操作系统

实验报告

实验名称： 进程通信

院系： 计算机学院·网络空间安全学院

班级：

学号：

姓名：

指导老师：匡林爱

日期：2020年10月20日

1. **实验目的和要求：**

要求设计实现进程间通信(Interprocess Communication)的程序，而不是同一进程的多个线程之间的通信。Win32 API提供了多种进程间通信的方法，包括剪贴板(Clipboard)、COM(OLE)、动态数据交换(Dynamic Data Exchange，DDE)、文件映射(File Mapping)、邮件槽(Mailslot)、管道(Pipes)、远程过程调用(RPC)、Windows套接字(Windows Sockets)等。本实验采用相对比较容易实现的有名管道(Named Pipe)实现进程间的通信。学生也可采用File Mapping或Mailslot自行设计本实验。File Mapping的使用可参考P\_C2文件夹中程序，该程序就是使用File Mapping作为进程通信工具来实现诸进程共享循环缓冲区和缓冲区指针in、out等的共享的，再结合信号量对象(semaphore object)来实现生产者-消费者问题的同步算法的。另外，采用Mailslot实现进程通信的程序在文件夹Mailslot中。邮件槽是一种不定长和不可靠的单向消息通信机制。Named Pipe和File Mapping都是双向通信方式。

【**注**】本实验用的Named Pipe属于共享文件的通信，而“进程同步”实验中用的File Mapping属于共享存储区的通信。

1. **实验内容：**

本实验是在Windows 2000/XP+VC++6.0环境下利用Win32 API设计实现的；进程通信机制采用有名管道；程序分两个部分，即Named pipe server和Named pipe client。

在Windows 2000/XP中的管道是一条在进程间以字节流方式传送的通信通道，Win32 API提供两种管道：无名管道(anonymous pipes)和赋名管道(named pipes)。无名管道用于有关联的进程之间的信息交换(通信)，其典型的使用方式是用于标准I/O的重定向，从而使子进程可以同父进程交换信息。为了实现双向交换信息，需要建立两个无名管道。无名管道不能实现无关联进程之间的通信。

有名管道(named pipes)可用于无关联进程，甚至不同计算机中的进程之间的通信。典型的使用方式是：有名管道的服务器(*named-pipe server*)进程用一个众所周知的名字，利用CreateNamedPipe创建一个named pipe，而知道管道名字的客户进程(*named-pipe client*)就可以用OpenFile打开此管道并获得其句柄，并使用此句柄调用读操作(ReadFile)和写操作(WriteFile)来读写管道，从而实现与服务器进程的通信。注：上述ReadFile和WriteFile是阻塞方式工作的，也可使用非阻塞方式的ReadFileEx和WriteFileEx来读写管道。

在Windows 2000/XP系统中，要使用管道实现通信，都需要通过Win32 API调用来完成。本实验涉及的API函数如下：

**1．CreateNamedPipe函数**

**功能**：

创建一个有名管道实例(an instance of a named pipe)，返回管道句柄(handle)。随后对管道的操作都是对该handle进行的。管道服务器不仅可用此函数创建第一个命名管道实例，而且根据需要，以后还可以用此函数(管道名字不变)创建多个实例。

**格式**：

HANDLE CreateNamedPipe(

LPCTSTR lpName, // pipe name

DWORD dwOpenMode, // pipe open mode

DWORD dwPipeMode, // pipe-specific modes

DWORD nMaxInstances, // maximum number of instances

DWORD nOutBufferSize, // output buffer size

DWORD nInBufferSize, // input buffer size

DWORD nDefaultTimeOut, // time-out interval

LPSECURITY\_ATTRIBUTES lpSecurityAttributes // SD

);

