netlink socket 编程 why & how

作者: Kevin Kaichuan He@2005-1-5

翻译整理:duanjigang @2008-9-15<duanjigang1983@126.com>

原文: http://www.linuxjournal.com/article/7356

开发和维护内核是一件很繁杂的工作，因此，只有那些最重要或者与系统性能息息相关的代码才将其安排在内核中。其它程序，比如 GUI，管理以及控制部分的代码，一般都会作为用户态程序。在 linux 系统中，把系统的某个特性分割成在内核中和在用户空间中分别实现一部分的做法是很常见的(比如 linux 系统的防火墙就分成了内核态的 Netfilter 和用户态的 iptables)。然而，内核程序与用户态的程序又是怎样行通讯的呢？

答案就是通过各种各样的用户态和内核态的 IPC(interprocess communication)机制来实现。比如系统调用，ioctl 接口，proc 文件系统以及 netlink socket,本文就是要讨论 netlink socekt 并向读者展示这种用网络通讯接口方式实现的 IPC 机制的优点。

介绍：

netlink socekt 是一种用于在内核态和用户态进程之间进行数据传输的特殊的 IPC。它通过为内核模块提供一组特殊的 API，并为用户程序提供了一组标准的 socket 接口的方式，实现了一种全双工的通讯连接。类似于 TCP/IP 中使用 AF_INET 地址族一样，netlink socket 使用地址族 AF_NETLINK。每一个 netlink socket 在内核头文件 include/linux/netlink.h 中定义自己的协议类型。

下面是 netlink socket 目前的特性集合以及它支持的协议类型：

- **NETLINK_ROUTE**: 用户空间的路由守护程序之间的通讯通道，比如 BGP,OSPF,RIP 以及内核数据转发模块。用户态的路由守护程序通过此类型的协议来更新内核中的路由表。
- **NETLINK_FIREWALL**:接收 IPV4 防火墙代码发送的数据包。
- **NETLINK_NFLOG**:用户态的 iptables 管理工具和内核中的 netfilter 模块之间通讯的通道。
- **NETLINK_ARPD**:用来从用户空间管理内核中的 ARP 表。

为什么以上的功能在实现用户程序和内核程序通讯时，都使用 netlink 方法而不是系统调用，ioctls 或者 proc 文件系统呢？原因在于：为新的特性添加一个新的系统调用,ioctls 或者一个 proc 文件的做法并不是很容易的一件事情，因为我们要冒着污染内核代码并且可能破坏系统稳定性的风险去完成这件事情。然而，netlink socket 却是如此的简单，你只需要在文件 netlink.h 中添加一个常量来标识你的协议类型，然后，内核模块和用户程序就可以立刻使用 socket 风格的 API 进行通讯了！

Netlink 提供了一种异步通讯方式，与其他 socket API 一样，它提供了一个 socket 队列来缓冲或者平滑瞬时的消息高峰。发送 netlink 消息的系统调用在把消息加入到接收者的消息对列后，会触发接收者的接收处理函数。接收者在接收处理函数上下文中，可以决定立即处理消息还是把消息放在队列中，在以后其它上下文去处理它(因为我们希望接收处理函数执行的尽可能快)。系统调用与 netlink 不同，它需要一个同步的处理，因此，当我们使用一个系统调用来从用户态传递消息到内核时，如果处理这个消息的时间很长的话，内核调度的粒度就会受到影响。

内核中实现系统调用的代码都是在编译时静态链接到内核的，因此，在动态加载模块中去包含一个系统调用的做法是不合适的，那是大多数设备驱动的做法。使用 netlink socket 时,动态加载模块中的 netlink 程序不会和 linux 内核中的 netlink 部分产生任何编译时依赖关系。

Netlink 优于系统调用，ioctls 和 proc 文件系统的另外一个特点就是它支持多点传送。一个进程可以把消息传输给一个 netlink 组地址，然后任意多个进程都可以监听那个组地址(并且接收消息)。这种机制为内核到用户态的事件分发提供了一种近乎完美的解决方案。

系统调用和 ioctl 都属于单工方式的 IPC，也就是说，这种 IPC 会话的发起者只能是用户态程序。但是，如果内核有一个紧急的消息想要通知给用户态程序时，该怎么办呢？如果直接使用这些 IPC 的话，是没办法做到这点的。通常情况下，应用程序会周期性的轮询内核以获取状态的改变，然而，高频度的轮询势必会增加系统的负载。Netlink 通过允许内核初始化会话的方式完美的解决了此问题，我们称之为 netlink socket 的双工特性。

