Ficha nº 7 – Comunicação interprocesso com Named pipes

Âmbito da matéria

Os exercícios desta ficha abordam:

- Manipulação ficheiros através de funções da API Windows para ficheiros.
- Comunicação entre processos com modelo cliente-servidor.
- Named pipes.
- Servidor de named-pipes single-threaded e multi-threaded.

Pressupostos

- Conhecimento de algoritmia e de programação em C / C++.
- Manipulação de ficheiros em C.
- Conhecimentos de conceitos de base em SO (1º semestre).
- Conhecimento da matéria das aulas anteriores e das aulas teóricas (*threads* e sincronização)

Referências bibliográficas

- Material das aulas teóricas
- Capítulos 2 e 8 do Livro Windows System Programming (da bibliografia) (25-28, 379-384-390)
- MSDN:

Named pipes

https://msdn.microsoft.com/en-us/library/windows/desktop/aa365590%28v=vs.85%29.aspx

Synchronous and Overlapped Pipe I/O

https://msdn.microsoft.com/en-us/library/windows/desktop/aa365788(v=vs.85).aspx

Exemplo Servidor Named pipe multi-threaded

https://docs.microsoft.com/en-us/windows/win32/ipc/multithreaded-pipe-server

Estes links apontam para o topo dos tópicos. Os sítios contêm links para os assuntos mais específicos subordinados ao tema.

Introdução e contexto

Esta ficha aborda *named pipes*. Em Win32, os *named pipes*:

- Estão fortemente associados ao modelo cliente-servidor
- Um *named pipe* é um recurso do qual pode haver várias instâncias. Cada instância diz respeito ao servidor e um cliente em particular. Clientes diferentes terão instâncias diferentes.
- Se os clientes se mantiverem a interagir como o servidor será necessário uma *thread* por cliente no lado do servidor.
- O mesmo *named pipe* suporta comunicação bidirecional, mas não em simultâneo.
- A bidireccionalidade do *named pipe* evita a necessidade de os clientes criarem "pipes para resposta"
- A escrita e leitura em simultâneo poderá implicar o uso de operações de leitura/escrita assíncronas
- Os *named pipes* suportam diversos modos (por exemplo, *byte/message*) e essa configuração tem que coincidir entre servidor e cliente. Nem sempre o *default* é o que se pretende.

É bastante arriscado assumir que este resumo é tudo o que há para saber acerca de *named pipes* em Windows. Existem bastantes pormenores e o uso deste recurso é mais complexo do que o equivalente noutros sistemas (por exemplo, Unix). A exposição nas aulas teóricas é importante.

Padrão de uso simplificado para os casos mais comuns

Servidor

- O servidor chama a função **CreateNamedPipe** para criar a primeira instância (e seguintes) do *named pipe*
- O servidor usa a função **ConnectNamedPipe** para aguardar um pedido de ligação à instância do *named pipe*.
    - O atendimento dessa instância pode ser feito numa *thread* independente, libertando o servidor para lidar com outros clientes
- No final da interação com o cliente o servidor garante que os dados já tenham sido lidos pelo cliente usando a função **FlushFileBuffers** E depois desligar a instância do *named pipe* com **DisconnectNamedPipe** e **CloseHandle**

Cliente

- O cliente utiliza a função **CreateFile** ou **CallNamedPipe** para obter um *handle* para uma instancia do named pipe
- A função **WaitNamedPipe** permite ao ciente aguardar que exista uma instância do servidor do *named pipe* disponível (sada cliente que se liga ocupa uma instância).
- O cliente interage com o servidor de acordo com um protocolo qualquer predefinido para essa aplicação cliente-servidor
    - Pode ser necessário ajustrar as propriedades do named pipe aberto para coincidir com a configuração no servidor: função **SetNamedPipeHandleState**
- O cliente termina a interação fechando os *handles*, libertando a instância do *named pipe*.

O protocolo e significado da informação a circular no *named pipe* é da inteira responsabilidade dos programas intervenientes. O sistema apenas faz a gestão do recurso.

-> Esta introdução (resumo) não dispensa as aulas teóricas sobre este assunto.

