现
读写自旋锁
reset_init
Rest_init
mount_fs
kenel_thread
numa_default_policy
mount
wait_for_completion
ext4文件系统
初始化文件系统
之后，系统进入
ext4文件管理
系统
Kernel_init
smp_init
memory_dev_init
cpu_dev_init
devtmpfs_init
do_basic_setup
Do_basic_setup
driver_init
devices_init
cpu_set_init_smp
设备初始化
设备初始化分为三类
块设备，字符设备，其
他设备，分别进行初始
化。
do_initcalss
Do_initcalls
do_one_initcall
subsys_initcall(ac97_bus_init)
sock_init
audit_init
遍历可执行的初始化程序列表
bus_register
Do_initcall_level
声音子模块
通过initcall函数
初始化声音系统和
音频控制器
net_dey_init
audit_watch_init
proto_init
inet_init
网络模块
网络设备与协议
初始化结束后，
实现与协议无关
的socket的实现
模块

- 用户对设备的操作一般通过文件访问操作来完成。
  - 文件的调用包含**open**、**read**、**write**、**close**、**ioctl**等。
  - 需要在用户程序的**API**调用与驱动程序之间建立起关系。

2. The Linux Device Model

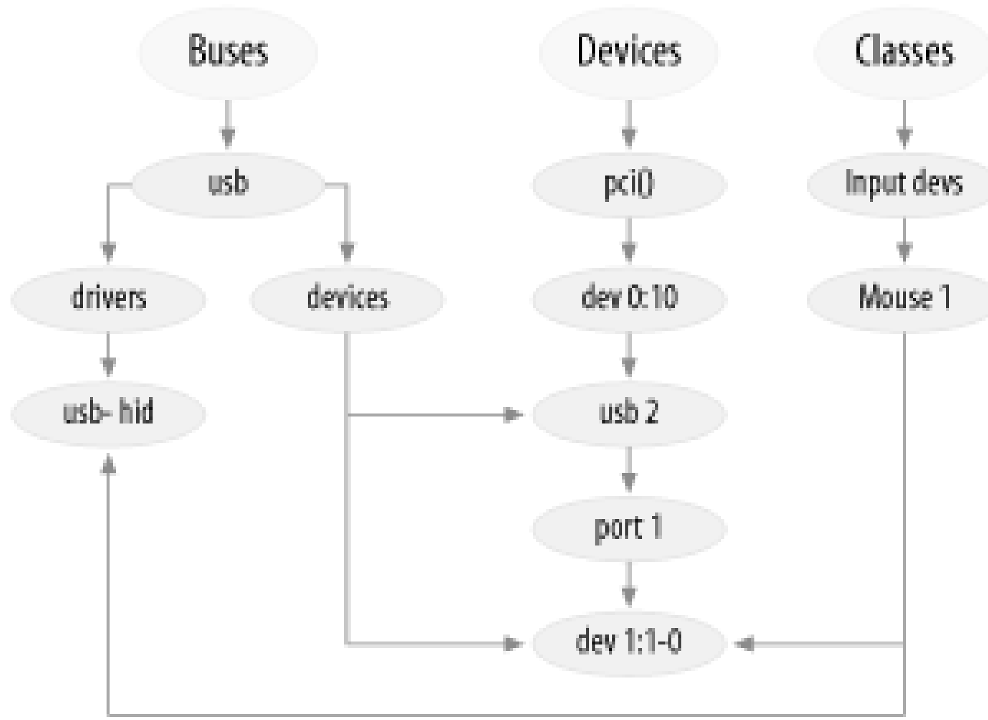
Linux 设备模型

设备模型是使用面向对象的思想(封装、继承、多态)抽象出来的产物，主要功能及特点:

- 提供设备**引用计数**
- 观察设备**状态**
- 对设备进行分类，并以**树**的形式展示系统的所有设备，电源管理；按照正确的**顺序** power down 设备
- 通过 sysfs 和用户空间进行**交换数据**，可以使用命令 tree 观察 /sys 目录
- 结构复杂

```
pku@pku-laptop:/sys$ ls
block  bus  class  devices  firmware  kernel  module  power
pku@pku-laptop:/sys$
```

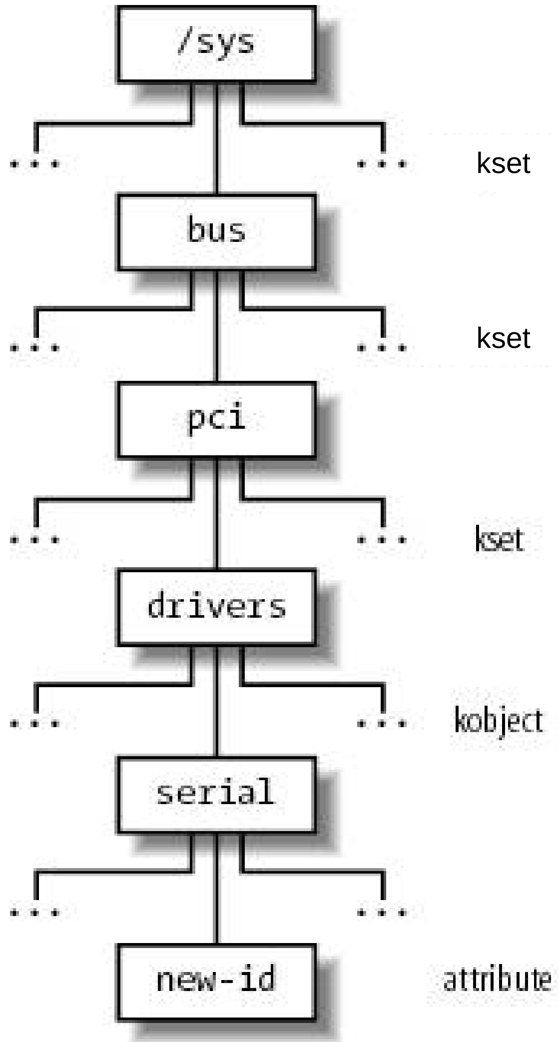
```
   |   |   |-- destroy_node
   |   |   |-- devices
   |   |   |-- drivers
   |   |   |   `-- sbp2
   |   |   |       |-- bind
   |   |   |       |-- device_ids
   |   |   |       |-- name
   |   |   |       `-- unbind
   |   |   |-- ignore_drivers
   |   |   `-- rescan
   |   |-- pci
   |   |   |-- devices
   |   |   |   |-- 0000:00:00.0
   |   |   |   |-- 0000:00:01.0
   |   |   |   |-- 0000:00:07.0
   |   |   |   |-- 0000:00:07.1
   |   |   |   |-- 0000:00:07.2
   |   |   |   |-- 0000:00:07.3
   |   |   |   |-- 0000:00:0f.0
   |   |   |   |-- 0000:00:10.0
   |   |   |   |-- 0000:00:11.0
   |   |   |   `-- 0000:00:12.0
   |   |   |-- drivers
   |   |   |   |-- ENS1371
```



Buses, Devices, Drivers, and Classes

- Buses
  - A channel between processor and devices
  - Represented by the bus_type structure
  - The match method tests if a device can be handled by a driver
  - The hotplug method generates an event of this kind of bus
- Devices
  - Every device in Linux is represented by the struct device
  - The device's "parent" device is some sort of bus or host controlller
  - bus_id[] is a string that uniquely identifies the device on the bus
- Drivers
  - The driver core tracks all known drivers to match up new devices
  - Drivers are defined by the **struct device_driver**
  - The probe method is used to query the existence of a device
  - The shutdown method is called to quiesce the device
- Classes
  - An abstraction by functions (i.e. /sys/class/net)
  - the best way of exporting information to user space

An example of device driver model hierarchy



- The *bus* kset includes a *pci* kset, which, in turn, includes a *drivers* kset. This kset contains a *serial* kobject corresponding to the device driver for the serial port having a single *new-id* attribute.

重要数据结构 Kobject

- 在内核源代码目录<linux/kobject.h>中定义

```
struct kobject {
    const char    *name;
    struct list_head   entry;
    struct kobject    *parent;
    struct kset       *kset;
    struct kobj_type   *ktype;
    struct kernfs_node *sd;
    struct kref       kref;
  #ifdef CONFIG_DEBUG_KOBJECT_RELEASE
    struct delayed_work    release;
  #endif
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
  };
```

Kobject一般要嵌入到其他设备数据结构中，比如 struct cdev 结构中有 kobject

```
struct kref {
        atomic_t refcount;
};
```

Kernel Event

- The Kernel Event Layer implements a kernel-to-user notification system on top kobjects
  - int kobject_uevent(struct kobject *kobj, enum kobject_action action);
- Internally, the kernel events go from kernel-space out to user-space via netlink. Netlink is a high-speed multicast socket used to transmit networking information. Using netlink means that obtaining kernel events from user-space is as simple as blocking on a **socket**.
  - socket(**AF_NETLINK**, SOCK_RAW, netlink_type)
- When udev receives the uevent message (through netlink)
  - Match rules in /etc
  - Execute the shell cmd
    - Upload/unload the driver module
    - Create/remove device file in dev directory

Kobj_type

- < include/linux/kobject.h>

```
struct kobj_type {
        void (*release)(struct kobject *kobj);
        const struct sysfs_ops *sysfs_ops;
        struct attribute **default_attrs;
        const struct kobj_ns_type_operations
                *(*child_ns_type)(struct kobject *kobj);
        const void *(*namespace)(struct kobject *kobj);
};
```

引用计数

- 操作函数
  - struct kobject *kobject_get(struct kobject *kobj);

    引用计数增加 1
  - void kobject_put(struct kobject *kobj);

    引用计数减少 1

如何使用 kobject

- 静态初始化：自己指定ktype
  - memset() kobject 内存区为 0
  - **对kobject的一些域进行设置,主要是ktype，设置ktype的release实现方法**
  - 调用 kobject_init(*kobj)函数
- 动态初始化：默认的ktype
  - 直接调用 kobject_create(void)
  - **ktype采用默认类型，默认ralease方法**
  - 调用 kobject_init(*kobj)函数

kset

- **Kset： 聚合多个 kobject 对象：**

<linux/kobject.h>:

