# Linux多路复用与多连接管理

实验名称：Linux多路复用与多连接管理

学时安排：2课时 指导老师：李赞

实验类别：验证型、设计型 实验要求：1人1组

学号： 姓名： 班级：

## 一、实验目的和任务

1.理解Linux系统TCP多连接的多种管理机制。

2.理解Linux系统通信不同应用场合的实现。

3.掌握使用信号量实现对进程运行同步控制。

## 二、实验设备介绍

1.软件需求： win10操作系统，VMware workstation，ubuntu18。

2.硬件需求: PC内存大于1G，硬盘空间大于20G。

## 三、注意事项和要求

1.理解Linux平台多种通信机制的特点和区别。

2.分析程序是否满足任务需求。

3.程序运行结果截屏后放入实验报告中。

## 四、实验内容和步骤

TCP是一对多的连接模式，在服务端管理多个socket对象的方式有多进程、多线程、select函数实现IO复用、epoll模式。

### 4.1Linux中创建新进程

创建进程可使用的方法有fork、system、exec函数调用。

#### 4.1.1调用fork函数创建新进程

fork函数复制正在运行的进程代码，新进程与原进程都执行fork函数后的语句， fork函数后面的代码都要执行，区别是系统将其分为父子进程。fork函数在父进程中返回值是子进程的PID值，fork函数在子进程中返回值则是0。一般父子进程的工作内容是不同，因此设一个if语句，根据fork返回值情况使他们执行不同的任务。

fork函数会在新旧进程中返回不同的值。

1）在父进程中，fork返回新创建子进程的进程ID；

2）在子进程中，fork返回0；

3）如果出现错误，fork返回一个负值；

//045.c fork创建进程1

#include<stdio.h>

#include<unistd.h>

int gval=10;

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

{

pid\_t pid;

int lval=20;

gval++,lval+=5;

pid=fork();

if(pid==0)

{

gval+=2,lval+=2;

printf("This is Child Proc:[%d,%d]\n",gval,lval);

printf("Child进程ID号为:[%d]\n",getpid());

printf("Child进程的父进程ID号为:[%d]\n",getppid());

}else

{

gval-=2,lval-=2;

printf("This is Parent Proc:[%d,%d]\n",gval,lval);

printf("Parent进程ID号为:[%d]\n",getpid());

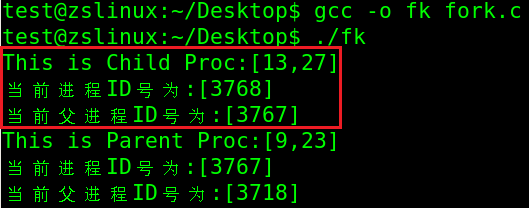
printf("Parent进程的父进程ID号为:[%d]\n",getppid());

}

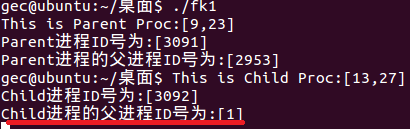
return 0;

}

该程序运行后执行fork方法后，系统将复制程序代码创建第二个相同的进程，两个进程对fork函数返回值是不同的，父进程fork方法得到的是子进程的pid，子进程fork方法返回值是0。所以输出的结果并非是if两个分支都执行了，而是父进程输出的是pid大于零的情部分，子进程输出的是pid为零的情况。



但是在ubuntu中运行得到结果却是有点意外，子进程获得的父进程ID是1，因为父进程执行完毕后就结束了，该进程就不存在了，init进程接管了子进程，这时子进程使用getppid得到的值是1。如果在父进程中添加sleep函数，子进程得到父进程ID。



//046.c fork创建进程2

#include<stdio.h>

#include<unistd.h>

int gval=10;

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

{

pid\_t pid;

int lval=20;

gval++,lval+=5;

pid=fork();

if(pid==0)

{//子进程执行

gval+=2,lval+=2;

printf("This is Child Proc:[%d,%d]\n",gval,lval);

printf("Child进程ID号为:[%d]\n",getpid());

printf("Child进程的父进程ID号为:[%d]\n",getppid());

}else

{//父进程执行

//先休息一会

sleep(1);

gval-=2,lval-=2;

printf("This is Parent Proc:[%d,%d]\n",gval,lval);

