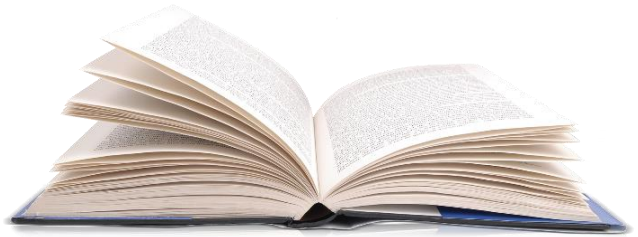

从零学Linux驱动
一口Linux

想入门和进阶Linux驱动，
请加关注一口君的公众号：一口Linux

01

如何学习Linux驱动



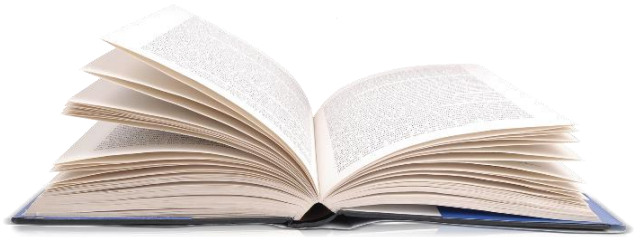
公众号：一口Linux

学习Linux驱动基础知识

- 1. C语言
- 2. Linux使用
- 3. 硬件：
  - 电路图（信号线、数据线、中断）
  - 常见总线I2C、SPI、UART
- 4. Linux系统编程-文件IO

02

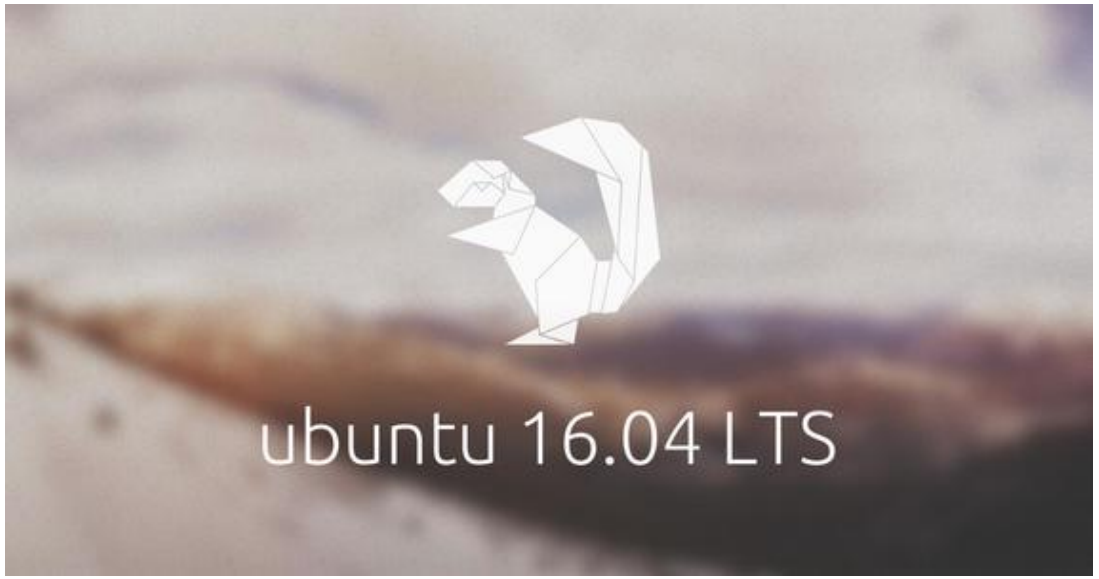
Linux环境安装



公众号：一口Linux

安装软件 1

- **ubuntu**
  - Ubuntu是一个以桌面应用为主的Linux操作系统，其名称来自非洲南部祖鲁语或豪萨语的“ubuntu"一词，意思是“人性”“我的存在是因为大家的存在"，是非洲传统的一种价值观。



公众号：一口Linux

安装软件 2

- **vmware**
  - VMware Workstation（中文名“威睿工作站”）是一款功能强大的桌面虚拟计算机软件，提供用户可在单一的桌面上同时运行不同的操作系统，和进行开发、测试 、部署新的应用程序的最佳解决方案。

公众号：一口Linux

安装软件 3

- **Source Insight**

公众号：一口Linux

内核源码

- Linux kernel
  - linux 3.14内核

- 所有资料下载地址

本文档请关注公众号：一口Linux
后台回复：ubuntu

安装步骤

CSDN：一口Linux

《linux环境搭建-ubuntu16.04安装》

《Source Insight给Linux内核创建工程》

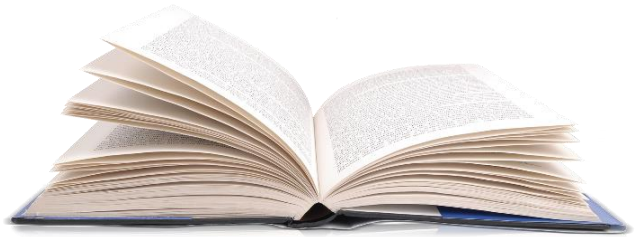
下载地址

- **#ubuntu下载地址**
- *http://mirrors.aliyun.com/ubuntu-releases/16.04/ubuntu-16.04.7-desktop-amd64.iso*

- **#vmware下载地址**
- *https://www.vmware.com/go/getworkstation-win*

03

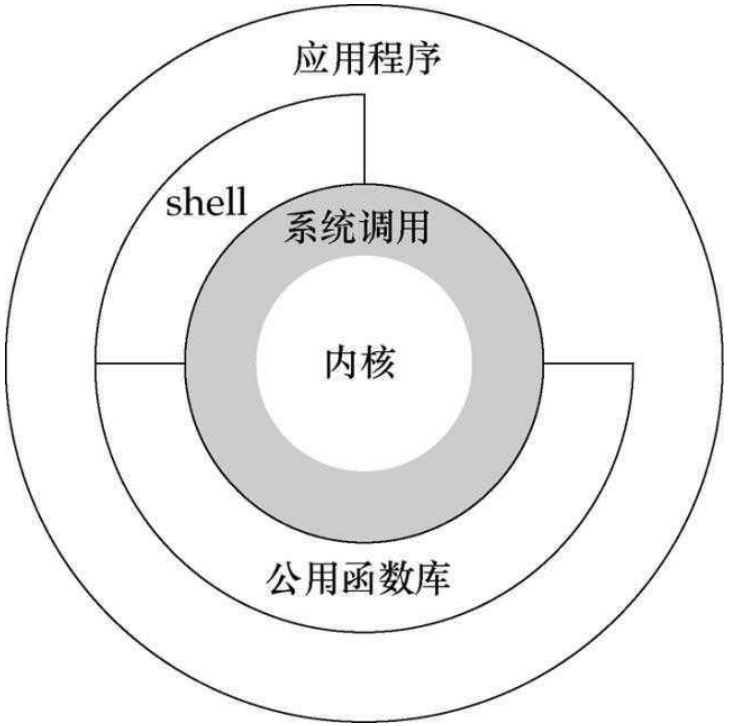
Linux内核、目录



公众号：一口Linux

Linux内核

- 从技术上说 linux 是一个内核
- “内核”指的是一个提供硬件抽象层、磁盘及文件系统控制、多任务等功能的系统软件。一个内核不是一套完整的操作系统。
- 通常我们使用的 linux 系统是一个集 linux 内核、工具集、各种库、桌面管理器、应用程序等一体的一个发布包 (发行版)



公众号：一口Linux

主流的 Linux 发行版

- Debian GNU/Linux
- Red Hat Linux
- Fedora Core
- Ubuntu Linux
- SUSE Linux
- Gentoo Linux
- Asianux
- Slackware Linux
- Turbo Linux
- CentOS

公众号：一口Linux

Linux内核版本

- 目前linux系统采用 A.B.C.D 的版本号管理方式
  - A 表示linux的主版本号
  - B 表示linux的次版本号，B 为偶数表示稳定版本，奇数表示开发中的版本
  - C 表示linux的发行版本号
  - D 表示更新版本号
- 主版本（X.Y）
  - **1.0　2.0　2.2　2.4　2.6　3.x**
- 稳定版本发布一般 1,2月
  - **2.0.40 2.2.12　2.4.18　2.6.15　3.13**
- 稳定更新版本
  - **2.6.23.1~ 2.6.23.7**
- Kernel.org上目前维护的主版本
  - 2.2
  - 2.4
  - 2.6
  - 3.x

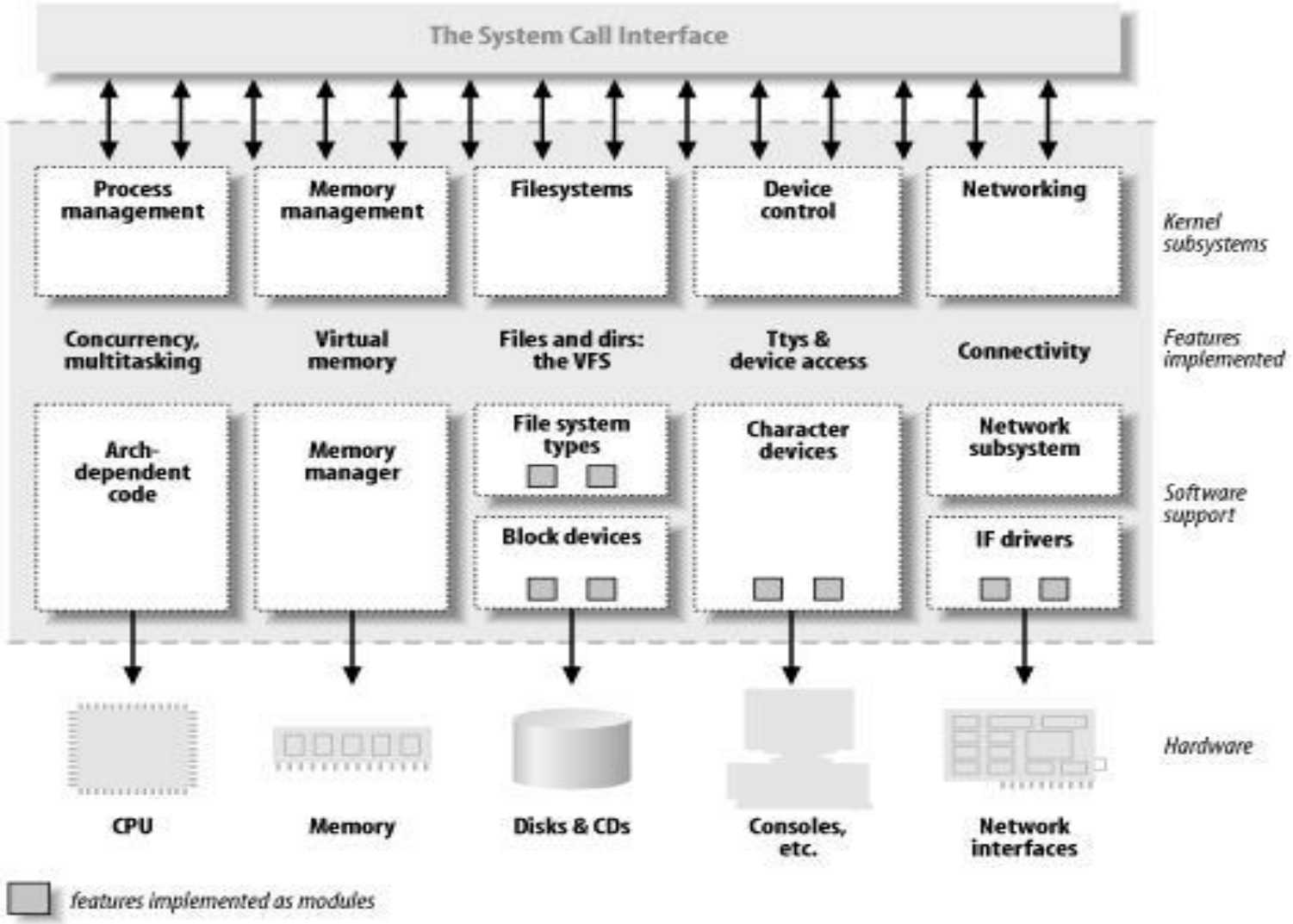
公众号：一口Linux

Linux内核子系统

- 进程管理
- 内存管理
- 文件系统
- 网络协议
- 设备管理

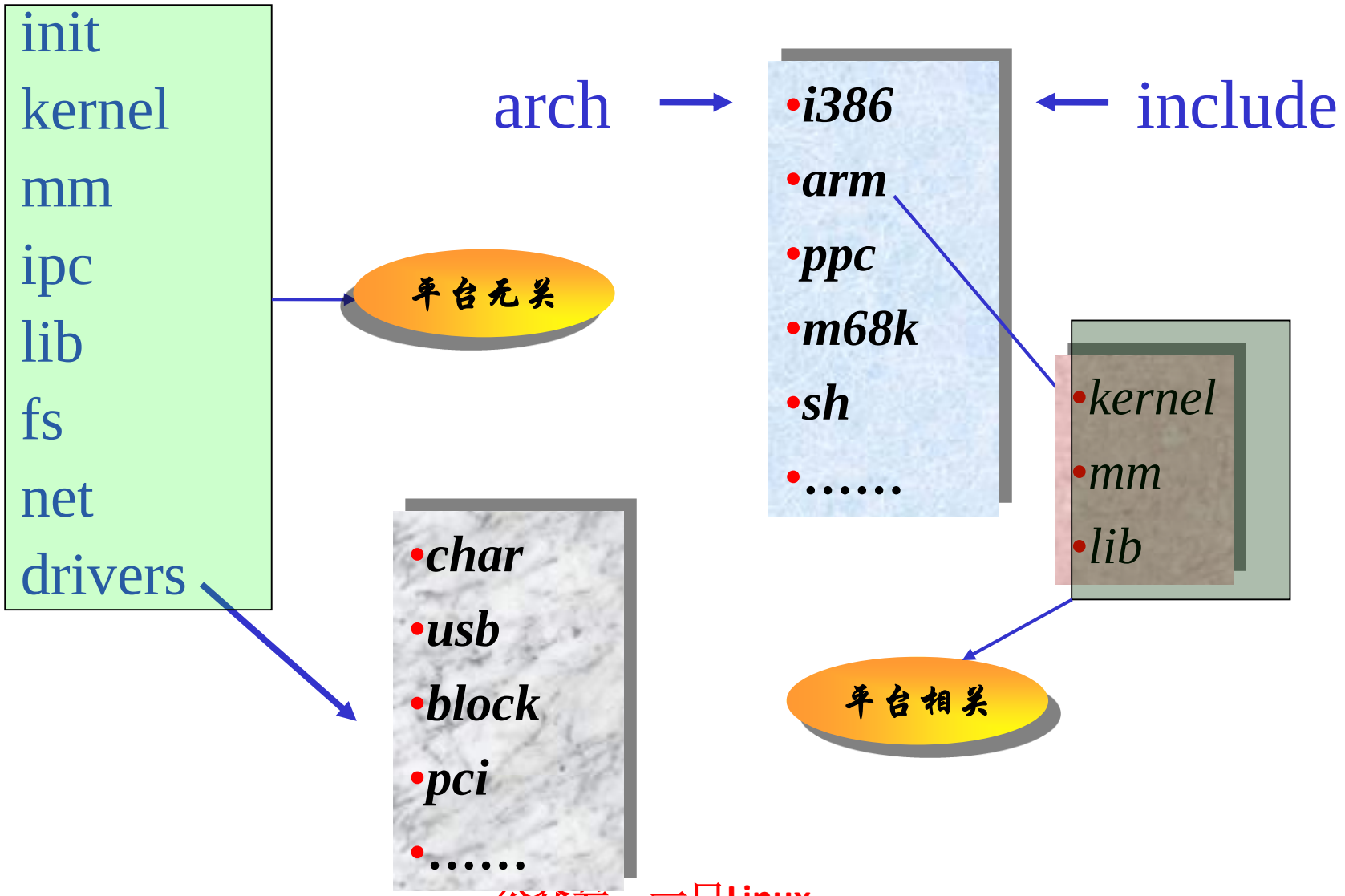
公众号：一口Linux

Linux内核模块结构图



公众号：一口Linux

Linux内核代码结构 1



公众号：一口Linux

Linux系统源代码目录结构（2）

kernel/	内核管理的核心代码（系统相关代码在arch/*/kernel中）
arch/	体系结构相关代码，每个目录代码一个体系结构
include/: include/asm-<*arch*> include/linux	Linux 头文件 体系结构相关头文件 Linux kernel core 头文件
lib/	内核的库代码 (zlib, crc32...)，和处理器体系结构相关的代码在arch/*/lib下
drivers/	系统中驱动程序代码
fs/	文件系统
ipc/	IPC（进程通讯代码）
mm/	内存管理代码，和处理器体系机构相关的代码在arch/*/mm下
net/	网络协议代码

公众号：一口Linux

Linux系统源代码目录结构（3）

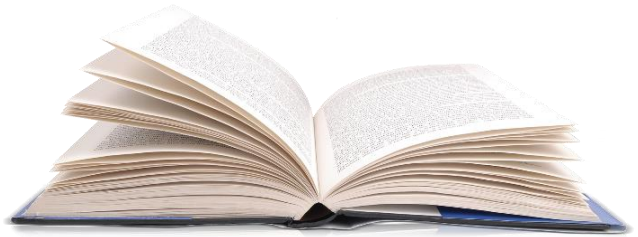
scripts/	编译配置脚本文件
sound/	声卡驱动程序
Documentation/	内核文档目录
README	概要和编译介绍
Makefile	顶层Makefile文件 (sets arch and version)
MAINTAINERS	每个部分的维护者
COPYING	Linux 版权许可 (GNU GPL)
CREDITS	Linux 主要贡献者
REPORTINGBUGS	Bug 报告说明

内核下载

- 下载内核
  - http://kernel.org
  - wget http://kernel.org/pub/linux/kernel/...
  - wget http://kernel.org/.../.../linux-*.bz2.sign
  - gpg –verify linux-2.xx.tar.bz2.sign
  - tar xvf linux-2.xx.tar.bz2

公众号：一口Linux

04 模块化编程



公众号：一口Linux

实例

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("PD");

static int hello_init(void)
{
    printk("hello_init \n");
    return 0;
}
static void hello_exit(void)
{
    printk("hello_exit \n");
    return;
}

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

必须包含的头文件

功能代码

printk:内核的printf函数，用于输出信息

模块的入口：完成模块的加载

模块的出口：完成模块的卸载

公众号：一口Linux

Makefile

```
ifneq ($(KERNELRELEASE),)
obj-m:=hello.o
else
KDIR :=/lib/modules/$(shell uname -r)/build
PWD  :=$(shell pwd)
all:
	make -C $(KDIR) M=$(PWD) modules
clean:
	rm -f *.ko *.o *.mod.o *.symvers *.cmd  *.mod.c *.order
endif
```

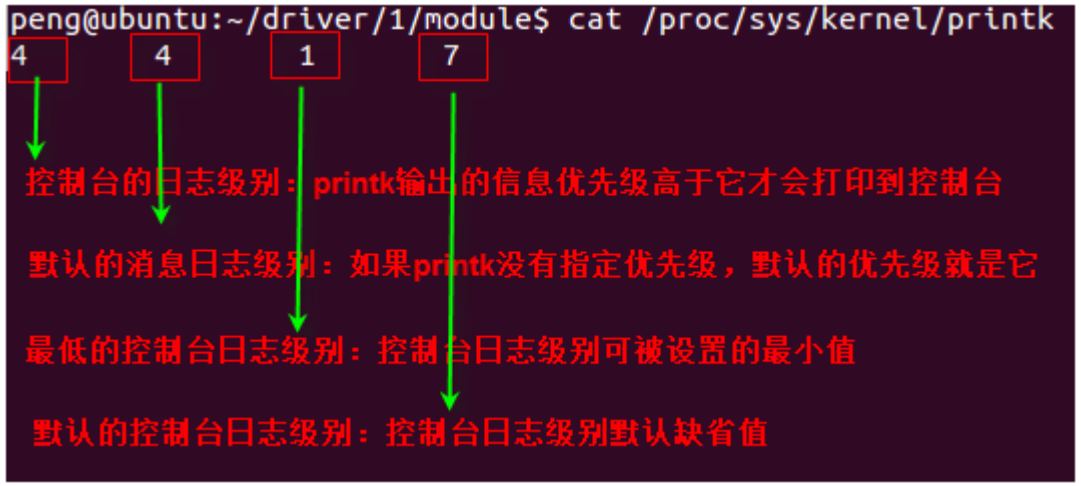
公众号：一口Linux

编译加载查看命令

- make
  - 编译
- lsmod
  - 显示模块
- insmod/rmmod
  - 安装/卸载模块
- dmesg
  - 打印log信息

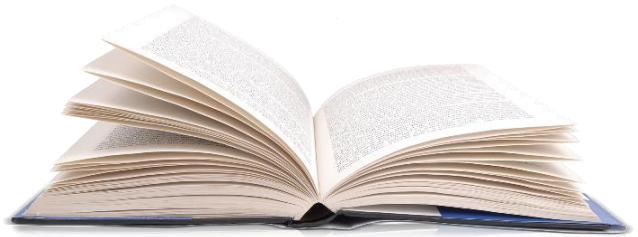
printk

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

#define KERN_DEFAULT    KERN_SOH "d"/* the default kernel loglevel */
```



公众号：一口Linux

05 Makefile



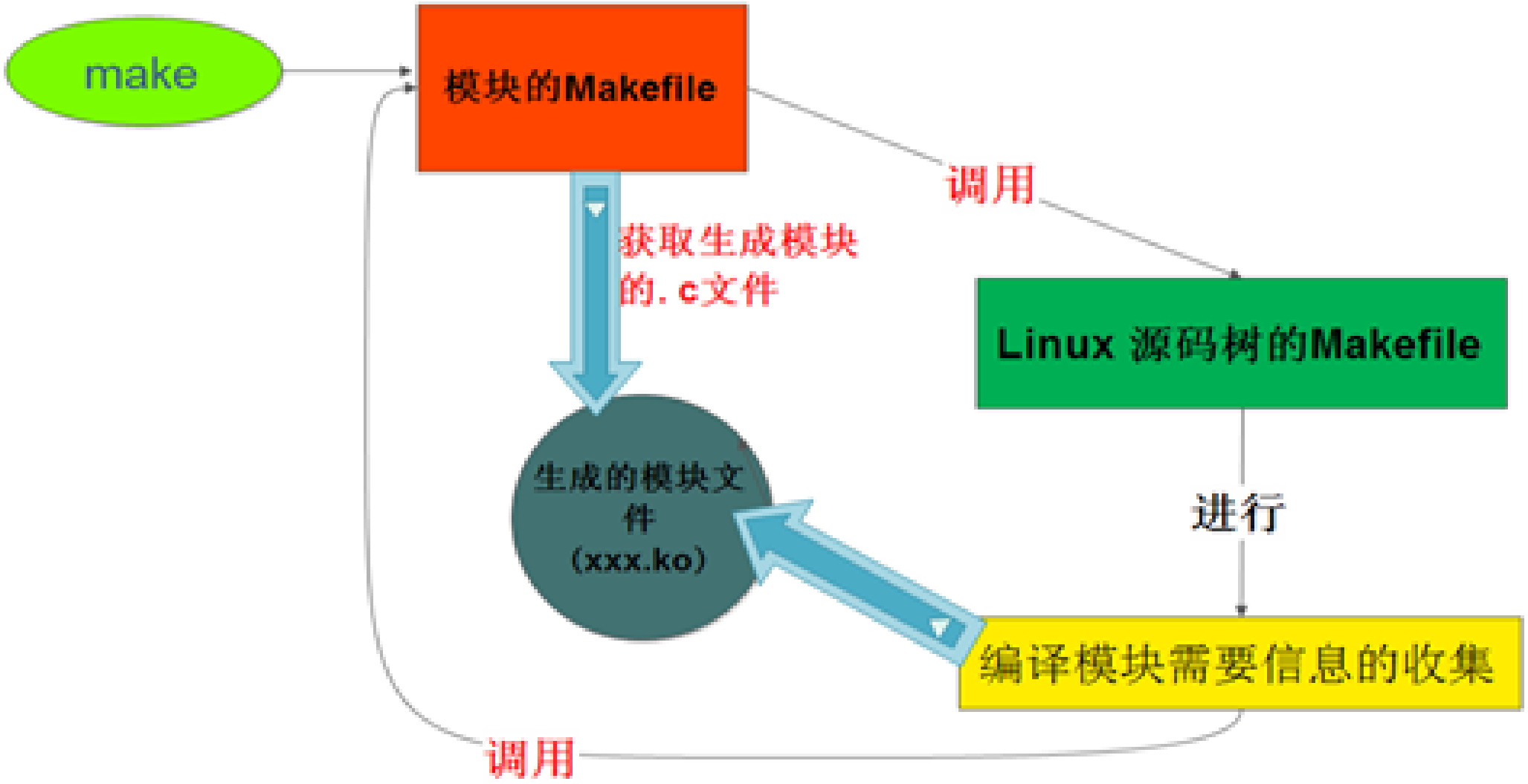
公众号：一口Linux

Makefile

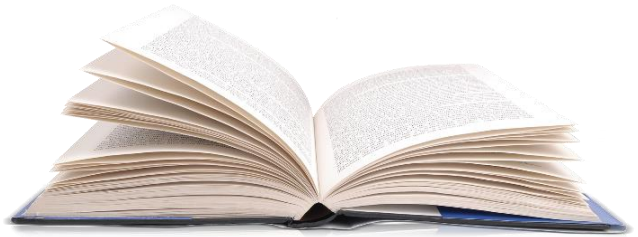
```
ifneq ($(KERNELRELEASE),)
$(info "2nd")
obj-m:=hello.o
else
KDIR :=/lib/modules/$(shell uname -r)/build
PWD  :=$(shell pwd)
all:
	$(info "1st")
	make -C $(KDIR) M=$(PWD) modules
clean:
	rm -f *.ko *.o *.mod.o *.symvers *.cmd  *.mod.c *.order
endif
```

公众号：一口Linux

Makefile


make
模块的Makefile
调用
获取生成模块的.c文件
Linux 源码树的Makefile
生成的模块文件（xxx.ko）
进行
编译模块需要信息的收集
调用

06 符号导出



公众号：一口Linux

什么是符号？

- 这里的符号主要指的是全局变量和函数

- Linux内核采用的是以模块化形式管理内核代码。内核中的每个模块相互之间是相互独立的，也就是说A模块的全局变量和函数，B模块是无法访问的。

符号表

- Ubuntu中
  - Linux内核的全局符号表在+
  - /usr/src/linux-headers-xxxxx-generic-pae/Module.symvers

- 某个单独编译的内核
  - 根目录下

实例

模块A	模块B

static int global_var = 100;
 	
static void show(void)
{
printk("show(): global_var =%d \n",global_var);
}
	extern int global_var;
	extern void show(void);
	
	static int hello_init(void)
	{
	show();
	return 0;
	}
	module_init(hello_init);

步骤

- 1.编译模块A,然后加载模块A,在模块A编译好后，在它的当前目录会看到一个Module.symvers文件，这里存放的就是我们模块A导出的符号。

- 2.将模块A编译生成的Module.symvers文件拷贝到模块B目录下，然后编译B模块
- 3. 先加载A模块然后加载模块B。

- 4.通过dmesg查看模块打印的信息。

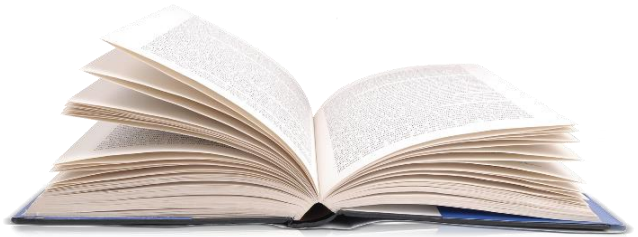
注意

- 1. 加载的时候，必须先加载a模块，再加载b模块
- 2. 卸载：必须先卸载b模块，再卸载a模块

公众号：一口Linux

07

模块传参



公众号：一口Linux

代码

- 关注公众号 ：一口Linux
- 后台回复 ：ubuntu

给模块传递参数

include\linux\ Moduleparam.h

原型： module_param(name,type,perm)

参数：

@name用来接收参数的变量名

@type参数的数据类型

@perm指定参数访问权限

类型

Linux 内核支持的模块参数类型	
bool	布尔值(true/false)，关联的变量应该是int类型
invbool	bool的反值，例如赋值为true，实际值为false
charp	字符指针类型，内核为用户提供的字符串分配内存，并设置此指针保存其首地址
int	整型
long	长整型
short	短整型
uint	无符号整型
ulong	无符号长整型
ushort	无符号短整型

公众号：一口Linux

其他原型

