空天信息安全与可信计算
教育部重点实验室

Linux分析与安全设计

1 第四章 Linux I/O设备和驱动安全






第四章 Linux I/O设备和驱动安全

- Linux 设备驱动和类型
- Linux I/O操作类型
- 内核与驱动的关系
- Linux驱动开发
- Linux内核调试
- 键盘过滤驱动原理与实现

Linux设备类型

什么是设备驱动？

- 使计算机和设备进行相互通信的特殊程序。相当于硬件的接口，操作系统只有通过这个接口，才能控制硬件设备的工作

- 驱动程序被比作“ 硬件的灵魂”、“硬件的主宰”、“硬件和系统之间的桥梁”

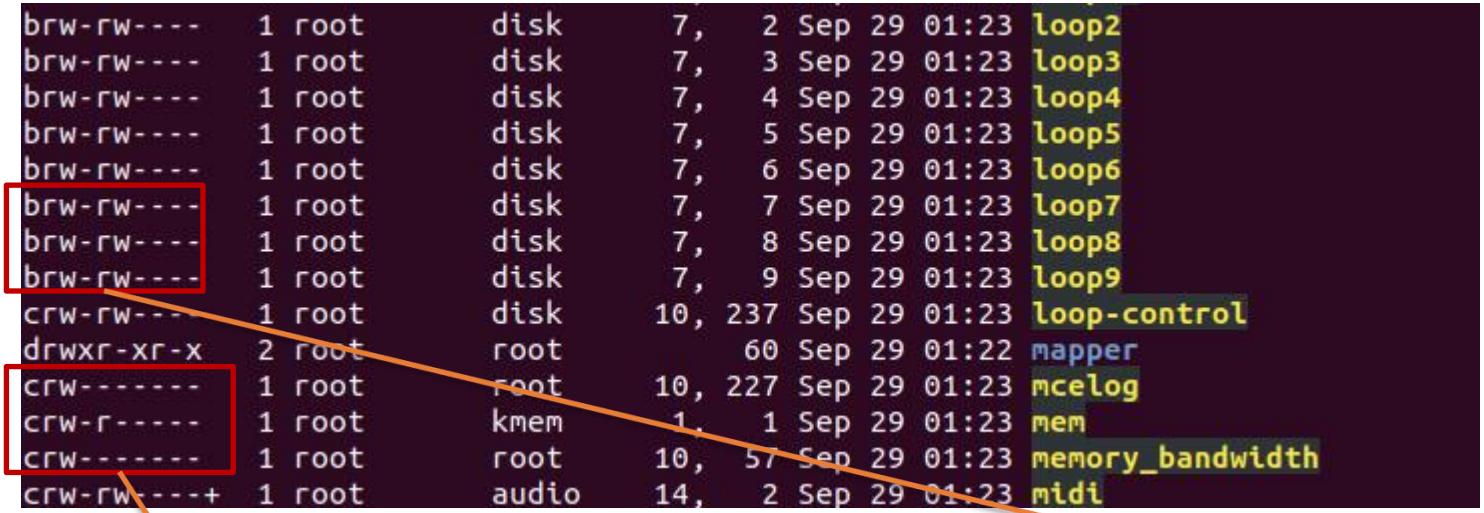
- 驱动程序和硬件是通过如下方式通信的：
  - 硬件上通常包括寄存器和I/O空间
  - 驱动程序通过特殊指令来读写这些寄存器或者I/O空间，从而实现驱动程序和硬件的通信。

Linux设备类型

- Linux设备驱动分类
  - ✓ **字符设备**以字节为单位的I/O传输，这种字符流的传输率通常比较低，常见的字符设备有鼠标、键盘、触摸屏等
  - ✓ **块设备**是以块为单位进行传输的，常见的块设备是磁盘
  - ✓ **网络设备**是一种比较特殊的设备，涉及网络协议层

Linux设备节点

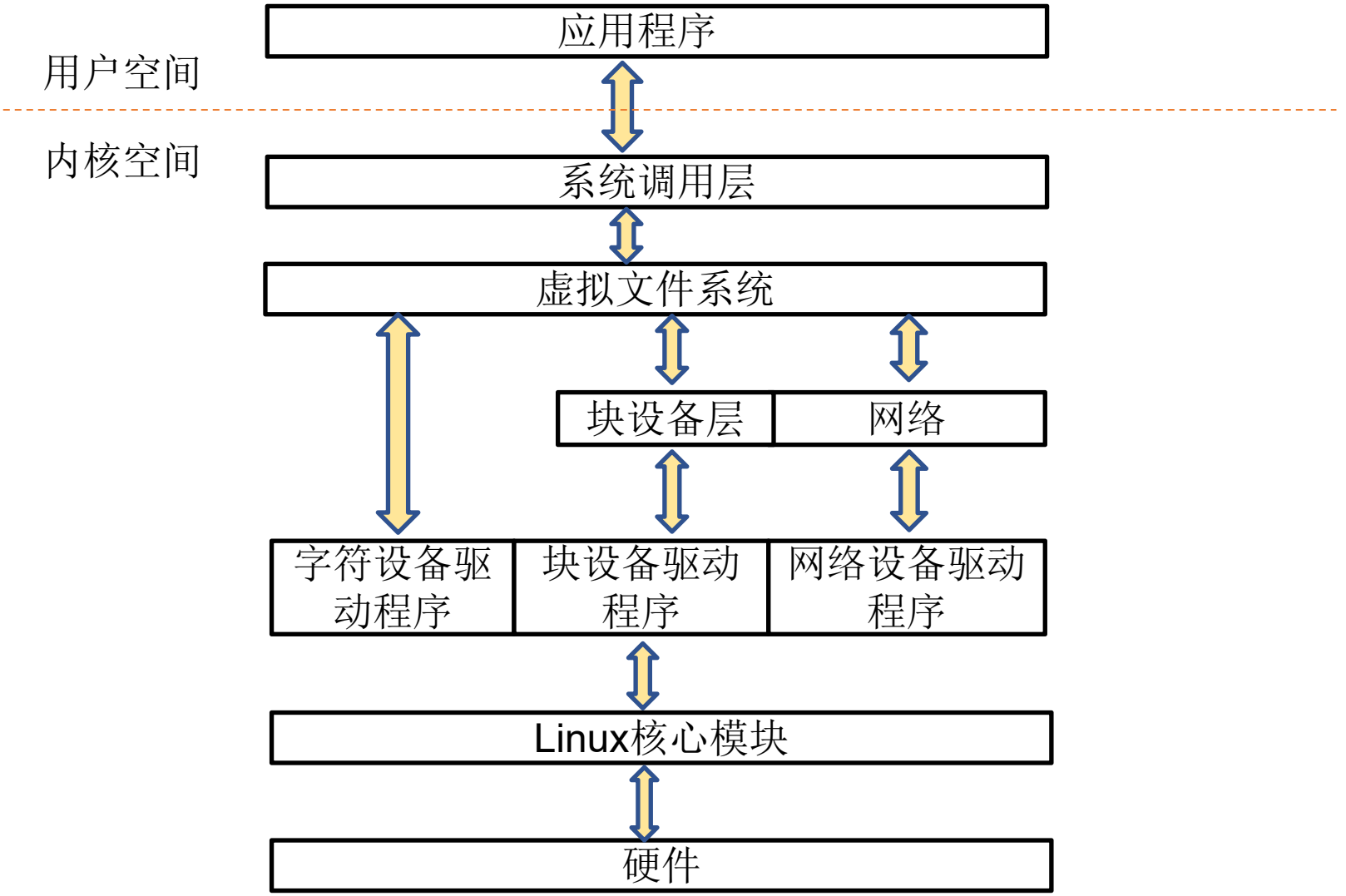
ls –al /dev 查看dev目录下的设备文件



b开头的是块设备文件

c开头的是字符设备文件

Linux设备驱动框架



Linux设备驱动框架

- Linux设计哲学，一切皆文件
- Linux各种设备都以文件的形式存放在/dev目录下，称为设备文件
- dev目录是一个动态生成的、使用devtmpfs虚拟文件系统挂载的、基于RAM的虚拟文件系统
- 应用程序可以打开、关闭和读写这些设备文件，完成对设备的操作，就像操作普通的数据文件一样
- 如果应用程序想使用驱动程序提供的服务或者操作设备，那么需要通过访问该设备文件来完成

Linux设备驱动框架

```
jwang@ubuntu:/dev$ ls
agpgart             loop3               rtc         tty25
autofs              loop4               rtc0        tty26
block               loop5               sda         tty27
bsg                 loop6               sda1        tty28
btrfs-control       loop7               sda2        tty29
bus                 loop-control        sda5        tty3
cdrom               mapper              sg0         tty30
cdrom1              mcelog              sg1         tty31
cdrw                mem                 sg2         tty32
cdrw1               net                 shm         tty33
char                network_latency     snapshot    tty34
console             network_throughput  snd         tty35
core                null                sr0         tty36
cpu                 port                sr1         tty37
cpu_dma_latency     ppp                 stderr      tty38
```