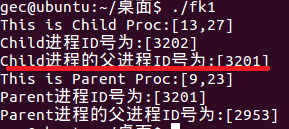
printf("Parent进程ID号为:[%d]\n",getpid());

printf("Parent进程的父进程ID号为:[%d]\n",getppid());

}

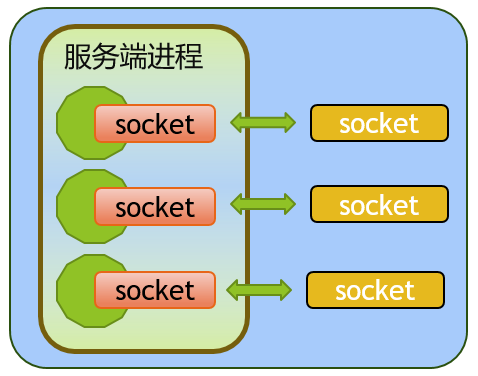
return 0;

}



### 4.2基于进程的并发服务器模型

每个客户端的连接服务端都创建子进程提供TCP服务，子进程复制父进程所有资源，子进程获得父进程套接字文件描述符。（主程序先建立一个全局Socket连接句柄后，fork()出一子进程，子进程获得了父进程数据空间、堆和栈的复制品。然后主子进程共同使用该连接句柄。这时此socket的引用计数为2, 任一进程关闭后对引用计数会-1,直到引用计数=0时，socket关闭。在某些情况下主进程需断开该socket并重新连接，此时此socket会无法断开。）在TCP多连接服务端可以采用创建并发进程的方式管理多个客户的连接。使用进程管理多个socket对象注意：1.注册进程的退出响应函数。



#### 4.2.1基于多进程的回声服务器echo\_mpserv.c

服务器端收到客户连接请求后，通过fork函数创建子进程，而每一个单独的进程处理socket通信。

// 055.c 基于fork方法多进程的回声服务《TCP/IP网络编程》page174

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <signal.h>

#include <sys/wait.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#define BUF\_SIZE 30

void error\_handling(char \*message);

//处理子进程的结束

void read\_childproc(int sig);

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

{

int serv\_sock,clnt\_sock;

struct sockaddr\_in serv\_adr,clnt\_adr;

pid\_t pid;

struct sigaction act;

socklen\_t adr\_sz;

int str\_len,state;

char buf[BUF\_SIZE];

if(argc!=2)

{

printf("Usage : %s <port>\n",argv[0]);

exit(1);

}

act.sa\_handler=read\_childproc;

sigemptyset(&act.sa\_mask);

act.sa\_flags=0;

//接收子进程的返回值

state=sigaction(SIGCHLD,&act,0);

//创建TCP服务端socket文件资源

serv\_sock=socket(PF\_INET,SOCK\_STREAM,0);

memset(&serv\_adr,0,sizeof(serv\_adr));

serv\_adr.sin\_family=AF\_INET;

serv\_adr.sin\_addr.s\_addr=htonl(INADDR\_ANY);

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

//绑定TCP监听端口

if(bind(serv\_sock,(struct sockaddr\*)&serv\_adr,sizeof(serv\_adr))==-1)

error\_handling("bind() error.");

//开启TCP监听

if(listen(serv\_sock,5)==-1)

error\_handling("listen() error.");

while(1)

{

adr\_sz=sizeof(clnt\_adr);

//每一个客户端连接，服务端都创建一个新socket资源用于配对通信

clnt\_sock=accept(serv\_sock,(struct sockaddr \*)&clnt\_adr,&adr\_sz);

if(clnt\_sock==-1) continue;

else puts("New client connected....");

pid=fork();

if(pid==-1)

{//线程创建失败

close(clnt\_sock);

continue;

}

if(pid==0)//子进程由此处运行，父进程不执行该段代码

{//子进程创建成功

close(serv\_sock);

//循环接收客户端发来的数据，直到发来长度为0

while((str\_len=read(clnt\_sock,buf,BUF\_SIZE))!=0)

//将客户端信息原样回复

write(clnt\_sock,buf,str\_len);

close(clnt\_sock);

puts("clinet disconnected...");

return 0;

