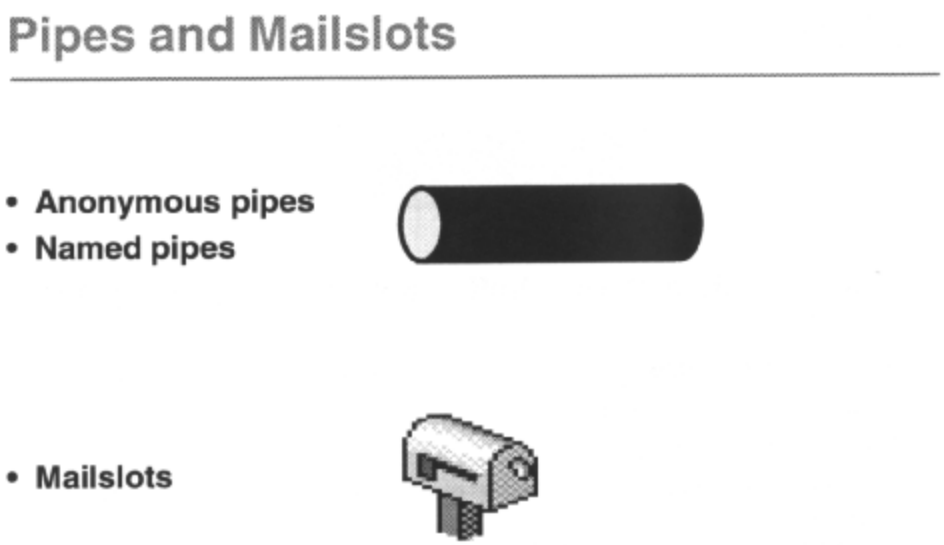


In this chapter, we look at how to use *anonymous pipes*, *named pipes* and *mailslots* to pass data between processes.

The client/server model is used to build distributed applications. A client application request service is provided by a server application. The client and server require a protocol which must be implemented by both before a service may be rendered or accepted.

A symmetric protocol is where either side may play the master or slave roles. An asymmetric protocol is where one side is immutably recognized as the master, with the other as the slave.

No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a “client process” and a “server process’.

Named pipes and mailslots are designed around the client/server model of IPC. They both represent a named endpoint of a communications link between processes. Named pipes provide bi-directional virtual circuit connections, and mailslots provide uni-directional datagram-style communication. Both can be used across the network.

When using named pipes and mailslots, there is an asymmetry in establishing communication between the client and server so that each has a different role. A server application normally listens, using a wellknown name, for service requests, remaining dormant until a connection is requested by a client’s connection using the server name. At such a time the server process wakes up and services the client, performing whatever appropriate actions the client requests of it.

Objectives

By the time you have completed this chapter, you should be able to:

- Create and use an anonymous pipe to pass data in one direction between related processes on the same machine.
- Create and use named pipes to pass data in both directions between processes that need not be related, possibly across a network.
- Create and use mailslots to pass data in one direction between processes that need not be related, possibly across a network.

A Pipe

- **A pipe is a data storage buffer maintained by the system in memory**
- **Processes access pipes using standard file I/O API**
  - **E.g.** `WriteFile()` **and** `ReadFile()`
- **Two types of pipe:**
  - **Anonymous pipes are unidirectional and are used between related processes**
  - **Named pipes are bidirectional, more flexible and may be used across a network**

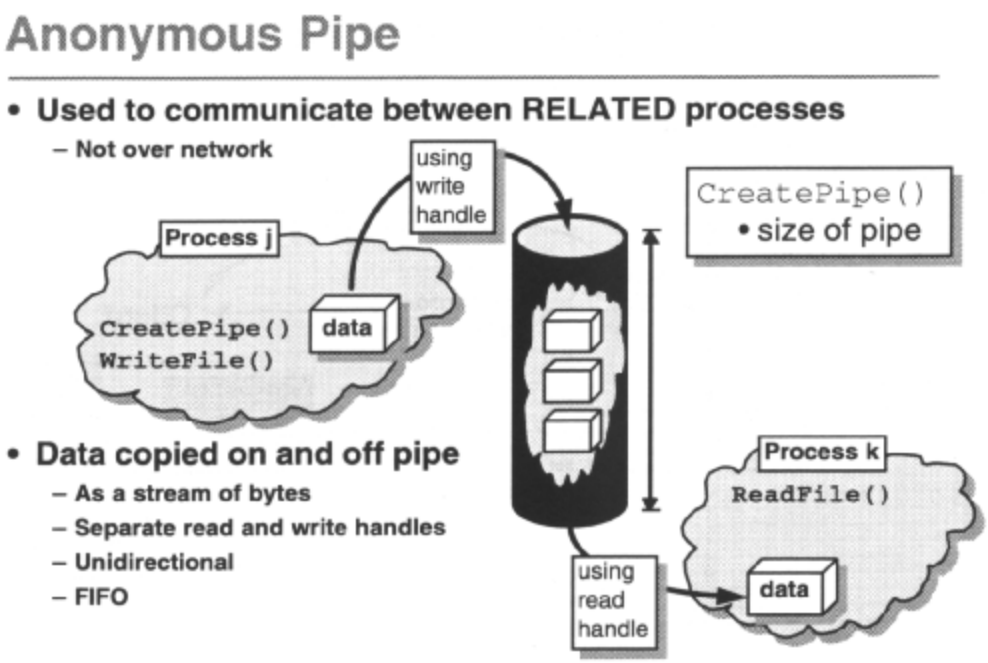
A pipe is a data storage buffer maintained in memory by the system, which can be used to pass a continuous stream of data between processes. Pipes are treated by the operating system as pseudo files.

