桂林电子科技大学

**实验七 进程控制与通信**  实验报告

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | **实验7 进程控制与通信** | | | | | | | |  | 辅导员意见：  成绩 辅导员  签 名 |
| 院 系 | 计算机与信息安全学院 | | | 专业 | | 计算机科学与技术 | | |
| 学 号 | 1600300128 | | | 姓名 | | 王涛 | | |
| 实验日期 | 2018 | 年 | 12 | | 月 | | 28 | 日 |
|  |  | | | | | | | |

## 实验目的

1、 理解Linux下的进程控制和通信机制；

2、 掌握编写C语言程序实现Linux进程管理和通信；

## 实验要求

1、理解掌握进程基本概念、进程属性和一些特殊进程；

2、掌握使用linux进程管理API，包括进程创建、执行新代码、进程退出等；

3、掌握同主机进程间的管道通信进制；

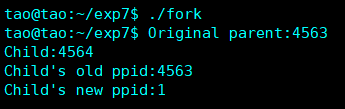
4、理解信号和信号集。

## 实验步骤

### 一、编辑、调试、运行以下程序

1、运行fork.c，理解系统调用fork，记录实验结果并对实验结果进行解析

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | #include <unistd.h>  #include <sys/types.h>  #include <stdio.h>  #include <errno.h>  int main(int argc,char \*\*argv)  {  pid\_t pid,old\_ppid,new\_ppid;  pid\_t child,parent;  parent = getpid();  if((child = fork()) < 0)  {  fprintf(stderr,"%s:fork of child  failed:%s\n",argv[0],strerror(errno));  exit(1);  }  else if(child == 0)  {  old\_ppid = getppid();  sleep(2);  new\_ppid = getppid();  }  else  {  sleep(1);  exit(0);  }  printf("Original parent:%d\n",parent);  printf("Child:%d\n",getpid());  printf("Child's old ppid:%d\n",old\_ppid);  printf("Child's new ppid:%d\n",new\_ppid);  exit(0);  } |



解析：调用fork函数之前先记录原始父进程的id结果是4563；

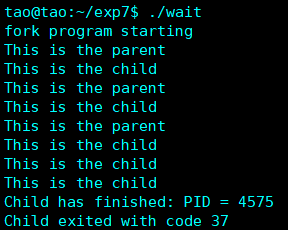
调用fork创建子进程之后子进程执行的是记录父进程id后睡眠2秒在获取父进程id；调用fork之后父进程睡眠1秒后运行结束；

所以子进程第一次获取父进程id时父进程未运行结束记录的父进程id为4563，而第二次记录时父进程已经运行结束子进程的父进程变成了init,故getppid()得到的结果为1；

最后getpid()输出的是子进程的id为4564。

2、运行wait.c，理解wait系统调用，记录实验结果，并对实验结果进行解析。

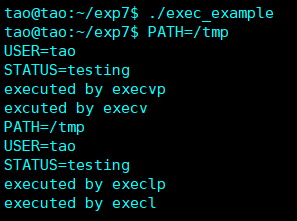
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | #include <sys/types.h>  #include <sys/wait.h>  #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  int main()  {  pid\_t pid;  char \*message;  int n;  int exit\_code;  printf("fork program starting\n");  pid = fork();  switch(pid)  {  case -1:  exit(1);  case 0:  message = "This is the child";  n = 5;  exit\_code = 37;  break;  default:  message = "This is the parent";  n = 3;  exit\_code = 0;  break;  }  for(; n > 0; n--) {  puts(message);  sleep(1);  }  /\* This section of the program waits for the child process to finish. \*/  if(pid) {  int stat\_val;  pid\_t child\_pid;  child\_pid = wait(&stat\_val);  printf("Child has finished: PID = %d\n", child\_pid);  if(WIFEXITED(stat\_val))  printf("Child exited with code %d\n", WEXITSTATUS(stat\_val));  else  printf("Child terminated abnormally\n");  }  exit (exit\_code);  } |



解析：调用fork()之后子进程设置n=5，父进程设置n=3，父进程循环输出三次，子进程循环输出5次，每次睡眠1s。父进程先输出完执行到wait之后等待子进程执行结束。子进程如果执行exit(exit\_code) 正常结束会将exit\_code传给父进程，WIFEXITED宏用来判断子进程是否正常退出，正常退出后WEXITSTATUS宏用来判段进程状态得到的结果就是子进程的exit\_code即37。