}

else close(clnt\_sock);//创建子进程失败，清除socket资源

}

//结束监听，终止TCP服务。

close(serv\_sock);

return 0;

}

//处理子进程的结束

void read\_childproc(int sig)

{

int status;

//获得子进程的返回值

pid\_t id=waitpid(-1,&status,WNOHANG);

if(WIFEXITED(status))

{

printf("Removed proc id: %d\n",id);

printf("Child send:%d \n",WEXITSTATUS(status));

}

}

//错误处理是将信息输出到标准输出，然后退出进程。

void error\_handling(char \* message)

{

fputs(message,stderr);

fputc('\n',stderr);

exit(1);

}

#### 4.2.2回声客户端程序echo\_client.c

为方便与服务端配合运行，将回声客户端程序在此列出，通过启动多个客户端实例来验证并发服务器功能。

// 056.c基于fork方法多进程的回声客户端《TCP/IP网络编程》page75

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#define BUF\_SIZE 1024

void error\_handling(char \*message);

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

{

int sock;

char message[BUF\_SIZE];

int str\_len;

struct sockaddr\_in serv\_adr;

if(argc!=3)

{

printf("Usage : %s <IP> <port>\n",argv[0]);

exit(1);

}

//创建TCP客户端socket文件资源

sock=socket(PF\_INET,SOCK\_STREAM,0);

if(sock==-1)

error\_handling("socket() 创建失败.");

memset(&serv\_adr,0,sizeof(serv\_adr));

serv\_adr.sin\_family=AF\_INET;

serv\_adr.sin\_addr.s\_addr=inet\_addr(argv[1]);

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

//连接服务端

if(connect(sock,(struct sockaddr\*)&serv\_adr,sizeof(serv\_adr))==-1)

error\_handling("connect() failed.");

else//连接成功

puts("Connected success .....");

while(1)

{

fputs("Input message(Q to quit):",stdout);

fgets(message,BUF\_SIZE,stdin);

if(!strcmp(message,"q\n")||!strcmp(message,"Q\n"))

break;//收到q字符则退出

write(sock,message,strlen(message));

str\_len=read(sock,message,BUF\_SIZE-1);

message[str\_len]=0;

printf("Message from server: %s",message);

}

close(sock);

return 0;

}

//错误处理是将信息输出到标准输出，然后退出进程。

void error\_handling(char \* message)

{

fputs(message,stderr);

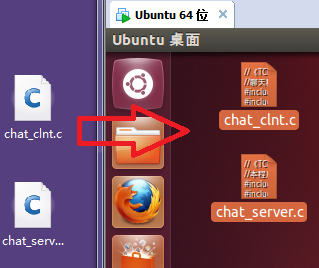
fputc('\n',stderr);

exit(1);

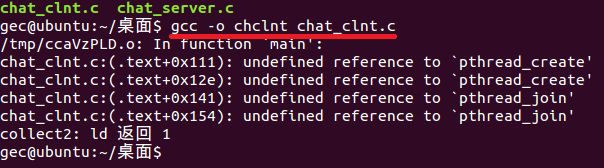
}

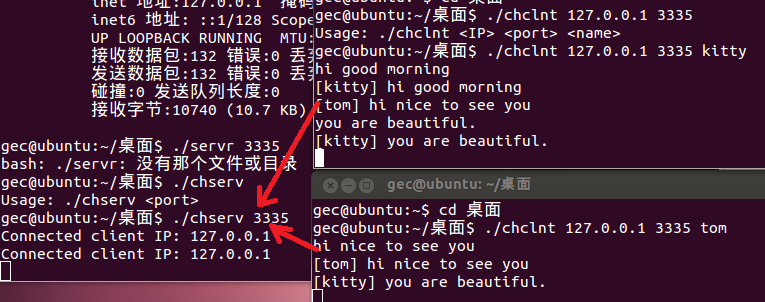
### 4.3使用线程的网络程序

Linux平台还支持通过创建线程执行对每个客户端的socket通讯任务。启动ubuntu64系统，安装的是ubuntu18。



直接使用gcc命令会出现编译不成功信息，编译线程需要添加 -lpthread选项。





#### 4.3.1聊天客户端线程分离发送和接收

在TCP多连接服务端可以采用创建并发线程的方式管理多个客户的连接。使用线程管理多个socket对象注意：1.每个线程要使用detach方式进行分离，线程结束会自动清除其资源。

