莲花九天落

Linux ioctl的实现 (2013-01-30 22:35:53)

阿释密达沙加

Qing 微博

加好友 发纸条

写留言 加关注

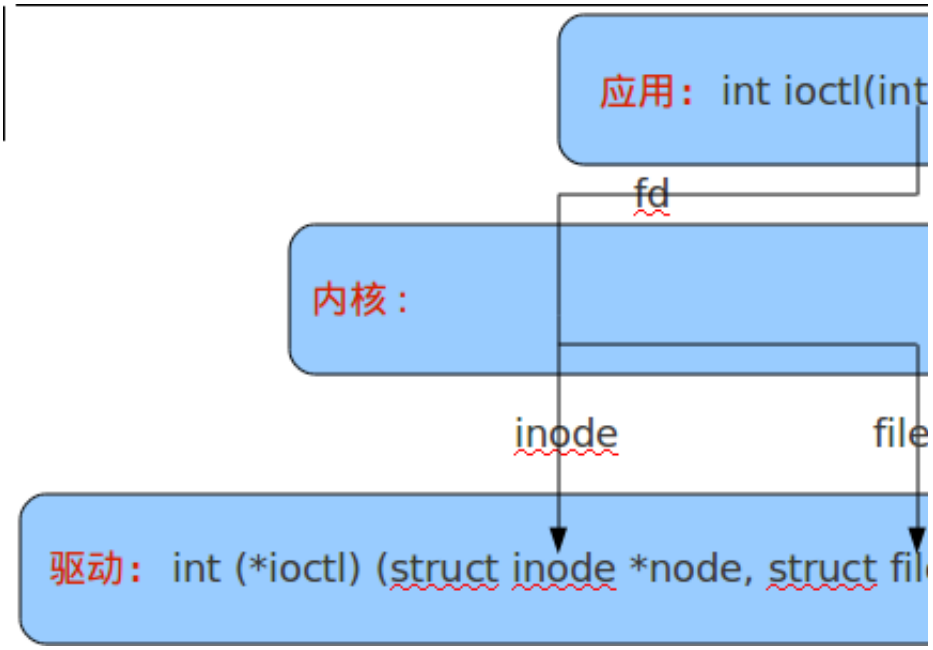
博客等级：10
博客积分：256
博客访问：9,377
关注人气：9
荣誉徽章：

一、ioctl的简介：

虽然在文件操作结构体"struct file_operations"中有很多对应的设备操作函作函数。如CD-ROM的驱动，想要一个弹出光驱的操作，这种操作并不作结构体也不会有对应的函数操作。

出于这样的原因，ioctl就有它的用处了————一些没办法归类的函数定的命令来实现对应的操作。所以，ioctl函数里面都实现了多个的对硬相应的操作。

来个图来说一下应用层与驱动函数的ioctl之间的联系：



上面的图可以看出，fd通过内核后找到对应的inode和file结构体指针并传
改(类型改了没什么关系)。

简单介绍一下函数：

int (*ioctl) (struct inode * node, struct file *filp, unsigned int cmd, unsig

参数：

1)inode和file：ioctl的操作有可能是要修改文件的属性，或者访问硬件。
文件属性的话，就要用到这两个结构体了，所以这里传来了它们的指针

2)cmd：命令，接下来要长篇大论地说。

3)arg：参数，接下来也要长篇大论。

返回值：

1)如果传入的非法命令，ioctl返回错误号-EINVAL。

2)内核中的驱动函数返回值都有一个默认的方法，只要是正数，内核就
传给应用层，如果是负值，内核就会认为它是错误号了。

loctl里面多个不同的命令，那就要看它函数的实现来决定返回值了。打
数，那返回值也就可以像read一样返回。

当然，不返回也是可以的。

二、ioctl的cmd

说白了，cmd就是一个数，如果应用层传来的数值在驱动中有对应的操

来个最简单的ioctl实现：3rd_char_4/1st

1)要先定义个命令，就用一个简单的0，来个命令的头文件，驱动和应用

```
1 #ifndef _TEST_CMD_H
2 #define _TEST_CMD_H
3
4 #define TEST_CLEAR 0
5
6 #endif
```