Processes access a pipe using pseudo file handles, as though it were a file. Data is written to and read from the pipe using the standard Win32 or C-Runtime file I/O API calls. Because pipes are pseudo files, some of the standard I/O functions will behave slightly differently between pipes and real files. The SDK documentation is an exact reference.

Data is added to the pipe as it is written in by one process, and removed as it is read off the pipe ‘at the other end” by another process. Data in a pipe is accessed sequentially. Items are read off in the same order in which they were written, that is to say, in *“First In, First Out”* (HFO) order.

There are two types of pipe: anonymous and named pipes. Anonymous pipes are unidirectional and are used to pass data between related processes. Named pipes can be bi-directional and may be used across a network, between related or unrelated processes.

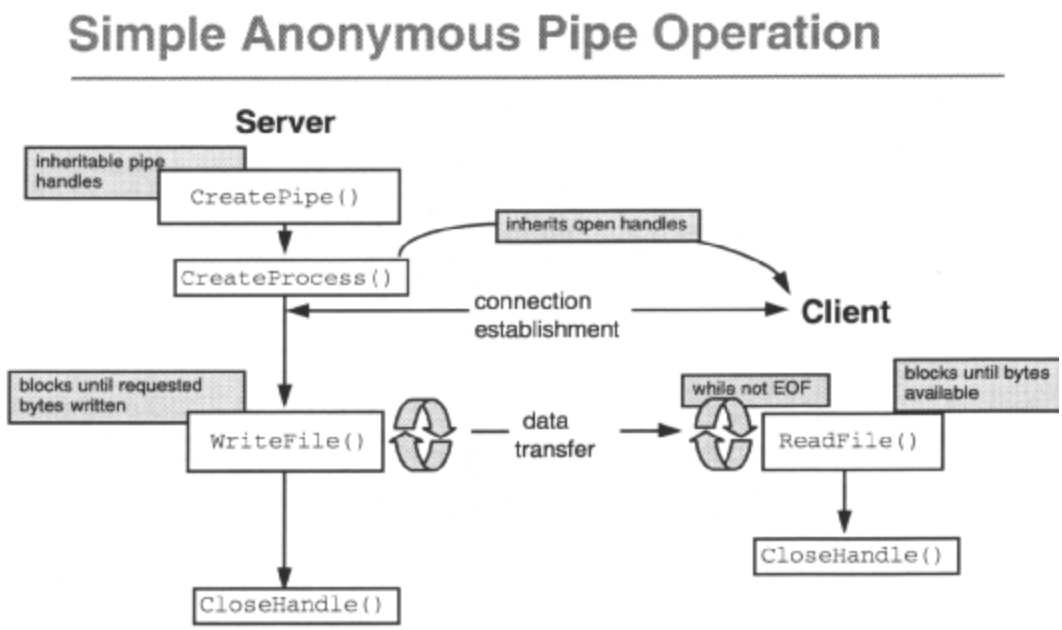
The pipe is a *‘non-seeking’* device, and trying to set the file pointer, via `SetFilePointer()`, on a pipe handle has no meaning or effect.



Anonymous pipes are so called because they have no universally understood name, but are identified by handles. One process creates the pipe and obtains a read handle (with read access to the pipe) and a write handle (with write access to the pipe). Another process must have one of these handles before it can communicate with the process that created the pipe.

Anonymous pipe handles can be inherited by related processes, and the pipe can then be used to communicate between these processes. This is normally how two processes communicate using an anonymous pipe.