最后，netlink socket 提供了一组开发者熟悉的 BSD 风格的 API 函数，因此，相对于使用神秘的系统调用 API 或者 ioctl 而言，netlink 开发培训的费用会更低些。

与 BSD 的 Routing socket 的关系

在 BSD TCP/IP 的协议栈实现中，有一种特殊的 socket 叫做 Routing socket.它的地址族为 AF_ROUTE, 协议族为 PF_ROUTE, socket 类型为 SOCK_RAW. 这种 Routing socket 是用户态进程用来向内核中的路由表增加或者删除路由信息用的。在 Linux 系统中，netlink socket 通过协议类型 NETLINK_ROUTE 实现了与 Routing socket 相同的功能，可以说，netlink socket 提供了 BSD Routing socket 功能的超集。

Netlink Socket 的 API

标准的 socket API 函数-socket(), sendmsg(), recvmsg()和 close()-都能够被用户态程序直接调用来访问 netlink socket.你可以访问 man 手册来获取这些函数的详细定义。在本文，我们只讨论怎样在 netlink socket 的上下文中为这些函数选择参数。这些 API 对于使用 TCP/IP socket 写过一些简单网络程序的读者来说应该很熟悉了。

使用 socket()函数创建一个 socket，输入：

int socket(int domain, int type, int protocol)

socket 域(地址族)是 AF_NETLINK,socket 的类型是 SOCK_RAW 或者 SOCK_DGRAM,因为 netlink 是一种面向数据包的服务。

协议类型选择 netlink 要使用的类型即可。下面是一些预定义的 netlink 协议类型：

NETLINK_ROUTE, NETLINK_FIREWALL, NETLINK_ARPD, NETLINK_ROUTE6

和 NETLINK_IP6_FW. 你同样可以很轻松的在 netlink.h 中添加自定义的协议类型。

每个 netlink 协议类型可以定义高达 32 个多点传输的组。每个组用一个比特位来表示，1<<i，0<=i<=31.

当一组用户态进程和内核态进程协同实现一个相同的特性时，这个方法很有用，因为发送多点传输的 netlink 消息可以减少系统调用的次数，并且减少了相关应用程序的个数，这些程序本来是要用来处理维护多点传输组之间关系而带来的负载的。

bind()函数

跟 TCP/IP 中的 socket 一样，netlink 的 bind()函数把一个本地 socket 地址(源 socket 地址)与一个打开的 socket 进行关联，netlink 地址结构体如下:

```
struct sockaddr_nl
{
  sa_family_t     nl_family;  /* AF_NETLINK    */
  unsigned short nl_pad;      /* zero          */
  __u32           nl_pid;     /* process pid */
  __u32           nl_groups;  /* mcast groups mask */
} nladdr;
```

当上面的结构体被 bind()函数调用时，sockaddr_nl 的 nl_pid 属性的值可以设置为访问 netlink socket 的当前进程的 PID，nl_pid 作为这个 netlink socket 的本地地址。应用程序应该选择一个唯一的 32 位整数来填充 nl_pid 的值。

NL_PID 公式 1: nl_pid = getpid();

公式一使用进程的 PID 作为 nl_pid 的值，如果这个进程只需要一个该类型协议的 netlink socket 的话，选用进程 pid 作为 nl_pid 是一个很自然的做法。

换一种情形，如果一个进程的多个线程想要创建属于各个线程的相同协议类型的 netlink socket 的话，公式二可以用来为每个线程的 netlink socket 产生 nl_pid 值。

NL_PID 公式 2: pthread_self() << 16 | getpid();

采用这种方法，同一进程的不同线程都能获取属于它们的相同协议类型的不同 netlink socket。事实上，

即便是在一个单独的线程里，也可能需要创建同一协议类型的多个 netlink socket。所以开发人员需要更多聪明才智去创建不同的 nl_pid 值，然而本文中不会就如何创建多个不同的 nl_pid 的值进行过多的讨论