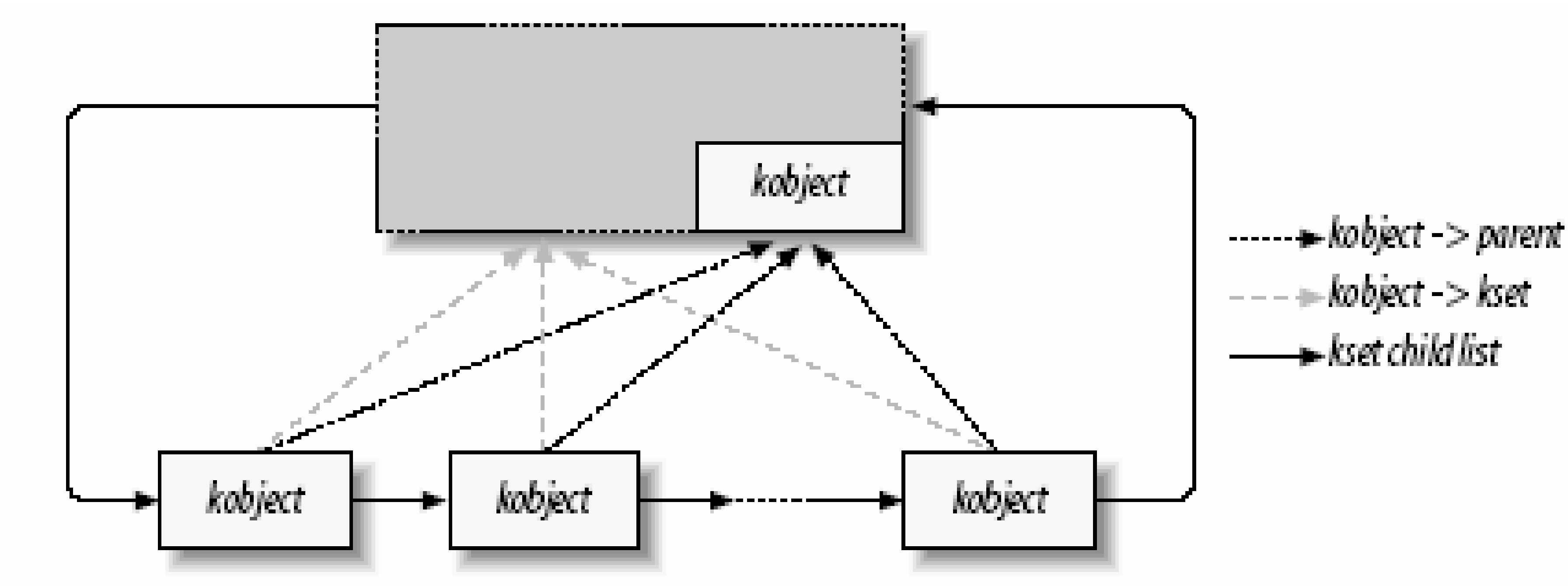
```
struct kset {
        struct list_head list;
        spinlock_t list_lock;
        struct kobject kobj;
        const struct kset_uevent_ops *uevent_ops;
};
```

kobj is a kobject representing the base class for this set
uevent_ops points to a structure that describes the hotplug behavior of kobjects in this kset.

Cont.

```
/**
 * struct kset - a set of kobjects of a specific type, belonging to a
* specific subsystem. A kset defines a group of kobjects.  They can be
*  individually  different “types” but overall these kobjects all want to be
* grouped together and operated on in the same manner.  ksets are
*used to define the attribute callbacks and other common events that
*happen to a kobject.
 * @list: the list of all kobjects for this kset
 * @list_lock: a lock for iterating over the kobjects
 * @kobj: the embedded kobject for this kset (recursion, isn't it fun...)
 * @uevent_ops: the set of uevent operations for this kset.  These are
 * called whenever a kobject has something happen to it so that the
 * kset  can add new environment variables, or filter out the uevents if
 * so desired.
*/
```

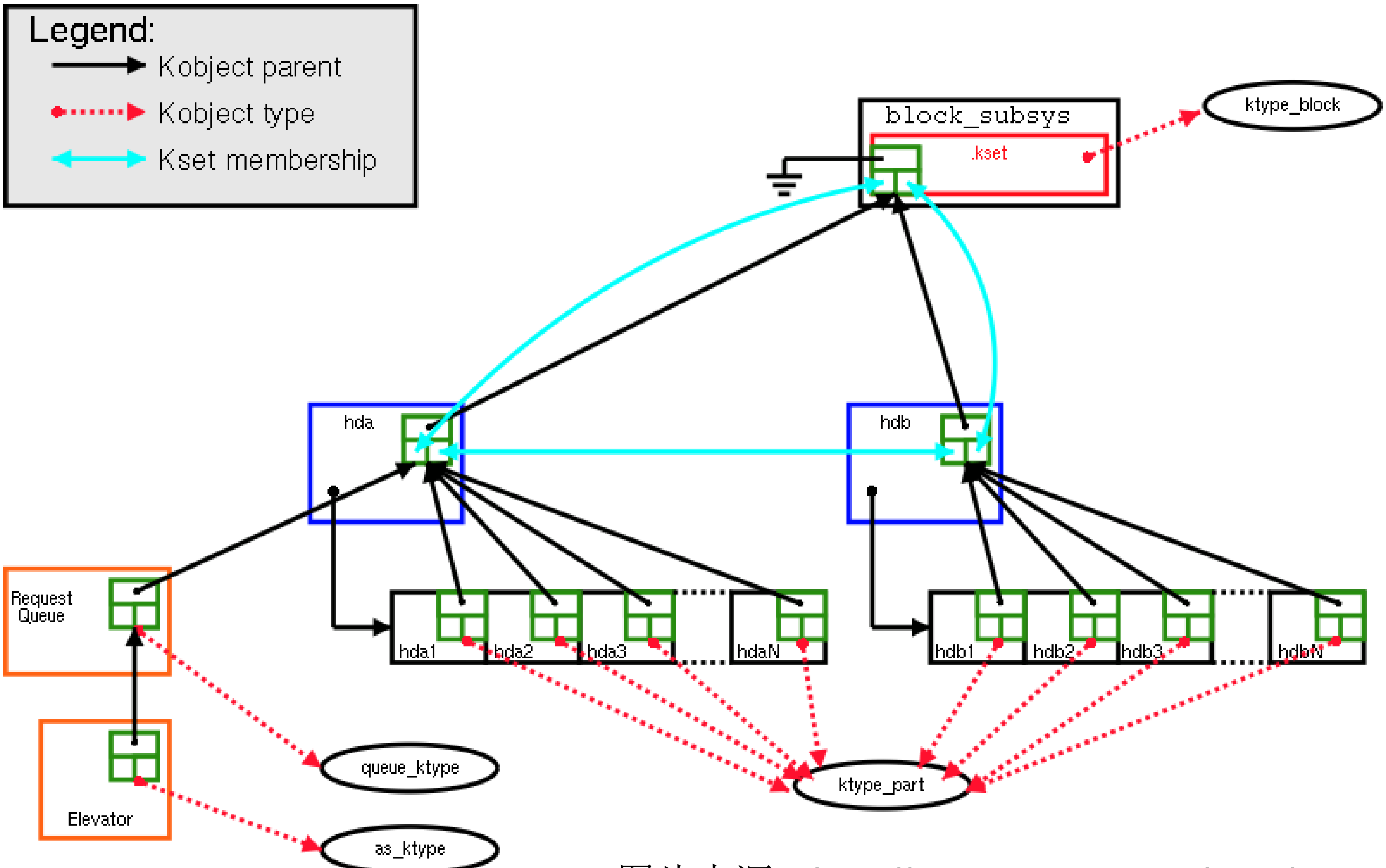
Kset 和 kobject 关系



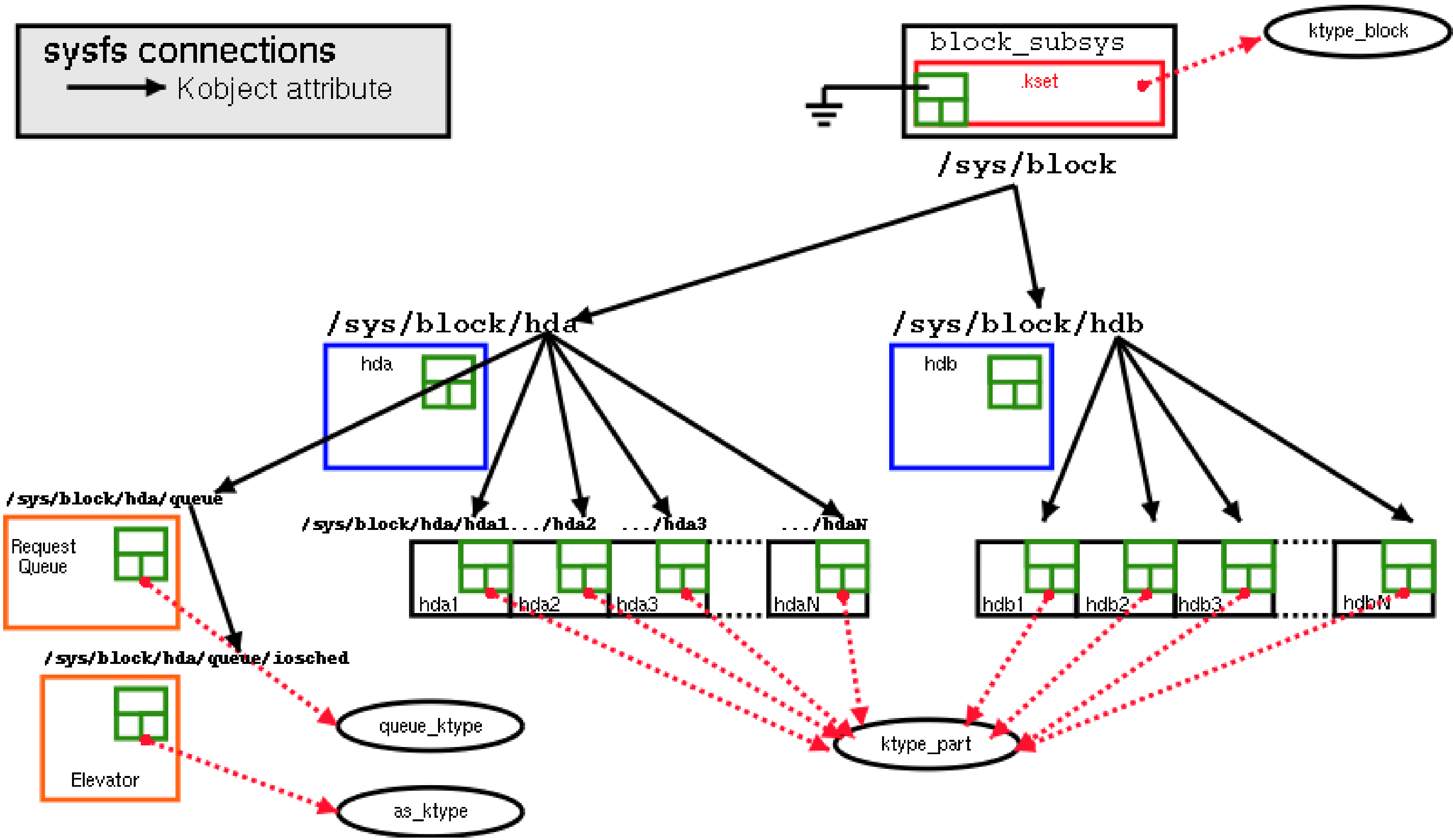
ksets

- 往 kset 添加 kobject
  - kobject_add(struct kobject *kobj); 添加 kobject 到sysfs
  - kobject_register( ) 调用 kobject_init( ) 和 kobject_add( )
- 从kset 移除 kobject
  - kobject_del( )
  - kobject_unregister( ) 调用kobject_del( ) 和kobject_put( )
- 添加到 kset 的 kobject 的位置
  - If the kobject's **parent** pointer is set, the kobject will map to a **subdirectory** in sysfs inside its parent.
  - If the parent pointer is not set, the kobject will map to a subdirectory inside **kset->kobj**.
  - If neither the parent nor the kset fields are set in the given kobject, the kobject is assumed to have no parent and will map to a **root-level directory** in sysfs.
  - The name of the directory representing the kobject in sysfs is given by kobj->name.

