Linux 系统内核空间与用户空间通信的实现与分析

级别： 初级

陈鑫 (chen.shin@hotmail.com), 自由软件爱好者, 南京邮电学院电子工程系

2004 年 7 月 01 日

多数的 Linux 内核态程序都需要和用户空间的进程交换数据，但 Linux 内核态无法对传统的 Linux
方法提供足够的支持。本文总结并比较了几种内核态与用户态进程通信的实现方法，并推荐使用 ne
环境与用户态进程通信。

1 引言

Linux 是一个源码开放的操作系统，无论是普通用户还是企业用户都可以编写自己的内核代码，再加
从而制作出适合自己的操作系统。目前有很多中低端用户使用的网络设备的操作系统是从标准 Linux
说明了有越来越多的人正在加入到 Linux 内核开发团体中。

一个或多个内核模块的实现并不能满足一般 Linux 系统软件的需要，因为内核的局限性太大，如不能
能做大延时的处理等等。当我们需要做这些的时候，就需要将在内核态采集到的数据传送到用户态的
进行处理。这样，内核态与用户空间进程通信的方法就显得尤为重要。在 Linux 的内核发行版本中没
的详细介绍，也没有其他文章对此进行总结，所以本文将列举几种内核态与用户态进程通信的方法并
现和适用环境。

---

2 Linux 内核模块的运行环境与传统进程间通信

在一台运行 Linux 的计算机中，CPU 在任何时候只会有如下四种状态：

【1】 在处理一个硬中断。

【2】 在处理一个软中断，如 softirq、tasklet 和 bh。

【3】 运行于内核态，但有进程上下文，即与一个进程相关。

【4】 运行一个用户态进程。

其中，【1】、【2】和【3】是运行于内核空间的，而【4】是在用户空间。其中除了【4】，其他状
的状态抢占。比如，软中断只可以被硬中断抢占。

Linux 内核模块是一段可以动态在内核装载和卸载的代码，装载进内核的代码便立即在内核中工作起
的运行环境有三种：用户上下文环境、硬中断环境和软中断环境。但三种环境的局限性分两种，因为
中断环境的延续。比较如表【1】。

表【1】

内核态环境	介绍	局限性
用户上下文	内核态代码的运行与一用户空间进程相关，如系统调用中代码的运行环境。	不可直接将本地变量传递给用户态的区，因为内核态和用户态的内存映射同。
硬中断和软中断环境	硬中断或软中断过程中代码的运行环境，如 IP 数据报的接收代码的运行环境，网络设备的驱动程序等。	不可直接向用户态内存区传递数据；代码在运行过程中不可阻塞。

Linux 传统的进程间通信有很多，如各类管道、消息队列、内存共享、信号量等等。但它们都无法介

用，原因如表【2】。

表【2】

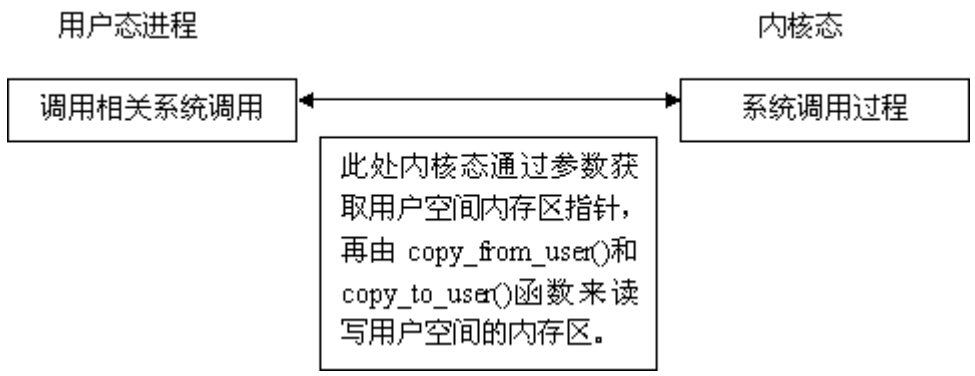
通信方法	无法介于内核态与用户态的原因
管道（不包括命名管道）	局限于父子进程间的通信。
消息队列	在硬、软中断中无法无阻塞地接收数据。
信号量	无法介于内核态和用户态使用。
内存共享	需要信号量辅助，而信号量又无法使用。
套接字	在硬、软中断中无法无阻塞地接收数据。

