一、实验目的及实验内容 （本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析）	小题分：

实验目的：

熟悉数据链路层数据的获取方法，能够从数据链路层获取网络层、传输层和应用层的数据，掌握 ARP 协议。

实验要求：

1. 阅读课本第 10、11 章。
2. 了解数据链路层数据的获取方法，包括设置套接口以捕获链路帧的编程方法、从套接口读取链路帧的编程方法、定位 IP 包头的编程方法、定位 TCP 报头的编程方法、定位 UDP 报头的编程方法和定位应用层报文数据的编程方法；
3. 使用 SOCK_PACKET 编写 ARP 请求程序：
   了解 ARP 协议；
   使用发送 ARP 请求数据
   使用 ARP 命令查看 ARP 表并验证；

二、实验环境及实验步骤 （本次实验所使用的器件、仪器设备等的情况；具体的实验步骤）	小题分：

实验环境：

Ubuntu18.04

实验步骤：

下面将以 7 个步骤进行详细说明：

1. 设置套接口以捕获链路帧的编程方法
2. 从套接口读取链路帧的编程方法
3. 定位 IP 包头的编程方法
4. 定位 TCP 报头的编程方法
5. 定位 UDP 报头的编程方法
6. 定位应用层报文数据的编程方法
7. 使用 SOCK_PACKET 编写 ARP 请求程序

1. 设置套接口以捕获链路帧的编程方法

使用 socket 函数创建：

```
int fd;                                  /*fd是套接口的描述符*/
fd = socket(AF_INET, SOCK_PACKET, htons(0x0003));
```

这里要注意的是第 2 和 3 个参数。

第 1 个参数 AF_INET 表示因特网协议族，这个上个实验也用的这个，表示 ipv4 协议。

第 2 个参数 SOCK_PACKET 表示截取数据帧的层次在物理层，网络协议栈对数据不做处理。

第 3 个参数值 0x0003 表示截取的数据帧的类型为不确定，处理所有的包。

2. 从套接口读取链路帧的编程方法

这里从套接口读取的数据为链路层的帧，对于常用的以太网帧如下：

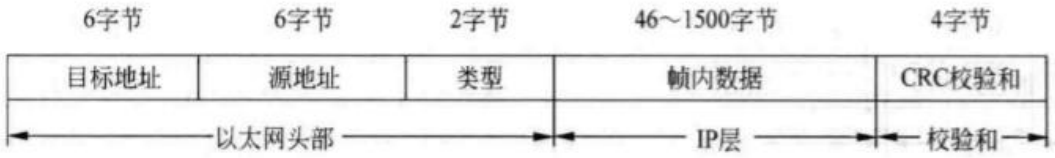


图 11-10　以太网帧示意图

在头文件<netinet/if ether.h>中定义了如下常量：

```
#define ETH_ALEN    6       /* Octets in one ethernet addr   */
#define ETH_TLEN    2       /* Octets in ethernet type field */
#define ETH_HLEN    14      /* Total octets in header.   */
#define ETH_ZLEN    60      /* Min. octets in frame sans FCS */
#define ETH_DATA_LEN    1500        /* Max. octets in payload    */
#define ETH_FRAME_LEN   1514        /* Max. octets in frame sans FCS */
```

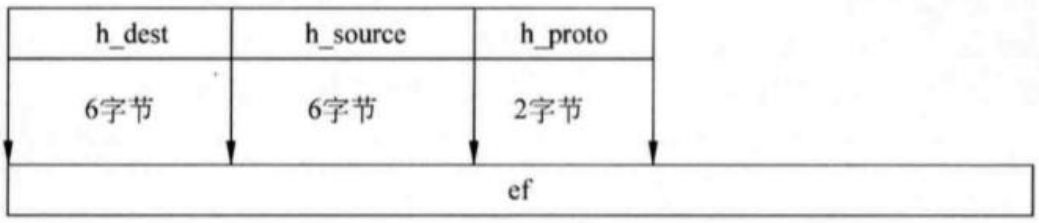
以太网头部结构定义为如下形式：

```
struct ethhdr {
    unsigned char   h_dest[ETH_ALEN];   /* destination eth addr */
    unsigned char   h_source[ETH_ALEN]; /* source ether addr    */
    __be16      h_proto;        /* packet type ID field */
} __attribute__((packed));
```