```
module_param_string(name,string,len,perm);

module_param_arry(name,type,num_point,perm);

MODULE_PARM_DESC(dbg,
"Boolean to enable debugging (0/1 == off/on)");
```

公众号：一口Linux

```
 5 static char *whom = "hello \n";
 6 static int var = 1;
 7
 8 static int hello_init(void)
 9 {
10     printk("hello_init %s \n",whom);
11     return 0;
12 }
13 static void hello_exit(void)
14 {
15     printk("hello_exit  %d\n",var);
16     return;
17 }
18 MODULE_LICENSE("GPL");
19 module_param(whom,charp,0644);
20 module_param_named(var_out,var,int,0644);
21
22 module_init(hello_init);
23 module_exit(hello_exit);
```

测试

- insmod hello.ko var=1 whom= “yikoupeng”

sysfs

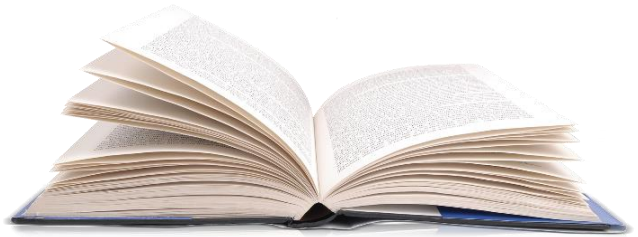
Sysfs: 内核给一些重要的资源创建的目录或者文件

每个模块会在/sys/module下创建一个同名的文件夹

```
root@ubuntu:/sys/module/hello/parameters# ls -l
total 0
-rw-r--r-- 1 root root 4096 Mar 17 05:35 var_out
-rw-r--r-- 1 root root 4096 Mar 17 05:35 whom
```

公众号：一口Linux

08 字符设备号



公众号：一口Linux

modinfo

```
18 MODULE_LICENSE("GPL");
19 module_param(whom,charp,0644);
20 module_param_named(var_out,var,int,0644);
21
22 MODULE_PARM_DESC(var,
23         "Boolean to enable debugging (0/1 == off/on)");
24 MODULE_AUTHOR("daniel peng");
25 MODULE_DESCRIPTION("yikou test");
26 MODULE_LICENSE("GPL");
27 MODULE_ALIAS("yikoupeng");
```

公众号：一口Linux

设备分类

- 字符设备
- 块设备
- 网络设备

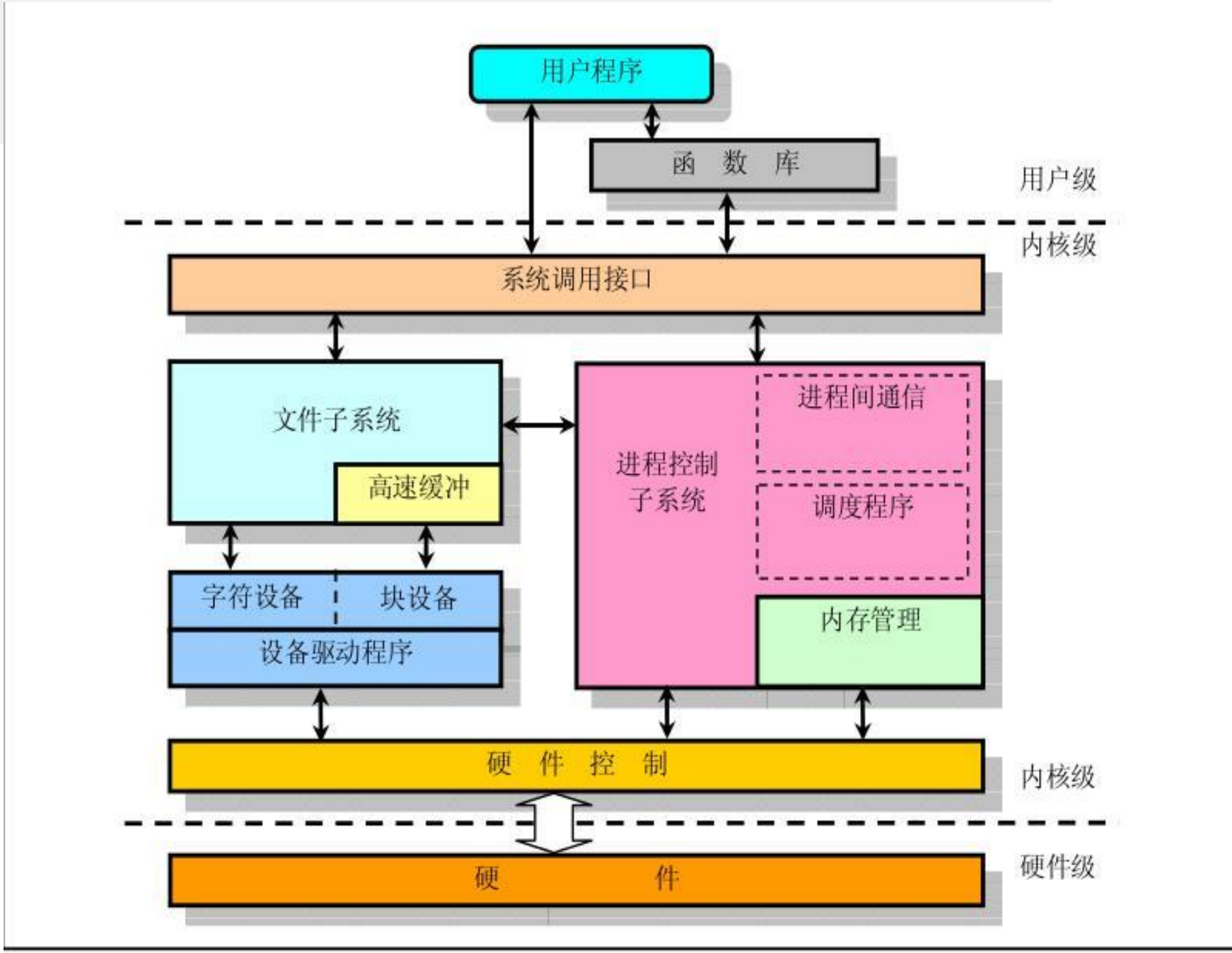
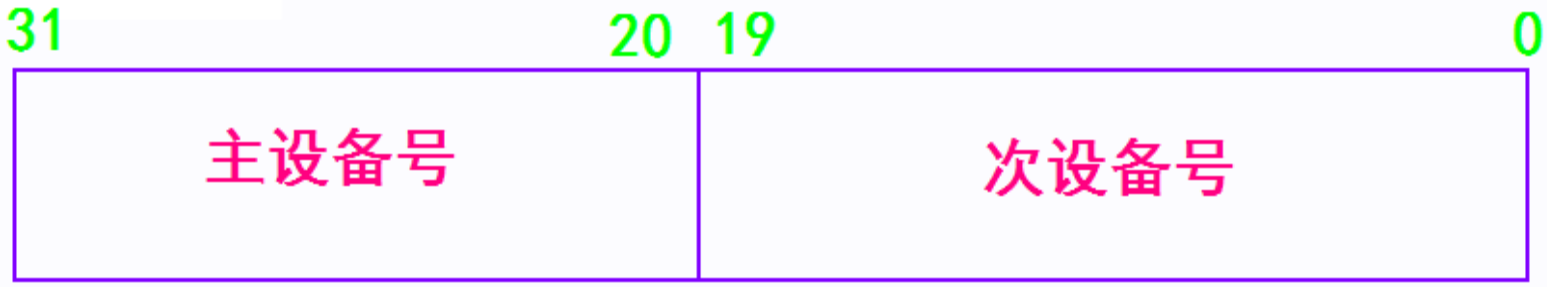


图 2-4 内核结构框图

设备号

- 无符号整型值
  - 高12位：主设备号
  - 低20位：次设备号



-

- 查看设备号
  - cat /proc/devices

公众号：一口Linux

构造设备号

- typedef __kernel_dev_t dev_t;
- typedef __u32 __kernel_dev_t;
- typedef uint32_t __u32;

- #define MINORBITS 20
- #define MINORMASK ((1U << MINORBITS) - 1)

- #define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))
- #define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))
- #define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))

公众号：一口Linux

注册/注销设备号

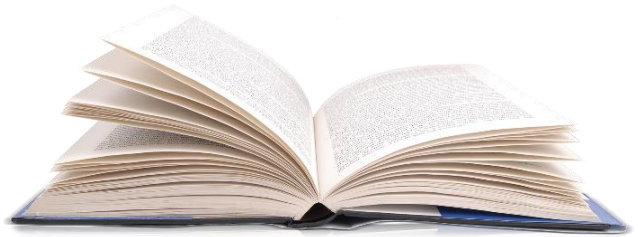
```
/**
 * register_chrdev_region() -  register a range of device numbers
 * @from: the first in the desired range of device numbers; must include
 *        the major number.
 * @count: the number of consecutive device numbers required
 * @name: the name of the device or driver.
 *
 * Return value is zero on success, a negative error code on failure.
 */
int register_chrdev_region(dev_t from, unsigned count, const char *name)
```

```
/**
 * unregister_chrdev_region() -  return a range of device numbers
 * @from: the first in the range of numbers to unregister
 * @count: the number of device numbers to unregister
 *
 * This function will unregister a range of @count device numbers,
 * starting with @from.  The caller should normally be the one who
 * allocated those numbers in the first place...
 */
void unregister_chrdev_region(dev_t from, unsigned count)
```

公众号：一口Linux

09

字符设备架构



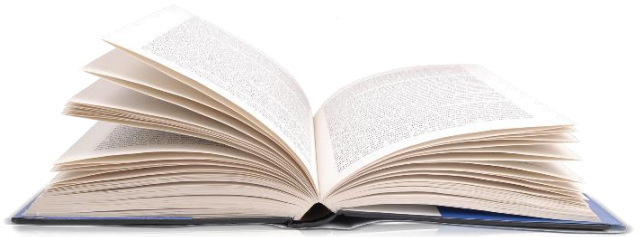
公众号：一口Linux

架构图

公众号：一口Linux

10

字符设备注册



公众号：一口Linux

cdev

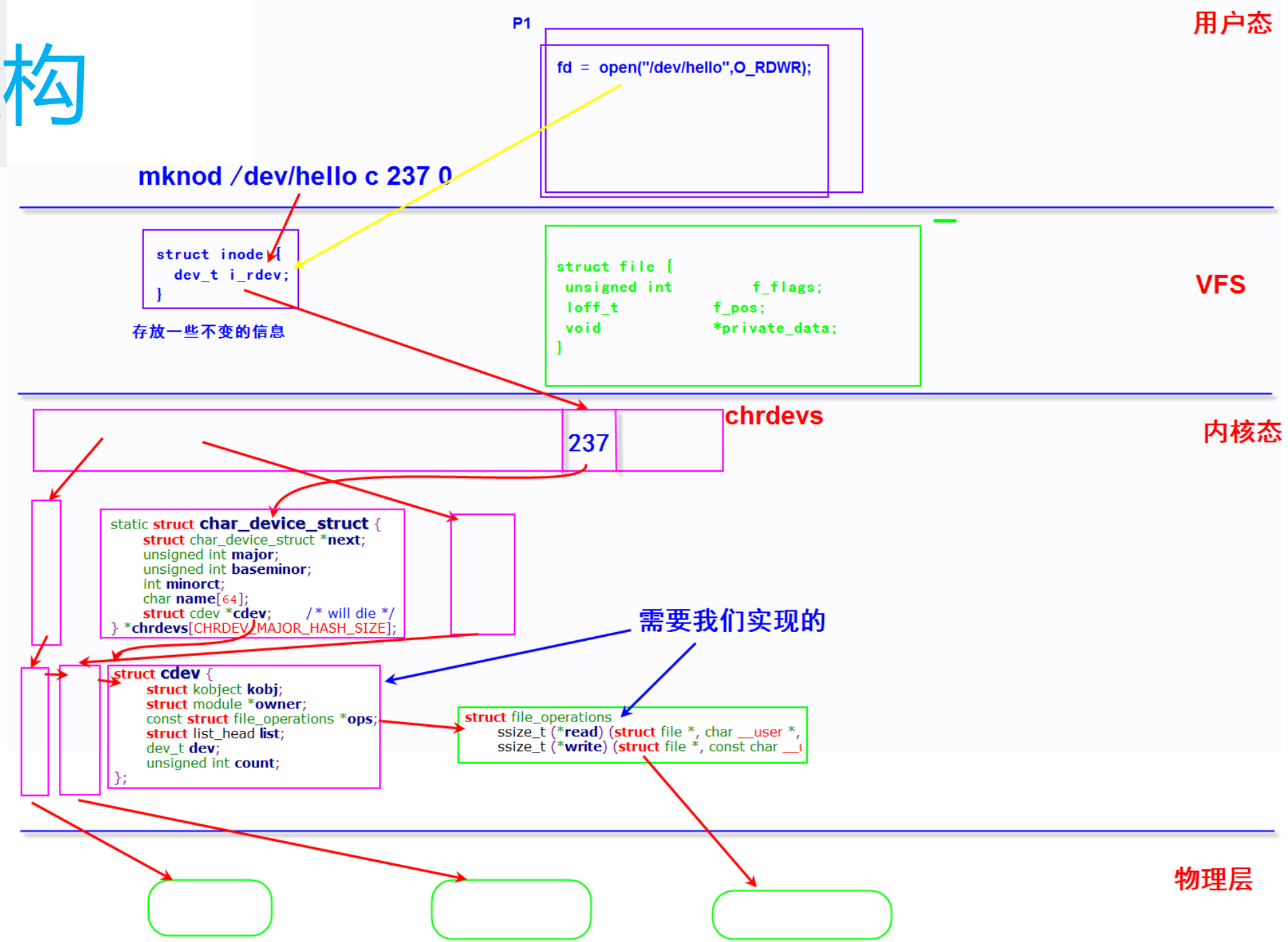
- void cdev_init(struct cdev *, const struct file_operations *);
- int cdev_add(struct cdev *, dev_t, unsigned);
- void cdev_del(struct cdev *);

公众号：一口Linux

struct file_operations

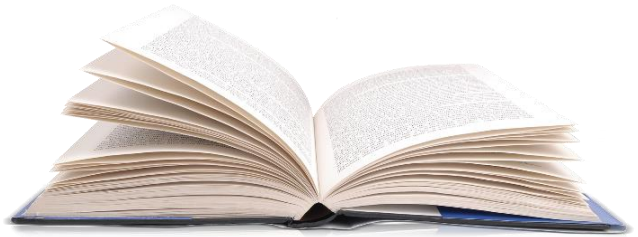
```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*iterate) (struct file *, struct dir_context *);
        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 lor
        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, u
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, un
        int (*setlease)(struct file *, long, struct file_lock **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,
                          loff_t len);
        int (*show_fdinfo)(struct seq_file *m, struct file *f);
} ? end file_operations ? ;
```

字符设备架构


用户态
P1
fd = open("/dev/hello",O_RDWR);
mknod /dev/hello c 237 0
struct inode {
dev_t i_rdev;
}
存放一些不变的信息
struct file {
unsigned int f_flags;
loff_t f_pos;
void *private_data;
}
VFS
chrdevs
237
内核态
static struct char_device_struct {
struct char_device_struct *next;
unsigned int major;
unsigned int baseminor;
int minorct;
char name[64];
struct cdev *cdev; /* will die */
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
需要我们实现的
struct cdev {
struct kobject kobj;
struct module *owner;
const struct file_operations *ops;
struct list_head list;
dev_t dev;
unsigned int count;
};
struct file_operations
ssize_t (*read) (struct file *, char __user *,
ssize_t (*write) (struct file *, const char
物理层

11

字符设备注册-更简单的方式



公众号：一口Linux

register_chrdev

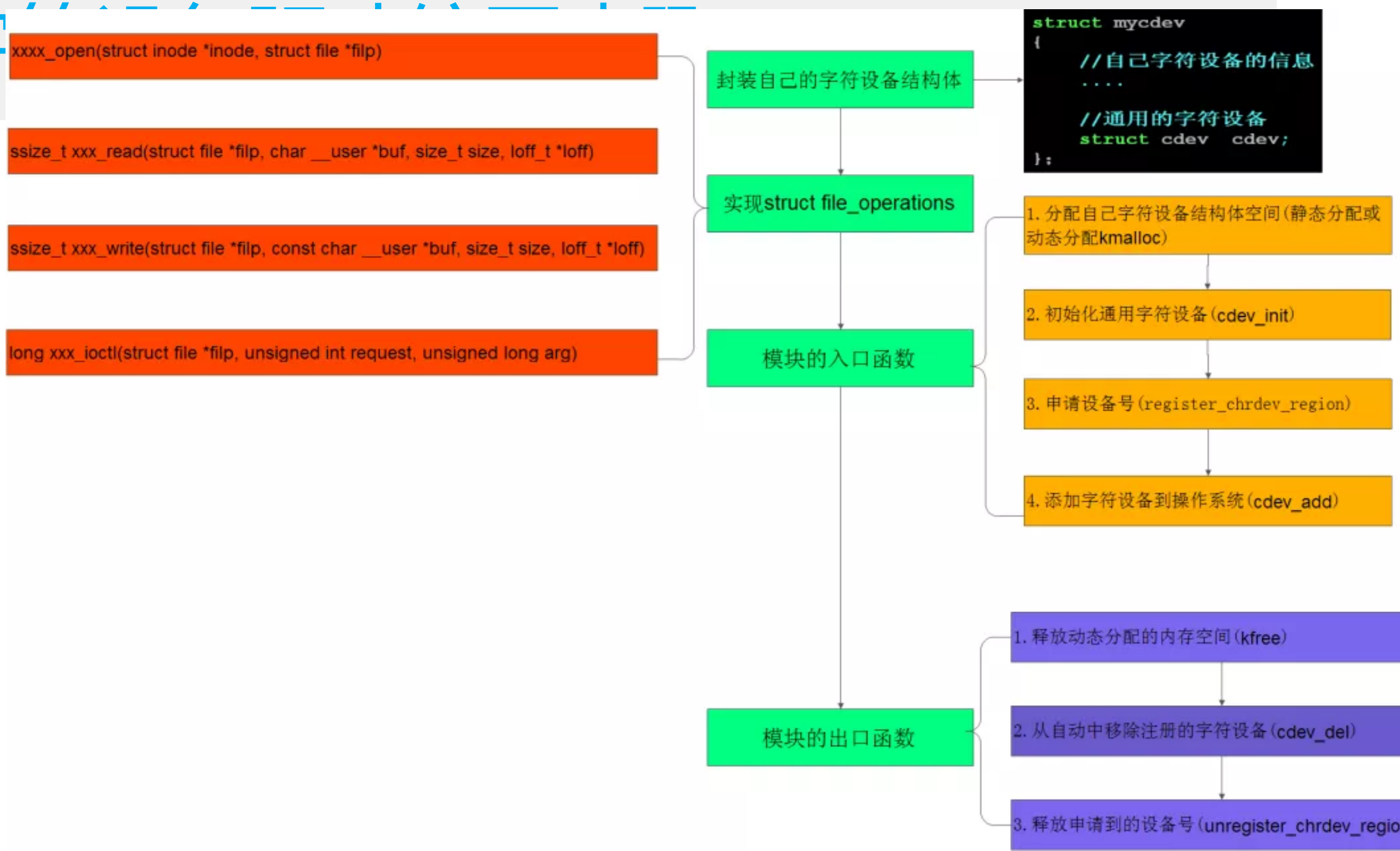
```
static inline int register_chrdev(unsigned int major, const char *name,
                  const struct file_operations *fops)
{
    return __register_chrdev(major, 0, 256, name, fops);
}
/**
 * __register_chrdev() - create and register a cdev occupying a range of minors
 * @major: major device number or 0 for dynamic allocation
 * @baseminor: first of the requested range of minor numbers
 * @count: the number of minor numbers required
 * @name: name of this range of devices
 * @fops: file operations associated with this devices
 *
 * If @major == 0 this functions will dynamically allocate a major and return
 * its number.
 *
 * If @major > 0 this function will attempt to reserve a device with the given
 * major number and will return zero on success.
 *
 * Returns a -ve errno on failure.
 *
 * The name of this device has nothing to do with the name of the device in
 * /dev. It only helps to keep track of the different owners of devices. If
 * your module name has only one type of devices it's ok to use e.g. the name
 * of the module here.
 */
```

unregister_chrdev

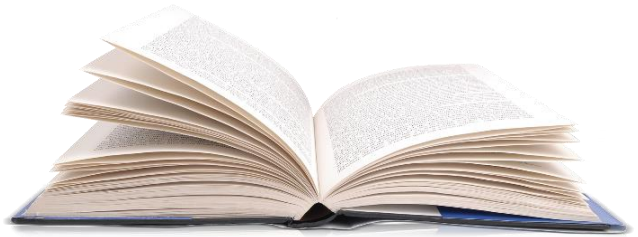
```
static inline void unregister_chrdev(unsigned int major, const char *name)
{
        __unregister_chrdev(major, 0, 256, name);
}
```

公众号：一口Linux

字


xxxx_open(struct inode *inode, struct file *filp)
ssize_t xxx_read(struct file *filp, char __user *buf, size_t size, loff_t *loff)
ssize_t xxx_write(struct file *filp, const char __user *buf, size_t size, loff_t *loff)
long xxx_ioctl(struct file *filp, unsigned int request, unsigned long arg)
封装自己的字符设备结构体
struct mycdev
{
//自己字符设备的信息
....
//通用的字符设备
struct cdev cdev;
};
实现struct file_operations
模块的入口函数
1. 分配自己字符设备结构体空间(静态分配或动态分配kmalloc)
2. 初始化通用字符设备(cdev_init)
3. 申请设备号(register_chrdev_region)
4. 添加字符设备到操作系统(cdev_add)
模块的出口函数
1. 释放动态分配的内存空间(kfree)
2. 从自动中移除注册的字符设备(cdev_del)
3. 释放申请到的设备号(unregister_chrdev_region)

12 自动创建设备节点



公众号：一口Linux

udev

- udev 是一个工作在用户空间的工具，它能根据系统中硬件设备的状态动态的更新设备文件，包括设备文件的创建，删除，权限等。这些文件通常都定义在/dev 目录下，但也可以在配置文件中指定。

- udev 运行在用户模式，而非内核中。

- 当插入新设备——加入驱动模块——在sysfs上注册新的数据后，udev会创新新的设备节点。

公众号：一口Linux

创建一个类

```
/* This is a #define to keep the compiler from merging different
 * instances of the __key variable */
#define class_create(owner, name)       \
({                                      \
    static struct lock_class_key __key;     \
    __class_create(owner, name, &__key);   \
})
```

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

https://blog.csdn.net/daocaokafei

参数：

@owner THIS_MODULE

@name 类名字

返回值

可以定义一个struct class的指针变量cls接受返回值，然后通过IS_ERR(cls)判断是否失败，如果成功这个宏返回0，失败返回非9值（可以通过PTR_ERR(cls)来获得失败返回的错误码）

公众号：一口Linux

注销类

```
/**
 * class_destroy -  destroys a struct class structure
 * @cls: pointer to the struct class that is to be destroyed
 *
 * Note, the pointer to be destroyed must have been created with a call
 * to class_create().
 */
void class_destroy(struct class *cls)
{
	if ((cls == NULL) || (IS_ERR(cls)))
		return;

	class_unregister(cls);
}
```

公众号：一口Linux

导出设备信息到用户空间

```
/**
 * device_create - creates a device and registers it with sysfs
 * @class: pointer to the struct class that this device should be registered to
 * @parent: pointer to the parent struct device of this new device, if any
 * @devt: the dev_t for the char device to be added
 * @drvdata: the data to be added to the device for callbacks
 * @fmt: string for the device's name
 *
 * This function can be used by char device classes.  A struct device
 * will be created in sysfs, registered to the specified class.
 *
 * A "dev" file will be created, showing the dev_t for the device, if
 * the dev_t is not 0,0.
 * If a pointer to a parent struct device is passed in, the newly created
 * struct device will be a child of that device in sysfs.
 * The pointer to the struct device will be returned from the call.
 * Any further sysfs files that might be required can be created using this
 * pointer.
 *
 * Returns &struct device pointer on success, or ERR_PTR() on error.
 *
 * Note: the struct class passed to this function must have previously
 * been created with a call to class_create().
 */
struct device *device_create(struct class *class, struct device *parent,
                             dev_t devt, void *drvdata, const char *fmt, ...)
{
    va_list vargs;
    struct device *dev;

    va_start(vargs, fmt);
    dev = device_create_vargs(class, parent, devt, drvdata, fmt, vargs);
    va_end(vargs);
    return dev;
}
```

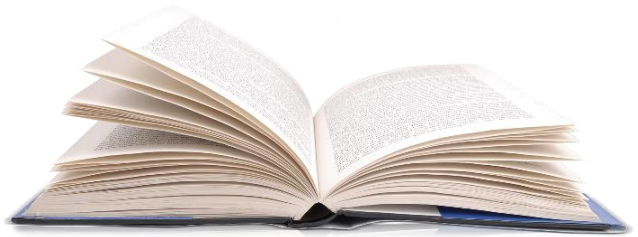
注销设备信息