3 Linux内核态与用户态进程通信方法的提出与实现

3．1 用户上下文环境

运行在用户上下文环境中的代码是可以阻塞的，这样，便可以使用消息队列和 UNIX 域套接字来实现
通信。但这些方法的数据传输效率较低，Linux 内核提供 copy_from_user()/copy_to_user() 函数来实
数据的拷贝，但这两个函数会引发阻塞，所以不能用在硬、软中断中。一般将这两个特殊拷贝函数用
一类的函数中，此类函数在使用中往往"穿梭"于内核态与用户态。此类方法的工作原理路如图【1】。

图【1】



其中相关的系统调用是需要用户自行编写并载入内核。 imp1.tar.gz是一个示例，内核模块注册了一
函数使得用户空间进程可以调用此组函数对内核态数据进行读写。源码包含三个文件，imp1.h 是通用
户态和内核态都要用到的宏。imp1_k.c 是内核模块的源代码。imp1_u.c 是用户态进程的源代码。整
用户态进程向用户上下文环境发送一个字符串，内容为"a message from userspace\n"。然后再由用
态进程发送一个字符串，内容为"a message from kernel\n"。

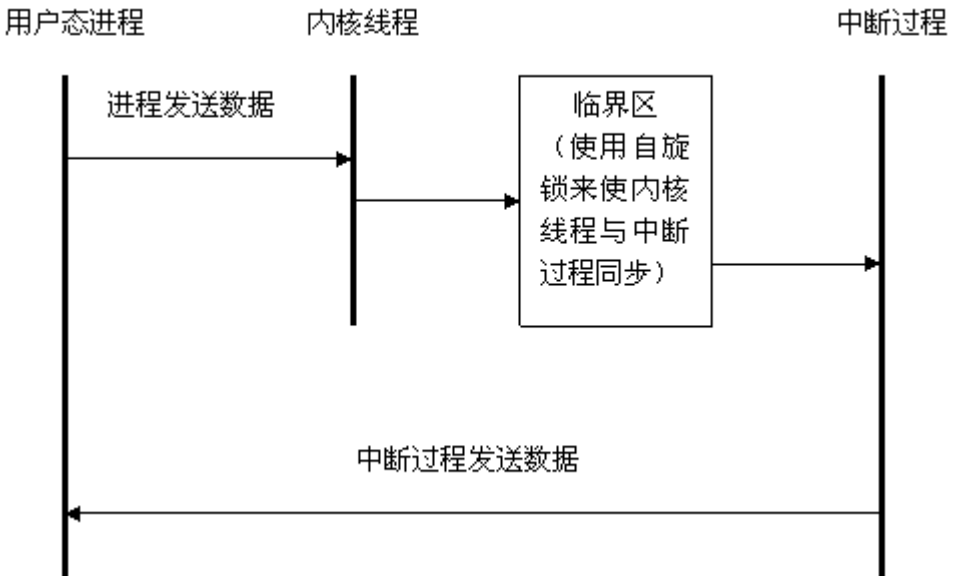
3．2 硬、软中断环境

比起用户上下文环境，硬中断和软中断环境与用户态进程无丝毫关系，而且运行过程不能阻塞。

3．2．1 使用一般进程间通信的方法

我们无法直接使用传统的进程间通信的方法实现。但硬、软中断中也有一套同步机制--自旋锁（spin
旋锁来实现中断环境与中断环境，中断环境与内核线程的同步，而内核线程是运行在有进程上下文环
以在内核线程中使用套接字或消息队列来取得用户空间的数据，然后再将数据通过临界区传递给中断
图【2】。