Block subsystem的例子



图片来源：http://ant.comm.ccu.edu.tw/



图片来源：http://ant.comm.ccu.edu.tw/course/4305077_Driver/0_Lectures/14_LinuxDeviceModel.ppt

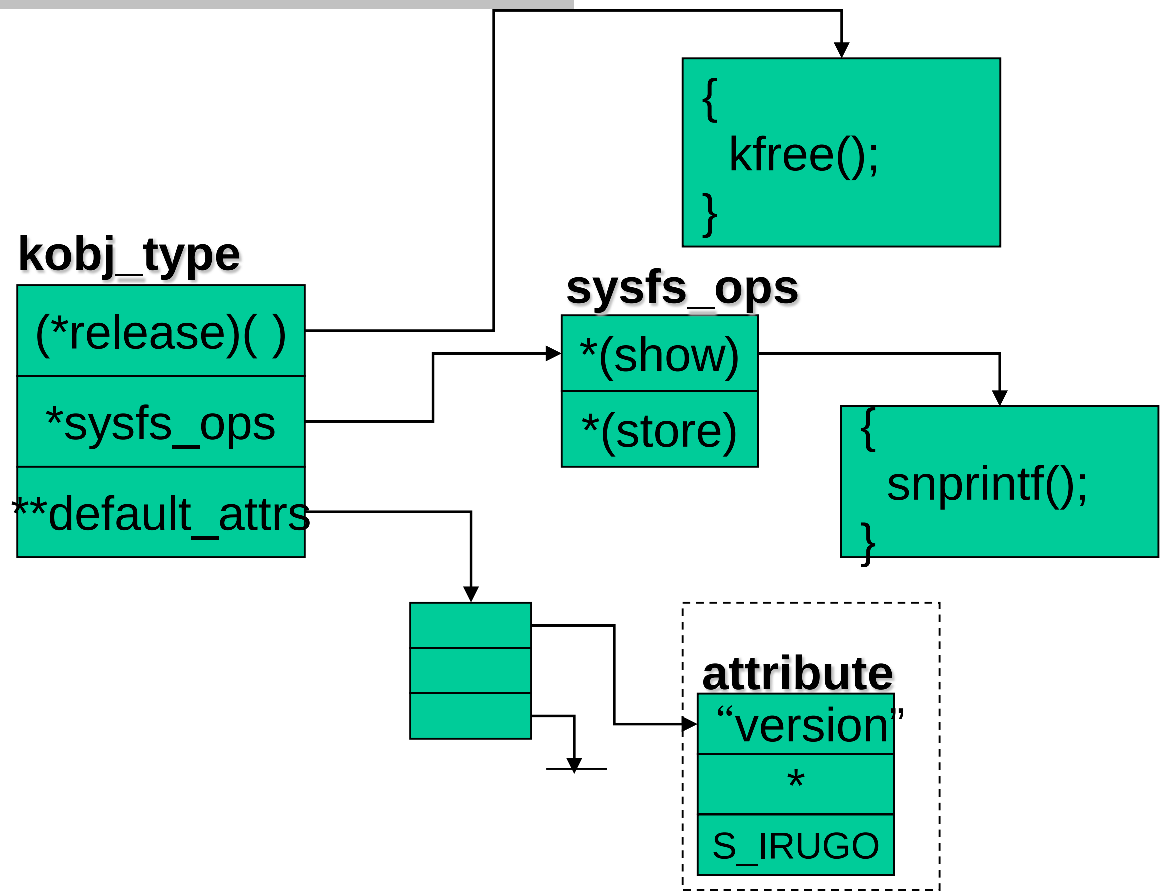
sysfs 中的kobject的属性

- **kobject** 要导出属性，以文件的形式表现，文件内的值，是属性的值，一般一个属性和一个值对应
- 相关结构和操作在头文件**<linux/sysfs.h>**中定义

```
struct attribute {
  const char *name;
  mode_t mode;
};
 struct sysfs_ops {
     ssize_t  (*show)(struct kobject *, struct attribute *,char *);
    ssize_t  (*store)(struct kobject *,struct attribute *,const char *,
    size_t);
};
```

对象指针关系

The show() method is invoked on read.
The store() method is invoked on write.
The size of the buffer is always PAGE_SIZE or smaller.



图片来源：http://ant.comm.ccu.edu.tw/course/4305077_Driver/0_Lectures/14_LinuxDeviceModel.ppt

Low-level sysfs operations

- 添加删除属性
  - static inline int __must_check **sysfs_create_file**(struct kobject *kobj, struct attribute *attr);
  - static inline void **sysfs_remove_file**(struct kobject *kobj, struct attribute *attr);
- 也可以是符号link
  - int __must_check **sysfs_create_link**(*kobj, struct kobject *target, char *name);
  - void **sysfs_remove_link**(*kobj, char *name);

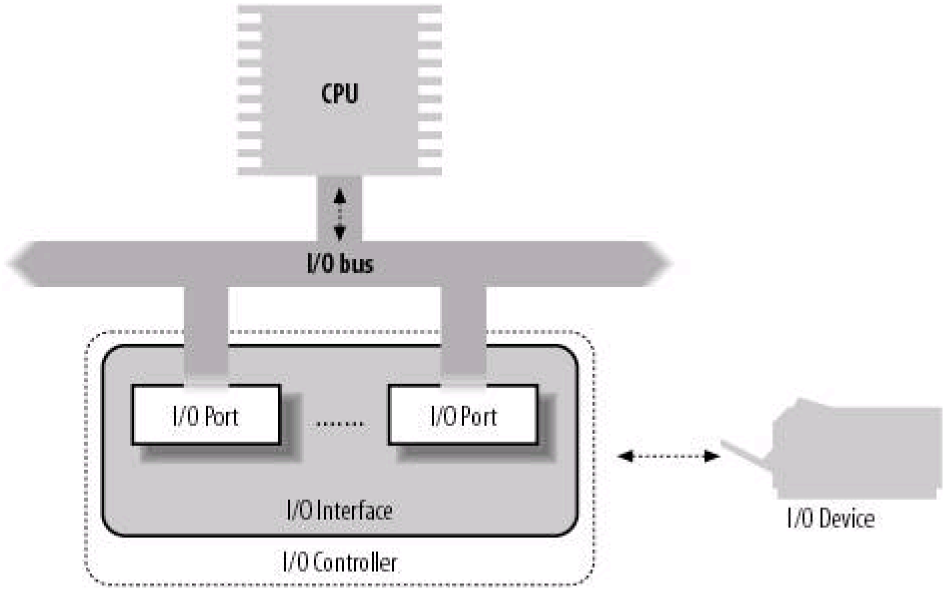
Kobject,ktype,kset,subsystem-summary

- Kobject
  - The fundamental structure of the device model
  - Track the **reference counting** of objects
  - **Hotplug event** generation
- Ktype
  - Store kobject's **release()** method
  - **Every** kobject has an associated ktype
  - **sysfs_ops** and **default_attrs**
- Kset
  - A container for kbojects with the **same ktype**
  - Represent a sysfs **directory**
  - Kset also contains its **own kobject**

3. Linux Device I/O

I/O bus

- **I/O bus**:The data path that connects a **CPU** to an **I/O device**. The 80 x 86 microprocessors use 16 of their address pins to address I/O devices and 8, 16, or 32 of their data pins to transfer data. The I/O bus, in turn, is connected to each I/O device by means of a hierarchy of hardware components including up to three elements: I/O ports , interfaces, and device controllers.



- I/O ports
  - Each device connected to the I/O bus has its own set of **I/O addresses**, which are usually called I/O ports.
  - I/O ports may also be **mapped into addresses of the physical address space**. The processor is then able to communicate with an I/O device by issuing assembly language instructions that operate directly on memory. Modern hardware devices are more suited to mapped I/O, because it is faster and can be combined with DMA.
  - I/O ports of each device are structured into a set of specialized registers
  - **detecting which I/O ports have been assigned** to I/O devices may not be easy--the kernel keeps track of I/O ports assigned to each hardware device by means of **resources**
- **I/O interface**
  - a hardware circuit inserted between a group of I/O ports and the corresponding device controller.
  - It acts as an interpreter that translates the values in the I/O ports into commands and data for the device. In the opposite direction, it detects changes in the device state and correspondingly updates the I/O port that plays the role of status register.
  - This circuit can also be connected through an IRQ line to a Programmable Interrupt Controller, so that it **issues interrupt requests** on behalf of the device.
- A complex device may require a **device controller** to drive it.

I/O端口和I/O内存

- **X86**处理器有独立的**I/O**空间；大多数嵌入式微控制器（如**ARM**、**PowerPC**等）不提供**I/O**空间，仅提供内存空间

- 设备通常会提供一组寄存器来对设备进行控制、读写，以及获取设备状态，这些寄存器可能会位于**I/O**空间中（**I/O**端口），也可能位于内存空间中（**I/O**内存）

- 习惯上也用**I/O**端口泛指**I/O**端口和**I/O**内存

- **I/O**端口是一种资源，系统用**cgroup**管理资源

分配I/O Port

- 分配I/O 端口
  - 避免资源冲突
  - /proc/ioports
- 操作函数
  - **#include <linux/ioport.h>**
  - **struct resource *request_region(ulong first, ulong size, const char *name)**
    - **Allocate n ports with first**
    - **Name is the name of the device**
    - **Return non-NULL on success**
    - **When allocation fails**
      - **Try other ports**
      - **Remove the device module using those ports**
  - **void release_region(ulong start, ulong size)**
  - **int __deprecated check_region(resource_size_t s,resource_size_t n)**
    - **check to see whether a given set of I/O ports is available**

```
[root@localhost char]# cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-006f : keyboard
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
02f8-02ff : serial
```

If you are unable to allocate a needed set of ports, above is the place to look to see who got them first

操作I/O Port

- **#include <asm-generic/io.h>**
- **8-bit**
  - **static inline u8 inb(unsigned long addr)**
  - **static inline void outb(u8 b, unsigned long addr)**
- **16-bit**
  - **static inline u16 inw(unsigned long addr)**
  - **static inline void outw(u8 b, unsigned long addr)**

32-bit
  - **static inline u32 inl(unsigned long addr)**
  - **static inline void outl(u32 b, unsigned long addr)**
- **No 64-bit port I/O operations are defined. Even on 64-bit architectures, the port address space uses a 32-bit (maximum) data path**

I/O memory

- I/O memory: memory-mapped registers and device memory
  - 例如：PCI I/O memory bar
- 最好不要直接访问
  - wrapper functions
- 分配 I/O memory
  - 避免地址冲突
  - /proc/iomem
- I/O memory 的操作
  - **include <linux/ioport.h>**
  - **struct resource *request_mem_region(ulong start, ulong n, char *name)**
  - **void release_mem_region(ulong start, ulong n);**
  - **int check_mem_region(ulong start, ulong n)**

  - **#include <asm-generic/io.h>**
  - **Void *ioremap(ulong phy_addr, ulong size)**
  - **Void *ioremap_nocache(ulong phy_addr, ulong size)**
  - **Void iounmap(void *addr)**

I/O memory 的访问

```
#/arch/x86/include/asm/io.h
build_mmio_read(readb, "b", unsigned char, "=q", :"memory")
build_mmio_read(readw, "w", unsigned short, "=r", :"memory")
build_mmio_read(readl, "l", unsigned int, "=r", :"memory")

#define build_mmio_read(name, size, type, reg, barrier) \
static inline type name(const volatile void __iomem *addr) \
{ type ret; asm volatile("mov" size " %1,%0":reg (ret) \
:"m" (*(volatile type __force *)addr) barrier); return ret; }

static inline unsigned char readb(const volatile void __iomem *addr)
```