```
/**
 * device_destroy - removes a device that was created with device_create()
 * @class: pointer to the struct class that this device was registered with
 * @devt: the dev_t of the device that was previously registered
 *
 * This call unregisters and cleans up a device that was created with a
 * call to device_create().
 */
void device_destroy(struct class *class, dev_t devt)
{
	struct device *dev;

	dev = class_find_device(class, NULL, &devt, __match_devt);
	if (dev) {
		put_device(dev);
		device_unregister(dev);
	}
}
```

公众号：一口Linux

13

字符设备read、write接口



公众号：一口Linux

系统调用read、write

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

参数

fd ：文件描述符

buf ： 读取的数据存放的缓冲区

Count: 缓冲区大小

返回值

实际读取字节的个数

出错： 负值

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

参数

fd ：文件描述符

buf ： 要**写入的数据**存放的缓冲区

Count: 实际要写入的数据的大小

返回值

实际**写入字节**的个数

出错： 负值

公众号：一口Linux

内核接口read

```
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)(struct file *filp, char __user *buf, size_t count, lofft *f_pos);

参数：

filp: 待操作的设备文件file结构体指针

buf: 待写入所读取数据的用户空间缓冲区指针

count:待读取数据字节数

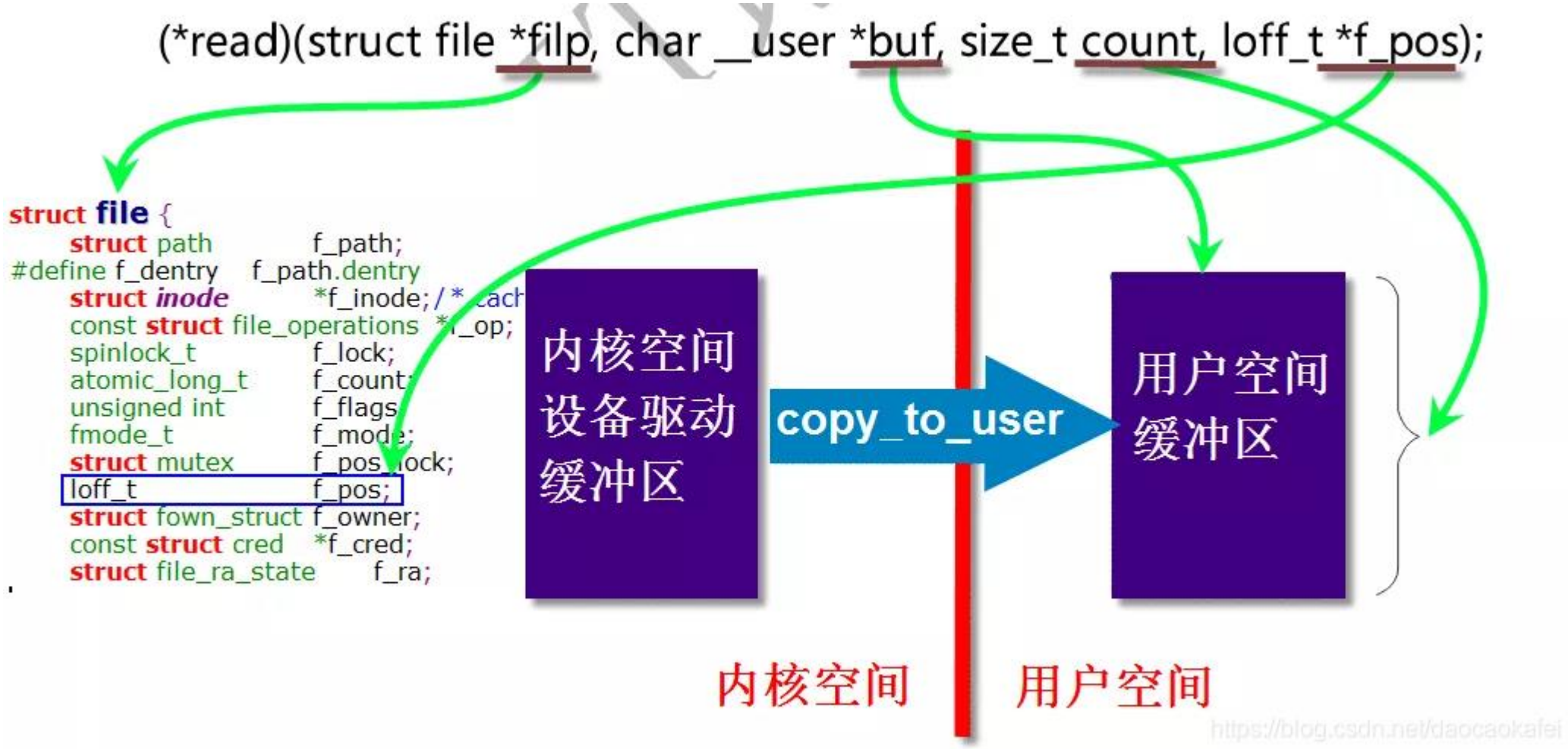
f_pos:待读取数据文件位置，读取完成后根据实际读取字节数重新定位

__user ：是一个空的宏，主要用来显示的告诉程序员它修饰的指针变量存放的是用户空间的地址。

返回值：

成功实际读取的字节数，失败返回负值

read



公众号：一口Linux

copy_to_user

```
: static inline unsigned long __must_check copy_to_user(void __user *to, const void *from, unsigned long n)
. {
```

公众号：一口Linux

内核接口write

```
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 (*write)(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos);

参数：

filp:待操作的设备文件file结构体指针

buf:待写入所读取数据的用户空间缓冲区指针

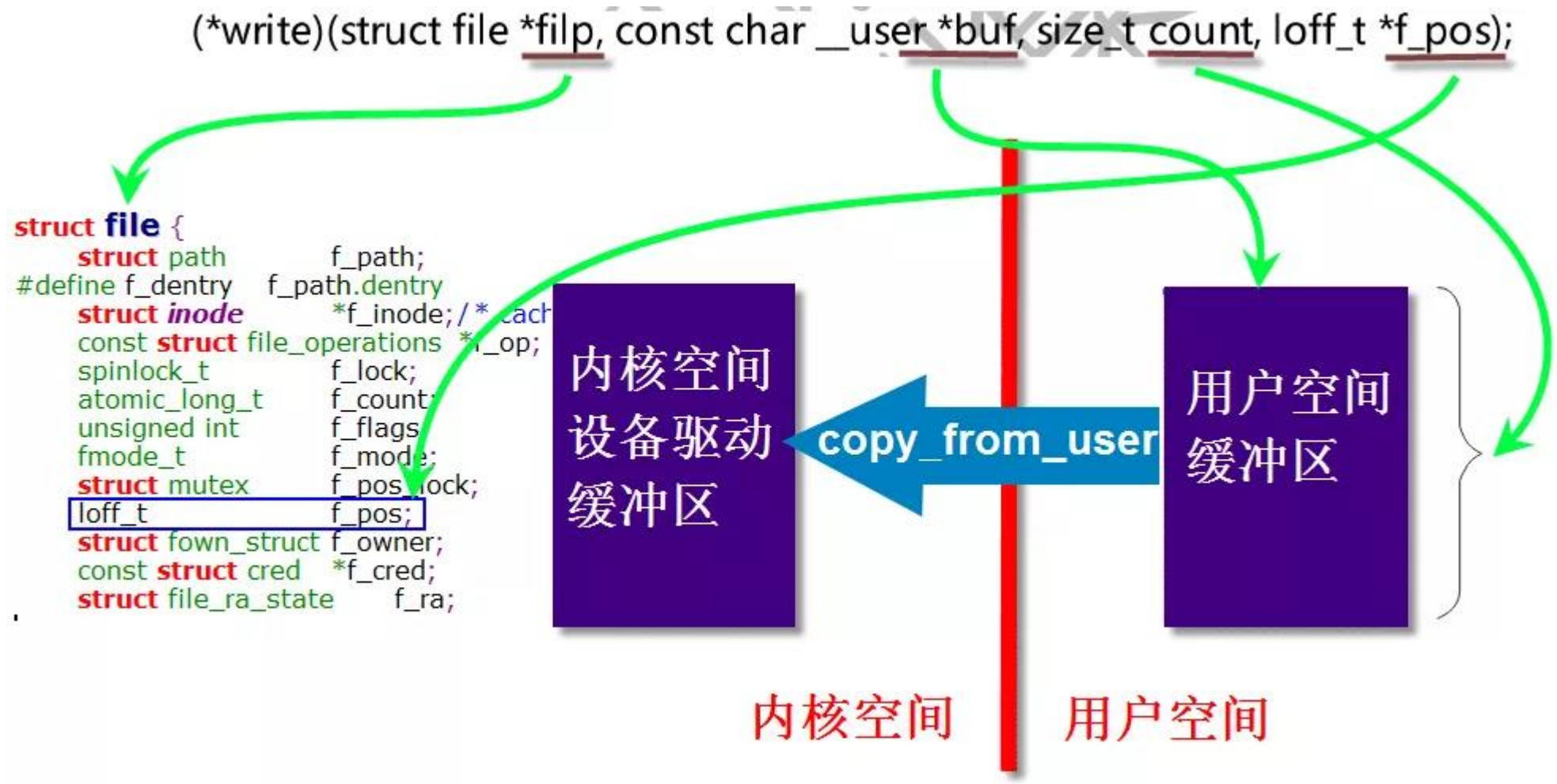
count:待读取数据字节数

f_pos:待读取数据文件位置，写入完成后根据实际写入字节数重新定位

返回：

成功实际写入的字节数，失败返回负值

write



公众号：一口Linux

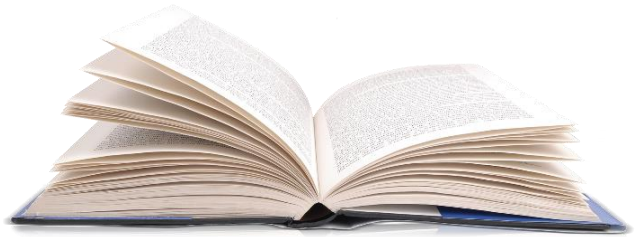
copy_from_user

```
static inline unsigned long __must_check copy_from_user(void *to, const void __user *from, unsigned long n)
```

公众号：一口Linux

14

字符设备ioctl接口



公众号：一口Linux

为什么引入ioctrl？

- 工程师要对设备进行读写数据之外，还希望可以对设备进行控制。

- 例如:
- 针对串口设备，驱动层除了需要提供对串口的读写之外，还需提供对串口波特率、奇偶校验位、终止位的设置，这些配置信息需要从应用层传递一些基本数据，仅仅是数据类型不同。

公众号：一口Linux

应用层系统调用

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);
```

参数：

@fd:打开设备文件的时候获得文件描述符

@ cmd:第二个参数:给驱动层传递的命令，需要注意的时候，驱动层的命令和应用层的命令一定要统一

@第三个参数:"..."在C语言中，很多时候都被理解成可变参数。

返回值

成功：0

失败：-1，同时设置errno

unlocked_ioctl

```
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
```

参数：

@file：vfs层为打开字符设备文件的进程创建的结构体，用于存放文件的动态信息

@ cmd: 用户空间传递的命令，可以根据不同的命令做不同的事情

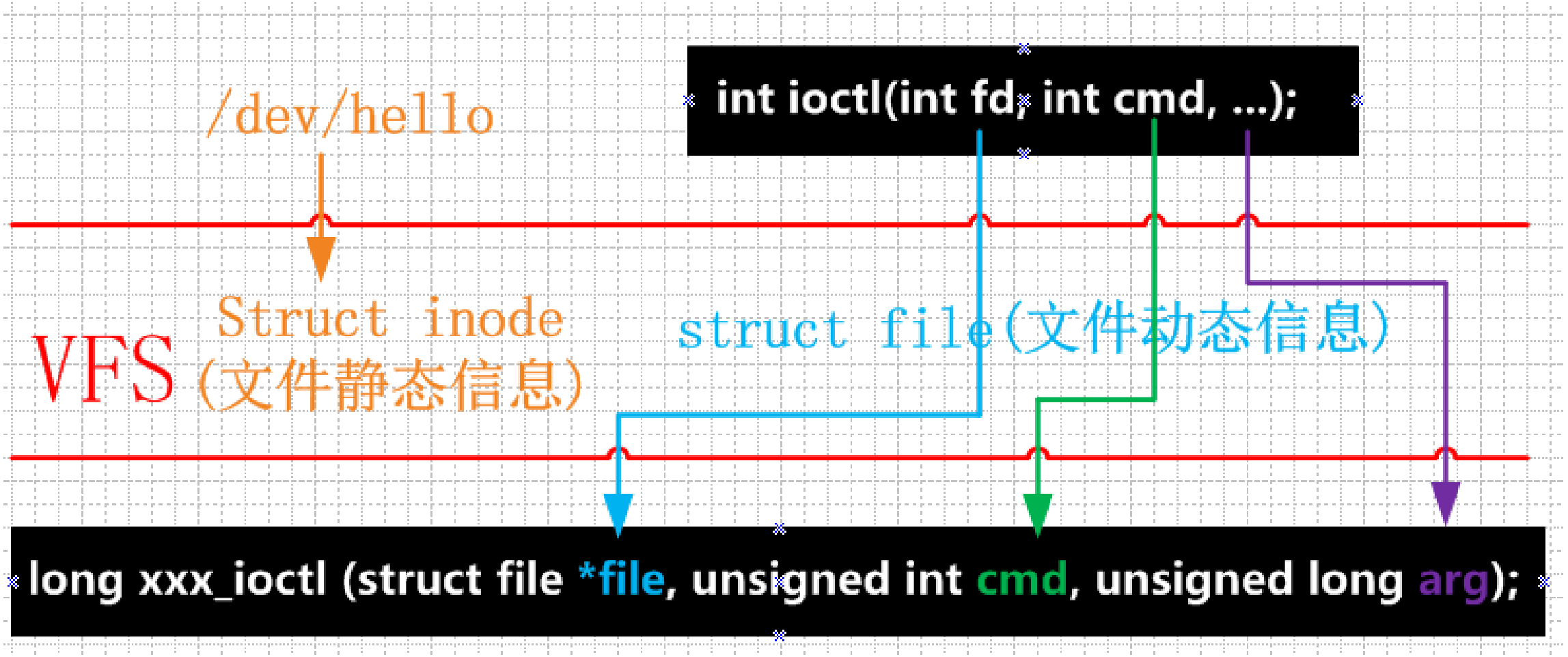
@第三个参数: 用户空间的数据，主要这个数据可能是一个地址值(用户空间传递的是一个地址)，也可能是一个数值，也可能没值

返回值

成功：0

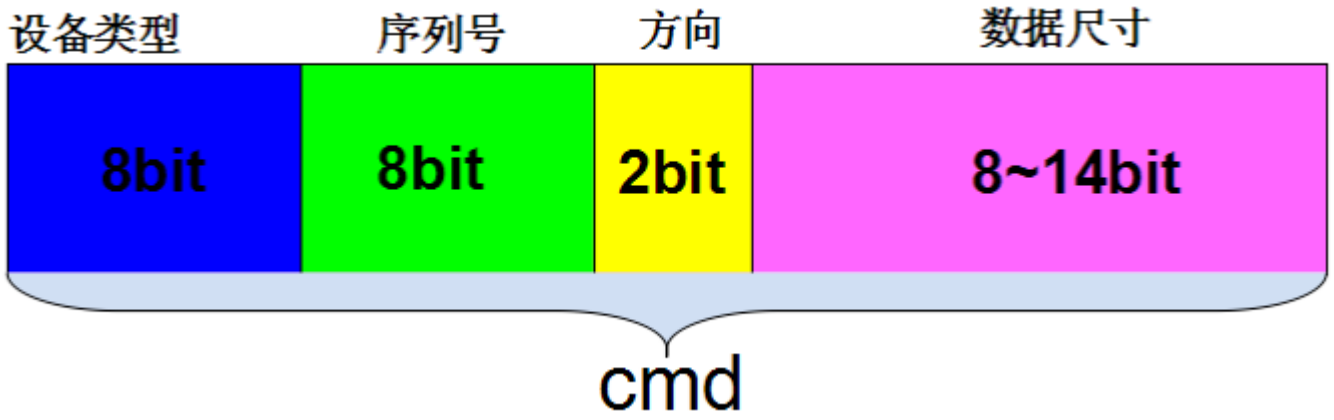
失败：带错误码的负值

调用关系



公众号：一口Linux

cmd



设备类型	类型或叫幻数，代表一类设备，一般用一个字母或者1个8bit的数字
序列号	代表这个设备的第几个命令
方 向	表示是由内核空间到用户空间，或是用户空间到内核空间，入：只读，只写，读写，其他
数据尺寸	表示需要读写的参数大小

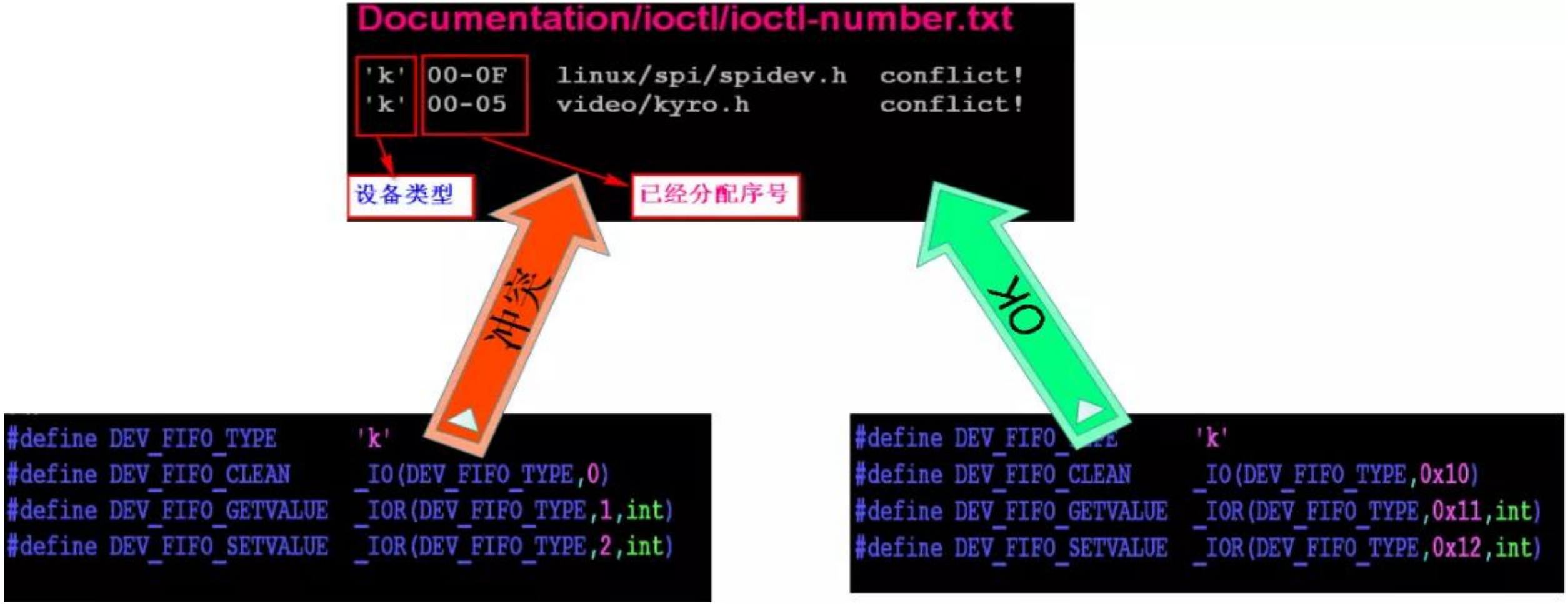
公众号：一口Linux

封装命令

```
/***********************************************
 *type : 设备类型
 *nr   : 命令序号
 *size : 用户传递的数据类型(int ,char ,struct name ..)
 ***********************************************/
/* used to create numbers */
_IO(type,nr)              :没有数据传递的命令
_IOR(type,nr,datatype)    :从驱动中读取数据
_IOW(type,nr,datatype)    :向驱动中写入数据
_IOWR(type,nr,datatype)   :双向传送
-----------------------------------------------------
_IOC_NONE                 :值为0,无数据传输
_IOC_READ                 :值为1,从设备驱动读取数据
_IOC_WRITE                :值为2,向设备驱动写入数据
_IOC_READ|_IOC_WRITE      :双向数据传送
-----------------------------------------------------
/* used to decode ioctl numbers.. */
_IOC_DIR(cmd)             :从命令中提取方向
_IOC_TYPE(cmd)            :从命令中提取设备类型
_IOC_NR(cmd)              :从命令中提取序号
_IOC_SIZE(cmd)            :从命令中提取数据大小
```

公众号：一口Linux

举例



公众号：一口Linux

如何检查命令、地址正确性？

- 可以通过宏_IOC_TYPE（nr）来判断应用程序传下来的命令type是否正确；

- 可以通过宏_IOC_DIR(nr)来得到命令是读还是写，然后再通过宏access_ok(type,addr,size)来判断用户层传递的内存地址是否合法。

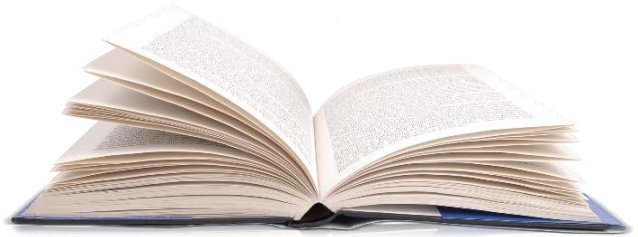
公众号：一口Linux

```
if(_IOC_TYPE(cmd)!=DEV_FIFO_TYPE){
    pr_err("cmd   %u,bad magic 0x%x/0x%x.\n",cmd,_IOC_TYPE(cmd),DEV_FIFO_TYPE);
    return-ENOTTY;
  }
  if(_IOC_DIR(cmd)&_IOC_READ)
    ret=!access_ok(VERIFY_WRITE,(void __user*)arg,_IOC_SIZE(cmd));
  else if( _IOC_DIR(cmd)&_IOC_WRITE )
    ret=!access_ok(VERIFY_READ,(void   __user*)arg,_IOC_SIZE(cmd));
  if(ret){
    pr_err("bad   access %ld.\n",ret);
    return-EFAULT;
  }
```

