<http://blogold.chinaunix.net/u3/94916/showart_1915146.html>

Netlink 是一种特殊的 socket，它是 Linux 所特有的，类似于 BSD 中的AF\_ROUTE 但又远比它的功能强大，目前在最新的 Linux 内核（2.6.14）中使用netlink 进行应用与内核通信的应用很多，包括：

路由 daemon（NETLINK\_ROUTE），

1-wire 子系统（NETLINK\_W1），

用户态 socket 协议（NETLINK\_USERSOCK），

防火墙（NETLINK\_FIREWALL），

socket 监视（NETLINK\_INET\_DIAG），

netfilter 日志（NETLINK\_NFLOG），

ipsec 安全策略（NETLINK\_XFRM），

SELinux 事件通知（NETLINK\_SELINUX），

iSCSI 子系统（NETLINK\_ISCSI），

进程审计（NETLINK\_AUDIT），

转发信息表查询（NETLINK\_FIB\_LOOKUP），

netlink connector(NETLINK\_CONNECTOR),

netfilter 子系统（NETLINK\_NETFILTER），

IPv6 防火墙（NETLINK\_IP6\_FW），

DECnet 路由信息（NETLINK\_DNRTMSG），

内核事件向用户态通知（NETLINK\_KOBJECT\_UEVENT），

通用 netlink（NETLINK\_GENERIC）。

Netlink 是一种在内核与用户应用间进行双向数据传输的非常好的方式，用户态应用使用标准的 socket API 就可以使用 netlink 提供的强大功能，内核态需要使用专门的内核 API 来使用 netlink。

Netlink 相对于系统调用，ioctl 以及 /proc 文件系统而言具有以下优点：

1，为了使用 netlink，用户仅需要在 include/linux/netlink.h 中增加一个新类型的 netlink 协议定义即可，如 #define NETLINK\_MYTEST 17 然后，内核和用户态应用就可以立即通过 socket API 使用该 netlink 协议类型进行数据交换。但系统调用需要增加新的系统调用，ioctl 则需要增加设备或文件， 那需要不少代码，proc 文件系统则需要在 /proc 下添加新的文件或目录，那将使本来就混乱的 /proc 更加混乱。

2. netlink是一种异步通信机制，在内核与用户态应用之间传递的消息保存在socket缓存队列中，发送消息只是把消息保存在接收者的socket的接 收队列，而不需要等待接收者收到消息，但系统调用与 ioctl 则是同步通信机制，如果传递的数据太长，将影响调度粒度。

3．使用 netlink 的内核部分可以采用模块的方式实现，使用 netlink 的应用部分和内核部分没有编译时依赖，但系统调用就有依赖，而且新的系统调用的实现必须静态地连接到内核中，它无法在模块中实现，使用新系统调用的应用在编译时需要依赖内核。

4．netlink 支持多播，内核模块或应用可以把消息多播给一个netlink组，属于该neilink 组的任何内核模块或应用都能接收到该消息，内核事件向用户态的通知机制就使用了这一特性，任何对内核事件感兴趣的应用都能收到该子系统发送的内核事件，在后面的文章中将介绍这一机制的使用。

5．内核可以使用 netlink 首先发起会话，但系统调用和 ioctl 只能由用户应用发起调用。

6．netlink 使用标准的 socket API，因此很容易使用，但系统调用和 ioctl则需要专门的培训才能使用。

用户态使用 netlink

用户态应用使用标准的socket APIs， socket(), bind(), sendmsg(), recvmsg() 和 close() 就能很容易地使用 netlink socket，查询手册页可以了解这些函数的使用细节，本文只是讲解使用 netlink 的用户应该如何使用这些函数。注意，使用 netlink 的应用必须包含头文件 linux/netlink.h。当然 socket 需要的头文件也必不可少，sys/socket.h。

为了创建一个 netlink socket，用户需要使用如下参数调用 socket():

socket(AF\_NETLINK, SOCK\_RAW, netlink\_type)