4. Char Drivers

内核内部保存字符设备信息的结构struct cdev

- *<linux/cdev.h>*

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list; //字符设备文件的索引节点
    dev_t dev; //设备号
    unsigned int count; //该驱动管理的设备个数
} __randomize_layout;
```

对cdev的操作

- **静态内存定义初始化：**
  void cdev_init(struct cdev *, const struct file_operations *);

- **动态内存定义初始化：**
  struct cdev *cdev_alloc(void);

- **int cdev_add(struct cdev *, dev_t, unsigned);**
  - 字符设备注册到系统中

- **void cdev_del(struct cdev *);**
  - 删除设备

设备号的内部表示

- **<*linux/types.h*>** 中定义**dev_t**

  typedef __kernel_dev_t dev_t

  typedef __u32 __kernel_dev_t

  typedef unsigned int __u32

- **dev_t** 长度 **32-bit**， **12 bits** 设置为 **major number , 20 bits** 设置为**minor number.**
- 为了获得设备号，应该使用**<*linux/kdev_t.h*>**中定义的宏来获得：
  - **MAJOR(dev_t dev);**
  - **MINOR(dev_t dev);**
- 知道**major number**和**minor number**，可以得到**dev_t**类型数据：
  - **MKDEV(int major, int minor);**

在内核中分配设备号

- 如果程序员**提前知道要分配的设备号**，可以这样分配：

  int register_chrdev_region(dev_t first, unsigned int count, char *name);

  – **可以分配多个设备号**，声明于*<linux/fs.h>中*

  – First：内部表示的设备号起始号

  – Count：分配的设备号的数目，如果数目个数超过了minor number的表示范围，那么major number 继续增加

  – Name is the name of the device that should be associated with this number range; it will appear in */proc/devices* and sysfs.

释放设备号

- void unregister_chrdev_region(dev_t first, unsigned int count);

如何让内核自己分配设备号?

- int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);
  - dev: output-only, on successful completion, hold the first number in your allocated range.
  - firstminor: should be the requested first minor number to use; it is usually 0.
  - The count and name parameters work like those given to request_chrdev_region.

使用 mknod 创建设备文件

- mknod /dev/mydevice c 220 0

设备文件名

major number

字符设备

minor number

- 其他参数:
  - b create a block (buffered)
  - special file c, u create a character (unbuffered)
  - special file p create a FIFO pipe

内核空间和用户空间之间的数据传递

- 不能直接拷贝数据，需要内核提供的专门负责数据拷贝的几个函数
- 这几个函数分别是：

  <asm-generic/uaccess.h>

  – unsigned long copy_to_user(void _ _user *to, const void *from, unsigned long count);

  – unsigned long copy_from_user(void *to, const void _ _user *from, unsigned long count);

Cont.

- 如果要传递的数据类型(ptr参数指向的类型)可知，且是“单值的”，那么可以使用如下更快速的数据传递方法：
- put_user(datum, ptr)
- _ _put_user(datum, ptr)
- get_user(local, ptr)
- _ _get_user(local, ptr)

address verification

- <*asm-generic/uaccess.h*>:
- int access_ok(int type, const void *addr, unsigned long size);
  - Checks if a user space pointer is **valid**
  - type: Type of access, VERIFY_READ or VERIFY_WRITE
    - Note that VERIFY_WRITE is a superset of VERIFY_READ, if it is safe to write to a block, it is always safe to read from it
  - addr: User space pointer to start of block to check
  - size: Size of block to check, is a byte count

File Operations

- file_operations structure 在 <linux/fs.h> 中定义
- 通过定义 file_operations 结构来说明设备驱动程序能完成的功能

大部分域是函数指针。所以，驱动程序设计者需要“完成”这些函数(如果需要的话).

```
struct file_operations {
    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 (*read_iter) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*write_iter) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*iterate) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    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 (*mremap)(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 *, struct dentry *, 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);
    long (*fallocate)(struct file *file, int mode, loff_t offset,  loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
    #ifndef CONFIG_MMU
       unsigned (*mmap_capabilities)(struct file *);
    #endif
};
```

对各个域的说明

- struct module *owner
  - File operations的所有者(module)
  - 防止在模块操作还被使用的时候卸载模块
  - 基本上都被简单的初始化为宏THIS_MODULE
    - 在*<linux/module.h>*中定义

Cont.

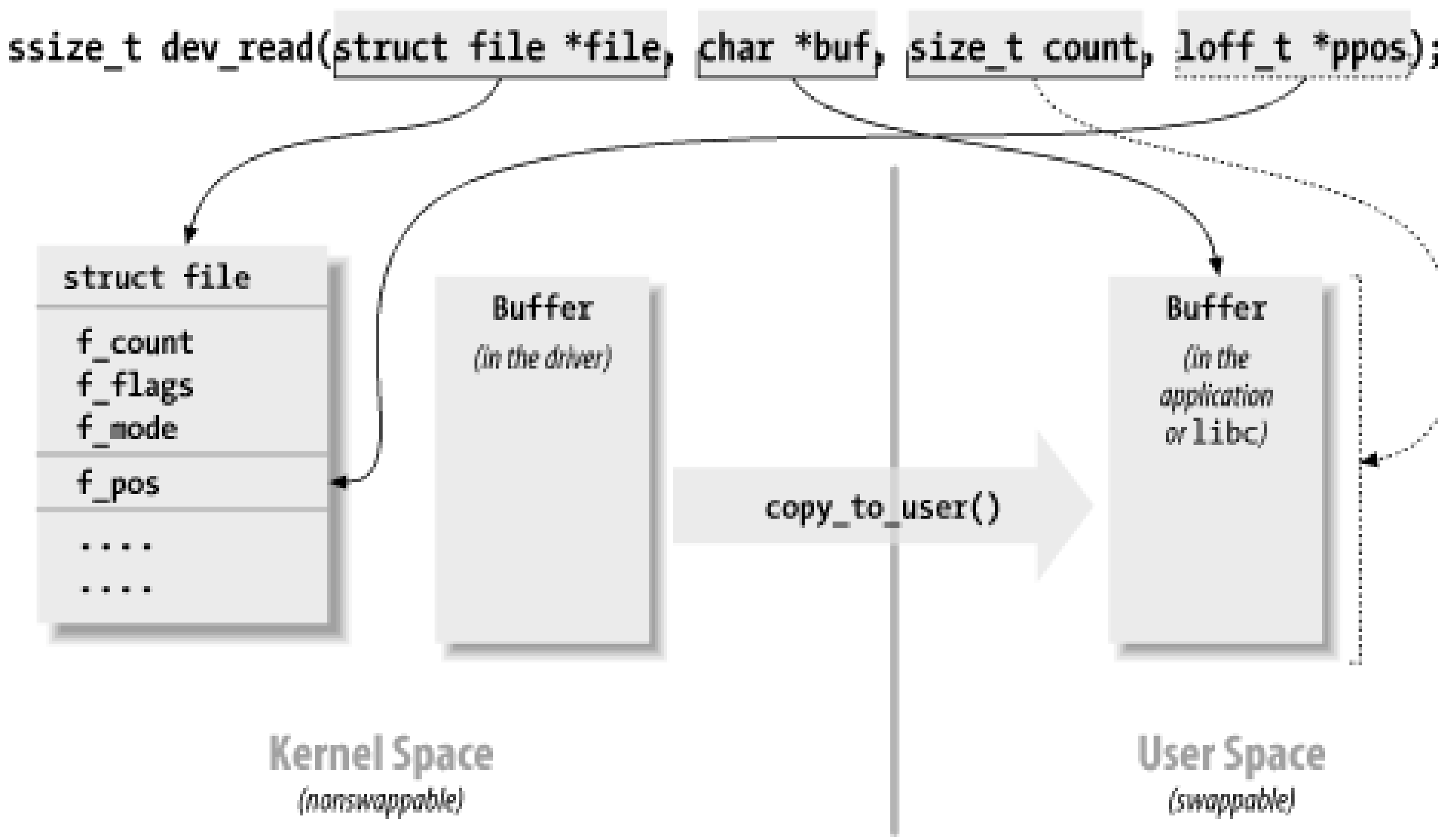
- loff_t (*llseek) (struct file *, loff_t, int);
  - 改变当前read/write的位置（position），返回值(一个正数)是新的位置，出错时函数返回值为负数
  - loff_t: a “long offset”，>=64-bit (即使在32-bit平台)。
  - If this function pointer is NULL, seek calls will modify the position counter in the file structure in potentially **unpredictable ways**.

Cont.

- ssize_t (*read) (struct file *, char _ _user *, size_t, loff_t *);

  - Used to retrieve data from the device.
  - A null pointer in this position causes the read system call to fail with -EINVAL ("Invalid argument").
  - A nonnegative return value represents the number of bytes successfully read (the return value is a "signed size" type, usually the native integer type for the target platform).

Cont.



ssize_t dev_read(struct file *file, char *buf, size_t count, loff_t *ppos);
struct file
f_count
f_flags
f_mode
f_pos
....
....
Buffer
(in the driver)
copy_to_user()
Buffer
(in the application or libc)
Kernel Space
(nonswappable)
User Space
(swappable)

Cont.

- ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t);
  - 异步读，即函数返回时读操作有可能还没结束
  - If this method is NULL, all operations will be processed (synchronously) by read instead.

Cont.

- ssize_t (*write) (struct file *, const char _ _user *, size_t, loff_t *);

  – Sends data to the device
  – If NULL, -EINVAL is returned to the program calling the write system call.
  – The return value, if nonnegative, represents the number of bytes successfully written.

Cont.

- ssize_t (*aio_write)(struct kiocb *, const char _ _user *, size_t, loff_t *);

  – Initiates an asynchronous write operation on the device.

Cont.

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

常见的nonblocking的文件处理技术

  - The poll method is the back end of three system calls: poll, epoll, and select, all of which are used to query whether a read or write to one or more file descriptors would block.
  - The poll method should return a bit mask indicating whether non-blocking reads or writes are possible, and, possibly, provide the kernel with information that can be used to put the calling process to sleep until I/O becomes possible.
  - If a driver leaves its poll method NULL, the device is assumed to be both readable and writable without blocking.

Cont.

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

  - The ioctl system call offers a way to issue device-specific commands (such as formatting a track of a floppy disk, which is neither reading nor writing).
  - a few ioctl commands are recognized by the kernel without referring to the fops table. If the device doesn't provide an ioctl method, the system call returns an error for any request that isn't predefined (-ENOTTY, "No such ioctl for device").

Cont.

- int (*mmap) (struct file *, struct vm_area_struct *);
  - mmap is used to request a mapping of device memory to a process's address space.
  - If this method is NULL, the mmap system call returns -ENODEV.

Cont.

- int (*open) (struct inode *, struct file *);
  - Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method.
  - If this entry is NULL, opening the device always succeeds, but your driver isn't notified.

Cont.

- int (*release) (struct inode *, struct file *);
  - This operation is invoked when the file structure is being released
  - Note that release isn't invoked every time a process calls close. Whenever a file structure is shared (for example, after a fork or a dup), release won't be invoked until all copies are closed.
  - If you need to flush pending data when any copy is closed, you should implement the flush method.

Cont.

- int (*flush) (struct file *);
  - The flush operation is invoked when a process closes its copy of a file descriptor for a device
  - Currently, flush is used in very few drivers; the SCSI tape driver uses it, for example, to ensure that all data written makes it to the tape before the device is closed.
  - If flush is NULL, the kernel simply ignores the user application request.

Cont.

- int (*fsync) (struct file *, struct dentry *, int);
  - This method is the back end of the fsync system call, which a user calls to flush any pending data.
  - If this pointer is NULL, the system call returns -EINVAL.

Cont.

- int (*aio_fsync)(struct kiocb *, int);
  - This is the asynchronous version of the fsync method.

Cont.

- int (*fasync) (int, struct file *, int);
  - This operation is used to notify the device of a change in its FASYNC flag.
  - The field can be NULL if the driver doesn't support asynchronous notification.

Cont.

- int (*lock) (struct file *, int, struct file_lock *);
  - The lock method is used to implement file locking
  - locking is an indispensable feature for regular files but is almost never implemented by device drivers.

Cont.

- ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
- ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
- These methods implement **scatter/gather read and write** operations.
- Applications occasionally need to do a single read or write operation involving multiple memory areas; these system calls allow them to do so without forcing extra copy operations on the data.
- If these function pointers are left NULL, the read and write methods are called (perhaps more than once) instead.

Cont.

- ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);
  - This method implements the read side of the sendfile system call, which moves the data from one file descriptor to another with a minimum of copying.
  - It is used, for example, by a web server that needs to send the contents of a file out a network connection.
  - Device drivers usually leave sendfile NULL.

Cont.

- ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);

  - sendpage is the other half of sendfile;
  - it is called by the kernel to send data, one page at a time, to the corresponding file.
  - Device drivers do not usually implement sendpage.

初始化file_operations的方法

- struct file_operations scull_fops =