公众号：一口Linux

15

进程、文件描述符、file、inode、设备号关系



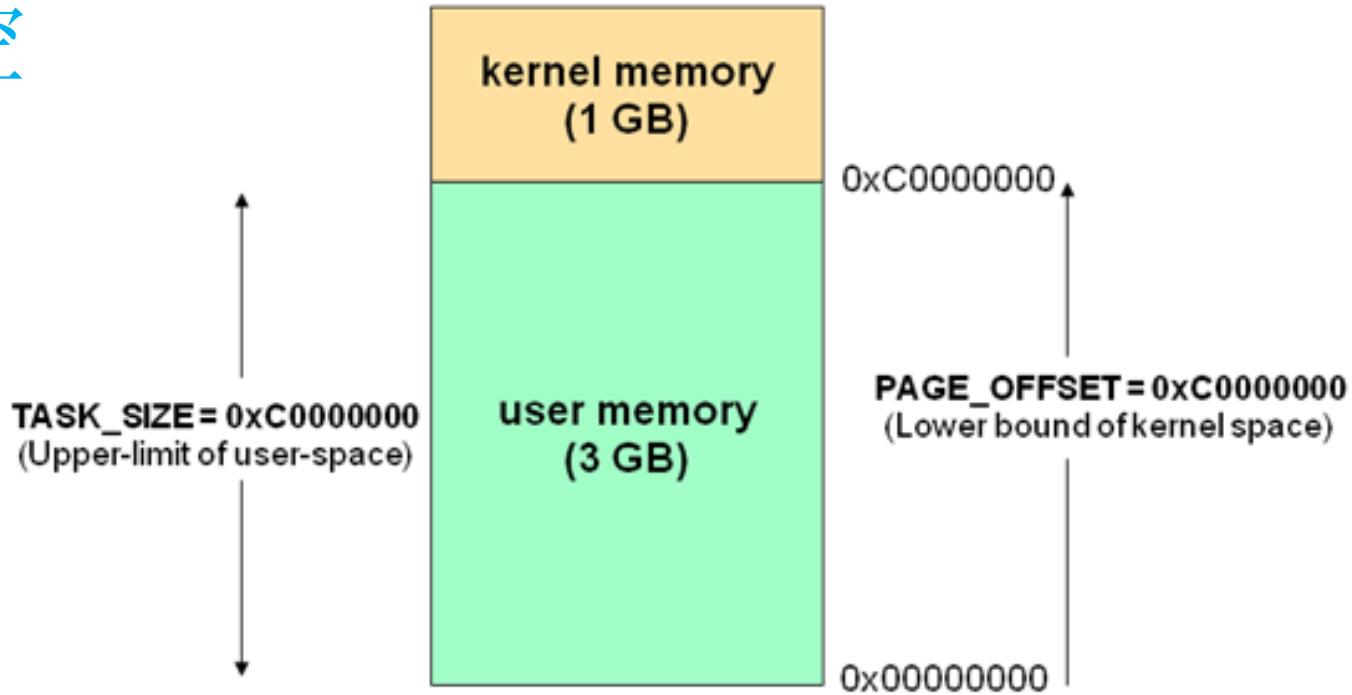
公众号：一口Linux

用户空间和内核空间地址问题

通常32位Linux内核地址空间划分：

0~3G为用户空间
3~4G为内核空间

程序中看到的地址都是虚拟地址

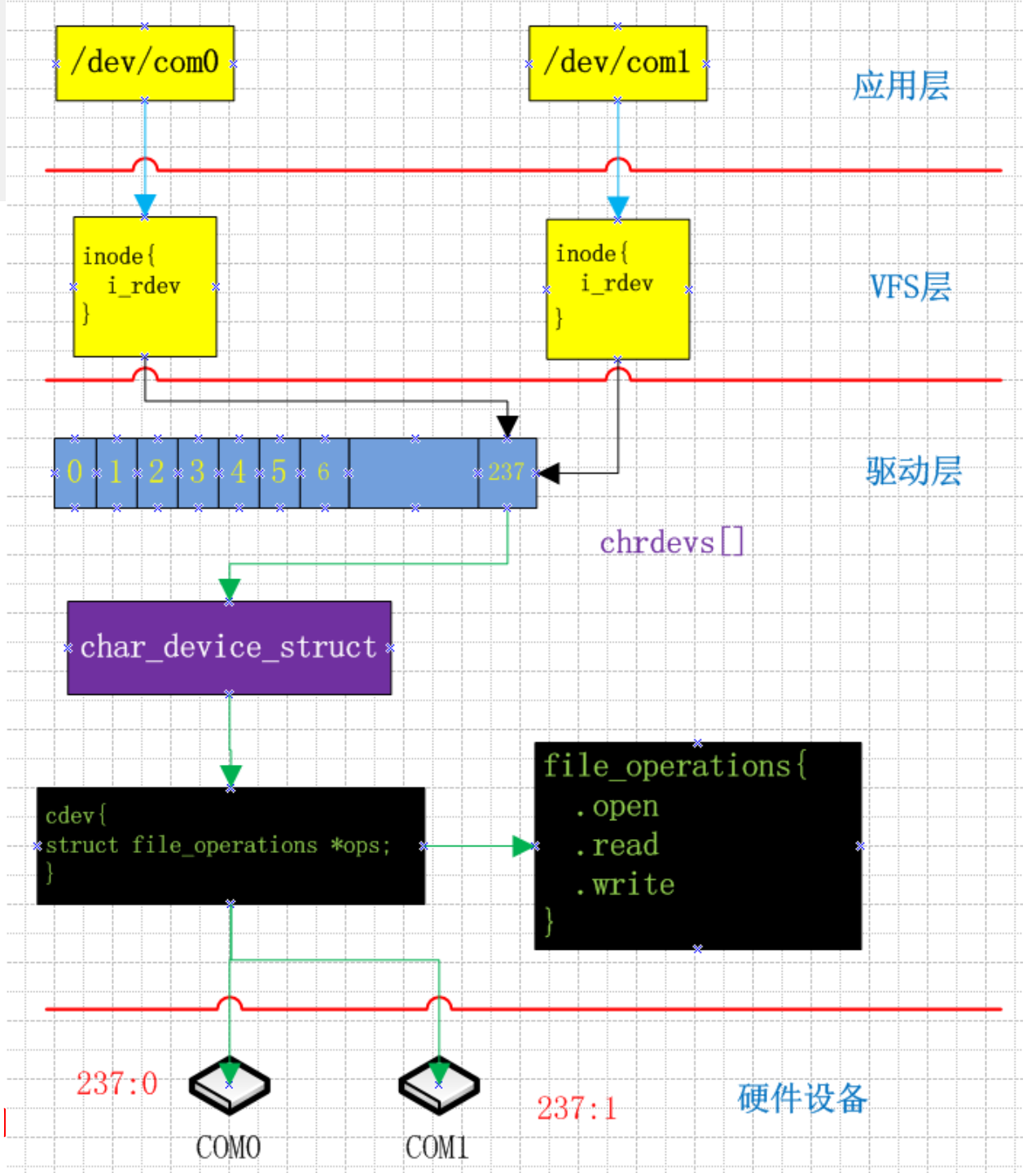


文件、inode

mknod /dev/com0 c 237 0

mknod /dev/com1 c 237 1

1. Inode中存放静态信息
2. 可以多个文件对应一个设备
3. 设备可以不存在



公众号：一

进程与文件描述符

1. lsof
2. /proc/pid/fd/

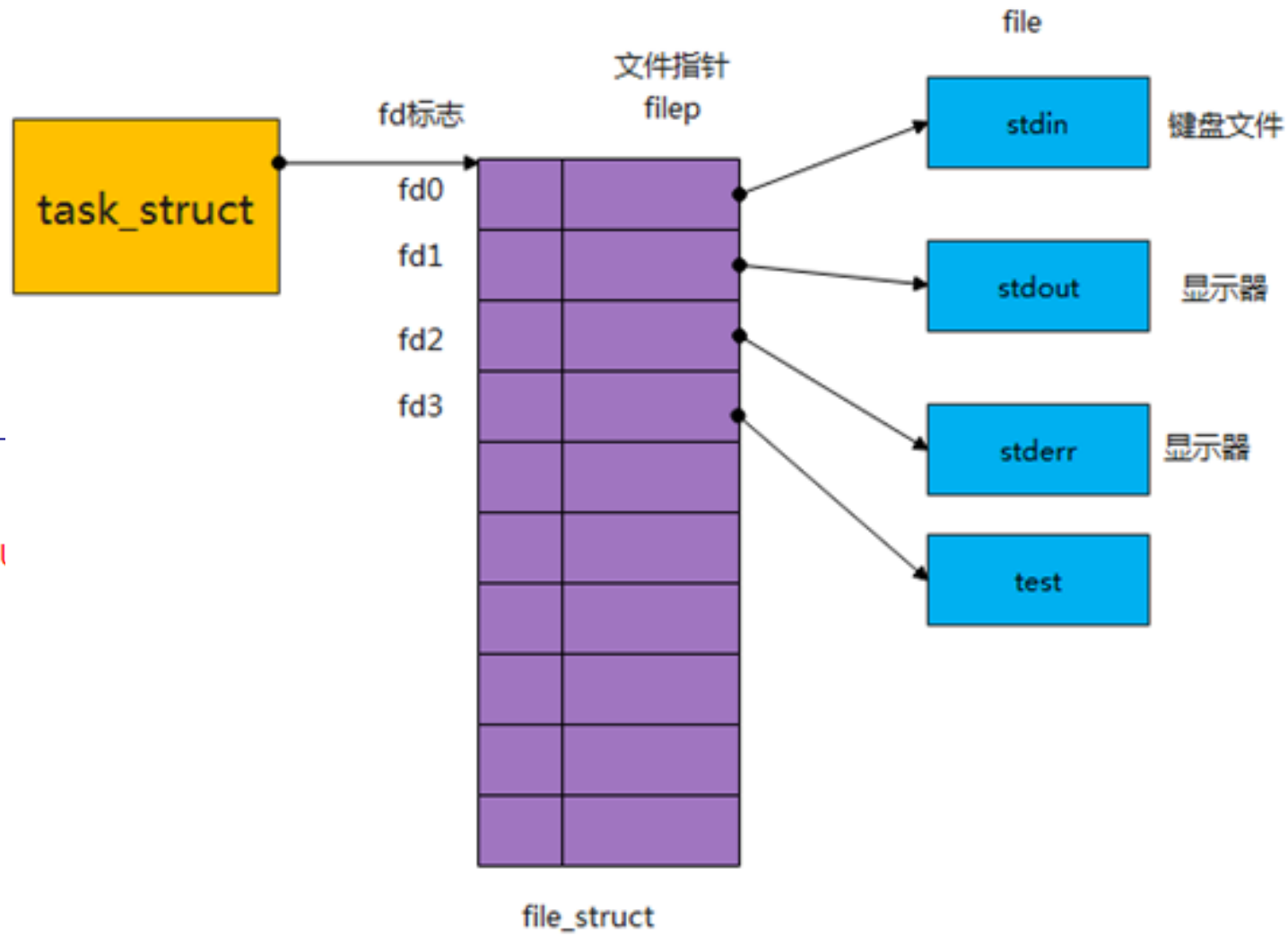
```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
main()
{
    int fd;
    fd = open("test",O_RDWR|O_CREAT);
    if(fd<0)
    {
        perror("open fail \n");
        return;
    }
    sleep(100);
    printf("open ok \n ");
}
```

```
root@ubuntu:/home/peng/driver/2/cdev# ./a.out &
[1] 7298
root@ubuntu:/home/peng/driver/2/cdev# lsof test
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
a.out   7298 root    3u   REG    8,1        0 939477 test
root@ubuntu:/home/peng/driver/2/cdev# ls -l /proc/7298/fd
总用量 0
lrwx------ 1 root root 64 Aug 17 01:57 0 -> /dev/pts/0
lrwx------ 1 root root 64 Aug 17 01:57 1 -> /dev/pts/0
lrwx------ 1 root root 64 Aug 17 01:57 2 -> /dev/pts/0
lrwx------ 1 root root 64 Aug 17 01:57 3 -> /home/peng/driver/2/cdev/test
```

进程与文件描述符

```
01338: /* open file information */
01339:       struct files_struct *files;
```

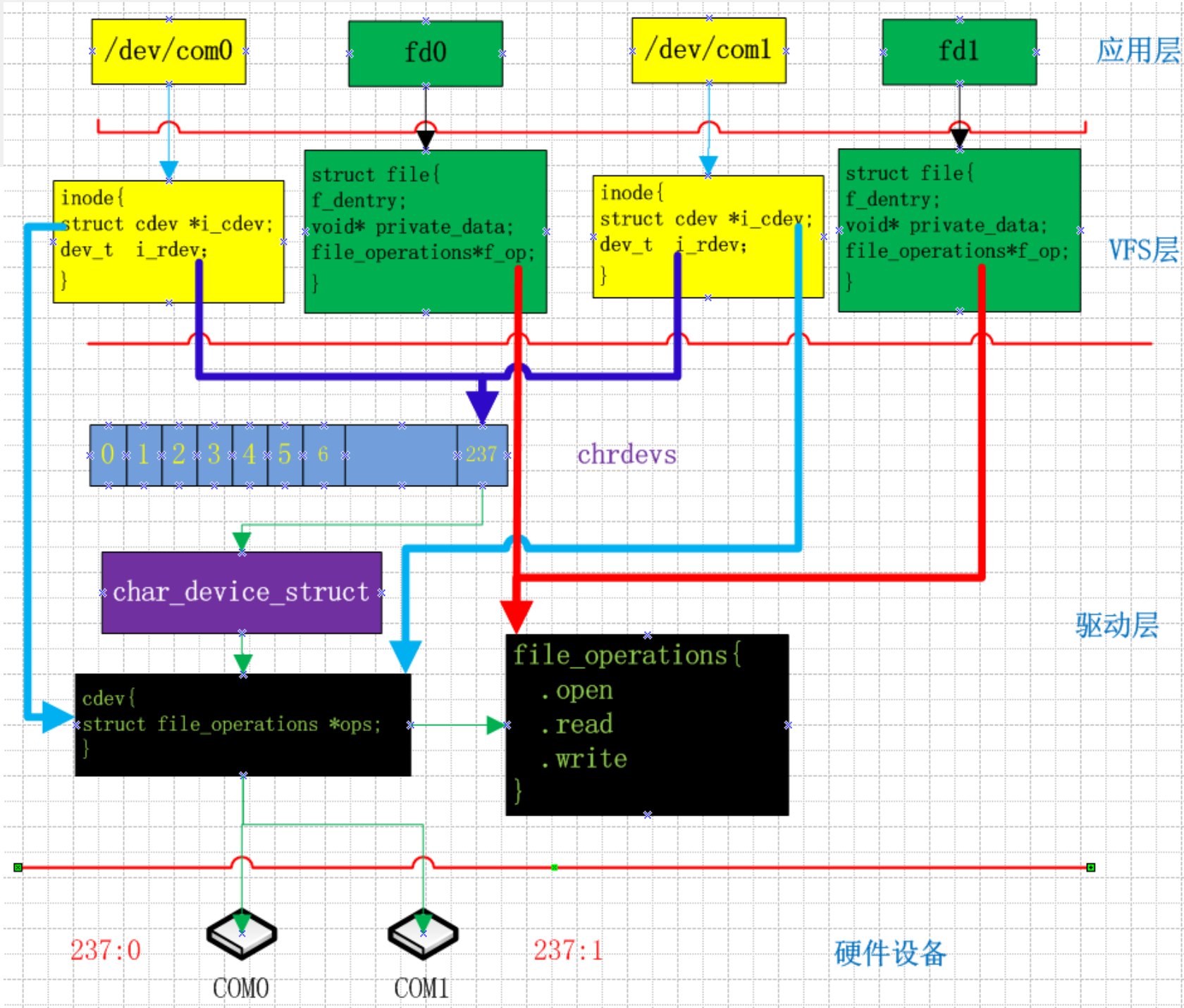
```
struct files_struct {
  /*
   * read mostly part
   */
      atomic_t count;
      struct fdtable __rcu *fdt;
      struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
      spinlock_t file_lock ____cacheline_aligned_
      int next_fd;
      unsigned long close_on_exec_init[1];
      unsigned long open_fds_init[1];
      struct file __rcu * fd_array[NR_OPEN_DEFAU
};
```



1. file存放动态信息
2. file与fd一一对应

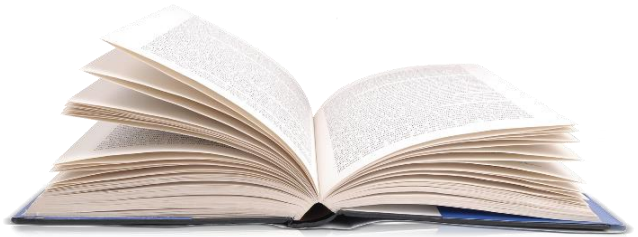
打开文件

```
fd0 = open("/dev/com0",O_RDWR);
fd1 = open("/dev/com1",O_RDWR);
```



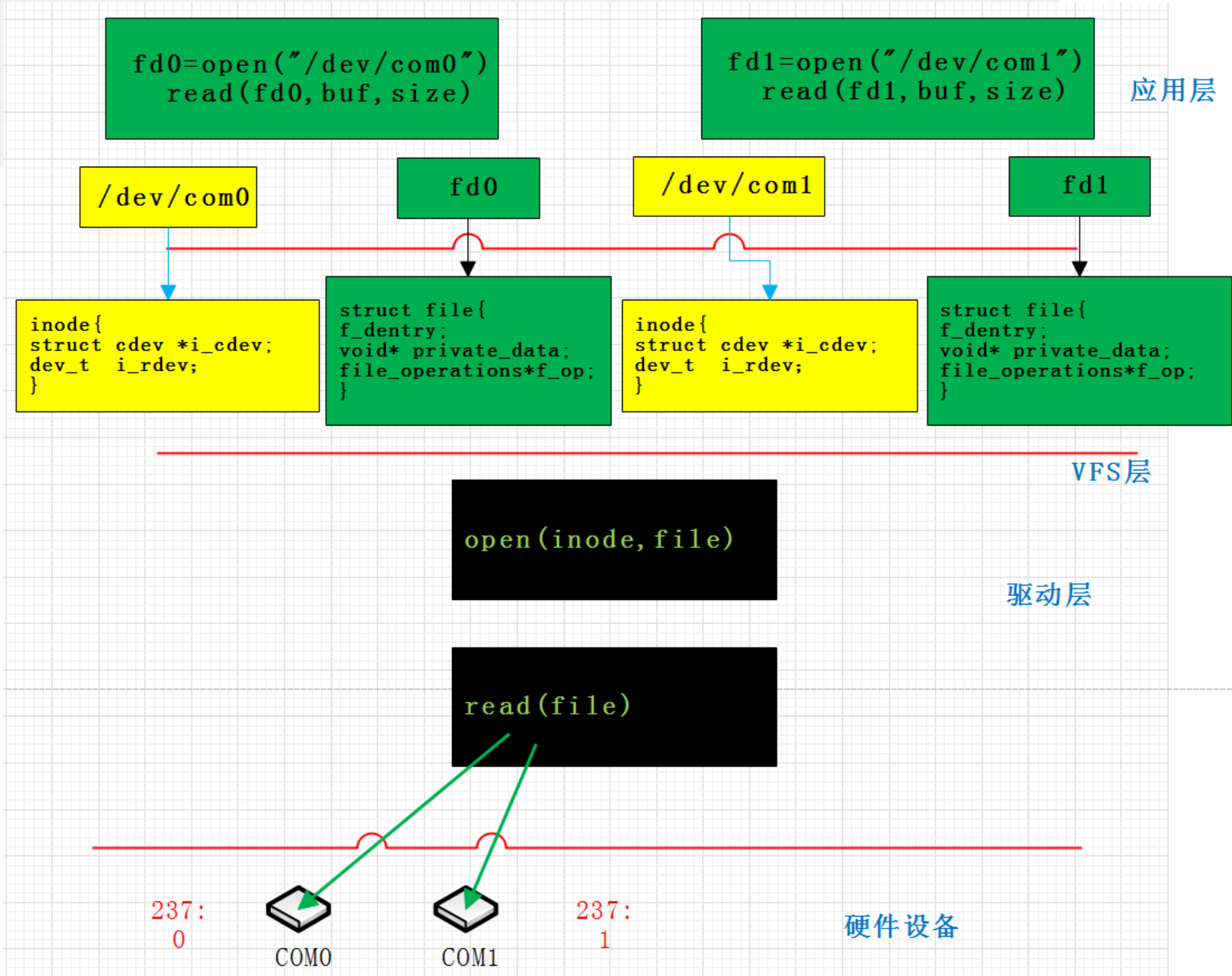
16

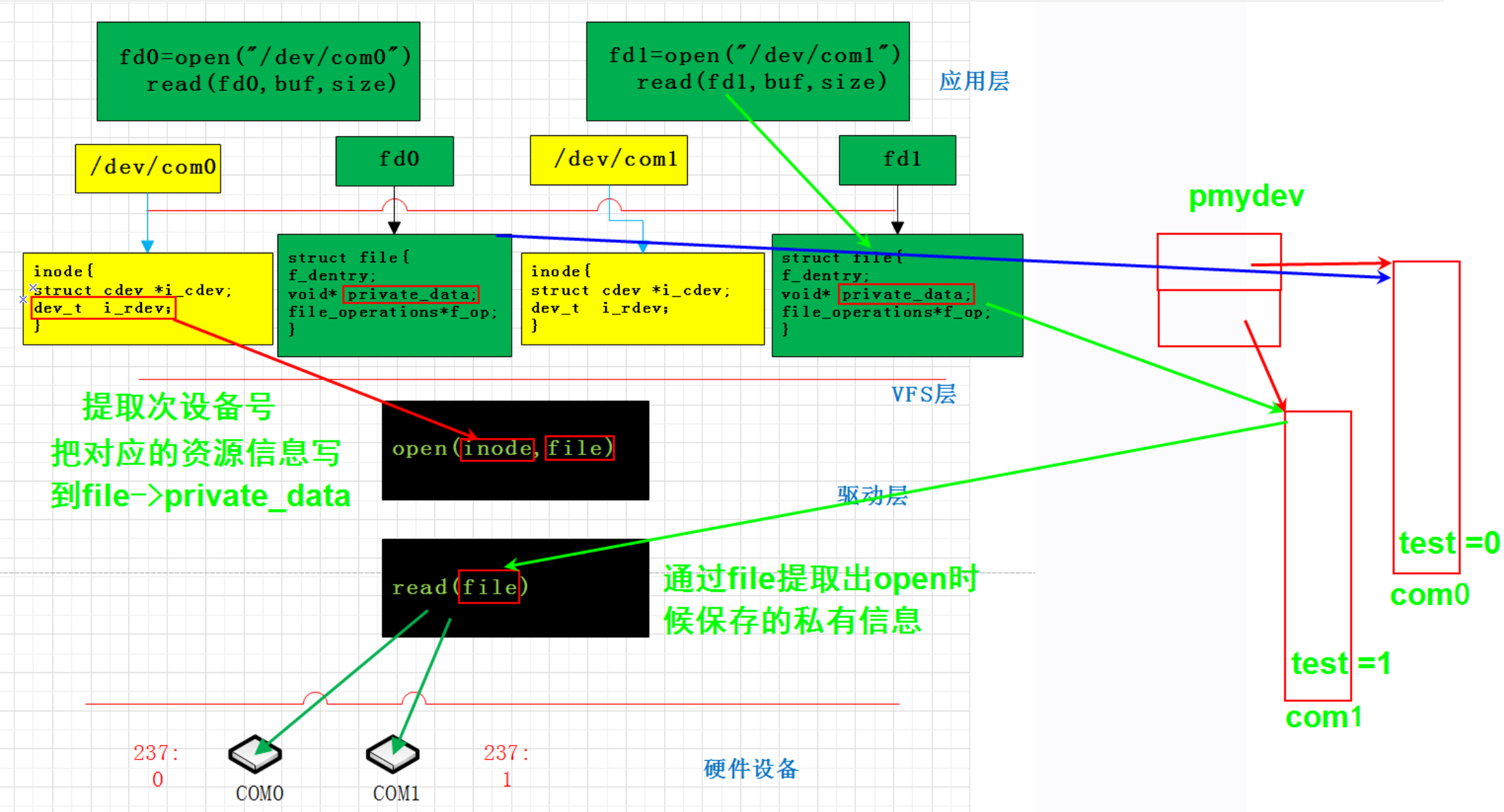
如何同时支持多个次设备？



公众号：一口Linux

框架


fd0=open("/dev/com0")
read(fd0,buf,size)
fd1=open("/dev/com1")
read(fd1,buf,size)
应用层
/dev/com0
fd0
/dev/com1
fd1
inode{
struct cdev *i_cdev;
dev_t i_rdev;
}
struct file{
f_dentry;
void* private_data;
file_operations*f_op;
}
inode{
struct cdev *i_cdev;
dev_t i_rdev;
}
struct file{
f_dentry;
void* private_data;
file_operations*f_op;
}
VFS层
open(inode,file)
驱动层
read(file)
237:
0
COM0
COM1
237:
1
硬件设备


fd0=open("/dev/com0")
read(fd0,buf,size)
fd1=open("/dev/com1")
read(fd1,buf,size)
应用层
/dev/com0
fd0
/dev/com1
fd1
pmydev
inode{
struct cdev *i_cdev;
dev_t i_rdev;
}
struct file{
f_dentry;
void* private_data;
file_operations*f_op;
}
inode{
struct cdev *i_cdev;
dev_t i_rdev;
}
struct file{
f_dentry;
void* private_data;
file_operations*f_op;
}
VFS层
提取次设备号
把对应的资源信息写
到file->private_data
open(inode,file)
驱动层
read(file)
通过file提取出open时
候保存的私有信息
test =0
com0
test =1
com1
237:
0
COM0
COM1
237:
1
硬件设备

实现

定义

```
#define MAX_COM_NUM 2

struct mydev{
      char *reg;
      int test;
};
struct mydev *pmydev[MAX_COM_NUM];
struct cdev cdev;
```

初始化

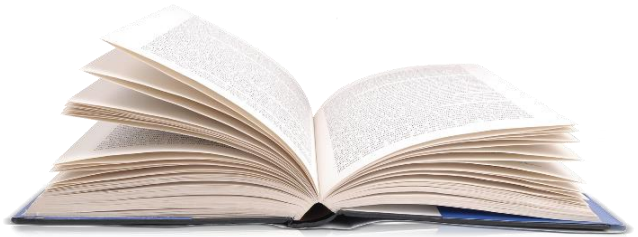
```
for(i=0;i<MAX_COM_NUM;i++)
{
     pmydev[i] =kmalloc(sizeof(struct mydev), GFP_KERNEL);
}

for(i=0;i<MAX_COM_NUM;i++)
{
     pmydev[i]->test = i;
}

cdev_init(&cdev,&dev_fifo_ops);
devno = MKDEV(major,0);
error = cdev_add(&cdev,devno,2);
```

```
static int dev_fifo_open (struct inode *inode, struct file *file)
{
     struct mydev *cd;

     printk("monor = %d \n",MINOR(inode->i_rdev));
     cd = pmydev[MINOR(inode->i_rdev)];

     file->private_data = cd;
     return 0;
}
```

```
ssize_t dev_fifo_read (struct file *file, char __user *buf, size_t size, loff_t *pos
{
     struct mydev *cd;

     cd = (struct mydev *)file->private_data;
     printk("read()       file->private_data        cd->test=%d\n",cd->test);

     if(copy_to_user(buf, &minor, size)){
          return -EFAULT;
     }

     return size;
}
```

17

Linux中的并发机制



公众号：一口Linux

并发

- 多个执行单元同时进行或多个执行单元微观串行执行，宏观并行执行。

公众号：一口Linux

竟态

- 并发的执行单元对共享资源（硬件资源和软件上的全局变量）的访问而导致的竟态状态。

公众号：一口Linux

临界资源

- **临界资源**
  - 多个进程访问的资源
- **临界区**
  - 多个进程访问的代码段

并发场合1

- **单CPU之间进程间的并发**
- 时间片轮转，调度进程。A进程访问打印机，时间片用完，OS调度B进程访问打印机。

并发场合2

- **单cpu上进程和中断之间并发**
- 　CPU必须停止当前进程的执行去执行中断，
- 如：进程A访问串口，此时产生中断请求，此时OS必须放弃进程的执行，去执行中断处理函数，进行中断处理。

并发场合3

- **多cpu之间**
- CPU 1访问打印机，CPU 2 也访问打印机。

并发场合4

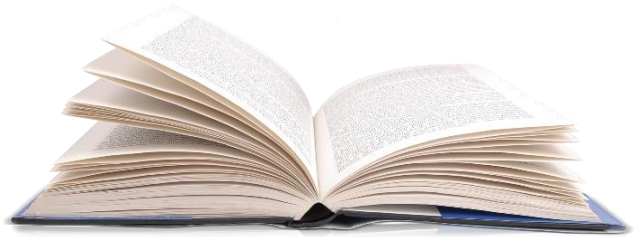
- **单CPU上中断之间的并发**
- 中断都有优先级，如果CPU在处理中断时候，来了一个优先级更高的中断，此时CPU就会放弃此次中断处理而去响应优先级的中断
- 如：中断A访问串口，中断B产生，中断B优先级>中断A，CPU会立即响应中断B，而B也要访问串口资源。

Linux并发解决方案

- **中断**
  - 中断屏蔽，【一般不使用】
  - 单cpu：OK
  - 多CPU：单个CPU屏蔽了中断，其他CPU上运行的进程还是有可能访问设备，产生竟态。
  - 一般不使用，一旦中断屏蔽，很多事情都做不了了，访问共享资源的时间尽量短，以便尽快恢复中断。
- **原子操作**
  - 不可分割，
- **自旋锁**
- **信号量**
- **互斥体**