如果应用程序想要接收特定协议类型的发往指定多播组的 netlink 消息的话，所有接收组的比特位应该进行与运算，形成 sockaddr_nl 的 nl_groups 域的值。否则的话，nl_groups 应该设置为 0，以便应用程序只能够收到发送给它的 netlink 消息。在填充完结构体 nladdr 后，作如下的绑定工作:

bind(fd, (struct sockaddr*)&nladdr, sizeof(nladdr));

发送一个 netlink 消息

为了能够把一个 netlink 消息发送给内核或者别的用户进程，类似于 UDP 数据包发送的 sendmsg()函数一样，我们需要另外一个结构体 struct sockaddr_nl nladdr 作为目的地址。如果这个 netlink 消息是发往内核的话，nl_pid 属性和 nl_groups 属性都应该设置为 0。

如果这个消息是发往另外一个进程的单点传输消息，nl_pid 应该设置为接收者进程的 PID，nl_groups 应该设置为 0，假设系统中使用了公式 1。

如果消息是发往一个或者多个多播组的话，应该用所有目的多播组的比特位与运算形成 nl_groups 的值。然后我们就可以将 netlink 地址应用到结构体 struct msghdr msg 中，供函数 sendmsg()来调用：

```
struct msghdr msg;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);
```

netlink 消息同样也需要它自身的消息头，这样做是为了给所有协议类型的 netlink 消息提供一个通用的背景。

由于 linux 内核的 netlink 部分总是认为在每个 netlink 消息体中已经包含了下面的消息头，所以每个应用程序在发送 netlink 消息之前需要提供这个头信息：

```
struct nlmsghdr
{
  __u32 nlmsg_len;    /* Length of message */
  __u16 nlmsg_type;   /* Message type*/
  __u16 nlmsg_flags; /* Additional flags */
  __u32 nlmsg_seq;    /* Sequence number */
  __u32 nlmsg_pid;    /* Sending process PID */
};
```

nlmsg_len 需要用 netlink 消息体的总长度来填充，包含头信息在内，这个是 netlink 核心需要的信息。

mlmsg_type 可以被应用程序所用，它对于 netlink 核心来说是一个透明的值。Nsmsg_flags 用来该对消息体进行另外的控制，会被 netlink 核心代码读取并更新。Nlmsg_seq 和 nlmsg_pid 同样对于 netlink 核心部分来说是透明的，应用程序用它们来跟踪消息。

因此，一个 netlink 消息体由 nlmsghdr 和消息的 payload 部分组成。一旦输入一个消息，它就会进入一个被 nlh 指针指向的缓冲区。我们同样可以把消息发送个结构体 struct msghdr msg:

```
struct iovec iov;
iov.iov_base = (void *)nlh;
iov.iov_len = nlh->nlmsg_len;
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
```

在完成了以上步骤后，调用一次 sendmsg()函数就能把 netlink 消息发送出去：

```
sendmsg(fd, &msg, 0);
```

接收 netlink 消息：

接收程序需要申请足够大的空间来存储 netlink 消息头和消息的 payload 部分。它会用如下的方式填充结构体 struct msghdr msg,然后使用标准函数接口 recvmsg()来接收 netlink 消息，假设 nlh 指向缓冲区：

```
struct sockaddr_nl nladdr;
struct msghdr msg;
struct iovec iov;

iov.iov_base = (void *)nlh;
iov.iov_len = MAX_NL_MSG_LEN;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);

msg.msg_iov = &iov;
msg.msg_iovlen = 1;
recvmsg(fd, &msg, 0);
```

当消息正确接收后，nlh 应该指向刚刚接收到的 netlink 消息的头部分。Nladdr 应该包含接收到消息体的目的地信息，这个目的地信息由 pid 和消息将要发往的多播组的值组成。Netlink.h 中的宏定义 NLMSG_DATA(nlh)返回指向 netlink 消息体的 payload 的指针。调用 close(fd)就可以关闭掉 fd 描述符代表的 netlink socket.

