目录

[5 网络编程 2](#_Toc532490187)

[5.1 网络编程基础知识 2](#_Toc532490188)

[5.1.1 TCP/IP 四层模型 2](#_Toc532490189)

[5.1.2 数据封装和IP地址 2](#_Toc532490190)

[5.1.3 网络端口号及TCP协议 2](#_Toc532490191)

[5.2 TCP编程 3](#_Toc532490192)

[5.2.1创建socket 3](#_Toc532490193)

[5.2.2 字节序 3](#_Toc532490194)

[5.2.3 地址结构 3](#_Toc532490195)

[5.2.4 ipv4 地址和字符地址之间的转换 4](#_Toc532490196)

[5.2.5 TCP客户端服务器编程模型 4](#_Toc532490197)

[5.2.6 基于tcp的服务器网络编程 5](#_Toc532490198)

[5.2.6 基于tcp的客户端网络编程 8](#_Toc532490199)

[5.2.6 tcp的连接和关闭过程 9](#_Toc532490200)

[5.2.7 服务器多并发概念 10](#_Toc532490201)

[5.2.8 多进程模型网络编程 10](#_Toc532490202)

[5.2.9 多线程模型网络编程 17](#_Toc532490203)

[5.3 UDP编程 22](#_Toc532490204)

[5.3.1 UDP 传输流程 22](#_Toc532490205)

[5.3.2 udp 常用系统调用 22](#_Toc532490206)

[5.3.3 udp编程 22](#_Toc532490207)

[5.3.4 域名解析 26](#_Toc532490208)

[5.3.5 通过域名访问网络编程 28](#_Toc532490209)

[多进程和多线程思考 30](#_Toc532490210)

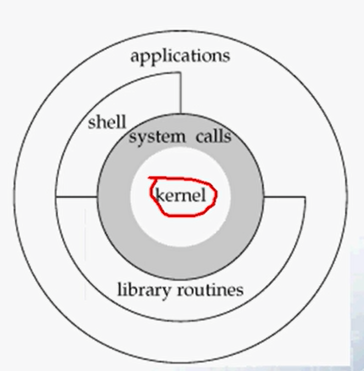
# 1 Linux IO系统

## linux 系统基本介绍

### 1.1.1 Linux 系统基本构成

Linux操作系统的基本构成：内核，系统调用，

c库,shell,文件系统，应用程序。



内核负责管理系统进程、内存、设备驱动程序、文件和网络系统，控制系统和硬件之间的通信。

内核源码：/usr/src目录下面。

shell 提供用户操作系统的接口。

/ 表示根文件系统

进程：操作系统分配资源的最小单位。

线程：一个进程内部由多个线程，线程共享内存。

initrd:最小文件系统。

Linux启动过程：bootloader🡪加载内核🡪加载最小文件系统🡪加载硬盘上的根文件系统🡪启动1号进程/sbin/init

ls -l /boot/ 目录里面 initrd.img即最小文件系统，vmlinuz-4.15.0加载的内核映像。

init进程任务：

执行/etc/init.d/目录下的所有脚本文件，启动某些系统服务。

执行/sbin/getty初始化0、1、2、（标准输入、标准输出、标准错误）

执行/bin/login启动用户登录程序

管理孤儿进程。

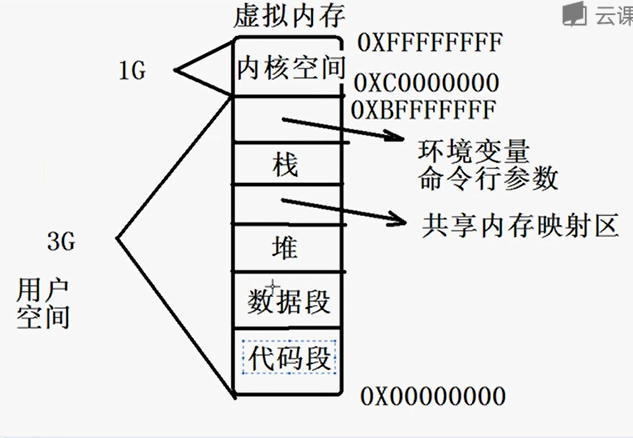
### Linux 系统内存管理

虚拟内存大小：和处理器位数有关。

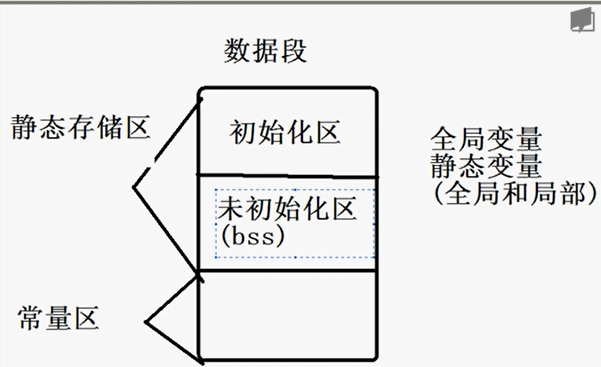
#### 进程内存分配：

一个进程的虚拟地址空间大小为2^n,n为处理器位数。典型的32位处理器共有4G虚拟地址空间。分为最高位1G内核空间，3G用户空间。

对一个用户进程虚拟地址空间分配：



数据段分为三段：



### linux 系统调用介绍

系统调用就是内核提供的一些特殊的函数，用户进程访问内核空间和使用内核服务必须使用系统调用。

进程状态：

内核态：进程运行再内核空间。

用户态：进程运行在用户空间。

## 文件IO

### 1.2.1 标准c的IO

typedef struct iobuf{

int cnt; //剩余的字节数

char \*ptr; //下一个字符位置

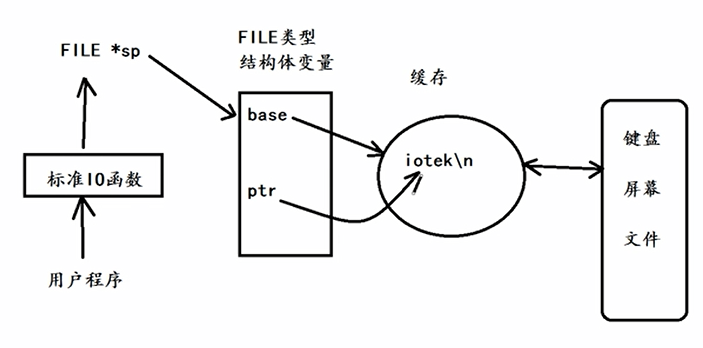
char \*base; //内存缓冲区的位置

int flag; //文件访问模式

int fd; //文件描述符，访问外存

}FILE;

标准c的IO都是带缓存的，所谓缓存意思就是对文件的读写是首先保存到内存缓冲区中的，延时一段时间再保存到外存去。



#### IO 缓存类型

全缓存：要求填满整个缓存区后才进行I/O系统调用操作。对于磁盘文件通常使用全缓存。

行缓存：涉及一个终端时（标准输入和输出），使用行缓存。行缓存满自动输出，碰到换行符自动输出。

无缓存：标准错误stderr不带缓存，使得错误信息能尽快显示出来。

行缓存案例：

#include<stdio.h>

int main(void)

{

printf("hello iotek");

while(1){

sleep(1);

}

return 0;

}

实验：

mkdir src bin obj include

gcc -o bin/line\_buf src/line\_buf.c

bin/line\_buf

发现并没有输出hello iotek。这是因为printf是行缓存的。缓存满，程序结束或者遇到换行符才会输出。

改程序：printf("hello iotek");为printf("hello iotek\n");再次运行程序发现输出了hello iotek。

### 1.2.2 文件I/O系统调用

系统调用都是不带缓存的，一般响应要求高的直接使用系统调用。

文件描述符：非负整数，标识一个文件。STDIN\_FILENO、STDOUT\_FILENO、STDERR\_FILENO对应标准输入0，标准输出1，标准错误2。

在ubuntu 中输入>man 系统调用名字 可以找到系统调用的帮助。

#### open函数

#inlcude<sys/types.h>//所有头文件都在内核源码include 目录下

#include<sys/stat.h>

#include<fcntl.h>

int opent(const char \*pathname,int flags);

int opent(const char \*pathname,int flags,mode\_t mode);

