****

Computer Networks

Lab Report

Name ：谭演锋

Student ID ：202130100456

Class ：计算机全英联合班

Collaborator(s) : 梁卓文 郭荣锦 杨竣尧

# Task 4 Socket Programming

1. **Experimental Purposes**

**Objective**：

1. Master the basic principles and methods of Socket network programming
2. Understand the underlying principle of operation of Socket deeply
3. **Requirement：**
4. Basic: Implement a client and a server similar with ftp protocol; complete the basic function of file transfer.

List the files/directories

Change working-directory

Download file/s

Upload files/s

Create/remove files/directories

Etc…

1. Submit content: document report （include **design document**, **using explanation**）, source code，the content above store in 2 folder apart（Doc、Src）

Explain the design of your file transfer program(FTP).

High-layer encapsulation Socket can not be used (like java class library, MFC and so on)

1. Explain the usage of socket.h or winsock.h

**3. Experimental Environment**

Platform: Windows

Developing Language: C

Developing Environment: Visual C++

Group: 4 members/group

1. **Design the FTP**
2. The flow chart of FTP(Part of)

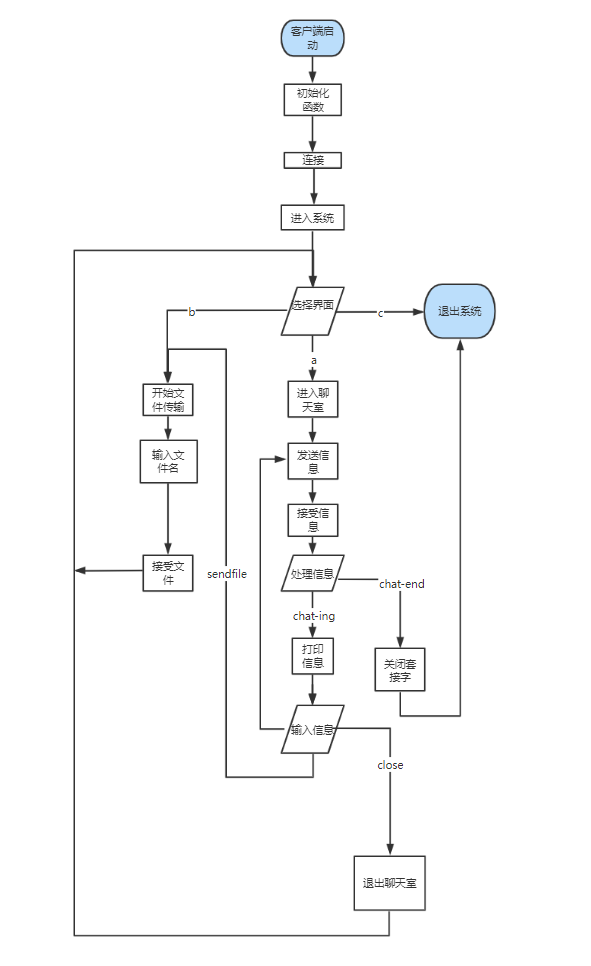
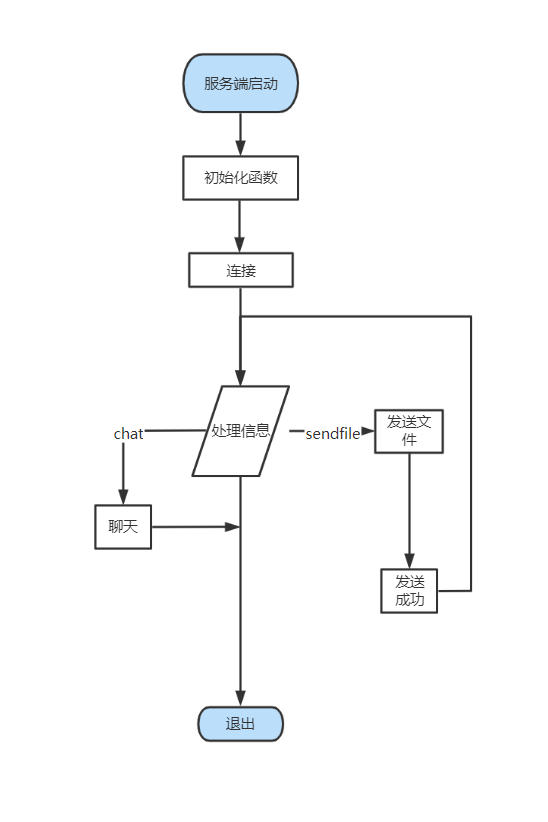
****

Figure. Server and Client

1. **Main Implementation**

Overview of implementation

Processing information is the difficulty and breakthrough point in this development process. In order to combine chat and send files into one program, flag bits are specially set to mark the attributes of sent information. The sent information is encapsulated in a structure, and the two parties process it according to the header information of the sent information when communicating. The server only needs to call a processing information function to run, and the client independently designs the processing function of sending files and the processing function of chatting according to the needs selected by the user. The advantage of this is that the implementation of the program is more organized, the development is more simple and specific, and the later is easy to maintain, optimize and improve.

**5.1 Implementation of system processing information**

Due to the limitation of the size and capacity of the send cache, it is difficult to transfer large files, but in the actual application environment, simple file transfer can not meet the needs of its users, if the transfer of files is one-time, the system often causes file transfer failure or information loss due to insufficient allocation of memory. In order to solve this problem, the design of this course takes the idea of: the server splits the file to be sent, transmits the data a few times, the client splices the fragments of the data, writes a few times until the information is received.

Here is the general idea behind how the system processes information:

Process of sending file information:

1. The client input the file name and send it to the client, marking position 1;
2. The server receives the file name, finds whether there is a file, sends the file size to the client, marks the location 2, and opens up the memory space of the send cache; If not, a message without file is sent, marking position 6;
3. The client receives the file size according to the file size to open up memory space and send a request to accept, marking position 3; Receive no file message continue to enter the file name and send, marking position 1;
4. The server begins to send the file after receiving the request, marking the position 4; After the transmission is completed, send the successful transmission signal, marking the position 5; Close the connection.

**5.3 Implementation of sending file by packet**

**5.3.1 The data structure used for sending message**

In order to more convenient understanding of its implementation principle, the code is attached to supplement the explanation

|  |
| --- |
| enum MSGTAG  **{**  MSG\_FILENAME **=** 1**,//file name**  MSG\_FILESIZE **=** 2**,//file size**  MSG\_READY\_READ **=** 3**,//tell client be ready to read**  MSG\_SENDFILE **=** 4**,//sending file**  MSG\_SUCCESSED **=** 5**,//succeed in sending file**  MSG\_OPENFILE\_FAILD **=** 6**,//failure in opening file**  ENDSEND**=**7**,//stop sending file**  CHAT\_END **=** 10，**//end of program**  CHAT\_ING**=**11**//start chatting**  MSG\_DIRECTORY = 12,**//make a directory**  MSG\_DELETEFILE = 15,**//delete file**  MSG\_UPLOAD = 16,**//upload file**  MSG\_PATHNAME = 17, **//change path**  MSG\_UPFILENAME = 18, **//upload-first step**  MSG\_UPFILE = 19, **//upload-second step**  MSG\_UP\_READY\_READ = 20 **//upload-third step**  **};**  #pragma pack(1)  #define PACKET\_SIZE (10240 -sizeof(int)\*3)//define packet length  struct MsgHeader  **{**  enum MSGTAG msgID**;**//Current message mark  union MyUnion  **{**  struct  **{**  char fileName**[**256**];**  int fileSize**;**  char filePath**[**1024**];**  int pathSize**;**  **}**fileInfo**;**  struct  **{**  char dirPath**[**1024**];**  **}**dirInfo;  struct  **{**  int nStart**;**  int nsize;  char buf**[**PACKET\_SIZE**];**  **}**packet**;**  struct  **{**  char chatbuf**[**BUFSIZ**];**  **}**CHAT**;**  **};**  **};**  #pragma pack() |
|  |

