原 Linux-内核通信之netlink机制-详解

2016年06月05日 19:07:30 阅读数：7651 更多

前言：

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。用户态和内核态的通讯机制IPC(interprocess communication)机制：比如系统调用，ioctl接口，proc文件系统以及netlink socket。

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。

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，内核调度的力度就会受到影响。

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。

Netlink Socket 的API

标准的socket API函数-socket(), sendmsg(), recvmsg()和close()；

1、使用socket()函数创建一个socket，输入：int socket(int domain, int type, int protocol)；

2、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;
```

3、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。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。

4、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。

5、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(sock_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。

1、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。
2、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。
3、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。
4、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);
```

5、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);

实例：

```
1 用户空间：
2 #include <sys/stat.h>
3 #include <unistd.h>
4 #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>
#include <errno.h>

#define NETLINK_TEST 25
#define MAX_PAYLOAD 1024 // maximum payload size

int sock_fd;
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
struct iovec iov;
struct msghdr msg;

void init_netlink()
{
        // Create a socket
        sock_fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_TEST);
        if(sock_fd == -1)
        {
                printf("error getting socket: %s", strerror(errno));
                return;
        }
        // To prepare binding
        memset(&msg,0,sizeof(msg));
        memset(&src_addr, 0, sizeof(src_addr));
        //src_address
        src_addr.nl_family = AF_NETLINK;
        src_addr.nl_pid = getpid(); // self pid
        src_addr.nl_groups = 0; // multi cast
        bind(sock_fd, (struct sockaddr*)&src_addr, sizeof(src_addr));

        //dest_address
        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 */

        /* Fill the netlink message header */
        nlh=(struct nlmsghdr *)malloc(NLMSG_SPACE(MAX_PAYLOAD));
        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), "connect to kernel");
        iov.iov_base = (void *)nlh;
        iov.iov_len = nlh->nlmsg_len;
        //iov.iov_len = NLMSG_SPACE(MAX_PAYLOAD);
        memset(&msg, 0, sizeof(msg));
        msg.msg_name = (void *)&dest_addr;
        msg.msg_namelen = sizeof(dest_addr);
        msg.msg_iov = &iov;
        msg.msg_iovlen = 1;
}

int main(int argc, char* argv[])
{
        int state;
        int state_smg = 0;

        init_netlink();
        printf(" Sending message. ...\n");
        state_smg = sendmsg(sock_fd,&msg,0);
        if(state_smg == -1)
        {
        printf("get error sendmsg = %s\n",strerror(errno));
        }

        memset(nlh,0,NLMSG_SPACE(MAX_PAYLOAD));
```

```
        printf(" Waiting message. ...\n");
                                        // Read message from kernel

        while(1)
        {
                printf("In while recvmsg\n");
                state = recvmsg(sock_fd, &msg, 0);
                if(state<0)
                {
                    printf("state<1");
                }
                printf("In while\n");
                printf("Received message: %s\n",(char *) NLMSG_DATA(nlh));
        }

        close(sock_fd);

        return 0;
}
```

内核空间：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/timer.h>
#include <linux/time.h>
#include <linux/types.h>
#include <net/sock.h>
#include <net/netlink.h>

#define NETLINK_TEST 25
#define MAX_MSGSIZE 1024

int pid;
int err;
struct sock *nl_sk = NULL;
int flag = 0;
char str[100];


int stringlength(char *s)
{
        int slen = 0;
        for(; *s; s++)
        {
                slen++;
        }
        return slen;
}
void send_netlink_data(char *message)
{
        struct sk_buff *skb_1;
        struct nlmsghdr *nlh;
        int len = NLMSG_SPACE(MAX_MSGSIZE);
        int slen = 0;

        skb_1 = alloc_skb(len,GFP_KERNEL);
        if(!skb_1)
        {
                printk(KERN_ERR "my_net_link:alloc_skb_1 error\n");
        }
        slen = stringlength(message);
        nlh = nlmsg_put(skb_1,0,0,0,MAX_MSGSIZE,0);

        NETLINK_CB(skb_1).pid = 0;
        NETLINK_CB(skb_1).dst_group = 0;

        memcpy(NLMSG_DATA(nlh),message,slen+1);
```

```
48
49          printk("my_net_link:send = %d, message '%s'.\n",slen,(char *)NLMSG_DATA(nlh));
50
51          netlink_unicast(nl_sk,skb_1,pid,MSG_DONTWAIT);
52  }
53  void recv_netlink_data(struct sk_buff *__skb)
54  {
55          struct sk_buff *skb;
56          struct nlmsghdr *nlh;
57          //struct completion cmpl;
58          int i=10;
59          printk("net_link: data is ready to read.\n");
60          skb = skb_get (__skb);
61          if(skb->len >= NLMSG_SPACE(0))
62          {
63                  nlh = nlmsg_hdr(skb);
64                  memcpy(str, NLMSG_DATA(nlh), sizeof(str));
65                  printk("Message received:%s\n",str) ;
66                  pid = nlh->nlmsg_pid;
67                  while(i--)
68                  {
69                          //init_completion(&cmpl);
70                          //wait_for_completion_timeout(&cmpl,1 * HZ);
71                          send_netlink_data("From kernel messages!");
72                  }
73                  flag = 1;
74                  kfree_skb(skb);
75          }
76  }
77
78
79  // Initialize netlink
80  int netlink_init(void)
81  {
82          nl_sk = netlink_kernel_create(&init_net, NETLINK_TEST, 1, recv_netlink_data, NULL, THIS_MODULE);
83          if(!nl_sk)
84          {
85                  printk(KERN_ERR "my_net_link: create netlink socket error.\n");
86                  return 1;
87          }
88          printk("my_net_link_3: create netlink socket ok.\n");
89          return 0;
90  }
91
92  static void netlink_exit(void)
93  {
94          if(nl_sk != NULL)
95          {
96                  sock_release(nl_sk->sk_socket);
97          }
98
99      printk("my_net_link: self module exited\n");
100 }
101
102 module_init(netlink_init);
103 module_exit(netlink_exit);
104
105 MODULE_AUTHOR("suntianyu");
106 MODULE_LICENSE("GPL");
```

http://www.360doc.com/content/13/0912/17/496343_314005658.shtml

linux netlink机制介绍与实例：http://blog.csdn.net/zcabcd123/article/details/8272656

Linux中与内核通信的Netlink机制（实例）：http://blog.chinaunix.net/uid-20788636-id-1841429.html

2.6.24以上内核中netlink使用方法：http://blog.csdn.net/wangjingfei/article/details/5288460

linux下epoll网络模型编程：http://www.360doc.com/content/14/0102/11/12892305_342001086.shtml