3、运行exec\_example.c，理解exec函数族，记录实验结果并对实验结果进行解析

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | #include <unistd.h>  int main(void)  {  char \*envp[] = {"PATH=/tmp", "USER=zhangruixia", "STATUS=testing", NULL};  char \*argv\_execv[] = {"echo", "excuted by execv", NULL};  char \*argv\_execvp[] = {"echo", "executed by execvp", NULL};  char \*argv\_execve[] = {"env", NULL};  if (fork() == 0)  if (execl("/bin/echo", "echo", "executed by execl", NULL) < 0)  perror("Err on execl");  if (fork() == 0)  if (execlp("echo", "echo", "executed by execlp", NULL) < 0)  perror("Err on execlp");  if (fork() == 0)  if (execle("/usr/bin/env", "env", NULL, envp) < 0)  perror("Err on execle");  if (fork() == 0)  if (execv("/bin/echo", argv\_execv) < 0)  perror("Err on execv");  if (fork() == 0)  if (execvp("echo", argv\_execvp) < 0)  perror("Err on execvp");  if (fork() == 0)  if (execve("/usr/bin/env", argv\_execve, envp) < 0)  perror("Err on execve");  } |



解析：此程序对exec函数族功能示例，只有第6个是系统调用，1-5是封装的库函数。

exec前三个接着l，表示list，直接列举参数，后三个接着v，表示vector故先定义参数向量传参；

字母p是指在环境变量PATH的目录里去查找要执行的可执行文件故只写可执行文件的名字，字母e是指给可执行文件指定环境变量。

### 二、根据要求设计编写程序

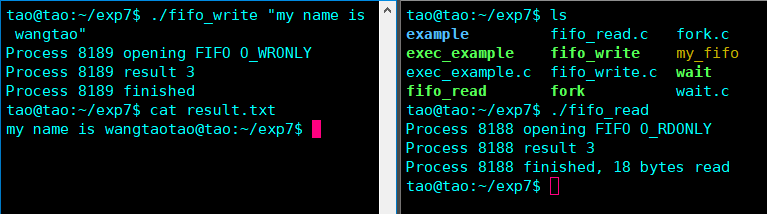
编写程序，要求使用FIFO实现两个进程间的通信，写进程fifo\_write.c实现发送数据到有名管道，读进程fifo\_read.c从有名管道中读取数据，缓冲区数据设置为“my name is 自己的姓名”。

fifo\_write.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  #include <fcntl.h>  #include <limits.h>  #include <sys/types.h>  #include <sys/stat.h>  #define FIFO\_NAME "./my\_fifo"  #define BUFFER\_SIZE 100  int main(int argc, char\*\* argv)  {  int pipe\_fd;  int res;  int open\_mode = O\_WRONLY;  if (access(FIFO\_NAME, F\_OK) == -1) {  res = mkfifo(FIFO\_NAME, 0777);  if (res != 0) {  fprintf(stderr, "Could not create fifo %s\n", FIFO\_NAME);  exit(EXIT\_FAILURE);  }  }  printf("Process %d opening FIFO O\_WRONLY\n", getpid());  pipe\_fd = open(FIFO\_NAME, open\_mode);  printf("Process %d result %d\n", getpid(), pipe\_fd);  if (pipe\_fd != -1) {  res = write(pipe\_fd, argv[1], strlen(argv[1]));  if (res == -1) {  fprintf(stderr, "Write error on pipe\n");  exit(EXIT\_FAILURE);  }  close(pipe\_fd);  }  else {  exit(EXIT\_FAILURE);  }  printf("Process %d finished\n", getpid());  exit(EXIT\_SUCCESS);  } |

fifo\_read.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  #include <fcntl.h>  #include <limits.h>  #include <sys/types.h>  #include <sys/stat.h>  #define FIFO\_NAME "./my\_fifo"  #define BUFFER\_SIZE PIPE\_BUF  int main()  {  int pipe\_fd;  int res;  int res\_fd;  int open\_mode = O\_RDONLY;  char buffer[BUFFER\_SIZE + 1];  int bytes\_read = 0;  memset(buffer, '\0', sizeof(buffer));  printf("Process %d opening FIFO O\_RDONLY\n", getpid());  pipe\_fd = open(FIFO\_NAME, open\_mode);  printf("Process %d result %d\n", getpid(), pipe\_fd);  res\_fd = open("result.txt", O\_WRONLY|O\_CREAT, 0644);  if (pipe\_fd != -1) {  do {  res = read(pipe\_fd, buffer, BUFFER\_SIZE);  bytes\_read += res;  write(res\_fd, buffer, res);  } while (res > 0);  close(pipe\_fd);  close(res\_fd);  } else {  exit(EXIT\_FAILURE);  }  printf("Process %d finished, %d bytes read\n", getpid(), bytes\_read);  exit(EXIT\_SUCCESS);  } |



