**原文：**[**http://blog.csdn.net/haomcu/article/details/7371835**](http://blog.csdn.net/haomcu/article/details/7371835)

**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。

**创建用户态netlink socket**

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。

**bind**

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：

1. **struct** sockaddr\_nl {
2. sa\_family\_t nl\_family;
3. unsigned **short** nl\_pad;
4. \_\_u32 nl\_pid;
5. \_\_u32 nl\_groups;
6. };

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

**struct msghdr**

结构 struct msghdr 需如下设置：

1. **struct** msghdr msg;
2. memset(&msg, 0, **sizeof**(msg));
3. msg.msg\_name = (**void** \*)&(nladdr);
4. msg.msg\_namelen = **sizeof**(nladdr);

其中 nladdr 为消息接收者的 netlink 地址。

**struct nlmsghdr**

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。

1. **struct** nlmsghdr {
2. \_\_u32 nlmsg\_len;
3. \_\_u16 nlmsg\_type;
4. \_\_u16 nlmsg\_flags;
5. \_\_u32 nlmsg\_seq;
6. \_\_u32 nlmsg\_pid;
7. };

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：

1. #define NLM\_F\_REQUEST 1
2. #define NLM\_F\_MULTI     2
3. #define NLM\_F\_ACK        4
4. #define NLM\_F\_ECHO      8
5. #define NLM\_F\_ROOT     0x100
6. #define NLM\_F\_MATCH    0x200
7. #define NLM\_F\_ATOMIC  0x400
8. #define NLM\_F\_DUMP      (NLM\_F\_ROOT|NLM\_F\_MATCH)
9. #define NLM\_F\_REPLACE  0x100
10. #define NLM\_F\_EXCL       0x200
11. #define NLM\_F\_CREATE   0x400
12. #define NLM\_F\_APPEND   0x800

标志NLM\_F\_REQUEST用于表示消息是一个请求，所有应用首先发起的消息都应设置该标志。

标志NLM\_F\_MULTI 用于指示该消息是一个多部分消息的一部分，后续的消息可以通过宏NLMSG\_NEXT来获得。

宏NLM\_F\_ACK表示该消息是前一个请求消息的响应，顺序号与进程ID可以把请求与响应关联起来。

标志NLM\_F\_ECHO表示该消息是相关的一个包的回传。

标志NLM\_F\_ROOT 被许多 netlink 协议的各种数据获取操作使用，该标志指示被请求的数据表应当整体返回用户应用，而不是一个条目一个条目地返回。有该标志的请求通常导致响应消息设置NLM\_F\_MULTI标志。注意，当设置了该标志时，请求是协议特定的，因此，需要在字段 nlmsg\_type 中指定协议类型。

标志 NLM\_F\_MATCH 表示该协议特定的请求只需要一个数据子集，数据子集由指定的协议特定的过滤器来匹配。

标志 NLM\_F\_ATOMIC 指示请求返回的数据应当原子地收集，这预防数据在获取期间被修改。

标志 NLM\_F\_DUMP 未实现。

标志 NLM\_F\_REPLACE 用于取代在数据表中的现有条目。

标志 NLM\_F\_EXCL\_ 用于和 CREATE 和 APPEND 配合使用，如果条目已经存在，将失败。

标志 NLM\_F\_CREATE 指示应当在指定的表中创建一个条目。

标志 NLM\_F\_APPEND 指示在表末尾添加新的条目。

内核需要读取和修改这些标志，对于一般的使用，用户把它设置为 0 就可以，只是一些高级应用（如 netfilter 和路由 daemon 需要它进行一些复杂的操作）

字段 nlmsg\_seq 和 nlmsg\_pid 用于应用追踪消息，前者表示顺序号，后者为消息来源进程 ID。下面是一个示例：

1. #define MAX\_MSGSIZE 1024
2. **char** buffer[] = "An example message";
3. **struct** nlmsghdr nlhdr;
4. nlhdr = (**struct** nlmsghdr \*)malloc(NLMSG\_SPACE(MAX\_MSGSIZE));
5. strcpy(NLMSG\_DATA(nlhdr),buffer);
6. nlhdr->nlmsg\_len = NLMSG\_LENGTH(strlen(buffer));
7. nlhdr->nlmsg\_pid = getpid();
8. nlhdr->nlmsg\_flags = 0;