18

一个有问题的并发控制



公众号：一口Linux

举例

- 要求
- 多个进程打开同一个设备的情况。
- 如果设备忙（有进程打开）返回-EBUSY；
- 不用互斥机制用flag来实现，分析一下可能出现的问题。

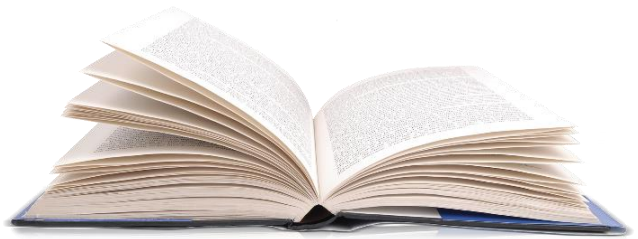
代码

```
open（ ）
{
   if(flag!=0)
   {
       return -EBUSY;
    }//在此处被调度
    flage=1;
}
release()
{
    flage=0;
}
```


P1
flage = 1
P2
int flage = 1; // 1 : available 0:
static int hello_open (struct
{
if(flage == 0)
{
return -EBUSY;
}
open
时间片到了
close
在这段代码之间产生
调度都会出现两个任
务同时打开一个设备
int flage = 1; // 1 : available 0:
static int hello_open (struct
{
if(flage == 0)
{
return -EBUSY;
}
printk("hello_open()\n");
flage = 0;
return 0;
}
任务P1/P2都成功打开设备
printk("hello_open()\n");
flage = 0;
return 0;
}
时间片到了
该时间内都有可能任务切换
static int hello_release (stru
{
printk("hello_release()\n");
flage = 1;
return 0;
}
static int hello_release (stru
{
printk("hello_release()\n");
flage = 1;
return 0;
}

19

Linux驱动-原子操作



公众号：一口Linux

概念

- 原子操作是指不被打断的操作，即它是最小的执行单位。

- 最简单的原子操作就是一条条的汇编指令(不包括一些伪指令，伪指令会被汇编器解释成多条汇编指令)。

公众号：一口Linux

定义

在 linux 中原子操作对应的数据结构为 atomic_t，定义如下：

```
typedef struct {
        int counter;
} atomic_t;
```

公众号：一口Linux

常用操作操作1

- **赋值操作**
- #define atomic_set(v,i) (((v)->counter) = (i))
- **读操作**
- #define atomic_read(v) (*(volatile int *)&(v)->counter)
- **加操作**
- static inline void atomic_add(int i, atomic_t *v)

常用操作操作2

- #define atomic_inc_and_test(v)
  (atomic_inc_return(v) == 0)
- **将原子变量的值加1，并判断是不是0，中间操作不会被打断**
- **为0返回为真**

- #define atomic_dec_and_test(v)
  (atomic_dec_return(v) == 0)
- **将原子变量的值减1，并判断是不是0，中间操作不会被打断**
- **为0返回为真**

公众号：一口Linux

举例

- **初始化**
- **static atomic_t v= ATOMIC_INIT(1);**
- **Open调用**
- **if(!atomic_dec_and_test(&v))**
- **//减一之后判断是否是0**
- { //如果原子变量不是1
- **atomic_inc(&v);**
- return -EBUSY;
- }

公众号：一口Linux

P1

fd=open()

v -1

p2

fd=open()

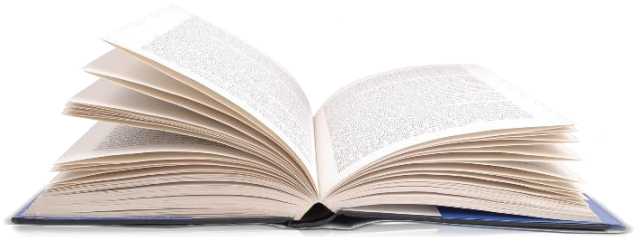
```
static int hello_open (struct inod
{
    if(! atomic_dec_and_test(&v))
    {
        atomic_inc(&v);
        return -EBUSY;
    }

    return 0;
}
```

```
static int hello_open (struct inod
{
    if(! atomic_dec_and_test(&v))
    {
        atomic_inc(&v);
        return -EBUSY;
    }
    return 0;
}
```

```
static int hello_release (struct inode *inode, struct file *filep)
{
    atomic_inc(&v);
    printk("hello_release()\n");
    return 0;
}
```

20

Linux内核-互斥锁



公众号：一口Linux

互斥体概述

- 互斥概念
- 信号量是在并行处理环境中对多个处理器访问某个公共资源进行保护的机制，mutex用于互斥操作。

- mutex的语义相对于信号量要简单轻便一些，在锁争用激烈的测试场景下，mutex比信号量执行速度更快，可扩展性更好，另外mutex数据结构的定义比信号量小。

公众号：一口Linux

mutex的使用注意事项：

- 同一时刻只有一个线程可以持有mutex。
- 只有锁持有者可以解锁，不能在一个进程中持有mutex，在另外一个进程中释放他。
- 不允许递归地加锁和解锁。
- 当进程持有mutex时，进程不可以退出。
- mutex必须使用官方API来初始化。
- mutex可以睡眠，所以不允许在中断处理程序或者中断下半部中使用，例如tasklet、定时器等。

公众号：一口Linux

初始化

静态定义：

DEFINE_MUTEX(name);

动态初始化mutex：

mutex_init(&mutex);

公众号：一口Linux

互斥锁的操作

- mutex_lock(struct mutex*)
  - 为指定的mutex上锁，如果不可用则睡眠
- mutex_unlock(struct mutex*)
  - 为指定的mutex解锁
- mutex_trylock(struct mutex*)
  - 尝试获取指定的mutex，如果成功则返回1；否则锁被获取，返回值是0
- mutex_is_lock(struct mutex*)
  - 如果锁已被征用，则返回1；否则返回0
-

公众号：一口Linux

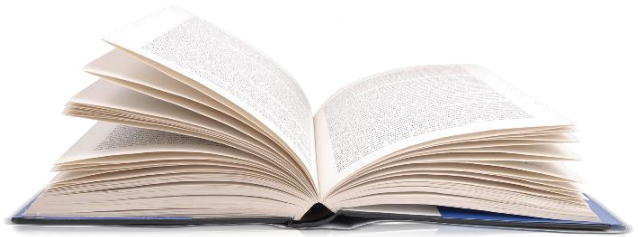
使用实例

```
struct mutex mutex;
mutex_init(&mutex); /*定义*/
//加锁
mutex_lock(&mutex);

//临界区

//解锁
mutex_unlock(&mutex);
```

公众号：一口Linux

21

Linux内核-信号量



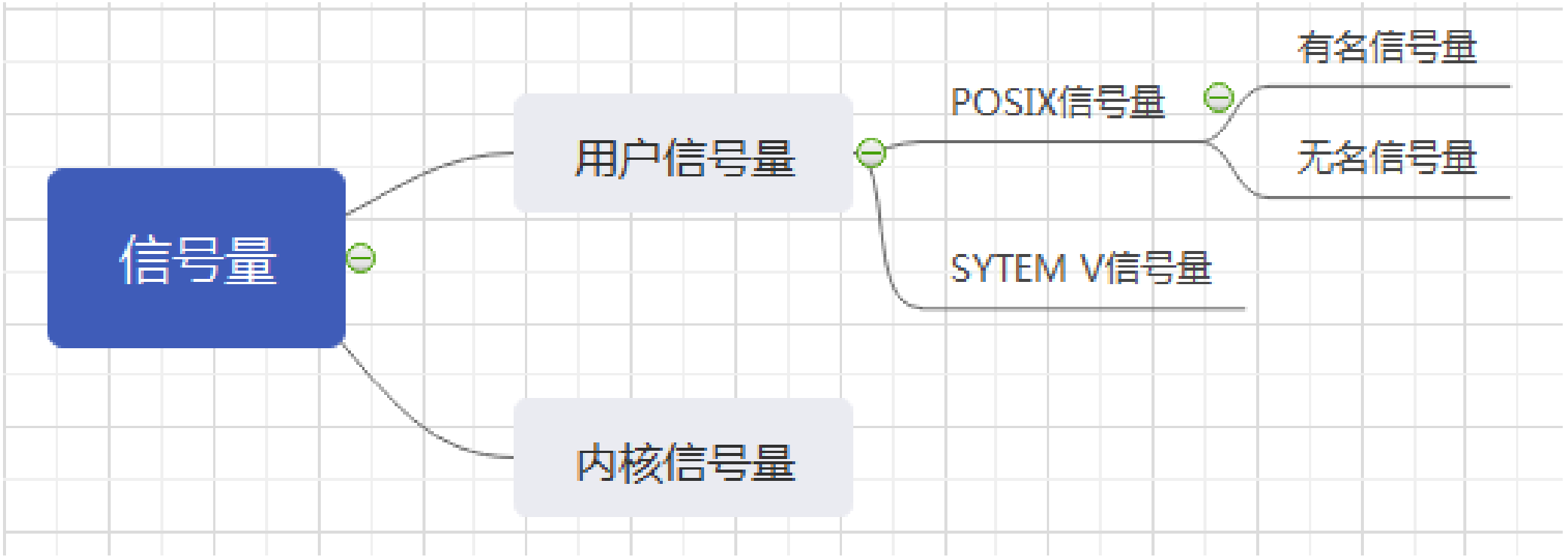
公众号：一口Linux

信号量概念

- 信号量又称为信号灯
- 它是用来协调不同进程间的数据对象的，而最主要的应用是共享内存方式的进程间通信。本质上，信号量是一个计数器，它用来记录对某个资源（如共享内存）的存取状况。

公众号：一口Linux

Linux下几种信号量



公众号：一口Linux

核心操作

- P
- 如果有一个任务想要获得已经被占用的信号量时，信号量会将其放入一个等待队列然后让其睡眠。
- V
- 当持有信号量的进程将信号释放后，处于等待队列中的一个任务将被唤醒（因为队列中可能不止一个任务），并让其获得信号量。
- 举例

公众号：一口Linux

初始化

```
#include <linux/semaphore.h>

struct semaphore sem;
sema_init(&sem,1);
```

PV

- P

```
extern int __must_check down_interruptible(struct semaphore *sem);
extern int __must_check down_killable(struct semaphore *sem);
extern int __must_check down_trylock(struct semaphore *sem);
extern int __must_check down_timeout(struct semaphore *sem, long jiffies);
```

- V

```
extern void up(struct semaphore *sem);
```

公众号：一口Linux

举例

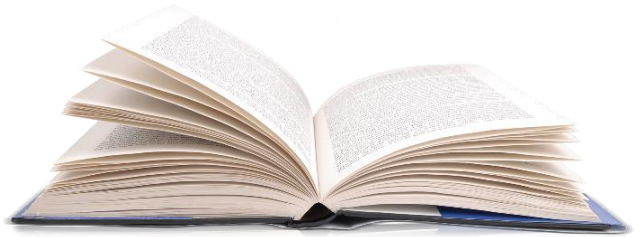
- **struct semaphore sem;**
- **sema_init(&sem,1);**

```
int hello_open(struct inode *inode,struct file *filp)
{
        if(down_interruptible(&sem))
                        return -ERESTARTSYS;
        return 0;
}
int hello_release(struct inode *inode,struct file *filp)
{
        up(&sem);
        return 0;
}
```

公众号：一口Linux

22

自旋锁spinlock-使用



公众号：一口Linux

概念

- 内核当发生访问资源冲突的时候，可以有两种锁的解决方案选择：一个是原地等待 一个是挂起当前进程，调度其他进程执行（睡眠）。

- Spinlock 是内核中提供的一种比较常见的锁机制，自旋锁是“原地等待”的方式解决资源冲突的，即，一个线程获取了一个自旋锁后，另外一个线程期望获取该自旋锁，获取不到，只能够原地“打转”（忙等待）。

公众号：一口Linux

自旋锁的优点

- 自旋锁**不会使线程状态发生切换**，一直处于用户态，即线程一直都是active的；**不会使线程进入阻塞状态，减少了不必要的上下文切换**，执行速度快。

- 非自旋锁在获取不到锁的时候会进入阻塞状态，从而进入内核态，当获取到锁的时候需要从内核态恢复，需要线程上下文切换。（线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能）。

公众号：一口Linux

注意事项

- 进程拥有自旋锁的时候，该cpu上是禁止抢占的
- 一般用于多cpu之间的资源竞争
- 由于自旋锁的这个**忙等待**的特性，注定了它使用场景上的限制 —— 自旋锁不应该被长时间的持有（消耗 CPU 资源），一般应用在中断上下文。

公众号：一口Linux

定义

动态的：

```
spinlock_t lock;
spin_lock_init (&lock);
```

静态的：

```
DEFINE_SPINLOCK(lock);
```

公众号：一口Linux

锁申请/释放

加锁

```
spin_lock(&lock);
```

解锁

```
spin_unlock(&lock);
```

步骤

- 我们要访问临界资源需要首先申请自旋锁
- 获取不到锁就自旋，如果能获得锁就进入临界区
- 当自旋锁释放后，自旋在这个锁的任务即可获得锁并进入临界区，退出临界区的任务必须释放自旋锁

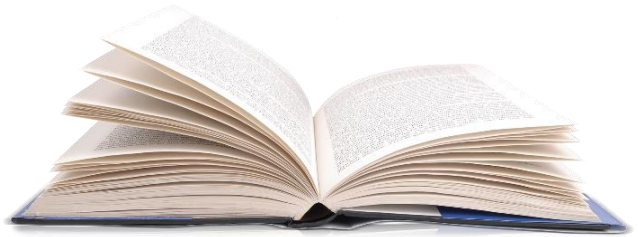
使用实例

```
static spinlock_t lock;
static int flage = 1;

spin_lock_init(&lock);

static int hello_open (struct inode *inode, struct file *filep)
{
    spin_lock(&lock);
    if(flage != 1)
    {
        spin_unlock(&lock);
        return -EBUSY;
    }
    flage = 0;
    spin_unlock(&lock);

    return 0;
}
static int hello_release (struct inode *inode, struct file *filep)
{
    flage = 1;
    return 0;
}
```

公众号：一口Linux

23

自旋锁-死锁



公众号：一口Linux

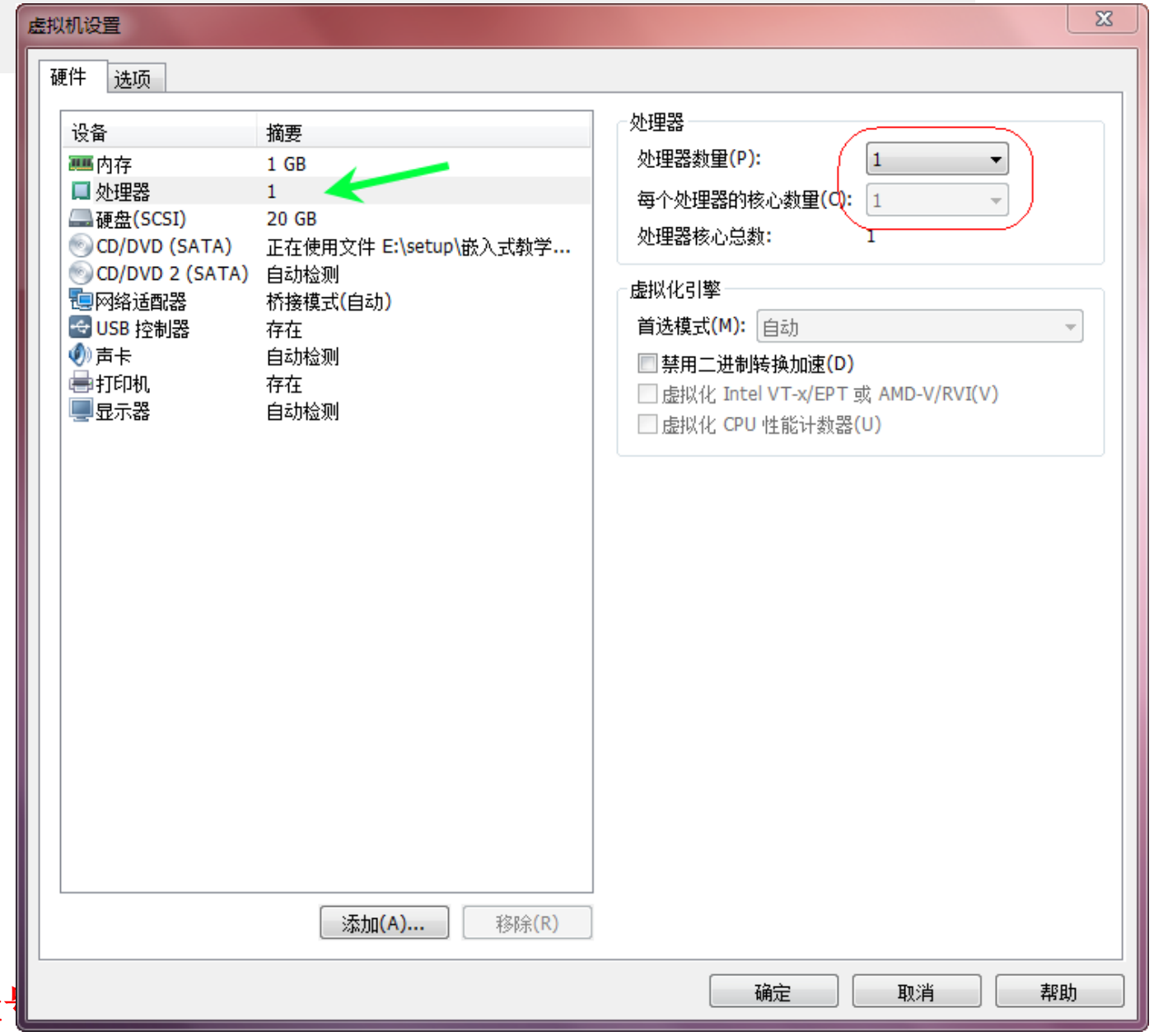
自旋锁的死锁

- **死锁的2种情况**
- 1）拥有自旋锁的进程A在内核态阻塞了，内核调度B进程，碰巧B进程也要获得自旋锁，此时B只能自旋转。而此时抢占已经关闭，（单核）不会调度A进程了，B永远自旋，产生死锁。
- 2）进程A拥有自旋锁，中断到来，CPU执行中断函数，中断处理函数，中断处理函数需要获得自旋锁，访问共享资源，此时无法获得锁，只能自旋，产生死锁。

死锁举例-环境

只在单CPU系统中会出现死锁这个现象

ps –ef | grep softirq



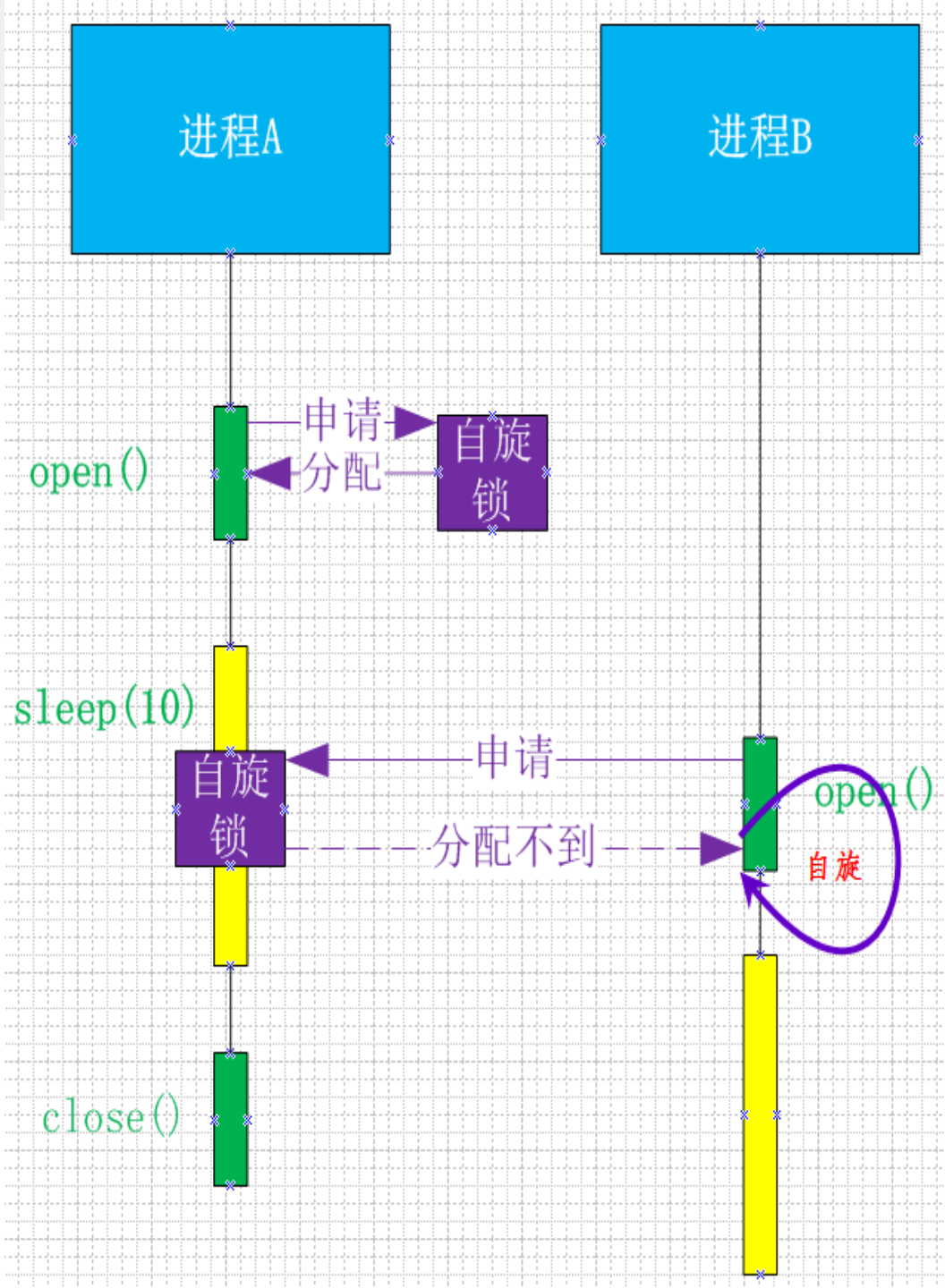
死锁举例

1. 进程A在open（）字符设备后，对应的内核函数会申请自旋锁，此时自旋锁空闲，申请到自旋锁，进程A随即进入执行sleep（）函数进入休眠；

2. 在进程A 处于sleep期间，自旋锁一直属于进程A所有；

3. 运行进程B，进程B执行open函数，对应的内核函数也会申请自旋锁，此时自旋锁归进程A所有，所以进程B进入自旋状态；

4. 因为此时抢占已经关闭，系统死锁。



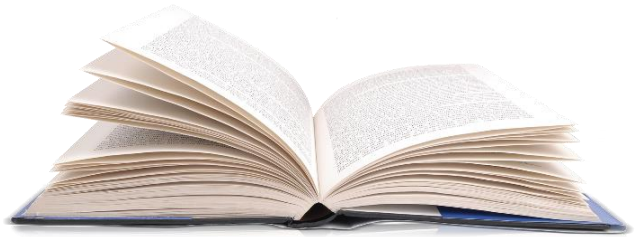
公众号：一口Linux

如何避免死锁

- 1. 如果中断处理函数中也要获得自旋锁，那么驱动程序需要在拥有自旋锁时禁止中断；
- 2. 自旋锁必须在可能的最短时间内拥有
- 3. 避免某个获得锁的函数调用其他同样试图获取这个锁的函数，否则代码就会死锁；
  不论是信号量还是自旋锁，都不允许锁拥有者第二次获得这个锁，如果试图这么做，系统将挂起；
- 4. 锁的顺序规则
  a) 按同样的顺序获得锁;
  b) 如果必须获得一个局部锁和一个属于内核更中心位置的锁，则应该首先获取自己的局部锁；
  c) 如果我们拥有信号量和自旋锁的组合，则必须首先获得信号量；在拥有自旋锁时调用down(可导致休眠)是个严重的错误的。

公众号：一口Linux

24 同步机制的总结



公众号：一口Linux

自旋锁和互斥体使用场合

需求	建议的加锁方法
低开销加锁	优先使用自旋锁
短期锁定	优先使用自旋锁
长期锁定	优先使用互斥体
中断上下文中加锁	使用自旋锁
持有锁需要睡眠	使用互斥体

在中断上下文中只能使用自旋锁，而在任务睡眠时只能使用互斥体。

公众号：一口Linux

举2个生活中的例子1：

- 我们要坐火车从南京到新疆，这个‘任务’特别的耗时，只能在车上等着车到站，但是我们没有必要一直睁着眼睛等着车到站，最好的情况就是我们上车就直接睡觉，醒来就到站，这样从人（用户）的角度来说，体验是最好的。
- 对比于进程，程序在等待一个耗时的任务的时候，没有必须要占用CPU，可以暂停当前任务使其进入休眠状态，当等待的事件发生之后再由其他任务唤醒，这种场景采用信号量、互斥锁比较合适。

公众号：一口Linux

举2个生活中的例子2：

- 我们在等待电梯、等待洗手间，这种场景需要等待的事件并不是很多，如果我们还要找个地方睡一觉，然后等电梯到了或者洗手间可以用了再醒来，那很显然这也没有必要，我们只需要排好队，刷一刷抖音就可以了。

- 对比于计算机程序，比如驱动在进入中断例程，在等待某个寄存器被置位，这种场景需要等待的时间很短暂，系统开销远小于进入休眠的开销，所以这种场景采用自旋锁比较合适。

公众号：一口Linux

信号量和互斥体

- 互斥体和信号量很相似，内核中两者共存会令人混淆。所幸，它们的标准使用方式都有简单规范：除非mutex的某个约束妨碍你使用，否则相比信号量要优先使用mutex。

- 当你写新代码时，只有碰到特殊场合（一般是很底层代码）才会需要使用信号量。因此建议选mutex。如果发现不能满足其约束条件，且没有其他别的选择时，再考虑选择信号量

公众号：一口Linux

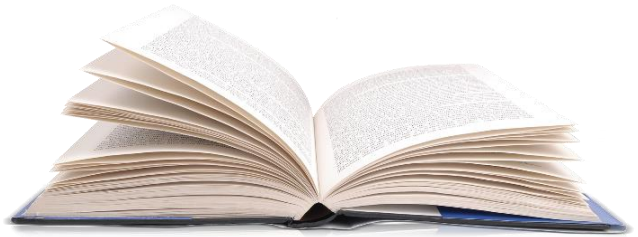
信号量和互斥体

- 互斥体和信号量很相似，内核中两者共存会令人混淆。所幸，它们的标准使用方式都有简单规范：除非mutex的某个约束妨碍你使用，否则相比信号量要优先使用mutex。
- 当你写新代码时，只有碰到特殊场合（一般是很底层代码）才会需要使用信号量。因此建议 选mutex。如果发现不能满足其约束条件，且没有其他别的选择时，再考虑选择信号量