### 三、实验扩展：进程间的通信方式

1、IPC机制：信号、共享内存和消息队列

(1)使用信号和共享内存实现生产消费者问题：

producer.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #include <stdlib.h>  #include <stdio.h>  #include <sys/shm.h>  #include <string.h>  #include "sem\_func.h"  #define ID 0xFF  #define ID1 0x111  #define EMPTY 0  #define FULL 1  int main(int argc,char \*argv[])  {  key\_t key\_id,shm\_id,sem\_key,sems;  char \*buf;  key\_id = ftok(argv[1],ID); //建立IPC通讯时指定的唯一ID值  if(key\_id == -1)  exit(1);  shm\_id = shmget(key\_id, 1024, IPC\_CREAT|IPC\_EXCL|0666); //创建一个共享内存对象返回标识id，用于存放生产物品的缓冲区  buf = (char \*)shmat(shm\_id,NULL,0); //把共享内存区对象映射到调用进程的地址空间  sem\_key = ftok(argv[1],ID1);  if(sem\_key == -1)  exit(1);  sems = semget(sem\_key, 2, IPC\_CREAT|IPC\_EXCL|0666); //创建信号集（0：empty；1：full)  init\_semaphore(sems,EMPTY,1); //初始化empty信号  init\_semaphore(sems,FULL,0); //初始化full信号  while(1){  P(sems,EMPTY);  printf("Set:");  scanf("%s",buf); //输入信号存入共享内存  if(strcmp(buf,"no")==0){  V(sems,FULL);  shmctl(shm\_id,IPC\_RMID,NULL);  semctl(sems,EMPTY,IPC\_RMID);  semctl(sems,FULL,IPC\_RMID);  break;  }  V(sems,FULL);  }  return 0;  } |

consumer.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #include <stdlib.h>  #include <stdio.h>  #include <sys/shm.h>  #include <string.h>  #include "sem\_func.h"  #define ID 0xFF  #define ID1 0x111  #define EMPTY 0  #define FULL 1  int main(int argc, char \*argv[])  {  key\_t key\_id, sem\_key, sems;  key\_id = ftok(argv[1], ID);  key\_t shm\_id = shmget(key\_id,0,0);  char \*buf = (char \*)shmat(shm\_id, NULL, 0); //取得存放生产物品共享内存  sem\_key = ftok(argv[1],ID1);  sems = semget(sem\_key,0,0); //取得信号集  while(1)  {  P(sems,FULL);  printf("Get:%s\n",buf);  if(strcmp(buf,"no") == 0){  V(sems,EMPTY);  break;  }  V(sems,EMPTY);  }  return 0;  } |

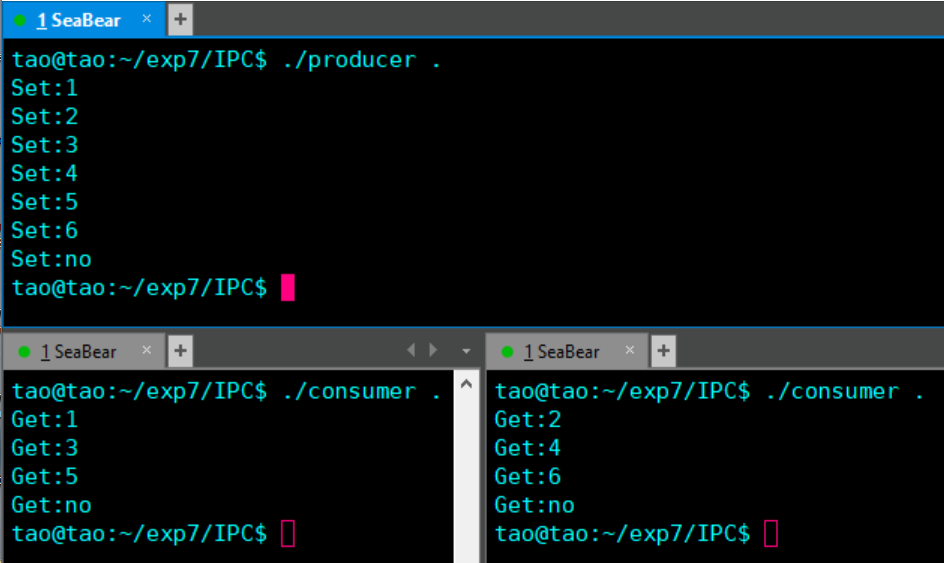
sem\_func.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #ifndef SEM\_FUNC\_H  #define SEM\_FUNC\_H  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  union semun  {  int val;  struct semid\_ds \*buf;  unsigned short int \*array;  struct seminfo \*\_\_buf;  };  void init\_semaphore(key\_t sid,int semnum,int val);  int P(key\_t semid, int semnum);  int V(key\_t semid, int semnum);  #endif |