内核空间的 netlink API 接口

内核空间的 netlink API 是由内核中的 netlink 核心代码支持的，在 net/core/af_netlink.c 中实现。从内核的角度来说，API 接口与用户空间的 API 是不一样的。内核模块通过这些 API 访问 netlink socket 并且与用户空间的程序进行通讯。如果你不想使用 netlink 预定义好的协议类型的话，可以在 netlink.h 中添加一个自定义的协议类型。例如，我们可以通过在 netlink.h 中插入下面的代码行，添加一个测试用的协议类型：

#define NETLINK_TEST 17

然后，就可以在 linux 内核的任何部分访问这个协议类型了。

在用户空间，我们通过 socket()调用来创建一个 netlink socket,但是在内核空间，我们调用如下的 API：

struct sock * netlink_kernel_create(int unit, void (*input)(struct sock *sk, int len));

参数 uint 是 netlink 协议类型，例如 NETLINK_TEST。函数指针，input,是 netlink socket 在收到消息时调用的处理消息的回调函数指针。

在内核创建了一个 NETLINK_TEST 类型的 netlink socket 后，无论什么时候，只要用户程序发送一个 NETLINK_TEST 类型的 netlink 消息到内核的话，通过 netlink_kernel_create()函数注册的回调函数 input()都会被调用。下面是一个实现了消息处理函数 input 的例子。

```
void input (struct sock *sk, int len)
{
  struct sk_buff *skb;
  struct nlmsghdr *nlh = NULL;
  u8 *payload = NULL;

  while ((skb = skb_dequeue(&sk->receive_queue))
          != NULL) {
  /* process netlink message pointed by skb->data */
  nlh = (struct nlmsghdr *)skb->data;
  payload = NLMSG_DATA(nlh);
  /* process netlink message with header pointed by
```

```
     * nlh       and payload pointed by payload
     */
    }
  }
```

回调函数 input()是在发送进程的系统调用 sendmsg()的上下文被调用的。如果 input 函数中处理消息很快的话，一切都没有问题。但是如果处理 netlink 消息花费很长时间的话，我们则希望把消息的处理部分放在 input()函数的外面，因为长时间的消息处理过程可能会阻止其它系统调用进入内核。取而代之，我们可以牺牲一个内核线程来完成后续的无限的的处理动作。

使用

```
skb = skb_recv_datagram(nl_sk)
```

来接收消息。nl_sk 是 netlink_kernel_create()函数返回的 netlink socket,然后，只需要处理 skb->data 指针指向的 netlink 消息就可以了。

这个内核线程会在 nl_sk 中没有消息的时候睡眠。因此，在回调函数 input()中我们要做的事情就是唤醒睡眠的内核线程，像这样的方式:

```
void input (struct sock *sk, int len)
{
   wake_up_interruptible(sk->sleep);
}
```

这就是一个升级版的内核与用户空间的通讯模型，它提高了上下文切换的粒度。

从内核中发送 netlink 消息

就像从用户空间发送消息一样，内核在发送 netlink 消息时也需要设置源 netlink 地址和目的 netlink 地址。假设结构体 struct sk_buff * skb 指向存储着要发送的 netlink 消息的缓冲区，源地址可以这样设置:

```
NETLINK_CB(skb).groups = local_groups;
NETLINK_CB(skb).pid = 0;      /* from kernel */
```

目的地址可以这样设置:

```
NETLINK_CB(skb).dst_groups = dst_groups;
NETLINK_CB(skb).dst_pid = dst_pid;
```

这些信息并不存储在 skb->data 中，相反，它们存储在 socket 缓冲区的 netlink 控制块 skb 中.

发送一个单播消息，使用:

int netlink_unicast(struct sock *ssk, struct sk_buff *skb, u32 pid, int nonblock);

ssk 是 by netlink_kernel_create()函数返回的 netlink socket, skb->data 指向需要发送的 netlink 消息体，如果使用公式一的话，pid 是接收程序的 pid，noblock 表明当接收缓冲区不可用时是否应该阻塞还是立即返回一个失败信息。

你同样可以从内核发送一个多播消息。下面的函数同时把一个 netlink 消息发送给 pid 指定的进程和 group 标识的多个组。

void netlink_broadcast(struct sock *ssk, struct sk_buff *skb, u32 pid, u32 group, int allocation);

group 的值是接收消息的各个组的比特位进行与运算的结果。Allocation 是内核内存的申请类型。通常情况下在中断上下文使用 GFP_ATOMIC,否则使用 GFP_KERNEL。这是由于发送多播消息时，API 可能需要申请一个或者多个 socket 缓冲区并进行拷贝所引起的。

从内核空间关闭 netlink socket

netlink_kernel_create()函数返回的 netlink socket 为 struct sock *nl_sk，我们可以通过访问下面的 API 来从内核空间关闭这个 netlink socket:

sock_release(nl_sk->socket);