2)驱动实现ioctl：

命令TEST_CLEAR的操作就是清空驱动中的kbuf。

```
122 int test_ioctl (struct inode *node, struct file *filp, unsigned int cmd
123 {
124 int ret = 0;
125 struct _test_t *dev = filp->private_data;
126
127 switch(cmd){
128 case TEST_CLEAR:
129 memset(dev->kbuf, 0, DEV_SIZE);
130 dev->cur_size = 0;
131 filp->f_pos = 0;
132 ret = 0;
133 break;
134 default:
135 P_DEBUG("error cmd!\n");
136 ret = - EINVAL;
137 break;
138 }
139
140 return ret;
141 }
```

3)再来个应用程序：

```
1 #include
2 #include
3 #include
4 #include
5 #include
6 #include "test_cmd.h"
```

```
7
8 int main(void)
9 {
10 char buf[20];
11 int fd;
12 int ret;
13
14 fd = open("/dev/test", O_RDWR);
15 if(fd < 0)
16 {
17 perror("open");
18 return -1;
19 }
20
21 write(fd, "xiao bai", 10); //1先写入
22
23 ioctl(fd, TEST_CLEAR); //2再清空
24
25 ret = read(fd, buf, 10); //3再验证
26 if(ret < 0)
27 {
28 perror("read");
29 }
30
31 close(fd);
32 return 0;
33 }
```

注：这里为了read返回出错，我修改了驱动的read、write函数的开始时的判断，一看就知道了。

4)验证一下：

```
[root: 1st]# insmod test.ko
major[253] minor[0]
hello kernel
[root: 1st]# mknod /dev/test c 253 0
[root: 1st]# ./app
[test_write]write 10 bytes, cur_size:[10]
```

[test_write]kbuf is [xiao bai]

read: No such device or address //哈哈！出错了！因为没数据读取。

按照上面的方法来定义一个命令是完全可以的，但内核开发人员发现这
如果有两个不同的设备，但它们的ioctl的cmd却一样的，哪天有谁不小心
了。因为这个文件里面同样有cmd对应实现。
为了防止这样的事情发生，内核对cmd又有了新的定义，规定了cmd都应

三、ioctl中的cmd

一个cmd被分为了4个段，每一段都有各自的意义，cmd的定义在。注：
平台相关的，ARM的定义在，但这文件也是包含别的文件，千找万找，

在中，cmd拆分如下：

解释一下四部分，全部都在和ioctl-number.txt这两个文档有说明。
1)幻数：说得再好听的名字也只不过是个0~0xff的数，占8bit(_IOC_TYP
的，像设备号申请的时候一样，内核有一个文档给出一些推荐的或者已

164 'w' all CERN SCI driver
165 'y' 00-1F packet based user level communications
166
167 'z' 00-3F CAN bus card
168
169 'z' 40-7F CAN bus card
170
可以看到'x'是还没有人用的，我就拿这个当幻数！

2)序数：用这个数来给自己的命令编号，占8bit(_IOC_NRBITS)，我的程

3)数据传输方向：占2bit(_IOC_DIRBITS)。如果涉及到要传参，内核要
应用层的角度来描述的。
1)_IOC_NONE：值为0，无数据传输。
2)_IOC_READ：值为1，从设备驱动读取数据。
3)_IOC_WRITE：值为2，往设备驱动写入数据。
4)_IOC_READ|_IOC_WRITE：双向数据传输。

4)数据大小：与体系结构相关，ARM下占14bit(_IOC_SIZEBITS)，如果
是sizeof(int)。

强调一下，内核是要求按这样的方法把cmd分类，当然你也可以不这样
的程序看上去很正宗。上面我的程序没按要求照样运行。

既然内核这样定义cmd，就肯定有方法让用户方便定义：
_IO(type,nr) //没有参数的命令
_IOR(type,nr,size) //该命令是从驱动读取数据
_IOW(type,nr,size) //该命令是从驱动写入数据
_IOWR(type,nr,size) //双向数据传输
上面的命令已经定义了方向，我们要传的是幻数(type)、序号(nr)和大小
类型，如int，上面的命令就会帮你检测类型的正确然后赋值sizeof(int)。

有生成cmd的命令就必有拆分cmd的命令：
_IOC_DIR(cmd) //从命令中提取方向
_IOC_TYPE(cmd) //从命令中提取幻数
_IOC_NR(cmd) //从命令中提取序数
_IOC_SIZE(cmd) //从命令中提取数据大小

越讲就越复杂了，既然讲到这，随便就讲一下预定义命令。
预定义命令是由内核来识别并且实现相应的操作，换句话说，一旦你使
动程序能够收到，因为内核拿掉就把它处理掉了。