第一个参数必须是 AF\_NETLINK 或 PF\_NETLINK，在 Linux 中，它们俩实际为一个东西，它表示要使用netlink，第二个参数必须是SOCK\_RAW或SOCK\_DGRAM， 第三个参数指定netlink协议类型，如前面讲的用户自定义协议类型NETLINK\_MYTEST， NETLINK\_GENERIC是一个通用的协议类型，它是专门为用户使用的，因此，用户可以直接使用它，而不必再添加新的协议类型。内核预定义的协议类型有：

#define NETLINK\_ROUTE 0 /\* Routing/device hook \*/

#define NETLINK\_W1 1 /\* 1-wire subsystem \*/

#define NETLINK\_USERSOCK 2 /\* Reserved for user mode socket protocols \*/

#define NETLINK\_FIREWALL 3 /\* Firewalling hook \*/

#define NETLINK\_INET\_DIAG 4 /\* INET socket monitoring \*/

#define NETLINK\_NFLOG 5 /\* netfilter/iptables ULOG \*/

#define NETLINK\_XFRM 6 /\* ipsec \*/

#define NETLINK\_SELINUX 7 /\* SELinux event notifications \*/

#define NETLINK\_ISCSI 8 /\* Open-iSCSI \*/

#define NETLINK\_AUDIT 9 /\* auditing \*/

#define NETLINK\_FIB\_LOOKUP 10

#define NETLINK\_CONNECTOR 11

#define NETLINK\_NETFILTER 12 /\* netfilter subsystem \*/

#define NETLINK\_IP6\_FW 13

#define NETLINK\_DNRTMSG 14 /\* DECnet routing messages \*/

#define NETLINK\_KOBJECT\_UEVENT 15 /\* Kernel messages to userspace \*/

#define NETLINK\_GENERIC 16

对于每一个netlink协议类型，可以有多达 32多播组，每一个多播组用一个位表示，netlink 的多播特性使得发送消息给同一个组仅需要一次系统调用，因而对于需要多拨消息的应用而言，大大地降低了系统调用的次数。

函数 bind() 用于把一个打开的 netlink socket 与 netlink 源 socket 地址绑定在一起。netlink socket 的地址结构如下：

struct sockaddr\_nl

{

sa\_family\_t nl\_family;

unsigned short nl\_pad;

\_\_u32 nl\_pid;

\_\_u32 nl\_groups;

};

、字段 nl\_family 必须设置为 AF\_NETLINK 或着 PF\_NETLINK，字段 nl\_pad 当前没有使用，因此要总是设置为 0，字段 nl\_pid 为接收或发送消息的进程的 ID，如果希望内核处理消息或多播消息，就把该字段设置为 0，否则设置为处理消息的进程 ID。字段 nl\_groups 用于指定多播组，bind 函数用于把调用进程加入到该字段指定的多播组，如果设置为 0，表示调用者不加入任何多播组。

传递给 bind 函数的地址的 nl\_pid 字段应当设置为本进程的进程 ID，这相当于 netlink socket 的本地地址。但是，对于一个进程的多个线程使用 netlink socket 的情况，字段 nl\_pid 则可以设置为其它的值，如：

pthread\_self() << 16 | getpid();

因此字段 nl\_pid 实际上未必是进程 ID,它只是用于区分不同的接收者或发送者的一个标识，用户可以根据自己需要设置该字段。函数 bind 的调用方式如下：

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

fd为前面的 socket 调用返回的文件描述符，参数 nladdr 为 struct sockaddr\_nl 类型的地址。 为了发送一个 netlink 消息给内核或其他用户态应用，需要填充目标 netlink socket 地址 ，此时，字段 nl\_pid 和 nl\_groups 分别表示接收消息者的进程 ID 与多播组。如果字段 nl\_pid 设置为 0，表示消息接收者为内核或多播组，如果 nl\_groups为 0，表示该消息为单播消息，否则表示多播消息。 使用函数 sendmsg 发送 netlink 消息时还需要引用结构 struct msghdr、struct nlmsghdr 和 struct iovec，结构 struct msghdr 需如下设置：

struct msghdr msg;

memset(&msg, 0, sizeof(msg));

msg.msg\_name = (void \*)&(nladdr);

msg.msg\_namelen = sizeof(nladdr);

