netlink 实现分析

（1）网络 file 对象

当网络双方的通信线路建立好之后，双发就可以开始互相传递数据，可以使用 write()和 read()传递数据。可见网络通信也同样使用了文件系统的架构，这里的 file 对象是比较特殊的，结构如图 1 所示：

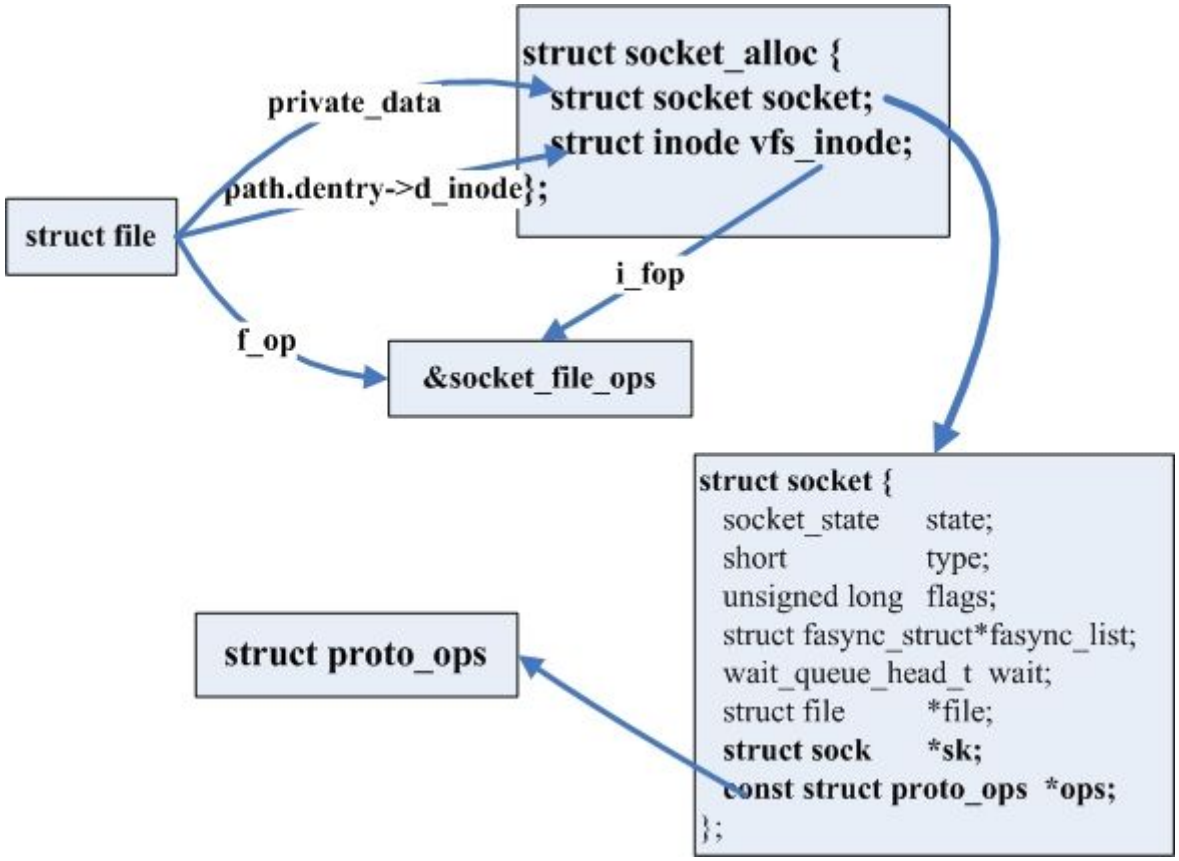

图 1 网络 file 对象结构

其中 static const struct file_operations **socket_file_ops** = {

```
    .owner = THIS_MODULE,
    .llseek = no_llseek,
    .aio_read =   sock_aio_read,对应 read()
    .aio_write =  sock_aio_write,对应 write()
    .poll =       sock_poll,
    .unlocked_ioctl = sock_ioctl,
    .mmap =       sock_mmap,
    .open =       sock_no_open,    /* special open code to disallow open via /proc */
    .release =    sock_close,
    .fasync =     sock_fasync,
    .sendpage =   sock_sendpage,
    .splice_write = generic_splice_sendpage,
    .splice_read =    sock_splice_read,
};
```