Exercícios

Estes exercícios são razoavelmente independentes entre si. No entanto, o código de um pode servir de base para ajudar a elaboração do seguinte, poupando algum tempo.

Os exercícios devem ser resolvidos recorrendo a projetos do tipo Win32 Console, inicialmente vazios. Os ficheiros podem ter extensão *.cpp*, mas deve optar por *.c* (a não ser que sejam mesmo necessários mecanismos de C++).

1. Utilizando apenas funções da API do Windows para manipular ficheiros, implemente um programa que copia um ficheiro passado pela linha de comandos e cria uma cópia cujo o nome também é passado quando o programa é invocado:

   mycopy ficheiro.c novo_ficheiro.c

   Este exercício não tem nenhuma complexidade algorítmica e destina-se apenas a apresentar as funções de ficheiros específicas de WIn32 (CreateFile, OpenFile, ReadFile, WriteFile, CloseHandle). Dada a sua simplicidade, pode ser feito em casa.

2. Os programas (apresentados na secção de listagens) *escritor* e *leitor* permitem a troca de mensagens entre dois processos através de Named Pipes. A troca de mensagens com um dado leitor termina quando o utilizador introduz a palavra "fim" que é lido pelo programa escritor.

   Construa e experimente o dois programas.

   Verifique o que acontece se forem lançados dois programas/processos leitor. Comunicarão estes em simultâneo com o único escritor?

   Nota: estes dois programas são usados nos exercícios seguntes.

3. Continue os programas do exercício anterior. De forma a ter um comportamento que permita que todos os leitores recebam as notificações enviadas pelo único escritor, lance uma *thread* no programa escritor responsável apenas pela interação com a consola (ler frase e divulgar pelos vários leitores ligados). A *thread* inicial tem o papel de criar e abrir uma nova instância de Named Pipe para cada novo leitor e os guardar num *array* de N posições. A nova *thread* continua a escrever para o *named pipe*, só que desta vez para todas as instâncias de named pipes abertas dos leitores que se encontram ligados.

4. Crie uma pasta partilhada na rede no seu computador e verifique se o colega do lado consegue aceder à pasta partilhada por si. Em caso afirmativo, peça a este mesmo colega para aceder ao seu escritor, através de um leitor a correr na máquina dele modificando o nome do Named Pipe de forma a incluir o IP da sua máquina.

5. Como atualmente termina o processo escritor? O que acontece se chegar ao limite de leitores possíveis de serem atendidos? Termine o processo escritor de um forma elegante (sem sair com exit() por uma *thread* auxiliar, mas sim com *return* da *thread* principal depois de garantir que todas a(s) *thread(s*) criada(s) terminaram) quando introduzida a palavra "fim".

6. Altere o programa leitor e escritor de forma que um destes programas execute a seguinte tarefa: transforme a mensagem recebida em maiúsculas e devolva a resposta ao emissor. O outro programa apenas envia e fica à espera do resultado. A interação é terminada quando se introduz a palavra "fim". A qual destes programas daria o nome servidor? E qual seria o cliente? O servidor que implementou permite atender quantos clientes em simultâneo?

7. Altere os programa servidor e cliente do exercício 6 de modo que o servidor antes de executar o primeiro pedido de transformação da *string*, recebe um pedido de autenticação composto por uma estrutura com login e password. No servidor existe um *array* de utilizadores registados, da primeira vez que recebe o pedido de um utilizador ainda por registar, o servidor cria uma entrada no *array* de utilizadores registados e autentica este utilizador. Caso o utilizador já exista, autentica somente se a password coincidir com a que se encontra registada.

8. No exercício anterior, é necessário algum mecanismo de sincronização caso o servidor seja *multi-threaded*? Se sim, implemente-o.

9. De forma a notificar todos os utilizadores caso um novo utilizador entre no sistema, crie um mecanismo parecido com o exercício 3, onde todos os utilizadores ligados são notificados quando um utilizador se autentica.