// 060.c 使用线程的聊天客户端《TCP/IP网络编程》page310 chat\_clnt.c

//聊天程序客户端

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#include <pthread.h>

#define BUF\_SIZE 100

#define NAME\_SIZE 20

void\* recv\_msg(void \*arg);

void\* send\_msg(void \*arg);

void error\_handling(char \*msg);

//全局变量

char name[NAME\_SIZE]="[default]";

char msg[BUF\_SIZE];

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

{

int sock;

struct sockaddr\_in serv\_adr;

pthread\_t snd\_thread,rcv\_thread;

void \* thread\_return;

if(argc!=4)

{

printf("Usage: %s <IP> <port> <name>\n",argv[0]);

exit(1);

}

sprintf(name,"[%s]",argv[3]);

//用于连接和通讯的socket资源

sock=socket(PF\_INET,SOCK\_STREAM,0);

memset(&serv\_adr,0,sizeof(serv\_adr));

serv\_adr.sin\_family=AF\_INET;

serv\_adr.sin\_addr.s\_addr=inet\_addr(argv[1]);

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

//连接目标主机

if(connect(sock,(struct sockaddr\*)&serv\_adr,sizeof(serv\_adr))==-1)

error\_handling("connect() error");

//创建一个线程管理发送数据

pthread\_create(&snd\_thread,NULL,send\_msg,(void \*)&sock);

//创建一个线程管理接收数据

pthread\_create(&rcv\_thread,NULL,recv\_msg,(void \*)&sock);

//等待发送线程结束

pthread\_join(snd\_thread,&thread\_return);

//等待接收线程结束

pthread\_join(rcv\_thread,&thread\_return);

//关闭socket文件资源

close(sock);

return 0;

}

//发送线程代码

void \* send\_msg(void \* arg)

{

int sock=\*((int \*)arg);

char name\_msg[NAME\_SIZE+BUF\_SIZE];

while(1)

{

fgets(msg,BUF\_SIZE,stdin);

if(!strcmp(msg,"q\n")||!strcmp(msg,"Q\n"))

{

close(sock);

exit(0);

}

sprintf(name\_msg,"%s %s",name,msg);

write(sock,name\_msg,strlen(name\_msg));

}

return NULL;

}

//接收信息线程代码

void \*recv\_msg(void \*arg)

{

int sock=\*((int \*)arg);

char name\_msg[NAME\_SIZE+BUF\_SIZE];

int str\_len;

while(1)

{

str\_len=read(sock,name\_msg,NAME\_SIZE+BUF\_SIZE-1);

if(str\_len==-1)

return (void\*)-1;

name\_msg[str\_len]=0;

fputs(name\_msg,stdout);

}

return NULL;

}

//输出出错信息后结束本进程

void error\_handling(char \* message)

{

fputs(message,stderr);

fputc('\n',stderr);

exit(1);

}

#### 4.3.2 多线程并发服务器端实现

下面程序的服务端针对每个客户连接请求，服务端都创建一个新线程进行通讯，更具体点描述是用同一个线程代码生成多个线程体，所以每个通讯的逻辑是完全一样的，但每个线程体使用的资源不一样。

// 061.c 使用线程的并发服务端《TCP/IP网络编程》page307 chat\_server.c

//本程序执行socket监听，信息发送操作

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#include <pthread.h>

#define BUF\_SIZE 100

#define MAX\_CLNT 256

void \* handle\_clnt(void \*arg);

void send\_msg(char \*msg,int len);

void error\_handling(char \*msg);

//全局变量

int clnt\_cnt=0;

int clnt\_socks[MAX\_CLNT];

pthread\_mutex\_t mutx;

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

{

int serv\_sock,clnt\_sock;

struct sockaddr\_in serv\_adr,clnt\_adr;

int clnt\_adr\_sz;

pthread\_t t\_id;

if(argc!=2)