参数：

pathname: 文件路径

flag: 用来说明对文件进行什么操作

O\_RDONLY： 只读

O\_WRONLY： 只写

O\_RDWR ： 读写

O\_NONBLOCK: 非阻塞模式执行文件操作函数

O\_CREAT： 如果文件不存在则创建该文件

O\_EXCL： 如果使用O\_CREAT选项且文件存在，则返回错误消息，可以测试文件是否存在

O\_DIRECTORY:如果pathname不是一个目录，则出错。测试是否一个目录。

O\_NOCTTY： 如果文件为终端，那么终端不可以调用open系统调用的那个进程的控制终端

O\_TRUNC： 只读或者只写打开模式，如果文件已经存在则删除文件中原有数据

O\_APPEND： 以追加的方式打开

mode：如果文件被新建，指定其权限未mode，mode是八进制权限码，0777表示文件所有者，组，其他人都拥有读写执行权限

#### close函数

#include<unistd.h>

int close(int fd);

功能：关闭一个打开的文件

返回：成功为0，出错-1

#### read函数

#include <unistd.h>

ssize\_t read(int fd,void \*buf,size\_t count);

返回：读到的字节数，若已经到文件尾部返回0，出错-1。读操作从文件当前位移处开始，成功返回之前增加实际读得的字节数。

读到的实际数据数少于要求读取的字节数：

1>读到文件尾端

2>读终端设备，一次最多一行

3>进程由于信号中断

4>网络读，网络缓存机构造成小于所要求读的字节数。

#### write 函数

#include <unistd.h>

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

返回：成功返回写入的字节数，出错-1。

如果打开文件时指定了O\_APPEND,则从文件末尾开始写。

#### 复制文件案例：

//头文件io.h

#endif #ifndef \_\_IO\_H\_\_

#define \_\_IO\_H\_\_

**extern** **void** copy(**int** fdin,**int** fdout);

#endif

//c文件copy.c，实现复制函数copy

#include <unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include <string.h>

#include <errno.h>

#include "io.h"

#include <fcntl.h>

#define BUFFER\_LEN 1024

**void** copy(**int** fd1,**int** fd2)

{

**char** buffer[BUFFER\_LEN];

    ssize\_t size;

**while**(size=read(fd1,buffer,BUFFER\_LEN)>0){//读正确

**if**(write(fd2,buffer,size)!=size){

**fprintf**(stderr,"write error:%s\n",**strerror**(**errno**));

**exit**(1);

            }

    }

**if**(size<0){//读错误

        //errno是系统设置的错误编码，是上次系统调用产生错误时设置

**fprintf**(stderr,"read error:%s\n",**strerror**(**errno**));

**exit**(1);

        }

}

//cp.c文件，复制一个文件源代码

#include <unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include <string.h>

#include <errno.h>

#include "io.h"

#include <fcntl.h>

**int** main(**int** argc,**char** \*argv[])

{

**if**(argc!=3){

**fprintf**(stderr,"usage:%s srcfile desfile\n",argv[0]);

**exit**(1);

    }

**int** fdin,fdout;

    //打开一个只读文件

    fdin=open(argv[1],O\_RDONLY);

**if**(fdin<0){

**fprintf**(stderr,"open error:%s\n",**strerror**(**errno**));

**exit**(1);

    }**else** {

**printf**("open file:%d\n",fdin);

    }

    fdout=open(argv[2],O\_WRONLY|O\_CREAT|O\_TRUNC,0777);

**if**(fdout<0){

**fprintf**(stderr,"open error:%s\n",**strerror**(**errno**));

**exit**(1);

    }**else** {

**printf**("open file:%d\n",fdout);

    }

    copy(fdin,fdout);

    close(fdin);

    close(fdout);

**return** 0;

}

//编译为模块

gcc -c src/copy.c -o obj/copy.o -Iinclude

//编译cp 程序

gcc -o bin/cp -Iinclude obj/copy.o src/cp.c

//测试是否成功复制

bin/cp src/copy.c ./copy.c

#### lseek 函数

#include<sys/types.h>

#include<uinistd.h>

off\_t lseek(int fd,off\_t offset,int whence);

返回：若成功则返回新的文件位移量，若出错为-1

功能：定位一个已经打开的文件。

offset :位移量

whence:定位的参考位置。

SEEK\_SET:文件开头

SEEK\_CUR:当前位置，此时offset可正可负

SEEK\_END:文件结尾，如果offset为正，会空出一些字节，这种文件称为空洞文件。

通过lseek获取文件大小，将whence设为SEEK\_END，offset=0,返回值就是文件大小。

lseek 还可以用来确定文件是否可以设置位移量。如果文件描述符引用的是一个管道或者FIFO,则lessk返回-1，并设置errno为EPIPE

printf("file length:%ld\n",lseek(fdin,0L,SEEK\_END));//获取文件大小

printf("file length:%ld\n",lseek(fdin,0L,SEEK\_CUR));//获取当前位置

### 1.2.3 读写文件缓冲区大小选择

df -k查看当前分区看到硬盘是哪个设备：

文件系统 1K-块 已用 可用 已用% 挂载点//根文件挂载点

/dev/sda1 30830500 18763656 10477700 65% /

sudo tune2fs -l /dev/sda1 查看硬盘具体信息发现有一项磁盘块大小

Block size: 4096

系统操作都是以Block size为单位的，所以在读写文件时，将缓冲区设为Block size大小，效率最高。

### 1.2.4 内核数据结构

一个打开的文件在内核中使用三种数据结构表示：

1. 文件描述符表

文件描述符标志fd

文件表项指针

1. 文件表项 : 操作文件的动态信息

文件状态标志：读、写、追加、同步、和非阻塞状态标志

当前文件偏移量

i节点表项指针

引用计数器

1. i节点 ：文件的静态信息

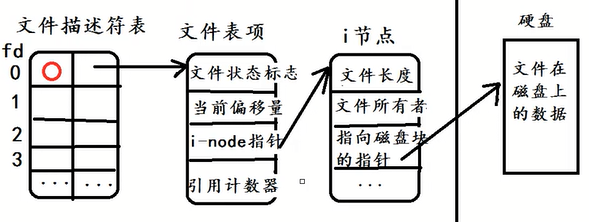
文件类型和对该文件的操作函数指针

当前文件长度

文件所有者

文件所在的设备、文件访问权限

指向文件数据在磁盘块上所在位置的指针等



### 1.2.5 原子操作

//文件读写并发出现竞争问题

#include <unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include <string.h>

#include <errno.h>

#include <fcntl.h>

**int** main(**int** argc,**char** \*argv[])

{

**if**(argc<3){

**fprintf**(stderr,"usage:%s content destfile\n",argv[0]);

**exit**(1);

    }

**int** fd;

    fd=open(argv[2],O\_WRONLY);

**if**(fd<0){

**perror**("open error\n");

**exit**(1);

    }

    lseek(fd,0L,SEEK\_END);

    sleep(10);

**size\_t** size=**strlen**(argv[1])\***sizeof**(**char**);

**if**(write(fd,argv[1],size)!=size){

**perror**("write error\n");

**exit**(1);

    }

    close(fd);

**return** 0;}

#### 实验

touch append.txt //创建文件

//打开两个终端，同时执行下面两句命令,向append.txt写入数据

bin/file\_append aaaaaa append.txt

bin/file\_append AAAAAA append.txt

结果发现append.txt里面只有AAAAAA,原因是两次程序执行都是在先获得文件的尾部后休眠，休眠10s后再写入。那么他们获得的尾部偏移是一样的，后写入的覆盖了前面写入的字符。

#### 如何避免出现这种情况

采用append方式打开文件，而不是使用lseek。

注释掉  lseek(fd,0L,SEEK\_END);