套接字文件描述符建立后，就可以从此描述符中读取数据，数据的格式为上述的以太网数据，即以太网帧。套接口建立以后，就可以从中循环读取捕获的链路层以太网帧。要建立一个大小为 ETH_FRAMELEN 的缓冲区，并将以太网的头部指向此缓冲区，例如：

```
char ef[ETH_FRAME_LEN];                    /*以太帧缓冲区*/
struct ethhdr*p_ethhdr;                    /*以太网头部指针*/
int n;
p_ethhdr = (struct ethhdr*)ef;             /*使 p_ethhdr 指向以太网帧的帧头*/
/*读取以太网数据,n 为返回的实际捕获的以太帧的帧长*/
n = read(fd, ef, ETH_FRAME_LEN);
```

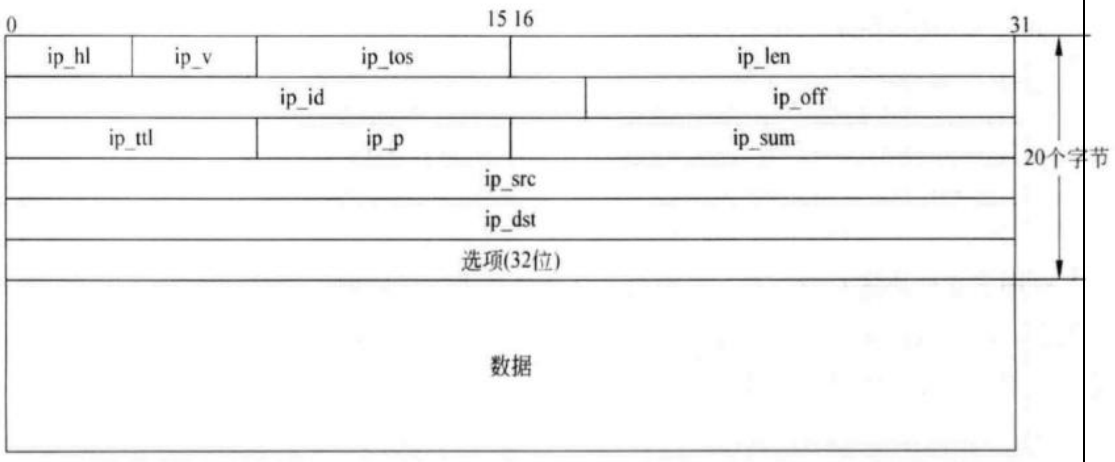
接收数据以后，缓冲区 ef 与以太网头部的对应关系如图：



因此，要获得以太网帧的目的 MAC 地址、源 MAC 地址和协议的类型，可以通过 p_ethhdr->h_dest、p_ethhdr->h_source 和 p_ethhdr->h proto 获得。

3. 定位 IP 包头的编程方法

获得以太网帧后，当协议为 0x0800 时，其负载部分为 IP 协议。IP 协议的数据结构如下：



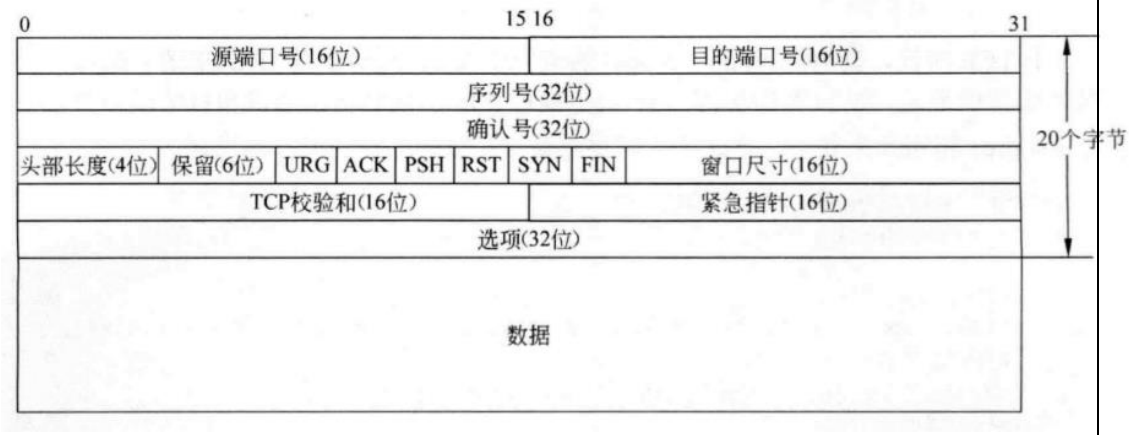
IP 头部的数据结构定义在头文件<netinet/ip.h>中，代码如下：

```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u8    ihl:4,
            version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
    __u8    version:4,
            ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
    __u8    tos;
    __be16  tot_len;
    __be16  id;
    __be16  frag_off;
    __u8    ttl;
    __u8    protocol;
    __sum16 check;
    __be32  saddr;
    __be32  daddr;
    /*The options start here. */
};
```