其中 nladdr 为消息接收者的 netlink 地址。

struct nlmsghdr 为 netlink socket 自己的消息头，这用于多路复用和多路分解 netlink 定义的所有协议类型以及其它一些控制，netlink 的内核实现将利用这个消息头来多路复用和多路分解已经其它的一些控制，因此它也被称为netlink 控制块。因此，应用在发送 netlink 消息时必须提供该消息头。

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 \*/

};

字段 nlmsg\_len 指定消息的总长度，包括紧跟该结构的数据部分长度以及该结构的大小，字段 nlmsg\_type 用于应用内部定义消息的类型，它对 netlink 内核实现是透明的，因此大部分情况下设置为 0，字段 nlmsg\_flags 用于设置消息标志，可用的标志包括：

/\* Flags values \*/

#define NLM\_F\_REQUEST 1 /\* It is request message. \*/

#define NLM\_F\_MULTI 2 /\* Multipart message, terminated by NLMSG\_DONE \*/

#define NLM\_F\_ACK 4 /\* Reply with ack, with zero or error code \*/

#define NLM\_F\_ECHO 8 /\* Echo this request \*/

/\* Modifiers to GET request \*/

#define NLM\_F\_ROOT 0x100 /\* specify tree root \*/

#define NLM\_F\_MATCH 0x200 /\* return all matching \*/

#define NLM\_F\_ATOMIC 0x400 /\* atomic GET \*/

#define NLM\_F\_DUMP (NLM\_F\_ROOT|NLM\_F\_MATCH)

/\* Modifiers to NEW request \*/

#define NLM\_F\_REPLACE 0x100 /\* Override existing \*/

#define NLM\_F\_EXCL 0x200 /\* Do not touch, if it exists \*/

#define NLM\_F\_CREATE 0x400 /\* Create, if it does not exist \*/

#define NLM\_F\_APPEND 0x800 /\* Add to end of list \*/

#define MAX\_MSGSIZE 1024

char buffer[] = "An example message";

struct nlmsghdr nlhdr;

nlhdr = (struct nlmsghdr \*)malloc(NLMSG\_SPACE(MAX\_MSGSIZE));

strcpy(NLMSG\_DATA(nlhdr),buffer);

nlhdr->nlmsg\_len = NLMSG\_LENGTH(strlen(buffer));

nlhdr->nlmsg\_pid = getpid(); /\* self pid \*/

nlhdr->nlmsg\_flags = 0;

结构 struct iovec 用于把多个消息通过一次系统调用来发送，下面是该结构使用示例：

struct iovec iov;  
iov.iov\_base = (void \*)nlhdr;  
iov.iov\_len = nlh->nlmsg\_len;  
msg.msg\_iov = &iov;  
msg.msg\_iovlen = 1;

在完成以上步骤后，消息就可以通过下面语句直接发送：

sendmsg(fd, &msg, 0);

应用接收消息时需要首先分配一个足够大的缓存来保存消息头以及消息的数据部分，然后填充消息头，添完后就可以直接调用函数 recvmsg() 来接收。

#define MAX\_NL\_MSG\_LEN 1024

struct sockaddr\_nl nladdr;

struct msghdr msg;

struct iovec iov;

struct nlmsghdr \* nlhdr;

nlhdr = (struct nlmsghdr \*)malloc(MAX\_NL\_MSG\_LEN);

iov.iov\_base = (void \*)nlhdr;

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);