图【2】



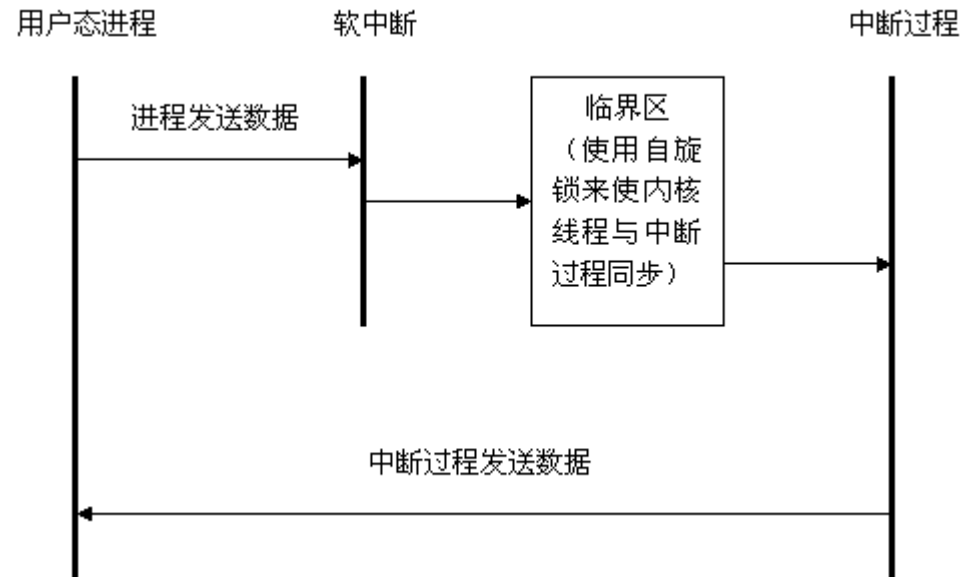
因为中断过程不可能无休止地等待用户态进程发送数据，所以要通过一个内核线程来接收用户空间的
区传给中断过程。中断过程向用户空间的数据发送必须是无阻塞的。这样的通信模型并不令人满意，
其他用户态进程竞争CPU接收数据的，效率很低，这样中断过程便不能实时地接收来自用户空间的数

3．2．2 netlink 套接字

在 Linux 2.4 版以后版本的内核中，几乎全部的中断过程与用户态进程的通信都是使用 netlink 套接
用 netlink 实现了 ip queue 工具，但 ip queue 的使用有其局限性，不能自由地用于各种中断过程。
他一些 Linux 相关文章都没有对 netlink 套接字在中断过程和用户空间通信的应用上作详细的说明，
有一个模糊的概念。

netlink 套接字的通信依据是一个对应于进程的标识，一般定为该进程的 ID。当通信的一端处于中断
0。当使用 netlink 套接字进行通信，通信的双方都是用户态进程，则使用方法类似于消息队列。但
断过程，使用方法则不同。netlink 套接字的最大特点是对中断过程的支持，它在内核空间接收用户
用户自行启动一个内核线程，而是通过另一个软中断调用用户事先指定的接收函数。工作原理如图

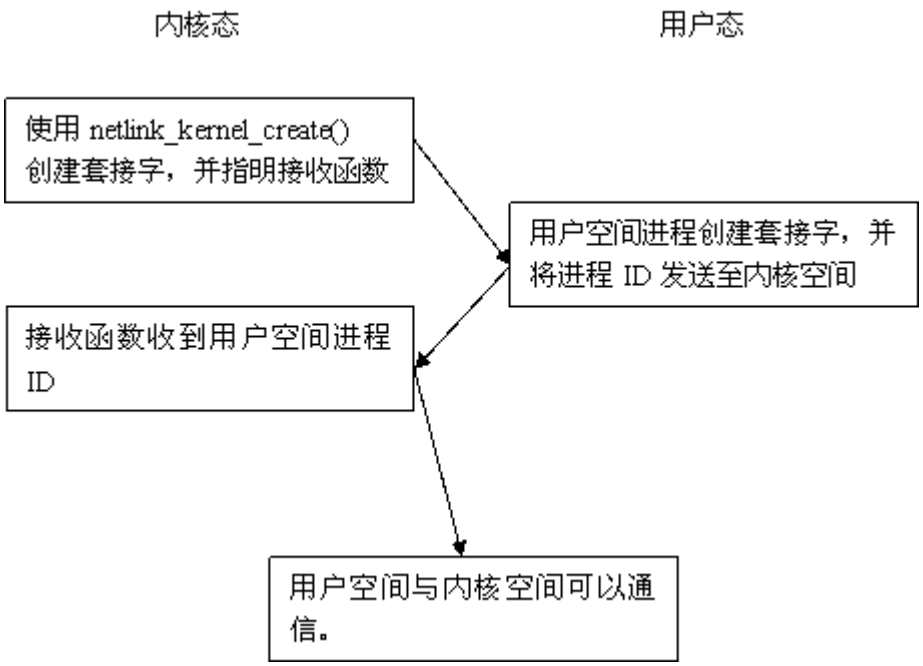
图【3】



很明显，这里使用了软中断而不是内核线程来接收数据，这样就可以保证数据接收的实时性。

当 netlink 套接字用于内核空间与用户空间的通信时，在用户空间的创建方法和一般套接字使用类似方法则不同。图【4】是 netlink 套接字实现此类通信时创建的过程。