公众号：一口Linux

25

Linux-IO模型



公众号：一口Linux

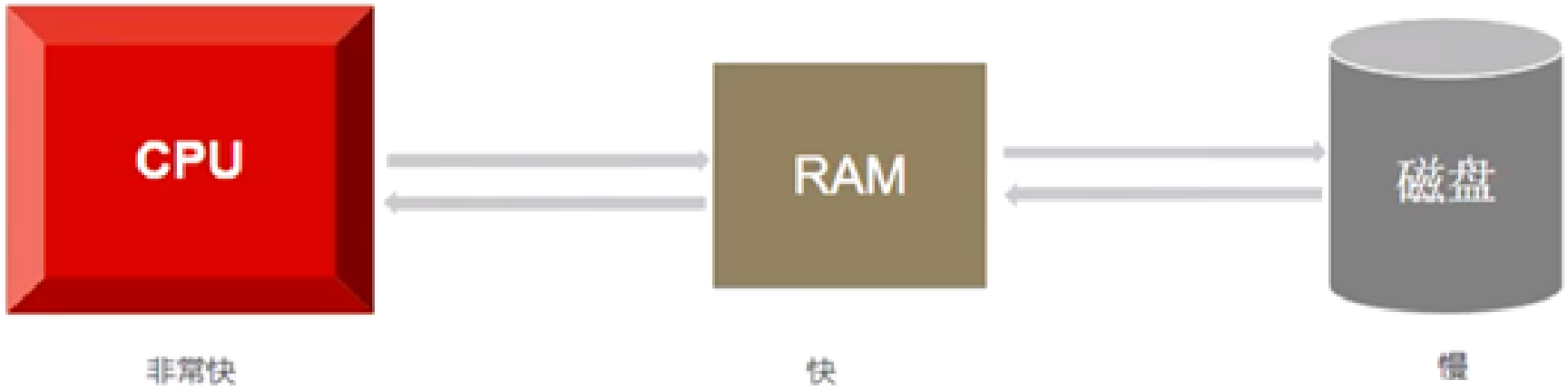
什么是IO？

- 在计算机系统中I/O就是输入（Input）和输出(Output)的意思，只要具有输入输出类型的交互系统都可以认为是I/O系统
- 也可以说I/O是整个操作系统数据交换与人机交互的通道
- 针对不同的操作对象，
  - 可以划分为磁盘I/O模型，网络I/O模型，内存映射I/O, Direct I/O、数据库I/O等。

公众号：一口Linux

那么数据被Input到哪，Output到哪呢？

- Input（输入）数据到内存中，Output（输出）数据到IO设备（磁盘、网络等需要与内存进行数据交互的设备）中



公众号：一口Linux

IO重要性

- 现在系统都有可能处理大量文件，大量数据库操作，而这些操作都依赖于系统的I/O性能，现在系统的瓶颈往往都是由于I/O性能造成的。

- 解决磁盘I/O性能慢的方法：
  - 系统架构中添加了缓存来提高响应速度；
  - 使用了固态硬盘（SSD）来替换传统机械硬盘；

- 一个系统的优化空间，往往都在低效率的I/O环节上，很少看到一个系统CPU、内存的性能是其整个系统的瓶颈。

公众号：一口Linux

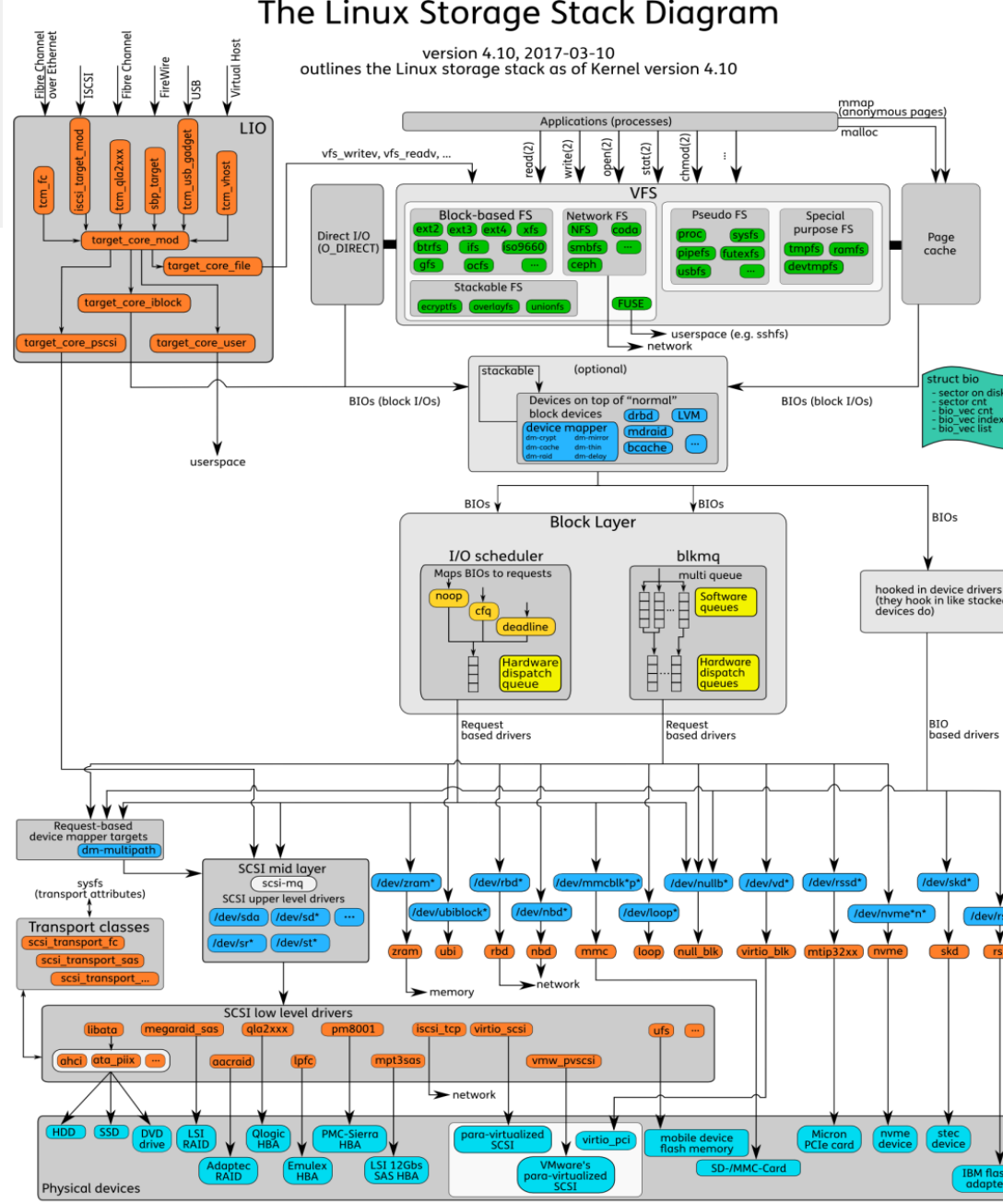
Linux IO栈

Linux下的IO栈致大致有三个层次：

1. 文件系统层，以 write(2) 为例，内核拷贝了write(2)参数指定的用户态数据到文件系统Cache中，并适时向下层同步

2. 块层，管理块设备的IO队列，对IO请求进行合并、排序（还记得操作系统课程学习过的IO调度算法吗？）

3. 设备层，通过DMA与内存直接交互，完成数据和具体设备之间的交互



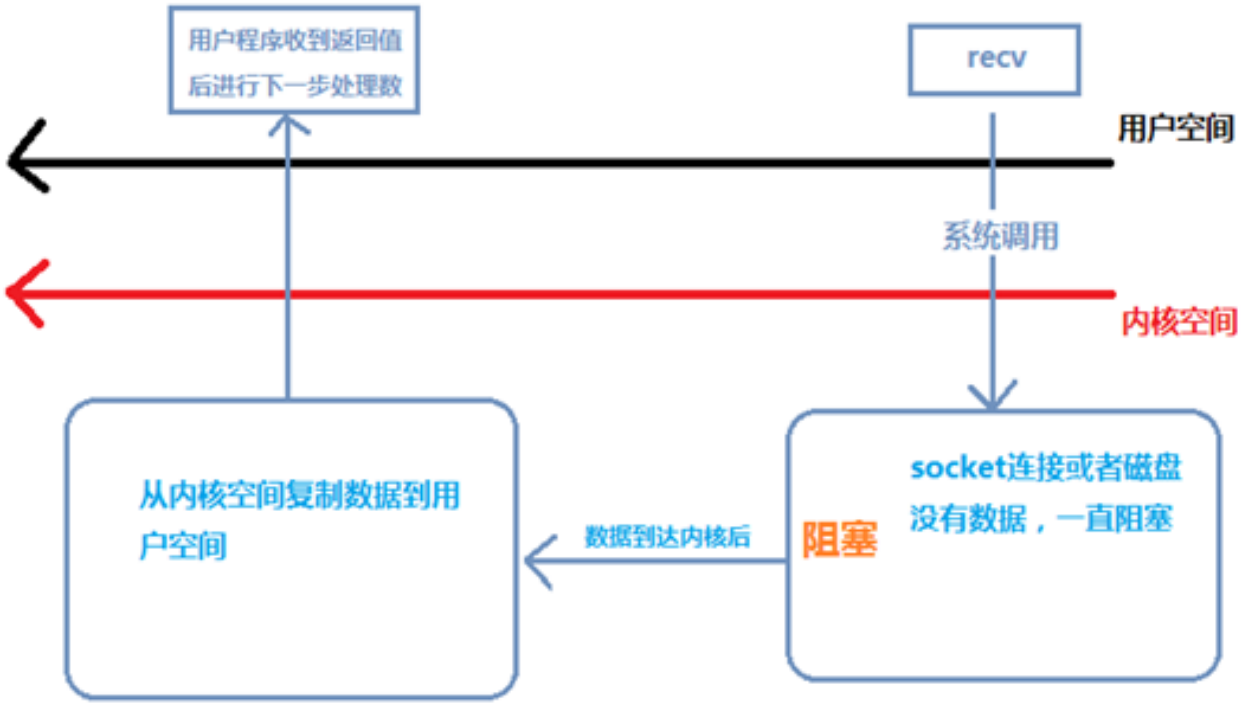
公众号：一口Linux

五种IO模型

- 阻塞式I/O
  非阻塞式I/O
  I/O复用（select，poll，epoll等）
  信号驱动式I/O（SIGIO）
  异步I/O（POSIX的aio_系列函数）

阻塞IO（Blocking IO）

指调用者在调用某一个函数后，一直在等待该函数的返回值，线程处于挂起状态。好比你去商场试衣间，里面有人，那你就一直在门外等着。



优点：

1. 能够及时返回数据，无延迟；
2. 对内核开发者来说这是省事了；
3. 阻塞期间不占用系统资源

缺点：

对用户来说处于等待就要付出性能的代价了；

公众号：一口Linux

非阻塞IO

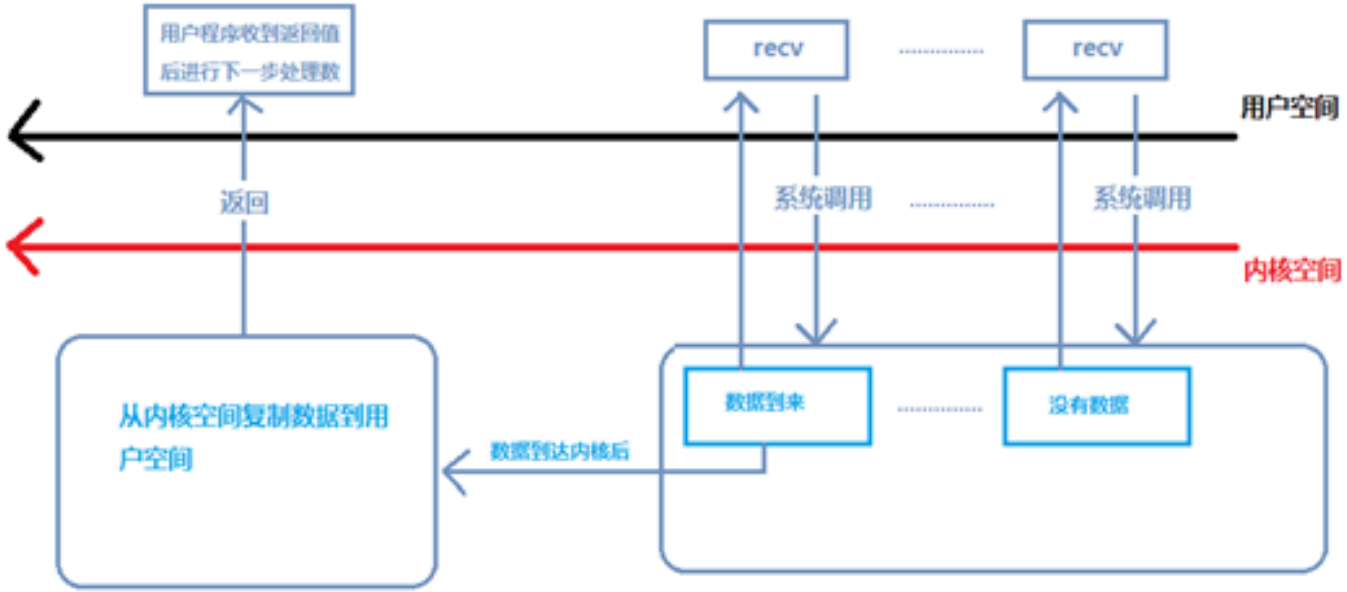
指调用者在调用某一个函数后，不等待该函数的返回值，线程继续运行其他程序

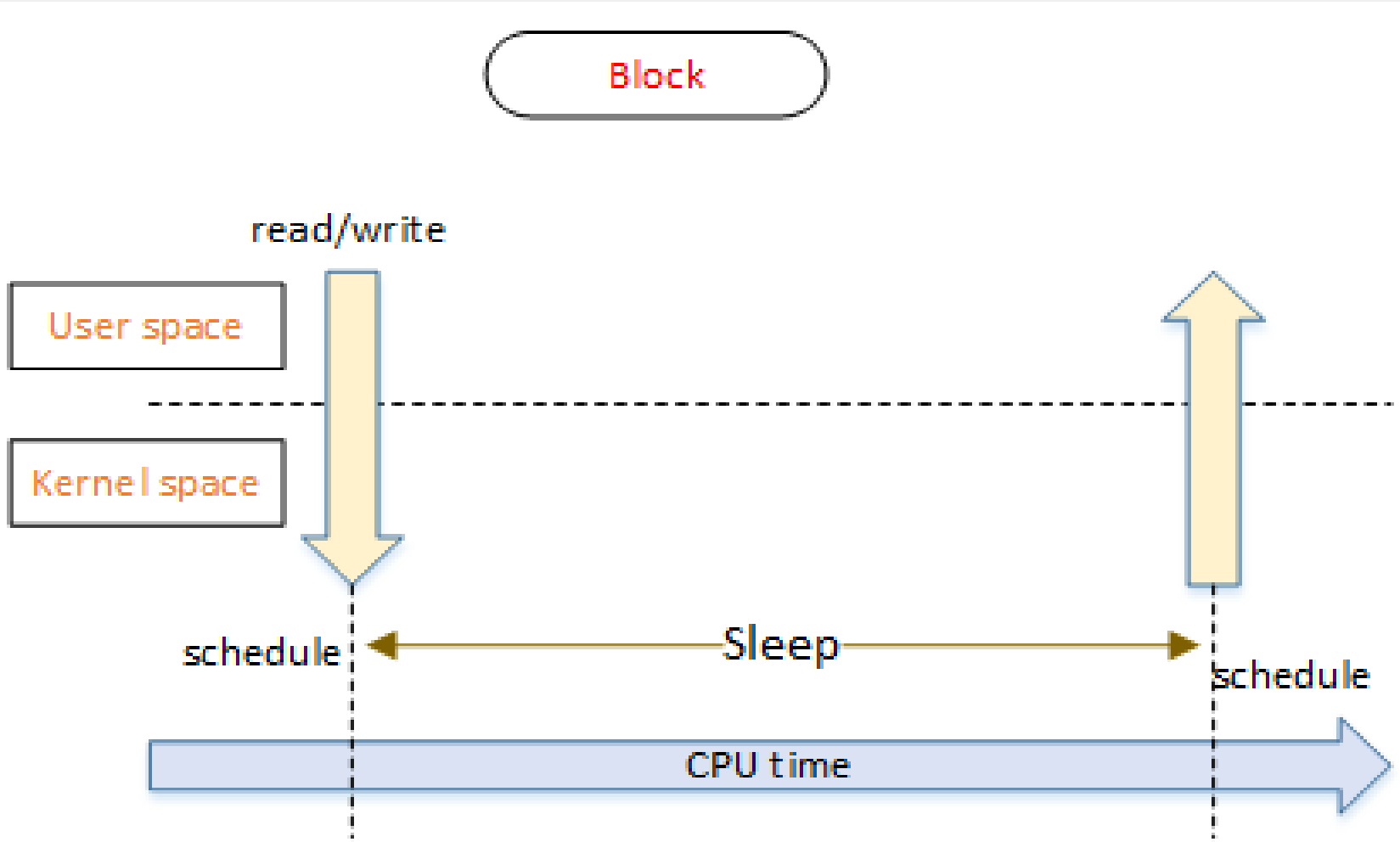
优点：

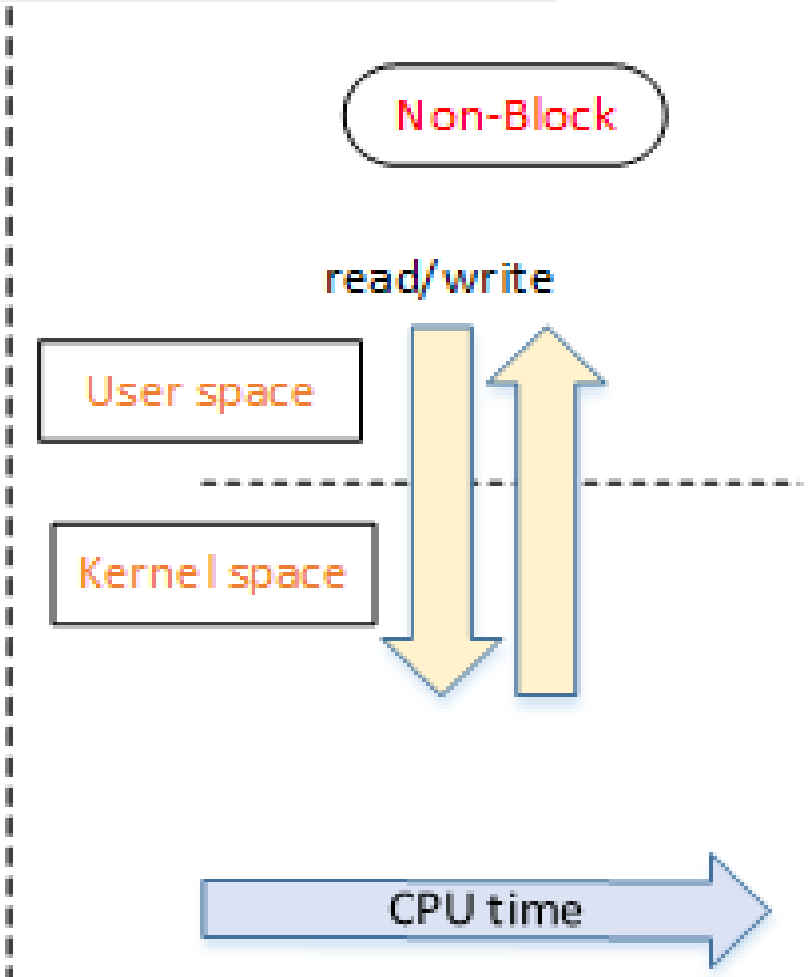
能够在等待任务完成的时间里干其他活了（包括提交其他任务，也就是“后台”可以有多个任务在同时执行）。

缺点：

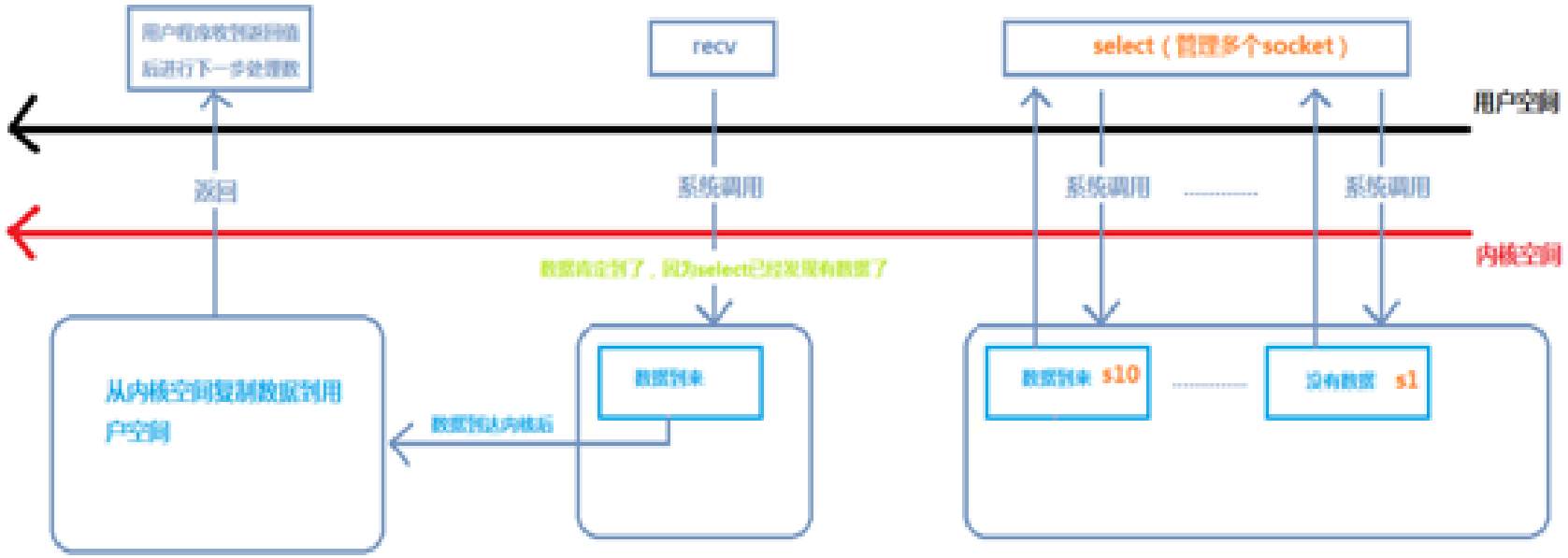
任务完成的响应延迟增大了，因为每过一段时间才去轮询一次read操作，而任务可能在两次轮询之间的任意时间完成。这会导致整体数据吞吐量的降低。




Block
read/write
User space
Kernel space
schedule
Sleep
schedule
CPU time


Non-Block
read/write
User space
Kernel space
CPU time

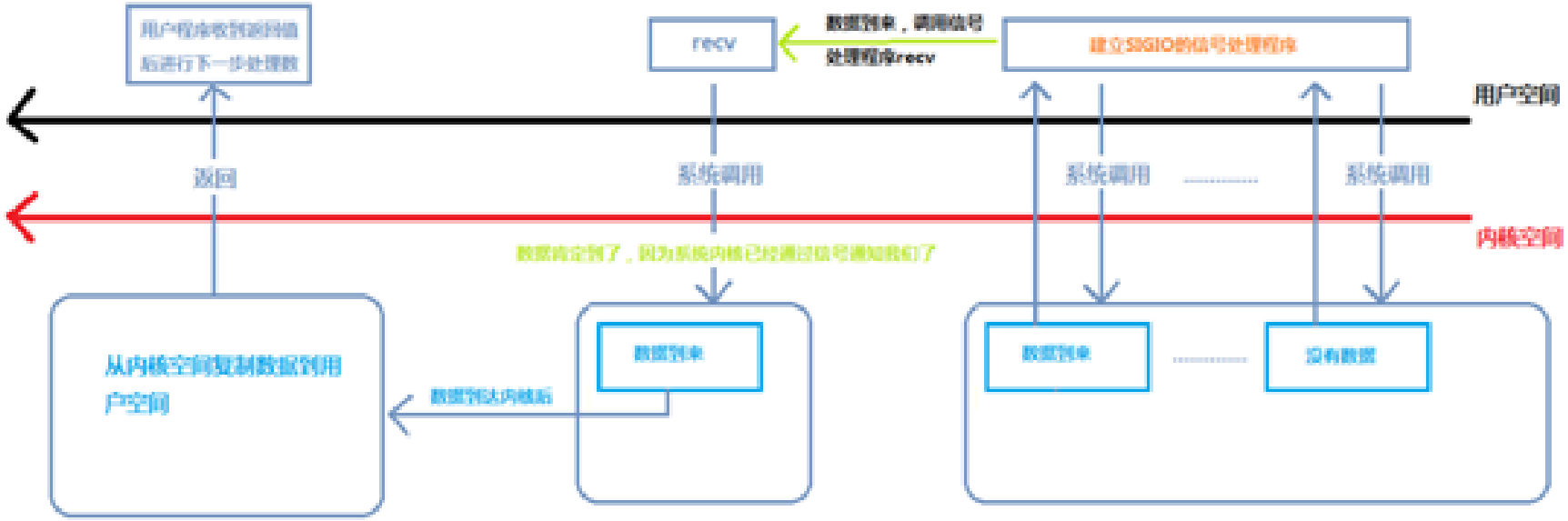
IO多路复用



这种模型其实和BIO是一模一样的，都是阻塞的，只不过在socket上加了一层代理select，select可以通过监控多个socket是否有数据，通过这种方式来提高性能。

一旦检测到一个或多个文件描述有数据到来，select函数就返回，这时再调用recv函数（这块也是阻塞的），数据从内核空间拷贝到用户空间，recv函数返回。

信号驱动IO



在用户态程序安装SIGIO信号处理函数（用sigaction函数或者signal函数来安装自定义的信号处理函数），即recv函数。然后用户态程序可以执行其他操作不会被阻塞。