{

printf("Usage: %s <port>\n",argv[0]);

exit(1);

}

pthread\_mutex\_init(&mutx,NULL);

//用于监听的socket资源

serv\_sock=socket(PF\_INET,SOCK\_STREAM,0);

memset(&serv\_adr,0,sizeof(serv\_adr));

serv\_adr.sin\_family=AF\_INET;

serv\_adr.sin\_addr.s\_addr=htonl(INADDR\_ANY);

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

//绑定监听端口

if(bind(serv\_sock,(struct sockaddr\*)&serv\_adr,sizeof(serv\_adr))==-1)

error\_handling("bind() error");

//执行监听

if(listen(serv\_sock,5)==-1)

error\_handling("listen() error");

while(1)

{

clnt\_adr\_sz=sizeof(clnt\_adr);

//每个客户连接请求，服务端都创建一个新socket资源用于通讯

clnt\_sock=accept(serv\_sock,(struct sockaddr\*)&clnt\_adr,&clnt\_adr\_sz);

pthread\_mutex\_lock(&mutx);

clnt\_socks[clnt\_cnt++]=clnt\_sock;

pthread\_mutex\_unlock(&mutx);

//创建一个线程管理与客户端的socket通讯

pthread\_create(&t\_id,NULL,handle\_clnt,(void \*)&clnt\_sock);

pthread\_detach(t\_id);

printf("Connected client IP: %s \n",inet\_ntoa(clnt\_adr.sin\_addr));

}

//关闭监听socket文件资源

close(serv\_sock);

return 0;

}

//线程代码

void \* handle\_clnt(void \* arg)

{

int clnt\_sock=\*((int \*)arg);

int str\_len=0,i;

char msg[BUF\_SIZE];

while((str\_len=read(clnt\_sock,msg,sizeof(msg)))!=0)

send\_msg(msg,str\_len);

pthread\_mutex\_lock(&mutx);

for(i=0;i<clnt\_cnt;i++)//remove disconnected client

{

if(clnt\_sock==clnt\_socks[i])

{

while(i++<clnt\_cnt-1)//向前移动数据元素

clnt\_socks[i]=clnt\_socks[i+1];

break;

}

}

clnt\_cnt--;

pthread\_mutex\_unlock(&mutx);

close(clnt\_sock);

return NULL;

}

void send\_msg(char \* msg,int len)

{

int i;

pthread\_mutex\_lock(&mutx);

for(i=0;i<clnt\_cnt;i++)

write(clnt\_socks[i],msg,len);

pthread\_mutex\_unlock(&mutx);

}

//输出出错信息后结束本进程

void error\_handling(char \* message)

{

fputs(message,stderr);

fputc('\n',stderr);

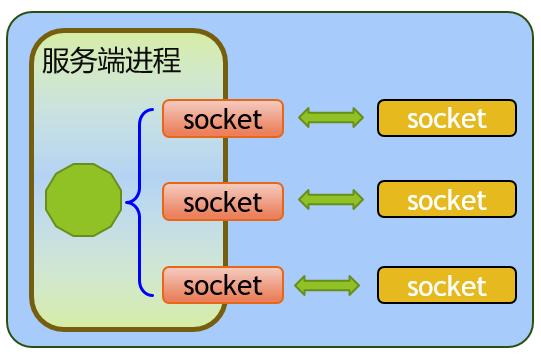
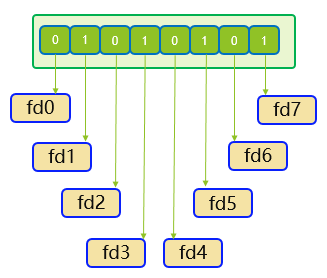
exit(1);

}

该服务程序最大的问题是没有设置退出响应，里面是个死循环。

### 4.4 多并发连接的select函数复用echo\_selectserv.c

在TCP通讯过程中，对每个客户端连接请求创建新进程代价大，进程间数据交换也需要复杂的方法。还可以通过复用的模式提供通讯，就是用一个进程向多个客户端提供服务。

#include <sys/select.h>

#inlcude <sys/time.h>

int select(int maxfd,fd\_set \* readset,fd\_set \*writeset,fd\_set \* exceptset,const struct timeval \* timeout);

fd\_set类型变量每一位代表了一个描述符。返回值-1代表发生了错误，返回0表示超时，大于0的值表示发事件的文件数量。select函数是阻塞运行，直到超时或发生事件。maxfd指示文件集合的最大值(+1)，它规定了有效的位的个数(每个位对应一个文件)，有上限要求例如512\*8个。