主设备号和次设备号

- 为了管理Linux下的所有设备，系统为设备编了号，每个设备号又分为主设备号和次设备号。主设备号用来区分不同种类的设备，而次设备号用来区分同一类型的多个设备

```
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  5 ttyprintk
  6 lp
  7 vcs
```

cat /proc/device 查看主设备号

```
crw--w----  1 root  tty  4,  22 Sep 28 09:09 tty22
crw--w----  1 root  tty  4,  23 Sep 28 09:09 tty23
crw--w----  1 root  tty  4,  24 Sep 28 09:09 tty24
crw--w----  1 root  tty  4,  25 Sep 28 09:09 tty25
crw--w----  1 root  tty  4,  26 Sep 28 09:09 tty26
crw--w----  1 root  tty  4,  27 Sep 28 09:09 tty27
crw--w----  1 root  tty  4,  28 Sep 28 09:09 tty28
```

主设备号为4，次设备号分别为22、23、24

Linux设备编号API

- 设备号是系统中非常珍贵的资源，内核必须避免发生两个设备驱动使用同一个主设备号的情况
- include/linux/fs.h头文件中包含分配设备编号的API函数

```
/* fs/char_dev.c */
#define CHRDEV_MAJOR_HASH_SIZE  255
extern int alloc_chrdev_region(dev_t *, unsigned, unsigned, const char *);
extern int register_chrdev_region(dev_t, unsigned, const char *);
```

register_chrdev_region函数需要主动指定设备号，可以连续分配多个，需要由驱动程序编写者来保证分配的主设备号没有被使用过

alloc_chrdev_region函数会自动分配一个主设备号，由操作系统来保证分配的主设备号没有被使用过

Linux里面设备号用dev_t表示，实际上就是u32类型

字符设备驱动

- 字符设备驱动管理的核心对象是以字符为数据流的设备。
- 从Linux内核设计的角度来看，需要有一个数据结构来对其进行抽象和描述，Linux内核使用struct cdev数据结构

字符设备驱动

include/linux/cdev.h中描述的cdev数据结构

```
struct cdev {
        struct kobject kobj;
        struct module *owner;
        const struct file_operations *ops;
        struct list_head list;
        dev_t dev;
        unsigned int count;
};
```

字符设备驱动

- 结构体解释
  - kobj
    用于linux设备驱动对象，结构体本身不单独使用，而是嵌套在其他高层结构中，用于组成拓扑关系
  - owner
    字符设备驱动程序所在的内核模块对象指针
  - ops
    字符设备驱动中最关键的一个操作函数，在和应用程序交互的过程中起到桥梁枢纽的作用

字符设备驱动

- 结构体解释
  - list
    用来将字符设备串成一个链表
  - dev
    字符设备的设备号，由主设备号和次设备号组成
  - count
    同属一个主设备号的次设备号的个数

块设备驱动

- 块设备驱动模型主要针对磁盘，Flash等存储类设备
- 块设备是一种具有一定结构的随机存取设备，对这种设备的读写是按**块**(所以叫块设备)进行的
- 块设备使用缓冲区来存放暂时的数据，待条件成熟后，从缓存一次性写入设备或者从设备一次性读到缓冲区

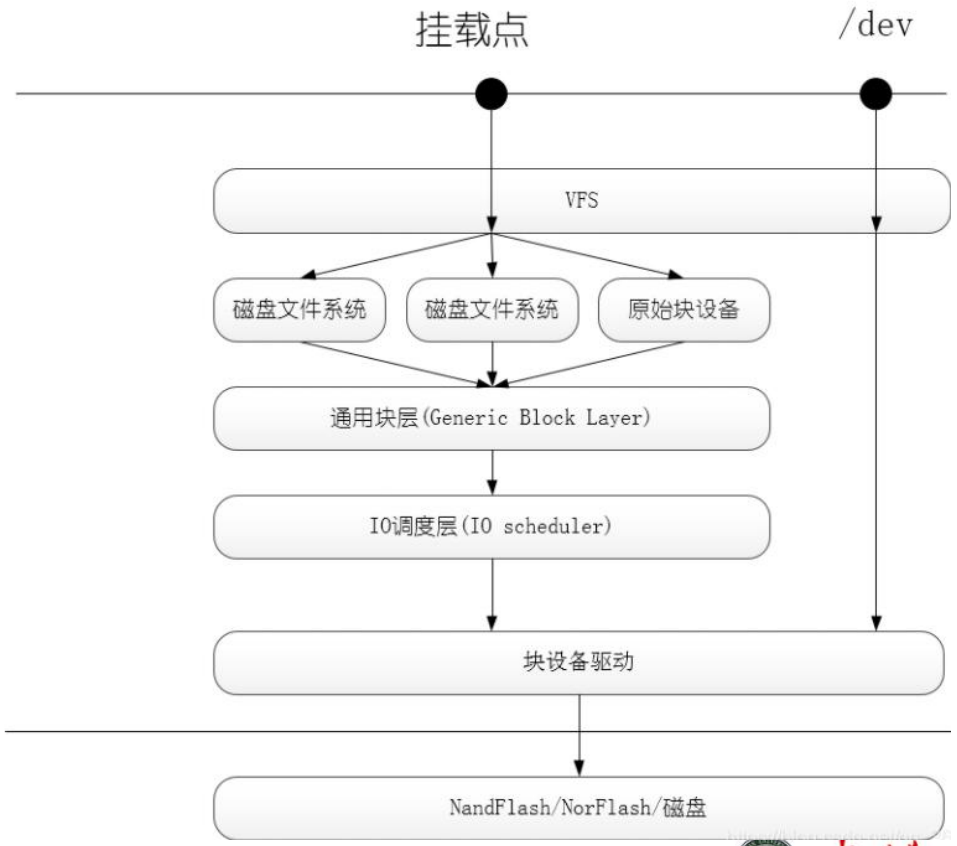
块设备驱动

- 块设备与字符设备的区别

设备	字符设备	块设备
基本单位	1byte	块，大小不尽相同，内核一半为512bytes
访问方式	顺序访问	随机访问
缓存	没有缓存，实时操作	有缓存，不是实时操作
接口	一般提供接口给应用层	块设备一般提供接口给文件系统
调用方式	是被用户程序调用	由文件系统程序调用

块设备模型

- 应用层程序有两种方式访问一个块设备：/dev和文件系统挂载点，前者和字符设备一样，通常用于配置，后者在mount之后就可以通过文件系统直接访问一个块设备



块设备驱动调用过程

- 块设备为了优化读写请求从而引入了“缓冲区”，所有数据的修改都是在“缓冲区”内修改，在关闭文件之前再写入到硬件中
- 内核使用一个buffer_head数据结构来描述缓冲区

```
struct buffer_head {
        unsigned long b_state;
        struct buffer_head *b_this_page;
        struct page *b_page;
        sector_t b_blocknr;
        size_t b_size;
        char *b_data;
        struct block_device *b_bdev;
        bh_end_io_t *b_end_io;
        void *b_private;
        struct list_head b_assoc_buffers;
        struct address_space *b_assoc_map;
        atomic_t b_count;
};
```

块设备驱动调用过程

- 结构体部分解释
  - b_state
    表示缓冲区状态，比如是否是脏的需要写回磁盘，是否可以被多个进程共享等
  - b_this_page
    页面内的缓冲区链表
  - b_page
    b_page指向了这个缓冲块对应的页缓存的物理页

块设备驱动调用过程

- 结构体部分解释
  - b_blocknr
    对应的磁盘块号
  - b_size
    映像的大小
  - b_data
    指向block所在的位置
  - b_count
    b_count引用计数，当b_count不为0的时候，这个缓冲块不能被交换出去