**struct iovec**

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：

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

**发送消息 接受消息**

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。

1. #define MAX\_NL\_MSG\_LEN 1024
2. **struct** sockaddr\_nl nladdr;
3. **struct** msghdr msg;
4. **struct** iovec iov;
5. **struct** nlmsghdr \* nlhdr;
6. nlhdr = (**struct** nlmsghdr \*)malloc(MAX\_NL\_MSG\_LEN);
7. iov.iov\_base = (**void** \*)nlhdr;
8. iov.iov\_len = MAX\_NL\_MSG\_LEN;
9. msg.msg\_name = (**void** \*)&(nladdr);
10. msg.msg\_namelen = **sizeof**(nladdr);
11. msg.msg\_iov = &iov;
12. msg.msg\_iovlen = 1;
13. 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  #define NLMSG\_ALIGN(len) ( ((len)+NLMSG\_ALIGNTO-1) & ~(NLMSG\_ALIGNTO-1) ) |

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

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

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\_SPACE(len) NLMSG\_ALIGN(NLMSG\_LENGTH(len)) |

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

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\_NEXT(nlh,len) ((len) -= NLMSG\_ALIGN((nlh)->nlmsg\_len), \  (struct nlmsghdr\*)(((char\*)(nlh)) + NLMSG\_ALIGN((nlh)->nlmsg\_len))) |

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\_OK(nlh,len) ((len) >= (int)sizeof(struct nlmsghdr) && \  (nlh)->nlmsg\_len >= sizeof(struct nlmsghdr) && \  (nlh)->nlmsg\_len <= (len)) |

宏NLMSG\_OK(nlh,len)用于判断消息是否有len这么长。

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

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

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。

**创建netlink socket和 接受消息**

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

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**](http://lib.csdn.net/base/datastructure)，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) {  nlh = (struct nlmsghdr \*)skb->data;  data = NLMSG\_DATA(nlh);  }  } |

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。

**内核发送消息**

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。

**释放netlink socket**

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。

原文地址 <http://blog.csdn.net/zcabcd123/article/details/8275891>

所有socket之间的通信，必须有个地址结构，Netlink也不例外。我们最熟悉的就是IPV4的地址了，netlink的地址结构如下：

1. **struct** sockaddr\_nl
2. {
3. sa\_family\_t nl\_family;          //必须为AF\_NETLINK或者PF\_NETLINK
4. unsigned **short**  nl\_pad;             //必须为0
5. \_\_u32       nl\_pid;             //通信端口
6. \_\_u32       nl\_groups;              //组播掩码
7. };

上面几个数据，最关键的是nl\_family（就对应IP通信中的AF\_INET）和nl\_pid。

nl\_pid就是一个约定的通信端口，用户态使用的时候需要用一个非0的数字，一般来说可以直接采用上层应用的进程ID（不用进程ID号码也没事，只要系统中不冲突的一个数字即可使用）。对于内核的地址，该值必须用0

nl\_groups用于一个消息同时分发给不同的接收者，是一种组播应用，本文不讲组播应用。

本质上，nl\_pid就是netlink的通信地址。除了通信地址，netlink还提供“协议”来标示通信实体，在创建socket的时候，需要指定netlink的通信协议号。每个协议号代表一种“应用”，上层可以用内核已经定义的协议和内核进行通信，获得内核已经提供的信息。具体支持的协议列表如下：

1. #define NETLINK\_ROUTE       0   /\* Routing/device hook              \*/
2. #define NETLINK\_UNUSED      1   /\* Unused number                \*/
3. #define NETLINK\_USERSOCK    2   /\* Reserved for user mode socket protocols  \*/
4. #define NETLINK\_FIREWALL    3   /\* Firewalling hook             \*/
5. #define NETLINK\_INET\_DIAG   4   /\* INET socket monitoring           \*/
6. #define NETLINK\_NFLOG       5   /\* netfilter/iptables ULOG \*/
7. #define NETLINK\_XFRM        6   /\* ipsec \*/
8. #define NETLINK\_SELINUX     7   /\* SELinux event notifications \*/
9. #define NETLINK\_ISCSI       8   /\* Open-iSCSI \*/
10. #define NETLINK\_AUDIT       9   /\* auditing \*/
11. #define NETLINK\_FIB\_LOOKUP  10
12. #define NETLINK\_CONNECTOR   11
13. #define NETLINK\_NETFILTER   12  /\* netfilter subsystem \*/
14. #define NETLINK\_IP6\_FW      13
15. #define NETLINK\_DNRTMSG     14  /\* DECnet routing messages \*/
16. #define NETLINK\_KOBJECT\_UEVENT  15  /\* Kernel messages to userspace \*/
17. #define NETLINK\_GENERIC     16
18. /\* leave room for NETLINK\_DM (DM Events) \*/
19. #define NETLINK\_SCSITRANSPORT   18  /\* SCSI Transports \*/
20. #define NETLINK\_ECRYPTFS    19