#### 4.4.1 echo\_selectserv.c

select函数 实现I/O复用服务器端，下面是基于I/O复用的回声服务器代码。

//037.c 文件传输接收端《TCP/IP网络编程》page 203 echo\_selectserv.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <sys/select.h>

#define BUF\_SIZE 100

void error\_handling(char \*message);

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

{

int serv\_sock,clnt\_sock;

struct sockaddr\_in serv\_adr,clnt\_adr;

struct timeval timeout;

fd\_set reads,cpy\_reads;

socklen\_t adr\_sz;

int fd\_max,str\_len,fd\_num,i;

char buf[BUF\_SIZE];

if(argc!=2)

{

printf("Usage : %s <port>\n",argv[0]);

exit(1);

}

//创建TCP客户端socket文件资源

serv\_sock=socket(PF\_INET,SOCK\_STREAM,0);

if(serv\_sock==-1)

error\_handling("socket() 创建失败.");

memset(&serv\_adr,0,sizeof(serv\_adr));

serv\_adr.sin\_family=AF\_INET;

serv\_adr.sin\_addr.s\_addr=htonl(INADDR\_ANY);

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

if(bind(serv\_sock,(struct sockaddr \*)&serv\_adr,sizeof(serv\_adr))==-1)

error\_handling("bind() error.");

if(listen(serv\_sock,5)==-1)

error\_handling("listen() error.");

FD\_ZERO(&reads);

FD\_SET(serv\_sock,&reads);

fd\_max=serv\_sock;

//采用定时循环的方法，检测每个socket资源，是否需要处理相应，会有空转的情况

//这种方式会产生CPU浪费，比如socket参数传递到内核，但是该socket没有事情处理

while(1)

{

cpy\_reads=reads;

timeout.tv\_sec=5;

timeout.tv\_usec=5000;

if((fd\_num=select(fd\_max+1,&cpy\_reads,0,0,&timeout))==-1)

break;

if(fd\_num==0) continue;

for(i=0;i<fd\_max+1;i++)

{

if(FD\_ISSET(i,&cpy\_reads))

{

if(i==serv\_sock)//connection request

{

adr\_sz=sizeof(clnt\_adr);

clnt\_sock=accept(serv\_sock,(struct sockaddr\*)&clnt\_adr,&adr\_sz);

FD\_SET(clnt\_sock,&reads);

if(fd\_max<clnt\_sock) fd\_max=clnt\_sock;

printf("connected client:%d \n",clnt\_sock);

}else

{

str\_len=read(i,buf,BUF\_SIZE);

if(str\_len==0)

{

FD\_CLR(i,&reads);

close(i);

printf("closed client:%d \n",i);

}else

{

write(i,buf,str\_len);

}

}

}

}

}

close(serv\_sock);

return 0;

}

//错误处理是将信息输出到标准输出，然后退出进程。

void error\_handling(char \* buf)

{

fputs(buf,stderr);

fputc('\n',stderr);

exit(1);

}

#### 4.4.2 echo\_client.c

这个是与echo\_selectserv.c配合运行的客户端程序。

//038.c 客户端程序《TCP/IP网络编程》page75

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#define BUF\_SIZE 1024

void error\_handling(char \*message);

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

{

int sock;

char message[BUF\_SIZE];

int str\_len;

struct sockaddr\_in serv\_adr;

if(argc!=3)

{

printf("Usage : %s <IP> <port>\n",argv[0]);

exit(1);

}

//创建TCP客户端socket文件资源

sock=socket(PF\_INET,SOCK\_STREAM,0);

if(sock==-1)

error\_handling("socket() 创建失败.");

memset(&serv\_adr,0,sizeof(serv\_adr));

serv\_adr.sin\_family=AF\_INET;

serv\_adr.sin\_addr.s\_addr=inet\_addr(argv[1]);