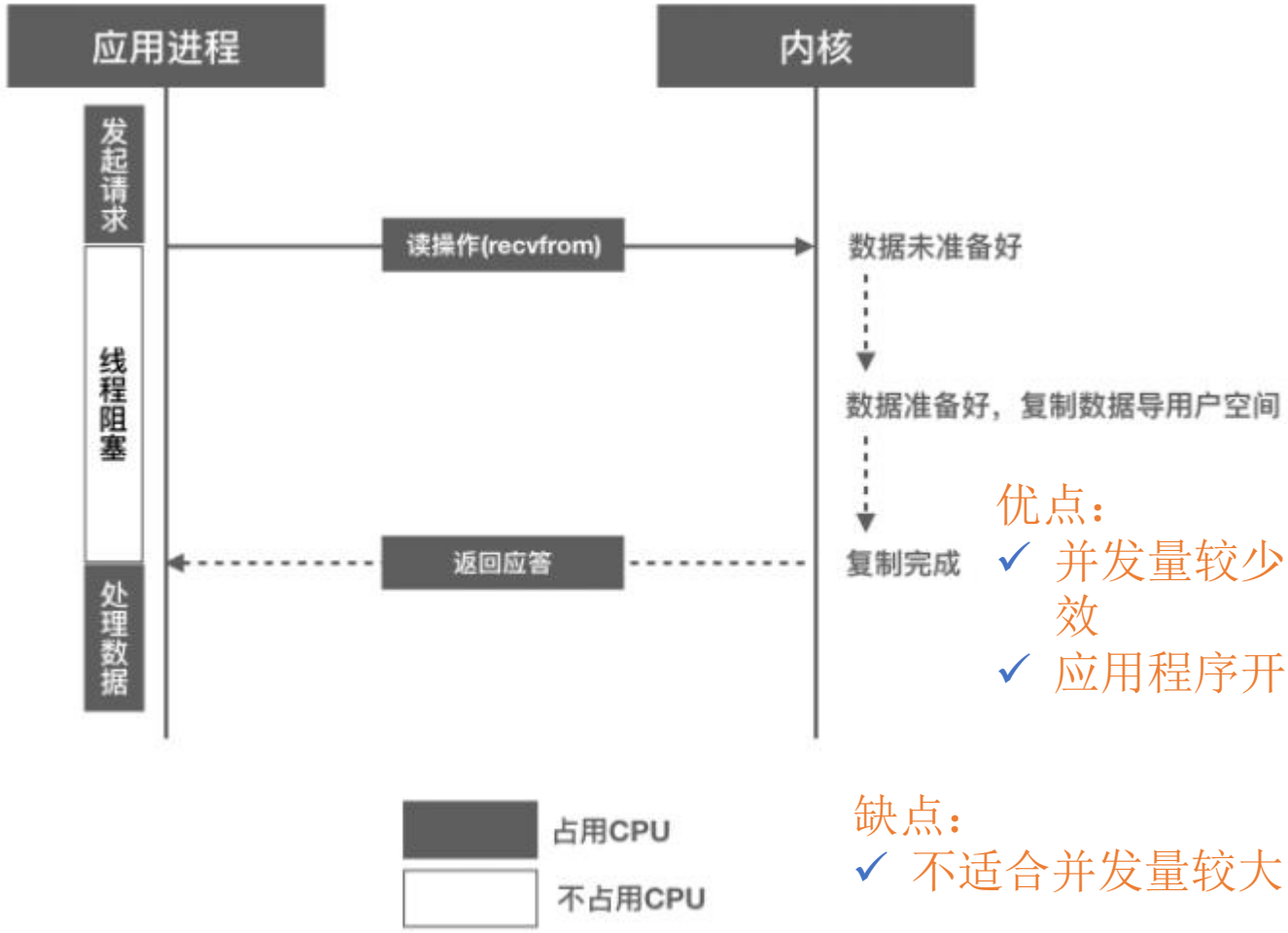
Linux I/O操作类型

- blocking I/O – 阻塞I/O
- nonblocking I/O – 非阻塞I/O
- signal-driven I/O – 信号驱动式I/O（异步阻塞）
- asynchronous I/O – 异步I/O
- io multiplexing – I/O多路复用

同步阻塞I/O

- 当应用线程发起I/O请求时，内核收到请求后进入等待数据阶段，此时应用线程会处于阻塞的状态。直到内核给应用线程返回结果，此时应用线程才解除阻塞的状态
- 同步阻塞模式的特点是内核 I/O 在执行等待数据读取到内核空间和将数据复制到用户空间的两个阶段，应用线程都被阻塞
- 同步阻塞模式简单直接，没有下面几种模式的线程切换、回调、通知等消耗，在并发量较少的网络通信场景下是最好的选择

同步阻塞I/O



优点：

- ✓ 并发量较少的网络通信场景较高效
- ✓ 应用程序开发简单

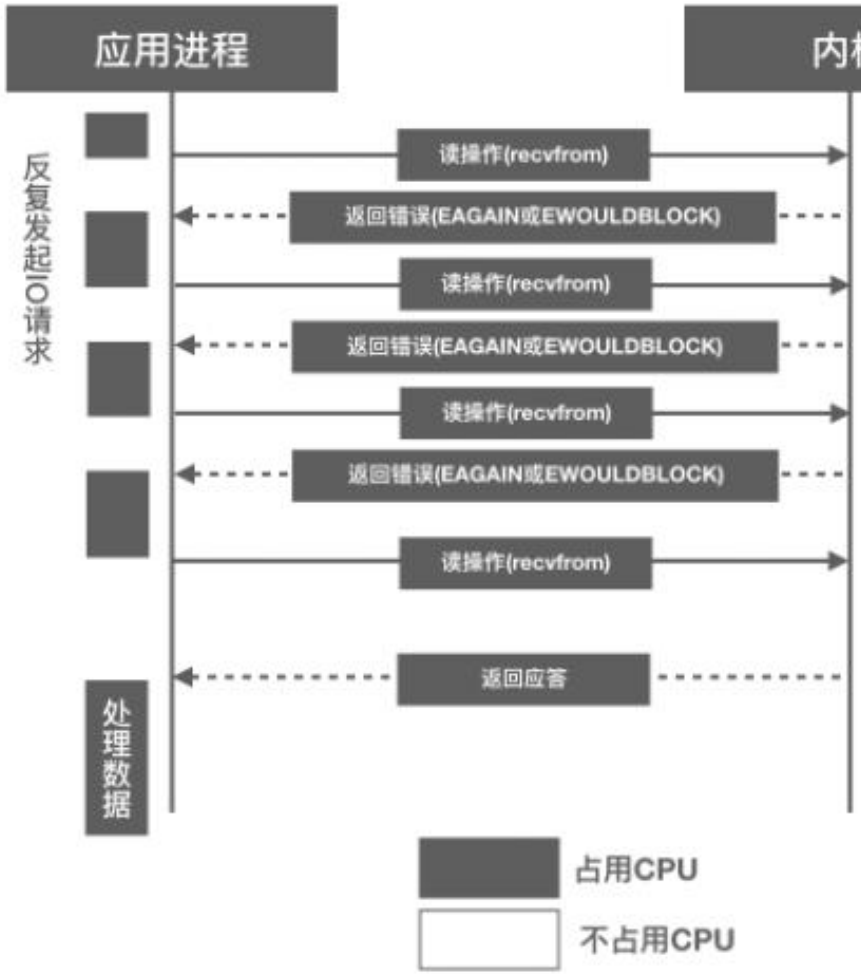
缺点：

- ✓ 不适合并发量较大的网络通信场景

同步非阻塞I/O

- 当用户线程发起读操作时，如果内核的I/O数据还没有准备好，那么它不会阻塞掉用户线程，而是会直接返回一个EAGAIN/EWOULDBLOCK 错误
- 从用户线程的角度，它发起一个读操作后立即就得到了一个结果，用户进程判断结果是EAGAIN/EWOULDBLOCK 之后会再次发起读操作
- 一旦内核中的I/O数据准备好了，并且又再次收到了用户进程的请求，那么它马上就将数据拷贝到了用户内存，然后返回