(Listagem de programas na próxima página)

Listagem de Programas

escritor.c

```
// ...
#define PIPE_NAME TEXT("\\\\.\\pipe\\teste")

int _tmain(int argc, LPTSTR argv[]) {
     DWORD n;
     HANDLE hPipe;
     TCHAR buf[256];

#ifdef UNICODE
     _setmode(_fileno(stdin), _O_WTEXT);
     _setmode(_fileno(stdout), _O_WTEXT);
#endif

     _tprintf(TEXT("[ESCRITOR] Criar uma cópia do pipe '%s' ..."
                   " (CreateNamedPipe)\n"), PIPE_NAME);
     hPipe = CreateNamedPipe(PIPE_NAME, PIPE_ACCESS_OUTBOUND, PIPE_WAIT |
                             PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE, 1,
                             sizeof(buf), sizeof(buf), 1000, NULL);
     if (hPipe == INVALID_HANDLE_VALUE) {
          _tprintf(TEXT("[ERRO] Criar Named Pipe! (CreateNamedPipe)"));
          exit(-1);
     }

     while (1) {

          _tprintf(TEXT("[ESCRITOR] Esperar ligação de um leitor..."
                        " (ConnectNamedPipe)\n"));
          if (!ConnectNamedPipe(hPipe, NULL)) {
               _tprintf(TEXT("[ERRO] Ligação ao leitor! (ConnectNamedPipe\n"));
               exit(-1);
          }

          do {
               _tprintf(TEXT("[ESCRITOR] Frase: "));
               _fgetts(buf, 256, stdin);
               buf[_tcslen(buf) - 1] = '\0';
               if (!WriteFile(hPipe, buf, _tcslen(buf) * sizeof(TCHAR), &n, NULL)) {
                    _tprintf(TEXT("[ERRO] Escrever no pipe! (WriteFile)\n"));
                    exit(-1);
               }
               _tprintf(TEXT("[ESCRITOR] Enviei %d bytes ao leitor..."
                             " (WriteFile)\n"), n);
          } while (_tcscmp(buf, TEXT("fim")));

          _tprintf(TEXT("[ESCRITOR] Desligar o pipe (DisconnectNamedPipe)\n"));
```

```
          if (!DisconnectNamedPipe(hPipe)) {
               _tprintf(TEXT("[ERRO] Desligar o pipe! (DisconnectNamedPipe)"));
               exit(-1);
          }
     }
     Sleep(2000);
     CloseHandle(hPipe);
     exit(0);
}
```

leitor.c

```
// ...
#define PIPE_NAME TEXT("\\\\.\\pipe\\teste")

int _tmain(int argc, LPTSTR argv[]) {
     TCHAR buf[256];
     HANDLE hPipe;
     int i = 0;
     BOOL ret;
     DWORD n;

#ifdef UNICODE
     _setmode(_fileno(stdin), _O_WTEXT);
     _setmode(_fileno(stdout), _O_WTEXT);
#endif

     _tprintf(TEXT("[LEITOR] Esperar pelo pipe '%s' (WaitNamedPipe)\n"),
                    PIPE_NAME);
     if (!WaitNamedPipe(PIPE_NAME, NMPWAIT_WAIT_FOREVER)) {
          _tprintf(TEXT("[ERRO] Ligar ao pipe '%s'! (WaitNamedPipe)\n"), PIPE_NAME);
          exit(-1);
     }
     _tprintf(TEXT("[LEITOR] Ligação ao pipe do escritor... (CreateFile)\n"));
     hPipe = CreateFile(PIPE_NAME, GENERIC_READ, 0, NULL, OPEN_EXISTING,
                         FILE_ATTRIBUTE_NORMAL, NULL);
     if (hPipe == NULL) {
          _tprintf(TEXT("[ERRO] Ligar ao pipe '%s'! (CreateFile)\n"), PIPE_NAME);
          exit(-1);
     }
     _tprintf(TEXT("[LEITOR] Liguei-me...\n"));

     while (1) {
          ret = ReadFile(hPipe, buf, sizeof(buf), &n, NULL);
          buf[n / sizeof(TCHAR)] = '\0';
          if (!ret || !n) {
               _tprintf(TEXT("[LEITOR] %d %d... (ReadFile)\n"), ret, n);
               break;
          }
```

```
            _tprintf(TEXT("[LEITOR] Recebi %d bytes: '%s'... (ReadFile)\n"), n, buf);
        }
        CloseHandle(hPipe);
        Sleep(200);
        return 0;
}
```