注意：fd为socket调用打开的netlink socket描述符。

在消息接收后，nlhdr指向接收到的消息的消息头，nladdr保存了接收到的消息的目标地址，宏NLMSG\_DATA(nlhdr)返回指向消息的数据部分的指针。

在linux/netlink.h中定义了一些方便对消息进行处理的宏，这些宏包括：

#define NLMSG\_ALIGNTO 4

/\*宏NLMSG\_ALIGN(len)用于得到不小于len且字节对齐的最小数值\*/

#define NLMSG\_ALIGN(len) ( ((len)+NLMSG\_ALIGNTO-1) & ~(NLMSG\_ALIGNTO-1) )

/\*宏NLMSG\_LENGTH(len)用于计算数据部分长度为len时实际的消息长度。它一般用于分配消息缓存\*/

#define NLMSG\_LENGTH(len) ((len)+NLMSG\_ALIGN(sizeof(struct nlmsghdr)))

/\*宏NLMSG\_SPACE(len)返回不小于NLMSG\_LENGTH(len)且字节对齐的最小数值，它也用于分配消息缓存\*/

#define NLMSG\_SPACE(len) NLMSG\_ALIGN(NLMSG\_LENGTH(len))

/\*宏NLMSG\_DATA(nlh)用于取得消息的数据部分的首地址，设置和读取消息数据部分时需要使用该宏\*/

#define NLMSG\_DATA(nlh) ((void\*)(((char\*)nlh) + NLMSG\_LENGTH(0)))

/\*宏NLMSG\_NEXT(nlh,len)用于得到下一个消息的首地址，同时len也减少为剩余消息的总长度，该宏一般在一个消息被分成几个部分发送或接收时使用\*/

#define NLMSG\_NEXT(nlh,len) ((len) -= NLMSG\_ALIGN((nlh)->nlmsg\_len), \

(struct nlmsghdr\*)(((char\*)(nlh)) + NLMSG\_ALIGN((nlh)->nlmsg\_len)))

/\*宏NLMSG\_OK(nlh,len)用于判断消息是否有len这么长\*/

#define NLMSG\_OK(nlh,len) ((len) >= (int)sizeof(struct nlmsghdr) && \

(nlh)->nlmsg\_len >= sizeof(struct nlmsghdr) && \

(nlh)->nlmsg\_len <= (len))

/\*宏NLMSG\_PAYLOAD(nlh,len)用于返回payload的长度\*/

#define NLMSG\_PAYLOAD(nlh,len) ((nlh)->nlmsg\_len - NLMSG\_SPACE((len)))

函数close用于关闭打开的netlink socket。

netlink内核API

netlink的内核实现在.c文件net/core/af\_netlink.c中，内核模块要想使用netlink，也必须包含头文件 linux/netlink.h。内核使用netlink需要专门的API，这完全不同于用户态应用对netlink的使用。如果用户需要增加新的 netlink协议类型，必须通过修改linux/netlink.h来实现，当然，目前的netlink实现已经包含了一个通用的协议类型 NETLINK\_GENERIC以方便用户使用，用户可以直接使用它而不必增加新的协议类型。前面讲到，为了增加新的netlink协议类型，用户仅需增 加如下定义到linux/netlink.h就可以：

#define NETLINK\_MYTEST 17

只要增加这个定义之后，用户就可以在内核的任何地方引用该协议。

在内核中，为了创建一个netlink socket用户需要调用如下函数：

struct sock \*

netlink\_kernel\_create(int unit, void (\*input)(struct sock \*sk, int len));

参数unit表示netlink协议类型，如NETLINK\_MYTEST，参数input则为内核模块定义的netlink消息处理函数，当有消 息到达这个netlink socket时，该input函数指针就会被引用。函数指针input的参数sk实际上就是函数netlink\_kernel\_create返回的 struct sock指针，sock实际是socket的一个内核表示数据结构，用户态应用创建的socket在内核中也会有一个struct sock结构来表示。下面是一个input函数的示例：

void input (struct sock \*sk, int len)

{

struct sk\_buff \*skb;

struct nlmsghdr \*nlh = NULL;

u8 \*data = NULL;

while ((skb = skb\_dequeue(&sk->receive\_queue)) != NULL)