更改open函数如下：

fd=open(argv[2],O\_WRONLY|O\_APPEND);//以追加方式写

再次实验发现append.txt 文件内容aaaaaaAAAAAAaaaaaaAAAAAA。写入正常

O\_APPEND模式写过程：

write(){

1. 从i节点中读取文件长度作为当前偏移量
2. 往文件中写入数据
3. 修改i节点中文件长度

}

文件追加：

对open使用O\_APPEND使得对文件偏移调整和追加称为原子操作

对open使用O\_CREAT和O\_EXEL使得文件是否存在和创建过程称为原子操作

### 1.2.6 dup和dup2函数

#include<unistd.h>

int dup(int oldfd);

int dup2(int oldfd,int newfd);

返回：成功返回新文件描述符，出错返回-1

功能：文件描述符的复制。实际意思是文件描述符表指针的复制。将oldfd的文件描述符表指针复制给newfd,以后对newfd的操作实际就是对原来oldfd关联的文件的操作。

作用：在进程间通信时可以用来改变进程的标准输入和标准输出设备。

oldfd:原先的文件描述符。newfd 新的描述符

由dup返回的新文件描述符一定是当前可用文件描述符中的最小数值。

dup2则可以用newfd参数指定新描述符的数值，如果newfd已经打开，则先将其关闭。如果oldfd等于newfd,则dup2返回newfd,而不关闭它。

#### 重定向：

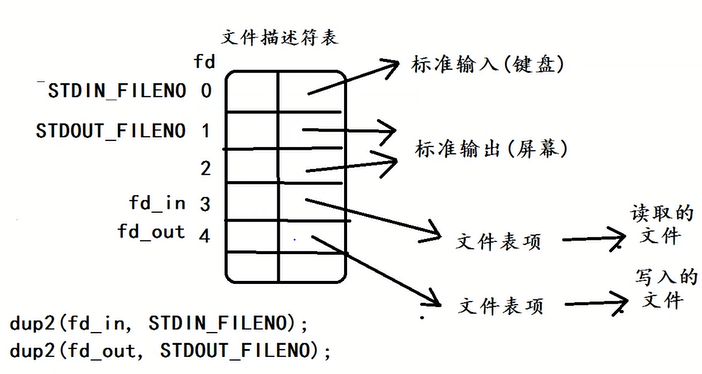
输入重定向 <

输出重定向 >

cat > cat.txt这样就把标准输出定向到cat.txt中。

如果文件不存在，重定向会新建文件；如果文件存在会覆盖原来文件内容。

追加重定向符号：>> << ；追加重定向不会覆盖原有内容。



STDIN\_FILENO的文件描述符表指针指向fd\_in关联的文件

STDOUT\_FILENO的文件描述符表指针指向fd\_out关联的文件

以后对STDIN\_FILENO的读取就是操fd\_in关联文件

STDOUT\_FILENO的写就是写入fd\_out关联的文件

# 5 网络编程

## 5.1 网络编程基础知识

### 5.1.1 TCP/IP 四层模型

**应用层：**Telnet(远程登陆);FTP(文件传输);SMTP(简单邮件传输);DNS(域名);

SNMP(简单网络管理)http(超文本传输)

**传输层：TCP,UDP**

**网络层：IP，ARP(**地址解析协议)

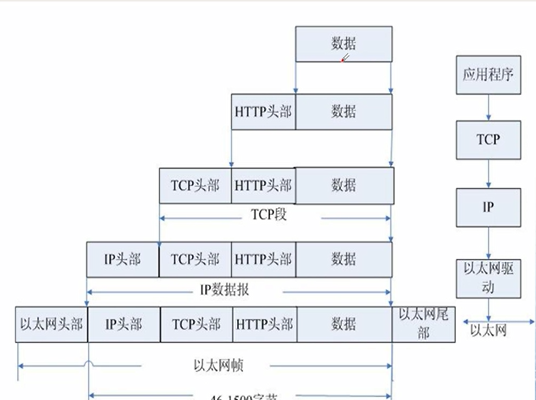
**物理层：**

应用层指定数据格式，传输层指定如何传输数据，网络层负责寻找路径，

物理层负责收发数据。

### 5.1.2 数据封装和IP地址

数据封装：

****

IP 地址：

长度32位，四个字节。使用点分十进制记录。

IP地址组成：网络类型 +网络ID+ 主机ID

192.168.0.112

C类网络类型：前3位110标识是网络类型，最后8位主机地址ID

中间为网络地址ID。前面24组成网段。

### 5.1.3 网络端口号及TCP协议

ip地址对应于一个网卡，那么网络端口号标识与哪个进程通信。

more /etc/services 用于查看各种服务的端口号。2000以下端口号用于特定服务，自己应用的特殊服务应该使用2000以上的端口。

80作为网页服务器的访问端口；21Ftp服务。

## 5.2 TCP编程

### 5.2.1创建socket

#include<sys/socket.h>

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

返回值：成功返回描述符，出错返回-1

参数：

domain： 网络层协议

AF\_INET ipv4

AF\_INET6 ipv6

protocol:

通常为0，表示按给定的域和套接字类型选择默认协议。

type: 传输层协议

SOCK\_STREAM tcp协议

SOCK\_DGRAM udp协议

### 5.2.2 字节序

大端字节序和小端字节序，网络传输采用大端字节序。

uint32\_t htonl(uint32\_t hostlong);//主机字节序转换为网络字节序host to net long

uint16\_t htons(uint16\_t hostshort)// 主机字节序转换为网络字节序16位数据

uint32\_t ntohl(uint32\_t netlong);//网络字节序转换为主机字节序

uint16\_t ntohs(uint16\_t netshort)//

### 5.2.3 地址结构

/\*linux地址格式，不常用\*/

#include <sys/socket.h>

struct sockaddr{

unsigned short sa\_family; // internet 地址簇 ，AF\_XXX

char sa\_data[14]; // 14bbytes 的协议地址

}

sa\_data 包含了一些远程电脑的地址、端口和套接字的数目

sa\_family 一般来说，IPV4使用AF\_INET.

在传递给需要地址结构的函数时，把指向该结构的指针转换成（struct sockaddr \*）传递进去。

//Ipv4地址格式，常用

struct in\_addr{

in\_addr\_t s\_addr; //in\_addr\_t 定义成uint32\_t

}

struct sockaddr\_in{

short int sin\_family; //地址类别AF\_INET.，主机字节序

unsigned short int sin\_port;//端口号 in\_port\_t 定义成uint16\_t，网络字节序

struct in\_addr sin\_addr; //远程ipv4地址

unsigned char sin\_zero[8] //0填充位

}

函数默认是Linux地址格式，需要将网络格式强制转换为linux地址格式。 因为网络格式方便，所以一般使用网络格式进行数据设置，设置完再转换为Linux地址格式。

### 5.2.4 ipv4 地址和字符地址之间的转换

#include <arpa/inet.h>

const char \*inet\_ntop(int domain,const void \*restrict addr,char \*restrict str,socklen\_t size)

返回：成功返回地址字符串指针，出错返回NULL

功能：网络字节序转换成点分十进制。ntop net to point

int inet\_pton(int domain,const char \*restrict str,void \*restrict addr);

返回：成功返回1，无效格式返回0，出错返回-1

功能:点分十进制转换为网络字节序

domain: internet 地址簇，如AF\_INET

addr: internet 地址，32位IPV4地址（网络字节序）

str: 地址字符串（点分十进制）

size: 地址字符串大小

例子：

struct sockaddr\_in sin; //定义一个sockaddr\_in 结构体

char buf[16]; //存放转换后的点分十进制格式ip地址，字符串格式。12个数字，三个点，//1个字符串结束符

memset(&sin,0,sizeof(sin));

sin.sin\_family=AF\_INET; //地址协议类型ipv4

sin.sin\_port=htons((short)3001); //填写端口号(网络字节序)

if(inet\_pton(AF\_INET,”192.168.2.1”,&sin.sin\_addr.s\_addr)<=0)

{//错误处理

}

printf(“%s\n”,inet\_ntop(AF\_INET,&sin.sin\_addr.s\_addr,buf,sizeof(buf)));