同步非阻塞I/O



优点：

- ✓ 在内核I/O数据准备阶段不会阻塞应用线程，适合对线程阻塞敏感的网络应用

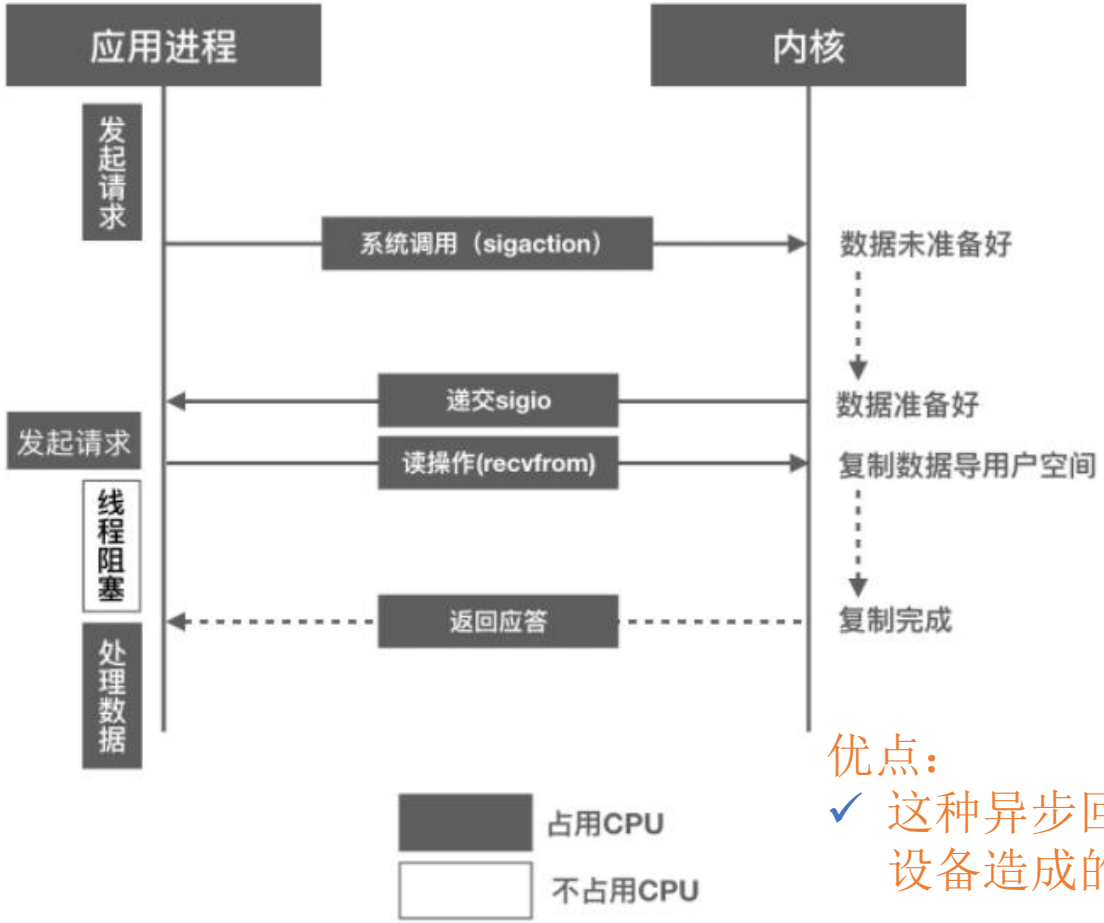
缺点：

- ✓ 轮询查询内核I/O数据状态，耗费大量CPU，效率低
- ✓ 需要不断轮序，增加开发难度

信号驱动I/O

- 信号驱动模式利用linux信号机制，通过sigaction函数将sigio读写信号以及handler回调函数注册到内核队列中，注册后应用进程不阻塞，可以去干别的工作
- 当网络I/O状态发生变化时触发SIGIO中断，通过调用应用程序的handler通知应用程序网络I/O就绪了
- 信号驱动的前半部分操作是异步行为，后面的网络数据操作仍然属于同步阻塞行为

信号驱动I/O



优点：

- ✓ 这种异步回调方式避免用户或内核主动轮询设备造成的资源浪费

缺点：

- ✓ handler是在中断环境下运行，多线程不稳定，而且平台兼容性不好，不是一个完善可靠的解决方案，实际应用场景少，较复杂，开发难度大

异步非阻塞I/O

- 异步I/O通过一系列异步API实现，是五种I/O模式中唯一个真正的异步模式
- 异步模式的读操作通过调用内核的aio_read函数来实现。应用线程调用aio_read，递交给内核一个用户空间下的缓冲区
- 内核收到请求后立刻返回，不阻塞应用线程。当网络设备的数据到来后，内核会自动把数据从内核空间拷贝到aio_read函数递交的用户态缓存。拷贝完成后以信号的方式通知用户线程，用户线程拿到数据后就可以执行后续操作

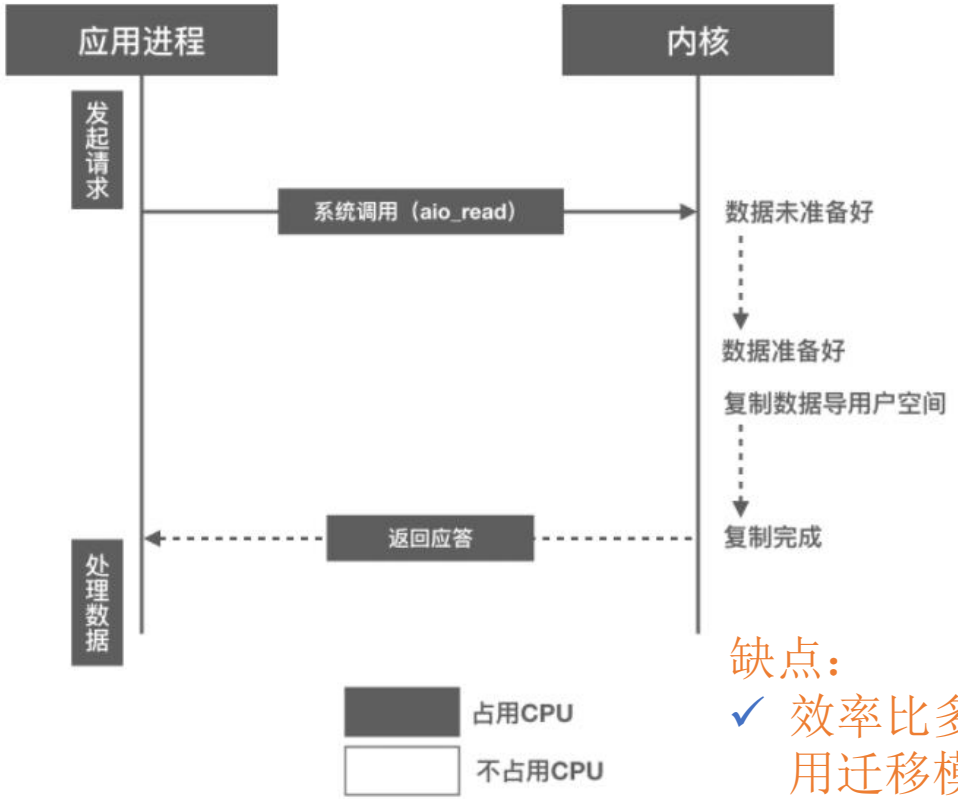
异步非阻塞I/O

- 异步I/O模式与信号驱动I/O的区别在于

  - ✓ 信号驱动I/O由内核通知应用程序什么时候可以开始I/O操作，异步I/O则由内核告诉应用程序I/O操作何时完成
  - ✓ 异步I/O主动把数据拷贝到用户空间，不需要调用recvfrom方法把数据从内核空间拉取到用户态空间
  - ✓ 异步I/O是一种推数据的机制，相比于信号处理I/O拉数据的机制效率会更高

- 异步I/O还是属于比较新的I/O模式，需要操作系统支持，Linux2.5版本首次提供异步I/O，Linux2.6及以后版本，异步I/O的API都属于标准提供。异步I/O目前没有太多的应用场景

异步非阻塞I/O



优点：

✓ 纯异步，高效率高性能

缺点：

✓ 效率比多路复用模式没有质的提升，成熟应用迁移模式的动力不足，一直没有大规模成熟应用来支撑

✓ 较复杂，开发难度大

I/O多路复用

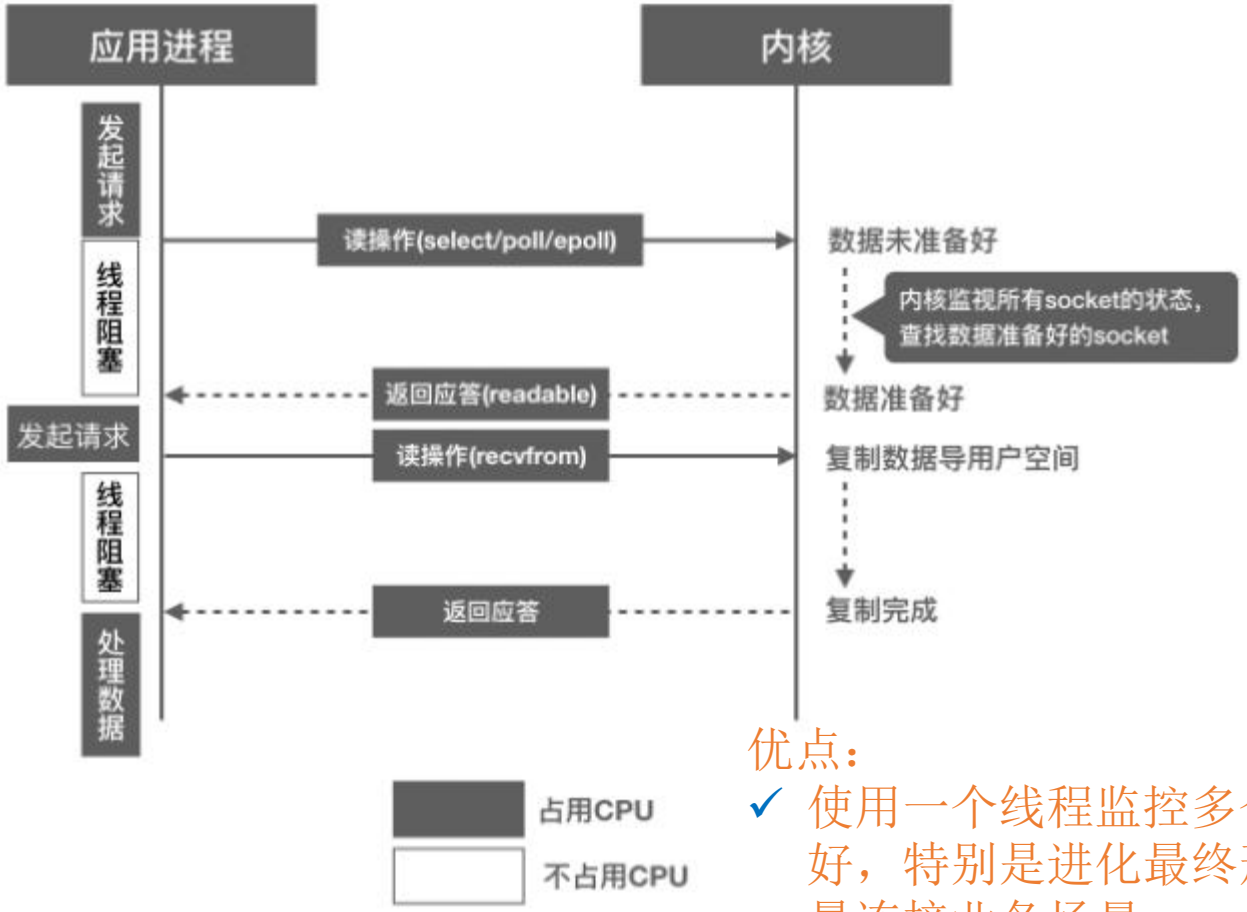
- 多路复用是目前大型互联网应用中最常见的一种I/O模型
- 应用进程中有一个**I/O状态管理器**，多个网络I/O注册到这个管理器上，管理器使用一个线程调用内核API来监听所有注册的网络I/O的状态变化情况，一旦某个连接的网络I/O状态发生变化，能够通知应用程序进行相应的读写操作
- 多路复用本质上是同步阻塞，但与传统的同步阻塞多线程模型相比，I/O 多路复用的最大优势是在处理I/O高并发场景时只使用一个线程就完成了大量的网络I/O状态的管理工作，系统资源开销小