一旦有数据到来，操作系统以信号的方式来通知用户态程序，用户态程序跳转到自定义的信号处理函数。

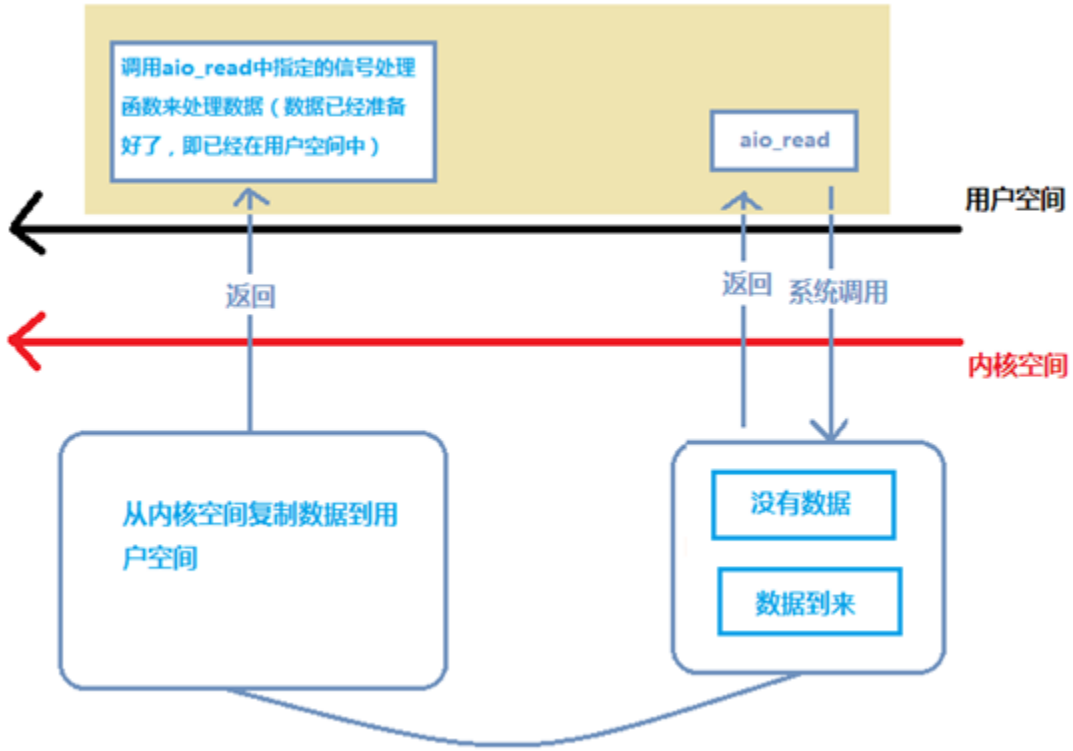
在信号处理函数中调用recv函数，接收数据。数据从内核空间拷贝到用户态空间后，recv函数返回。

公众号：一口Linux

异步IO

异步IO通过aio_read函数实现，aio_read提交请求，并递交一个用户态空间下的缓冲区。即使内核中没有数据到来，aio_read函数也立刻返回，应用程序就可以处理其他的事情。

当数据到来后，操作系统自动把数据从内核空间拷贝到aio_read函数递交的用户态缓冲区。拷贝完成以信号的方式通知用户态程序，用户态程序拿到数据后就可以执行后续操作。

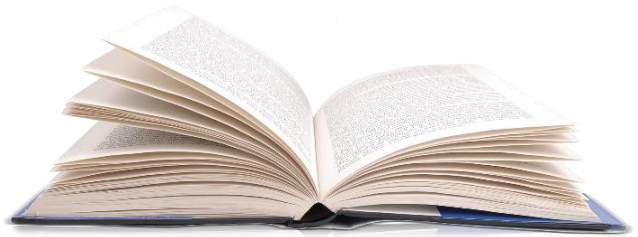


异步IO和信号驱动IO的不同？

- 在于信号通知用户态程序时数据所处的位置。异步IO已经把数据从内核空间拷贝到用户空间了；而信号驱动IO的数据还在内核空间，等着recv函数把数据拷贝到用户态空间。
- 异步IO主动把数据拷贝到用户态空间，不需要调用recv方法把数据从内核空间拉取到用户态空间。异步IO是一种推数据的机制，相比于信号处理IO拉数据的机制效率更高。
- 推数据是直接完成的，而拉数据是需要调用recv函数，调用函数会产生额外的开销，故效率低。

26

等待队列 wait queue



公众号：一口Linux

等待队列

- waitqueue (等待队列) 就是内核用于管理等待资源的进程，当某个进程获取的资源没有准备好的时候，可以通过调用 add_wait_queue() 函数把进程添加到 waitqueue 中，然后切换到其他进程继续执行。

- 当资源准备好，由资源提供方通过调用 wake_up() 函数来唤醒等待的进程。

公众号：一口Linux

定义

- 定义头文件：
- #include <linux/wait.h>

定义实例

```
wait_queue_head_t wq;

init_waitqueue_head(&wq);
```

公众号：一口Linux

阻塞接口

- wait_event(wq, condition)
- wait_event_timeout(wq, condition, timeout)
- wait_event_interruptible(wq, condition)
- 参数
- wq 等待队列
  condition 为条件表达式，当wake up后，condition为真时，唤醒阻塞的进程，为假时，继续睡眠
- 举例
  - havedata 0：没有数据 1：有数据，
  - havedata==1

公众号：一口Linux

解除阻塞接口（唤醒）

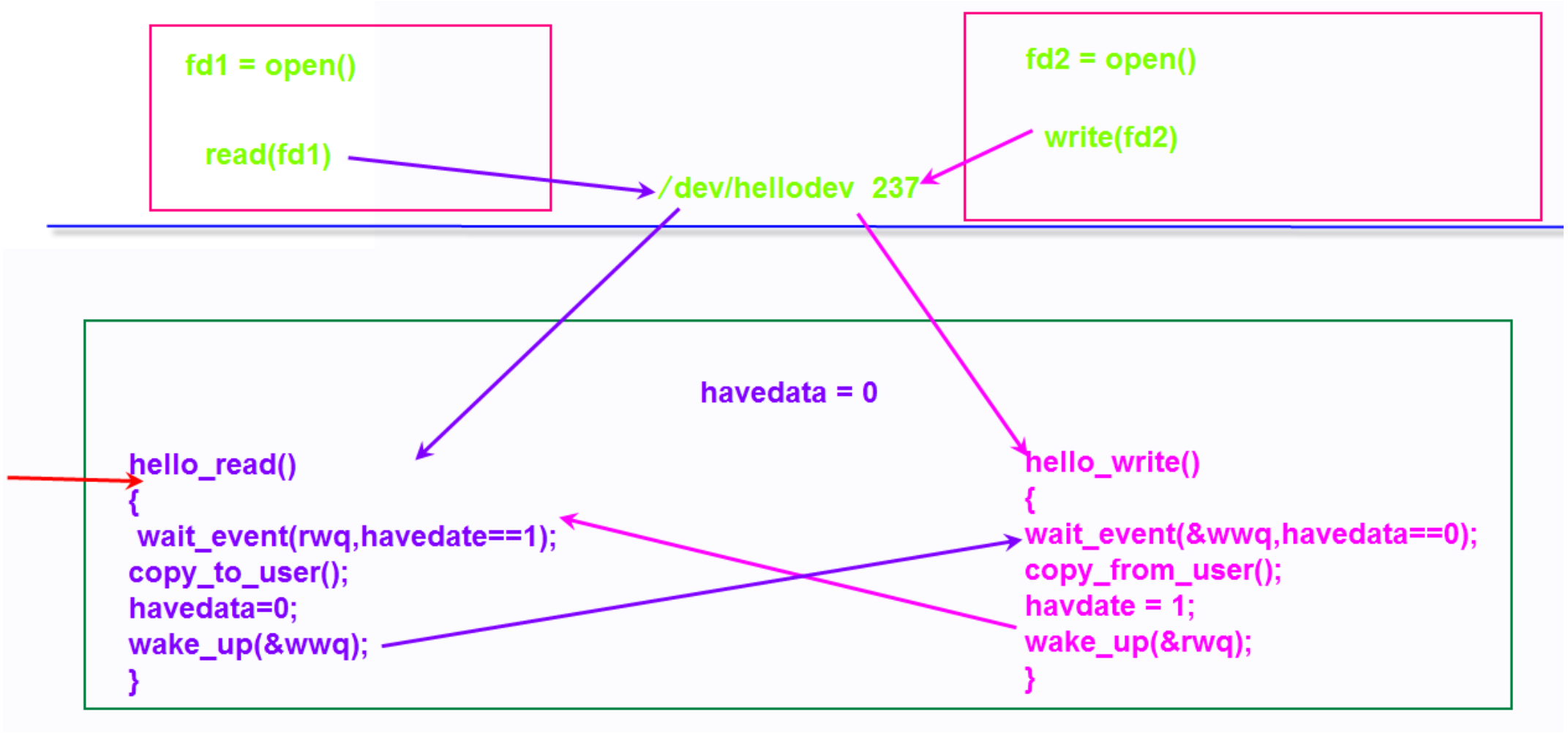
- #define wake_up(x)　　__wake_up(x, TASK_NORMAL, 1, NULL)

- #define wake_up_interruptible(x)　__wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)

实例

- 以字符设备为例，在没有数据的时候，在read函数中实现读阻塞，当向内核写入数据时，则唤醒阻塞在该等待队列的所有任务

- -画图


fd1 = open()
read(fd1)
/dev/hellodev 237
fd2 = open()
write(fd2)
havedata = 0
hello_read()
{
wait_event(rwq,havedate==1);
copy_to_user();
havedata=0;
wake_up(&wwq);
}
hello_write()
{
wait_event(&wwq,havedata==0);
copy_from_user();
havdate = 1;
wake_up(&rwq);
}

公众号：一口Linux

读操作

```
static ssize_t hello_read(struct file *filp,char __user *buf,size_t size,loff_t *poss)
{
    wait_event_interruptible(rwq,flage!=0);
    ..............
    flage=0;
    wake_up_interruptible(&wwq);
    return size;
}
```

公众号：一口Linux

写操作

```
static ssize_t hello_write(struct file *filp,const char __user *buf,size_t size,loff_t *poss)
{
    wait_event_interruptible(wwq,flage!=1);
    ..............
    flage=1;
    wake_up_interruptible(&rwq);
    return size;
}
```

公众号：一口Linux

如何支持非阻塞？

- 应用层

- fd=open("/dev/hello",O_RDONLY | O_NONBLOCK);

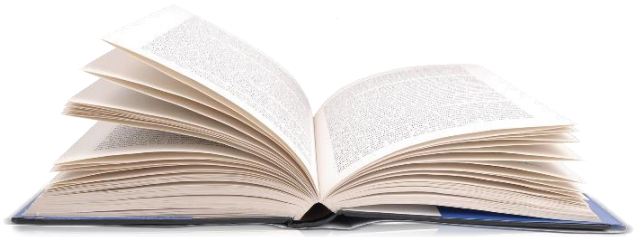
内核代码

```
static ssize_t hello_read(struct file *filp,char __user *buf,size_t size,loff_t *poss)
{
        int ret = 0;

        if(havedate==0)
        {
                if(filp->f_flags & O_NONBLOCK)
                {
                   return -EAGAIN;
                }
                wait_event_interruptible(rwq,flage!=0);
        }
      ..............
        flage=0;
        wake_up_interruptible(&wwq);
    return size;
}
```

公众号：一口Linux

27

字符设备poll方法实现



公众号：一口Linux

多路复用机制Select、poll、epoll

- I/O多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。

- select、poll和epoll是Linux API提供的I/O复用方式

API原型

- int **select** (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, **struct** timeval *timeout);

- **int poll** (struct pollfd ***fds**, **unsigned int** nfds, **int** timeout);

- int **epoll_create**(int size)；*//创建句柄，size：最大监听的数目*
- int **epoll_ctl**(int epfd, int op, int fd, struct epoll_event *event)；
- int **epoll_wait**(int epfd, struct epoll_event * events, int maxevents, int timeout);
-

1. select

- select系统调用是用来让我们的程序监视多个文件句柄的状态变化的。

- select 函数监视的文件描述符分3类，
  - writefds、readfds、和exceptfds。

- 调用后select函数会阻塞，直到有描述符就绪（有数据可读、可写、或者有except），或者超时（timeout指定等待时间，如果立即返回设为null即可），函数返回。

公众号：一口Linux

Select缺点

- select本质上是通过设置或者检查存放fd标志位的数据结构来进行下一步处理。这样所带来的缺点是：
- 1、 单个进程可监视的fd数量被限制，即能监听端口的大小有限。
- 2、 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低：
- 3、需要维护一个用来存放大量fd的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大

2. poll

- 和select函数一样，监听多个文件描述符，直到条件满足或超时的时候poll返回，通过遍历文件描述符来获取已经就绪的描述符。

- 不同于select使用三个位图来表示三个fdset的方式，poll使用一个 pollfd的指针实现。

- pollfd并没有最大数量限制（但是数量过大后性能也是会下降）

Poll缺点

- 缺点基本与select函数一致

- 1. poll返回后，通过遍历文件描述符来获取已经就绪的描述符。

- 2. 同时连接的大量客户端在一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其效率也会线性下降。

3. epoll

- epoll是在2.6内核中提出的，是select和poll的增强版本

- epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

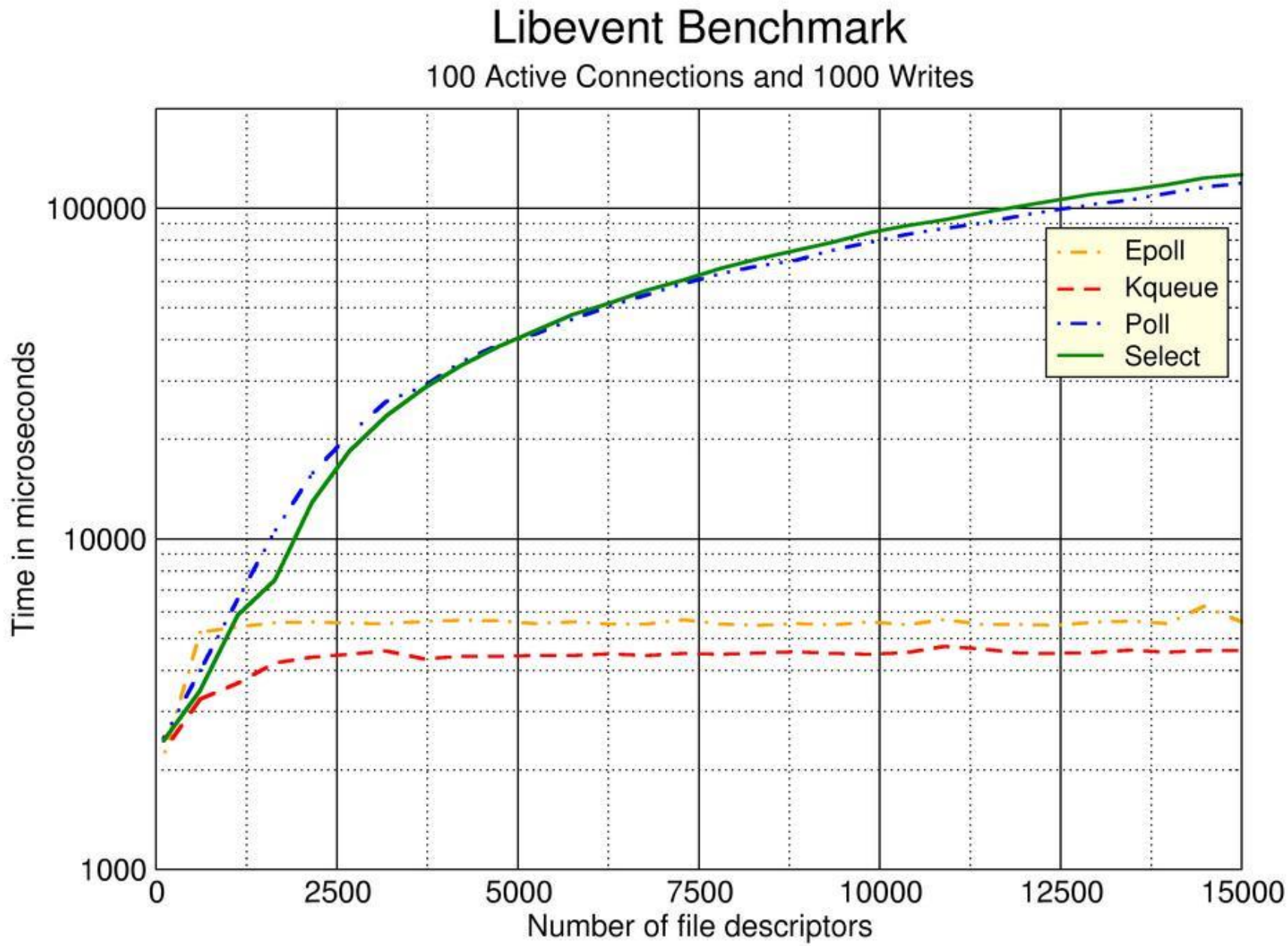
公众号：一口Linux

Epoll优点

- 1、监视的描述符数量不受限制
- 它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。
- 2、IO的效率不会随着监视fd的数量的增长而下降
- epoll不同于select和poll轮询的方式，而是通过每个fd定义的回调函数来实现的。只有就绪的fd才会执行回调函数。
- 3. epoll使用一个文件描述符管理多个描述符
- 将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

公众号：一口Linux

性能对比


Libevent Benchmark
100 Active Connections and 1000 Writes
Time in microseconds
100000
10000
1000
0
2500
5000
7500
10000
12500
15000
Number of file descriptors
Epoll
Kqueue
Poll
Select

.poll方法实现

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

返回值：event事件

事件类型

常量	说明
POLLIN	普通或优先级带数据可读
POLLRDNORM	普通数据可读
POLLRDBAND	优先级带数据可读
POLLPRI	高优先级数据可读
POLLOUT	普通数据可写
POLLWRNORM	普通数据可写
POLLWRBAND	优先级带数据可写
POLLERR	错误发生
POLLHUP	发生挂起
POLLNVAL	描写叙述字不是一个打开的文件

poll_wait

- static inline
- void **poll_wait**(struct file * filp,
  wait_queue_head_t * wait_address,
  poll_table *p)

```
typedef struct poll_table_struct {
        poll_queue_proc _qproc;
        unsigned long _key;
} poll_table;
```

公众号：一口Linux

poll()函数典型模板

```
static unsigned int xxx_poll(struct file *filp, poll_table *wait)
{
    unsigned int mask = 0;
    struct xxx_dev *dev = filp->private_data; /*获得设备结构体指针*/
     ...
    poll_wait(filp, &dev->r_wait, wait);//加读等待队列头
    poll_wait(filp, &dev->w_wait, wait);//加写等待队列头
    if (...)//可读
    {
        mask |= POLLIN | POLLRDNORM; /*表示数据可获得*/
    }
    if (...)//可写
    {
        mask |= POLLOUT | POLLWRNORM; /*标示数据可写入*/
      }
      ...
      return mask;
}
```

公众号：一口Linux

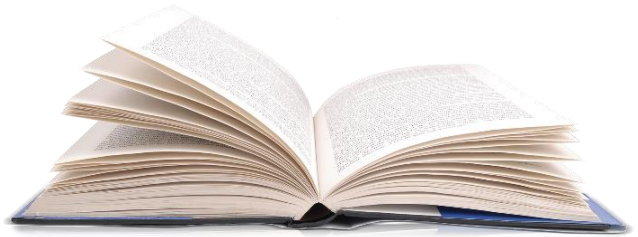
参考实例

```c
static unsigned int hello_poll (struct file *filep, struct poll_table_struct *ptable)
{
    unsigned int mask = 0;

    poll_wait(filep,&rq,ptable);
    poll_wait(filep,&wq,ptable);
    if(have_data == 1)
    {
        mask |= POLLIN;//可读
    }
    if(have_data == 0)
    {
        mask |= POLLOUT;//可写
    }
    return mask;
}
```

公众号：一口Linux

28

Linux 轮询编程Select



公众号：一口Linux

轮询编程-select

- int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
  - numfds的值是需要检查的号码最高的文件描述符加1。
  - readfds、writefds、exceptfds
    - 读、写和异常处理的文件描述符集合，
  - timeout参数是一个指向struct timeval类型的指针，它可以使select()在等待timeout时间后若没有文件描述符准备好则返回。
    - NULL 阻塞　　0：非阻塞　　非0 ： 超时时间

公众号：一口Linux

代码举例

- 演示
  - void FD_CLR(int fd, fd_set *set); 从集合set 删除fd
  - int FD_ISSET(int fd, fd_set *set); 判断fd是不是在集合中？
  - void FD_SET(int fd, fd_set *set); 将fd加入到set中
  - void FD_ZERO(fd_set *set); 清空set

  - #include <sys/select.h>
  - #include <sys/time.h>
  - #include <sys/types.h>
  - #include <unistd.h>