分为三类：
1)可用于任何文件的命令
2)只用于普通文件的命令
3)特定文件系统类型的命令

其实上面的我三类我也没搞懂，反正我自己随便编了几个数当命令都没
已经使用的幻数就行了。

讲了这么多，终于要上程序了，修改一下上一个程序，让它看起来比较
/3rd_char/3rd_char_4/2nd
1)先改一下命令：

```
1 #ifndef _TEST_CMD_H
2 #define _TEST_CMD_H
```

```
3
4 #define TEST_MAGIC 'x' //定义幻数
5 #define TEST_MAX_NR 1 //定义命令的最大序数，只有一个命令当然
6
7 #define TEST_CLEAR _IO(TEST_MAGIC, 0)
8
9 #endif
```

2)既然这么辛苦改了cmd，在驱动函数当然要做一些参数检验：

```
122 int test_ioctl (struct inode *node, struct file *filp, unsigned int cmd
123 {
124 int ret = 0;
125 struct _test_t *dev = filp->private_data;
126
127
128 if(_IOC_TYPE(cmd) != TEST_MAGIC) return - EINVAL;
129 if(_IOC_NR(cmd) > TEST_MAX_NR) return - EINVAL;
130
131 switch(cmd){
132 case TEST_CLEAR:
133 memset(dev->kbuf, 0, DEV_SIZE);
134 dev->cur_size = 0;
135 filp->f_pos = 0;
136 ret = 0;
137 break;
138 default:
139 P_DEBUG("error cmd!\n");
140 ret = - EINVAL;
141 break;
142 }
143
144 return ret;
145 }
```

每个参数的传入都会先检验一下幻数还有序数是否正确。

3)应用程序的验证：

结果跟上一个完全一样，因为命令的操作没有修改

```
[root: 2nd]# insmod test.ko
major[253] minor[0]
hello kernel
[root: 2nd]# mknod /dev/test c 253 0
[root: 2nd]# ./app
[test_write]write 10 bytes, cur_size:[10]
[test_write]kbuf is [xiao bai]
read: No such device or address
```

五、ioctl中的arg之整数传参。

上面讲的例子都没有使用ioctl的传参。这里先要说一下ioctl传参的方式。

应用层的ioctl的第三个参数是"..."，这个跟printf的"..."可不一样，printf中
而ioctl最多也只能传一个，"..."的意思是让内核不要检查这个参数的类型
数，只要你传入的个数是1.

一般会有两种的传参方法：

1)整数，那可是省力又省心，直接使用就可以了。

2)指针，通过指针的就传什么类型都可以了，当然用起来就比较烦。

先说简单的，使用整数作为参数：

例子，实现个命令，通过传入参数更改偏移量，虽然llseek已经实现，这

1)先加个命令：

```
1 #ifndef _TEST_CMD_H
2 #define _TEST_CMD_H
3
4 #define TEST_MAGIC 'x' //定义幻数
5 #define TEST_MAX_NR 2 //定义命令的最大序数
6
7 #define TEST_CLEAR _IO(TEST_MAGIC, 1)
8 #define TEST_OFFSET _IO(TEST_MAGIC, 2)
9
10 #endif
```

这里有人会问了，明明你是要传入参数，为什么不用_IOW而用_IO定义

原因有二：

1)因为定义数据的传输方向是为了好让驱动的函数验证数据的安全性，会恶意传参(回想一下copy_to_user)。

2)个人喜好，方便我写程序介绍另一种传参方法，说白了命令也只是一了。

2)更新test_ioctl

```
122 int test_ioctl (struct inode *node, struct file *filp, unsigned int cmd
123 {
124 int ret = 0;
125 struct _test_t *dev = filp->private_data;
126
127
128 if(_IOC_TYPE(cmd) != TEST_MAGIC) return - EINVAL;
129 if(_IOC_NR(cmd) > TEST_MAX_NR) return - EINVAL;
130
131 switch(cmd){
132 case TEST_CLEAR:
133 memset(dev->kbuf, 0, DEV_SIZE);
134 dev->cur_size = 0;
135 filp->f_pos = 0;
136 ret = 0;
137 break;
138 case TEST_OFFSET: //根据传入的参数更改偏移量
139 filp->f_pos += (int)arg;
140 P_DEBUG("change offset!\n");
141 ret = 0;
142 break;
143 default:
144 P_DEBUG("error cmd!\n");
145 ret = - EINVAL;
146 break;
147 }
148
149 return ret;
150 }
```