到目前为止，我们已经演示了 netlink 编程概念的最小代码框架。接着我们会使用 NETLINK_TEST 协议类型，并且假设它已经被添加到内核头文件中了。这里列举的内核模块代码只是与 netlink 相关的，所以，你应该把它插入到一个完整的内核模块代码当中去，这样的完整代码在其它代码中可以找到很多。

内核与用户空间的单工通讯:

在这个例子中，一个用户态进程发送一个 netlink 消息给内核模块，内核模块把消息回送给发送进程，下面是用户态的代码：

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/socket.h>
```

```
#define NETLINK_TEST 17
#define MAX_PAYLOAD 1024    /* maximum payload size*/
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
struct iovec iov;
int sock_fd;
struct msghdr msg;

int main(int argc, char* argv[])
{
        sock_fd = socket(PF_NETLINK, SOCK_RAW,NETLINK_TEST);
        memset(&msg, 0, sizeof(msg));
        memset(&src_addr, 0, sizeof(src_addr));
        src_addr.nl_family = AF_NETLINK;
        src_addr.nl_pid = getpid();    /* self pid */
        src_addr.nl_groups = 0;    /* not in mcast groups */
        bind(sock_fd, (struct sockaddr*)&src_addr, sizeof(src_addr));
        memset(&dest_addr, 0, sizeof(dest_addr));
        dest_addr.nl_family = AF_NETLINK;
        dest_addr.nl_pid = 0;      /* For Linux Kernel */
        dest_addr.nl_groups = 0; /* unicast */

        nlh=(struct nlmsghdr *)malloc(NLMSG_SPACE(MAX_PAYLOAD));
        /* Fill the netlink message header */
        nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
        nlh->nlmsg_pid = getpid();    /* self pid */
        nlh->nlmsg_flags = 0;
        /* Fill in the netlink message payload */
        strcpy(NLMSG_DATA(nlh), "Hello you!");
```

```
        iov.iov_base = (void *)nlh;
        iov.iov_len = nlh->nlmsg_len;
        msg.msg_name = (void *)&dest_addr;
        msg.msg_namelen = sizeof(dest_addr);
        msg.msg_iov = &iov;
        msg.msg_iovlen = 1;

        sendmsg(sock_fd, &msg, 0);

        /* Read message from kernel */
        memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));
        recvmsg(sock_fd, &msg, 0);
        printf(" Received message payload: %s\n",
        NLMSG_DATA(nlh));

         /* Close Netlink Socket */
        close(sock_fd);
}
```

下面是内核代码:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/sched.h>
#include <net/sock.h>
#include <linux/netlink.h>

#define NETLINK_TEST 17
struct sock *nl_sk = NULL;
void nl_data_ready (struct sock *sk, int len)
```

```
{
        wake_up_interruptible(sk->sk_sleep);
}

void test_netlink(void)
{
        struct sk_buff * skb = NULL;
        struct nlmsghdr * nlh = NULL;
        int err;
        u32 pid;

        nl_sk = netlink_kernel_create(NETLINK_TEST, nl_data_ready);
        /* wait for message coming down from user-space */
        skb = skb_recv_datagram(nl_sk, 0, 0, &err);

        nlh = (struct nlmsghdr *)skb->data;
        printk("%s: received netlink message payload:%s\n", __FUNCTION__, (char*)NLMSG_DATA(nlh));

        pid = nlh->nlmsg_pid; /*pid of sending process */
        NETLINK_CB(skb).groups = 0; /* not in mcast group */
        NETLINK_CB(skb).pid = 0;            /* from kernel */
        NETLINK_CB(skb).dst_pid = pid;
        NETLINK_CB(skb).dst_groups = 0;    /* unicast */
        netlink_unicast(nl_sk, skb, pid, MSG_DONTWAIT);
        sock_release(nl_sk->sk_socket);
}

int init_module()
{
        test_netlink();
```

```
        return 0;
}
void cleanup_module( )
{
}
MODULE_LICENSE("GPL");
MODULE_AUTHOR("duanjigang");
```

在加载完上面的内核代码生成的模块以后，运行用户态程序，可以看到下面输出:

Received message payload: Hello you!

并且，下面的信息会在 dmesg 中输出:

netlink_test: received netlink message payload:

Hello you!