I/O多路复用

多路复用的基本工作流程：
1.应用程序将网络I/O注册到状态管理器；
2.状态管理器通过调用内核API来确认所管理的网络I/O的状态；
3.状态管理器探知到网络I/O的状态发生变化后，通知应用程序进行实质的同步阻塞读写操作

目前Linux主要有三种状态管理器：select，poll，epoll。epoll是Linux目前大规模网络并发程序开发的首选模型，在绝大多数情况下性能远超select和poll

I/O多路复用



优点：

- ✓ 使用一个线程监控多个网络连接状态，性能好，特别是进化最终形态epoll模式，适合大量连接业务场景

缺点：

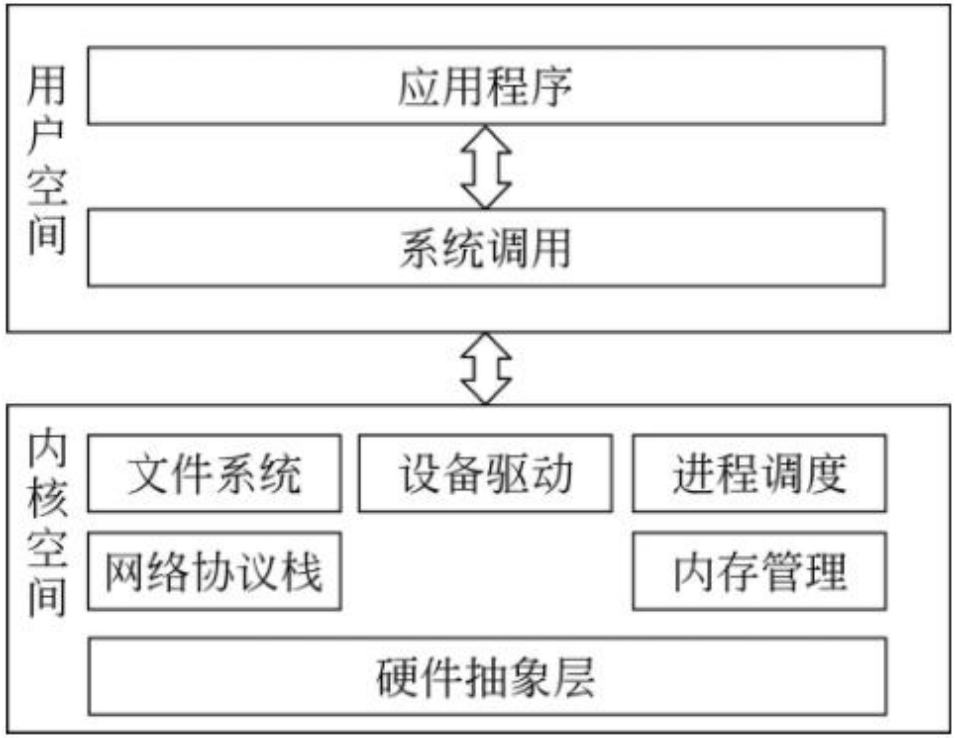
- ✓ 较复杂，应用开发难度大

I/O模型总结

- 五种Linux的I/O模式各有特色，各有自己的应用场景
- 一些简单的低并发Socket通信时大多数还是使用多线程加同步阻塞的方式，效率和其他模式差不多，实现起来会简单很多
- 目前市面上流行的高并发网络通信框架都是使用的多路复用模型，经过大量实际项目验证，多路复是目前最成熟的高并发网络通信I/O模型。信号驱动I/O目前不太成熟，基本上没有见过使用场景
- 纯异步模式，内核把所有事情做了，看起来很美好，Java也提供了响应的实现，但由于效率比多路复用模式没有质的提升，成熟应用迁移模式的动力不足，一直没有大规模成熟应用来支撑

内核与驱动的关系

- Linux操作系统与设备驱动之间的关系如图所示。Linux内核就是一个调用库，应用程序通过调用Linux提供的API函数来实现操作，Linux内核通过与驱动通信实现对硬件的有效管理



Linux驱动开发

Linux内核支持很多种模块，驱动程序就是其中最重要的一种

- 可以使用insmod（insert module的缩写）命令将模块加入正在运行的内核，也可以使用rmmod（remove module的缩写）命令将一个未使用的模块从内核中删除，类似于Windows下的DLL库
- 模块在内核启动时装载称为静态装载，在内核已经运行时装载称为动态装载。模块可以扩充内核所期望的任何功能，但通常用于实现设备驱动程序

Linux驱动开发

应用程序开发与驱动程序开发的差异：

- 内核及驱动程序开发时不能访问C库，因为C库是使用内核中的系统调用来实现的，而且是在用户空间实现的。驱动程序只能访问有限的系统调用，或者汇编程序
- 内核及驱动程序开发时必须使用GNU C，因为Linux操作系统从一开始就使用的是GNU C
- 内核支持异步中断、抢占和SMP，因此内核及驱动程序开发时必须时刻注意同步和并发。

Linux驱动开发

一个模块的最基本框架代码如下：

```
int __init xxx_init(void)
{
    /*这里是模块加载时的初始化工作*/
    return 0;
}
void __exit xxx_exit(void)
{
    /*这里是模块卸载时的销毁工作*/
}
module_init(xxx_init);                    /*指定模块的初始化函数的宏*/
module_exit(xxx_exit);                    /*指定模块的卸载函数的宏*/
```

Linux驱动开发

Linux下模块开发重要的两个函数：

- module_init(hello_init)
  - ✓ 驱动代码只有一个入口点和一个出口点，把驱动加载到内核中，会执行module_init函数定义的函数，在上面代码中就是hello_init函数
- module_exit(hello_exit)
  - ✓ 当驱动从内核被卸载时，会调用module_exit函数定义的函数，在上面代码中就是hello_exit函数