```
{ .owner = THIS_MODULE,
  .llseek = scull_llseek,
  .read = scull_read,
  .write = scull_write,
  .ioctl = scull_ioctl,
  .open = scull_open,
  .release = scull_release,
};
```

struct file

- 在<linux/fs.h>中定义
- 对每个打开的文件（包括表示设备的特殊文件）都和一个 struct file相关联
- 在内核调用open的时候，**创建**此结构；当文件不再使用的时候，内核释放该结构

```
struct file {
    union {
        struct list_head  fu_list;
        struct rcu_head       fu_rcuhead;
    } f_u;
    struct path         f_path;
    struct inode      *f_inode;
    const struct file_operations  *f_op;
    spinlock_t    f_lock;
    atomic_long_t          f_count;
    unsigned int           f_flags;
    fmode_t                f_mode;
    struct mutex      f_pos_lock;
    loff_t             f_pos;
    struct fown_struct     f_owner;
    const struct cred      *f_cred;
    struct file_ra_state   f_ra;
    u64       f_version;
#ifdef CONFIG_SECURITY
    void               *f_security;
#endif
    void                  *private_data;
#ifdef CONFIG_EPOLL
    struct list_head f_ep_links;
    struct list_head f_tfile_llink
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space   *f_mapping;
};
```

结构中的域说明

- fmode_t f_mode;
  - 这个域用来指出文件是可读的还是可写的，或者又可读又可写。
  - FMODE_READ FMODE_WRITE.
  - You might want to check this field for read/write permission in your open or ioctl function, but you don't need to check permissions for read and write, because the kernel checks before invoking your method. An attempt to read or write when the file has not been opened for that type of access is rejected without the driver even knowing about it.

Cont.

- loff_t f_pos;
  - 当前的read/write位置
  - loff_t is a 64-bit value on all platforms (long long in gcc terminology).
  - read and write should update a position using the pointer they receive as the last argument instead of acting on filp->f_pos directly.
  - llseek method can change the file position.

Cont.

- unsigned int f_flags;
  - Such as O_RDONLY, O_NONBLOCK, and O_SYNC.
  - All the flags are defined in the header <*linux/fcntl.h*>.

Cont.

- struct file_operations *f_op;
  - 文件操作
  - The kernel assigns the pointer as part of its implementation of open and then reads it when it needs to dispatch any operations.
  - You can change the file operations associated with your file, and the new methods will be effective after you return to the caller.

Cont.

- void *private_data;
  - private_data is a useful resource for preserving state information across system calls.
  - You can use the field to point to allocated data, but then you must remember to free that memory in the release method before the file structure is destroyed by the kernel.
  - The open system call sets this pointer to NULL before calling the open method for the driver.

Cont.

- struct dentry *f_dentry;

  – 文件的dentry
  – You can use f_dentry to visit i-node

    structure:filp->f_dentry->d_inode

inode Structure

Inode结构包含与file相关的大量信息，写驱动程序的时候主要注意下面两个域：

- dev_t i_rdev;
  - For inodes that represent device files, this field contains the **actual device number**.
- struct cdev *i_cdev;
  - struct cdev is the kernel's internal structure that represents char devices; this field contains a pointer to that structure when the inode refers to a char device file.

obtain the major and minor number

- unsigned int iminor(struct inode *inode);
- unsigned int imajor(struct inode *inode);

- 开始编写字符设备驱动程序

编写字符设备驱动程序的一般步骤

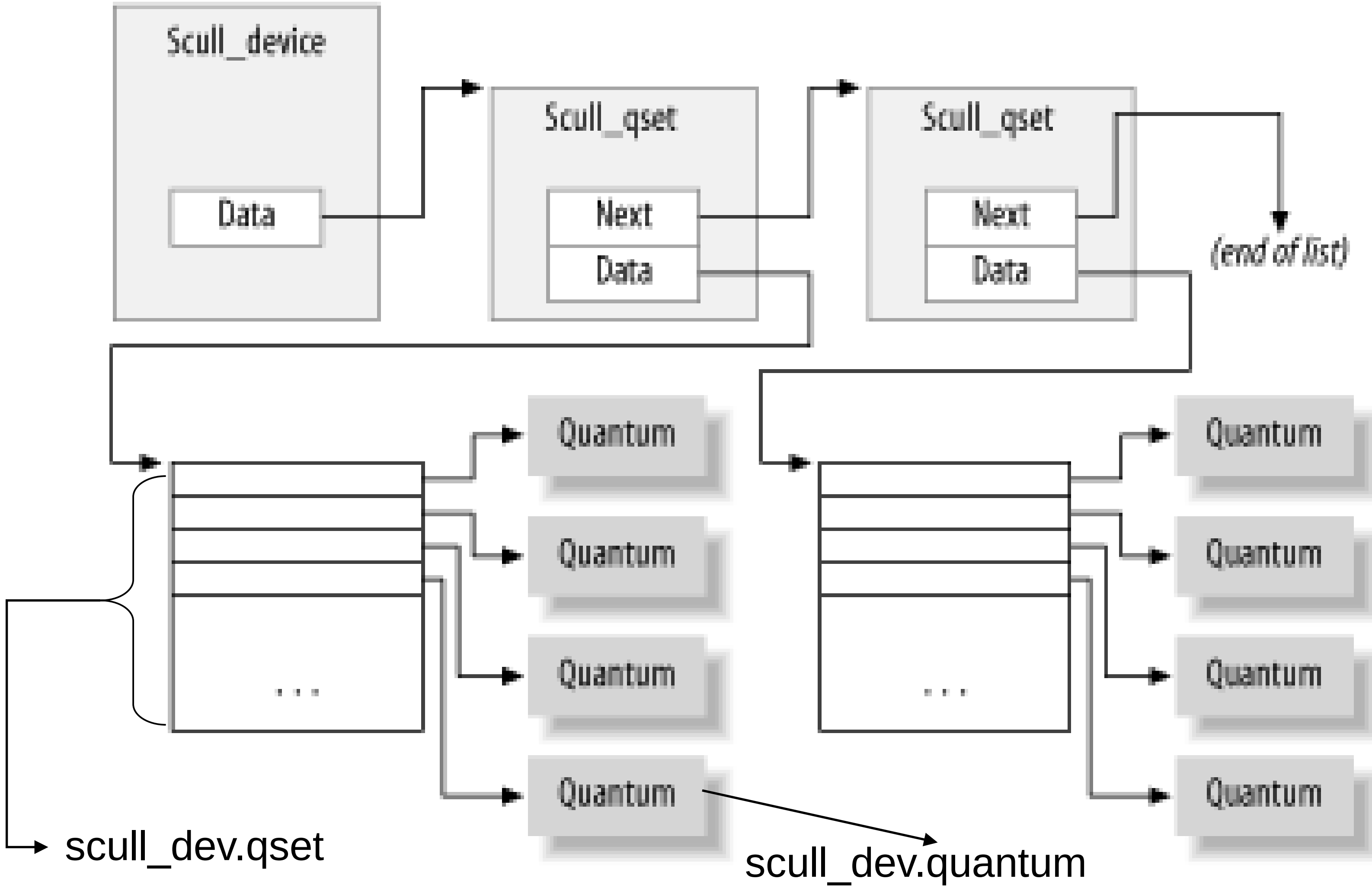
- 确定字符设备需要的设备号，并分配设备号；注册设备
- 定义和申明设备驱动程序中**特定的数据结构**，这个数据结构可以主要保存以下信息：
  - 内存指针和内存相关的信息
  - 嵌入字符设备类型struct cdev，当然也可不嵌入，根据需要定义
  - 定义同步或互斥变量等相关信息
- 要注意不同调用之间的信息的传递问题

Cont.

- 编写设备初始化代码
- 编写设备驱动程序需要的其他操作
- 考虑和中断操作的相关代码
- 等等

sample--scull

- scull的作用是让使用者可把一块内存区域当成字符设备使用，即scull驱动的目标设备是一块内存区域
  - 不依赖任何硬件
  - 只是为了展示kernel与char driver之间的接口，没有实用功能



Scull 设备内部的内存结构

和数据存放相关的数据结构

```
struct scull_qset {
  void **data;
  struct scull_qset *next;
};
```

scull_dev

```
struct scull_dev {
    struct scull_qset *data;  /* Pointer to first quantum set */
    int quantum;              /* the current quantum size */
    int qset;                 /* the current array size */
    unsigned long size;       /* amount of data stored here */
    unsigned int access_key;  /* used by sculluid and scullpriv */

    struct semaphore sem;     /* mutual exclusion semaphore */

    struct cdev cdev;         /* Char device structure */

};
```

使用信号量做读、写互斥

Struct cdev 嵌入到了scull_dev数据结构中

例子中对信号量进行操作函数

- 这个例子中，把信号量当成互斥量来进行操作，也就是说其是二值信号量
- int down_interruptible(struct semaphore *sem);
  - 如果信号量可用，那么down_interruptible 把信号量值减去1
  - 如果信号量不可用，那么down_interruptible让当前进程处于等待状态，而且这种等待状态可以被用户中断，比如可以被信号中断。
  - 相当于加锁操作
- void up(struct semaphore *sem);
  - 释放信号量
  - 相当于释放锁操作
- DECLARE_MUTEX(name);
  - 初始化信号量

初始化 file_operations 结构 scull_fops

```
struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .ioctl =    scull_ioctl,
    .open =     scull_open,
    .release =  scull_release,
};
```

文件操作赋值

- **Open**时基于磁盘**inode**创建内存**inode**时，**init_special_inode**确认是否是字符设备文件
- 如果是, 则将**i_fop**赋值为**def_chr_fops**
- **do_last→finish_open→do_dentry_open** 将**f_op**赋值为**i_fop**，并调用**f_op→open**，即执行**def_chr_fops→ open**（**chrdev_open**）
- **chrdev_open**把**cdev→ops**赋值给**file→ f_op**

模块初始化和退出

- module_init(scull_init_module);
- module_exit(scull_cleanup_module);

使用shell 编写自动加载模块和创建特殊设备的脚本

来源于教材的例子scull_load：

```
#!/bin/sh
module="scull"
device="scull"
mode="664"
# invoke insmod with all arguments we got
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod ./$module.ko $* || exit 1
# remove stale nodes
rm -f /dev/${device}[0-3]
```

Cont.