{

/\* process netlink message pointed by skb->data \*/

nlh = (struct nlmsghdr \*)skb->data;

data = NLMSG\_DATA(nlh);

/\* process netlink message with header pointed by

\* nlh and data pointed by data

\*/

}

}

函数input()会在发送进程执行sendmsg()时被调用，这样处理消息比较及时，但是，如果消息特别长时，这样处理将增加系统调用 sendmsg()的执行时间，对于这种情况，可以定义一个内核线程专门负责消息接收，而函数input的工作只是唤醒该内核线程，这样sendmsg将 很快返回。

函数skb = skb\_dequeue(&sk->receive\_queue)用于取得socket sk的接收队列上的消息，返回为一个struct sk\_buff的结构，skb->data指向实际的netlink消息。

函数skb\_recv\_datagram(nl\_sk)也用于在netlink socket nl\_sk上接收消息，与skb\_dequeue的不同指出是，如果socket的接收队列上没有消息，它将导致调用进程睡眠在等待队列nl\_sk- >sk\_sleep，因此它必须在进程上下文使用，刚才讲的内核线程就可以采用这种方式来接收消息。

下面的函数input就是这种使用的示例：

void input (struct sock \*sk, int len)  
{  
 wake\_up\_interruptible(sk->sk\_sleep);  
}

当内核中发送netlink消息时，也需要设置目标地址与源地址，而且内核中消息是通过struct sk\_buff来管理的， linux/netlink.h中定义了一个宏：

#define NETLINK\_CB(skb)         (\*(struct netlink\_skb\_parms\*)&((skb)->cb))

来方便消息的地址设置。下面是一个消息地址设置的例子：

NETLINK\_CB(skb).pid = 0;  
NETLINK\_CB(skb).dst\_pid = 0;  
NETLINK\_CB(skb).dst\_group = 1;

字段pid表示消息发送者进程ID，也即源地址，对于内核，它为 0， dst\_pid 表示消息接收者进程 ID，也即目标地址，如果目标为组或内核，它设置为 0，否则 dst\_group 表示目标组地址，如果它目标为某一进程或内核，dst\_group 应当设置为 0。

在内核中，模块调用函数 netlink\_unicast 来发送单播消息：

int netlink\_unicast(struct sock \*sk, struct sk\_buff \*skb, u32 pid, int nonblock);

参数sk为函数netlink\_kernel\_create()返回的socket，参数skb存放消息，它的data字段指向要发送的 netlink消息结构，而skb的控制块保存了消息的地址信息，前面的宏NETLINK\_CB(skb)就用于方便设置该控制块， 参数pid为接收消息进程的pid，参数nonblock表示该函数是否为非阻塞，如果为1，该函数将在没有接收缓存可利用时立即返回，而如果为0，该函 数在没有接收缓存可利用时睡眠。

内核模块或子系统也可以使用函数netlink\_broadcast来发送广播消息：

void netlink\_broadcast(struct sock \*sk, struct sk\_buff \*skb, u32 pid, u32 group, int allocation);

前面的三个参数与netlink\_unicast相同，参数group为接收消息的多播组，该参数的每一个代表一个多播组，因此如果发送给多个多播组，就把该参数设置为多个多播组组ID的位或。参数allocation为内核内存分配类型，一般地为GFP\_ATOMIC或GFP\_KERNEL， GFP\_ATOMIC用于原子的上下文（即不可以睡眠），而GFP\_KERNEL用于非原子上下文。

在内核中使用函数sock\_release来释放函数netlink\_kernel\_create()创建的netlink socket：

void sock\_release(struct socket \* sock);