In order to facilitate the understanding of the function data structure it constructs, the following describes the details of its server unpacking:

* Packet size (PCKETSIZE) : 10204 bytes, where the flag bit accounts for one byte, the file start position accounts for one byte, the packet data size accounts for one byte, so the sent file information accounts for (10240-sizeof(int)) bytes; In particular, when sending the file content (flag bit is 4), it is not necessary to send the file name and file size, so it does not occupy the space of the package.
* Use nstart to record the location of the file information to be sent, that is, the starting location of the file information when the next packet is sent.
* Record the packet data size with nsize.
* If nstart+nsize+1>f-size, it means that this packet is the last packet, and the sender judges the completion of sending according to this condition.

**5.3.2 Server send file**

The server first send the msgID to tell the Client that it should receive the file. Then packeting the data according to the filesize by function “memcpy” for copying data. Finally, use send function to sedd data. We use a for loop in order to avoid the file is too large

|  |
| --- |
| bool sendFile**(**SOCKET clifd**,** struct MsgHeader**\*** pmsg**)**  **{**  //告诉客户端准备接受文件了  //如果文件的长度大于每个数据包能传送的大小(packet\_size)那么就分快  struct MsgHeader msg**;**  msg**.**msgID **=** MSG\_READY\_READ**;**    **for** **(**size\_t i **=** 0**;** i **<** g\_fileSize**;** i **+=** PACKET\_SIZE**)**  **{**  msg**.**packet**.**nStart **=** i**;**  //判断是否为最后一个包，获取最后一个包的长度  **if** **(**i **+** PACKET\_SIZE**+**1 **>**g\_fileSize**)**  **{**  msg**.**packet**.**nsize **=** g\_fileSize **-** i**;**  **}**  **else**  **{**  msg**.**packet**.**nsize **=** PACKET\_SIZE**;**  **}**  memcpy**(**msg**.**packet**.**buf**,** g\_fileBuf **+** msg**.**packet**.**nStart**,** msg**.**packet**.**nsize**);**  **if** **(**SOCKET\_ERROR **==** send**(**clifd**,** **(**char**\*)&**msg**,** **sizeof(**struct MsgHeader**),** 0**))**  **{**  printf**(**"文件发送失败%d\n"**,**WSAGetLastError**());**  **return** **false;**  **}**  **}**    **return** TRUE**;**  **}** |

**5.3.3 Client receive file**

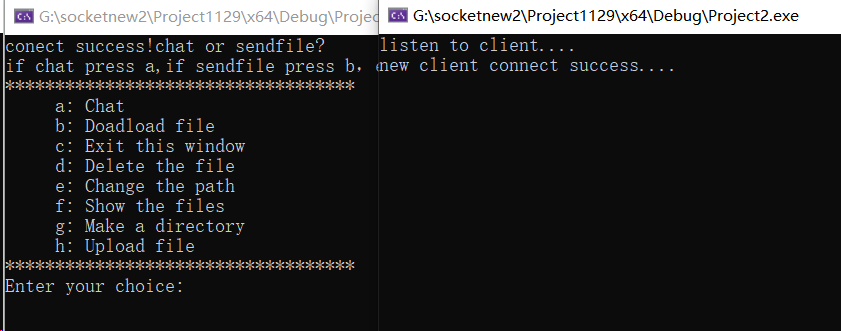
Once the client receive message with msgID == MSG\_READY\_READ, it calls the writeFil e function. Then use the “memcpy” function again to copy the data. Finally, if nStart + nsize >= g\_fileSize which means client receiving all the data, client starts to write the file. Then after finishing it, send a successful message to the server

|  |
| --- |
| bool writeFile**(**SOCKET serfd**,** struct MsgHeader**\*** pmsg**)**  **{**  //如果g\_fileBUf为空，则分配内存失败  **if** **(**g\_fileBuf **==** **NULL)**  **{**  **return** FALSE**;**  **}**  //将指针移到开始读的位置  int nStart **=** pmsg**->**packet**.**nStart**;**  //获取包长度  int nsize **=** pmsg**->**packet**.**nsize**;**  //拼接  memcpy**(**g\_fileBuf**+**nStart**,**pmsg**->**packet**.**buf**,** nsize**);**  //判断文件是否完整，服务器是否发完了数据，接受到最后一个包时开始写入  **if(**nStart **+** nsize **>=** g\_fileSize**)**  **{**  FILE**\*** pwrite **=** fopen**(**g\_fileName**,** "wb"**);**  **if** **(**pwrite **==** **NULL)**  **{**  printf**(**"write file error..\n"**);**  **return** **false;**  **}**  fwrite**(**g\_fileBuf**,** **sizeof(**char**),** g\_fileSize**,** pwrite**);**  //写完释放缓存，关闭读指针  fclose**(**pwrite**);**  free**(**g\_fileBuf**);**  g\_fileBuf **=** **NULL;**  //发送写入成功消息  struct MsgHeader msg**;**  msg**.**msgID **=** MSG\_SUCCESSED**;**  send**(**serfd**,** **(**char**\*)&**msg**,** **sizeof(**struct MsgHeader**),** 0**);**  **}**  **return** **true;**  **}** |

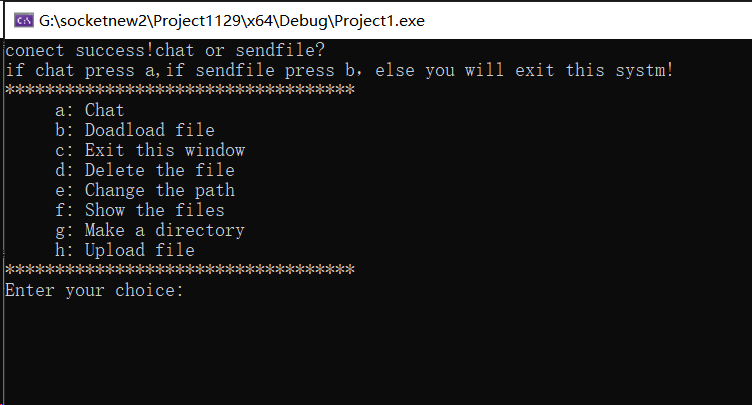
1. **Using Explanation**
   1. **Introduction**

The project uses the existing computer resources to develop a software that can effectively carry out online chat, file transmission and directory processing, which is divided into client and server. The main work is as follows:

1. The server is turned on to listen to the client, receive the connection request and establish a connection.
2. When the client is opened, it will automatically connect to the preset server. When both sides are opened, the two can start communication.