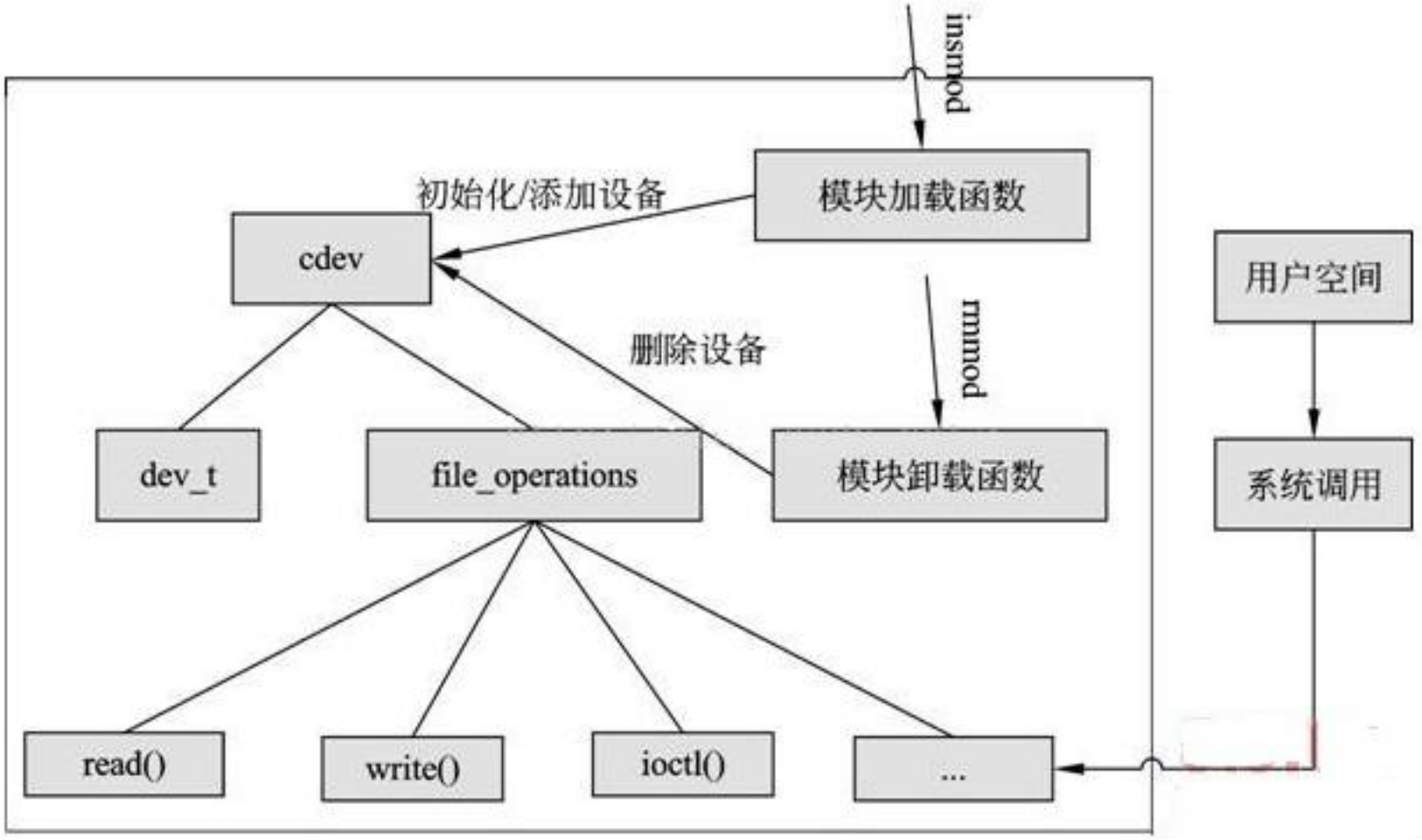
Linux驱动开发

一个简单的Hello world内核模块

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

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("Hcamal");

int hello_init(void)
{
    printk(KERN_INFO "Hello World\n");
    return 0;
}

void hello_exit(void)
{
    printk(KERN_INFO "Goodbye World\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- ✓ 需要使用内核函数printk
- ✓ 驱动代码只有一个入口点和一个出口点
- ✓ 把驱动加载到内核中，会执行module_init函数定义的函数，在上面代码中就是hello_init函数
- ✓ 当驱动从内核被卸载时，会调用module_exit函数定义的函数，在上面代码中就是hello_exit函数

Linux驱动开发



Linux驱动开发

一个简单的字符设备驱动

（1）根据外部设备的特点，实现file_operations结构所需要的函数
（2）调用函数cdev_alloc()函数向系统动态申请一个cdev结构实例
（3）调用函数cdev_init()初始化cdev实例，并建立cdev实例与file_operations实例之间的连接
（4）调用函数alloc_chrdev_region()向系统申请一个设备号
（5）调用函数cdev_add()向系统添加一个设备
（6）调用函数cdev_del()从系统删除一个cdev结构实例

例子

Linux驱动开发

- 驱动程序提供的一组设备驱动接口函数给linux内核使用，具体是通过一个叫做file operations的结构体注册给linux内核的

```
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 (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **);
                                                                    1583,14-21    59%
```

File operations

（1）open： int (*open) (struct inode *, struct file *);（打开设备，响应open系统调用）

（2）release： int (*release) (struct inode *, struct file *);（关闭设备，响应close系统调用）

（3）read：ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);（从设备读取数据，响应read系统调用）

（4）write：ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);（向设备写入数据，响应write系统调用）

分配设备号

（1）静态申请

开发者自己选择一个数字作为主设备号，通过函数 register_chardev_region 向内核申请

（2）动态分配：

使用 alloc_chrdev_region 由内核分配一个可用的主设备号（推荐使用）

分配设备驱动结构体

可以采用静态和动态两种方法

（1）静态分配：struct cdev dev;

（2）动态分配：

```
struct cdev*  dev = cdev_alloc();
```

设备初始化

cdev变量的初始化使用cdev_init()函数来完成：

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

cdev：待初始化的cdev结构

fops：设备对应的操作函数集

建立设备与驱动操作方集file_operations之间的连接关系

添加和删除设备

✓字符设备的注册使用cdev_add()函数来完成

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

(1)设备指针

(2)主设备号 dev

(2)次设备号 count

✓字符设备的注销使用cdev_del()函数来完成

```
void cdev_del(struct cdev *p); :
```

/proc文件系统

- /proc文件系统是linux提供的一个虚拟的文件系统，它存在于内存当中
- /proc下面的每个文件都绑定于一个内核函数，用户读取文件时，该函数动态地生成文件的内容。也可以通过写/proc文件修改内核参数
- /proc文件系统相当于在内核和用户空间打开了一个通信窗口
- 利用cat /proc/devices查看申请到的设备名，设备号

/proc文件系统

/proc目录下的文件分析　/proc/$pid 关于进程$pid的信息目录。每个进程在/proc 下有一个名为其进程号的目录。
/proc/cmdline　内核启动的命令行
/proc/cpuinfo　处理器信息，如类型、制造商、型号和性能。
/proc/devices　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
/proc/dma　显示当前使用的DMA通道。
/proc/filesystems　列出了可供使用的文件系统类型，通常是编入内核的文件系统类型，但可以通过模块加入新的类型
/proc/interrupts　显示使用的中断号，中断名称，以及这些中断从系统启动后产生的次数
/proc/ioports　当前使用的I/O端口。
/proc/kallsyms　内核符号表。安装新的模块后，会在这里体现出来
/proc/kcore　系统物理内存映象。与物理内存大小完全一样，但不实际占用这么多内存；(记住：除非拷贝到文件中，/proc下没有任何东西占用任何磁盘
/proc/kmsg　内核输出的消息。也被送到syslog。
/proc/loadavg　系统的平均负载，前3个是过去1分钟，5分钟，15分钟的负载，然后是正在运行的任务数和总任务数，最后是上次运行的进程号
/proc/meminfo　存储器使用信息，包括物理内存和swap。
/proc/modules　当前加载了哪些核心模块。
/proc/partitions　系统当前挂载硬盘的分区信息
/proc/pci　系统的pci总线信息
/proc/net　网络协议状态信息。
/proc/self　到查看/proc的程序的进程目录的符号连接。当2个进程查看/proc时，是不同的连接。这主要便于程序得到它自己的进程目录。
/proc/slabinfo　系统中slab缓存的分配信息
/proc/stat　系统的一些状态信息
/proc/swaps　系统采用的交换区信息
/proc/sysrq-trigger　用于启动sysRq键 $>echo 1 > sysrq-trigger
/proc/uptime　系统启动的时间长度和空闲的时间长度。供uptime使用
/proc/version　内核版本

查看设备驱动是否加载：　cat /proc/devices

/dev下添加设备

两种方法：

（1）使用mknod命令

mknod 设备名 设备类型 主设备号 从设备号

（2）udev

udev是2.6内核版本后 Linux 默认的设备管理工具。udev 以守护进程的形式运行，通过侦听内核发出来的 uevent 来管理 /dev目录下的设备文件。

Kobject

- Kobject 是Linux 2.6引入的新的设备管理机制，在内核中由struct kobject表示。
- 通过这个数据结构使所有设备在底层都具有统一的接口，kobject提供基本的对象管理Kobject是组成设备模型的基本结构，bus，devices， drivers等都是通过kobject连接起来并且形成了一个树状结构。这个树状结构就与/sys相对应

类型	所包含的内容	内核数据结构	对应/sys项
设备 (Devices)	设备是此模型中最基本的类型，以设备本身的连接按层次组织	struct device	/sys/devices/*/*/.../
驱动 (Drivers)	在一个系统中安装多个相同设备，只需要一份驱动程序的支持	struct device_driver	/sys/bus/pci/drivers/*/
总线 (Bus)	在整个总线级别对此总线上连接的所有设备进行管理	struct bus_type	/sys/bus/*/
类别 (Classes)	这是按照功能进行分类组织的设备层次树；如 USB 接口和 PS/2 接口的鼠标都是输入设备，都会出现在/sys/class/input/下	struct class	/sys/class/*/

Kobject

Kobject数据结构：

```
struct kobject {
        const char  *name;          // kobject的名字，且作为一个目录的名字
         struct list_head    entry;          // 用于挂接链表
        struct kobject                  *parent;     // 父亲节点
        struct kset                     *kset;          // 所属的kset
        struct kobj_type    *ktype;         // 指向kobject的属性描述符
        struct kernfs_node              *sd;  //对应sysfs的文件目录
        struct kref                     kref;           // 引用计数
        unsigned int state_initialized:1; // 是否初始化标志
        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;
};
```