注意函数netlink\_kernel\_create()返回的类型为struct sock，因此函数sock\_release应该这种调用：

sock\_release(sk->sk\_socket);

sk为函数netlink\_kernel\_create()的返回值。

在源代码包中 给出了一个使用 netlink 的示例，它包括一个内核模块 netlink-exam-kern.c 和两个应用程序 netlink-exam-user-recv.c, netlink-exam-user-send.c。内核模块必须先插入到内核，然后在一个终端上运行用户态接收程序，在另一个终端上运行用户态发送程序，发送程序读取参数指定的文本文件并把它作为 netlink 消息的内容发送给内核模块，内核模块接受该消息保存到内核缓存中，它也通过proc接口出口到 procfs，因此用户也能够通过 /proc/netlink\_exam\_buffer 看到全部的内容，同时内核也把该消息发送给用户态接收程序，用户态接收程序将把接收到的内容输出到屏幕上。

示例：

内核模块 netlink-exam-kern.c：

//kernel module: netlink-exam-kern.c

#include <linux/config.h>

#include <linux/module.h>

#include <linux/netlink.h>

#include <linux/sched.h>

#include <net/sock.h>

#include <linux/proc\_fs.h>

#define BUF\_SIZE 16384

static struct sock \*netlink\_exam\_sock;

static unsigned char buffer[BUF\_SIZE];

static unsigned int buffer\_tail = 0;

static int exit\_flag = 0;

static DECLARE\_COMPLETION(exit\_completion);

static void recv\_handler(struct sock \* sk, int length)

{

wake\_up(sk->sk\_sleep);

}

static int process\_message\_thread(void \* data)

{

struct sk\_buff \* skb = NULL;

struct nlmsghdr \* nlhdr = NULL;

int len;

DEFINE\_WAIT(wait);

daemonize("mynetlink");

while (exit\_flag == 0) {

prepare\_to\_wait(netlink\_exam\_sock->sk\_sleep, &wait, TASK\_INTERRUPTIBLE);

schedule();

finish\_wait(netlink\_exam\_sock->sk\_sleep, &wait);

while ((skb = skb\_dequeue(&netlink\_exam\_sock->sk\_receive\_queue))

!= NULL) {

nlhdr = (struct nlmsghdr \*)skb->data;

if (nlhdr->nlmsg\_len < sizeof(struct nlmsghdr)) {

printk("Corrupt netlink message.\n");

continue;

}

len = nlhdr->nlmsg\_len - NLMSG\_LENGTH(0);

if (len + buffer\_tail > BUF\_SIZE) {

printk("netlink buffer is full.\n");

}

else {

memcpy(buffer + buffer\_tail, NLMSG\_DATA(nlhdr), len);

buffer\_tail += len;

}

nlhdr->nlmsg\_pid = 0;

nlhdr->nlmsg\_flags = 0;

NETLINK\_CB(skb).pid = 0;

NETLINK\_CB(skb).dst\_pid = 0;

NETLINK\_CB(skb).dst\_group = 1;

netlink\_broadcast(netlink\_exam\_sock, skb, 0, 1, GFP\_KERNEL);

}

}

complete(&exit\_completion);

return 0;

}

static int netlink\_exam\_readproc(char \*page, char \*\*start, off\_t off,

int count, int \*eof, void \*data)

{

int len;

if (off >= buffer\_tail) {

\* eof = 1;

return 0;

}

else {

len = count;

if (count > PAGE\_SIZE) {

len = PAGE\_SIZE;

}

if (len > buffer\_tail - off) {

len = buffer\_tail - off;

}

memcpy(page, buffer + off, len);

\*start = page;

return len;

}

}

static int \_\_init netlink\_exam\_init(void)

{

netlink\_exam\_sock = netlink\_kernel\_create(NETLINK\_GENERIC, 0, recv\_handler, THIS\_MODULE);

if (!netlink\_exam\_sock) {

printk("Fail to create netlink socket.\n");

return 1;

}

kernel\_thread(process\_message\_thread, NULL, CLONE\_KERNEL);

create\_proc\_read\_entry("netlink\_exam\_buffer", 0444, NULL, netlink\_exam\_readproc, 0);

return 0;

}

static void \_\_exit netlink\_exam\_exit(void)

{

exit\_flag = 1;

wake\_up(netlink\_exam\_sock->sk\_sleep);

wait\_for\_completion(&exit\_completion);

sock\_release(netlink\_exam\_sock->sk\_socket);