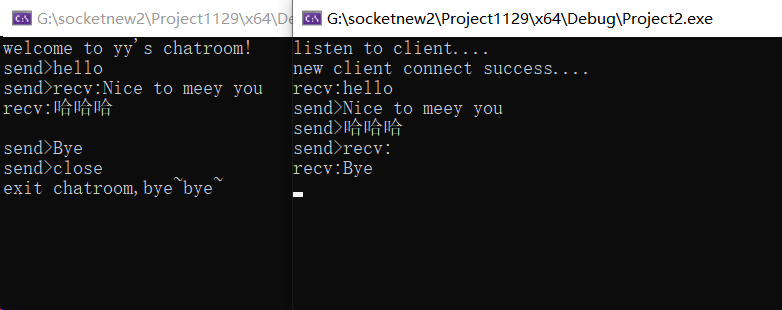


1. Print function option on the client interface. The client can select the function to be performed through commands, such as "b" to download the files and "h" to upload files.



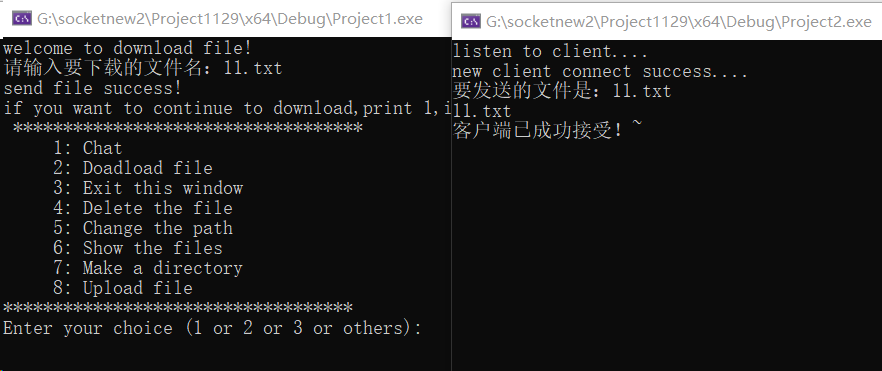
1. The server determines which function to perform based on the information sent by the client.
2. Chatting

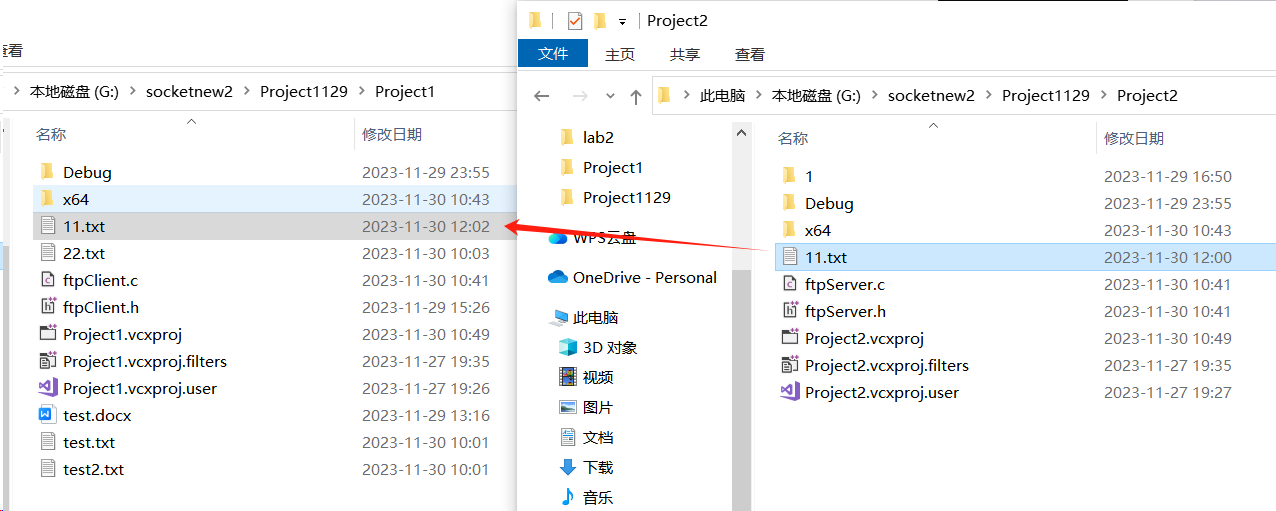
* During the chat, the server can enter the "close" command to actively close the connection to the client.
* During the chat, the client can enter the "close" command to close the connection with the client; Enter the "sendfile" command to terminate the conversation and request to send the file.
* After the server successfully sends the file, the client can choose to continue chatting.



1. Download files

Enter the file you want to download. If there is no other path in front of the file, the file path defaults to the root directory of the server.

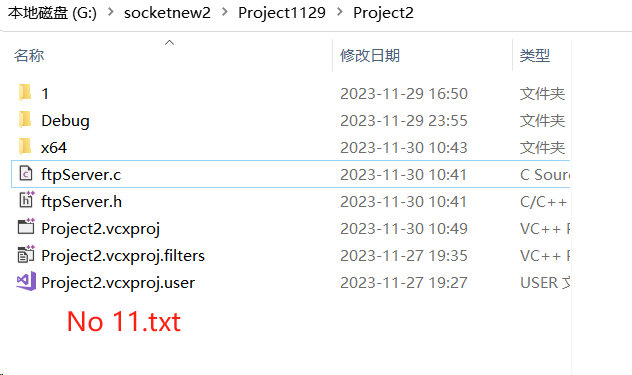
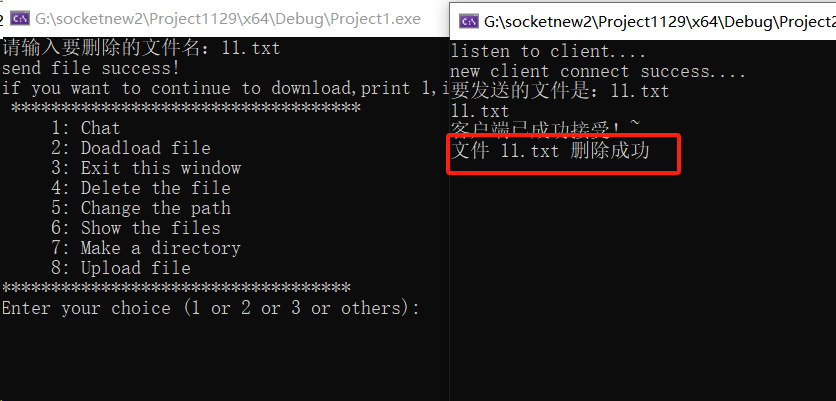




You can see that the server responds to the received request successfully, and the client downloads the file in its path.