sem\_func.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #include "sem\_func.h"  //初始化信号  void init\_semaphore(key\_t sid,int semnum,int val)  {  union semun semopts;  semopts.val = val;  semctl(sid,semnum,SETVAL,semopts);  }  //P操作  int P(key\_t semid, int semnum)  {  struct sembuf semOp ;  semOp.sem\_num = semnum;  semOp.sem\_op = -1;  semOp.sem\_flg = SEM\_UNDO;  return semop(semid, &semOp, 1);  }  //V操作  int V(key\_t semid, int semnum)  {  struct sembuf semOp ;  semOp.sem\_num = semnum;  semOp.sem\_op = 1;  semOp.sem\_flg = SEM\_UNDO;  return semop(semid, &semOp, 1);  } |

结果：



(2)使用消息队列实现简单聊天室

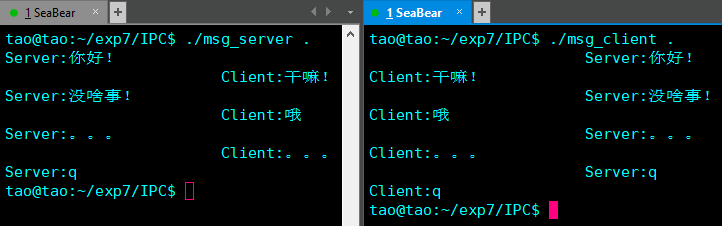
msg\_server.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <string.h>  #include <stdio.h>  #include <stdlib.h>  #define ID 0x111  #define MSGSND 200  #define MSGRCV 100  typedef struct Msg{  long type;  char content[256];  }Msg;  int main(int argc, char \*argv[])  {  Msg msg;  key\_t msg\_key,msg\_id;  msg\_key = ftok(argv[1],ID);  if(msg\_key == -1)  exit(1);  msg\_id = msgget(msg\_key,IPC\_CREAT|IPC\_EXCL|0666); //创建消息队列  while(1)  {  printf("Server:");  scanf("%s",msg.content);  msg.type = MSGSND; //设置发送码  msgsnd(msg\_id, &msg, strlen(msg.content)+1,IPC\_NOWAIT); //发送消息  if(strcmp(msg.content,"q") == 0)  {  msgctl(msg\_id,IPC\_RMID,NULL); //关闭消息队列  break;  }  msgrcv(msg\_id, &msg, 256, MSGRCV, 0); //等待接受  printf("\t\t\tClient:%s\n",msg.content);  }  return 0;  } |

msg\_client.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <string.h>  #include <stdio.h>  #include <stdlib.h>  #define ID 0x111  #define MSGSND 100  #define MSGRCV 200  typedef struct Msg{  long type;  char content[256];  }Msg;  int main(int argc, char \*argv[])  {  Msg msg;  key\_t msg\_key = ftok(argv[1],ID);  if(msg\_key == -1)  exit(1);  key\_t msg\_id = msgget(msg\_key,0); //获取服务端消息队列  while(1)  {  msgrcv(msg\_id, &msg, 256, MSGRCV, 0); //等待接收消息  printf("\t\t\tServer:%s\n",msg.content);  printf("Client:");  scanf("%s",msg.content);  msg.type = MSGSND; //设置发送码  if(strcmp(msg.content,"q") == 0)  break;  msgsnd(msg\_id, &msg, strlen(msg.content)+1, IPC\_NOWAIT); //发送消息  }  return 0;  } |