TSET_OFFSET命令就是根据传参更改偏移量，不过这里要注意一个问知道从应用传来的参数是什么类型，不然就没法使用。在这个函数里，

也得用int。

3)再改一下应用程序：

```
1 #include
2 #include
3 #include
4 #include
5 #include
6
7 #include "test_cmd.h"
8
9 int main(void)
10 {
11 char buf[20];
12 int fd;
13 int ret;
14
15 fd = open("/dev/test", O_RDWR);
16 if(fd < 0)
17 {
18 perror("open");
19 return -1;
20 }
21
22 write(fd, "xiao bai", 10); //先写入
23
24 ioctl(fd, TEST_OFFSET, -10); //再改偏移量
25
26 ret = read(fd, buf, 10); //再读数据
27 printf(" buf is [%s]\n", buf);
28 if(ret < 0)
29 {
30 perror("read");
31 }
32
33 close(fd);
34 return 0;
```

```
35 }
```

4)验证一下

```
[root: 3rd]# insmod test.ko
major[253] minor[0]
hello kernel
[root: 3rd]# mknod /dev/test c 253 0
[root: 3rd]# ./app
[test_write]write 10 bytes, cur_size:[10]
[test_write]kbuf is [xiao bai]
[test_ioctl]change offset! //更改偏移量
[test_read]read 10 bytes, cur_size:[0] //没错误，成功读取！
 buf is [xiao bai]
```

上面的传参很简单把，接下来说一下以指针传参。
考虑到参数不可能永远只是一个正数这么简单，如果要传多一点的东西

六、ioctl中的arg之指针传参。

一讲到从应用程序传来的指针，就得想起我邪恶的传入了非法指针的例道的指针，都得先检验指针的安全性。

说到这检验又有两种方法：
1)用的时候才检验。
2)一进来ioctl就检验。

先说用的时候检验，说白了就是用copy_xx_user系列函数，下面实现一

1)先定义个命令

```
1 #ifndef _TEST_CMD_H
2 #define _TEST_CMD_H
3
4 struct ioctl_data{
5 unsigned int size;
6 char buf[100];
7 };
8
9 #define DEV_SIZE 100
```

```
10
11 #define TEST_MAGIC 'x' //定义幻数
12 #define TEST_MAX_NR 3 //定义命令的最大序数
13
14 #define TEST_CLEAR _IO(TEST_MAGIC, 1)
15 #define TEST_OFFSET _IO(TEST_MAGIC, 2)
16 #define TEST_KBUF _IO(TEST_MAGIC, 3)
17
18 #endif
```

这里有定义多了一个函数，虽然这个命令是涉及到了指针的传参，但我用上。

该命令的操作是传进一个结构体指针，驱动根据结构体的内容修改kbuf

2)来个实现函数：

```
122 int test_ioctl (struct inode *node, struct file *filp, unsigned int cmd
123 {
124 int ret = 0;
125 struct _test_t *dev = filp->private_data;
126 struct ioctl_data val;
127
128
129 if(_IOC_TYPE(cmd) != TEST_MAGIC) return - EINVAL;
130 if(_IOC_NR(cmd) > TEST_MAX_NR) return - EINVAL;
131
132 switch(cmd){
133 case TEST_CLEAR:
134 memset(dev->kbuf, 0, DEV_SIZE);
135 dev->cur_size = 0;
136 filp->f_pos = 0;
137 ret = 0;
138 break;
139 case TEST_OFFSET: //根据传入的参数更改偏移量
140 filp->f_pos += (int)arg;
141 P_DEBUG("change offset!\n");
142 ret = 0;
143 break;
144 case TEST_KBUF: //修改kbuf
```

```
145 if(copy_from_user(&val, (struct ioctl_data *)arg, sizeof(struct ioctl
146 ret = - EFAULT;
147 goto RET;
148 }
149 memset(dev->kbuf, 0, DEV_SIZE);
150 memcpy(dev->kbuf, val.buf, val.size);
151 dev->cur_size = val.size;
152 filp->f_pos = 0;
153 ret = 0;
154 break;
155 default:
156 P_DEBUG("error cmd!\n");
157 ret = - EINVAL;
158 break;
159 }
160
161 RET:
162 return ret;
163 }
```