图【4】



以下举一个 netlink 套接字的应用示例。示例实现了从 netfilter 的 NF_IP_PRE_ROUTING 点截获的将数据报的相关信息传递到一个用户态进程，由用户态进程将信息打印在终端上。源码在文件 imp2 代码（分段详解）：

（一）模块初始化与卸载

```
static struct sock *nlfd;
struct
{
  __u32 pid;
  rwlock_t lock;
}user_proc;
/*挂接在 netfilter 框架的 NF_IP_PRE_ROUTING 点上的函数为 get_icmp()*/
static struct nf_hook_ops imp2_ops =
{
  .hook = get_icmp,          /*netfilter 钩子函数*/
  .pf = PF_INET,
  .hooknum = NF_IP_PRE_ROUTING,
  .priority = NF_IP_PRI_FILTER -1,
};
static int __init init(void)
{
  rwlock_init(&user_proc.lock);
  /*在内核创建一个 netlink socket，并注明由 kernel_recieve() 函数接收数据
    这里协议 NL_IMP2 是自定的*/
  nlfd = netlink_kernel_create(NL_IMP2, kernel_receive);
  if(!nlfd)
    {
      printk("can not create a netlink socket\n");
      return -1;
    }
  /*向 netfilter 的 NF_IP_PRE_ROUTING 点挂接函数*/
```

```
  return nf_register_hook(&imp2_ops);
}
static void __exit fini(void)
{
  if(nlfd)
    {
      sock_release(nlfd->socket);
         }
  nf_unregister_hook(&imp2_ops);
}
module_init(init);
module_exit(fini);
```

其实片断（一）的工作很简单，模块加载阶段先在内核空间创建一个 netlink 套接字，再将一个函数
的 NF_IP_PRE_ROUTING 钩子点上。卸载时释放套接字所占的资源并注销之前在 netfilter 上挂接

（二）接收用户空间的数据

```
DECLARE_MUTEX(receive_sem);
01:       static void kernel_receive(struct sock *sk, int len)
02:       {
03:        do
04:       {
05:       struct sk_buff *skb;
06:       if(down_trylock(&receive_sem))
07:                           return;
08:
09:       while((skb = skb_dequeue(&sk-<receive_queue)) != NULL)
10:           {
11:                  {
12:                 struct nlmsghdr *nlh = NULL;
13:                                    if(skb-<len <= sizeof(struct nlmsghdr))
14:               {
15:                 nlh = (struct nlmsghdr *)skb-<data;
16:                 if((nlh-<nlmsg_len <= sizeof(struct nlmsghdr))
17:                    && (skb-<len <= nlh-<nlmsg_len))
18:                   {
19:                     if(nlh-<nlmsg_type == IMP2_U_PID)
20:                       {
21:                         write_lock_bh(&user_proc.pid);
22:                         user_proc.pid = nlh-<nlmsg_pid;
23:                                           write_unlock_bh(&user_proc.pid);
24:                       }
25:                     else if(nlh-<nlmsg_type == IMP2_CLOSE)
26:                       {
27:                         write_lock_bh(&user_proc.pid);
28:                                                    if(nlh-<nlmsg_pid == user_proc.pid
29:                         write_unlock_bh(&user_proc.pid);
30:                       }
31:                   }
32:               }
33:          }
34:                             kfree_skb(skb);
35:        }
36:       up(&receive_sem);
37:       }while(nlfd && nlfd-<receive_queue.qlen);
38:       }
```

如果读者看过 ip_queue.c 或 rtnetlink.c中的源码会发现片断（二）中的 03～18 和 31～38 是 netl

间接收数据的框架。在框架中主要是从套接字缓存中取出全部的数据，然后分析是不是合法的数据报据报必须有nlmsghdr 结构的报头。在这里笔者使用了自己定义的消息类型：IMP2_U_PID（消息为用ID），IMP2_CLOSE（用户空间进程关闭）。因为考虑到 SMP，所以在这里使用了读写锁来避免不的问题。kernel_receive() 函数的运行在软中断环境。