}

module\_init(netlink\_exam\_init);

module\_exit(netlink\_exam\_exit);

MODULE\_LICENSE("GPL");

//application sender: netlink-exam-user-send.c

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <linux/netlink.h>

#define MAX\_MSGSIZE 1024

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

{

FILE \* fp;

struct sockaddr\_nl saddr, daddr;

struct nlmsghdr \*nlhdr = NULL;

struct msghdr msg;

struct iovec iov;

int sd;

char text\_line[MAX\_MSGSIZE];

int ret = -1;

if (argc < 2) {

printf("Usage: %s atextfilename\n", argv[0]);

exit(1);

}

if ((fp = fopen(argv[1], "r")) == NULL) {

printf("File %s dosen't exist.\n");

exit(1);

}

sd = socket(AF\_NETLINK, SOCK\_RAW,NETLINK\_GENERIC);

memset(&saddr, 0, sizeof(saddr));

memset(&daddr, 0, sizeof(daddr));

saddr.nl\_family = AF\_NETLINK;

saddr.nl\_pid = getpid();

saddr.nl\_groups = 0;

bind(sd, (struct sockaddr\*)&saddr, sizeof(saddr));

daddr.nl\_family = AF\_NETLINK;

daddr.nl\_pid = 0;

daddr.nl\_groups = 0;

nlhdr = (struct nlmsghdr \*)malloc(NLMSG\_SPACE(MAX\_MSGSIZE));

while (fgets(text\_line, MAX\_MSGSIZE, fp)) {

memcpy(NLMSG\_DATA(nlhdr), text\_line, strlen(text\_line));

memset(&msg, 0 ,sizeof(struct msghdr));

nlhdr->nlmsg\_len = NLMSG\_LENGTH(strlen(text\_line));

nlhdr->nlmsg\_pid = getpid(); /\* self pid \*/

nlhdr->nlmsg\_flags = 0;

iov.iov\_base = (void \*)nlhdr;

iov.iov\_len = nlhdr->nlmsg\_len;

msg.msg\_name = (void \*)&daddr;

msg.msg\_namelen = sizeof(daddr);

msg.msg\_iov = &iov;

msg.msg\_iovlen = 1;

ret = sendmsg(sd, &msg, 0);

if (ret == -1) {

perror("sendmsg error:");

}

}

close(sd);

}

//application receiver: netlink-exam-user-recv.c

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <linux/netlink.h>

#define MAX\_MSGSIZE 1024

int main(void)

{

struct sockaddr\_nl saddr, daddr;

struct nlmsghdr \*nlhdr = NULL;

struct msghdr msg;

struct iovec iov;

int sd;

int ret = 1;

sd = socket(AF\_NETLINK, SOCK\_RAW,NETLINK\_GENERIC);

memset(&saddr, 0, sizeof(saddr));

memset(&daddr, 0, sizeof(daddr));

saddr.nl\_family = AF\_NETLINK;

saddr.nl\_pid = getpid();

saddr.nl\_groups = 1;

bind(sd, (struct sockaddr\*)&saddr, sizeof(saddr));

nlhdr = (struct nlmsghdr \*)malloc(NLMSG\_SPACE(MAX\_MSGSIZE));

while (1) {

memset(nlhdr, 0, NLMSG\_SPACE(MAX\_MSGSIZE));

iov.iov\_base = (void \*)nlhdr;

iov.iov\_len = NLMSG\_SPACE(MAX\_MSGSIZE);

msg.msg\_name = (void \*)&daddr;

msg.msg\_namelen = sizeof(daddr);

msg.msg\_iov = &iov;

msg.msg\_iovlen = 1;

ret = recvmsg(sd, &msg, 0);

if (ret == 0) {

printf("Exit.\n");

exit(0);

}

else if (ret == -1) {

perror("recvmsg:");

exit(1);

}

printf("%s", NLMSG\_DATA(nlhdr));

}

close(sd);

}