**参数说明：**

**lpName**：管道名字符串，该名字必须是如下形式：\\.\pipe\*pipename*

**dwOpenMode**：指定pipe的打开模式。例如，取值PIPE\_ACCESS\_DUPLEX表示打开后对管道既可以读也可以写，即双向通信。

**dwPipeMode**：指定管道的类型，以及读，等待管道的模式。

**nMaxInstances**：指定能创建的最大实例数，其值范围在1~ PIPE\_UNLIMITED\_INSTANCES之间。

**nOutBufferSize**：输出缓冲区大小(字节数)

**nInBufferSize**：输入缓冲区大小(字节数)

**nDefaultTimeOut**：缺省的超时数，单位ms

**lpSecurityAttributes**：安全属性。NULL表示缺省的安全属性，以及返回的句柄不能被继承。

**2．ConnectNamedPipe函数**

**功能**：

named pipe服务器进程调用此函数，用于等待客户进程链接此命名管道的一个最新的实例。客户进程使用CreateFile or CallNamedPipe函数来连接命(有)名管道。

**格式**：

BOOL ConnectNamedPipe(

HANDLE hNamedPipe, // handle to named pipe

LPOVERLAPPED lpOverlapped // overlapped structure

);

**参数说明：**

**hNamedPipe**：命名管道实例的句柄，该句柄在调用CreateNamedPipe时获得。

**lpOverlapped**：指向一个OVERLAPPED结构。可取值NULL。

**返回值**：

当函数执行成功，返回非0值；当函数失败，返回0。如果client(客户)在Server(服务器)调用此函数之前已连接Pipe，则此函数返回0且GetLastError函数返回ERROR\_PIPE\_CONNECTED。如果一个客户在服务器调用CreateNamedPipe至调用ConnectNamedPipe时间之内连接管道，则这种情况就会发生。在这种情况下，虽然此函数返回0，但服务器和客户之间已建立了一个好的连接。即客户和服务器之间连接成功有两种情况：一是此函数返回非0值；二是此函数虽然返回0，但GetLastError函数返回ERROR\_PIPE\_CONNECTED。具体处理请参看本实验的参考程序。

**3．WaitNamedPipe函数**

**功能**：

调用进程(Named-Pipe Client)一直等待，直到给定时间超时，或者指定的named pipe可以被连接(即pipe服务器进程执行了ConnectNamedPipe操作)。

**格式**：

BOOL WaitNamedPipe(

LPCTSTR lpNamedPipeName, // pipe name

DWORD nTimeOut // time-out interval

);

**参数说明：**

**lpNamedPipeName**：命名管道名字。

**nTimeOut**：指定超时的毫秒数。

【注】若指定名字的管道不存在，则此函数立即返回(并不等待超时)。

**4．SetNamedPipeHandleState函数**

**功能**：

设置指定管道的读模式和阻塞模式(设置/改变管道的新模式)。

**格式**：

BOOL SetNamedPipeHandleState(

HANDLE hNamedPipe, // handle to named pipe

LPDWORD lpMode, // new pipe mode

LPDWORD lpMaxCollectionCount, // maximum collection count

LPDWORD lpCollectDataTimeout // time-out value

);

**参数说明**

**hNamedPipe**：管道的句柄。

**lpMode**：指定命名管道新的模式，包括读模式PIPE\_READMODE\_BYTE和PIPE\_READMODE \_MESSAGE(它们可以组合)以及等待模式PIPE\_WAIT或PIPE\_NOWAIT。

**lpMaxCollectionCount**：当handle在服务器端或服务器和客户在同一台计算机时，此值必须为NULL

**lpCollectDataTimeout**：当handle在服务器端或服务器和客户在同一台计算机时，此值必须为NULL

**5．WriteFile函数**

该函数已在“上机实验八 设备管理——磁盘I/O”实验中作了简单介绍，这里做一些补充：

当用此函数写一个管道时，如果管道缓冲区(其大小由调用CreateNamedPipe函数指定)已写满但仍有要写的数据，则写操作可能不能完成(即WriteFile函数不能返回)。但读端进程调用ReadFile读管道而使管道缓冲区空出更多可用空间时，最终能使写操作完成(即能使WriteFile函数返回)。

**6．ReadFile函数**

该函数已在“上机实验八 设备管理——磁盘I/O”实验中作了简单介绍，这里做一些补充：

(1) 该函数在出现下列情况之一才会返回：管道(pipe)写端的写操作已经完成；已经读到函数所请求读的字节数的信息；出现一个错误。因此，对于管道而言，当写端进程没有写完全部数据时，读端进程的读不会结束(阻塞方式)，这在某种程度上起到了同步作用。