第145行，因为指针是从用户程序传来，所以必须检查安全性。

3)来个应用程序

```
9 int main(void)
10 {
11 char buf[20];
12 int fd;
13 int ret;
14
15 struct ioctl_data my_data= {
16 .size = 10,
17 .buf = "123456789"
18 };
19
20 fd = open("/dev/test", O_RDWR);
21 if(fd < 0)
22 {
23 perror("open");
```

```
24 return -1;
25 }
26
27 write(fd, "xiao bai", 10);
28
29 ioctl(fd, TEST_KBUF, &my_data);
30
31 ret = read(fd, buf, 10);
32 printf(" buf is [%s]\n", buf);
33 if(ret < 0)
34 {
35 perror("read");
36 }
37
38 close(fd);
39 return 0;
40 }
```

4)再来验证一下：

```
[root: 4th]# ./app
[test_write]write 10 bytes, cur_size:[10]
[test_write]kbuf is [xiao bai]
[test_read]read 10 bytes, cur_size:[0]
 buf is [123456789] //成功!
```

注：类似copy_xx_user的函数含有put_user、get_user等，我就不细说了。

下面说第二种方法：进入ioctl后使用access_ok检测。

声明一下：下面的验证方法是不正确的。如果不想看下去的话，今天的

先说一下access_ok的使用

access_ok(type, addr, size)

使用：检测地址的安全性

参数：

type：用于指定数据传输的方向，VERIFY_READ表示要读取应用层数据

数据。注意：这里和IOR IOW的方向相反。如果既读取又写入，那就使

addr：用户空间的地址

size：数据的大小

返回值：

成功返回1，失败返回0。

既然知道怎么用，就直接来程序了：

1)定义命令

```
1 #ifndef _TEST_CMD_H
2 #define _TEST_CMD_H
3
4 struct ioctl_data{
5 unsigned int size;
6 char buf[100];
7 };
8
9 #define DEV_SIZE 100
10
11 #define TEST_MAGIC 'x' //定义幻数
12 #define TEST_MAX_NR 3 //定义命令的最大序数
13
14 #define TEST_CLEAR _IO(TEST_MAGIC, 1)
15 #define TEST_OFFSET _IO(TEST_MAGIC, 2)
16 #define TEST_KBUF _IOW(TEST_MAGIC, 3, struct ioctl_data)
17
18 #endif
```

这里终于要用_IOW了！

2)实现ioctl

```
122 int test_ioctl (struct inode *node, struct file *filp, unsigned int cmd
123 {
124 int ret = 0;
125 struct _test_t *dev = filp->private_data;
126
127
128 if(_IOC_TYPE(cmd) != TEST_MAGIC) return - EINVAL;
129 if(_IOC_NR(cmd) > TEST_MAX_NR) return - EINVAL;
130
131 if(_IOC_DIR(cmd) & _IOC_READ)
132 ret = access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE
133 else if(_IOC_DIR(cmd) & _IOC_WRITE)
```

```
134 ret = access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(
135 if(!ret) return - EFAULT;
136
137 switch(cmd){
138 case TEST_CLEAR:
139 memset(dev->kbuf, 0, DEV_SIZE);
140 dev->cur_size = 0;
141 filp->f_pos = 0;
142 ret = 0;
143 break;
144 case TEST_OFFSET: //根据传入的参数更改偏移量
145 filp->f_pos += (int)arg;
146 P_DEBUG("change offset!\n");
147 ret = 0;
148 break;
149 case TEST_KBUF: //修改kbuf
150 memset(dev->kbuf, 0, DEV_SIZE);
151 memcpy(dev->kbuf, ((struct ioctl_data *)arg)->buf,
152 ((struct ioctl_data *)arg)->size);
153 dev->cur_size = ((struct ioctl_data *)arg)->size;
154 filp->f_pos = 0;
155 ret = 0;
156 break;
157 default:
158 P_DEBUG("error cmd!\n");
159 ret = - EINVAL;
160 break;
161 }
162
163 return ret;
164 }
```