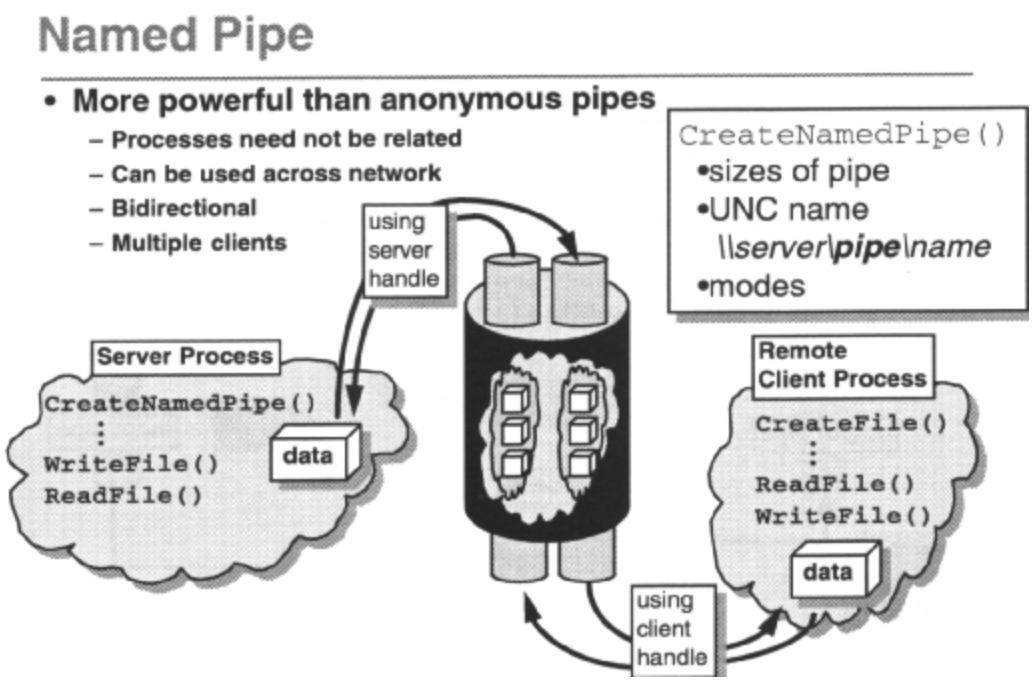
Because anonymous pipes have no universal name and are referenced using machine-specific pipe handles, they cannot be used over a network. They are also unidirectional. Data flow in both directions between two processes requires two different pipes.



Pipe handles are inheritable if `CreatePipe()` has the *'inherit both handles'* field of the security attribute parameter set to TRUE, and if `CreateProcess()` has the *'inherit open handles'* parameter set to TRUE. `DuplicateHandle()` can be used to duplicate inheritable duplicates of non-inheritable handles, and vice versa. `SetStdHandle()` is used to redirect the standard file handles, and `GetStdHandle()` is used to ascertain the standard file handles. Here is the theory:

1. Create inheritable input and output pipes using `CreatePipe()`.
2. Duplicate a non-inheritable copy of *stdout* using `DuplicateHandle()`.This is used to restore *stdout* later and is not inherited by the new process.
3. Duplicate a non-inheritable copy of the read handle of the output pipe using `DuplicateHandle()`. Close the inheritable copy of the read handle of the output pipe. This ensures that this process has a read handle to the output pipe that is not inherited by the new process.
4. Set *stdout* to be the inheritable write handle of the output pipe using `SetStdHandle()`. This will be inherited by the new process.
5. Duplicate a non-inheritable copy of *stdin* using `DuplicateHandle()`. This is used to restore *stdin* later and is not inherited by the new process.
6. Duplicate a non-inheritable copy of the write handle of the input pipe using `DuplicateHandle()`. Close the inheritable copy of the write handle of the input pipe. This ensures that this process has a write handle to the input pipe that is not inherited by the new process.
7. Set *stdin* to be the inheritable read handle of the input pipe using `SetStdHandle()`. This will be inherited by the new process.
9. Create child process using `CreateProcess()`, making sure open handles are inherited.
10. Reset *stdin* and *stdout* handles from duplicates; `SetStdHandle()`.

Step 6 is particularly important. We don't want the write end of the input pipe to be inherited by the child. The child will read from the input pipe until it gets EOF. This is sent when the last handle to the write end of the input pipe is closed. If the child inherits a write handle to the input pipe, then even if the parent closes the write handle of the input pipe, the child still has one. EOF will not be sent and the child will block indefinitely



Named pipes are much more powerful than anonymous pipes. The differences between named pipes and anonymous pipes are:

- Named pipes can be bi-directional or *duplex;* processes can use a named pipe both to receive and send information.
- Named pipes may be used to pass data between unrelated processes as well as related processes.
- Named pipes can be used between processes running on the same machine, or on different machines linked across a network.
- Named pipes can be used to pass a stream of bytes, or application-defined *messages.*
- Named pipes are used to connect servers to clients; a *server* process creates a named pipe instance and waits for a connection to it. Subject to security validation, any available named pipe instance can be connected to and used by any *client* process that knows its name.