(2) 当一个命名管道的读模式设置成PIPE\_READMODE\_MESSAGE，而所读的消息长度大于此函数所指定要读的字节数时，则ReadFile函数出错返回FALSE，此时调用GetLastError函数，获得的出错代号为ERROR\_MORE\_DATA。这种情况实际上数据已经正确读入，只是没有读完而已，可继续读。具体请看本实验的源程序。

**7．DisconnectNamedPipe函数**

**功能**：用于在named pipe服务器端断开与管道客户进程的连接。

**格式**：

BOOL DisconnectNamedPipe(

HANDLE hNamedPipe // handle to named pipe

);

**8．FlushFileBuffers函数**

**功能**：将指定文件的缓冲数据写入磁盘并请缓冲区。

**格式**：

BOOL FlushFileBuffers(

HANDLE hFile // handle to file

);

**8．其它函数**

CreateFile： 该函数已在“上机实验八 设备管理——磁盘I/O”实验中作了简单介绍，这里用此函数打开一个命名管道；

CreateThread：创建线程，参看“创建线程”实验的介绍。

CloseHandle：关闭句柄。

CreateProcess：创建进程，参看“创建进程”实验的介绍。

1. **实验步骤：**

1．首先创建一个工作线程(WorkThread)，它的任务是创建若干个进程(pipe client)；

2．主线程开始无限循环：调用CreateNamedPipe创建一个有名管道实例；调用ConnectNamedPipe等待管道客户进程请求服务；若有客户进程请求管道连接，则创建一个子线程来为该客户提供服务(与该客户实现管道双向通信)；主线程返回循环开头。

3．若没有客户请求连接，则关闭刚创建的管道实例(named pipe instance)的句柄后重新循环。

管道客户程序的总体框架是：

1．调用CreateFile函数获得服务器创建的管道实例的句柄；

2．调用WaitNamedPipe函数，等待(请求)与服务器的管道连接；

3．若连接成功，如需要可则调用SetNamedPipeHandleState按需要设置管道的读模式等；

4．执行如下的循环操作：

(1) 调用WriteFile函数向服务器发送请求信息；

(2) 等待并接收服务器的应答信息(用ReadFile函数实现)。这样循环往复，实现与服务器的双向通信，直到完成通信任务后结束进程。

服务器创建的为客户提供服务的子线程也是用ReadFile和WriteFile函数实现与客户进程的管道通信的，此处不再赘述。

1. **实验源程序**

**PipeClient.cpp:**

#include <iostream>

#include <windows.h> //Sleep()

#include <stdio.h>

#include <string.h>

using namespace std;

#define BUFFSIZE 128

void MyErrExit(char \*Err)

{

cout << "Error : " << Err << endl;

\_flushall(); //清除缓冲区

getchar(); //等待按键

exit(0);

}

char Request[][128] = { "Hello, I am pipe client",

"My name is Tom. What is your name?",

"Can you tell me something about pipe?",

"Thank you, Peter. Goodbye." };

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

{

if (argc < 2)

MyErrExit("CommandLine");

HANDLE hPipe;

LPVOID lpvMessage;

CHAR chBuf[BUFFSIZE], chwBuf[BUFFSIZE];

char name[20 + BUFFSIZE] = "Client ";

BOOL fSuccess;

DWORD cbRead, cbWritten, dwMode;

DWORD Messagesize;

int xx = 0;

LPTSTR lpszPipename = "\\\\.\\pipe\\mynamedpipe";

char mutexName[20] = "smutex";

strcat\_s(mutexName, argv[1]);

strcat\_s(name, argv[1]);

strcat\_s(name, " : ");

// Try to open a named pipe; wait for it, if necessary.

while (1)

{

hPipe = CreateFile(

lpszPipename, // pipe name

GENERIC\_READ | // read and write access

GENERIC\_WRITE,

0, // no sharing

NULL, // no security attributes

OPEN\_EXISTING, // opens existing pipe

0, // default attributes

NULL); // no template file

// Break if the pipe handle is valid.