1. Delete files

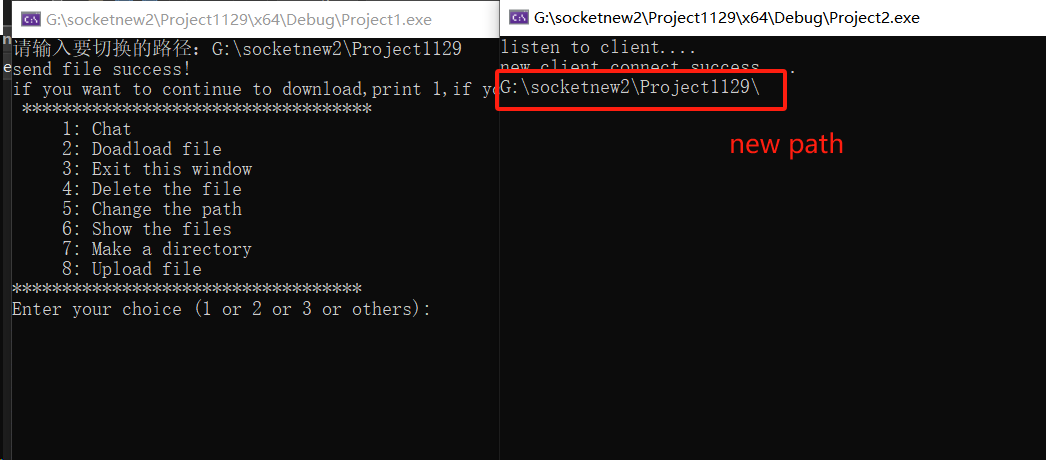
Enter the name of the file to be deleted. If there is no path before the file, the file is in the root directory of the server by default.



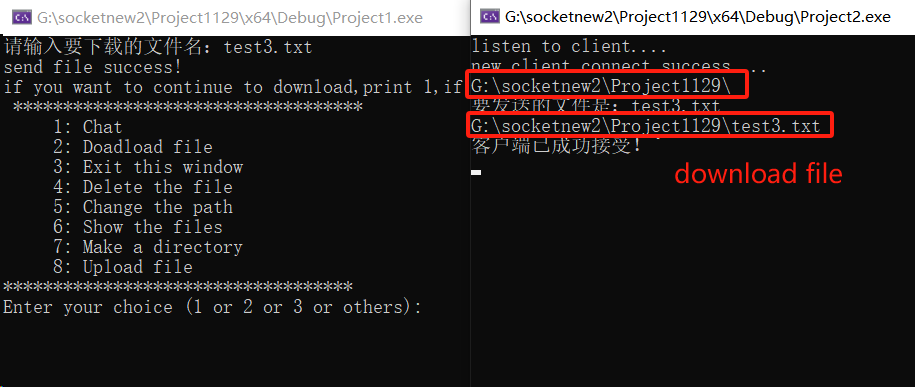
The server replies that the file is deleted successfully. The 11.txt file in the root directory is deleted.

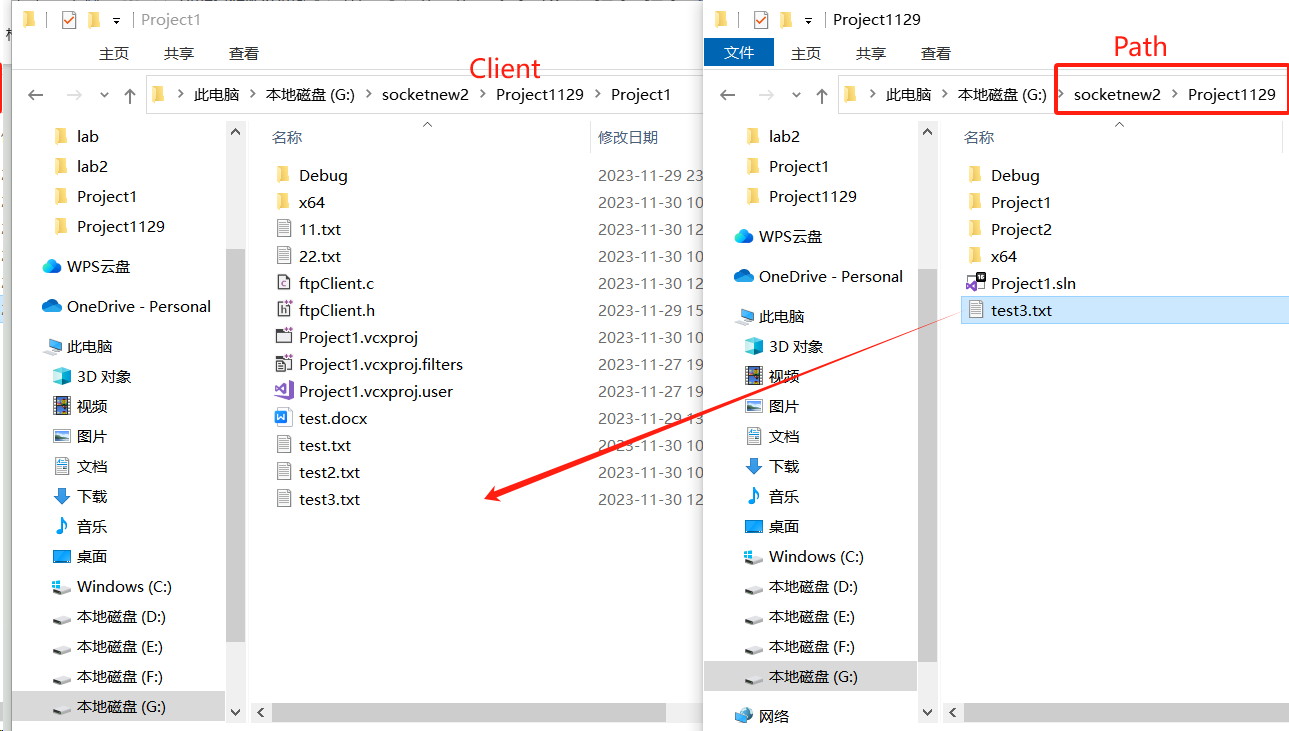
1. Change the path

Enter the path you want to change, and the root of the server will be changed to the path you entered



You can see that the server prints the updated path, indicating that the path has been successfully modified.Then use the function of downloading files to test whether the server path has been changed.