(Resumo do API na próxima página)

Resumo das funções API mais centrais a estes exercícios

Notas

- Existem mais funções que estas. **Em momento algum deve ser feita a suposição que estas funções "chegam"**.
- Atenção à questão TCHAR/ …A() / …W().
  Tanto aparecem as versões agnósticas, como as A ou as W. Devem usar sempre as agnósticas.
- Os *handles* devem ser fechados quando o objeto deixa de ser necessário (*CloseHandle*).

A descrição das funções foi retirada do material online da Microsoft. Esse conteúdo está, naturalmente, sujeito a propriedade intelectual da Microsoft. O material é aqui colocado em contexto de divulgação académica e não deve ser transcrito para outros contextos.

- **Ficheiros e handles**

No que diz respeito a Named Pipes

OpenFile

Creates, opens, reopens, or deletes a file.

This function has limited capabilities and is not recommended. For new application development, use the CreateFile function.

CreateFileA

Creates or opens a file or I/O device. The most commonly used I/O devices are as follows: file, file stream, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, and pipe. The function returns a handle that can be used to access the file or device for various types of I/O depending on the file or device and the flags and attributes specified.

https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea

```
HANDLE CreateFileA(
  LPCSTR                lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

CloseHandle

Closes an open object handle.

https://docs.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-closehandle

```
BOOL CloseHandle(
  HANDLE hObject
);
```

ReadFile

Reads data from the specified file or input/output (I/O) device. Reads occur at the position specified by the file pointer if supported by the device.
This function is designed for both synchronous and asynchronous operations. For a similar function designed solely for asynchronous operation, see ReadFileEx.

https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile

```
BOOL ReadFile(
  HANDLE       hFile,
  LPVOID       lpBuffer,
  DWORD        nNumberOfBytesToRead,
  LPDWORD      lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

A pointer to an OVERLAPPED structure is required if the *hFile* parameter was opened with **FILE_FLAG_OVERLAPPED**, otherwise it can be **NULL**.

OVERLAPPED structure

Contains information used in asynchronous (or *overlapped*) input and output (I/O).

https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-overlapped

```
typedef struct _OVERLAPPED {
  ULONG_PTR Internal;
  ULONG_PTR InternalHigh;
  union {
    struct {
      DWORD Offset;
      DWORD OffsetHigh;
    } DUMMYSTRUCTNAME;
    PVOID Pointer;
  } DUMMYUNIONNAME;
  HANDLE    hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

hEvent

A handle to the event that will be set to a signaled state by the system when the operation has completed. The user must initialize this member either to zero or a valid event handle using the CreateEvent function before passing this structure to any overlapped functions. This event can then be used to synchronize simultaneous I/O requests for a device.

Functions such as ReadFile and WriteFile set this handle to the nonsignaled state before they begin an I/O operation. When the operation has completed, the handle is set to the signaled state.

Functions such as GetOverlappedResult and the synchronization wait functions reset auto-reset events to the nonsignaled state. Therefore, you should use a manual reset event; if you use an auto-reset event, your application can stop responding if you wait for the operation to complete and then call **GetOverlappedResult** with the *bWait* parameter set to **TRUE**.

Any unused members of this structure should always be initialized to zero before the structure is used in a function call. Otherwise, the function may fail and return **ERROR_INVALID_PARAMETER**.

ReadFileEx

Reads data from the specified file or input/output (I/O) device. It reports its completion status asynchronously, calling the specified completion routine when reading is completed or canceled and the calling thread is in an alertable wait state.
To read data from a file or device synchronously, use the ReadFile function.

https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfileex

```
BOOL ReadFileEx(
  HANDLE                          hFile,
  LPVOID                          lpBuffer,
  DWORD                           nNumberOfBytesToRead,
  LPOVERLAPPED                    lpOverlapped,
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

lpCompletionRoutine

A pointer to the completion routine to be called when the read operation is complete and the calling thread is in an alertable wait state. For more information about the completion routine, see FileIOCompletionRoutine.