if (hPipe != INVALID\_HANDLE\_VALUE)

break;

// Exit if an error other than ERROR\_PIPE\_BUSY occurs.

if (GetLastError() != ERROR\_PIPE\_BUSY)

MyErrExit("1 Could not open pipe");

// All pipe instances are busy, so wait for 20 seconds.

if (!WaitNamedPipe(lpszPipename, 20000))

MyErrExit("2 Could not open pipe");

}

// The pipe connected; change to message-read mode.

dwMode = PIPE\_READMODE\_MESSAGE;

fSuccess = SetNamedPipeHandleState(

hPipe, // pipe handle

&dwMode, // new pipe mode

NULL, // don't set maximum bytes

NULL); // don't set maximum time

if (!fSuccess)

MyErrExit("SetNamedPipeHandleState");

// Send a message to the pipe server.

strcpy\_s(chwBuf, name);

strcat\_s(chwBuf, Request[xx]);

lpvMessage = chwBuf;

Messagesize = strlen(chwBuf) + 1;

fSuccess = WriteFile(

hPipe, // pipe handle

lpvMessage, // message

Messagesize, // message length

&cbWritten, // bytes written

NULL); // not overlapped

if (!fSuccess)

MyErrExit("WriteFile");

do

{

fSuccess = ReadFile(

hPipe, // pipe handle

chBuf, // buffer to receive reply

BUFFSIZE, // size of buffer

&cbRead, // number of bytes read

NULL); // not overlapped

if (!fSuccess && GetLastError() != ERROR\_MORE\_DATA)

MyErrExit("ReadFile");

cout << "Server : " << chBuf << endl;

xx++;

if (xx > 3)

break;

strcpy\_s(chwBuf, name);

strcat\_s(chwBuf, Request[xx]);

lpvMessage = chwBuf;

Messagesize = strlen(chwBuf) + 1;

fSuccess = WriteFile(

hPipe, // pipe handle

lpvMessage, // message

Messagesize, // message length

&cbWritten, // bytes written

NULL); // not overlapped

if (!fSuccess)

MyErrExit("WriteFile");

Sleep(1000);//等待1000ms，为了便于观察并发执行的情况

} while (fSuccess);//while (! fSuccess); // repeat loop if ERROR\_MORE\_DATA

CloseHandle(hPipe);

cout << "\n \*\*\*\*\*\*\*\* pipe client" << argv[1] << " will exit \*\*\*\*\*\*\*\*" << endl;

cout << "\n Press Enter key to exit." << endl;

\_flushall(); //清除缓冲区

getchar(); //等待按键

return 1;

}

**PipeServer.cpp:**

#include <windows.h>

#include <iostream.>

using namespace std;

#define BUFSIZE 128

#define PIPE\_TIMEOUT 2000

HANDLE createProcess(char\* name, int num);

VOID MyErrExit(char \*);

DWORD InstanceThread(LPVOID);

DWORD WorkThread(LPVOID);

char Reply[][128] = { "Hello, I am pipe server",

"My name is Peter. Tom, welcome.",

"Certianly, pipe can be used in\n\t interprocess communication.",

"Bye-bye, Tom." };

int threadNo = 0; // serial number of thread instance

CRITICAL\_SECTION cs\_Screen; // CRITICAL\_SECTION object

// The following example is a multithreaded pipe server. It has

// a main thread with a loop that creates a pipe instance and

// waits for a pipe client to connect. When a pipe client

// connects, the pipe server creates a thread to service that

// client and then continues to execute the loop. It is

// possible for a pipe client to connect successfully to the

// pipe instance in the interval between calls to the

// CreateNamedPipe and ConnectNamedPipe functions. If this

// happens, ConnectNamedPipe returns zero, and GetLastError

// returns ERROR\_PIPE\_CONNECTED.