公众号：一口Linux

演示

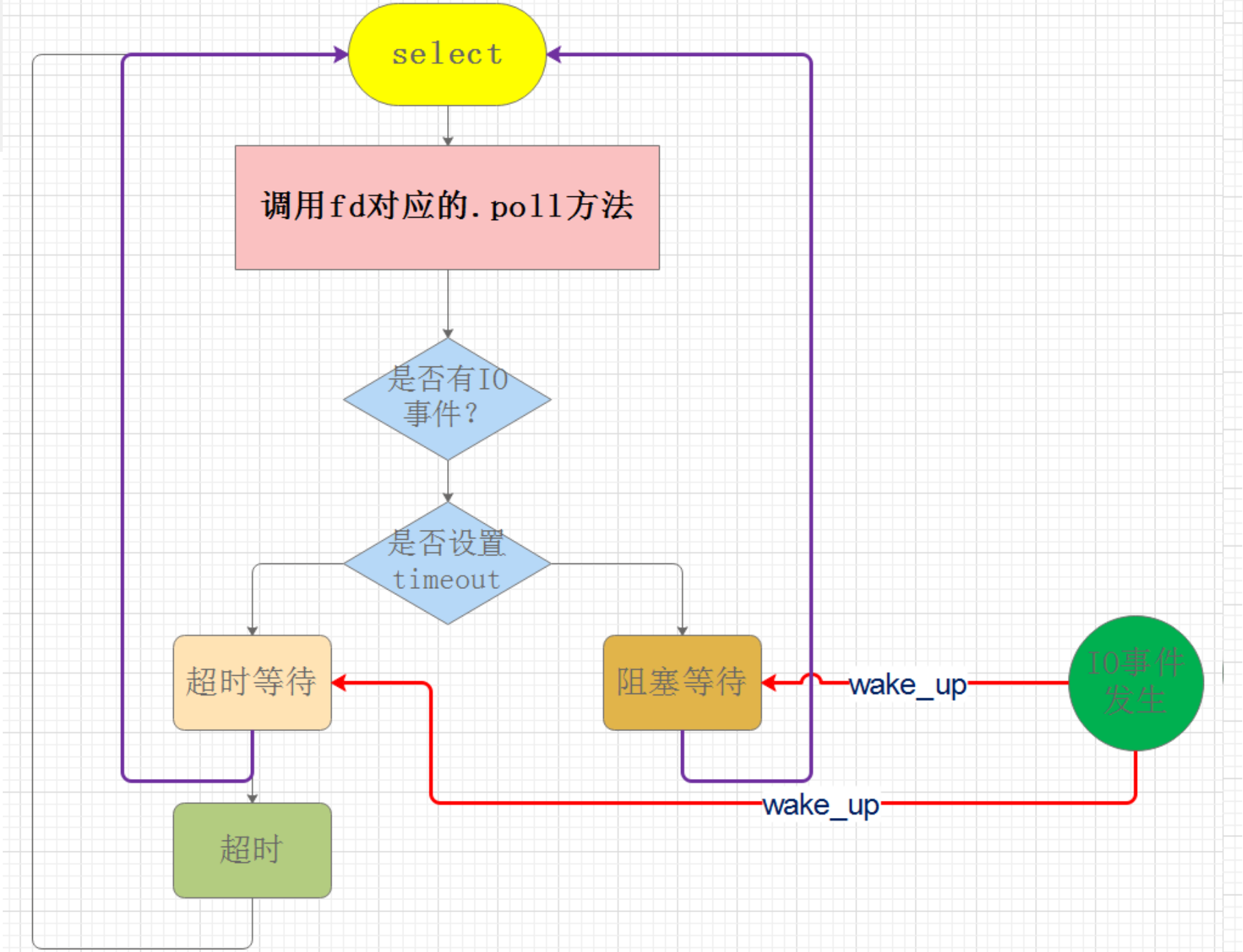
- 演示

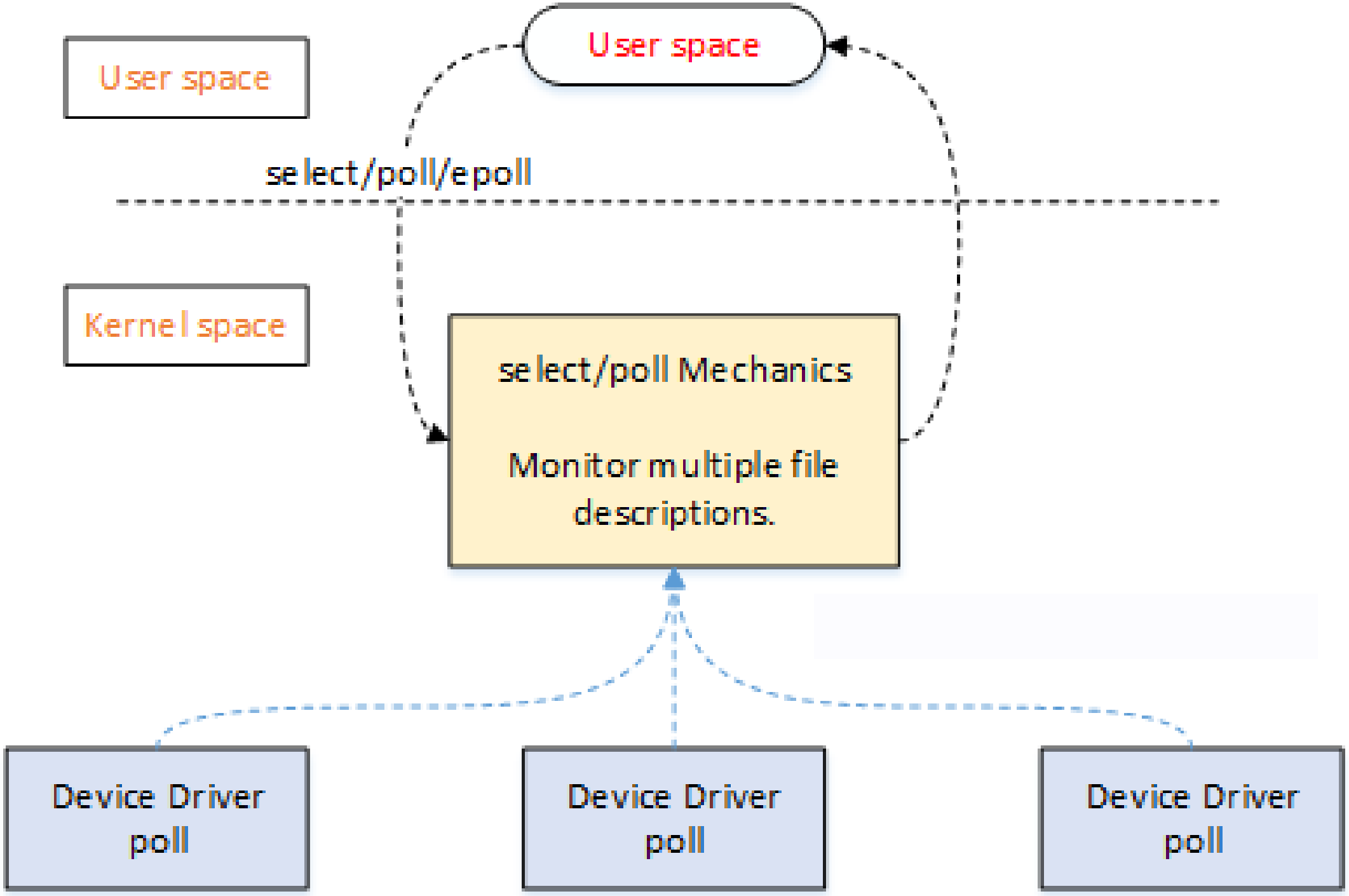
公众号：一口Linux

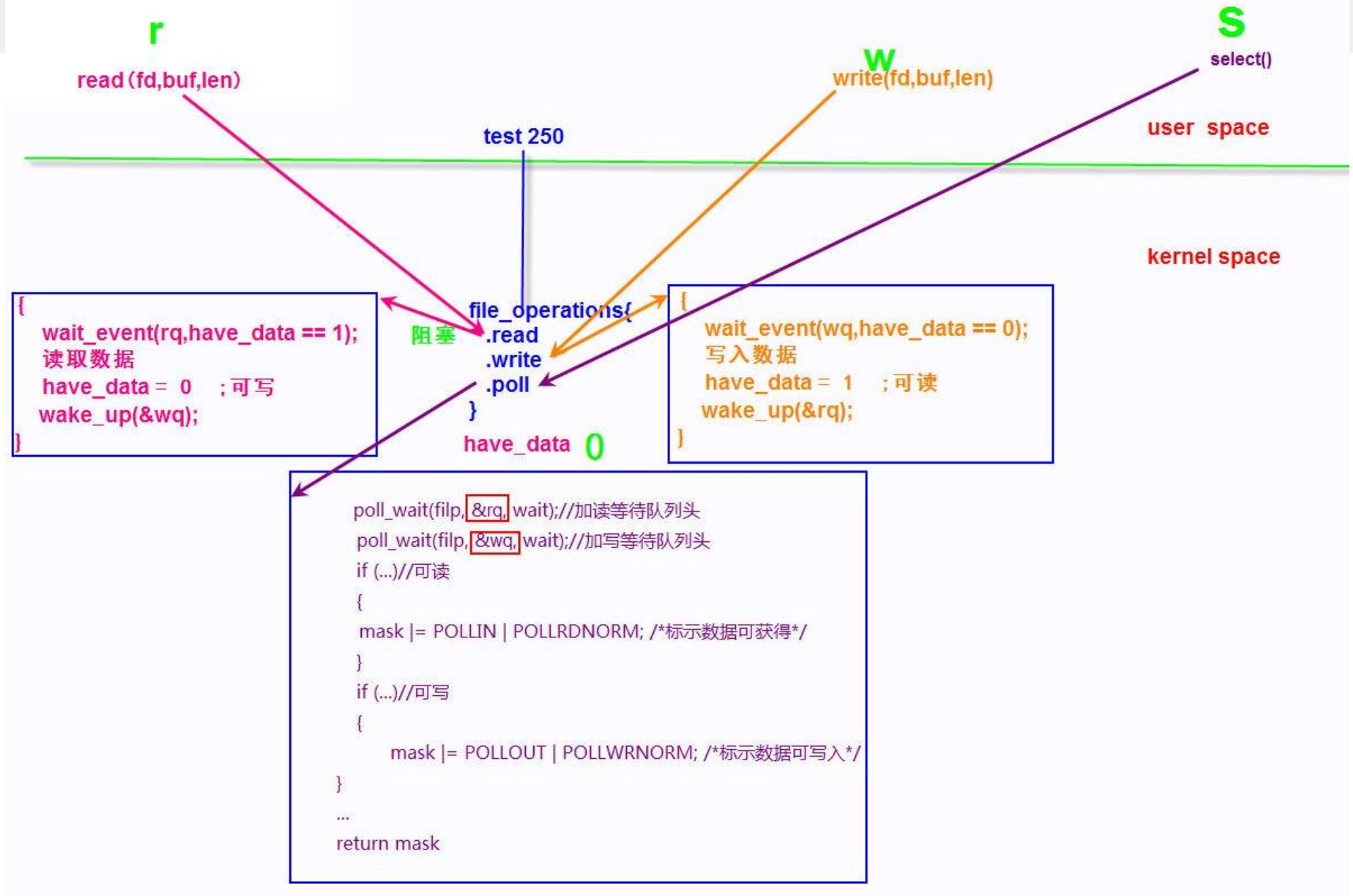
.poll操作流程

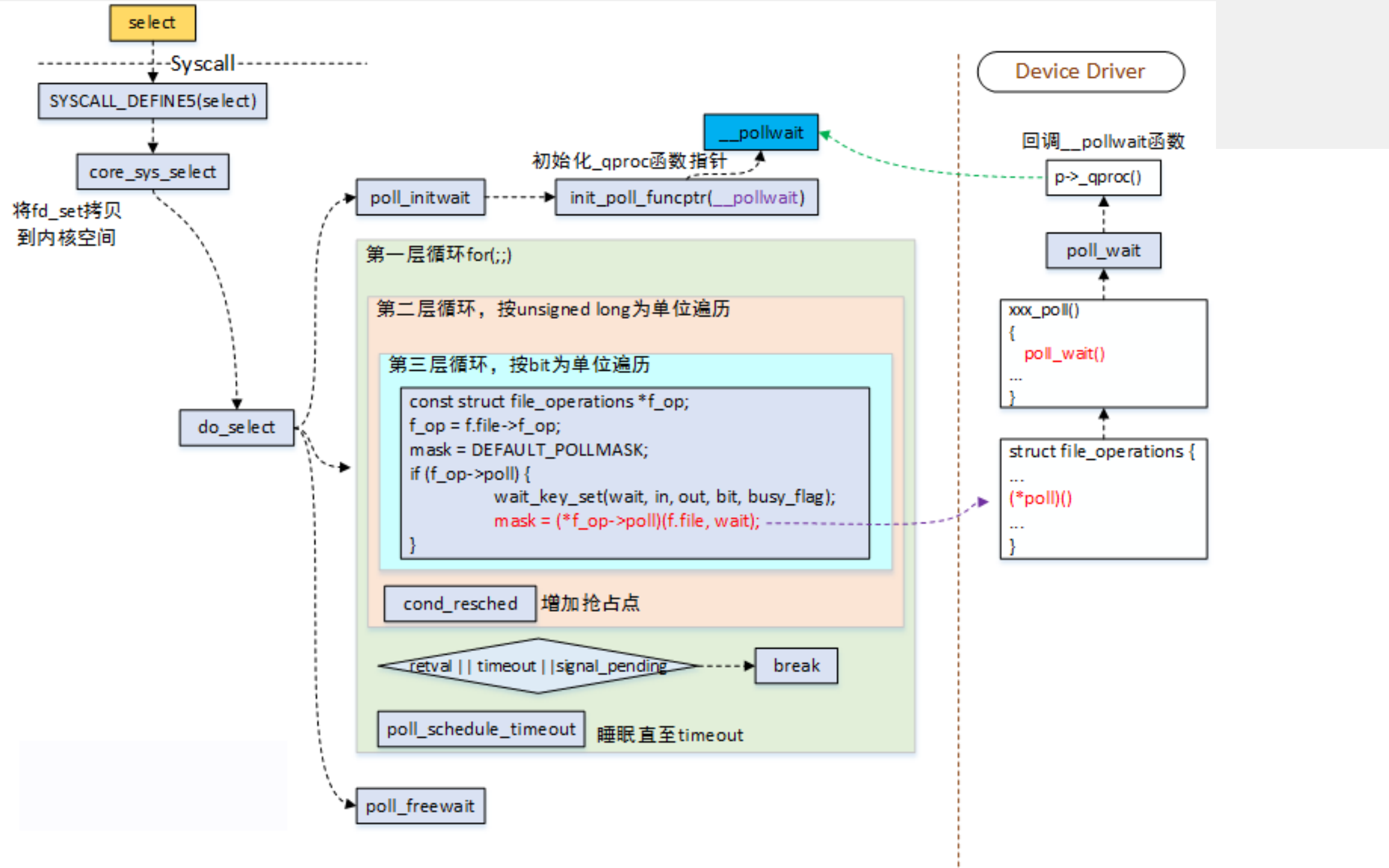
- 3种机制原理是相同的，流程如下
- 1. 先依次调用fd对应的struct file.f_op->poll()方法（如果有提供实现的话），尝试检查每个提供待检测IO的fd是否已经有IO事件就绪
- 2. 如果已经有IO事件就绪，则直接所收集到的IO事件返回，本次调用结束
- 3. 如果暂时没有IO事件就绪，则根据所给定的超时参数，选择性地进入等待
- 4. 如果超时参数指示不等待，则本次调用结束，无IO事件返回
- 5. 如果超时参数指示等待（等待一段时间或持续等待），则将当前select/poll/epoll的调用任务挂起
- 6. 当所检测的fd任何一个有新的IO事件发生时，会将上述的处于等待的任务唤醒。任务被唤醒之后，重新执行1中的IO事件收集过程，将此时收集到的IO事件返回，本次的调用过程结束。

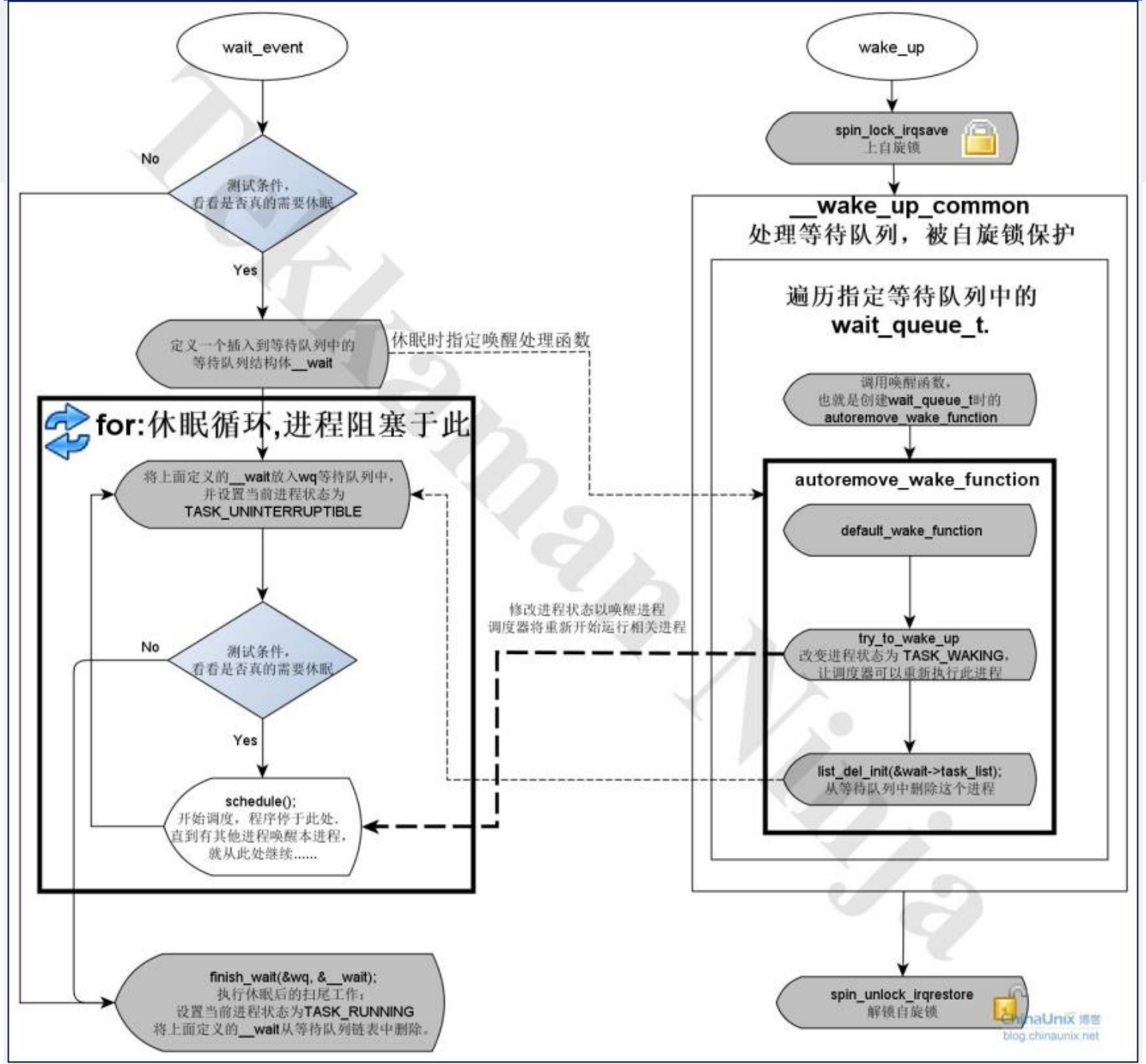
公众号：一口Linux

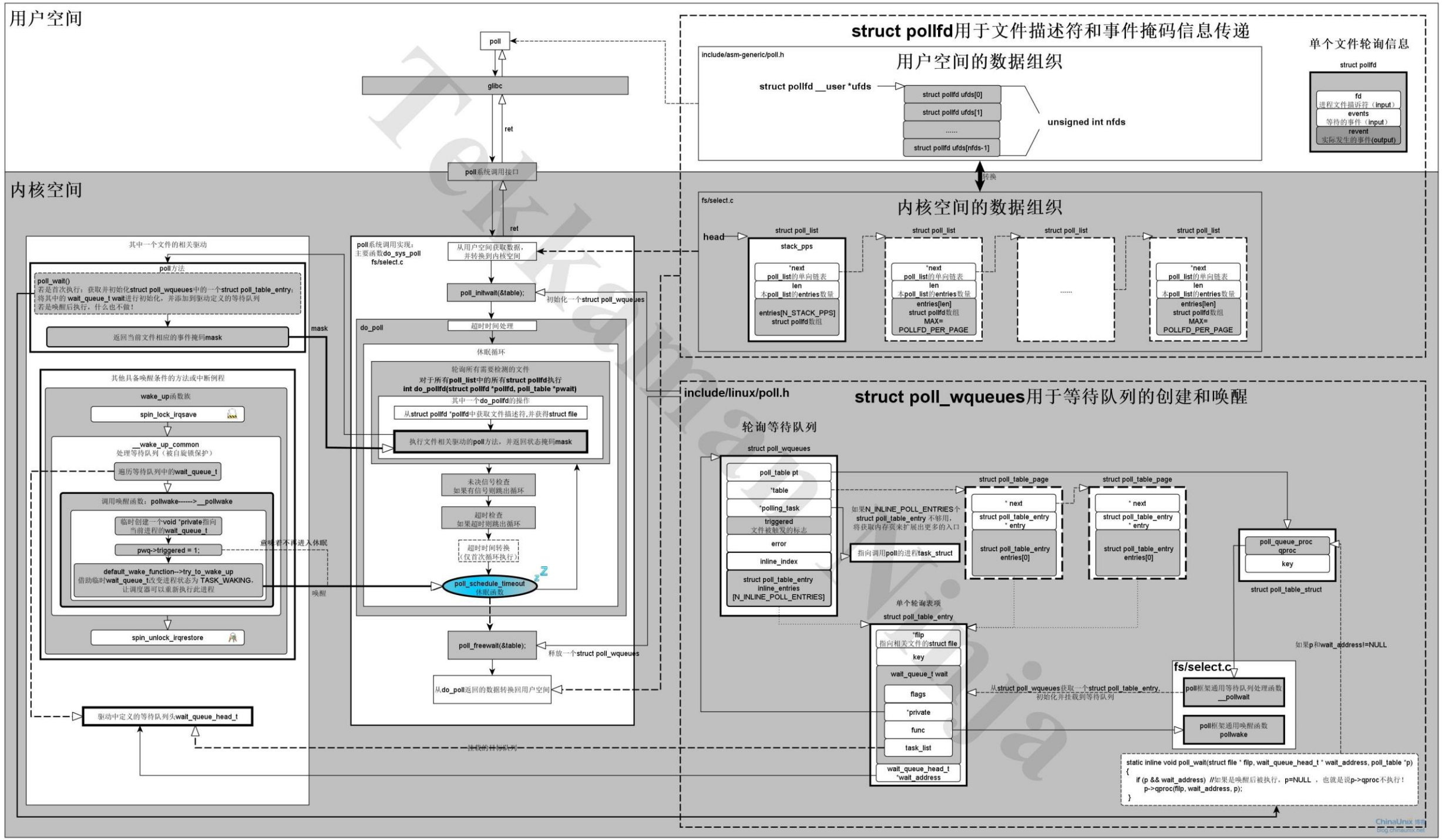

select
调用fd对应的.poll方法
是否有IO事件？
是否设置timeout
超时等待
阻塞等待
wake_up
IO事件发生
wake_up
超时


User space
User space
select/poll/epoll
Kernel space
select/poll Mechanics
Monitor multiple file descriptions.
Device Driver poll
Device Driver poll
Device Driver poll


r
read (fd,buf,len)
w
write(fd,buf,len)
s
select()
user space
test 250
kernel space
{
wait_event(rq,have_data == 1);
读取数据
have_data = 0 ;可写
wake_up(&wq);
}
阻塞
file_operations{
.read
.write
.poll
}
have_data 0
{
wait_event(wq,have_data == 0);
写入数据
have_data = 1 ;可读
wake_up(&rq);
}
poll_wait(filp, &rq, wait);//加读等待队列头
poll_wait(filp, &wq, wait);//加写等待队列头
if (...)//可读
{
mask |= POLLIN | POLLRDNORM; /*标示数据可获得*/
}
if (...)//可写
{
mask |= POLLOUT | POLLWRNORM; /*标示数据可写入*/
}
...
return mask

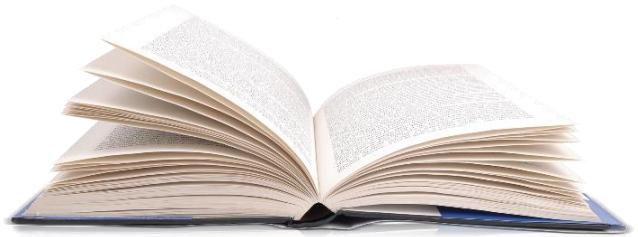

select
Syscall
SYSCALL_DEFINE5(select)
core_sys_select
将fd_set拷贝
到内核空间
do_select
poll_initwait
__pollwait
初始化_qproc函数指针
init_poll_funcptr(__pollwait)
第一层循环for(;;)
第二层循环，按unsigned long为单位遍历
第三层循环，按bit为单位遍历
const struct file_operations *f_op;
f_op = f.file->f_op;
mask = DEFAULT_POLLMASK;
if (f_op->poll) {
wait_key_set(wait, in, out, bit, busy_flag);
mask = (*f_op->poll)(f.file, wait);
}
cond_resched
增加抢占点
retval || timeout || signal_pending
break
poll_schedule_timeout
睡眠直至timeout
poll_freewait
Device Driver
回调__pollwait函数
p->_qproc()
poll_wait
xxx_poll()
{
poll_wait()
...
}
struct file_operations {
...
(*poll)()
...
}


wait_event
No
测试条件，
看看是否真的需要休眠
Yes
定义一个插入到等待队列中的
等待队列结构体__wait
休眠时指定唤醒处理函数
for:休眠循环,进程阻塞于此
将上面定义的__wait放入wq等待队列中，
并设置当前进程状态为
TASK_UNINTERRUPTIBLE
No
测试条件，
看看是否真的需要休眠
Yes
schedule();
开始调度，程序停于此处，
直到有其他进程唤醒本进程，
就从此处继续.....
finish_wait(&wq, &__wait);
执行休眠后的扫尾工作：
设置当前进程状态为TASK_RUNNING
将上面定义的__wait从等待队列链表中删除。
修改进程状态以唤醒进程
调度器将重新开始运行相关进程
wake_up
spin_lock_irqsave
上自旋锁
__wake_up_common
处理等待队列，被自旋锁保护
遍历指定等待队列中的
wait_queue_t.
调用唤醒函数，
也就是创建wait_queue_t时的
autoremove_wake_function
autoremove_wake_function
default_wake_function
try_to_wake_up
改变进程状态为 TASK_WAKING，
让调度器可以重新执行此进程
list_del_init(&wait->task_list);
从等待队列中删除这个进程
spin_unlock_irqrestore
解锁自旋锁
ChinaUnix 博客
blog.chinaunix.net


用户空间
内核空间
poll
glibc
ret
poll系统调用接口
ret
struct pollfd用于文件描述符和事件掩码信息传递
include/asm-generic/poll.h
用户空间的数据组织
struct pollfd __user *ufds
struct pollfd ufds[0]
struct pollfd ufds[1]
.....
struct pollfd ufds[nfds-1]
unsigned int nfds
单个文件轮询信息
struct pollfd
fd
进程文件描述符（input）
events
等待的事件（input）
revent
实际发生的事件(output)
转换
fs/select.c
内核空间的数据组织
head
struct poll_list
stack_pps
*next
poll_list的单向链表
len
本poll_list的entries数量
entries[N_STACK_PPS]
struct pollfd数组
entries[len]
struct pollfd数组
MAX=
POLLFD_PER_PAGE
其中一个文件的相关驱动
poll方法
poll_wait()
若是首次执行：获取并初始化struct poll_wqueues中的一个struct poll_table_entry：
将其中的 wait_queue_t wait进行初始化，并添加到驱动定义的等待队列
若是唤醒后执行，什么也不做！
返回当前文件相应的事件掩码mask
mask
其他具备唤醒条件的方法或中断例程
wake_up函数族
spin_lock_irqsave
__wake_up_common
处理等待队列（被自旋锁保护）
遍历等待队列中的wait_queue_t
调用唤醒函数：pollwake------>__pollwake
临时创建一个void *private指向
当前进程的wait_queue_t
pwq->triggered = 1;
default_wake_function-->try_to_wake_up
借助临时wait_queue_t改变进程状态为 TASK_WAKING，
让调度器可以重新执行此进程
spin_unlock_irqrestore
意味着不再进入休眠
唤醒
驱动中定义的等待队列头wait_queue_head_t
poll系统调用实现：
主要函数do_sys_poll
fs/select.c
从用户空间获取数据，
并转换到内核空间
poll_initwait(&table);
初始化一个struct poll_wqueues
do_poll
超时时间处理
休眠循环
轮询所有需要检测的文件
对于所有poll_list中的所有struct pollfd执行
int do_pollfd(struct pollfd *pollfd, poll_table *pwait)
其中一个do_pollfd的操作
从struct pollfd *pollfd中获取文件描述符,并获得struct file
执行文件相关驱动的poll方法，并返回状态掩码mask
未决信号检查
如果有信号则跳出循环
超时检查
如果超时则跳出循环
超时时间转换
（仅首次循环执行）
poll_schedule_timeout
休眠函数
poll_freewait(&table);
释放一个struct poll_wqueues
从do_poll返回的数据转换回用户空间
include/linux/poll.h
struct poll_wqueues用于等待队列的创建和唤醒
轮询等待队列
struct poll_wqueues
poll_table pt
*table
*polling_task
triggered
文件被触发的标志
error
inline_index
struct poll_table_entry
inline_entries
[N_INLINE_POLL_ENTRIES]
如果N_INLINE_POLL_ENTRIES个
struct poll_table_entry 不够用，
将获取内存页来扩展出更多的入口
指向调用poll的进程task_struct
struct poll_table_page
* next
struct poll_table_entry
* entry
struct poll_table_entry
entries[0]
struct poll_table_struct
poll_queue_proc
qproc
key
单个轮询表项
struct poll_table_entry
*filp
指向相关文件的struct file
key
wait_queue_t wait
flags
*private
func
task_list
wait_queue_head_t
*wait_address
从struct poll_wqueues获取一个struct poll_table_entry，
初始化并挂载到等待队列
如果p和wait_address!=NULL
fs/select.c
poll框架通用等待队列处理函数
__pollwait
poll框架通用唤醒函数
pollwake
static inline void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
{
if (p && wait_address) //如果是唤醒后被执行，p=NULL ， 也就是说p->qproc不执行！
p->qproc(filp, wait_address, p);
}
ChinaUnix博客
blog.chinaunix.net

29

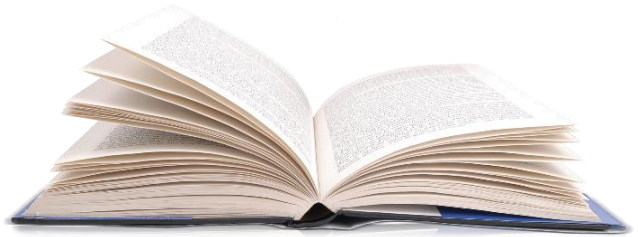
信号IO



公众号：一口Linux

30

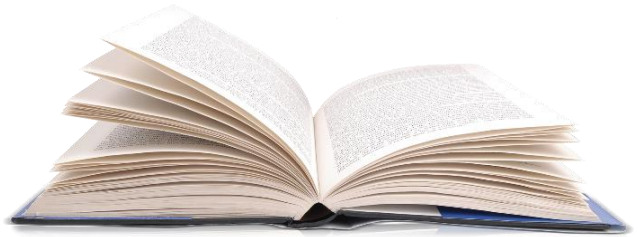
Platform总线



公众号：一口Linux

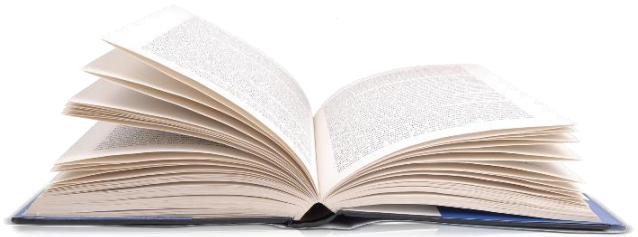
31

Platform进阶



公众号：一口Linux

32 内核定时器



公众号：一口Linux

- Source Insight

公众号：一口Linux