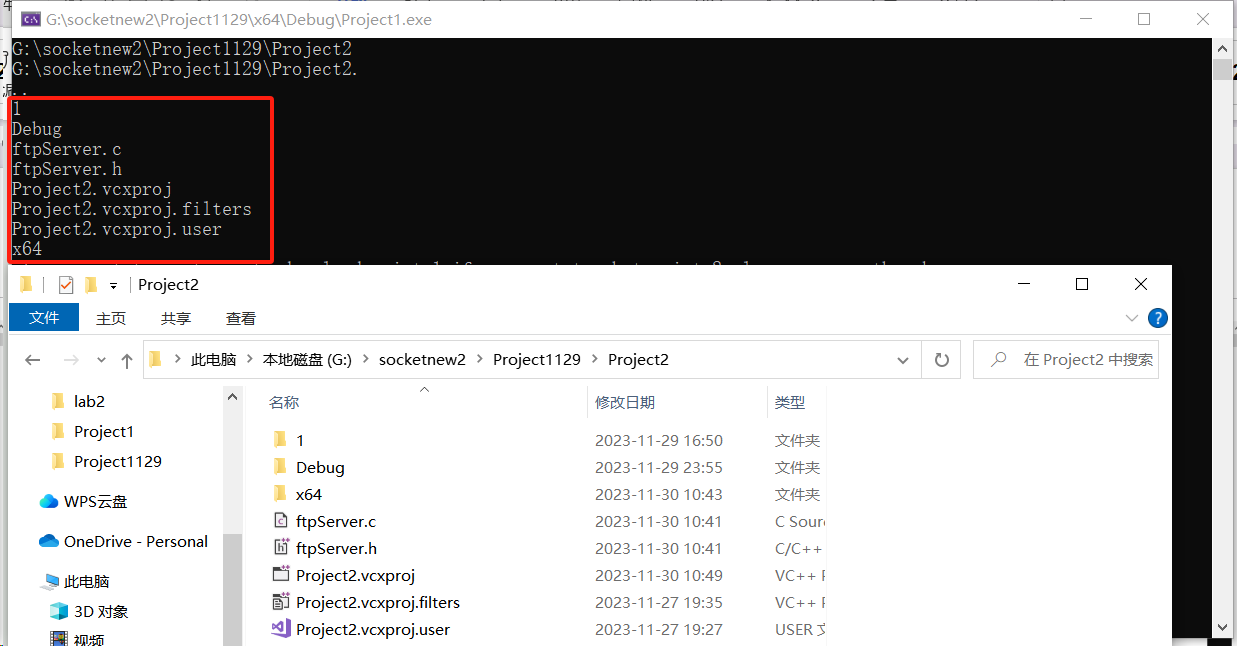




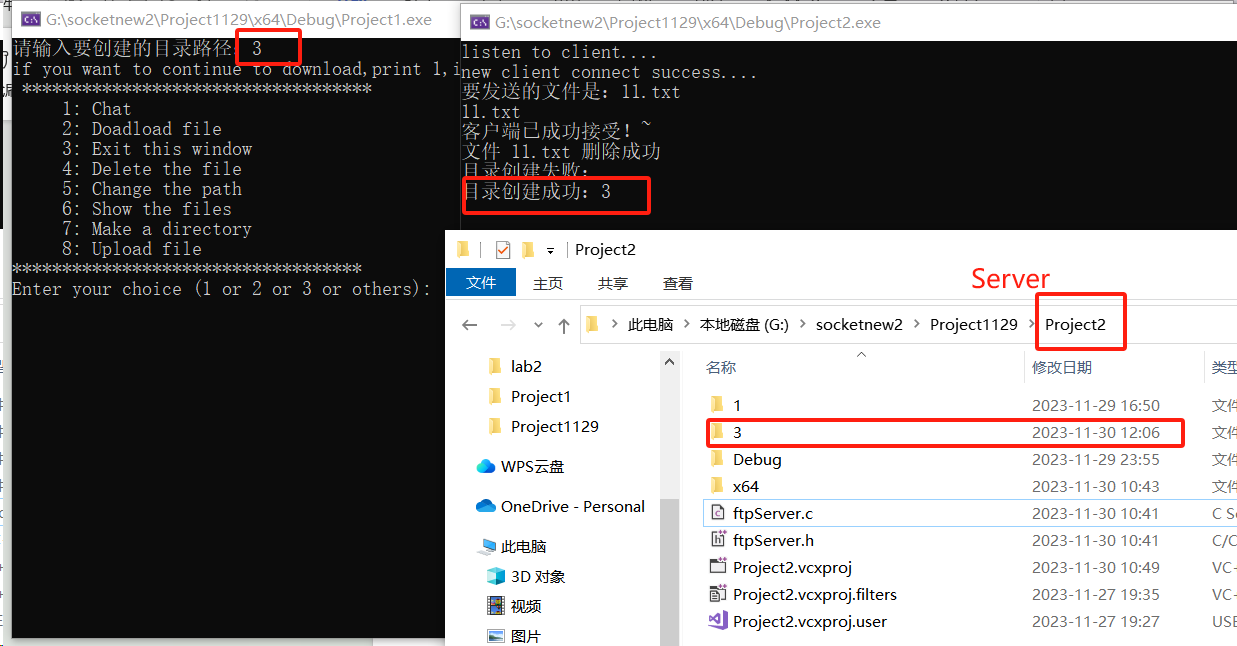
As you can see in the figure above, we changed the path to a directory above the original path of the server, and then entered the file we want to download, test3.txt (without adding an absolute address). Finally, the client successfully downloads the file in its own path, indicating that the path is successfully changed.

1. Show the files

Enter the directory or folder path. The client displays all file names or folders in the path.

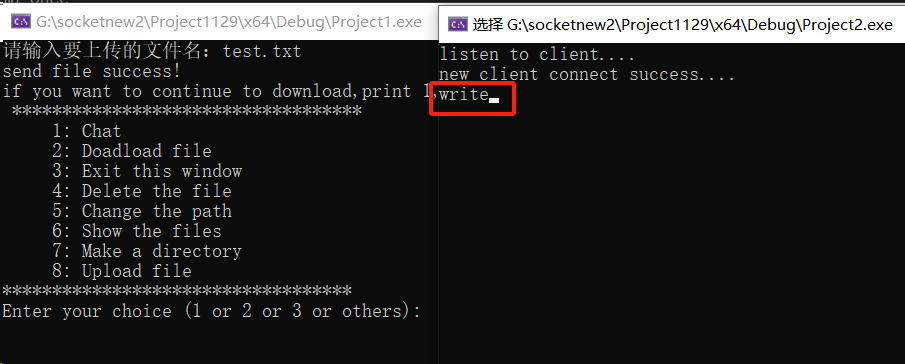


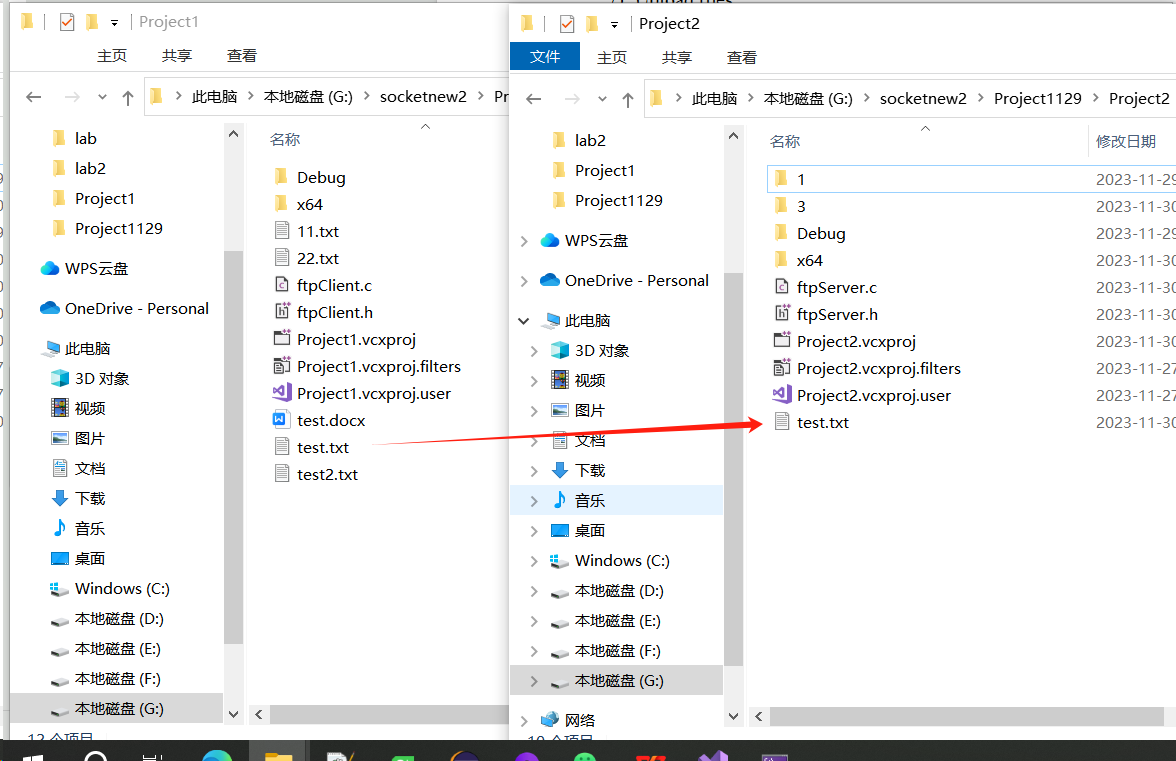
1. Make a directory



1. Upload files

Enter the file to be uploaded. If the address is not absolute, the file in the directory where the client resides will be uploaded by default. The file will be displayed in the server directory.