内核与用户空间之间的多点传输通讯:

在这个例子中，两个用户空间程序监听相同的 netlink 多播组。内核模块通过 netlink socket 向这个组发送一个消息，所有的应用程序都收到了这个消息，下面是用户态的代码:

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/socket.h>

#define NETLINK_TEST 17
#define MAX_PAYLOAD 1024  /* maximum payload size*/
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
```

```
struct iovec iov;
int sock_fd;
struct msghdr msg;

int main(int argc, char* argv[])
{

        sock_fd=socket(PF_NETLINK, SOCK_RAW, NETLINK_TEST);
        memset(&src_addr, 0, sizeof(src_addr));
        memset(&msg, 0, sizeof(msg));

        src_addr.nl_family = AF_NETLINK;
        src_addr.nl_pid = getpid();    /* self pid */
        /* interested in group 1<<0 */
        src_addr.nl_groups = 1;
        bind(sock_fd, (struct sockaddr*)&src_addr, sizeof(src_addr));
        memset(&dest_addr, 0, sizeof(dest_addr));
        nlh = (struct nlmsghdr *)malloc(NLMSG_SPACE(MAX_PAYLOAD));
        memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));

        iov.iov_base = (void *)nlh;
        iov.iov_len = NLMSG_SPACE(MAX_PAYLOAD);
        msg.msg_name = (void *)&dest_addr;
        msg.msg_namelen = sizeof(dest_addr);
        msg.msg_iov = &iov;
        msg.msg_iovlen = 1;

        printf("Waiting for message from kernel\n");

         /* Read message from kernel */
```

```
        recvmsg(sock_fd, &msg, 0);

        printf("Received message payload: %s\n", NLMSG_DATA(nlh));
        close(sock_fd);
}
```

下面是内核态的代码:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/sched.h>
#include <net/sock.h>
#include <linux/netlink.h>

#define MAX_PAYLOAD 1024
#define NETLINK_TEST 17

struct sock *nl_sk = NULL;

void nl_data_ready (struct sock *sk, int len)
{
        wake_up_interruptible(sk->sk_sleep);
}

void test_netlink(void)
{
        struct sk_buff *skb = NULL;
        struct nlmsghdr *nlh;

        nl_sk = netlink_kernel_create(NETLINK_TEST, nl_data_ready);
```

```
        skb = alloc_skb(NLMSG_SPACE(MAX_PAYLOAD),GFP_KERNEL);
        nlh = (struct nlmsghdr *)skb->data;
        nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
        nlh->nlmsg_pid = 0;    /* from kernel */
        nlh->nlmsg_flags = 0;
        nlh = (struct nlmsghdr *) skb_put(skb, NLMSG_SPACE(MAX_PAYLOAD));
        strcpy(NLMSG_DATA(nlh), "Greeting from kernel!");
        /* sender is in group 1<<0 */
        NETLINK_CB(skb).groups = 1;
        NETLINK_CB(skb).pid = 0;    /* from kernel */
        NETLINK_CB(skb).dst_pid = 0;    /* multicast */
        /* to mcast group 1<<0 */
        NETLINK_CB(skb).dst_groups = 1;
        /*multicast the message to all listening processes*/
        netlink_broadcast(nl_sk, skb, 0, 1, GFP_KERNEL);
        //printk("%s\n", NLMSG_DATA(nlh));
        sock_release(nl_sk->sk_socket);
}

int init_module()
{
        test_netlink();
        return 0;
}
void cleanup_module( )
{

}
MODULE_LICENSE("GPL");
MODULE_AUTHOR("duanjigang");
```

假设用户态的代码编译后生成的可执行程序为 nl_recv,我们可以运行 nl_recv 的两个实例:

./nl_recv &

Waiting for message from kernel

./nl_recv &

Waiting for message from kernel

然后，当我们加载了内核模块后，两个 nl_recv 的实例都应该会输出如下的消息:

Received message payload: Greeting from kernel!

Received message payload: Greeting from kernel!

总结: Netlink socket 是用户空间程序和内核模块之间一种很灵活的通讯接口。它为应用程序和内核程序都提供了一个方便使用的 API。它还提供了高级的通讯特性，比如全双工，缓冲 I/O,多点传送和异步通讯，这些都是其他内核/用户态的 IPC 所不具有的功能。