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

//连接服务端

if(connect(sock,(struct sockaddr\*)&serv\_adr,sizeof(serv\_adr))==-1)

error\_handling("connect() failed.");

else//连接成功

puts("Connected success .....");

while(1)

{

fputs("Input message(Q to quit):",stdout);

fgets(message,BUF\_SIZE,stdin);

if(!strcmp(message,"q\n")||!strcmp(message,"Q\n"))

break;//收到q字符则退出

write(sock,message,strlen(message));

str\_len=read(sock,message,BUF\_SIZE-1);

message[str\_len]=0;

printf("Message from server: %s",message);

}

close(sock);

return 0;

}

//错误处理是将信息输出到标准输出，然后退出进程。

void error\_handling(char \* message)

{

fputs(message,stderr);

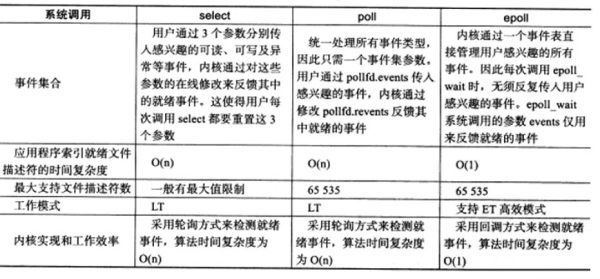
fputc('\n',stderr);

exit(1);

}

select模式支持多个连接，当某连接上有数据有读写要求时将产生事件，每次循环检测到这个事件后就进行实际的读写操作，这样的方式看起问题不大，但实际上内核处理过程会频繁进行参数传递来确定socket文件上是否有事件，没有事件的情况下都浪费了传值操作，select模式没有发挥内核最大的性能。

### 4.5 支持大批量并发连接的epoll模式



epoll模式下操作系统负责保存监视对象文件描述符，epoll\_wait函数等待文件描述符的变化，并且epoll方式将发生变化的文件描述符专门集中在一起，这样能大大提高程序执行效率，并降低资源消耗，epoll是Linux目前大规模网络并发程序开发的首选模型，经过测试配合ulimit -a命令，epoll模式支持65534个连接。

select在BSD里面1984年实现的，1997年才实现了poll，当时硬件实在太弱，一台服务器处理1千多个链接简直就是神一样的；2002, 大神 Davide Libenzi 实现了epoll。

Linux中提供的epoll相关函数如下：

int epoll\_create(int size);

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

epoll\_create函数创建一个epoll机制。

**epoll\_ctl**函数注册要监听的事件类型，事件类型有：

EPOLL\_CTL\_ADD 注册新的fd到epfd中

EPOLL\_CTL\_MOD 修改已注册的fd的监听事件

EPOLL\_CTL\_DEL 从epfd中删除一个fd

epoll\_wait等待事件的就绪，成功时返回就绪的事件数目，调用失败时返回 -1，等待超时返回 0。

//039.c《TCP/IP网络编程》page 279 epoll\_serv.c 交互 echo\_client.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#include <sys/epoll.h>

#include <fcntl.h>

#include <errno.h>

#define BUF\_SIZE 100

#define EPOLL\_SIZE 50

void setnonblockingmode(int fd);

void error\_handling(char \* message);

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

{

int server\_sock,clnt\_sock;

struct sockaddr\_in serv\_addr,clnt\_adr;

socklen\_t adr\_sz;

int str\_len,i;

char buf[BUF\_SIZE];

struct epoll\_event \*ep\_events;

struct epoll\_event event;

int epfd,event\_cnt;

if(argc!=2)

{

printf("Usage :%s <PORT>\n",argv[0]);

exit(1);

}

server\_sock=socket(PF\_INET,SOCK\_STREAM,0);

if(server\_sock==-1)

error\_handling("socket() create failed.");

memset(&serv\_addr,0,sizeof(serv\_addr));

serv\_addr.sin\_family=AF\_INET;

//IP地址

serv\_addr.sin\_addr.s\_addr=htonl(INADDR\_ANY);