As can be seen from the above figure, the client uploads the file named test.txt, and the server returns "write", which indicates that the file has been written in its own path. You will eventually see the test.txt file in the server path.

1. Exit

Enter “c” or “3” to exit the file transfer program(FTP)



* 1. **Characteristic**

1. The client can apply to the server for sending large files larger than 100m.
2. The communication between the two sides supports the concurrent function, that is, one party can send multiple data or accept multiple data at the same time without blocking.
3. **Explaination of winsock.h**

**7.1 Socket function**

The `socket` function is used to create a new socket, which involves requesting a socket resource from the system. This function is utilized for both client and server applications in network programming.

|  |
| --- |
| int socket(int domain, int type, int protocol); |

domain: Protocol domain, also known as protocol family. Commonly used protocol families include AF\_INET (IPv4), AF\_INET6 (IPv6), AF\_LOCAL (or AF\_UNIX, Unix domain Socket), AF\_ROUTE, etc. The protocol family determines the address type of the socket, and the corresponding address must be used in communication. For example, AF\_INET determines the use of an IPv4 address (32 bits) combined with a port number (16 bits), while AF\_UNIX determines the use of an absolute file path as the address.

type: Specifies the socket type. Common socket types include SOCK\_STREAM, SOCK\_DGRAM, SOCK\_RAW, SOCK\_PACKET, SOCK\_SEQPACKET, etc. Stream socket (SOCK\_STREAM) is a connection-oriented socket, used for connection-oriented TCP service applications. Datagram socket (SOCK\_DGRAM) is a connectionless socket, corresponding to connectionless UDP service applications.

protocol: Specifies the protocol. Common protocols include IPPROTO\_TCP, IPPROTO\_UDP, IPPROTO\_STCP, IPPROTO\_TIPC, etc., corresponding to TCP transport protocol, UDP transport protocol, STCP transport protocol, TIPC transport protocol. The first parameter can only be AF\_INET, the second parameter can only be SOCK\_STREAM, and the third parameter can only be 0. Unless system resources are exhausted, the socket function generally does not return failure.

Return value: Returns a socket if successful, -1 on failure, and the error reason is stored in errno.

**7.2 gethostbyname function**

Convert an IP address or domain name into the address expressed by the hostent structure.

|  |
| --- |
| struct hostent \*gethostbyname(const char \*name); |

Parameters: name: Domain name or hostname, for example, "192.168.1.3" or "www.freecplus.net."

Return Value:

- If successful, returns a pointer to a `hostent` structure.

- If unsuccessful, returns NULL.

‘gethostbyname’ is used only for clients. It converts a string representation of an IP address to a structured IP address. It typically does not return an error as long as the address format is correct. In case of failure, it does not set the value of errno.

**7.3 connect function**

Send connect request to the server

|  |
| --- |
| int connect(int sockfd, struct sockaddr \* serv\_addr, int addrlen); |

Function Description:

The `connect` function is used to connect the socket specified by the parameter `sockfd` to the server specified by the parameter ‘serv\_addr’. The parameter `addrlen` is the length of the sockaddr structure.

Return Value:

- If successful, returns 0.

- If unsuccessful, returns -1, and the error reason is stored in `errno`.

‘connect’ function is used only for clients. If the server's address or port is incorrect, or the server is not running, the `connect` function will fail.

**7.4 bind function**

The server binds the address and port used for communication to the socket.

|  |
| --- |
| int bind(int sockfd, const struct sockaddr \*addr,socklen\_t addrlen); |

- Parameters:

- sockfd: The socket to be bound.

- addr: The structure holding the server's address and port for communication.

- addrlen: The size of the `addr` structure.

- Return Value:

- If successful, returns 0.

- If unsuccessful, returns -1, and the error reason is stored in `errno`.

If there is an error in binding, such as an incorrect address or a port already in use, the bind function will report an error. Otherwise, it generally does not return an error.

**7.5 listen function**

The listen function transforms an actively connected socket into a passively connected socket, allowing this socket to accept connection requests from other sockets and become a server socket.

|  |
| --- |
| int listen(int sockfd, int backlog); |

Parameter:

sockfd: A socket that has already been bound. The socket returned by the `socket` function is an actively connected socket. In server programming, the programmer wants this socket to accept incoming connection requests, i.e., passively waiting for clients to connect. Since the system defaults to considering a socket as actively connected, the programmer needs to inform the system. This is accomplished by calling the `listen` function.

Here is the English translation of the provided text:

 backlog: This parameter involves some network details, can be a bit tricky, filling in 5 or 10 is acceptable, generally not exceeding 30. After calling 'listen', the server-side socket can use 'accept' to accept client connection requests.

Return value: Returns 0 if successful, -1 if unsuccessful, and the error reason is stored in 'errno'. The 'listen' function generally does not return errors.

7.6 accept function

The server accepts the connection from the client.

|  |
| --- |
| int accept(int sockfd,struct sockaddr \*addr,socklen\_t \*addrlen); |

sockfd: is a socket that has already been listened to.

addr: is used to store the client's address information, expressed using the 'sockaddr' structure. If client address information is not needed, it can be set to 0.

addrlen: is used to store the length of the 'addr' parameter. If 'addr' is set to 0, 'addrlen' should also be set to 0.

accept function waits for a connection from the client. If no client connects, it continues to wait. This waiting behavior is referred to as blocking.

After accept successfully waits for a client connection, it creates a new socket. The return value of the function is this new socket. The server uses this new socket for exchanging messages with the client.

Return value: Returns 0 if successful, -1 if unsuccessful, and the error reason is stored in 'errno'.

If accept is interrupted or encounters other reasons during the waiting process, it returns -1, indicating failure. In case of failure, 'accept' can be called again.

**7.7 send function**

The 'send' function is used to send data through a socket to the other end. Whether it is a client or a server, the application uses the 'send' function to send data to the other end of a TCP connection.

|  |
| --- |
| ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags); |

sockfd'is a socket that has an established connection.

buf is the memory address of the data to be sent. It can be the address of basic C data types, an array, structure, or string. Whatever is in the memory will be sent.

len is the length of the data to be sent, representing the valid data length in 'buf'.

flags is set to 0; other numerical values are not significant.

The function returns the number of characters already sent. If an error occurs, it returns -1, and the error information is indicated by 'errno'.

Note that even if the network is disconnected or the socket is closed by the other end, the send function will not immediately report an error. It may take a few seconds for an error to be reported.

If the send function returns an error (<= 0), it indicates that the communication link is no longer available.

**7.8 recv function**

The recv function is used to receive data sent by the remote socket.