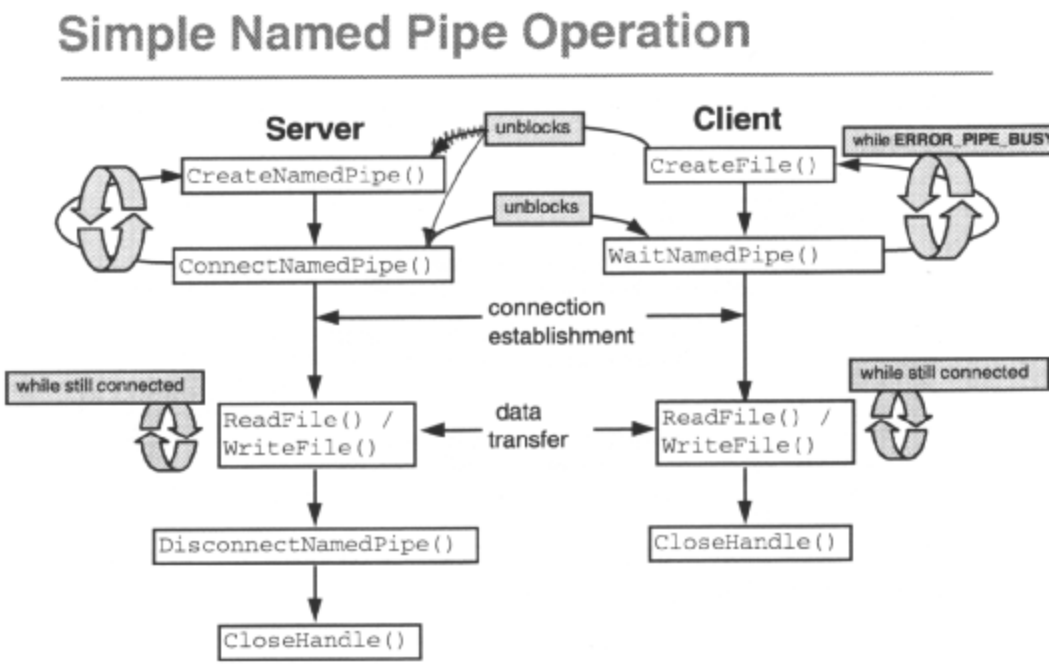
Named pipes provide a *virtual circuit* interface; a telephone style connection. Once a connection is established by a willing receiver, data may be sent and received over the connection channel. The destination for the data is set up when the connection is made, and doesn't need to be sent with each data block. The client and server communicate by using the file I/O APIs plus dedicated named pipe API functions.

A named pipe is a pseudo file (on Windows NT, named pipes have their own file system). When a pipe is created by the server or opened by the client, a handle is returned. The pipe is written to, and read from, using standard Win32 or C-Runtime file I/O functions with this handle. Unlike disk files, however, pipes are temporary, and when every handle to a pipe has been closed, the pipe and all the data it contains are deleted.

A server process can create multiple instances of the same named pipe, to meet the concurrent needs of multiple client processes. Note that Windows 95 has no support for named pipe servers, only named pipe clients.

A 16-bit client application (Windows 3.x, DOS or OS/2) may use a named pipe that was created previously by a Win32 server application.

On Windows NT, named pipe clients and servers may connect locally on the same machine, in which case networking doesn't need to be enabled.



The server process creates one or more instances of a pipe, using `CreateNamedPipe()` . A handle is returned and used in all subsequent API calls. The name is in the form *\\servername\\pipe\\pipename.* Servers use a server name of "." to indicate the local machine, as servers cannot create pipes on remote machines. Clients must specify the name of the server to open a pipe on a remote server or "." for a local pipe. The *pipename* part of the name must conform to the rules for Win32 filenames, but no actual file is created for the pipe.

`CreateNamedPipe()` also allows the specification of the pipe *type, read mode, access modes* and some other control attributes. Some of these can be changed on a per-instance basis with `SetNamedPipeHandleState()` , but others are adopted by subsequent pipe instances and cannot be changed. These are all investigated in later slides.