### 5.2.5 TCP客户端服务器编程模型

客户端调用序列：

调用socket 函数创建套接字

调用connect 连接服务器端

调用I/O函数(read/write)与服务器通讯

服务器端调用序列

调用socket 函数创建套接字

调用bind绑定本地地址和端口

调用Listen启动监听

调用accept从已连接队列中提取客户连接

调用I/O函数(read/write)与客户端通讯

调用close关闭套接字

绑定地址函数：

#include<sys/socket.h>

int bind(sockfd,const struct sockaddr \*addr,socklen\_t len);

返回：成功返回0，出错返回-1

查找绑定到的套接字的地址：

#include<sys/socket.h>

int getsockname(int sockfd,struct sockaddr \*restrict addr,socklen\_t \*restrict alenp);

返回：成功返回0，出错返回-1

获取对方地址：

#include<sys/socket.h>

int getpeername(int sockfd,struct sockaddr \*restrict addr,socklen\_t \*restrict alenp);

返回：成功返回0，出错返回-1

监听连接：

#inlcude<sys/socket.h>

int listen(int sockfd,int backlog);

返回：成功返回0，出错返回-1。backlog 指定进行客户端连接排队的队列长度。

接收连接：

int accept(int sockfd,struct sockaddr\*restrict addr,socklen\_t \*restrict len);

地址参数用来存储来自客户端的地址。

返回值：fd不是服务器之前创建的socketfd。这是和客户端通信的socketfd。

请求连接：

int commect(int sockfd,const struct sockaddr \*addr,socklen\_t len);

返回：成功返回0，出错返回-1

close((int sockfd) //关闭网络连接

### 5.2.6 基于tcp的服务器网络编程

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<unistd.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<signal.h>

#include<time.h>

int sockfd;

//

void sig\_handler(int signo)

{

**if**(signo==SIGINT){

    printf("server close\n");

//步骤6

    close(sockfd);

    exit(1);

}

}

void out\_addr(struct sockaddr\_in \*clientaddr)

{

    int port=ntohs(clientaddr->sin\_port);

    char ip[16];

    memset(ip,0,sizeof(ip));

//将Ip地址从网络字节序转换成点分十进制

    inet\_ntop(AF\_INET,&clientaddr->sin\_addr.s\_addr,ip,sizeof(ip));

    printf("client:%s(%d) connected\n",ip,port);

}

//连接成功后，向客户端发送系统时间

void do\_serverice(int fd)

{

    long t =time(0);

     char \*s =ctime(&t);

    size\_t size=strlen(s)\*sizeof(char);

**if**(write(fd,s,size)!=size){

        perror("write error");

    }

}

int main(int argc, char \*argv[])

{

**if**(argc<2){

        printf("usage:%s #port\n",argv[0]);

        exit(1);

    }

**if**(signal(SIGINT,sig\_handler)==SIG\_ERR){//注册信号处理函数

    perror("signal sigint error");

    exit(1);

}

/\* 步骤1：创建socket套接字

 socket 创建在内核中\*/

sockfd=socket(AF\_INET,SOCK\_STREAM,0);

struct sockaddr\_in serveraddr;

memset(&serveraddr,0,sizeof(serveraddr));

serveraddr.sin\_family=AF\_INET;

//端口号要求网络字节序，命令行传递的是字符串，需要先转换为int.

serveraddr.sin\_port=htons(atoi(argv[1]));

/\*步骤2

一台主机有多个网络接口和ip地址，如果只关心某个地址的请求连接，可以指定一个具体的ip.要响应所有接口ip就需要一个特殊的地址INADDR\_ANY.

#define INADDR\_ANY (uint32\_t )0x00000000

\*/

serveraddr.sin\_addr.s\_addr=INADDR\_ANY;

**if**(bind(sockfd,(struct sockaddr\*)&serveraddr,sizeof(serveraddr))<0){

    perror("bind error");

    exit(1);

}

/\*步骤3

最多同时10个连接。相当listen于创建了一个单独的线程用来处理连接请求

\*/

**if**(listen(sockfd,10)<0){

    perror("listen error");

    exit(1);

}

/\*步骤4：调用accept函数从队列中获取一个客户端的请求连接，并返回新的socket描述符，如果没有客户端连接，调用此函数后会阻塞\*/

struct sockaddr\_in clientaddr;

socklen\_t clientaddr\_len=sizeof(clientaddr);

**while**(1){

    int fd=accept(sockfd,(struct sockaddr\*)&clientaddr,& clientaddr\_len);

**if**(fd<0){

        perror("accept error");

**continue**;

}

/\*步骤5：调用IO函数(read/write)和连接的客户端进行双向通信\*/

out\_addr(&clientaddr);

do\_serverice(fd);

/\*步骤6：通信完成，关闭和客户端连接的socket\*/

close(fd);

}

**return** 0;

}

实验测试：

./time\_tcp\_service 8888 //服务器启动

client:127.0.0.1(34494) connected //测试结果

//开启新终端，使用telnet测试

chao@chao:~$ telnet 127.0.0.1 8888

Trying 127.0.0.1...

Connected to 127.0.0.1.

Escape character is '^]'.

Mon Dec 10 14:33:45 2018 //测试结果得到服务器发送时间

Connection closed by foreign host.

//使用http测试，在浏览器输入

<http://127.0.0.1:8888/>

Mon Dec 10 15:13:59 2018 //测试结果得到服务器发送时间

注意：服务器需要指定端口号，而客户端不需要指定，由系统分配。

### 5.2.6 基于tcp的客户端网络编程

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<unistd.h>

int main(int argc, char \*argv[])

{

**if**(argc<3){

        printf("usage:%s ip port\n",argv[0]);

        exit(1);

    }

    /\*步骤1\*/

     int sockfd=socket(AF\_INET,SOCK\_STREAM,0);

**if**(sockfd<0){

        perror("socket error");

        exit(1);

    }

    struct sockaddr\_in serveraddr;

    memset(&serveraddr,0,sizeof(serveraddr));

    serveraddr.sin\_family=AF\_INET;

    //端口号要求网络字节序，命令行传递的是字符串，需要先转换为int.

    serveraddr.sin\_port=htons(atoi(argv[2]));

    //将ip地址转化为网络字节序

    inet\_pton(AF\_INET,argv[1],&serveraddr.sin\_addr.s\_addr);

    /\*步骤2：客户端调用connect函数连接到服务器端\*/

**if**(connect(sockfd,(struct sockaddr\*)&serveraddr,sizeof(serveraddr))<0)

    {

        perror("connect error");

        exit(1);

    }

    /\*步骤3：读写\*/

    char buffer[1024];

    memset(buffer,0,sizeof(buffer));

    size\_t size;

**if**((size=read(sockfd,buffer,sizeof(buffer)))<0){

        perror("read error");

        }

**if**(write(STDOUT\_FILENO,buffer,size)!=size){

        perror("write error");

    }

    /\*步骤4：关闭socket\*/

    close(sockfd);

**return** 0;

}

实验：

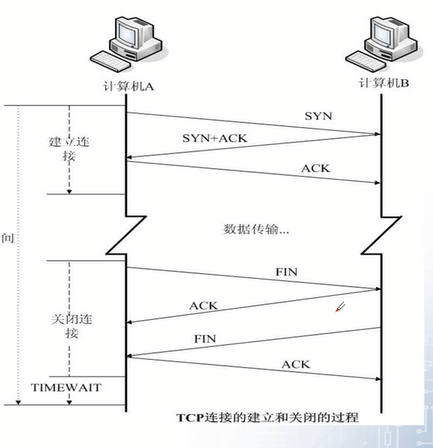
chao@chao:~/linux/network$ ./time\_tcp\_service 8888

client:127.0.0.1(34690) connected

chao@chao:~/linux/network$ ./time\_tcp\_client 127.0.0.1 8888

Mon Dec 10 16:21:34 2018

### 5.2.6 tcp的连接和关闭过程



计算机A是客户端，计算机B是服务器

建立连接需要三次握手：

1. 由客户端发送SYN报文
2. 由服务器返回SYN+ACK报文
3. 最后由客户端发送ACK建立最终连接。

关闭需要两次握手

由提出关闭方发送FIN报文，当对方返回ACK时连接关闭。

gcc -o msg.o -c msg.c //将msg.c编译成模块

### 5.2.7 服务器多并发概念

//连接成功后，向客户端发送系统时间

void do\_serverice(int fd)