The recv function is employed to receive data sent by the remote end through the socket. Whether it is a client or a server, the application uses the 'recv' function to receive data from the other end of a TCP connection.

|  |
| --- |
| ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags); |

Here is the English translation of the provided text:

sockfd is a socket that has an established connection.

buf is the memory address used for receiving data. It can be the address of basic C data types, an array, structure, or string, as long as it is a block of memory.

len is the length of the data to be received, and it should not exceed the size of 'buf' to avoid memory overflow.

flags is set to 0; other numerical values are not significant.

The function returns the number of characters already received. In case of an error, it returns -1, and 'errno' is not set on failure.

If the remote end of the socket has not sent any data, the 'recv' function will wait. If data is sent by the remote end, the function returns the number of characters received. In case of an error, it returns -1. If the socket is closed by the remote end, the return value is 0.

If the recv function returns an error (<= 0), it indicates that the communication channel is no longer available.

**7.9 The server-side program binds an address**

1) Particular IP Address

|  |
| --- |
| m\_servaddr.sin\_addr.s\_addr = inet\_addr("192.168.149.129"); // 指定ip地址 |

2)Any IP address

|  |
| --- |
| m\_servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY); // 本主机的任意ip地址 |

**7.10 The server-side program binds an address.**

|  |
| --- |
| m\_servaddr.sin\_port = htons(5000); // 通信端口 |

**7.11 The client-side program specifies the IP address of the server**

|  |
| --- |
| struct hostent\* h;  if ( (h = gethostbyname("118.89.50.198")) == 0 ) // 指定服务端的ip地址。  { printf("gethostbyname failed.\n"); close(sockfd); return -1; } |

**7.12 The client-side program specifies the communication port of the server**

|  |
| --- |
| servaddr.sin\_port = htons(5000); |

1. **Individual report**
2. **My responsible part**

I realized the two functions of **change a path** and **upload files**, and completed the corresponding window instructions. In addition, I also participated in the design of FTP system and **MSGTAG**, helped the team quickly understand the principle of socket implementation, and set up the code framework of socket.

1. **Specific operation**
2. Change a path

I define MSG\_PathName =17 in client.h and server.h as the message type for change the default path of server.

The client requests to change the path：

|  |
| --- |
| void changeFilePath(SOCKET serfd) {  char pathName[1024];  printf("请输入要切换的路径：");  gets\_s(pathName, 1024);  //strncpy(uploadpath, pathname, sizeof(uploadpath) - 1);  //uploadpath[sizeof(uploadpath) - 1] = '\0'; // ensure null-terminatio  //"C:\srp\main.py"  struct MsgHeader path;  path.msgID = MSG\_PATHNAME;  strcpy(path.fileInfo.filePath, pathName);  path.fileInfo.pathSize = strlen(pathName);  if (SOCKET\_ERROR == send(serfd, (char\*)&path, sizeof(struct MsgHeader), 0))  {  err("send");  return;  }  } |

The server processes the request and creates the directory under the root directory：

First, it declares a character array named pathName with a size of 1024. This array will be used to store the file path entered by the user.

Then it uses the gets\_s function to read a line of input from the user and store it in the pathName array. The gets\_s function is used to read input from the standard input and ensures that the input does not exceed the size of the array to prevent buffer overflow.

Then it copies the contents of the pathName array to the filePath member of the fileInfo struct within the path struct using the strcpy function. This ensures that the file path entered by the user is stored in the path struct.

Finally, it sends the path struct to the server using the send function. The send function is used to send data over the specified socket. If an error occurs during the sending process (indicated by SOCKET\_ERROR), it calls the err function (which is not shown in the provided code snippet) to handle the error and returns from the function.

The server response to change the path：

|  |
| --- |
| bool changePath(SOCKET clifd, struct MsgHeader\* pmsg)  {  strcpy(uploadPath, pmsg->fileInfo.filePath);  //"C:\Log\FRE\_Log\_20231114.log"  char ch = '\\';  strncat(uploadPath, &ch, 1);  printf("%s\n", uploadPath);  //for(int i =0;i< uploadPathSize;)  struct MsgHeader msg;  msg.msgID = MSG\_SUCCESSED;  send(clifd, (char\*)&msg, sizeof(struct MsgHeader), 0);  return true;  } |

The server uses the strcpy function to copy the contents of pmsg->fileInfo.filePath (file path stored in the filePath member of the fileInfo struct within pmsg) to the uploadPath variable. This updates the uploadPath variable with the new file path.

Then we use the strncat function to concatenate the ch character to the uploadPath string. This adds a backslash character at the end of the file path.

Finally it assigns the value MSG\_SUCCESSED to the msgID member of the msg struct. This value indicates that the message type is related to the success status of the operation. And we send the massage to the client, it has been success.

1. Upload files

I define MSG\_UPFILENAME=18, MSG\_UPFILE = 19, MSG\_UP\_READY\_READ = 20 in client.h and server.h as the message type for deleting files in socket communication, as mentioned in above design document. We have 2 steps for client upload file.

**First step**

The client requests to upload a file：

|  |
| --- |
| void uploadFileName(SOCKET serfd){  char fileName[1024];  printf("请输入要上传的文件名：");  gets\_s(fileName, 1024);  struct MsgHeader file;  file.msgID = MSG\_UPFILENAME;  strcpy(file.fileInfo.fileName, fileName);  FILE\* pread = fopen(fileName, "rb");  //找不到就告诉客户端发送失败  if (pread == NULL){  printf("找不到[%s]文件..\n", fileName);  }  //找到文件后，获取文件大小  fseek(pread, 0, SEEK\_END);  g\_fileSize2 = ftell(pread);//函数 ftell 用于得到文件位置指针当前位置相对于文件首的偏移字节数  fseek(pread, 0, SEEK\_SET);//读指针移到文件开始地方，准备开始读  //把文件大小发给客户端  file.fileInfo.fileSize = g\_fileSize2;  char tfname[200] = { 0 }, text[100];  \_splitpath(fileName, NULL, NULL, tfname, text);//获取文件的名字和属性  strcat(tfname, text);  //把文件名发给客户端  strcpy(file.fileInfo.fileName, tfname);  //读取文件内容  //分配内存空间  g\_fileBuf = calloc(g\_fileSize2 + 1, sizeof(char));  if (g\_fileBuf == NULL){  printf("内存不足，请重试\n");  return false;  }  fread(g\_fileBuf, sizeof(char), g\_fileSize2, pread);  g\_fileBuf[g\_fileSize2] = '\0';  fclose(pread);  send(serfd, (char\*)&file, sizeof(struct MsgHeader), 0);  } |

We use the gets\_s function to read a line of input from the user and store it in the fileName array. Then the code set msgID = MSG\_UPFILENAME anduse the strcpy function to copy the contents of the fileName array to the fileName member of the fileInfo struct within the file struct.

Then we open the file specified by fileName in binary read mode using fopen. IIf the file is found, it uses fseek with SEEK\_END to move the file pointer to the end of the file and retrieves the file size using ftell. The file size is stored in the fileSize member of the fileInfo struct within the file struct. It uses \_splitpath function to split the fileName into its components (drive, directory, file name, and extension), storing the file name and extension in the tfname array.It copies the contents of tfname to the fileName member of the fileInfo struct within the file struct. This ensures that the modified file name (without the directory information) is stored in the file struct.

