自由天空

< 2014年8月 >

日	一	二	三	四	五	六
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

搜索

找找看

谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

文章分类(50)

LINUX netlink机制

Posted on 2009-11-01 22:42 放飞自我 阅读(13169) 评论(4) 编辑 收藏

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。

Linux 配置与应用(18)
Linux 设备驱动(11)
Linux 网络协议栈(3)
TCP / IP(2)
数据结构与算法(5)
数据库(1)
网络编程(10)

文章档案(50)
2012年8月 (4)
2010年7月 (2)
2010年1月 (8)
2009年12月 (6)
2009年11月 (30)

Linux 网络领域
epoll
Linux 网络驱动
Linux内核路由
linux网络kernel
Linux下ip命令手册
Linux中的通知链技术
netlink
netlink 编程介绍
visualsvn 使用
网络流量统计程序

Linux 网络应用
http cache
http cache

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：

--放飞自我

```
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           */
```

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：

```
#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
#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。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);
```

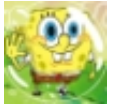
注意函数netlink_kernel_create()返回的类型为struct sock，因此函数sock_release应该这种调用：

```
   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

分类: Linux 网络协议栈



放飞自我
关注 - 7
粉丝 - 17
+加关注

Feedback

#1楼
2012-08-07 10:18 by forfunforlove

很有帮助，很详细。

支持(0) 反对(0)

#2楼
2012-10-22 10:51 by Johann

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 ...
请问这个缓存队列大小可以设置吗？ 我遇到的问题是内核发送过多消息，用户层来不及处理

支持(0) 反对(0)

#3楼
2014-03-27 13:28 by Tsihang

请问源码包怎么下载呢？

支持(0) 反对(0)

#4楼[楼主]
2014-03-27 14:09 by 放飞自我

@Tsihang
很少来这里了，直接下载linux内核代码即可

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:

· 新华网宣布向云南地震灾区捐款100万元
· 研究人员发现新漏洞：多数手机都可被完全控制
· 星巴克提供了全美最快免费公共无线网络
· 多亏Sketch，我这个小码农可以自己设计App了
· 2014年最流行的应用服务器
» 更多新闻...

最新知识库文章:

· 父子页面之间跨域通信的方法
· Android开发在路上：少去踩坑，多走捷径
· 从用户行为打造活动交互设计闭环——2014年世界杯竞猜活动设计总结
· 如何通过一个问题，完成最成功的技术面试
· 我所理解的技术领导力
» 更多知识库文章...