The **ReadFileEx** function ignores the OVERLAPPED structure's **hEvent** member. An application is free to use that member for its own purposes in the context of a **ReadFileEx** call. **ReadFileEx** signals completion of its read operation by calling, or queuing a call to, the completion routine pointed to by *lpCompletionRoutine*, so it does not need an event handle.

LPOVERLAPPED_COMPLETION_ROUTINE callback function

An application-defined callback function used with the ReadFileEx and WriteFileEx functions. It is called when the asynchronous input and output (I/O) operation is completed or canceled and the calling thread is in an alertable state (by using the SleepEx, MsgWaitForMultipleObjectsEx, WaitForSingleObjectEx, or WaitForMultipleObjectsEx function with the fAlertable parameter set to TRUE).
The LPOVERLAPPED_COMPLETION_ROUTINE type defines a pointer to this callback function. FileIOCompletionRoutine is a placeholder for the application-defined function name.

https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/nc-minwinbase-lpoverlapped_completion_routine

```
LPOVERLAPPED_COMPLETION_ROUTINE LpoverlappedCompletionRoutine;

void LpoverlappedCompletionRoutine(
  DWORD dwErrorCode,
  DWORD dwNumberOfBytesTransfered,
  LPOVERLAPPED lpOverlapped
)
```

GetOverlappedResult

Retrieves the results of an overlapped operation on the specified file, named pipe, or communications device. To specify a timeout interval or wait on an alertable thread, use GetOverlappedResultEx.

https://docs.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-getoverlappedresult

```
BOOL GetOverlappedResult(
  HANDLE       hFile,
  LPOVERLAPPED lpOverlapped,
  LPDWORD      lpNumberOfBytesTransferred,
  BOOL         bWait
);
```

GetOverlappedResultEx

Retrieves the results of an overlapped operation on the specified file, named pipe, or communications device within the specified time-out interval. The calling thread can perform an alertable wait.

https://docs.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-getoverlappedresultex

```
BOOL GetOverlappedResultEx(
  HANDLE       hFile,
  LPOVERLAPPED lpOverlapped,
  LPDWORD      lpNumberOfBytesTransferred,
  DWORD        dwMilliseconds,
  BOOL         bAlertable
);
```

WriteFile

Writes data to the specified file or input/output (I/O) device.
This function is designed for both synchronous and asynchronous operation. For a similar function designed solely for asynchronous operation, see WriteFileEx.

https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile

```
BOOL WriteFile(
  HANDLE       hFile,
  LPCVOID      lpBuffer,
  DWORD        nNumberOfBytesToWrite,
  LPDWORD      lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

WriteFileEx

Writes data to the specified file or input/output (I/O) device. It reports its completion status asynchronously, calling the specified completion routine when writing is completed or canceled and the calling thread is in an alertable wait state.
To write data to a file or device synchronously, use the WriteFile function.

https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefileex

```
BOOL WriteFileEx(
  HANDLE                          hFile,
  LPCVOID                         lpBuffer,
  DWORD                           nNumberOfBytesToWrite,
  LPOVERLAPPED                    lpOverlapped,
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

- **Named pipes**

Tema desta ficha
Parte da funcionalidade é feita recorrendo às funções genérica de ficheiros apresentadas acima

CreateNamedPipeA

Creates an instance of a named pipe and returns a handle for subsequent pipe operations. A named pipe server process uses this function either to create the first instance of a specific named pipe and establish its basic attributes or to create a new instance of an existing named pipe.