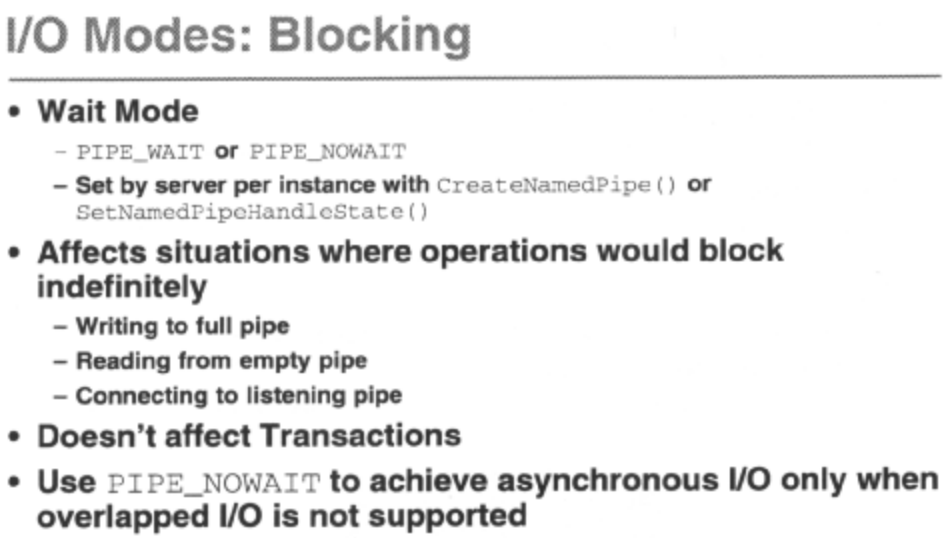
Having created an instance of the pipe, the server indicates that it is willing to receive incoming calls, using `ConnectNamedPipe()` . This call will block until a client process tries to initiate a connection with `CreateFile()`.

Subject to security validation, another client process connects to the other end of the pipe by name, using `CreateFile()` to open its handle to the pipe. If the pipe exists, but no instances of the pipe are available, `GetLastError()` returns ERROR_PIPE_BUSY and the client process can use `WaitNamedPipe()` to wait for one to become available, specifying a timeout period or waiting indefinitely. `WaitNamedPipe()` will eventually be successful when the server calls `ConnectNamedPipe()` , at which time the client can retry CreateFile() . The read/write access specified by a client using `CreateFile()` must be compatible with the access mode specified by the server when it was created.

Once connected, the server and client processes communicate by writing to and reading from the pipe using the Win32 or C-Runtime file I/O API or the dedicated named pipe API. Overlapped I/O is supported on a Windows NT system. The behavior of these functions depends on the type of the pipe, the read mode of the pipe handle, and on the I/O modes that are in effect for the pipe handle.

When the client and server have finished using the pipe, either the server can call DisconnectNamedPipe() to close the connection to the client process, or the client can call `CloseHandle()` . Either will invalidate the clients handle, and any unread data in the pipe is discarded. To ensure that all data written to the pipe has been read by the client, the server can first call `FlushFileBuffers()`. Once the client has been disconnected, the server can call `CloseHandle()` to close the pipe handle, or it can use `ConnectNamedPipe()` to listen and connect the instance to a new client.

The simplest server process would create a single instance of a pipe, connect to a single client, communicate with the client, disconnect the pipe, close the pipe handle and terminate. Typically, however, a server process will be created to communicate with multiple client processes. This could be done with a single pipe instance connecting to and disconnecting from each client in sequence, but to handle multiple clients simultaneously you would need to create multiple pipe instances.



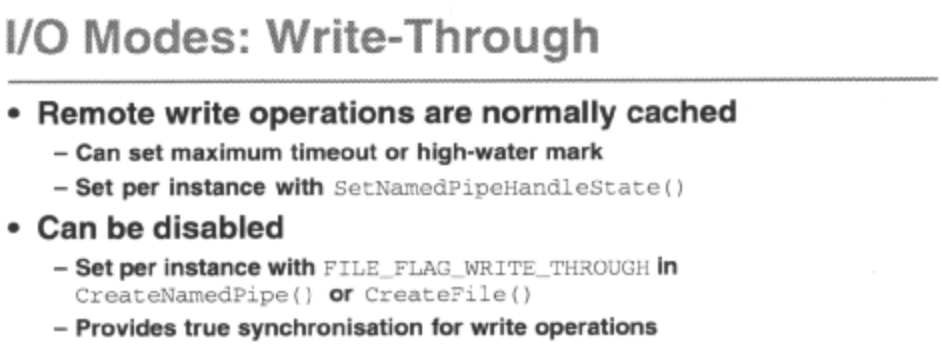
The ***'wait mode'*** of a named pipe handle determines what happens with pipe operations that would normally block indefinitely; this is only important when trying to read an empty pipe, write to a full pipe or connect to a listening pipe. The wait mode of a pipe handle can be set per instance by `CreateNamedPipe()` or `SetNamedPipeHandleState()`.

If the pipe is not empty, `ReadFile()` returns TRUE immediately, reading one or more bytes, and indicates the number of bytes read. This is true whatever the wait mode of the pipe handle. The wait mode of the pipe handle is important when the pipe is empty. If the pipe is empty and the wait mode of the pipe handle is PIPE_WAIT `ReadFile()` blocks and only returns when one or more bytes is available. If the pipe is empty and the wait mode of the pipe handle is PIPE_NOWAIT, `ReadFile()` returns FALSE immediately, with `GetLastError()` returning ERROR_NO_DATA.