```
major=$(awk "\\$2==\"$module\" {print \\$1}" /proc/devices)
mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3
# give appropriate group/permissions, and change the group.
# Not all distributions have staff, some have "wheel" instead.
group="staff"
grep -q '^staff:' /etc/group || group="wheel"
chgrp $group /dev/${device}[0-3]
chmod $mode  /dev/${device}[0-3]
```

int scull_init_module(void)

```
int scull_init_module(void)
{
    int result, i;
    dev_t dev = 0;
/*
 * Get a range of minor numbers to work with, asking for a dynamic
 * major unless directed otherwise at load time.
 */
    if (scull_major) {
        dev = MKDEV(scull_major, scull_minor);
        result = register_chrdev_region(dev, scull_nr_devs, "scull");
    } else {
        result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs, "scull");
        scull_major = MAJOR(dev);
    }
    if (result < 0) {
        printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
        return result;
    }
```

```c
/*  * allocate the devices -- we can't have them static, as the number can be
    specified at load time */
scull_devices = kmalloc(scull_nr_devs * sizeof(struct scull_dev), GFP_KERNEL)
if (!scull_devices) {
        result = -ENOMEM;
        goto fail;  /* Make this more graceful */
}
memset(scull_devices, 0, scull_nr_devs * sizeof(struct scull_dev));
        /* Initialize each device. */
for (i = 0; i < scull_nr_devs; i++) {
        scull_devices[i].quantum = scull_quantum;
        scull_devices[i].qset = scull_qset;
        init_MUTEX(&scull_devices[i].sem);
        scull_setup_cdev(&scull_devices[i], i);
}
```

Cont.

```
 /* At this point call the init function for any friend device */
    dev = MKDEV(scull_major, scull_minor + scull_nr_devs);
/*另外两个driver，提供了更高级的操作 */
    dev += scull_p_init(dev);  //阻塞式读、写的实现，异步通知的实现
    dev += scull_access_init(dev); //对设备打开权限控制的实现
#ifdef SCULL_DEBUG /* only when debugging */
    scull_create_proc();
#endif
    return 0; /* succeed */
  fail:
    scull_cleanup_module();
    return result;
}
```

scull_setup_cdev

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
    int err, devno = MKDEV(scull_major, scull_minor + index);

    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno, 1); //注册

    /* Fail gracefully if need be */
    if (err)
        printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

scull_cleanup_module

```
void scull_cleanup_module(void)
{
    int i;
    dev_t devno = MKDEV(scull_major, scull_minor);
    /* Get rid of our char dev entries */
    if (scull_devices) {
        for (i = 0; i < scull_nr_devs; i++) {
            scull_trim(scull_devices + i); //释放设备链接的内存区域
            cdev_del(&scull_devices[i].cdev);
        }
        kfree(scull_devices);
    }
```

Cont.

```
#ifdef SCULL_DEBUG /* use proc only if debugging */
    scull_remove_proc();
#endif

    /* cleanup_module is never called if registering failed */
    unregister_chrdev_region(devno, scull_nr_devs);
```
→ 释放设备号

```
   /* and call the cleanup functions for friend devices */
    scull_p_cleanup();
    scull_access_cleanup();
}
```

container_of

- 这个宏用来用结构类型中的域的地址来求包含这个域的结构的地址

```
/*container_of - cast a member of a structure out to the containing structure
* @ptr: the pointer to the member.
* @type: the type of the container struct this is embedded in.
* @member: the name of the member within the struct.
*/
#define container_of(ptr, type, member)  ({
const typeof( ((type *)0)->member ) *__mptr = (ptr);
(type *)( (char *)__mptr - offsetof(type,member) );})

#define offsetof(TYPE, MEMBER)   ((size_t) &((TYPE *)0)->MEMBER)
```

- 比如
  - struct a {

```
struct a {
    int b;
    int c;
};
```

struct a temp;　如果知道 temp.c 的地址，那么temp的地址也就可以求得：&temp.c – sizeof(int)

scull_open

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */

    dev = container_of(inode->i_cdev, struct scull_dev, cdev); //由设备文件inode找到
                                                               //scull_cdev结构
    filp->private_data = dev; /* for other methods, e.g. read/write…*/

    /* now trim to 0 the length of the device if open was write-only */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
        scull_trim(dev); /* ignore errors */
        up(&dev->sem);
    }
    return 0;        /* success */
}
```

scull_follow

//找到第n个qset，如果没有就分配

```
struct scull_qset *scull_follow(struct scull_dev *dev, int n)
{
    struct scull_qset *qs = dev->data;

      /* Allocate first qset explicitly if need be */
    if (! qs) {
            qs = dev->data = kmalloc(sizeof(struct scull_qset),
            GFP_KERNEL);
            if (qs == NULL)  //没有可分配的内存
                  return NULL;  /* Never mind */
            memset(qs, 0, sizeof(struct scull_qset));
    }
```

scull_follow

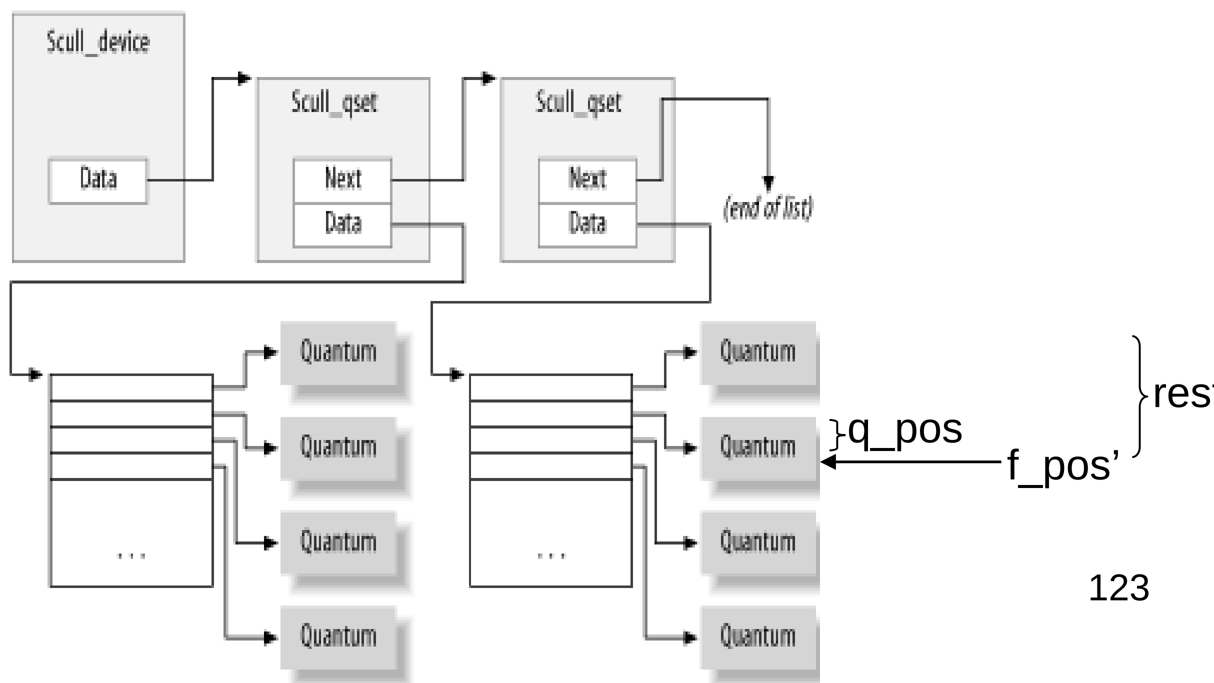
```
    /* Then follow the list */
    while (n--) {
            if (!qs->next) {
                    qs->next = kmalloc(sizeof(struct scull_qset), GFP_KERNEL);
                    if (qs->next == NULL)
                            return NULL;  /* Never mind */
                    memset(qs->next, 0, sizeof(struct scull_qset));
            }
            qs = qs->next;
            continue;
    }
    return qs;
}
```

scull_read

```
ssize_t scull_read(struct file *filp, char_user *buf, size_t count, loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;  /* the first listitem */
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset; /* how many bytes in the listitem */
    int item, s_pos, q_pos, rest;
    ssize_t retval = 0; //返回值

    if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    if (*f_pos >= dev->size)
            goto out;
    if (*f_pos + count > dev->size)
            count = dev->size - *f_pos;
```

```
    /* find listitem, qset index, and offset in the quantum */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;
    /* follow the list up to the right position (defined elsewhere) */
    dptr = scull_follow(dev, item);
    if (dptr == NULL || !dptr->data || ! dptr->data[s_pos])
            goto out; /* don't fill holes */
    /* read only up to the end of this quantum */
    if (count > quantum - q_pos)
            count = quantum - q_pos;
    if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
            retval = -EFAULT;
            goto out;
    }
    *f_pos += count;
    retval = count;
  out:
    up(&dev->sem);
    return retval;
}
```



scull_write

```
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset;
    int item, s_pos, q_pos, rest;
    ssize_t retval = -ENOMEM; /* value used in "goto out" statements */
    if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    /* find listitem, qset index and offset in the quantum */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;
    /* follow the list up to the right position */
    dptr = scull_follow(dev, item);
    if (dptr == NULL) //follow过程中内存耗尽，位置不对
      goto out;
```

Cont.

```
if (!dptr->data) {  //没有qset数组则分配
        dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
        if (!dptr->data)
                goto out;
        memset(dptr->data, 0, qset * sizeof(char *));
}
if (!dptr->data[s_pos]) { //第s_pos个quantum不存在则分配
        dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
        if (!dptr->data[s_pos])
                goto out;
}
/* write only up to the end of this quantum */
if (count > quantum - q_pos)
        count = quantum - q_pos;
```

Cont.

```
   if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
           retval = -EFAULT;
           goto out;
    }
    *f_pos += count;
    retval = count;
     /* update the size */
    if (dev->size < *f_pos)
           dev->size = *f_pos;
  out:
    up(&dev->sem);
    return retval;
}
```

ioctl

- int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
- 作用：从驱动程序中得到数据或者往驱动程序传递数据和命令

Device specific cmd

ioctl 相关的辅助宏定义

- 命令编号被分割成四段：

  DIR | SIZE | TYPE | NR

- _IOC_DIR(cmd) 方向
  - 可能的值
    - _IOC_NONE (no data transfer),_IOC_READ, _IOC_WRITE, and _IOC_READ|_IOC_WRITE (data is transferred both ways).
  - Data transfer is seen from the application's point of view; _IOC_READ means reading from the device, so the driver must write to user space.
- _IOC_TYPE(cmd) 类型，与设备相关，避免冲突
- _IOC_NR(cmd) number，命令序号
- _IOC_SIZE(cmd) 长度，传递的数据大小