{

size\_t len;

char buffer[20];

if((len=read(fd,buff,20))<0){

perror(“read error”);

}

    long t =time(0);

     char \*s =ctime(&t);

    size\_t size=strlen(s)\*sizeof(char);

**if**(write(fd,s,size)!=size){

        perror("write error");

    }

}

实验：

重新编译服务器程序，运行：

./time\_tcp\_service 8888

连接客户端1：

连接客户端2：

./time\_tcp\_client 127.0.0.1 8888

发现只有客户端1连接成功，客户端2没有连接成功，而且服务器和客户端都阻塞。

服务器和客户端都阻塞原因：

服务器等待读客户端发送的数据，而客户端等待服务器发送的数据。因此形成了死锁。

客户端1连接成功，客户端2没有连接成功的原因：

服务器只有一个进程，服务器阻塞因而无法继续执行，也就没有办法接受第二个客户端的连接请求。

解决办法：

服务器针对每一个客户端的请求创建一个子进程 ，该子进程单独服务于该客户端。

还可以使用多线程及IO多路转换

/

### 5.2.8 多进程模型网络编程

#### 自定义应用协议网络通信读写函数：

//头文件msg.h

#ifndef \_\_MSG\_H\_\_

#define \_\_MSG\_H\_\_

#include <sys/types.h>

typedef struct{

    char head[10];//协议头部

    char checknum;//校验码

    char buff[512];

}Msg;

//发送自定义协议的message,发送的数据存放在buff中。

extern int write\_msg(int sockfd,char \*buff,size\_t len);

extern int read\_msg(int sockfd,char \*buff,size\_t len);

#endif

//读写函数msg.c

#include "msg.h"

#include <unistd.h>

#include <string.h>

#include <memory.h>

//计算校验码

static unsigned char msg\_check(Msg \*message)

{

    unsigned char s=0;

    int i;

    for(i=0;i<sizeof(message->head);i++){

        s+=message->head[i];

    }

    for(i=0;i<sizeof(message->buff);i++){

        s+=message->buff[i];

    }

    return s;

}

//发送自定义协议的message,发送的数据存放在buff中。

 int write\_msg(int sockfd,char \*buff,size\_t len)

{

    Msg message;

    memset(&message,0,sizeof(message));

    strcpy(message.head,"iotek2012");

    memcpy(message.buff,buff,len);

    message.checknum=msg\_check(&message);

    if(write(sockfd,&message,sizeof(message))!=sizeof(message)){

        return -1;

    }

}

 int read\_msg(int sockfd,char \*buff,size\_t len)

{

    Msg message;

    memset(&message,0,sizeof(message));

    size\_t size;

    if((size=read(sockfd,&message,sizeof(message)))<0){

        return -1;

    }else if(size==0){

    return 0;

    }

    unsigned char s=msg\_check(&message);

    //验证数据

    if((s==(unsigned char)message.checknum)&&(!strcmp("iotek2012",message.head)))

    {

        memcpy(buff,message.buff,len);

        return sizeof(message);

    }

    return -1;

}

#### 自定义应用协议网络通信服务器程序：

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<unistd.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<signal.h>

#include<time.h>

#include "msg.h"

#include<errno.h>

#include <sys/wait.h>

**int** sockfd;

//

**void** sig\_handler(**int** signo)

{

**if**(signo==SIGINT){

**printf**("server close\n");

        //步骤6

        close(sockfd);

**exit**(1);

    }

**if**(signo==SIGCHLD){

**printf**("child process deaded...\n");

        wait(0);

    }

}

**void** out\_addr(**struct** sockaddr\_in \*clientaddr)

{

**int** port=ntohs(clientaddr->sin\_port);

**char** ip[16];

**memset**(ip,0,**sizeof**(ip));

//将Ip地址从网络字节序转换成点分十进制

    inet\_ntop(AF\_INET,&clientaddr->sin\_addr.s\_addr,ip,**sizeof**(ip));

**printf**("client:%s(%d) connected\n",ip,port);

}

//连接成功后，向客户端发送系统时间

void do\_serverice(int fd)

{

    /\*和客户端双向通信

\*/

    char buff[512];

    while(1){//循环读直到数据读完

        memset(buff,0,sizeof(buff));

        printf("start read and write...\n");

        size\_t size;

        if((size=read\_msg(fd,buff,sizeof(buff)))<0){

            perror("protocal error");

            break;

        }else if(size==0){//注意：如果读过程客户端断开，那么读完数据，读取长度0

            break;

        }else{

            printf("%s\n",buff);

            if(write\_msg(fd,buff,sizeof(buff))<0){

//注意：如果读过程客户端断开，那么写错误EPIPE，类似管道

                if(errno==EPIPE){

                    break;

                }

                perror("protocal error");

                }

        }

    }

}

int main(int argc, char \*argv[])

{

    if(argc<2){

        printf("usage:%s #port\n",argv[0]);

        exit(1);

    }

if(signal(SIGINT,sig\_handler)==SIG\_ERR){//注册信号处理函数

    perror("signal sigint error");

    exit(1);

}

if(signal(SIGCHLD,sig\_handler)==SIG\_ERR){//注册信号处理函数

    perror("signal sigchld error");

    exit(1);

}

/\* 步骤1：创建socket套接字

 socket 创建在内核中\*/

sockfd=socket(AF\_INET,SOCK\_STREAM,0);

struct sockaddr\_in serveraddr;

memset(&serveraddr,0,sizeof(serveraddr));

serveraddr.sin\_family=AF\_INET;

//端口号要求网络字节序，命令行传递的是字符串，需要先转换为int.

serveraddr.sin\_port=htons(atoi(argv[1]));

/\*步骤2

一台主机有多个网络接口和ip地址，如果只关心某个地址的请求连接，可以指定一个具体的ip.要响应所有接口ip就需要一个特殊的地址INADDR\_ANY.

#define INADDR\_ANY (uint32\_t )0x00000000

\*/

serveraddr.sin\_addr.s\_addr=INADDR\_ANY;

if(bind(sockfd,(struct sockaddr\*)&serveraddr,sizeof(serveraddr))<0){

    perror("bind error");

    exit(1);

}

/\*步骤3

最多同时10个连接。相当listen于创建了一个单独的线程用来处理连接请求

\*/

if(listen(sockfd,10)<0){

    perror("listen error");

    exit(1);

}

/\*步骤4：调用accept函数从队列中获取一个客户端的请求连接，并返回新的socket描述符，如果没有客户端连接，调用此函数后会阻塞\*/

struct sockaddr\_in clientaddr;

socklen\_t clientaddr\_len=sizeof(clientaddr);

while(1){

    int fd=accept(sockfd,(struct sockaddr\*)&clientaddr,& clientaddr\_len);

if(fd<0){

        perror("accept error");

        continue;

}

/\*步骤5：调用IO函数(read/write)和连接的客户端进行双向通信

    启动新进程

\*/

    pid\_t pid=fork();

    if(pid<0){

        continue;

    }else if(pid==0){ //子进程继承父进程fd

        out\_addr(&clientaddr);

        do\_serverice(fd);

    /\*步骤6：通信完成，关闭和客户端连接的socket\*/

        close(fd);

        break;

    }else {          //父进程

        close(fd); //不再需要使用fd

    }

}

return 0;

}

#### 自定义应用协议网络通信客户端程序：

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<unistd.h>

#include"msg.h"

**int** main(**int** argc, **char** \*argv[])

{

**if**(argc<3){

**printf**("usage:%s ip port\n",argv[0]);

**exit**(1);

    }

    /\*步骤1\*/

**int** sockfd=socket(AF\_INET,SOCK\_STREAM,0);

**if**(sockfd<0){

**perror**("socket error");

**exit**(1);

    }

**struct** sockaddr\_in serveraddr;

**memset**(&serveraddr,0,**sizeof**(serveraddr));

    serveraddr.sin\_family=AF\_INET;