If the pipe is not full (more bytes available than the number you want to write), `WriteFile()` returns TRUE immediately, writing the number of bytes specified. This is true whatever the wait mode of the pipe handle. The wait mode of the pipe handle is important when the pipe is full. If the pipe is full and the wait mode of the pipe handle is PIPE_WAIT `WriteFile()` blocks and only returns when the specified number of bytes is free on the pipe. If the pipe is full and the wait mode of the pipe handle is PIPE_NOWAIT `WriteFile()` returns TRUE immediately, not writing anything for a message-mode pipe, or writing as much as it can for a byte-mode pipe, indicating the number of bytes written.

A pipe-connection operation is affected by the wait mode of a pipe handle only when there is no client connected or waiting to connect to the pipe instance. If the wait mode of the pipe handle is PIPE_WAIT, `ConnectNamedPipe()` blocks and only returns when a client process connects to the pipe instance by calling either `CreateFile()` or `CallNamedPipe()` If the wait mode of the pipe handle is PIPE_NOWAIT, `ConnectNamedPipe()` returns FALSE immediately, with `GetLastError()` returning ERROR_PIPE_LISTENING.

The wait mode has no effect on transaction operations on the pipe. Note that non-blocking mode is supported for compatibility with LanMan 2.0. If overlapped I/O is supported, it should be used to achieve asynchronous I/O with named pipes. Note also the differences between reading from an empty pipe and writing to a full pipe, for byte-mode and message-mode pipes. By default, the wait mode of a named pipe is PIPE_WAIT.

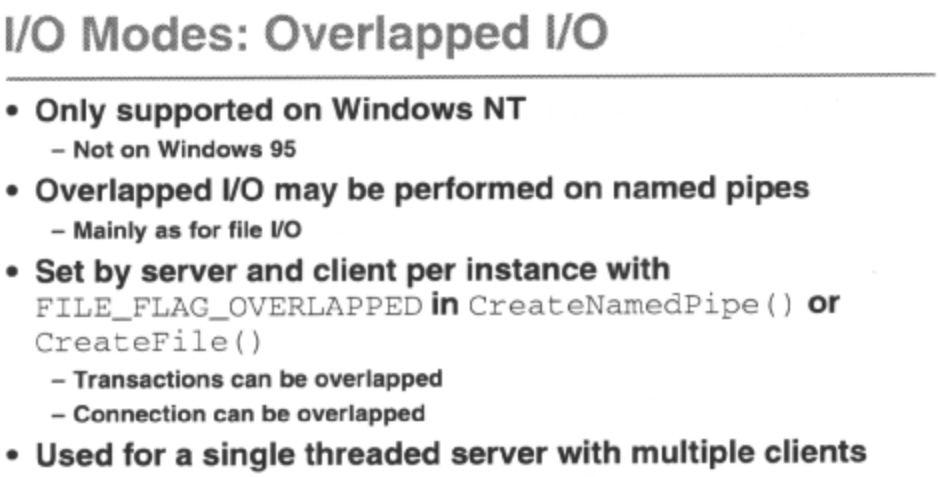


Normally, when data is written to a client or a server on a remote machine, a caching mechanism is enforced to enhance the efficiency of network operations. Data is buffered until a minimum number of bytes to be written have accumulated, or until a maximum time period has elapsed, allowing multiple writes to be combined into a single network transmission. In this way, a write operation may complete successfully when the data is in the outbound buffer cache, but before it has been transmitted across the network. This mode of operation is enabled or disabled by the *'write-through mode'* of the pipe.

At the client end of the pipe, the number of bytes and timeout period before transmission can be amended using the `SetNamedPipeHandleState()` function. However, this performance enhancement can be prevented altogether by specifying FILEFLAG_WRITE_THROUGH in the `CreateNamedPipe()` call when the pipe instance is created, or in `CreateFile()` when the client connects to the pipe. The write-through mode prevents transmission from being delayed, and ensures that the write operation will not complete until the data is in the pipe buffer on the remote machine. This *'write-through'* to the remote client is useful for applications that need true synchronization with every write operation.

By default, write through is disabled. The following code fragment creates a write-through mode pipe:

```
HANDLE hPipe = CreateNamedPipe('\\\\.\\pipe\\pipe, /*pipename*/
      FILEFLAG_WRITE_THROUGH
      | PIPE_ACCESS_DUPLEX, /*read/write access,write-through*/
      0,/* defaults */
      PIPE_UNLIMITED_INSTANCES, /* maximum simultaneous instances      */
      1024,                      /* output buffer size */
      1024,                      /* input buffer size */
      10000,           /* default timeout for WaitNamedPipe */
      NULL)            /* no security attr, not inheritable */
```



Refer to the *'File I/O'* chapter for a detailed discussion of asynchronous and extended I/O.