- *<asm-generic/ioctl.h> 构造命令编号的宏：*
  - _IO(type,nr) (for a command that has no argument)
  - _IOR(type,nr,datatype) (for reading data from the driver)
  - _IOW(type,nr,datatype) (for writing data)
  - _IOWR(type,nr,datatype) (for bidirectional transfers)

an identifying letter or number

sequence number to distinguish ioctls from each other

type of the data going into the kernel or coming out of the kernel

命令定义

- #define SCULL_IOC_MAGIC 'k'
- #define SCULL_IOCRESET _IO(SCULL_IOC_MAGIC, 0)
- #define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC, 1, int)
- #define SCULL_IOCSQSET _IOW(SCULL_IOC_MAGIC, 2, int)
- #define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC, 3)
- #define SCULL_IOCTQSET _IO(SCULL_IOC_MAGIC, 4)
- #define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC, 5, int)
- #define SCULL_IOCGQSET _IOR(SCULL_IOC_MAGIC, 6, int)
- #define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC, 7)
- #define SCULL_IOCQQSET _IO(SCULL_IOC_MAGIC, 8)
- #define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)
- #define SCULL_IOCXQSET _IOWR(SCULL_IOC_MAGIC,10, int)
- #define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC, 11)
- #define SCULL_IOCHQSET _IO(SCULL_IOC_MAGIC, 12)
- #define SCULL_IOC_MAXNR 14

scull_ioctl

```
int scull_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
{
    int err = 0, tmp;
    int retval = 0;
    /* * extract the type and number bitfields, and don't decode
     * wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
     */
    if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
    if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;
    /** the direction is a bitmask, and VERIFY_WRITE catches R/W
     * transfers. `Type' is user-oriented, while
     * access_ok is kernel-oriented, so the concept of "read" and
     * "write" is reversed */
    if (_IOC_DIR(cmd) & _IOC_READ)
            err = !access_ok(VERIFY_WRITE, (void_user *)arg, IOC_SIZE(cmd));
    else if (_IOC_DIR(cmd) & _IOC_WRITE)
            err =  !access_ok(VERIFY_READ, (void user *)arg, IOC_SIZE(cmd));
    if (err) return -EFAULT;
```

Cont.

```
switch(cmd) {
      case SCULL_IOCRESET:
      scull_quantum = SCULL_QUANTUM;
      scull_qset = SCULL_QSET;
      break;
      case SCULL_IOCSQUANTUM: /* Set: arg points to the value */
      if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
      retval = __get_user(scull_quantum, (int __user *)arg);
      break;
      case SCULL_IOCTQUANTUM: /* Tell: arg is the value */
      if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
      scull_quantum = arg;
      break;
```

Cont.

```
case SCULL_IOCGQUANTUM: /* Get: arg is pointer to result */
retval = __put_user(scull_quantum, (int __user *)arg);
break;

case SCULL_IOCQQUANTUM: /* Query: return it (it's positive) */
return scull_quantum;

case SCULL_IOCXQUANTUM: /* eXchange: use arg as pointer */
if (! capable (CAP_SYS_ADMIN))
      return -EPERM;
tmp = scull_quantum;
retval = __get_user(scull_quantum, (int __user *)arg);
if (retval == 0)
      retval = __put_user(tmp, (int __user *)arg);
break;
```

Cont.

```
case SCULL_IOCHQUANTUM: /* sHift: like Tell + Query */
if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
tmp = scull_quantum;
scull_quantum = arg;
return tmp;

case SCULL_IOCSQSET:
if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
retval = __get_user(scull_qset, (int __user *)arg);
break;

case SCULL_IOCTQSET:
if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
scull_qset = arg;
break;
```

Cont.

```
case SCULL_IOCGQSET:
retval = __put_user(scull_qset, (int __user *)arg);
break;

case SCULL_IOCQQSET:
return scull_qset;

case SCULL_IOCXQSET:
if (! capable (CAP_SYS_ADMIN))
    return -EPERM;
tmp = scull_qset;
retval = __get_user(scull_qset, (int __user *)arg);
if (retval == 0)
    retval = put_user(tmp, (int __user *)arg);
break;
```

Cont.

```
  case SCULL_IOCHQSET:
        if (! capable (CAP_SYS_ADMIN))
              return -EPERM;
        tmp = scull_qset;
        scull_qset = arg;
        return tmp;
  /*
   * The following two change the buffer size for scullpipe.
   * The scullpipe device uses this same ioctl method, just to
   * write less code. Actually, it's the same driver, isn't it?
   */
  case SCULL_P_IOCTSIZE:
        scull_p_buffer = arg;
        break;
  case SCULL_P_IOCQSIZE:
        return scull_p_buffer;
  default:  /* redundant, as cmd was checked against MAXNR */
        return -ENOTTY;
 }
 return retval;
}
```

用户空间使用ioctl

- 在用户空间, ioctl 系统调用有下面的原型:
  int ioctl (int fd, unsigned long cmd, ...);

```
int quantum;
ioctl(fd,SCULL_IOCSQUANTUM, &quantum);          /* Set by pointer */
ioctl(fd,SCULL_IOCTQUANTUM, quantum);           /* Set by value */
ioctl(fd,SCULL_IOCGQUANTUM, &quantum);          /* Get by pointer */
quantum = ioctl(fd,SCULL_IOCQQUANTUM);          /* Get by return value */
ioctl(fd,SCULL_IOCXQUANTUM, &quantum);          /* Exchange by pointer */
quantum = ioctl(fd,SCULL_IOCHQUANTUM, quantum); /* Exchange by value*/
```

Capabilities and Restricted Operations

<linux/sched.h>

- int capable(int capability); 调用进程是否有capability的能力, 对设备文件细粒度许可控制

```
if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
```

<linux/capability.h>

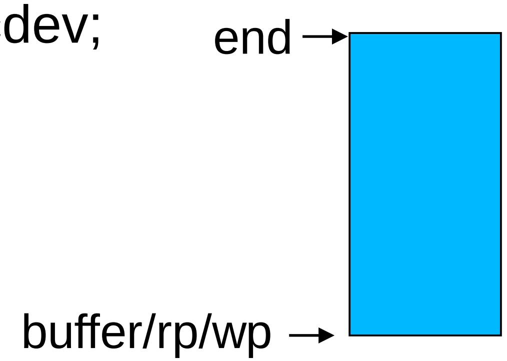
- CAP_DAC_OVERRIDE
  - 这个能力来推翻在文件和目录上的存取限制(数据存取控制, 或者 DAC).
- CAP_NET_ADMIN
  - 进行网络管理任务的能力, 包括那些能够影响网络接口的.
- CAP_SYS_MODULE
  - 加载或去除内核模块的能力.
- CAP_SYS_RAWIO
  - 进行 "raw" I/O 操作的能力. 包括存取设备端口或者直接和 USB 设备通讯.
- CAP_SYS_ADMIN
  - 一个捕获-全部（catch-all）的能力, 提供对许多系统管理操作的能力.
- CAP_SYS_TTY_CONFIG
  - 进行 tty 配置任务的能力.

- 实现有PIPE特点的字符设备驱动程序

PIPE字符设备的特点

- 是字符设备
- 读写数据的顺序是先进先出，FIFO
- 如果没有数据可读，读数据的进程或线程需要等待
- 如果没有空间可写或者正在读数据，写过程阻塞
- 读写数据的时候的操作要互斥，即读的时候不能写；写的时候不能读
- 不能在同一时刻有多个读操作在进行；也不能在同一时刻有多个写操作在进行

- ```
  struct scull_pipe {
          wait_queue_head_t inq, outq;          /* read and write queues */
          char *buffer, *end;                   /* begin of buf, end of buf */
          int buffersize;                       /* used in pointer arithmetic */
          char *rp, *wp;                        /* where to read, where to write */
          int nreaders, nwriters;               /* number of openings for r/w */
          struct fasync_struct *async_queue;    /* asynchronous readers */
          struct semaphore sem;                 /* mutual exclusion semaphore */
          struct cdev cdev;                     /* Char device structure */
  };
  ```


  ```

struct file_operations

```
struct file_operations scull_pipe_fops = {
    .owner = THIS_MODULE,
    .llseek = no_llseek,
    .read =         scull_p_read,
    .write =   scull_p_write,
    .poll =     scull_p_poll,
    .ioctl =    scull_ioctl,
    .open =         scull_p_open,
    .release =  scull_p_release,
    .fasync =scull_p_fasync,
};
```

异步通知

- 异步通知的意思是：一旦设备就绪，则主动通知应用程序，这样应用程序根本就不需要查询设备状态，这一点非常类似于硬件上“中断”的概念，比较准确的称谓是“信号驱动**(SIGIO)**的异步**I/O”**。
- **Linux**系统中，异步通知使用信号来实现
  - 进程执行时，**ctrl+c**发出**SIGINT**信号，**kill**发出**SIGTERM**信号
  - 查看**Linux**下的用户空间的信号处理函数使用

Cont.

- 用户空间需要的操作

```
signal(SIGIO, input_handler);
                //让input_handler()处理SIGIO信号
fcntl(fd, F_SETOWN, getpid()); //通知谁
oflags= fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, oflags| FASYNC);
//打开 FASYNC, 驱动的 fasync 方法被调用
```

Cont.

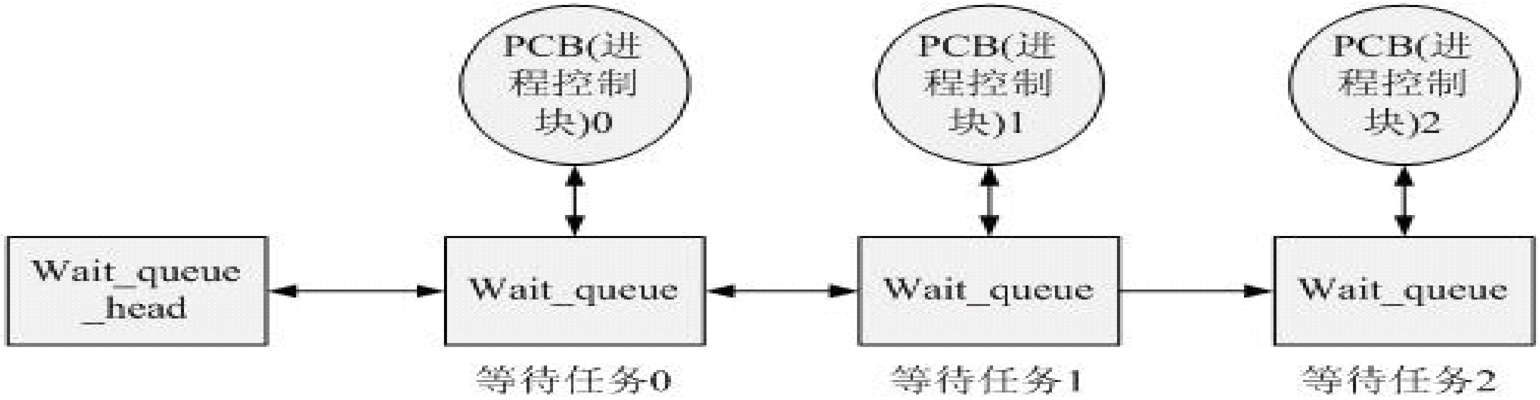
- 驱动程序需要的变化

驱动中的**fasync**函数将被执行
fasync中关键函数为**fasync_helper**
内核中释放信号的函数为**kill_fasync**

等待队列

- 一个等待队列就是一个等待特定的事件**进程列表**
- 在 Linux 中, 一个等待队列由一个“等待队列头”来管理
  - wait_queue_head_t 类型的结构
  - 定义在<linux/wait.h>中
- 等待队列头定义和初始化
  - **DECLARE_WAIT_QUEUE_HEAD**(name);
  - 或者动态地:

    wait_queue_head_t my_queue;
    init_waitqueue_head(&my_queue);



简单睡眠

- 任何睡眠的进程必须在它醒来时检查以确保它在等待的条件为真.
- Linux 内核中睡眠的最简单方式是一个宏定义, 称为**wait_event**(有几个变体); 它结合了处理睡眠的细节和进程在等待的条件的检查.
  - wait_event(queue, condition)
  - wait_event_interruptible(queue, condition)
  - wait_event_timeout(queue, condition, timeout)
  - wait_event_interruptible_timeout(queue, condition, timeout)

  queue：等待队列头.

  condition:：布尔表达式，在睡眠前后被求值（可能被任意次地求值, 因此不应当有任何边界效应）。进程睡眠直到condition求值为真
- 尽量使用 wait_event_interruptible：可能被信号中断，返回整数值（非零值意味着睡眠被某些信号打断，并且你的驱动可能应当返回-ERESTARTSYS）
- wait_event_timeout 和 wait_event_interruptible_timeout：超时后, 返回一个 0 值而不管条件是如何求值的.

唤醒

- 必须有其他的线程 (一个不同的进程, 或者是一个中断处理) 唤醒睡眠进程
- 基本的唤醒函数称为 **wake_up**（有几个变体，下面是其中两个）
  - void wake_up(wait_queue_head *q);
  - void wake_up_interruptible(wait_queue_head *q);
- 如果使用可中断的睡眠，一般不用区分这两个函数（通常wake_up对应wait_event；wake_up_interruptible对应wait_event_interruptible）

scull_p_init