/sys文件系统

- sys文件系统是 Linux 内核中设计较新的一种虚拟的基于内存的文件系统
- sys文件系统与 proc 有些类似，但除了与 proc相同的具有查看和设定内核参数功能之外，还有为 Linux 统一设备模型作为管理之用
- 相比于 proc 文件系统，使用 sysfs导出内核数据的方式更为统一，并且组织的方式更好
- sysfs 提供了kobject对象层次结构的视图

/sys文件系统

```
jwang@ubuntu:/sys/class$ ls
ata_device  dma          input      power_supply  scsi_disk      vc
ata_link    dmi          leds       ppdev         scsi_generic   virtio-ports
ata_port    drm          mdio_bus   ppp           scsi_host      vtconsole
backlight   extcon       mem        printer       sound          watchdog
bdi         firmware     misc       pwm           spi_host
block       graphics     mmc_host   regulator     spi_master
bluetooth   hidraw       net        rfkill        spi_transport
bsg         hwmon        pci_bus    rtc           thermal
devfreq     i2c-adapter  powercap   scsi_device   tty
```

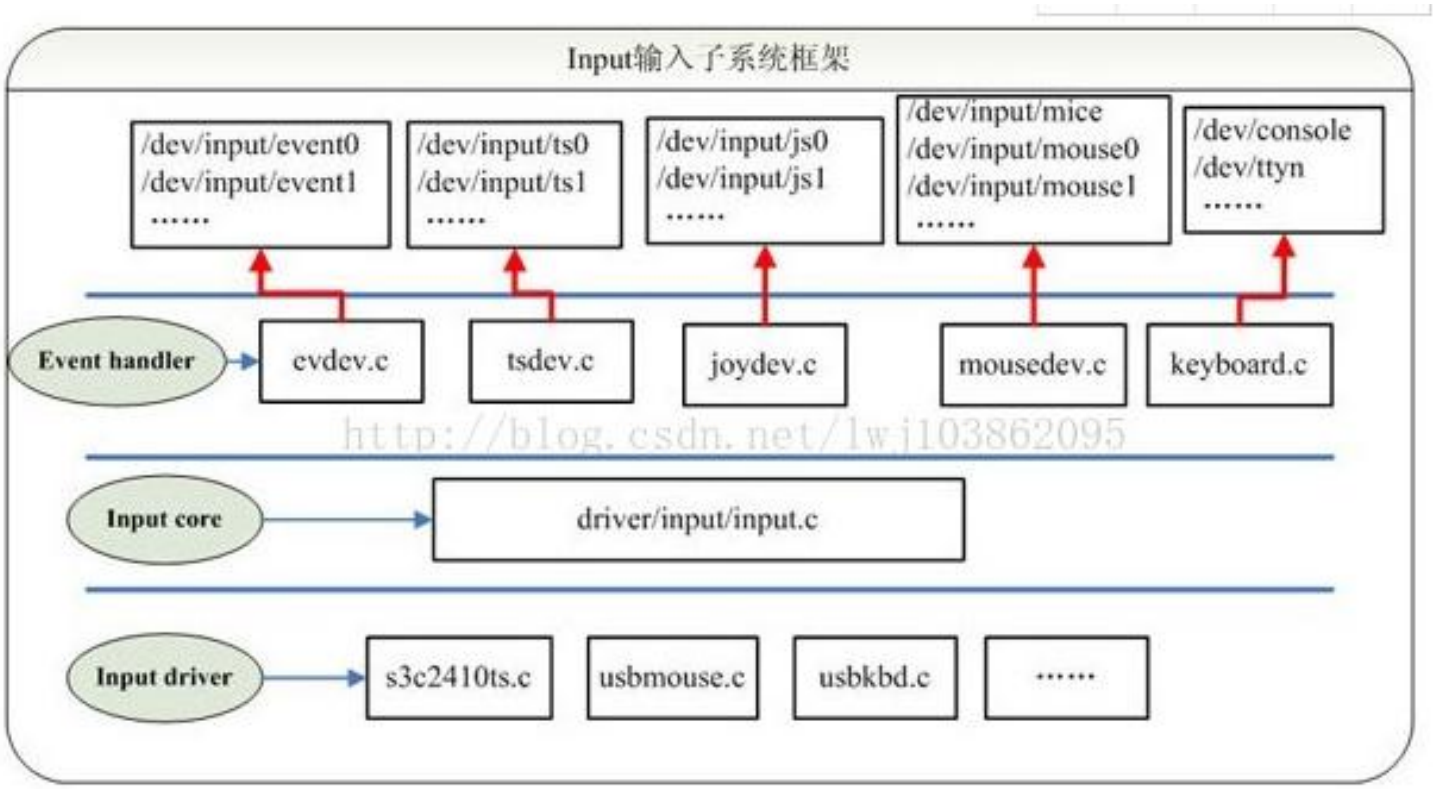
Linux键盘驱动

- Linux 输入子系统是 Linux内核用于管理各种输入设备（键盘，鼠标，遥控杆，书写板等等）的。
- Linux输入子系统从上到下由三层实现，分别为：
  - 输入子系统事件处理层（EventHandler）
  - 输入子系统核心层（InputCore）
  - 输入子系统设备驱动层

- ✓ 设备驱动层提供对硬件各寄存器的读写访问和将底层硬件对用户输入访问的响应转换为标准的输入事件，再通过核心层提交给事件处理层
- ✓ 核心层对下提供了设备驱动层的编程接口，对上又提供了事件处理层的编程接口
- ✓ 事件处理层就为我们用户空间的应用程序提供了统一访问设备的接口和驱动层提交来的事件处理

Linux键盘驱动

Linux的输入子系统框架如下图所示：



Linux键盘驱动

输入设备 —— input_dev

- 每个输入设备驱动中都实例化了一个input_dev对象，用于记录设备硬件相关信息、事件位图设置（支持的事件类型）以及其他参数

```
struct input_dev {
        const char *name;   // 设备名称
        struct input_id id; // 设备id，用于与input_handler匹配
        ......
        unsigned long evbit[BITS_TO_LONGS(EV_CNT)];      // 事件位图
        unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];    // 按键事件位图
        unsigned long relbit[BITS_TO_LONGS(REL_CNT)];    // 相对位移事件位图
        unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];    // 绝对位移事件位图
        struct device dev;
        struct list_head        h_list;     // 内核链表头
        struct list_head        node;       // 内核链表节点
        ......
};
```

Linux键盘驱动

事件处理器 —— input_handler

- 每个事件处理器也都实例化了input_handler对象，如evdev_handler、mousedev_handler、joydev_handler等等，input_handler类中提供了事件处理、与输入设备匹配连接相关的接口

```
struct input_handler {
        void *private;
        /* 事件处理接口 */
        void (*event)(struct input_handle *handle, unsigned int type, unsigned
        int code, int value);
        void (*events)(struct input_handle *handle,const struct input_value
        *vals, unsigned int count);
        /* 输入设备与事件处理器匹配接口 */
        bool (*match)(struct input_handler *handler, struct input_dev *dev);
        bool legacy_minors;
        int minor;
        ……
};
```

Linux键盘驱动

- 输入设备与事件处理器的匹配连接，input子系统维护了两条重要链表：
  - ✓输入设备链表 —— input_dev_list
  - ✓事件处理器链表 —— input_handler_list
- 两条链表在核心层input.c中静态声明并初始化：

```
1 static LIST_HEAD(input_dev_list);
2 static LIST_HEAD(input_handler_list);
```

Linux键盘驱动

- 当一个输入设备注册时，它会被加入到输入设备链表(input_dev_list)，同时遍历事件处理器链表(input_handler_list)进行匹配：

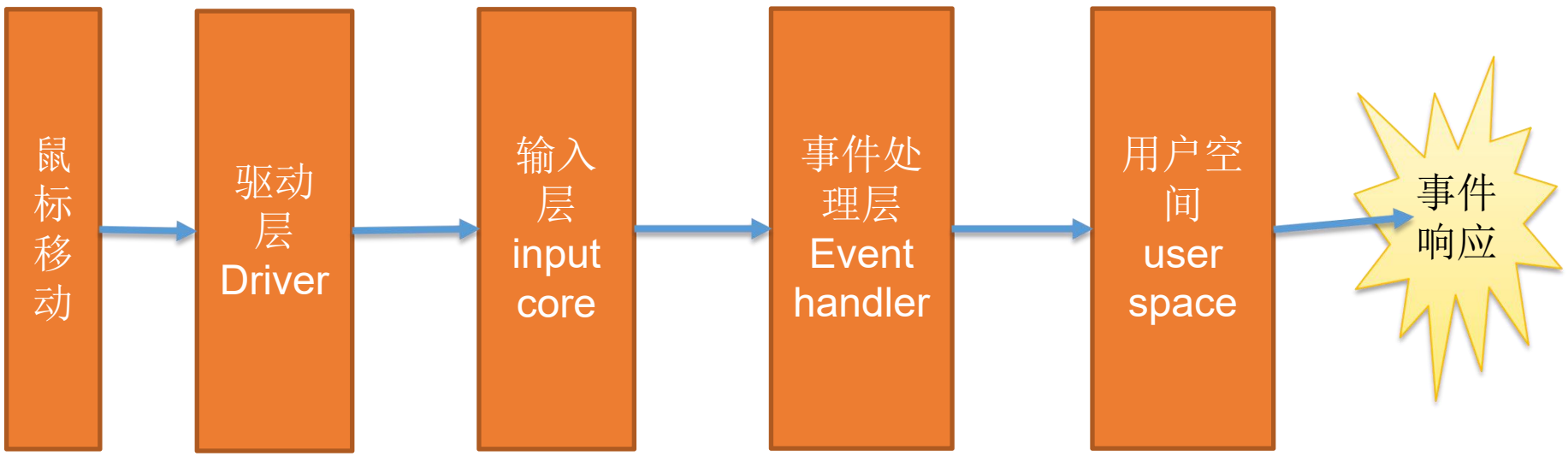
```
int input_register_device(struct input_dev *dev)
{
    ......
    // 将输入设备加入内核链表
        list_add_tail(&dev->node, &input_dev_list);
    // 遍历事件处理器链接匹配
        list_for_each_entry(handler, &input_handler_list, node)
        input_attach_handler(dev, handler);
    ......
}
```