As mentioned in the aforementioned chapter, file **I/O** operations on Windows NT using `ReadFile()` and `WriteFile()` may be performed either synchronously or asynchronously, and extended file I/O using `ReadFileEx()` and `WriteFileEx()` may only be performed asynchronously.

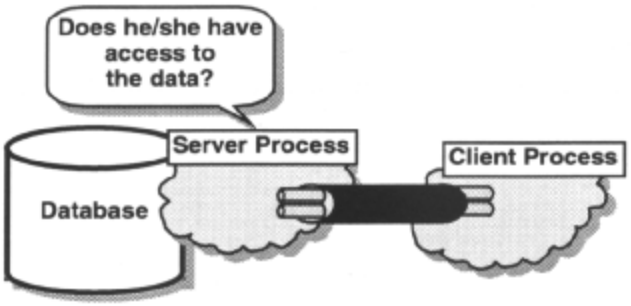
Asynchronous I/O is not supported in Windows 95.

On Windows NT, this is true when using the same routines with named pipes. Also, `TransactNamedPipe()` and `ConnectNamedPipe()` may be performed on a named pipe either synchronously or asynchronously. All these functions require an `OVERLAPPED` structure to be passed to them, specifying a handle to the Event object to be signaled when the operation is complete; the file pointer in this structure is ignored. To enable asynchronous (possibly overlapped) operations, the named pipe must be created and/or opened with the FILE_FLAG_OVERLAPPED flag set in the *'attributes'*. Any of the wait functions may be used to determine when an overlapped operation has completed.

Using overlapped I/O allows simultaneous operations to be performed on multiple files or pipes, or even to perform multiple operations simultaneously on the same pipe handle. This facility would be used in a single-threaded server handling communications with multiple clients.

Named Pipes and Security

- **Only supported on Windows NT**
  - **Not on Windows 95**
- `ImpersonateNamedPipeClient()`
  - **Server thread can assume security-access level of client**
  - **Ascertain whether client has access to privileged resources**
- `RevertToSelf()`
  - **Switch back to previous server-access level**

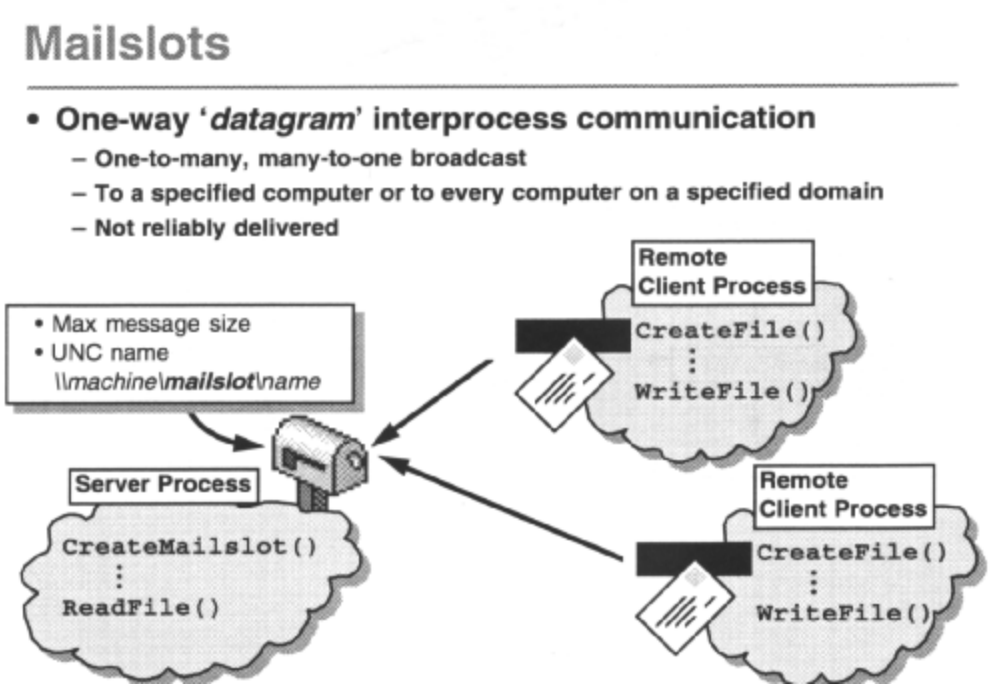


This discussion applies only to Windows NT, and not to Windows 95.

Like any other kernel-supplied, shareable object, any named pipe access is validated against the security profile for the pipe.

A server process can also assume the security token of a client process that is connected to a specified pipe instance using `ImpersonateNamedPipeC1ient()` Why would it do this?