https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createnamedpipea

```
HANDLE CreateNamedPipeA(
  LPCSTR                lpName,
  DWORD                 dwOpenMode,
  DWORD                 dwPipeMode,
  DWORD                 nMaxInstances,
  DWORD                 nOutBufferSize,
  DWORD                 nInBufferSize,
  DWORD                 nDefaultTimeOut,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

lpName

The unique pipe name. This string must have the following form:

\\.\pipe*pipename*

dwOpenMode

The open mode.

The function fails if *dwOpenMode* specifies anything other than 0 or the flags listed in the following tables.

This parameter must specify one of the following pipe access modes. The same mode must be specified for each instance of the pipe.

Mode	Meaning
PIPE_ACCESS_DUPLEX 0x00000003	The pipe is bi-directional; both server and client processes can read from and write to the pipe. This mode gives the server the equivalent of **GENERIC_READ** and **GENERIC_WRITE** access to the pipe. The client can specify **GENERIC_READ** or **GENERIC_WRITE**, or both, when it connects to the pipe using the CreateFile function.
PIPE_ACCESS_INBOUND 0x00000001	The flow of data in the pipe goes from client to server only. This mode gives the server the equivalent of **GENERIC_READ** access to the pipe. The client must specify **GENERIC_WRITE** access when connecting to the pipe. If the client must read pipe settings by calling the GetNamedPipeInfo or GetNamedPipeHandleState functions, the client must specify **GENERIC_WRITE** and **FILE_READ_ATTRIBUTES** access when connecting to the pipe.
PIPE_ACCESS_OUTBOUND 0x00000002	The flow of data in the pipe goes from server to client only. This mode gives the server the equivalent of **GENERIC_WRITE** access to the pipe. The client must specify **GENERIC_READ** access when connecting to the pipe. If the client must change pipe settings by calling the SetNamedPipeHandleState function, the client must specify **GENERIC_READ** and **FILE_WRITE_ATTRIBUTES** access when connecting to the pipe.

This parameter can also include one or more of the following flags, which enable the write-through and overlapped modes. These modes can be different for different instances of the same pipe.

Mode	Meaning
FILE_FLAG_FIRST_PIPE_INSTANCE 0x00080000	If you attempt to create multiple instances of a pipe with this flag, creation of the first instance succeeds, but creation of the next instance fails with **ERROR_ACCESS_DENIED**.
FILE_FLAG_WRITE_THROUGH 0x80000000	Write-through mode is enabled. This mode affects only write operations on byte-type pipes and, then, only when the client and server processes are on different computers. If this mode is enabled, functions writing to a named pipe do not return until the data written is transmitted across the network and is in the pipe's buffer on the remote computer. If this mode is not enabled, the system enhances the efficiency of network operations by buffering data until a minimum number of bytes accumulate or until a maximum time elapses.
FILE_FLAG_OVERLAPPED 0x40000000	Overlapped mode is enabled. If this mode is enabled, functions performing read, write, and connect operations that may take a significant time to be completed can return immediately. This mode enables the thread that started the operation to perform other operations while the time-consuming operation executes in the background. For example, in overlapped mode, a thread can handle simultaneous input and output (I/O) operations on multiple instances of a pipe or perform simultaneous read and write operations on the same pipe handle. If overlapped mode is not enabled, functions performing read, write, and connect operations on the pipe handle do not return until the operation is finished. The ReadFileEx and WriteFileEx functions can only be used with a pipe handle in overlapped mode. The ReadFile, WriteFile, ConnectNamedPipe, and TransactNamedPipe functions can execute either synchronously or as overlapped operations.

dwPipeMode

The pipe mode.

The function fails if *dwPipeMode* specifies anything other than 0 or the flags listed in the following tables.

One of the following type modes can be specified. The same type mode must be specified for each instance of the pipe.

Mode	Meaning
PIPE_TYPE_BYTE 0x00000000	Data is written to the pipe as a stream of bytes. This mode cannot be used with PIPE_READMODE_MESSAGE. The pipe does not distinguish bytes written during different write operations.
PIPE_TYPE_MESSAGE 0x00000004	Data is written to the pipe as a stream of messages. The pipe treats the bytes written during each write operation as a message unit. The GetLastError function returns **ERROR_MORE_DATA** when a message is not read completely. This mode can be used with either **PIPE_READMODE_MESSAGE** or **PIPE_READMODE_BYTE**.