若捕获的以太帧中 hproto 的取值为 0x0800，将类型为 iphdr 的结构指针指向帧头后面载荷数据的起始位置，则可以得到 IP 数据包的报头部分。通过 saddr 和 daddr 可以得到 IP 报文的源 IP 地址和目的 IP 地址。

4. 定位 TCP 报头的编程方法

TCP 的数据结构如图：



对应的数据结构在头文件<netinet/tcp.h>中定义，代码如下：

```
25  struct tcphdr {
26      __be16  source;
27      __be16  dest;
28      __be32  seq;
29      __be32  ack_seq;
30  #if defined(__LITTLE_ENDIAN_BITFIELD)
31      __u16   res1:4,
32          doff:4,
33          fin:1,
34          syn:1,
35          rst:1,
36          psh:1,
37          ack:1,
38          urg:1,
39          ece:1,
40          cwr:1;
41  #elif defined(__BIG_ENDIAN_BITFIELD)
42      __u16   doff:4,
43          res1:4,
44          cwr:1,
45          ece:1,
46          urg:1,
47          ack:1,
48          psh:1,
49          rst:1,
50          syn:1,
51          fin:1;
52  #else
53  #error  "Adjust your <asm/byteorder.h> defines"
54  #endif
55      __be16  window;
56      __sum16 check;
57      __be16  urg_ptr;
58  };
59
60  /*
```

对于 TCP 协议，其 IP 头部的 protocol 的值应该为 6，通过计算 IP 头部的长度可以得到 TCP 头部的地址，即 TCP 的头部为 IP 头部偏移 ihl*4。TCP 的源端口和目的端口可以通过成员 source 和 dest 来获得。

5. 定位 UDP 报头的编程方法

UDP 的数据结构如图：

UDP 的头部数据结构在文件<netinet/udp.h>中定义，代码如下：

```
struct udphdr {
        __be16  source;
        __be16  dest;
        __be16  len;
        __sum16 check;
};
```

对于 UDP 协议，其 IP 头部的 protocol 的值为 17，通过计算 IP 头部的长度可以得到 UDP 头部的地址，即 UDP 的头部为 IP 头部偏移 ihl*4。UDP 的源端口和目的端口可以通过成员 source 和 dest 来获得。