Perhaps to determine whether the request of a client process should be granted. An example has previously been cited using of a named pipe server to provide access to a database to which the server process has privileged access. When a client makes a request to the server, the client will typically have a lower level of security access. By assuming the security token of the client, the server can attempt to access the protected database and the system will grant or deny the server's access, based on the security level of the client. When finished, the server uses the `RevertToSelf()` function to restore its original security token.



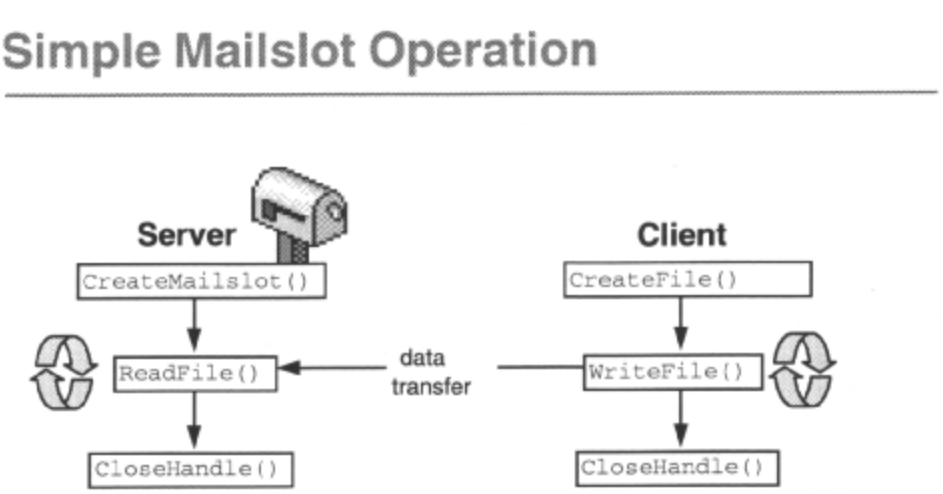
A *mailslot* is like a post office. One process can send messages, or *data grams,* to the *mailslot* and one process can take information from it. The *server* creates a *mailslot* and reads from it, the *client* writes to the mailslot. mailslots are connectionless; no one-to-one connection is ever made between client and server, as would be with named pipes, and 'one-off messages are sent between processes. By analogy with the national mail system, a datagram is rather like a letter; it contains an address and the data to be sent to that address. Furthermore, mailslots are second-class datagrams, whose reliable delivery is not guaranteed! Messages are free format.

Like a *named pipe*, a mailslot is a pseudo file (local mailslots have their own file system on Windows NT). When a mailslot is created by the server or opened by the client, a handle is returned. The mailslot is written to, and read from, by using standard Win32 or C-Runtime file **I/O** functions with this handle. Unlike disk files, however, mailslots are temporary, and when every handle to a mailslot has been closed, the mailslot and all the data it contains are deleted.

Mailslot messages can be sent over a network either to a specified computer or broadcast to every computer on a specified domain. In the latter case, if several processes within a domain each create a mailslot using the same name, every message that is addressed to that mailslot and sent to the domain is received by each participating process.

Unlike pipes, there is no built-in limit to the size of a mailslot buffer. The mailslot buffer is a FIFO buffer.

Note that mailslots cannot be created and read from on Windows 95, but a mailslot can be opened and written to.



The mailslot **API** is very much simpler than the named pipe API!

The server process creates a mailslot, using `CreateMailslot()` and specifies a name and other control attributes. A handle is returned, and is used in all subsequent API calls. The name is in the form *\\machinename\mailslot\mailslotname***.** Servers use a *machinename of* "." to indicate the local machine, as servers cannot create mailslots on remote machines. Clients have several options. For a local mailslot '.' is specified for the ***machinename.*** To write to a remote mailslot, the client specifies the ***machinename*** of a particular server, or the domain name of a group of servers. The ***mailslotname*** part of the name must conform to the rules for Win32 filenames, but no actual file is created for the mailslot.

Subject to security validation, another client process opens the mailslot by name, using `CreateFile()` with `GENERIC_WRITE` access mode and `FILE_SHARE_READ` share mode, to open its handle to the mailslot. The client processes can then communicate with the server by writing to the mailslot using the file I/O API, `WriteFile()`. The server can read from the mailslot using the file I/O API, `ReadFile()`. The exact behavior of the server read functions depends on the read timeout of the mailslot, set when the mailslot was created, or subsequently set with `SetMailslotInfo()`

When the client or server have finished using the mailslot, either can call `CloseHandle()` to invalidate their handle. The mailslot is deleted when all open handles to it are closed.

Summary

- A pipe is a FIFO data storage buffer to pass data between processes
  - Maintained by the system
- Anonymous pipes allow two related processes to communicate
- Named pipes provide bidirectional communication between unrelated processes
  - May be used across a network
  - Client-server model
- A mailslot is a one-way, unreliable interprocess communication
  - Normally used across a network
  - One-to many, many-to-one broadcast