One of the following read modes can be specified. Different instances of the same pipe can specify different read modes.

Mode	Meaning
PIPE_READMODE_BYTE 0x00000000	Data is read from the pipe as a stream of bytes. This mode can be used with either **PIPE_TYPE_MESSAGE** or **PIPE_TYPE_BYTE**.
PIPE_READMODE_MESSAGE 0x00000002	Data is read from the pipe as a stream of messages. This mode can be only used if **PIPE_TYPE_MESSAGE** is also specified.

One of the following wait modes can be specified. Different instances of the same pipe can specify different wait modes.

Mode	Meaning
PIPE_WAIT 0x00000000	Blocking mode is enabled. When the pipe handle is specified in the ReadFile, WriteFile, or ConnectNamedPipe function, the operations are not completed until there is data to read, all data is written, or a client is connected. Use of this mode can mean waiting indefinitely in some situations for a client process to perform an action.
PIPE_NOWAIT 0x00000001	Nonblocking mode is enabled. In this mode, ReadFile, WriteFile, and ConnectNamedPipe always return immediately. Note that nonblocking mode is supported for compatibility with Microsoft LAN Manager version 2.0 and should not be used to achieve asynchronous I/O with named pipes. For more information on asynchronous pipe I/O, see Synchronous and Overlapped Input and Output.

One of the following remote-client modes can be specified. Different instances of the same pipe can specify different remote-client modes.

Mode	Meaning
PIPE_ACCEPT_REMOTE_CLIENTS 0x00000000	Connections from remote clients can be accepted and checked against the security descriptor for the pipe.
PIPE_REJECT_REMOTE_CLIENTS 0x00000008	Connections from remote clients are automatically rejected.

nMaxInstances

The maximum number of instances that can be created for this pipe. The first instance of the pipe can specify this value; the same number must be specified for other instances of the pipe. Acceptable values are in the range 1 through **PIPE_UNLIMITED_INSTANCES** (255).

If this parameter is **PIPE_UNLIMITED_INSTANCES**, the number of pipe instances that can be created is limited only by the availability of system resources. If *nMaxInstances* is greater than **PIPE_UNLIMITED_INSTANCES**, the return value is **INVALID_HANDLE_VALUE** and GetLastError returns **ERROR_INVALID_PARAMETER**.

ConnectNamedPipe

Enables a named pipe server process to wait for a client process to connect to an instance of a named pipe. A client process connects by calling either the CreateFile or CallNamedPipe function.

https://docs.microsoft.com/en-us/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe

If hNamedPipe was opened with FILE_FLAG_OVERLAPPED, the lpOverlapped parameter must not be NULL. It must point to a valid OVERLAPPED structure. If hNamedPipe was opened with FILE_FLAG_OVERLAPPED and lpOverlapped is NULL, the function can incorrectly report that the connect operation is complete.
If hNamedPipe was created with FILE_FLAG_OVERLAPPED and lpOverlapped is not NULL, the OVERLAPPED structure should contain a handle to a manual-reset event object (which the server can create by using the CreateEvent function).
If hNamedPipe was not opened with FILE_FLAG_OVERLAPPED, the function does not return until a client is connected or an error occurs. Successful synchronous operations result in the function returning a nonzero value if a client connects after the function is called.

A named pipe server process can use ConnectNamedPipe with a newly created pipe instance. It can also be used with an instance that was previously connected to another client process; in this case, the server process must first call the DisconnectNamedPipe function to disconnect the handle from the previous client before the handle can be reconnected to a new client. Otherwise, ConnectNamedPipe returns zero, and GetLastError returns ERROR_NO_DATA if the previous client has closed its handle or ERROR_PIPE_CONNECTED if it has not closed its handle.

The behavior of ConnectNamedPipe depends on two conditions: whether the pipe handle's wait mode is set to blocking or nonblocking and whether the function is set to execute synchronously or in overlapped mode. A server initially specifies a pipe handle's wait mode in the CreateNamedPipe function, and it can be changed by using the SetNamedPipeHandleState function.
The server process can use any of the wait functions or SleepEx— to determine when the state of the event object is signaled, and it can then use the HasOverlappedIoCompleted macro to determine when the ConnectNamedPipe operation completes.

Windows 10, version 1709: Pipes are only supported within an app-container; ie, from one UWP process to another UWP process that's part of the same app. Also, named pipes must use the syntax "\\.\pipe\LOCAL\" for the pipe name.