Linux键盘驱动

- 匹配过程以事件处理器的id_table为判断条件1，如果输入设备与事件处理器匹配成功，则调用事件处理器的connect()方法进一步处理
- 输入设备调用input_register_device()函数注册时，如果与通用事件处理器(evdev)成功配对，最后就会生成字符设备节点(/dev/input/eventN)，这就是用户空间获取内核输入事件的接口

键盘过滤驱动原理

事件的响应过程，比如一个鼠标移动事件：



键盘过滤驱动原理

cat /proc/bus/input/devices 可以查看到当前input子系统下面的所有event设备。我们对计算机的输入(包括敲击键盘，移动鼠标等等操作)经过内核(底层驱动,input)处理最后就上报到这些event里面了。而这里event0,event1,..就是用来区分各个外设的

```
rong@ubuntu:~$ cat /proc/bus/input/devices
I: Bus=0019 Vendor=0000 Product=0001 Version=0000
N: Name="Power Button"
P: Phys=LNXPWRBN/button/input0
S: Sysfs=/devices/LNXSYSTM:00/LNXPWRBN:00/input/input0
U: Uniq=
H: Handlers=kbd event0
B: PROP=0
B: EV=3
B: KEY=10000000000000 0

I: Bus=0011 Vendor=0001 Product=0001 Version=ab41
N: Name="AT Translated Set 2 keyboard"
P: Phys=isa0060/serio0/input0
S: Sysfs=/devices/platform/i8042/serio0/input/input1
U: Uniq=
H: Handlers=sysrq kbd event1 leds
B: PROP=0
B: EV=120013
B: KEY=402000000 3803078f800d001 feffffdfffefffff fffffffffffffffe
B: MSC=10
B: LED=7
```

I：设备的ID
N ： 设备的名字
P ： 在系统层次结构中的设备的物理路
S ： sysfs 路径
U ： 设备的唯一识别码（如果设备有）
H ： 与设备关联的输入句柄列表.
B ： 位图
PROP：设备属性.
EV：设备支持的事件类型
KEY：此设备具有的按键/按钮。
MSC：设备支持的各种事件。.
LED：设备上的leds

键盘过滤驱动原理

在/usr/include/linux/input.h中定义了event事件的结构体，API和标准按键的编码等

```
23 struct input_event {
24     struct timeval time;
25     __u16 type;
26     __u16 code;
27     __s32 value;
28 };
```

- ✓ time：结构体内容依次是按键时间。
- ✓ type：事件类型：我们关注的是按键事件
- ✓ code：事件的代码。按键事件的类型代码code是EV_KEY，该代码为设备键盘代码。在该文头文件中已经定义的0-248中不同的键盘按键代码（/usr/include/linux/input.h）
- ✓ value：事件的值。按键事件的类型代码是EV_KEY,当按键按下时值为1，松开时值为0。

Linux内核调试

对Linux内核调试的方法很多，主要有以下几类

- 通过打印函数调试
- 获取内核信息，处理出错信息
- 内核源码调试

内核打印函数

Linux内核标准打印函数printk：

- ✓ printk函数与c库提供的print函数类似，但是前者会提供一个打印等级，内核根据这个等级判断是否在终端或者串口中打印输出
- ✓ include/linux/printk.h文件中定义了以下8个打印等级

```
#define KERN_EMERG      "<0>"   /* system is unusable                   */
#define KERN_ALERT      "<1>"   /* action must be taken immediately     */
#define KERN_CRIT       "<2>"   /* critical conditions                  */
#define KERN_ERR        "<3>"   /* error conditions                     */
#define KERN_WARNING    "<4>"   /* warning conditions                   */
#define KERN_NOTICE     "<5>"   /* normal but significant condition     */
#define KERN_INFO       "<6>"   /* informational                        */
#define KERN_DEBUG      "<7>"   /* debug-level messages                 */
```

内核出错信息oops

oops消息是内核提供的一种错误处理机制：

- oops消息包含系统错误的细节，如 CPU 寄存器的内容等
- 内核只能发布oops，这个过程包括向终端上输出错误消息，输出寄存器保存的信息，并输出可供跟踪的回溯线索。通常，发送完OOPS之后，内核会处于一种不稳定的状态
- oops的产生有很多可能原因，其中包括内存访问越界或非法的指令等

发生错误时可以直接分析oops信息，也可以借助工具分析

- ✓ ksymoops
- ✓ kallsyms

内核出错信息

系统发生严重错误时，将调用panic函数，kernel/panic.c文件实现了panic过程

```
void panic(const char *fmt, ...)
{
        ...
        if (!panic_blink)
                panic_blink = no_blink;
        if (panic_timeout > 0) {
                printk(KERN_EMERG "Rebooting in %d seconds..", panic_timeout);
                for (i = 0; i < panic_timeout * 1000; i += PANIC_TIMER_STEP) {
                        touch_nmi_watchdog();
                        if (i >= i_next) {
                                i += panic_blink(state ^= 1);
                                i_next = i + 3600 / PANIC_BLINK_SPD;
                        }
                        mdelay(PANIC_TIMER_STEP);
                }
        }
        ...
}
```

内核源码调试

使用qemu+gdb调试linux内核代码：

- 编译内核
  - ✓ make menuconfig
- qemu调试命令
  - ✓ qemu-system-x86_64 -m2048 -kernel ./arch/x86/boot/bzImage -s -S -nographic -serial mon:stdio -append "console=ttyS0 “

✓ # -m 指定内存数量
✓ # -kernel 指定 bzImage 的镜像路径
✓ # -s 等价于 -gdb tcp::1234 表示监听 1234 端口，用于 gdb 连接
✓ # -S 表示加载后立即暂停，等待调试指令。不设置这个选项内核会直接执行
✓ # -nographic 以及后续的指令用于将输出重新向到当前的终端中，这样就能方便的用滚屏查看内核的输出日志了。

内核源码调试

qemu运行后，在另一个终端执行gdb命令

vmlinux 是编译内核时生成的调试文件，在内核源码的根目录中。
gdb vmlinux
进入 gdb 的交互模式后，首先执行
show arch
当前架构一般是: i386:x86-64

连接 qemu 进行调试：
target remote :1234
设置断点
如果上面 qemu 是使用 qemu-kvm 执行的内核的话，就需要使用 hbreak 来设置断点，否则断点无法生效。
但是我们使用的是 qemu-system-x86_64，所以可以直接使用 b 命令设置断点。
b start_kernel
执行内核
c

内核源码调试

gdb常用命令

run	重新开始运行文件
start	单步执行，运行程序，停在第一执行语句
list	查看原代码，简写l
set	设置变量的值
next	单步调试（逐过程，函数直接执行）,简写n
step	单步调试（逐语句：跳入自定义函数内部执行）,简写s
backtrace	查看函数的调用的栈帧和层级关系,简写bt
frame	切换函数的栈帧,简写f
info	查看函数内部局部变量的数值,简写i
finish	结束当前函数，返回到函数调用点
continue	继续运行,简写c
print	打印值及地址,简写p
quit	退出gdb,简写q

问题及思考

1、Linux是通过文件来管理设备的，请查看/dev、/proc/devices /sys/dev文件夹，观察文件夹信息，并分析这三个文件夹在Linux设备管理中的作用？

2、Linux下I/O操作的类型有哪几种？有何区别？

3、Linux下用户程序是如何访问设备的？

4、Linux下如何创建一个字符设备驱动？

5、LINUX下监控键盘输入可以有哪几种方式？

推荐资料

- Ldd3, Linux设备驱动开发详解第三版(Linux device driver development 3)

实验要求

实验一、在linux下编写一个简单的字符设备驱动，并实现将驱动以.ko文件形式加载到内核。同时编写一个用户程序，实现对该设备的读操作，获取设备输出。

实验二、实现一个键盘监控程序，可以获取用户键盘的输入。

问题？

谢　谢！