6. 定位应用层报文数据的编程方法

定位了 UDP 和 TCP 头部地址后，其中的数据部分为应用层报文数据。根据 TCP 和 UDP 的协议获得应用程序指针的代码如下：

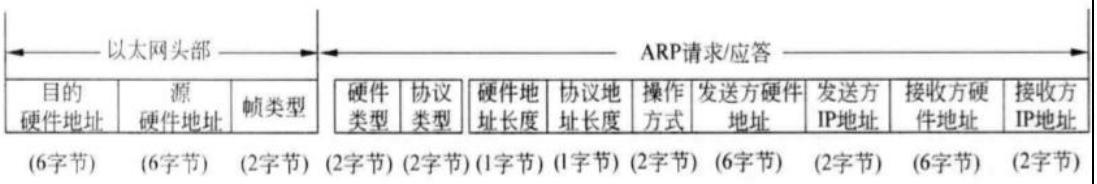
```
char*app_data = NULL;                              /*应用数据指针*/
int app_len = 0;                                   /*应用数据长度*/

/*获得 TCP 或者 UDP 的应用数据*/
if(p_iphdr->protocol==6)
{
    struct tcphdr*p_tcphdr = (struct tcphdr*)(p_iphdr+p_iphdr->ihl*4);
                                                   /*取得 TCP 报头*/
    app_data = p_tcphdr + 20;                      /*获得 TCP 协议部分的应用数据地址*/
    app_len = n - 16 - p_iphdr->ihl*4 - 20;/*获得 TCP 协议部分的应用数据长度*/
}else if(p_iphdr->protocol==17)
{
    struct udphdr*p_udphdr = (struct udphdr*)(p_iphdr+p_iphdr->ihl*4);
                                                   /*取得 UDP 报头*/
    app_data = p_udphdr + p_udphdr->len;           /*获得 UDP 协议部分的应用数据地址*/
    app_len = n - 16 - p_iphdr->ihl*4 - p_udphdr->len;
                                                   /*获得 UDP 协议部分的应用数据长度*/
}

printf("application data address:0x%x, length:%d\n",app_data,app_len);
                                                   /*打印应用数据的地址和长度*/
```

7. 使用 SOCK_PACKET 编写 ARP 请求程序

包含以太网头部数据的 ARP 协议数据结构如图：



arp 数据结构如下：

```
struct arphdr
{
    __be16      ar_hrd;                 /*硬件类型*/
    __be16      ar_pro;                 /*协议类型*/
    unsigned char   ar_hln;             /*硬件地址长度*/
    unsigned char   ar_pln;             /*协议地址长度*/
    __be16      ar_op;                  /*ARP 操作码*/
};
```

之后按照上面数据结构定义这个结构体如下：

```
struct arppacket
{
    struct arphdr ar_head;/*硬件类型、协议、地址长度、操作码*/
    unsigned char ar_sha[ETH_ALEN];/*发送方MAC*/
    struct in_addr ar_sip;/*发送方IP*/
    unsigned char ar_tha[ETH_ALEN];;/*目的MAC*/
    struct in_addr ar_tip;/*目的IP*/
}__attribute__((packed));
```

接下来就进入主函数了。

首先根据命令行的输入打印一些提示和错误信息：

```
    //输入错误，显示正确输入格式
    if(argc != 2)
    {
        perror("Usage: ./test xxx.xxx.xxx.xxx\n");
        exit(-1);
    }

    //参数错误，输入的不是ip地址
    if(inet_aton(argv[1], &pingaddr) < 0)
    {
        perror("not a correct ip address\n");
        exit(-1);
    }
```

接下来注册原始套接字：

```
// 注册原始套接字
int fd = socket(PF_PACKET, SOCK_RAW, htonl(0x0003));
```

各个参数在第 1 部分已经介绍过了。

接下来获得本地网卡信息：

```
// 获得网卡信息
char buf[1024];
struct ifconf ifc;
ifc.ifc_len = sizeof(buf);
ifc.ifc_buf = buf;
if(ioctl(fd, SIOCGIFCONF, &ifc) == -1)
{
    perror("get net interface error\n");
    exit(-1);
}
```

这里除了使用原始套接字外，还使用了 PF_PACKET 协议族，用于在链路层收发原始(raw)分组。所以，地址也不再是 sockaddr_in 而是采用 sockaddr_ll 地址，表示设备无关的物理层地址结构，如下：

```
struct sockaddr_ll {
        unsigned short  sll_family;   //这里使用PF_PACKET
        unsigned short  sll_protocol; //物理层协议
        int             sll_ifindex;  //接口号
        unsigned short  sll_hatype;   //报头类型
        unsigned char   sll_pkttype;  //分组类型
        unsigned char   sll_halen;    //地址长度
        unsigned char   sll_addr[8];  //物理层地址，即目的MAC地址
};
```

一开始以为使用 socket 发送数据需要绑定相关的端口和网址，所以就不断的尝试将获取到的相关地址写入 struct sockaddr_ll 中，到后面才发现 SOCK_RAW 模式下可以不需要绑定 MAC 地址，并且不需要其进行 IP 地址的相关操作，因

为网卡驱动程序接收到报文后会对自己组织的整个以太网数据帧进行处理，将它准确发送到目的地。所以我们只要确保自定义的数据帧准确无误就可以了。因为对于二层报文发送，没有根据目的地址进行选路的依据，所以发送者必须指定要使用的出接口，sockaddr_ll.sll_ifindex 就是指本地的网卡 index.