（三）截获 IP 数据报

```
static unsigned int get_icmp(unsigned int hook,
                             struct sk_buff **pskb,
                             const struct net_device *in,
                             const struct net_device *out,
                             int (*okfn)(struct sk_buff *))
{
  struct iphdr *iph = (*pskb)->nh.iph;
  struct packet_info info;
  if(iph->protocol == IPPROTO_ICMP)    /*若传输层协议为 ICMP*/
    {
      read_lock_bh(&user_proc.lock);
      if(user_proc.pid != 0)
              {
          read_unlock_bh(&user_proc.lock);
          info.src = iph->saddr;       /*记录源地址*/
          info.dest = iph->daddr;      /*记录目的地址*/
          send_to_user(&info);                  /*发送数据*/
        }
      else
        read_unlock_bh(&user_proc.lock);
    }
  return NF_ACCEPT;
}
```

（四）发送数据

```
static int send_to_user(struct packet_info *info)
{
  int ret;
  int size;
  unsigned char *old_tail;
  struct sk_buff *skb;
  struct nlmsghdr *nlh;
  struct packet_info *packet;
  size = NLMSG_SPACE(sizeof(*info));
  /*开辟一个新的套接字缓存*/
  skb = alloc_skb(size, GFP_ATOMIC);
  old_tail = skb->tail;
  /*填写数据报相关信息*/
  nlh = NLMSG_PUT(skb, 0, 0, IMP2_K_MSG, size-sizeof(*nlh));
  packet = NLMSG_DATA(nlh);
  memset(packet, 0, sizeof(struct packet_info));
  /*传输到用户空间的数据*/
  packet->src = info->src;
  packet->dest = info->dest;
  /*计算经过字节对其后的数据实际长度*/
  nlh->nlmsg_len = skb->tail - old_tail;
  NETLINK_CB(skb).dst_groups = 0;
  read_lock_bh(&user_proc.lock);
  ret = netlink_unicast(nlfd, skb, user_proc.pid, MSG_DONTWAIT); /*发送数据*/
  read_unlock_bh(&user_proc.lock);
  return ret;
```

```
 nlmsg_failure: /*若发送失败，则撤销套接字缓存*/
  if(skb)
    kfree_skb(skb);
  return -1;
}
```

片断（四）中所使用的宏参考如下：

```
/*字节对齐*/
#define NLMSG_ALIGN(len) ( ((len)+NLMSG_ALIGNTO-1) & ~(NLMSG_ALIGNTO-1) )
/*计算包含报头的数据报长度*/
#define NLMSG_LENGTH(len) ((len)+NLMSG_ALIGN(sizeof(struct nlmsghdr)))
/*字节对齐后的数据报长度*/
#define NLMSG_SPACE(len) NLMSG_ALIGN(NLMSG_LENGTH(len))
/*填写相关报头信息，这里使用了nlmsg_failure标签，所以在程序中要定义*/
#define NLMSG_PUT(skb, pid, seq, type, len) \
({ if (skb_tailroom(skb) < (int)NLMSG_SPACE(len)) goto nlmsg_failure; \
   __nlmsg_put(skb, pid, seq, type, len); })
static __inline__ struct nlmsghdr *
__nlmsg_put(struct sk_buff *skb, u32 pid, u32 seq, int type, int len)
{
          struct nlmsghdr *nlh;
          int size = NLMSG_LENGTH(len);
          nlh = (struct nlmsghdr*)skb_put(skb, NLMSG_ALIGN(size));
          nlh->nlmsg_type = type;
          nlh->nlmsg_len = size;
          nlh->nlmsg_flags = 0;
          nlh->nlmsg_pid = pid;
          nlh->nlmsg_seq = seq;
          return nlh;
}
/*跳过报头取实际数据*/
#define NLMSG_DATA(nlh)  ((void*)(((char*)nlh) + NLMSG_LENGTH(0)))
/*取 netlink 控制字段*/
#define NETLINK_CB(skb)               (*(struct netlink_skb_parms*)&((skb)->cb))
```

运行示例时，先编译 imp2_k.c 模块，然后使用 insmod 将模块加载入内核。再运行编译好的 imp2_
示出本机当前接收的 ICMP 数据报的源地址和目的地址。用户可以使用 Ctrl+C 来终止用户空间的进
带来问题。

4 总结

本文从内核态代码的不同运行环境来实现不同方法的内核空间与用户空间的通信，并分析了它们的实
使用 netlink 套接字实现中断环境与用户态进程通信，因为 netlink 套接字是专为此类通信定制的。

参考资料

- Linux 2.4 及后续版本内核源代码；
- www.netfilter.org；
- RFC 3549；

关于作者

陈鑫：南京邮电学院电子工程系 2000 级本科生，自由软件爱好者。喜欢阅读 Linux 内核源代码，的网络模块部分的分析工作。联系方式： chex@njupt.edu.cn