```
BOOL ConnectNamedPipe(
  HANDLE       hNamedPipe,
  LPOVERLAPPED lpOverlapped
);
```

DisconnectNamedPipe

Disconnects the server end of a named pipe instance from a client process.

https://docs.microsoft.com/en-us/windows/win32/api/namedpipeapi/nf-namedpipeapi-disconnectnamedpipe

If the client end of the named pipe is open, the DisconnectNamedPipe function forces that end of the named pipe closed. The client receives an error the next time it attempts to access the pipe. A client that is forced off a pipe by DisconnectNamedPipe must still use the CloseHandle function to close its end of the pipe.
The pipe exists as long as a server or client process has an open handle to the pipe.
When the server process disconnects a pipe instance, any unread data in the pipe is discarded. Before disconnecting, the server can make sure data is not lost by calling the FlushFileBuffers function, which does not return until the client process has read all the data.
The server process must call DisconnectNamedPipe to disconnect a pipe handle from its previous client before the handle can be connected to another client by using the ConnectNamedPipe function.

```
BOOL DisconnectNamedPipe(
  HANDLE hNamedPipe
);
```

FlushFileBuffers

Flushes the buffers of a specified file and causes all buffered data to be written to a file.

Typically the WriteFile and WriteFileEx functions write data to an internal buffer that the operating system writes to a disk or communication pipe on a regular basis. The FlushFileBuffers function writes all the buffered information for a specified file to the device or pipe.
Due to disk caching interactions within the system, the FlushFileBuffers function can be inefficient when used after every write to a disk drive device when many writes are being performed separately. If an application is performing multiple writes to disk and also needs to ensure critical data is written to persistent media, the application should use unbuffered I/O instead of frequently calling FlushFileBuffers. To open a file for unbuffered I/O, call the CreateFile function with the FILE_FLAG_NO_BUFFERING and FILE_FLAG_WRITE_THROUGH flags. This prevents the file contents from being cached and flushes the metadata to disk with each write. For more information, see CreateFile.

https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-flushfilebuffers

```
BOOL FlushFileBuffers(
  HANDLE hFile
);
```

SetNamedPipeHandleState

Sets the read mode and the blocking mode of the specified named pipe. If the specified handle is to the client end of a named pipe and if the named pipe server process is on a remote computer, the function can also be used to control local buffering.

https://docs.microsoft.com/en-us/windows/win32/api/namedpipeapi/nf-namedpipeapi-setnamedpipehandlestate

```
BOOL SetNamedPipeHandleState(
  HANDLE  hNamedPipe,
  LPDWORD lpMode,
  LPDWORD lpMaxCollectionCount,
  LPDWORD lpCollectDataTimeout
);
```

lpMode

The new pipe mode. The mode is a combination of a read-mode flag and a wait-mode flag. This parameter can be **NULL** if the mode is not being set. Specify one of the following modes.

Mode	Meaning
PIPE_READMODE_BYTE 0x00000000	Data is read from the pipe as a stream of bytes. This mode is the default if no read-mode flag is specified.
PIPE_READMODE_MESSAGE 0x00000002	Data is read from the pipe as a stream of messages. The function fails if this flag is specified for a byte-type pipe.

One of the following wait modes can be specified.

Mode	Meaning
PIPE_WAIT 0x00000000	Blocking mode is enabled. This mode is the default if no wait-mode flag is specified. When a blocking mode pipe handle is specified in the ReadFile, WriteFile, or ConnectNamedPipe function, operations are not finished until there is data to read, all data is written, or a client is connected. Use of this mode can mean waiting indefinitely in some situations for a client process to perform an action.
PIPE_NOWAIT 0x00000001	Nonblocking mode is enabled. In this mode, ReadFile, WriteFile, and ConnectNamedPipe always return immediately. Note that nonblocking mode is supported for compatibility with Microsoft LAN Manager version 2.0 and should not be used to achieve asynchronous input and output (I/O) with named pipes.