说明：如果 **file->f_op->write** 为空，则会调用 **file->f_op-> aio_write;**

file->f_op->read 为空，则会调用 **file->f_op-> aio_read**

从 **sock_aio_write()**开始的函数调用流程如下：

```
sock_aio_write()→
    do_sock_write()→
        __sock_sendmsg()→
            sock->ops->sendmsg(iocb, sock, msg, size);
```

从 **sock_aio_read()**开始的函数调用流程如下：

```
sock_aio_read()→
    do_sock_read()→
        __sock_recvmsg()→
            __sock_recvmsg_nosec()→
                sock->ops->recvmsg(iocb, sock, msg, size, flags);
```

可见发送和接受过程最终都是调用操作函数集 **sock->ops**，其结构如下：

```
struct proto_ops {
    int         family;
    struct module     *owner;
    int         (*release) (struct socket *sock);
    int         (*bind)   (struct socket *sock, struct sockaddr *myaddr, int sockaddr_len);
```

```
    int     (*connect)(struct socket *sock, struct sockaddr *vaddr,int sockaddr_len, int flags);
    int     (*socketpair)(struct socket *sock1,struct socket *sock2);
    int     (*accept)    (struct socket *sock,struct socket *newsock, int flags);
    int     (*getname)   (struct socket *sock, struct sockaddr *addr, int *sockaddr_len, int peer);
    unsigned int (*poll)         (struct file *file, struct socket *sock, struct poll_table_struct *wait);
    int     (*ioctl)     (struct socket *sock, unsigned int cmd, unsigned long arg);
    int     (*compat_ioctl) (struct socket *sock, unsigned int cmd, unsigned long arg);
    int     (*listen)    (struct socket *sock, int len);
    int     (*shutdown)  (struct socket *sock, int flags);
    int(*setsockopt)(struct socket *sock, int level,int optname, char __user *optval, unsigned int optlen);
    int(*getsockopt)(struct socket *sock, int level,int optname, char __user *optval, int __user *optlen);
    int(*compat_setsockopt)(struct socket *sock, int level,int optname, char *optval, unsigned int optlen);
    int(*compat_getsockopt)(struct socket *sock, int level,int optname, char *optval, int *optlen);
    int     (*sendmsg)(struct kiocb *iocb, struct socket *sock, struct msghdr *m, size_t total_len);
    int(*recvmsg)(struct kiocb *iocb, struct socket *sock,struct msghdr *m, size_t total_len, int flags);
    int     (*mmap)(struct file *file, struct socket *sock, struct vm_area_struct * vma);
    ssize_t  (*sendpage) (struct socket *sock, struct page *page, int offset, size_t size, int flags);
    ssize_t(*splice_read)(struct socket *sock,loff_t *ppos,struct pipe_inode_info*pipe,size_t len, unsigned
    int flags);
};
```

应用程序的 bind(),listen(),accept(),connect(),read(),write(),close()都最终调用 struct proto_ops 内的函数。

（2）netlink 网络 file 对象

当网络应用程序调用 socket()时，将会建立如图 1 所示的 file 结构。

例如：socket(AF_NETLINK, SOCK_RAW,NETLINK_GENERIC);

函数调用流程如下：

```
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)→
    sys_socket() [SYSCALL_DEFINE3(socket, int, family, int, type, int, protocol)] →
        sock_create()→
            __sock_create()→
                pf = rcu_dereference(net_families[family])→
                err = pf->create(net, sock, protocol, kern);→
        sock_map_fd()
```

在 static int __init netlink_proto_init(void)中可以看到 netlink 协议的注册函数：

sock_register(&netlink_family_ops);

其中 static const struct net_proto_family netlink_family_ops = {

.family = PF_NETLINK,

.create = netlink_create,

.owner = THIS_MODULE, /* for consistency 8) */

};

现在可以知道上边的 pf->create(net, sock, protocol, kern);是调用 netlink_create()