所以我们就要用到结构体 struct ifreq 和 ioctl()函数去获取本地的网卡 index。这里就要用到另外一个结构体 struct ifreq。

```
struct ifreq
{
#define IFHWADDRLEN 6
 union
 {
  char ifrn_name[IFNAMSIZ];
 } ifr_ifrn;

 union
 {
  struct sockaddr ifru_addr;
  struct sockaddr ifru_dstaddr;
  struct sockaddr ifru_broadaddr;
  struct sockaddr ifru_netmask;
  struct  sockaddr ifru_hwaddr;
  short ifru_flags;
  int ifru_ivalue;
  int ifru_mtu;
  struct  ifmap ifru_map;
  char ifru_slave[IFNAMSIZ];
  char ifru_newname[IFNAMSIZ];
  void __user * ifru_data;
  struct if_settings ifru_settings;
 } ifr_ifru;
};
```

如下：

```
for(i = 0; i < networkNum; i++)
{
    ifrPtr = ((struct ifreq*)buf) + i;
    if(ioctl(fd, SIOCGIFADDR, ifrPtr) == -1)
    {
        perror("get ip address error\n");
        exit(-1);
    }
    netaddr = ((struct sockaddr_in*)&(ifrPtr->ifr_addr))->sin_addr;

    if(ioctl(fd, SIOCGIFNETMASK, ifrPtr) == -1)
    {
        perror("get netmask error\n");
        exit(-1);
    }
    netmask = ((struct sockaddr_in*)&(ifrPtr->ifr_netmask))->sin_addr;
```

然后设置 sockaddr_ll 地址：

```
// 设置sockaddr_ll地址
if((pingaddr.s_addr & netmask.s_addr) == (netaddr.s_addr & netmask.s_addr))
{
    hwaddr.sll_family = PF_PACKET;
    hwaddr.sll_protocol = htons(ETH_P_ARP);
    hwaddr.sll_hatype = ARPHRD_ETHER;
    hwaddr.sll_pkttype = PACKET_OTHERHOST;
    hwaddr.sll_halen = ETH_ALEN;

    if(ioctl(fd, SIOCGIFINDEX, ifrPtr) == -1)
    {
        perror("get net interface index error\n");
        exit(-1);
    }
    hwaddr.sll_ifindex = ifrPtr->ifr_ifindex;

    if(ioctl(fd, SIOCGIFHWADDR, ifrPtr) == -1)
    {
        perror("get net interface hwaddr error\n");
        exit(-1);
    }
    memcpy(hwaddr.sll_addr, ifrPtr->ifr_hwaddr.sa_data, ETH_ALEN);

    flag = 1;
    break;
}
```

之后就可以构建 arp 请求包了：

```
    // 构建ARP请求包
    char ef[ETH_FRAME_LEN];
    //使p_eth指向以太网帧的帧头
    struct ethhdr *p_eth = (struct ethhdr*)ef;
    //目的以太网地址
    memset(p_eth->h_dest, 0xff, ETH_ALEN);
    //源以太网地址
    memcpy(p_eth->h_source, hwaddr.sll_addr, ETH_ALEN);
    //设置协议类型
    p_eth->h_proto = htons(ETH_P_ARP);

    //定位ARP包地址
    struct arppacket *p_arp = (struct arppacket*)(ef + ETH_HLEN);
    //硬件类型
    p_arp->ar_head.ar_hrd = htons(ARPHRD_ETHER);/*arp硬件类型*/
    p_arp->ar_head.ar_pro = htons(ETH_P_IP);    /*协议类型*/
    p_arp->ar_head.ar_hln = ETH_ALEN;           /*硬件地址长度*/
    p_arp->ar_head.ar_pln = 4;                   /*IP地址长度*/
    p_arp->ar_head.ar_op = htons(ARPOP_REQUEST);

    /*复制源以太网地址*/
    memcpy(p_arp->ar_sha, hwaddr.sll_addr, ETH_ALEN);
    /*源IP地址*/
    p_arp->ar_sip = netaddr;
    /*复制目的以太网地址*/
    memset(p_arp->ar_tha, 0, ETH_ALEN);
    //目的IP地址
    p_arp->ar_tip = pingaddr;
```