    //端口号要求网络字节序，命令行传递的是字符串，需要先转换为int.

    serveraddr.sin\_port=htons(**atoi**(argv[2]));

    //将ip地址转化为网络字节序

    inet\_pton(AF\_INET,argv[1],&serveraddr.sin\_addr.s\_addr);

    /\*步骤2：客户端调用connect函数连接到服务器端\*/

**if**(connect(sockfd,(**struct** sockaddr\*)&serveraddr,**sizeof**(serveraddr))<0)

    {

**perror**("connect error");

**exit**(1);

    }

    /\*步骤3：读写\*/

**char** buff[512];

**size\_t** size;

**char** \*prompt=">";

**while**(1){

**memset**(buff,0,**sizeof**(buff));

        write(STDOUT\_FILENO,prompt,1);

        size=read(STDIN\_FILENO,buff,**sizeof**(buff));

**if**(size<0) **continue**;

        buff[size-0] ='\0';

**if**(write\_msg(sockfd,buff,**sizeof**(buff))<0){

**perror**("write msg error");

**continue**;

        }**else**{ //写成功

**if**(read\_msg(sockfd,buff,**sizeof**(buff))<0){

**perror**("read msg error");

**continue**;

            }**else**{//读成功

**printf**("%s\n",buff);

            }

        }

   }

    /\*步骤4：关闭socket\*/

    close(sockfd);

**return** 0;

}

#### 多客户端连接实验：

服务器启动：

root@chao:/home/chao/linux/network/msg# ./echo\_tcp\_service 8888

client:127.0.0.1(39572) connected

start read and write...

hello world //接收

start read and write...

client:127.0.0.1(39580) connected

start read and write...

hello china //接收

start read and write...

hello china1 //接收

start read and write...

hello world1 //接收

start read and write...

客户端1连接：

chao@chao:~$ cd linux/network/msg/

chao@chao:~/linux/network/msg$ ./echo\_tcp\_client 127.0.0.1 8888

>hello world //发送

hello world

>hello world1

hello world1

客户端2连接

chao@chao:~$ cd linux/network/msg/

chao@chao:~/linux/network/msg$ ./echo\_tcp\_client 127.0.0.1 8888

>hello china

hello china

>hello china1

hello china1

### 5.2.9 多线程模型网络编程

以分离线程模式启动线程，当子线程结束会自释放占用的资源。不能用pthread\_join回收子线程。因为pthread\_join会导致主控线程阻塞，而无法接受新的连接。

#### 基于线程的服务器程序

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<unistd.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<signal.h>

#include<time.h>

#include "msg.h"

#include<errno.h>

#include <sys/wait.h>

#include <pthread.h>

**int** sockfd;

//

**void** sig\_handler(**int** signo)

{

**if**(signo==SIGINT){

**printf**("server close\n");

        //步骤6

        close(sockfd);

**exit**(1);

    }

}

**void** out\_addr(**struct** sockaddr\_in \*clientaddr)

{

**int** port=ntohs(clientaddr->sin\_port);

**char** ip[16];

**memset**(ip,0,**sizeof**(ip));

//将Ip地址从网络字节序转换成点分十进制

    inet\_ntop(AF\_INET,&clientaddr->sin\_addr.s\_addr,ip,**sizeof**(ip));

**printf**("client:%s(%d) connected\n",ip,port);

}

//连接成功后，向客户端发送系统时间

**void** do\_serverice(**int** fd)

{

    /\*和客户端双向通信\*/

**char** buff[512];

**while**(1){//循环读直到数据读完

**memset**(buff,0,**sizeof**(buff));

**size\_t** size;

**if**((size=read\_msg(fd,buff,**sizeof**(buff)))<0){

**perror**("protocal error");

**break**;

        }**else** **if**(size==0){//注意：如果读过程客户端断开，那么读完数据，读取长度0

**break**;

        }**else**{

**printf**("%s\n",buff);

**if**(write\_msg(fd,buff,**sizeof**(buff))<0){

    //注意：如果读过程客户端断开，那么写错误EPIPE，类似管道

**if**(**errno**==EPIPE){

**break**;

                }

**perror**("protocal error");

                }

        }

    }

}

**void** out\_fd(**int** fd)

{

**struct** sockaddr\_in addr;

    socklen\_t len =**sizeof**(addr);

//从fd中获得连接客户端的相关信息

**if**(getpeername(fd,(**struct** sockaddr\*)&addr,&len)<0){

**perror**("getpeername error");

**return** ;

    }

**char** ip[16];

**memset**(ip,0,**sizeof**(ip));

**int** port=ntohs(addr.sin\_port);//转为主机字节序

    inet\_ntop(AF\_INET,&addr.sin\_addr.s\_addr,ip,**sizeof**(ip));

**printf**("%16s(%5d)closed!\n",ip,port);

}

**void** \*th\_fn(**void**\*arg)

{

**int** fd=(**int**)arg;

    do\_serverice(fd);

    out\_fd(fd);

    close(fd);

**return** (**void**\*)0;

}

**int** main(**int** argc, **char** \*argv[])

{

**if**(argc<2){

**printf**("usage:%s #port\n",argv[0]);

**exit**(1);

    }

**if**(**signal**(SIGINT,sig\_handler)==SIG\_ERR){//注册信号处理函数

**perror**("signal sigint error");

**exit**(1);

    }

/\* 步骤1：创建socket套接字

 socket 创建在内核中\*/

sockfd=socket(AF\_INET,SOCK\_STREAM,0);

**struct** sockaddr\_in serveraddr;

**memset**(&serveraddr,0,**sizeof**(serveraddr));

serveraddr.sin\_family=AF\_INET;

//端口号要求网络字节序，命令行传递的是字符串，需要先转换为int.

serveraddr.sin\_port=htons(**atoi**(argv[1]));

/\*步骤2

一台主机有多个网络接口和ip地址，如果只关心某个地址的请求连接，可以指定一个具体的ip.要响应所有接口ip就需要一个特殊的地址INADDR\_ANY.

#define INADDR\_ANY (uint32\_t )0x00000000

\*/

serveraddr.sin\_addr.s\_addr=INADDR\_ANY;

**if**(bind(sockfd,(**struct** sockaddr\*)&serveraddr,**sizeof**(serveraddr))<0){

**perror**("bind error");

**exit**(1);

}

/\*步骤3

最多同时10个连接。相当listen于创建了一个单独的线程用来处理连接请求

\*/

**if**(listen(sockfd,10)<0){

**perror**("listen error");

**exit**(1);

}

/\*步骤4：调用accept函数从队列中获取一个客户端的请求连接，并返回新的socket描述符，如果没有客户端连接，调用此函数后会阻塞\*/

pthread\_attr\_t attr;

    pthread\_attr\_init(&attr);

    pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_DETACHED);

**while**(1){

**int** fd=accept(sockfd,NULL,NULL);

**if**(fd<0){

**perror**("accept error");

**continue**;

        }

/\*步骤5：调用IO函数(read/write)和连接的客户端进行双向通信

    启动线程

\*/

        pthread\_t th;

**int** err;

**if**((err=pthread\_create(&th,&attr,th\_fn,(**void**\*)fd))!=0){

**perror**("pthread\_creat error");

        }

       pthread\_attr\_destroy(&attr);