// The thread created to service each pipe instance reads

// requests from the pipe and writes replies to the pipe

// until the pipe client closes its handle. When this

// happens, the thread flushes the pipe, disconnects,

// closes its pipe handle, and terminates.

int main() // Pipe Server

{

BOOL fConnected;

DWORD dwThreadId;

HANDLE hPipe, hThrd, hwkThd;

int nProcess = 3; // number of pipe client

// name of the pipe

char add1[] = "\\\\.\\pipe\\mynamedpipe";

LPTSTR lpszPipename = add1;

// init. CRITICAL\_SECTION object cs\_Screen

InitializeCriticalSection(&cs\_Screen);

HWND hWindow;

// obtain a handle to the foreground window

hWindow = GetForegroundWindow();

// changes the position and dimensions of the specified window

MoveWindow(hWindow, 0, 10, 750, 385, TRUE);

// create a thread. it create processes of pipe client

hwkThd = CreateThread(

NULL, // no security attribute

0, // default stack size

(LPTHREAD\_START\_ROUTINE)WorkThread,

(LPVOID)nProcess, // thread parameter

0, // not suspended

&dwThreadId); // returns thread ID

if (hwkThd == NULL)

MyErrExit("CreateThread1");

// The main loop creates an instance of the named pipe and

// then waits for a client to connect to it. When the client

// connects, a thread is created to handle communications

// with that client, and the loop is repeated.

for (;;) // main loop

{

// create an instance of the named pipe

hPipe = CreateNamedPipe(

lpszPipename, // pipe name

PIPE\_ACCESS\_DUPLEX, // read/write access

PIPE\_TYPE\_MESSAGE | // message type pipe

PIPE\_READMODE\_MESSAGE | // message-read mode

PIPE\_WAIT, // blocking mode

PIPE\_UNLIMITED\_INSTANCES, // max. instances

BUFSIZE, // output buffer size

BUFSIZE, // input buffer size

PIPE\_TIMEOUT, // client time-out

NULL); // no security attribute

if (hPipe == INVALID\_HANDLE\_VALUE)

MyErrExit("CreatePipe");

// Wait for the client to connect; if it succeeds,

// the function returns a nonzero value. If the function returns

// zero, GetLastError returns ERROR\_PIPE\_CONNECTED.

fConnected = ConnectNamedPipe(hPipe, NULL) ?

TRUE : (GetLastError() == ERROR\_PIPE\_CONNECTED);

if (fConnected)

{

// Create a thread for this client.

threadNo++;

hThrd = CreateThread(

NULL, // no security attribute

0, // default stack size

(LPTHREAD\_START\_ROUTINE)InstanceThread,

(LPVOID)hPipe, // thread parameter

0, // not suspended

&dwThreadId); // returns thread ID

if (hThrd == NULL)

MyErrExit("CreateThread2");

}

else

// The client could not connect, so close the pipe.

CloseHandle(hPipe);

}

return 1;

}

DWORD InstanceThread(LPVOID lpvParam)

{

int nThread = threadNo;

CHAR chRequest[BUFSIZE];

CHAR chReply[BUFSIZE];

DWORD cbBytesRead, cbReplyBytes, cbWritten;

BOOL fSuccess;

int index = 0;

HANDLE hPipe;

// The thread's parameter is a handle to a pipe instance.

hPipe = (HANDLE)lpvParam;

while (1)

{

// Read client requests from the pipe.

fSuccess = ReadFile(

hPipe, // handle to pipe

chRequest, // buffer to receive data

BUFSIZE, // size of buffer

&cbBytesRead, // number of bytes read

NULL); // not overlapped I/O

if (!fSuccess || cbBytesRead == 0)

break;

EnterCriticalSection(&cs\_Screen);

cout << chRequest << endl;

LeaveCriticalSection(&cs\_Screen);

strcpy\_s(chReply, Reply[index]);

cbReplyBytes = strlen(Reply[index]) + 1;

index = (index + 1) % 4;

// Write the reply to the pipe.

fSuccess = WriteFile(

hPipe, // handle to pipe

chReply, // buffer to write from

cbReplyBytes, // number of bytes to write

&cbWritten, // number of bytes written

NULL); // not overlapped I/O

if (!fSuccess || cbReplyBytes != cbWritten)

MyErrExit("Write the pipe failrd.");

Sleep(1000); //等待1000ms，为了便于观察并发执行的情况

}

// Flush the pipe to allow the client to read the pipe's contents

// before disconnecting. Then disconnect the pipe, and close the

// handle to this pipe instance.