上面并没有用copy_to_user，而是通过access_ok来检测。

3)再来个应用程序：

```
9 int main(void)
10 {
11 char buf[20];
```

```
12 int fd;
13 int ret;
14
15 struct ioctl_data my_data= {
16 .size = 10,
17 .buf = "123456789"
18 };
19
20 fd = open("/dev/test", O_RDWR);
21 if(fd < 0)
22 {
23 perror("open");
24 return -1;
25 }
26
27 write(fd, "xiao bai", 10);
28
29 ret = ioctl(fd, TEST_KBUF, &my_data);
30 if(ret < 0)
31 {
32 perror("ioctl");
33 }
34
35 ret = read(fd, buf, 10);
36 printf(" buf is [%s]\n", buf);
37 if(ret < 0)
38 {
39 perror("read");
40 }
41
42 close(fd);
43 return 0;
44 }
```

4)验证一下：效果和上一个一样

```
[root: 5th]# ./app
[test_write]write 10 bytes, cur_size:[10]
```

[test_write]kbuf is [xiao bai]

[test_read]read 10 bytes, cur_size:[0]

buf is [123456789]

下面就要如正题了，这个驱动是有问题的，那就是验证安全性完全不起输出，不信可以自己传个邪恶地址(void *)0进去试一下。

修改应用程序一样代码：

29 ret = ioctl(fd, TEST_KBUF, &my_data);

上面是我做的错误实现，我本来想验证，只要经过access_ok检验，数据后照样会出错。

但是，copy_to_user同样是先调用access_ok再调用memcpy，它却没出错知道了麻烦指点一下。

我查了设备驱动第三版，在144页有这样的说法：

1.access_ok并没有做完的所有的内存检查，

2.大多数的驱动代码都不是用access_ok的，后面的内存管理会讲述。

在这里书本上有这样的约定：（都是我自己的理解）

1.传入指针需要检查安全性。memcpy函数尽量不要在内核中使用。

2.copy_to_user.copy_from_user.get_user.put_user函数会再拷贝数据access_ok。

3.如果在ioctl函数开头使用了accsee_ok检验数据，接下来的代码可以使的函数（书上有例子）

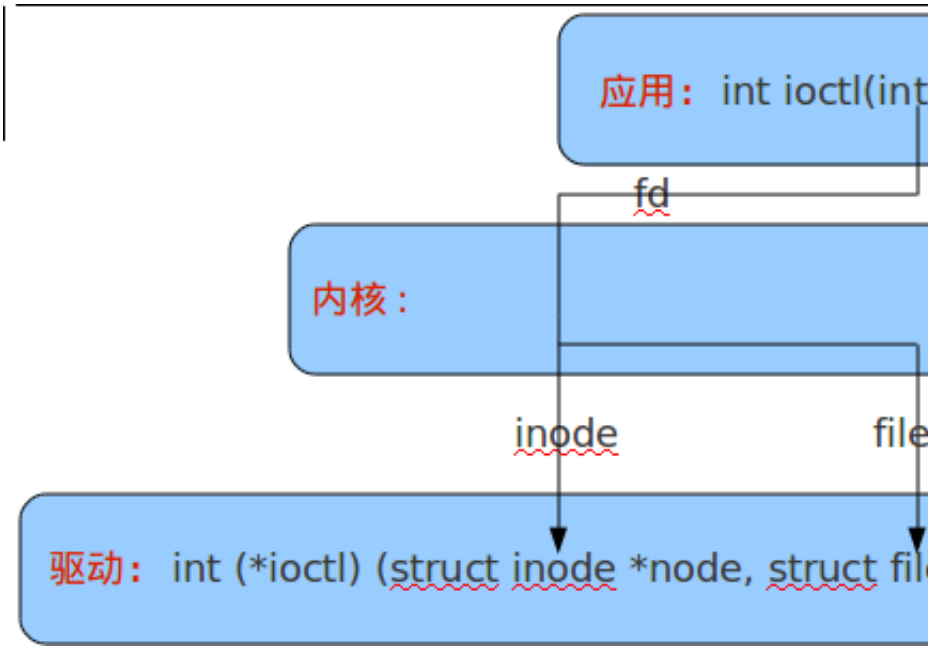
虽然还有写东西还没搞懂，但个人觉得，如果使用个access_ok要这么麻用copy_xx_user函数，省力又省心。

七、总结：

这次讲了ioctl的实现：

1)命令是怎么定义。

2)参数怎么传递。



0

喜欢

分享：

前一篇：poll 和 select
后一篇：[转载]io阻塞与io非阻塞之

登录名： 密码： 找回密码 注册

分享到微博 评论并转载此博文

验证码： 请点击后输入验证码 收听验证码

发评论

以上网友发言只代表其个人观点，不代表新浪网的

< 前一篇

poll 和 select