    }

**return** 0;

}

#### 基于线程的客户程序

因为客户端不需要多线程，因此程序不用改，可以使用多进程的客户端程序。

实验：

先启动服务器，然后由客户端发送数据给服务器

chao@chao:~/linux/network/echo\_tcp\_thread$ ./echo\_tcp\_service 8888

hello china 0

hello world 0

hello world 1

hello china 1

chao@chao:~/linux/network/echo\_tcp\_thread$ ./echo\_tcp\_client 127.0.0.1 8888

>hello world 0

hello world 0

>hello world 1

hello world 1

chao@chao:~/linux/network/echo\_tcp\_thread$ ./echo\_tcp\_client 127.0.0.1 8888

>hello china 0

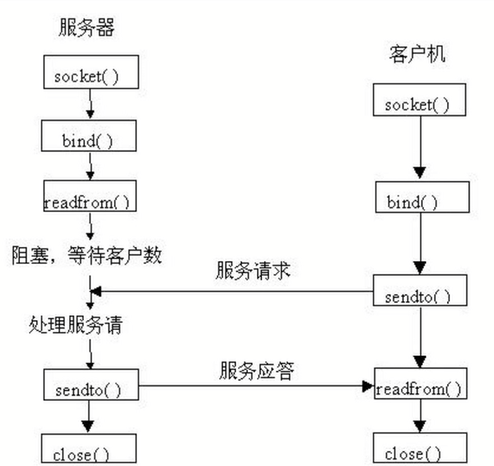
hello china 0

>hello china 1

hello china 1

# 5.3 UDP编程

## 5.3.1 UDP 传输流程



注意：客户端bind()并不是必要的。

## 5.3.2 udp 常用系统调用

#include<sys/socket.h>

ssize\_t send(int sockfd,const void \*buf,size\_t nbytes,int flag);//sockfd本机创建的sockfd

返回：成功返回发送字节数，出错返回-1

ssize\_t sento(int sockfd,const void \*buf,size\_t nbytes,int flag,const struct sockaddr \*destaddr,

socklen\_t destlen);

返回：成功返回发送的字节数，出错返回-1

#inlcude<sys/socket.h>

ssize\_t recv(int sockfd,void \*buf,size\_t nbytes,int flag);

ssize\_t recvfrom(int sockfd,void \*restrict buf,size\_t len,int flag,struct sockaddr \*restrict addr,

socklen\_t \*restrict addrlen);

返回：成功返回接收的字节数，出错返回-1

### 5.3.3 udp编程

#### 服务器程序：

#include<sys/socket.h>

#include<netdb.h>

#include <arpa/inet.h>

#include<unistd.h>

 #include<string.h>

 #include<stdlib.h>

#include<stdio.h>

#include<signal.h>

#include<time.h>

**int** sockfd;

**void** sig\_handler(**int** signo)

{

**if**(signo==SIGINT){

**printf**("server close\n");

        close(sockfd);

**exit**(1);

    }

}

**void** out\_addr(**struct** sockaddr\_in \*clientaddr)

{

**int** port=ntohs(clientaddr->sin\_port);

**char** ip[16];

**memset**(ip,0,**sizeof**(ip));

//将Ip地址从网络字节序转换成点分十进制

    inet\_ntop(AF\_INET,&clientaddr->sin\_addr.s\_addr,ip,**sizeof**(ip));

**printf**("client:%s(%d) connected\n",ip,port);

}

//连接成功后，向客户端发送系统时间

**void** do\_service(**int** fd)

{

**struct** sockaddr\_in clientaddr;

    socklen\_t len=**sizeof**(clientaddr);

**char** buffer[1024];

**memset**(buffer,0,**sizeof**(buffer));

**if**(recvfrom(sockfd,buffer,**sizeof**(buffer),0,(**struct** sockaddr\*)&clientaddr,&len)<0)

        {

**perror**("recvfrom error\n");

        }**else**{

            out\_addr(&clientaddr);

**printf**("client send info: %s\n",buffer);

**long** **int** t =**time**(0);

**char** \*ptr=**ctime**(&t);

**size\_t** size=**strlen**(ptr)\***sizeof**(**char**);

**if**(sendto(sockfd,ptr,size,0,(**struct** sockaddr\*)&clientaddr,len)<0){

**perror**("send error\n");

            }

        }

}

**int** main(**int** argc, **char** \*argv[])

{

**if**(argc<2){

**printf**("usage:%s #port\n",argv[0]);

**exit**(1);

    }

**if**(**signal**(SIGINT,sig\_handler)==SIG\_ERR){//注册信号处理函数

**perror**("signal sigint error");

**exit**(1);

    }

    //步骤1创建socket

    sockfd=socket(AF\_INET,SOCK\_DGRAM,0);

**if**(sockfd<0){

**perror**("socket error");

**exit**(1);

    }

**int** ret;

**int** opt=1;

    //设置套接字选项，使刚刚停掉的端口可以立即使用，还可以添加超时

**if**(setsockopt(sockfd,SOL\_SOCKET,SO\_REUSEADDR,&opt,**sizeof**(opt))<0){

**perror**("setsockopt error");

**exit**(1);

    }

**struct** timeval timeout={10,0};

//即timeout={4,0};或者timeout.tv\_sec=4; timeout.tv\_usec=0;

//设置接收超时

setsockopt(sockfd,SOL\_SOCKET,SO\_RCVTIMEO,(**char**\*)&timeout,**sizeof**(**struct** timeval));

    //步骤2调用bind函数对socket和地址进行绑定

**struct** sockaddr\_in serveraddr;

**memset**(&serveraddr,0,**sizeof**(serveraddr));

    serveraddr.sin\_family=AF\_INET;

    serveraddr.sin\_port=htons(**atoi**(argv[1]));

    serveraddr.sin\_addr.s\_addr=INADDR\_ANY;

**if**(bind(sockfd,(**struct** sockaddr\*)&serveraddr,**sizeof**(serveraddr))<0){

**perror**("bind error");

**exit**(1);

    }

**while**(1){

        do\_service(sockfd);

    }

}

#### 客户端程序：

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<unistd.h>

**int** main(**int** argc, **char** \*argv[])

{

**if**(argc<3){

**printf**("usage:%s ip port\n",argv[0]);

**exit**(1);

    }

**int** sockfd=socket(AF\_INET,SOCK\_DGRAM,0);

**if**(sockfd<0){

**perror**("socket error");

**exit**(1);

    }

**struct** sockaddr\_in serveraddr;

**memset**(&serveraddr,0,**sizeof**(serveraddr));

    serveraddr.sin\_family=AF\_INET;

    serveraddr.sin\_port=htons(**atoi**(argv[2]));

    inet\_pton(AF\_INET,argv[1],&serveraddr.sin\_addr.s\_addr);

    /\*

        connect 并不会发起真正的连接，只是将服务器地址保存在内核中，这样以后发送

和接收函数都可以不指定服务器地址。也不会接受除该服务器以外的其他计算机发来的信息。

这样可以避免不必要的错误。

    \*/

**if**(connect(sockfd,(**struct** sockaddr\*)&serveraddr,**sizeof**(serveraddr))<0){

**perror**("connect error");

**exit**(1);

    }

**char** buffer[1024]="hello iotek";

    /\*if(sendto(sockfd,buffer,sizeof(buffer),0,(struct sockaddr\*)&serveraddr,sizeof(serveraddr))<0){

        perror("sendto error\n");

        exit(1);\*/

**if**(send(sockfd,buffer,**sizeof**(buffer),0)<0){

**perror**("send error\n");

**exit**(1);

    }**else** {

**memset**(buffer,0,**sizeof**(buffer));

        //已经有服务器信息了，不需要再指定

**if**(recv(sockfd,buffer,**sizeof**(buffer),0)<0){

**perror**("recv error");

**exit**(1);

        }**else**{

**printf**("%s",buffer);

        }

    }

    close(sockfd);

**return** 0;

}

实验：

chao@chao:~/linux/network/time\_udp$ ./time\_udp\_server 8888

recvfrom error //由于添加了超时未接收数据错误

: Resource temporarily unavailable //由于添加了超时未接收数据错误

client:127.0.0.1(38696) connected //客户端连接

client send info: hello iotek //客户端连接

recvfrom error

: Resource temporarily unavailable

chao@chao:~/linux/network/time\_udp$ ./time\_udp\_client 127.0.0.1 8888

Thu Dec 13 10:06:06 2018

### 5.3.4 域名解析

#### 本地虚拟机ip和域名对应关系：

chao@chao:~$ more /etc/hosts

127.0.0.1 localhost

127.0.1.1 chao-virtual-machine

#### 域名解析函数：

struct hostent{

char \*h\_name; //正式主机名

char \*\*h\_aliases; //别名，字符串数组

int h\_addrtype; //协议类型

int h\_length; //网络地址大小

char \*\*h\_addr\_list; //指向网络地址的指针

}

#include<netdb.h>

struct hostent \*gethostent(void);//循环获取/etc/hosts里面的每一行

//只获取匹配的行，但是该函数是线程不稳定的，容易出现问题，不建议用

struct hostent \*gethostbyname(const char \*hostname);

void sethostent(int stayopen);

void endhostent(void);