生成的 file 对象结构如图 2 所示：

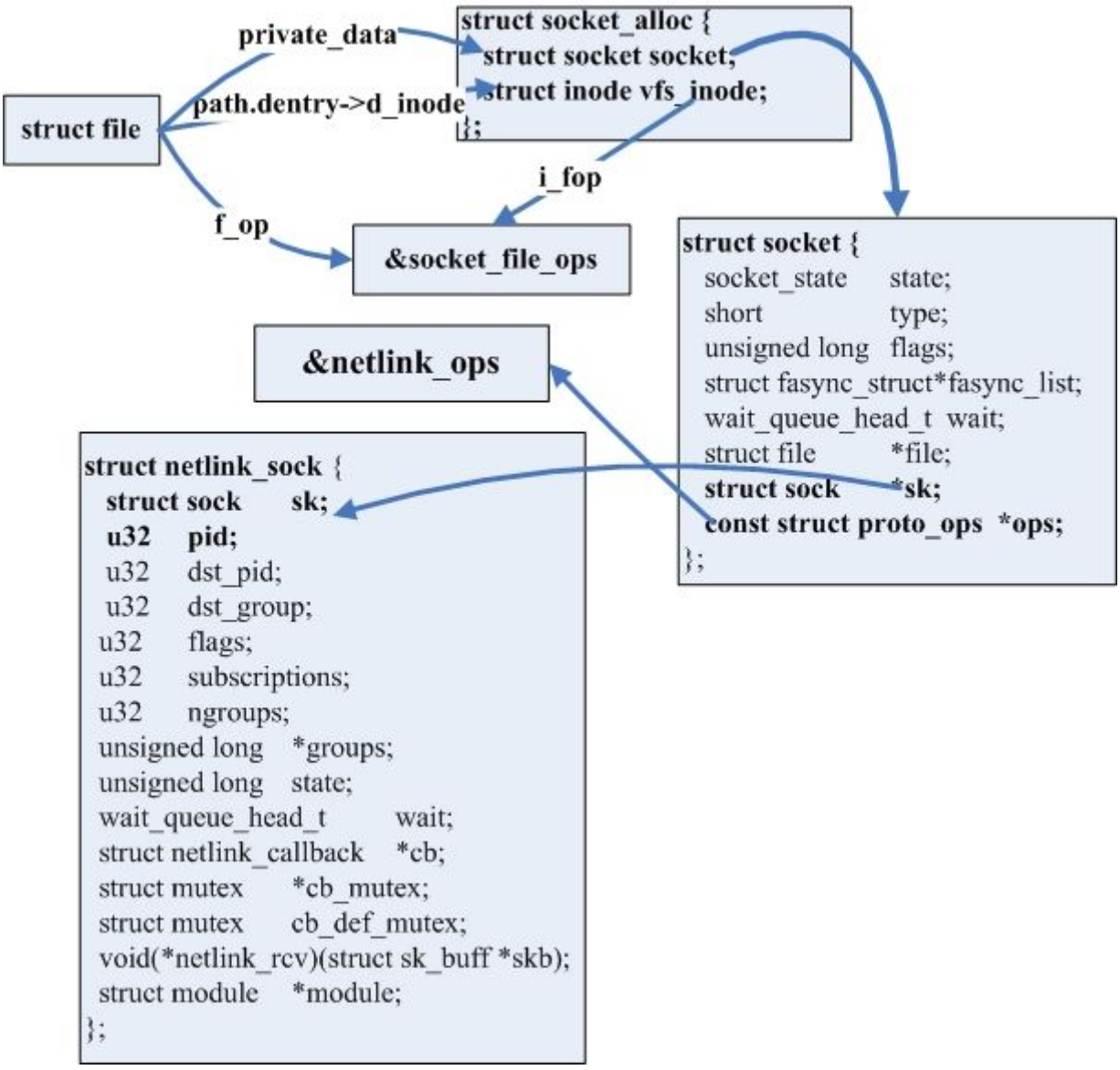

图 2 netlink 的 file 对象结构

其中 static const struct proto_ops **netlink_ops** = {

```
    .family =PF_NETLINK,
    .owner = THIS_MODULE,
    .release =      netlink_release,
    .bind =         netlink_bind,
    .connect =      netlink_connect,
    .socketpair = sock_no_socketpair,
    .accept =       sock_no_accept,
    .getname =      netlink_getname,
    .poll =         datagram_poll,
    .ioctl =    sock_no_ioctl,
    .listen = sock_no_listen,
    .shutdown =  sock_no_shutdown,
    .setsockopt = netlink_setsockopt,
    .getsockopt = netlink_getsockopt,
    .sendmsg =    netlink_sendmsg,
    .recvmsg =    netlink_recvmsg,
    .mmap =       sock_no_mmap,
    .sendpage =   sock_no_sendpage,
};
```

struct sock 对象是包含在 **netlink_sock** 中。

（3）netlink 消息接收端

一个 **netlink** 消息接收端口必须先在内核中注册后，其它内核路径才可能将消息发送给这个接收端。内核中 **netlink** 所有的接收端都必须在 **nl_tables** 中注册，如图 3 所示：