协议的用途很好理解，比如我们单纯创建一个上层应用，通过和NETLINK\_ROUTE协议通信，可以获得内核的路由信息。我需要利用netlink创建一个我自己的通信协议，因此我定义了一种新的协议。新协议的定义不能和内核已经定义的冲突，同时不能超过MAX\_LINKS这个宏的限定，MAX\_LINKS = 32。所以我定义的协议号为30。

**用户态操作netlink socket**

用户态创建netlink socket的基本过程和操作其他socket的API一模一样，区别就2点：  
1、 netlink有自己的地址；  
2、 netlink接收到的消息带一个netlink自己的消息头；

**用户态创建、销毁socket的过程：**  
1、 用socket函数创建，socket(PF\_NETLINK, SOCK\_DGRAM, NETLINK\_XXX)；第一个参数必须是PF\_NETLINK或者AF\_NETLINK，第二个参数用SOCK\_DGRAM和SOCK\_RAW都没问题，第三个参数就是netlink的协议号。  
2、 用bind函数绑定自己的地址。  
3、 用close关闭套接字。

1. **struct** sockaddr\_nl nladdr;
2. int s\_nlm\_socket;
3. //建立netlink socket
4. s\_nlm\_socket = socket(PF\_NETLINK, SOCK\_DGRAM, NETLINK\_XXX);
6. //bind
7. nladdr.nl\_family = PF\_NETLINK;
8. nladdr.nl\_pad    = 0;
9. nladdr.nl\_pid    = getpid();
10. nladdr.nl\_groups = 0;
12. bind(s\_nlm\_socket, (**struct** sockaddr\*)&nladdr, **sizeof**(nladdr));

**用户态发送消息：**

1. **struct** msghdr msg;
2. memset(&msg, 0, **sizeof**(msg));
3. msg.msg\_name = (**void** \*)&(nladdr);
4. msg.msg\_namelen = **sizeof**(nladdr);
5. #define MAX\_MSGSIZE 1024
6. **char** buffer[] = "An example message";
7. **struct** nlmsghdr nlhdr;
8. nlhdr = (**struct** nlmsghdr \*)malloc(NLMSG\_SPACE(MAX\_MSGSIZE));
9. strcpy(NLMSG\_DATA(nlhdr),buffer);
10. nlhdr->nlmsg\_len = NLMSG\_LENGTH(strlen(buffer));
11. nlhdr->nlmsg\_pid = getpid();
12. nlhdr->nlmsg\_flags = 0;
13. **struct** iovec iov;
14. iov.iov\_base = (**void** \*)nlhdr;
15. iov.iov\_len = nlh->nlmsg\_len;
16. msg.msg\_iov = &iov;
17. msg.msg\_iovlen = 1;
18. sendmsg(fd,&msg,0);

**用户态接受消息：**

1. #define MAX\_NL\_MSG\_LEN 1024
2. **struct** sockaddr\_nl nladdr;
3. **struct** msghdr msg;
4. **struct** iovec iov;
5. **struct** nlmsghdr \* nlhdr;
6. nlhdr = (**struct** nlmsghdr \*)malloc(MAX\_NL\_MSG\_LEN);
7. iov.iov\_base = (**void** \*)nlhdr;
8. iov.iov\_len = MAX\_NL\_MSG\_LEN;
9. msg.msg\_name = (**void** \*)&(nladdr);
10. msg.msg\_namelen = **sizeof**(nladdr);
11. msg.msg\_iov = &iov;
12. msg.msg\_iovlen = 1;
13. recvmsg(fd, &msg, 0);

[Netlink实现热拔插监控](http://blog.chinaunix.net/uid-24943863-id-3223000.html) 2012-05-26 10:11:56

分类： LINUX

    新的Linux内核使用udev代替了hotplug作为热拔插管理，虽然有udevd管理热拔插，但有时候我们还是需要在应用程序中检测热拔插事件以便快速地处理，比如在读写SD卡的时候拔下SD卡，那么需要立即检测出该情况，然后结束读写线程，防止VFS崩溃。Netlink是面向数据包的服务，为内核与用户层搭建了一个高速通道，是udev实现的基础。该工作方式是异步的，用户空间程序不必使用轮询等技术来检测热拔插事件。

    内核中使用uevent事件通知用户空间，uevent首先在内核中调用netlink\_kernel\_create()函数创建一个socket套接字,该函数原型在netlink.h有定义，其类型是表示往用户空间发送消息的NETLINK\_KOBJECT\_UEVENT,groups=1，由于uevent只往用户空间发送消息而不接受，因此其输入回调函数input和cb\_mutex都设置为NULL。

#include

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);

ue\_sk->sk = netlink\_kernel\_create(net, NETLINK\_KOBJECT\_UEVENT, 1, NULL, NULL, THIS\_MODULE);