设置目的 MAC，全为 0xFF,表示在局域网进行广播。

之后根据网卡信息设置本机 MAC 地址。

之后根据网卡信息设置本机 MAC 地址。

然后设置协议类型。

定位 ARP 包地址。

下面都是填充 p_arp 结构，就是我们自己定义的结构体。

接下来绑定网卡：

```
//绑定网卡
if(bind(fd, (struct sockaddr*)&hwaddr, sizeof(struct sockaddr_ll)) == -1)
{
    perror("bind network error\n");
    exit(-1);
}
```

使用 write 函数发送：

```
//发送以太网帧
write(fd, ef, 60);
```

最后可以读取并打印出目的 MAC 地址：

```
//读取并打印目的MAC
while(1)
{
    read(fd, ef, sizeof(ef));
    struct arppacket *recv_arp = (struct arppacket*)(ef+ETH_HLEN);
    if(recv_arp->ar_tip.s_addr == netaddr.s_addr)
    {
        printf("Find %s 's mac:",argv[1]);
        for (i = 0; i < ETH_ALEN - 1; i++)
            printf("%02x-", recv_arp->ar_sha[i]);
        printf("%02x\n", recv_arp->ar_sha[ETH_ALEN - 1]);

        break;
    }
}
return 0;
```

整个过程就到此结束了。

三、实验过程分析

（详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格、绘制曲线、波形等）

小题分：

1. 书上源程序需要自行设置 MAC 地址，这里采用自动读取本地网卡信息进行填充 MAC 信息。

```
// 获得网卡信息
char buf[1024];
struct ifconf ifc;
ifc.ifc_len = sizeof(buf);
ifc.ifc_buf = buf;
if(ioctl(fd, SIOCGIFCONF, &ifc) == -1)
{
    perror("get net interface error\n");
    exit(-1);
}
```

2. 在执行程序时需要以 root 权限进行执行，否则会提示权限不够：

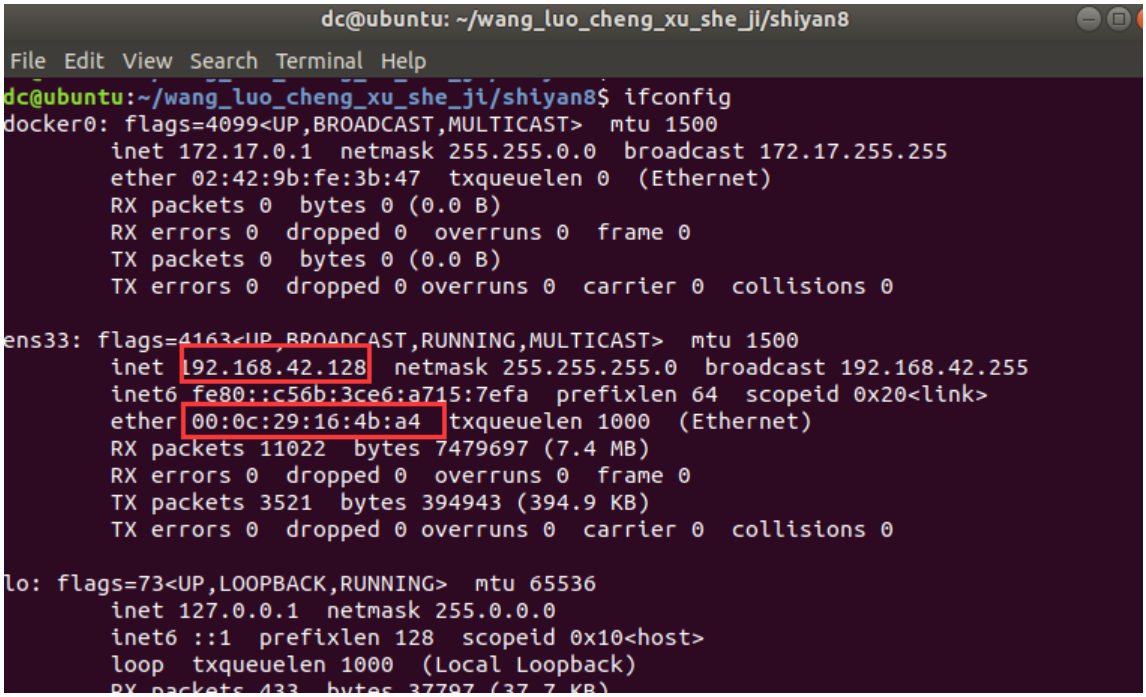
```
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan8$ ./test 192.168.42.1
create socket error
: Operation not permitted
```