#### 给虚拟机设置域名

第一个chao 是主机名；第二个www.chao-ubuntu18.com.cn称为别名，还可以设置第三个别名，也就是有多个别名。

root@chao:/home/chao# gedit /etc/hosts

添加一行： 192.168.1.118 chao www.chao-ubuntu18.com.cn

域名解析编程实例：

#include<netdb.h>

#include<stdio.h>

#include <stdlib.h>

#include<memory.h>

#include <arpa/inet.h>

**void** out\_addr(**struct** hostent \*h)

{

**printf**("hsotname:%s \n",h->h\_name);

**printf**("addrtype:%s\n",h->h\_addrtype==AF\_INET?"IPV4":"IPV6");

**char** ip[16];

**memset**(ip,0,**sizeof**(ip));

    inet\_ntop(h->h\_addrtype,h->h\_addr\_list[0],ip,**sizeof**(ip));

**printf**("ip address:%s\n",ip);

}

**int** main(**int** argc,**char** \*argv[])

{

**if**(argc<2){

**fprintf**(stderr,"usage:%s host\n",argv[0]);

**exit**(1);

    }

**struct** hostent \*h;

**while**((h=gethostent())!=NULL){

**if**(!**strcmp**(argv[1],h->h\_name)){  //判断主机名是否相等

            out\_addr(h);

**exit**(0);

        }**else**{

**int** i=0;

**while**(h->h\_aliases[i]!=NULL){

**if**(!**strcmp**(argv[1],h->h\_aliases[i])){ //判断别名是否相等

                    out\_addr(h);

**exit**(0);

                }

                i++;

            }

        }

    }

    endhostent();//释放调用gethhostent 占用的资源

**printf**("no %s exist\n",argv[1]);

**return** 0;

}

#### 实验：

root@chao:/home/chao/linux/network/dns# gcc gethost2.c -o gethost2

root@chao:/home/chao/linux/network/dns# ./gethost2 chao

hsotname:chao

addrtype:IPV4

ip address:192.168.1.118

root@chao:/home/chao/linux/network/dns# ./gethost2 www.chao-ubuntu18.com.cn

hsotname:chao

addrtype:IPV4

ip address:192.168.1.118

### 5.3.5 通过域名访问网络编程

客户端程序需要更改，服务器程序不变。下面程序是由time\_udp\_client更改

#include <netdb.h>

#include<sys/socket.h>

#include <arpa/inet.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<memory.h>

#include<unistd.h>

**int** is\_host(**struct** hostent \*host,**char** \*name)

{

**if**(!**strcmp**(host->h\_name,name))

**return** 1;

**int** i=0;

**while**(host->h\_aliases[i]!=NULL){

**if**(!**strcmp**(host->h\_aliases[i],name))

**return** 1;

            i++;

        }

**return** 0;

}

unsigned **int** get\_ip\_by\_name(**char** \*name)

{

    unsigned **int** ip=0;

**struct** hostent \*host;

**while**((host=gethostent())!=NULL){

**if**(is\_host(host,name)){

**memcpy**(&ip,host->h\_addr\_list[0],4);

**break**;

        }

    }

    endhostent();

**return** ip;

}

**int** main(**int** argc, **char** \*argv[])

{

**if**(argc<3){

**printf**("usage:%s ip port\n",argv[0]);

**exit**(1);

    }

**int** sockfd=socket(AF\_INET,SOCK\_DGRAM,0);

**if**(sockfd<0){

**perror**("socket error");

**exit**(1);

    }

**struct** sockaddr\_in serveraddr;

**memset**(&serveraddr,0,**sizeof**(serveraddr));

    serveraddr.sin\_family=AF\_INET;

    serveraddr.sin\_port=htons(**atoi**(argv[2]));

    //inet\_pton(AF\_INET,argv[1],&serveraddr.sin\_addr.s\_addr);

    unsigned **int** ip=get\_ip\_by\_name(argv[1]);//得到网络字节序地址

**if**(ip!=0){

        serveraddr.sin\_addr.s\_addr=ip;

    }**else**{

        inet\_pton(AF\_INET,argv[1],&serveraddr.sin\_addr.s\_addr);

    }

 /\*

        connect 并不会发起真正的连接，只是将服务器地址保存在内核中，这样以后发送

和接收函数都可以不指定服务器地址。也不会接受除该服务器以外的其他计算机发来的信息。

这样可以避免不必要的错误。

    \*/

**if**(connect(sockfd,(**struct** sockaddr\*)&serveraddr,**sizeof**(serveraddr))<0){

**perror**("connect error");

**exit**(1);

    }

**char** buffer[1024]="hello iotek";

    /\*if(sendto(sockfd,buffer,sizeof(buffer),0,(struct sockaddr\*)&serveraddr,sizeof(serveraddr))<0){

        perror("sendto error\n");

        exit(1);\*/

**if**(send(sockfd,buffer,**sizeof**(buffer),0)<0){

**perror**("send error\n");

**exit**(1);

    }**else** {

**memset**(buffer,0,**sizeof**(buffer));

        //已经有服务器信息了，不需要再指定

**if**(recv(sockfd,buffer,**sizeof**(buffer),0)<0){

**perror**("recv error");

**exit**(1);

        }**else**{

**printf**("%s",buffer);

        }

    }

    close(sockfd);

**return** 0;

}

实验：

./time\_udp\_severce 8888

./time\_udp\_client chao 8888

### 5.3.6 广播

192.168.1.255是192.168.1子网的广播地址

# 多进程和多线程思考

如果总结多进程和多线程的区别，你肯定能说，前者开销大，后者开销较小。确实，这就是最基本的区别。线程函数的可重入性：  
说到函数的可重入，和线程安全，我偷懒了，引用网上的一些总结。  
  
线程安全：概念比较直观。一般说来，一个函数被称为线程安全的，当且仅当被多个并发线程反复调用时，它会一直产生正确的结果。  
  
可重入：概念基本没有比较正式的完整解释，但是它比线程安全要求更严格。根据经验，所谓“重入”，常见的情况是，程序执行到某个函数foo()时，收到信 号，于是暂停目前正在执行的函数，转到信号处理函数，而这个信号处理函数的执行过程中，又恰恰也会进入到刚刚执行的函数foo()，这样便发生了所谓的重 入。此时如果foo()能够正确的运行，而且处理完成后，之前暂停的foo()也能够正确运行，则说明它是可重入的。  
线程安全的条件：  
要确保函数线程安全，主要需要考虑的是线程之间的共享变量。属于同一进程的不同线程会共享进程内存空间中的全局区和堆，而私有的线程空间则主要包括栈和寄 存器。因此，对于同一进程的不同线程来说，每个线程的局部变量都是私有的，而全局变量、局部静态变量、分配于堆的变量都是共享的。在对这些共享变量进行访 问时，如果要保证线程安全，则必须通过加锁的方式。  
可重入的判断条件：  
  
要确保函数可重入，需满足一下几个条件：  
1、不在函数内部使用静态或全局数据   
2、不返回静态或全局数据，所有数据都由函数的调用者提供。   
3、使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据。  
4、不调用不可重入函数。  
  
可重入与线程安全并不等同，一般说来，可重入的函数一定是线程安全的，但反过来不一定成立。它们的关系可用下图来表示：  
  
比如：strtok函数是既不可重入的，也不是线程安全的；加锁的strtok不是可重入的，但线程安全；而strtok\_r既是可重入的，也是线程安全的。  
  
如果我们的线程函数不是线程安全的，那在多线程调用的情况下，可能导致的后果是显而易见的——共享变量的值由于不同线程的访问，可能发生不可预料的变化，进而导致程序的错误，甚至崩溃。