当有事件发生的时候，调用 kobject\_uevent()函数，实际上最终是调用

netlink\_broadcast\_filtered(uevent\_sock, skb , 0, 1, GFP\_KERNEL ,

                                        kobj\_bcast\_filter, kobj);

完成广播任务。

    用户空间程序只需要创建一个socket描述符，将描述符绑定到接收地址，就可以实现热拔插事件的监听了。

**点击(此处)折叠或打开**

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <errno.h>
5. #include <sys/types.h>
6. #include <asm/types.h>
7. //该头文件需要放在netlink.h前面防止编译出现\_\_kernel\_sa\_family未定义
8. #include <sys/socket.h>
9. #include <linux/netlink.h>
10. void MonitorNetlinkUevent()
11. {
12. int sockfd;
13. struct sockaddr\_nl sa;
14. int len;
15. char buf[4096];
16. struct iovec iov;
17. struct msghdr msg;
18. int i;
19. memset(&sa,0,sizeof(sa));
20. sa.nl\_family=**AF\_NETLINK;**
21. sa.nl\_groups=**NETLINK\_KOBJECT\_UEVENT;**
22. sa.nl\_pid = 0;//getpid(); both is ok
23. memset(&msg,0,sizeof(msg));
24. iov.iov\_base=(void \*)buf;
25. iov.iov\_len=sizeof(buf);
26. msg.msg\_name=(void \*)&sa;
27. msg.msg\_namelen=sizeof(sa);
28. msg.msg\_iov=&iov;
29. msg.msg\_iovlen=1;
30. sockfd=socket(**AF\_NETLINK,SOCK\_RAW,NETLINK\_KOBJECT\_UEVENT**);
31. if(sockfd==-1)
32. printf("socket creating failed:%s\n",strerror(errno));
33. if(bind(sockfd,(struct sockaddr \*)&sa,sizeof(sa))==-1)
34. printf("bind error:%s\n",strerror(errno));
35. len=recvmsg(sockfd,&msg,0);
36. if(len<0)
37. printf("receive error\n");
38. else if(len<32||len>sizeof(buf))
39. printf("invalid message");
40. for(i=0;i<len;i++)
41. if(\*(buf+i)=='\0')
42. buf[i]='\n';
43. printf("received %d bytes\n%s\n",len,buf);
44. }
45. int main(int argc,char \*\*argv)
46. {
47. MonitorNetlinkUevent();
48. return 0;
49. }

创建socket描述符的时候指定协议族为AF\_NETLINK或者PF\_NETLINK,套接字type选择SOCK\_RAW或者SOCK\_DGRAM，Netlink协议并不区分这两种类型，第三个参数协议填充NETLINK\_KOBJECT\_UEVENT表示接收内核uevent信息。接着就绑定该文件描述

符到sockadd\_nl,注意该结构体nl\_groups是接收掩码，取～0是将接收所有来自内核的消息，我们接收热拔插只需要填NETLINK\_KOBJECT\_UEVENT即可。接下来调用recvmsg开始接收内核消息，recvmsg函数需要我们填充message报头，包括指定接收缓存等工作。该函数会阻塞直到有热拔插事件产生。

运行程序，然后我插入一个U盘，得到下面的结果：

1. $ ./netlink
2. received 289 bytes
3. add@/devices/pci0000:00/0000:00:1a.0/usb1/1-1/1-1.1
4. ACTION=add
5. DEVPATH=/devices/pci0000:00/0000:00:1a.0/usb1/1-1/1-1.1
6. SUBSYSTEM=usb
7. MAJOR=189
8. MINOR=8
9. DEVNAME=bus/usb/001/009
10. DEVTYPE=usb\_device
11. DEVICE=/proc/bus/usb/001/009
12. PRODUCT=781/5530/100
13. TYPE=0/0/0
14. BUSNUM=001
15. DEVNUM=009
16. SEQNUM=2306
17. 运行程序，拔掉U盘
18. $ ./netlink
19. received 294 bytes
20. remove@/devices/pci0000:00/0000:00:1a.0/usb1/1-1/1-1.1/1-1.1:1.0/host10/target10:0:0/10:0:0:0/bsg/10:0:0:0
21. ACTION=remove
22. DEVPATH=/devices/pci0000:00/0000:00:1a.0/usb1/1-1/1-1.1/1-1.1:1.0/host10/target10:0:0/10:0:0:0/bsg/10:0:0:0
23. SUBSYSTEM=bsg
24. MAJOR=253
25. MINOR=2
26. DEVNAME=bsg/10:0:0:0
27. SEQNUM=2345

程序正确地接收到了U盘热拔插事件，通过该信息用户程序可以在第一时间得到事件通知。事实上热拔插的时候产生的消息可不止一条呢，可以在revmsg的时候用一个循环接收更多的消息。