Next step we allocate memory using calloc to store the file content. The size of the allocated memory is g\_fileSize2 + 1 bytes, and each byte is the size of char. The g\_fileSize2 represents the file size obtained earlier.

Finally, we send the result message to the server to let the server send data.

The server receive the request and set parameter:

|  |
| --- |
| void receiveFile(SOCKET clifd, struct MsgHeader\* pmsg) {  strcpy(g\_fileName, pmsg->fileInfo.fileName);  //准备内存 pmsg->fileInfo.fileSize  g\_fileSize = pmsg->fileInfo.fileSize;  //申请空间  g\_fileBuf = calloc(g\_fileSize + 1, sizeof(char));  if (g\_fileBuf == NULL)//内存申请失败  {  printf("内存不足，请重试\n");  }  else  {  struct MsgHeader msg;  msg.msgID = MSG\_UPFILE;  //printf("文件发送失败\n");  if (SOCKET\_ERROR == send(clifd, (char\*)&msg, sizeof(struct MsgHeader), 0))  {  err("send");  return;  }  }  } |

The server use the strcpy function to copy the contents of pmsg->fileInfo.fileName to the g\_fileName variable which updates the g\_fileName variable with the received file name.

Then the server allocates memory using calloc to store the file content according to the sending file size.

If the memory allocation is successful, it send the message with message MSG\_UPFILE to ask the client start to send data.

**Second step**

The client starts to send a file：

|  |
| --- |
| bool upsendFile(SOCKET clifd, struct MsgHeader\* pmsg)  {  //告诉客户端准备接受文件了  //如果文件的长度大于每个数据包能传送的大小(packet\_size)那么就分快  struct MsgHeader msg;  msg.msgID = MSG\_UP\_READY\_READ;  for (size\_t i = 0; i < g\_fileSize2; i += PACKET\_SIZE)  {  msg.packet.nStart = i;  //判断是否为最后一个包，获取最后一个包的长度  if (i + PACKET\_SIZE + 1 > g\_fileSize2)  {  //printf("upsendFile1");  msg.packet.nsize = g\_fileSize2 - i;  }  else  {  msg.packet.nsize = PACKET\_SIZE;  }  memcpy(msg.packet.buf, g\_fileBuf + msg.packet.nStart, msg.packet.nsize);  if (SOCKET\_ERROR == send(clifd, (char\*)&msg, sizeof(struct MsgHeader), 0))  {  printf("upsendFile3");  printf("文件发送失败%d\n", WSAGetLastError());  return false;  }  }  return TRUE;  } |

The function enters a loop that iterates over the file content in chunks of size PACKET\_SIZE. PACKET\_SIZE represents the maximum size of each data packet that can be sent.Within the loop, it sets the nStart member of the packet struct within the msg struct to the current position (i) in the file content. It checks if the current chunk is the last one by comparing i + PACKET\_SIZE + 1 with g\_fileSize2 (the total file size). If it is the last chunk, it sets the nsize member of the packet struct to the remaining file size (g\_fileSize2 - i). Otherwise, it sets the nsize member to PACKET\_SIZE.

It uses memcpy to copy the chunk of data from g\_fileBuf (the file content buffer) starting at the current position (msg.packet.nStart) with a size of msg.packet.nsize into the buf member of the packet struct within the msg struct.

After it copy the data in a package, it sends the data to the server.

Finally after all the data has been sent, we quit the loop and the function return a TRUE。

The server starts to receive a file：

|  |
| --- |
| bool upwriteFile(SOCKET serfd, struct MsgHeader\* pmsg)  { //如果g\_fileBUf为空，则分配内存失败  if (g\_fileBuf == NULL){  return FALSE;  }  //将指针移到开始读的位置  int nStart = pmsg->packet.nStart;  //获取包长度  int nsize = pmsg->packet.nsize;  //拼接  memcpy(g\_fileBuf + nStart, pmsg->packet.buf, nsize);  //判断文件是否完整，服务器是否发完了数据，接受到最后一个包时开始写入  if (nStart + nsize >= g\_fileSize){  FILE\* pwrite = fopen(g\_fileName, "wb");  if (pwrite == NULL)  {  printf("write file error..\n");  return false;  }  fwrite(g\_fileBuf, sizeof(char), g\_fileSize, pwrite);  //写完释放缓存，关闭读指针  fclose(pwrite);  free(g\_fileBuf);  g\_fileBuf = NULL;  //发送写入成功消息  struct MsgHeader msg;  msg.msgID = MSG\_SUCCESSED;  if (SOCKET\_ERROR == send(serfd, (char\*)&msg, sizeof(struct MsgHeader), 0))  {  err("send");  return;  }  }  return true;  } |

First the server checks if g\_fileBuf is NULL, which would indicate a memory allocation failure. If g\_fileBuf is indeed NULL, it returns FALSE to indicate failure.

Then it retrieves the starting position (nStart) and size (nsize) of the packet from the packet struct within pmsg. It uses memcpy to copy the data from pmsg->packet.buf (the received packet data) into g\_fileBuf at the appropriate position (g\_fileBuf + nStart) with a size of nsize. After coping it checks if the received packet completes the file. If the sum of the starting position and size is greater than or equal to g\_fileSize (the total file size), it means that the server has received the last packet. If it's the last packet, it opens the file specified by g\_fileName in binary write mode ("wb") using fopen. If the file opening fails (indicated by a NULL return value), it prints an error message and returns false.

After reading all the data, it uses fwrite to write the contents of g\_fileBuf to the file. The size of each element is sizeof(char), and the number of elements to write is g\_fileSize (the total file size).It closes the file using fclose, freeing the resources associated with it. It frees the memory allocated for g\_fileBuf using free and sets it to NULL to indicate that the buffer is no longer valid.

Fianlly assigns the value MSG\_SUCCESSED to the msgID indicating client a successful write operation.

1. **The difficulties**

1) I met difficulties in connecting char data in the change path, since in the windows system we need “//” for a path, then I learn to use strcpy and strncat for me to manipulating char. In order to successfully change path, I also use a global variable for the whole codes.

2) I met difficulties in sending file wit large data. Then I try to divide the uploading file into 2 steps. The first step is used to check whether the memory is enough for the file to transmit. Then in the second step, I send data package by package for sending a large file. It is hard for me divide file at first. Then I learn the function fseek and ftell for me to calculate the data. And I use memcpy to send data package by package and receive data package by package.

# Conclusion

**Task4:**

From task 4, when I develop FTP functionality using low-level C language. I met various challenges, such as file retrieval based on user-specified filenames, handling packet size limitations, managing message and file transfer integration, and addressing issues like buffer overflow during message transmission. However, finally by deeply learning how to use winsocket(the socket library in windows) and the char function in the C, I finished the task by sending data by packet and using char function for operating the file data such as copying and connecting. After this experiment, I deeply learned the socket work flow, initial, bind , listen, accept and send in server and the connect in client. Then by truly sending files on the network, I have a deeply understanding of why sending by packet. By finishing the others small tasks, such as delete file, list all the file, I learn the file operation and clients send commend to server by send message. In order to finish all these tasks, I learn use the MSG\_ID in a message to indicate the different function that the server or the client should provide. Meanwhile, in order to implement the function that ordered by the users and in order to send data correctly, I learn the use of C standard library to manipulate the file and the char data, such as ftell, fseek, strcpy and strcat and so on.