//PORT

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

if(bind(server\_sock,(struct sockaddr\*)&serv\_addr,sizeof(serv\_addr))==-1)

error\_handling("bind() failed.");

if(listen(server\_sock,5)==-1)

error\_handling("listen() failed.");

epfd=epoll\_create(EPOLL\_SIZE);

//分配EPOLL\_SIZE个epoll\_event用于记录事件

ep\_events=malloc(sizeof(struct epoll\_event)\*EPOLL\_SIZE);

printf("ep\_events=malloc(). \n");

//设置socket为非阻塞模式

setnonblockingmode(server\_sock);

event.events=EPOLLIN;

event.data.fd=server\_sock;

epoll\_ctl(epfd,EPOLL\_CTL\_ADD,server\_sock,&event);

while(1)

{

event\_cnt=epoll\_wait(epfd,ep\_events,EPOLL\_SIZE,500);

if(event\_cnt==-1)

{

puts("epoll\_wait() error.");

break;

}

//printf("return epoll\_wait(). event\_cnt=%d\n",event\_cnt);

//遍历所有ep\_events对象

for(i=0;i<event\_cnt;i++)

{

//是监听socket对象事件则执行accept方法

if(ep\_events[i].data.fd==server\_sock)

{

adr\_sz=sizeof(clnt\_adr);

//每次accept会创建一个新socket用于数据通信

clnt\_sock=accept(server\_sock,(struct sockaddr\*)&clnt\_adr,&adr\_sz);

printf("connected client ID=: %d \n",clnt\_sock);

//设置socket为非阻塞模式

setnonblockingmode(clnt\_sock);

event.events=EPOLLIN|EPOLLET;

event.data.fd=clnt\_sock;

//注册监视对象文件描述符

epoll\_ctl(epfd,EPOLL\_CTL\_ADD,clnt\_sock,&event);

}else

{//是普通socket对象则执行读操作

str\_len=read(ep\_events[i].data.fd,buf,BUF\_SIZE);

if(str\_len==0)

{//收到字节长度为0，表示对方shutdown，则执行close操作

epoll\_ctl(epfd,EPOLL\_CTL\_DEL,ep\_events[i].data.fd,NULL);

close(ep\_events[i].data.fd);

printf("closed client:%d \n",ep\_events[i].data.fd);

break;

}else if(str\_len<0)

{//收到字节长度小于0

if(errno==EAGAIN) break;

}else

{//收到字节度大于0，执行回复操作

printf("Message from %d. :%s",ep\_events[i].data.fd,buf);

write(ep\_events[i].data.fd,buf,str\_len);

}

}

}

}

close(server\_sock);

close(epfd);

return 0;

}

void setnonblockingmode(int fd)

{

int flag=fcntl(fd,F\_GETFL,0);

fcntl(fd,F\_SETFL,flag|O\_NONBLOCK);

}

//错误处理是将信息输出到标准输出，然后退出进程。

void error\_handling(char \* message)

{

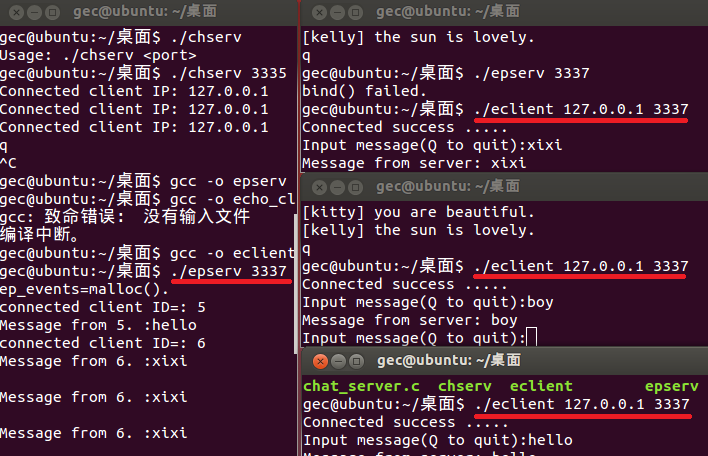
fputs(message,stderr);

fputc('\n',stderr);

exit(1);

}

下面是epoll模式运行示意图。



## 五、程序作业

### 5.1 调试运行聊天程序，将程序运行截图提交到学习通平台。