结果：



2、网络主机进程间的通信：socket网络编程技术

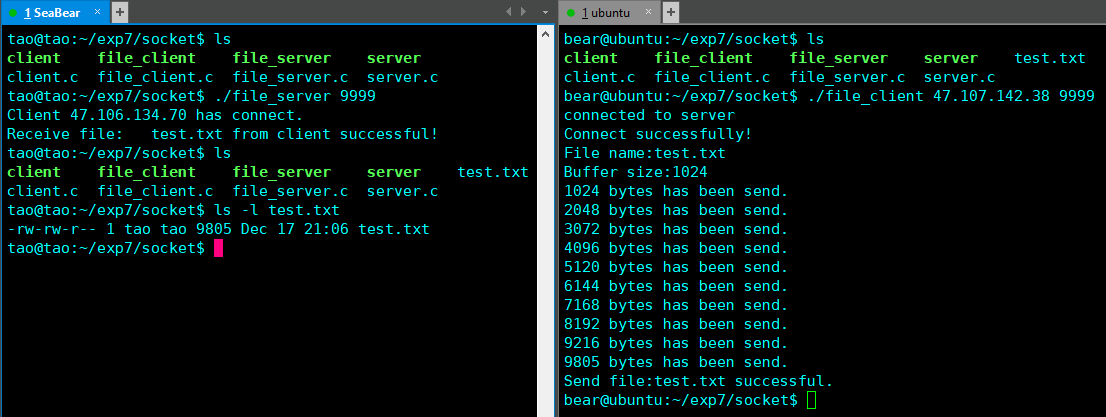
socket实现简单Linux主机间的文件传输

file\_server.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83 | #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <unistd.h>  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #define MSG\_SIZE 512  void main(int argc, char \*argv[])  {  int server\_sockfd; //服务器端socket  int client\_sockfd; //客户端socket  int len, sin\_size, size;  char msg[MSG\_SIZE+1],file\_name[MSG\_SIZE+1];  char \*buf; //数据传送的缓冲区  struct sockaddr\_in my\_addr; //服务器网络地址结构体  struct sockaddr\_in remote\_addr; //客户端网络地址结构体  FILE \*fp = NULL;  memset(&my\_addr,0,sizeof(my\_addr)); //数据初始化--清零  my\_addr.sin\_family = AF\_INET; //设置为IP通信  my\_addr.sin\_addr.s\_addr = INADDR\_ANY; //服务器IP地址--允许连接到所有本地地址上  my\_addr.sin\_port = htons(atoi(argv[1])); //服务器端口号  /\*创建服务器端socket--IPv4协议，面向连接通信，TCP协议\*/  if((server\_sockfd = socket(AF\_INET,SOCK\_STREAM,0)) < 0)  {  perror("socket error");  exit(1);  }  /\*将socket绑定到服务器的网络地址上\*/  if(bind(server\_sockfd, (struct sockaddr \*)&my\_addr, sizeof(struct sockaddr)) < 0)  {  perror("bind error");  exit(1);  }  /\*监听连接请求--监听队列长度为5\*/  if(listen(server\_sockfd, 5) < 0)  {  perror("listen error");  exit(1);  }  sin\_size = sizeof(struct sockaddr\_in);  /\*等待客户端连接请求到达\*/  if((client\_sockfd = accept(server\_sockfd,(struct sockaddr \*)&remote\_addr,&sin\_size)) < 0)  {  perror("accept error");  exit(1);  }  printf("Client %s has connect.\n",inet\_ntoa(remote\_addr.sin\_addr));  send(client\_sockfd,"Connect successfully!\n",22,0);  /\*创建传输文件\*/  len = recv(client\_sockfd,msg,MSG\_SIZE,0);  msg[len] = '\0';  strcpy(file\_name,msg);  if((fp = fopen(file\_name, "w")) == NULL)  {  perror("creat file error");  exit(1);  }  /\*获取缓冲区大小\*/  len = recv(client\_sockfd,msg,MSG\_SIZE,0);  msg[len] = '\0';  size = atoi(msg);  buf = (char \*)malloc(sizeof(char)\*size);  send(client\_sockfd,"ok",2,0); //发送给服务端开始传送数据的信号，防止msg缓冲区读入文件流  /\*数据流写入文件\*/  while((len = recv(client\_sockfd, buf, size, 0)) > 0)  {  if(fwrite(buf, sizeof(char), len, fp) < len)  {  perror("file write failed");  break;  }  bzero(buf, size); //清空数据缓冲区  }  printf("Receive file:\t%s from client successful!\n", file\_name);  fclose(fp); //关闭文件  close(client\_sockfd); //关闭socket  close(server\_sockfd);  exit(0);  } |