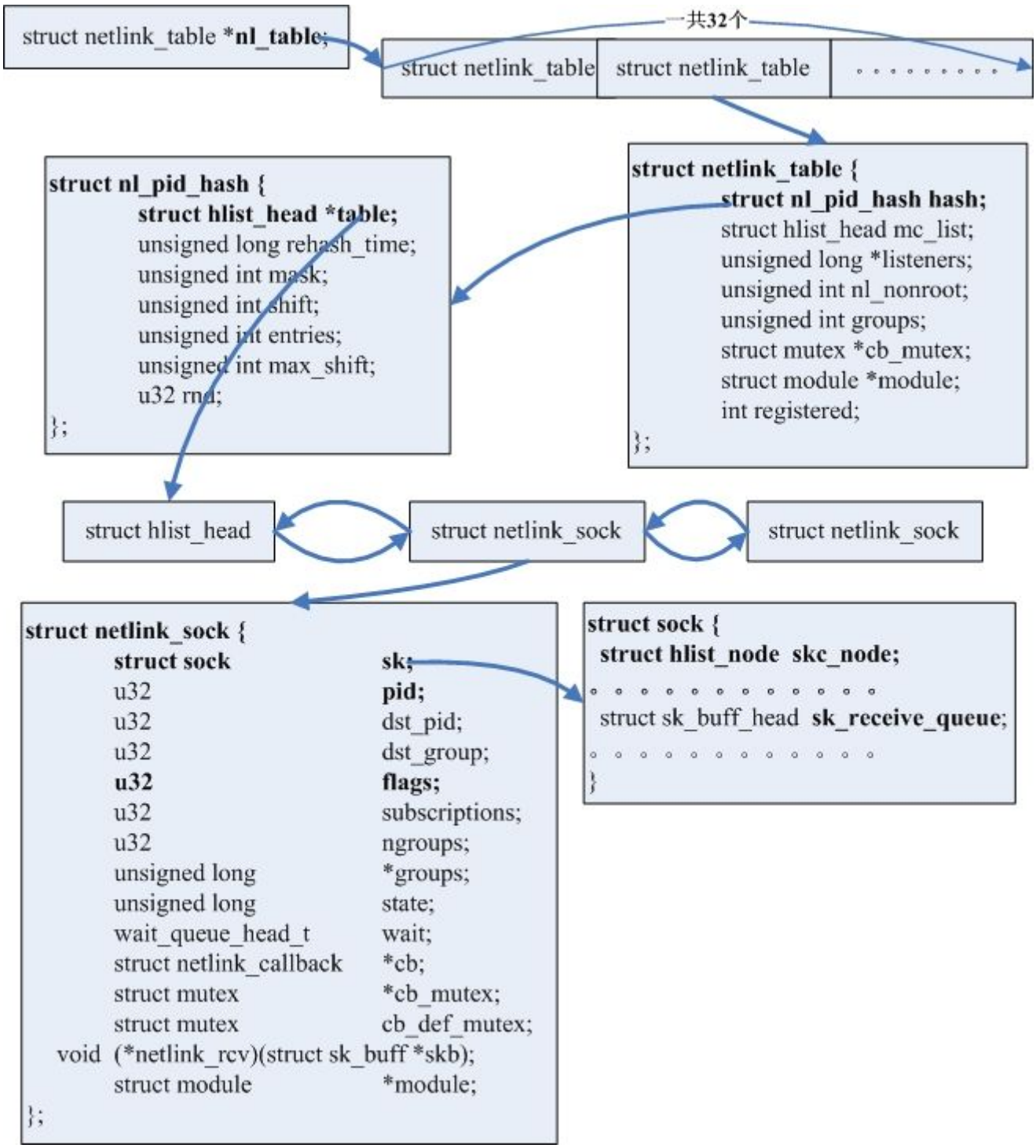

图 4 netlink 消息接收端

其中 struct netlink_sock 表示一个 netlink 接收端，每个 netlink_sock 必定包含在某个 netlink_table 中，netlink_table 表示一种 netlink 协议。

有 2 个字段来定位一个 netlink 接收端：1. netlink 协议类型 2. netlink 套接字 pid。发送端根据这两个字段定位一个 netlink 套接字，并把消息加入到接收端的 sk_receive_queue 队列中（图 4 所示）。

（3.1）内核路径注册 netlink 接收端

内核使用 struct sock *netlink_kernel_create(struct net *net, int unit, unsigned int groups,
void (*input)(struct sk_buff *skb),
struct mutex *cb_mutex, struct module *module)

创建并注册一个 netlink 接收端。

其中 input 为接收函数的地址，unit 为协议类型，由于是内核创建的所以 pid=0.