四、实验结果总结

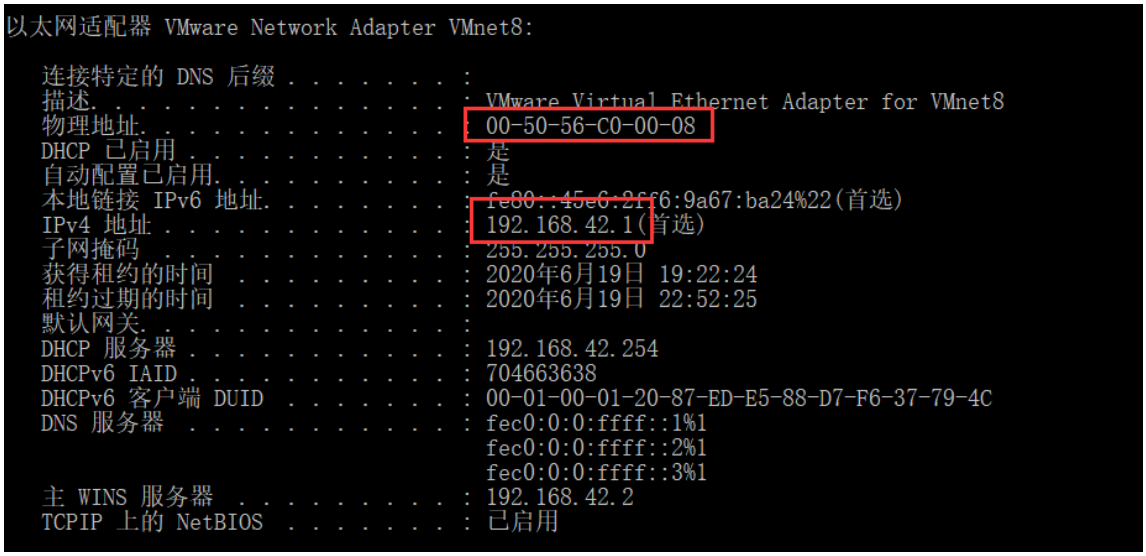
（对实验结果进行分析，完成思考题目，总结实验的新的体会，并提出实验的改进意见）

小题分：

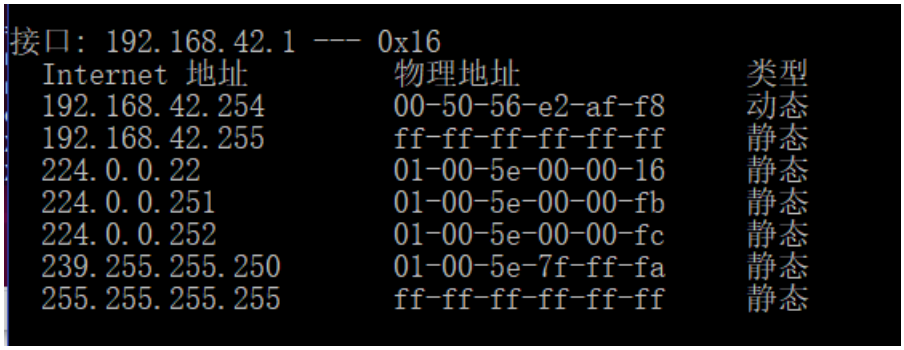
首先看一下虚拟机 ip 和 MAC 地址：



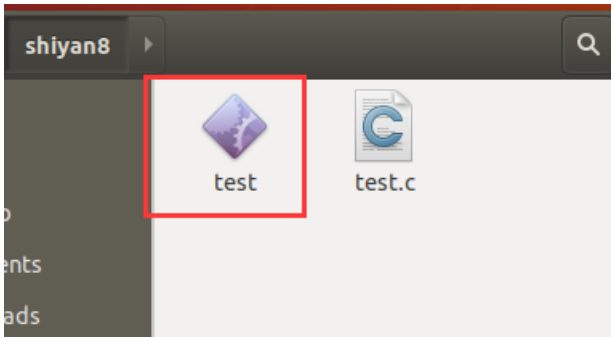
再看一下主机的 ip 和 MAC 地址：



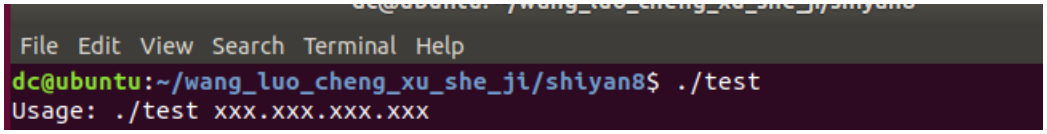
在发包之前主机的 arp 表是没有虚拟机的信息的：



然后在虚拟机测试，先把文件编译为可执行程序：

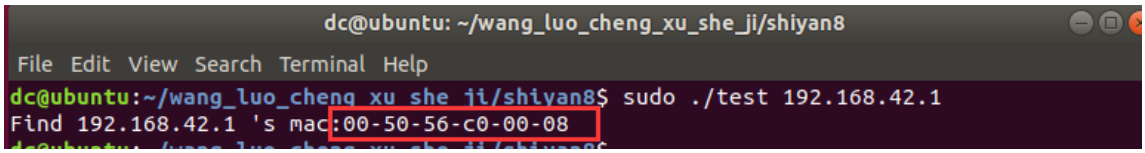


然后执行：