file\_client.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73 | #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <unistd.h>  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #define MSG\_SIZE 512  void main(int argc, char \*argv[])  {  int client\_sockfd;  int len, size, send\_size = 0;  struct sockaddr\_in remote\_addr; //服务器端网络地址结构体  char msg[MSG\_SIZE+1],file\_name[MSG\_SIZE+1];  char \*buf; //数据传送的缓冲区  FILE \*fp = NULL;  memset(&remote\_addr,0,sizeof(remote\_addr)); //数据初始化--清零  remote\_addr.sin\_family = AF\_INET; //设置为IP通信  remote\_addr.sin\_addr.s\_addr = inet\_addr(argv[1]); //服务器IP地址  remote\_addr.sin\_port = htons(atoi(argv[2])); //服务器端口号  /\*创建客户端socket--IPv4协议，面向连接通信，TCP协议\*/  if((client\_sockfd=socket(AF\_INET,SOCK\_STREAM,0))<0)  {  perror("socket error");  exit(1);  }  /\*将socket绑定到服务器的网络地址上\*/  if(connect(client\_sockfd,(struct sockaddr \*)&remote\_addr,sizeof(struct sockaddr)) < 0)  {  perror("connect error");  exit(1);  }  printf("connected to server\n");  len = recv(client\_sockfd,msg,MSG\_SIZE,0); //接收服务器端信息  msg[len] = '\0';  printf("%s",msg); //打印服务器端信息  /\*输入传输文件名称\*/  printf("File name:");  scanf("%s",file\_name);  if((fp = fopen(file\_name, "r")) == NULL)  {  perror("open file error");  exit(1);  }  send(client\_sockfd,file\_name,strlen(file\_name),0);  /\*输入缓冲区大小\*/  printf("Buffer size:");  scanf("%s",msg);  size = atoi(msg);  buf = (char \*)malloc(sizeof(char)\*size);  send(client\_sockfd,msg,strlen(msg),0);  if(recv(client\_sockfd,msg,MSG\_SIZE,0) > 0) //接收服务器确认消息  printf("Begin to send...\n");  /\*传输文件\*/  while((len = fread(buf, sizeof(char), size, fp)) > 0) // 每读取一段数据，便将其发送给客户端，循环直到文件读完为止  {  if(send(client\_sockfd, buf, len, 0) < 0)  {  perror("file send failed");  break;  }  send\_size += len;  printf("%d bytes has been send.\n",send\_size);  bzero(buf, size);  }  printf("Send file:%s successful.\n",file\_name);  fclose(fp); //关闭文件  close(client\_sockfd); //关闭socket  exit(0);  } |

结果：

## 问题记录和实验总结

问题记录：

1.使用socket传送文件时文件的前面部分内容缺失。

解决：原因是因为使用recv接受文件名消息时缓冲区没有满，客户端文件开始时，会继续接受导致读取了文件的部分内容，所有在客户端开始传输文件之前需要确保没有其余的recv接收内容。

2.主机间使用socket通讯客户端无法连接服务。

解决：和之前配置rsync服务遇到的问题一样，端口没有打开，确保服务开启的端口对外可访问才能连接成功。

这次实验主要学习了linux中进程的通信和一些特殊的进程，进程的通信主要有管道、IPC通信。管道通信包含有名管道和匿名管道，匿名管道是单工通信，进程创建管道后返回的两个文件描述符，不同进程用这两个文件描述符进行数据读写通信，但往往是父进程子进程之间的通信比较有局限性，有名管道通过创建管道文件fifo可以实现不同进程之间的通信。在拓展部份学习了IPC通信的信号、共享内存和消息队列。信号通信的信息量比较少，可以通过kill()发送信号和signal()接收达到同步效果，通过使用消息队列和共享内存了解了他们的通信机理。要注意共享内存是一块进程都可以访问的空间并且没有任何的同步与互斥机制，所以往往要使用信号量来实现对共享内存的存取的同步。消息队列则是消息的链表，存放在内核中并由消息队列标识符表示。IPC的通信可以在linux下通过ipcs查看。此外还学习了主机之间的通信socket是基于TCP/IP协议的，实验中通过简单的客户端服务端文件传输，明白了scoket相关系统调用的使用，理解了简单的使用方法。