FlushFileBuffers(hPipe);

DisconnectNamedPipe(hPipe);

CloseHandle(hPipe);

EnterCriticalSection(&cs\_Screen);

cout << "\n \*\*\*\*\*\* InstanceThread "

<< nThread << " exit." << endl << endl;

LeaveCriticalSection(&cs\_Screen);

return 1;

}

DWORD WorkThread(LPVOID lpPara)

{

int nClint = (int)lpPara;

char name[20] = "PipeClient";

for (int i = 1; i <= nClint; i++) // create pipe client

{

createProcess(name, i);

}

return 1;

}

HANDLE createProcess(char \*name, int num)

{

static int position = 1;

char cmdline[256];

char ich[8];

\_itoa\_s(num, ich, 10); // number of process to string

strcpy\_s(cmdline, name);

strcat\_s(cmdline, " ");

strcat\_s(cmdline, ich); // command-line

STARTUPINFO si = { sizeof(STARTUPINFO) };

si.dwFlags = STARTF\_USEPOSITION | // enable dwX and dwY

STARTF\_USESIZE | // enable dwXSize and dwYSize

STARTF\_USEFILLATTRIBUTE; // enable dwFillAttribute

si.dwXSize = 365; // width, in pixels, of the window

si.dwYSize = 380; // height, in pixels, of the window

si.lpTitle = cmdline; // title of window

// produces blue text on a whilte background

si.dwFillAttribute = FOREGROUND\_BLUE | BACKGROUND\_RED | BACKGROUND\_GREEN | BACKGROUND\_BLUE;

PROCESS\_INFORMATION pi;

DWORD dwx = 0, dwy = 0; // the upper left corner of a window

switch (position)

{

case 0: si.dwX = dwx + 770; // the upper left corner of a window 0

si.dwY = dwy;

break;

case 1: si.dwX = dwx; // the upper left corner of a window 1

si.dwY = dwy + 400;

break;

case 2: si.dwX = dwx + 385; // the upper left corner of a window 2

si.dwY = dwy + 400;

break;

case 3: si.dwX = dwx + 770; // the upper left corner of a window 3

si.dwY = dwy + 400;

break;

}

BOOL fsuccess = CreateProcess(

NULL, // no executable module

cmdline,// command line string

NULL, // the returned handle cannot be inherited by child processes

NULL, // the returned handle cannot be inherited by child processes

TRUE, // each inheritable open handle in the calling

// process is inherited by the new process

NORMAL\_PRIORITY\_CLASS | CREATE\_NEW\_CONSOLE,

// The new process has a new console

NULL, // new process uses the environment of the calling process

NULL, // new process is created with the same current drive

// and directory as the calling process

&si, // Pointer to a STARTUPINFO structure that specifies how

// the main window for the new process should appear

&pi // Pointer to a PROCESS\_INFORMATION structure that receives

// identification information about the new process

);

if (!fsuccess)

MyErrExit("CreateProcess");

position = (position + 1) % 4;

return pi.hThread;

}

VOID MyErrExit(char \*Err)

{

EnterCriticalSection(&cs\_Screen);