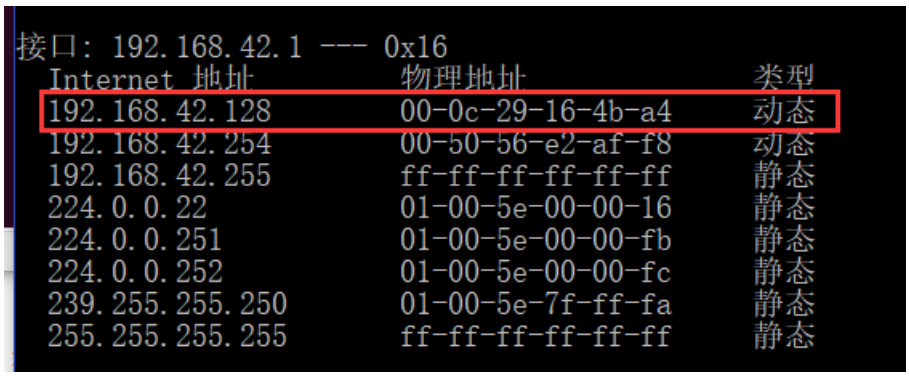
会提示使用方法。

接下来向主机发 arp 包：



可以看到成功找到主机的 MAC 地址，这和刚开始看的是一致的。

查看主机 arp 表：



可以看到虚拟机的信息成功加入到主机的 arp 表中。

实验成功！

实验总结：

通过这次实验学习到了 arp 数据链路层传递信息的方式，同时学习了使用 SOCK_PACKET 这个选项进行编程，书上的版本有些问题，没有执行成功，后来深入理解了之后，在书上的代码进行修改了之后，成功发送 arp 数据包。

另外还有注意的一点是，要注意对齐的方式，因为有些参数是 6 字节对齐的，有些是 4 字节对齐的，需要稍微注意一下。不过总的来说，看到最后结果成功，虚拟机成功向主机发送 arp 请求，并且能在主机的 arp 表中进行更新，也是收获满满！