```
int scull_p_init(dev_t firstdev)
{
	int i, result;
	result = register_chrdev_region(firstdev, scull_p_nr_devs, "scullp");
	if (result < 0) {
		printk(KERN_NOTICE "Unable to get scullp region, error %d\n", result);
		return 0;
	}
	scull_p_devno = firstdev;
	scull_p_devices = kmalloc(scull_p_nr_devs * sizeof(struct scull_pipe), GFP_KERNEL);
	if (scull_p_devices == NULL) {
		unregister_chrdev_region(firstdev, scull_p_nr_devs);
		return 0;
	}
	memset(scull_p_devices, 0, scull_p_nr_devs * sizeof(struct scull_pipe));
```

Cont.

```
    for (i = 0; i < scull_p_nr_devs; i++) {
        init_waitqueue_head(&(scull_p_devices[i].inq));
        init_waitqueue_head(&(scull_p_devices[i].outq));
        init_MUTEX(&scull_p_devices[i].sem);
        scull_p_setup_cdev(scull_p_devices + i, i);
    }
#ifdef SCULL_DEBUG
    create_proc_read_entry("scullpipe", 0, NULL,
    scull_read_p_mem, NULL);
#endif
    return scull_p_nr_devs;
}
```

scull_p_setup_cdev

```
static void scull_p_setup_cdev(struct scull_pipe *dev, int index)
{
    int err, devno = scull_p_devno + index;

    cdev_init(&dev->cdev, &scull_pipe_fops);
    dev->cdev.owner = THIS_MODULE;
    err = cdev_add (&dev->cdev, devno, 1);
    /* Fail gracefully if need be */
    if (err)
        printk(KERN_NOTICE "Error %d adding scullpipe%d",
    err, index);
}
```

scull_p_cleanup

```
void scull_p_cleanup(void)
{
	int i;
#ifdef SCULL_DEBUG
	remove_proc_entry("scullpipe", NULL);
#endif
	if (!scull_p_devices)
		return; /* nothing else to release */

	for (i = 0; i < scull_p_nr_devs; i++) {
		cdev_del(&scull_p_devices[i].cdev);
		kfree(scull_p_devices[i].buffer);
	}
	kfree(scull_p_devices);
	unregister_chrdev_region(scull_p_devno, scull_p_nr_devs);
	scull_p_devices = NULL; /* pedantic */
}
```

scull_p_open

```
static int scull_p_open(struct inode *inode, struct file *filp)
{
	struct scull_pipe *dev;

	dev = container_of(inode->i_cdev, struct scull_pipe, cdev);
	filp->private_data = dev;

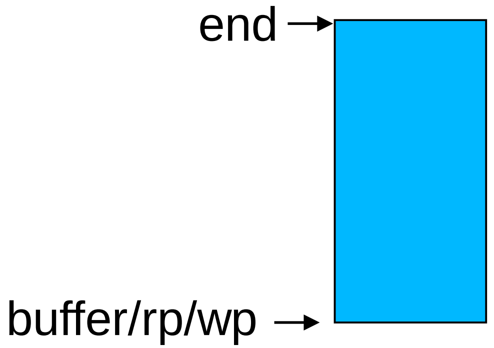
	if (down_interruptible(&dev->sem))
		return -ERESTARTSYS;
	if (!dev->buffer) {
		/* allocate the buffer */
		dev->buffer = kmalloc(scull_p_buffer, GFP_KERNEL);
		if (!dev->buffer) {
			up(&dev->sem);
			return -ENOMEM;
		}
	}
```

Cont.

```
dev->buffersize = scull_p_buffer;
dev->end = dev->buffer + dev->buffersize;
dev->rp = dev->wp = dev->buffer; /* rd and wr from the beginning */

/* use f_mode,not  f_flags: it's cleaner (fs/open.c tells why) */
if (filp->f_mode & FMODE_READ)
        dev->nreaders++;
if (filp->f_mode & FMODE_WRITE)
        dev->nwriters++;
up(&dev->sem);

return nonseekable_open(inode, filp); //通知内核设备不支持 llseek
}
```



scull_p_release

```
static int scull_p_release(struct inode *inode, struct file *filp)
{
	struct scull_pipe *dev = filp->private_data;
	/* remove this filp from the asynchronously notified filp's */
	scull_p_fasync(-1, filp, 0); //从异步通知列表中删除该文件指针
	down(&dev->sem);
	if (filp->f_mode & FMODE_READ)
		dev->nreaders--;
	if (filp->f_mode & FMODE_WRITE)
		dev->nwriters--;
	if (dev->nreaders + dev->nwriters == 0) {
		kfree(dev->buffer);
		dev->buffer = NULL; /* the other fields are not checked on open */
	}
	up(&dev->sem);
	return 0;
}
```

scull_p_fasync

```
static int scull_p_fasync(int fd, struct file *filp,
    int mode)
{
    struct scull_pipe *dev = filp->private_data;

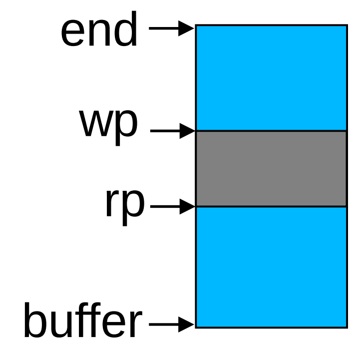
    return fasync_helper(fd, filp, mode, &dev-
    >async_queue); //异步通知队列操作

}
```

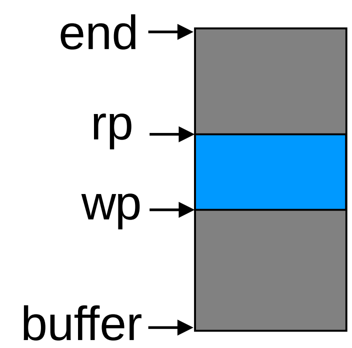
scull_p_read

```
static ssize_t scull_p_read (struct file *filp, char __user *buf, size_t count,
            loff_t *f_pos)
{
    struct scull_pipe *dev = filp->private_data;
    if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    while (dev->rp == dev->wp) { /* nothing to read */
            up(&dev->sem); /* release the lock */
            if (filp->f_flags & O_NONBLOCK)
                  return -EAGAIN;
            PDEBUG("\"%s\" reading: going to sleep\n", current->comm);
            if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp))) //返回非零值意味
                                                                           //着睡眠被信号打断
                  return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
            /* otherwise loop, but first reacquire the lock */
            if (down_interruptible(&dev->sem))
                  return -ERESTARTSYS;
    }
```

Cont.




```
    /* ok, data is there, return something */
    if (dev->wp > dev->rp)
            count = min(count, (size_t)(dev->wp - dev->rp));
    else /* the write pointer has wrapped, return data up to dev->end */
            count = min(count, (size_t)(dev->end - dev->rp));
    if (copy_to_user(buf, dev->rp, count)) {
            up (&dev->sem);
            return -EFAULT;
    }
    dev->rp += count;
    if (dev->rp == dev->end)
            dev->rp = dev->buffer; /* wrapped */
    up (&dev->sem);
    /* finally, awake any writers and return */
    wake_up_interruptible(&dev->outq);
    PDEBUG("\"%s\" did read %li bytes\n",current->comm, (long)count);
    return count;
}
```



scull_p_write

```
static ssize_t scull_p_write(struct file *filp, const char __user *buf, size_t count,
            loff_t *f_pos)
{
    struct scull_pipe *dev = filp->private_data;
    int result;

    if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;

    /* Make sure there's space to write */
    result = scull_getwritespace(dev, filp);
    if (result)
            return result; /* scull_getwritespace called up(&dev->sem)，非零值出错*/

    /* ok, space is there, accept something */
    count = min(count, (size_t)spacefree(dev));
```

Cont.

```
if (dev->wp >= dev->rp)
        count = min(count, (size_t)(dev->end - dev->wp)); /* to end-of-buf */
else /* the write pointer has wrapped, fill up to rp-1 */
        count = min(count, (size_t)(dev->rp - dev->wp - 1));
PDEBUG("Going to accept %li bytes to %p from %p\n", (long)count, dev->wp, buf);
if (copy_from_user(dev->wp, buf, count)) {
        up (&dev->sem);
        return -EFAULT;
}
dev->wp += count;
if (dev->wp == dev->end)
        dev->wp = dev->buffer; /* wrapped */
up(&dev->sem);\
/* finally, awake any reader */
wake_up_interruptible(&dev->inq);  /* blocked in read() and select() */
/* and signal asynchronous readers */
if (dev->async_queue)
        kill_fasync(&dev->async_queue, SIGIO, POLL_IN); //发异步读通知信号
PDEBUG("\"%s\" did write %li bytes\n",current->comm, (long)count);
return count;
}
```

scull_getwritespace

手动睡眠

```
static int scull_getwritespace(struct scull_pipe *dev, struct file *filp)
{
    while (spacefree(dev) == 0) { /* full */
        DEFINE_WAIT(wait);  //creation and initialization of a wait queue entry
        up(&dev->sem);
        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        PDEBUG("\"%s\" writing: going to sleep\n",current->comm);
        prepare_to_wait(&dev->outq, &wait, TASK_INTERRUPTIBLE); //add your wait queue entry
                                                                //to the queue, and set the process state
        if (spacefree(dev) == 0)
            schedule();
        finish_wait(&dev->outq, &wait); //Sets current thread back to running state and removes
                                        //the wait descriptor from the given waitqueue if still queued.
        if (signal_pending(current))  // whether we were awakened by a signal
            return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
    return 0;
}
```

spacefree

```
static int spacefree(struct scull_pipe *dev)
{
   if (dev->rp == dev->wp)
        return dev->buffersize - 1;
   return ((dev->rp + dev->buffersize - dev->wp) % dev->buffersize) - 1;
}
```

poll and select system call

- Nonblocking应用经常使用：
  - 确定自己是否可以非阻塞地读/写一个或多个打开的文件
  - 也可以用于阻塞一个进程，直到一组文件中的任一个可读/写

- The driver **adds a wait queue to the poll_table structure** by calling the function poll_wait:

```
void poll_wait (struct file *,
                wait_queue_head_t *,
                poll_table *);
```