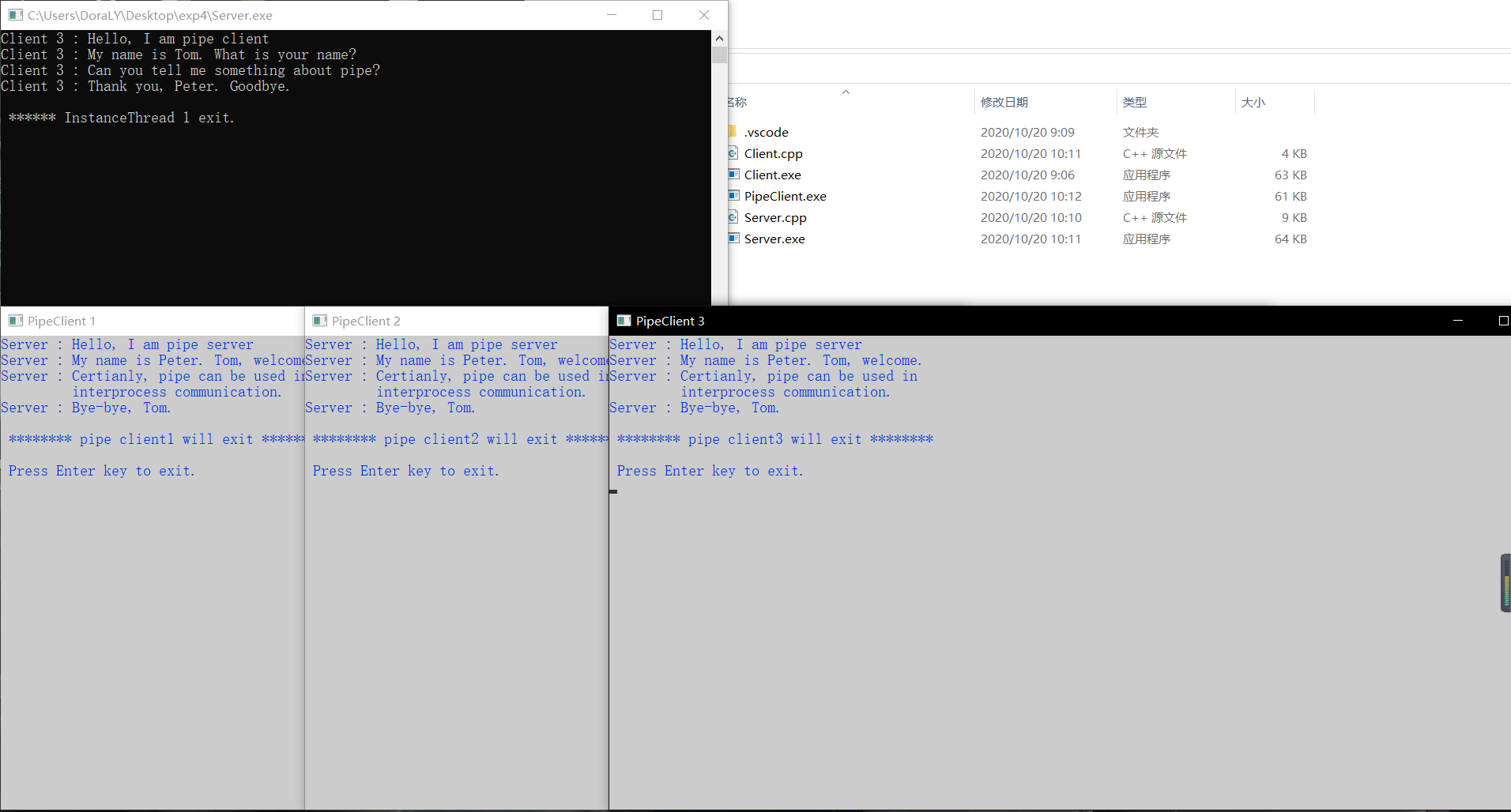
cout << "Error : " << Err << endl;

LeaveCriticalSection(&cs\_Screen);

exit(0);

}

1. **实验结果**



1. **实验总结：**

**通过这次进程通信实验，我了解到了Win32API提供的进程通信方法例如：剪贴板、COM、动态数据交换、文件映射、邮件槽、管道、远程过程调用、Windows套接字、有名管道等，尤其是本次实验我们所采用的有名管道法，我们通过创建管道客户进程以及创建子进程实现了管道通信。代码中调用了诸如CreateNamedPipe，ConnectNamedPipe等API函数，学会了他们的用法之后我觉得对我以后的代码生涯也大有益处！我在实验期间还详细查找了缓冲区的相关资料，通过阅读这些资料，我也弄懂了许多之前没有弄懂的问题。这次成功的进程通信实验也增强了我的自信心，我想通过自己的努力去实现其他方法的进程通信，总而言之，这次实验也和之前的实验一样，拓宽了我的眼界，使我更加熟悉了代码编写方面的操作，我现在已经有了一个比较纯熟的处理手段，我相信这对我以后的工作生活一定会起到很大的帮助！**