（3.2）用户进程注册 netlink 接收端

如果用户空间进程需要接收 netlink 消息，也必须创建一个 struct netlink_sock 结构，创建的过程要比内核的复杂一些，函数调用流程如下：

用户空间调用 bind()→陷入内核态

netlink_bind()→

netlink_insert()

其中 pid=进程号。

现在其它进程或者内核路径通过 netlink 协议类型和 netlink 套接字 pid 就可以找到这个 netlink 接收端，并把消息加入接收端的 sk_receive_queue 队列。

发送消息最终都是调用 netlink_broadcast()或者 netlink_unicast()。netlink 接收端接收消息也是从这两个函数开始。

（3.3）内核 netlink 接收端接收消息

由内核创建的 netlink 接收端(pid=0)接收过程如下：

netlink_unicast()→

netlink_getsockbypid()→

netlink_unicast_kernel()→

nlk->netlink_rcv(skb);

其中 nlk->netlink_rcv 指向 netlink_kernel_create()中的 input。

在发送函数内就会直接调用注册的接收函数处理消息。

（3.4）用户进程 netlink 接收端接收消息

用户进程创建的 netlink 接收端(pid > 0)，接收过程分为 2 个部分。

第一部分：其它内核路径将 netlink 消息数据包加入 sk_receive_queue 队列。

netlink_unicast()→

netlink_sendskb()→

skb_queue_tail(&sk->sk_receive_queue, skb);

第二部分：用户空间调用 read 接收数据（也可以调用其它函数比如 recvmsg()）。

read()→陷入内核态

netlink_recvmsg()→

skb_recv_datagram()→

__skb_recv_datagram()→

skb = skb_peek(&sk->sk_receive_queue);

用户空间通过调用接收函数来获取 netlink 消息。

（4）通信效率分析

现在可以看到：

1. 在内核中给内核注册的 netlink 接收端发送消息效率是比较高，因为不需要进行数据的一次拷贝，直接把发送端生成的 skb 作为参数，接收端直接处理这个 skb。

2. 但是用户空间和内核的通信效率就比较低了，因为需要进行数据的一次拷贝。用户空间使用 struct msghdr 结构存储消息，内核空间使用 skb 存储消息，所以必须进行消息类型的转换，而且还需要进行消息数据的拷贝。

用户空间和内核的通信必须进行数据的拷贝原因有 2 点：（在 misc 设备实现机制中也有讲解）

1. 用户空间使用的线性地址是在前 3 个 G 的空间，而内核使用的线性地址是在第 4 个 G，所以用户空间没法使用内核中的线性地址。

2. 当前运行进程是不断替换的，内核是使用当前运行进程的页表，虽然不同进程的页表的内核空间部分是相同的，但是用户空间部分是不同的，如果内核要长期使用某个用户空间的数据，将会出现问题(同样的线性地址，在不同的进程上下文中将会引用不同的物理地址)。

（5）实现零拷贝的两个理论方案

理论上是可以实现 netlink 数据的零拷贝，有两个方案：

1. 使某个进程的状态总是处于内核态。正常进行系统调用时，首先将 CPU 变为内核级，系统调用返回时又变为用户级。可以将用户代码和数据段的 DPL 改为 0，这样进程就总是处于内核级。

   这个技术在论文《IPv6 下基于病毒过滤防火墙的设计与实现》有提到，但是这种方案的用户空间可以随意修改内核的数据，系统会很不稳定。

   实现原理：CPU 的 cs 寄存器有 2 位的字段，用于指明 CPU 的当前特权级(CPL)，值为 0 表示内核级，3 表示用户级。当把__KERNEL_CS 宏产生的值装进 CS 寄存器，CPU 就是内核级，当把__USER_CS 宏产生的值装进 CS 寄存器，CPU 就是用户级，具体做法就是修改__USER_CS 宏产生的值使 CPU 变为 0。

2. 将 skb 指向的数据的物理地址，映射到进程的用户空间。

   这个技术在 misc 设备实现机制中已经应用到。