Chapter 9

INTERPROCESS COMMUNICATION

Table of Contents:

9.1 INTRODUCTION 9-1

9.2 SHARED MEMORY 9-2
9.2.1 UNIX System V Shared Memory
9.2.2 Win32 Shared Memory 9-3

9.3 MESSAGES 9-4
9.3.1 UNIX System V Messages 9-5

9.4 PIPES 9-7
9.4.1 Anonymous Pipes 9-7
9.4.2 Named Pipes 9-11

9.5 SOCKETS 9-15

INTRODUCTION

Interprocess communication (IPC) includes thread synchorization and data exchange between threads beyond the process boundaries. If threads belong to the same process, they execute in the same address space, i.e. they can access global (static) data or heap directly, without the help of the operating system. However, if threads belong to different processes, they cannot access each others address spaces without the help of the operating system.

There are two fundamentally different approaches in IPC:

- processes are residing on the same computer
- processes are residing on different computers

The first case is easier to implement because processes can share memory either in the user space or in the system space. This is equally true for uniprocessors and multiprocessors.

In the second case the computers do not share physical memory, they are connected via I/O devices (for example serial communication or Ethernet). Therefore the processes residing in different computers can not use memory as a means for communication.

Most of this chapter is focused on IPC on a single computer system, including four general approaches:

- Shared memory
- Messages
- Pipes
- Sockets

The synchronization objects considered in the previous chapter normally work across the process boundaries (on a single computer system). There is one addition necessary however: the synchronization objects must be named. The handles are generally private to the process, while the object names, like file names, are global and known to all processes.

```
h = init_CS("xxx");
h = init_semaphore(20,"xxx");
h = init_event("xxx");
h = init_condition("xxx");
h = init_message_buffer(100,"xxx");
```

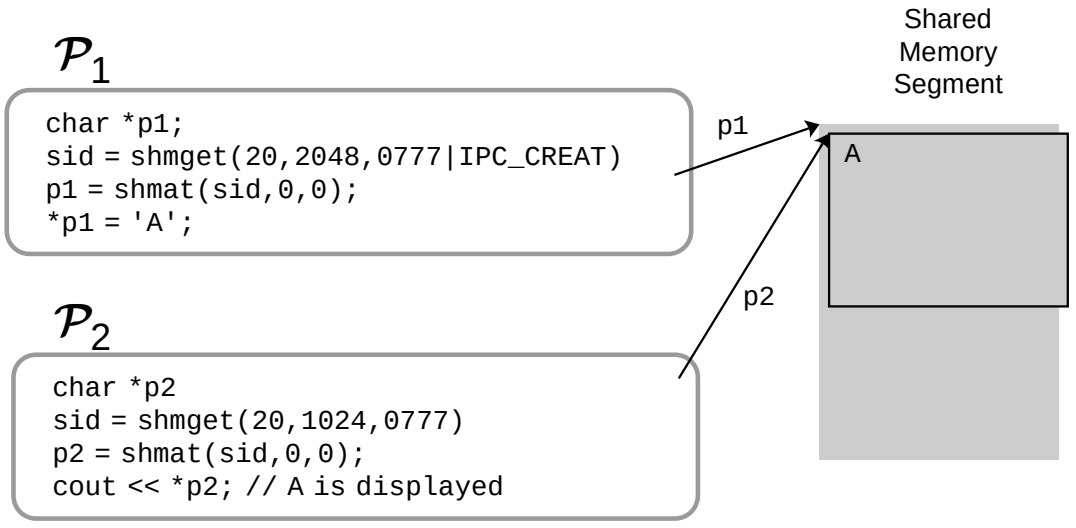
SHARED MEMORY

Two or several processes can map a segment of their virtual space into an identical segment of physical memory. Shared memory is the most efficient way of IPC, but it may require synchronization.

UNIX System V Shared Memory

System calls:

```
sid = shmget(key, size, flags); // Create shared memory segment (SMS)

     int sid ;       // Shared memory segment ID
     long key;       // Identification key  (instead of name)
     int size;       // Size of segment in bytes (0 for existing SMS)
     int flags;      // Access permission bits
                     // (can be OR-ed with IPC_CREAT)

p = shmat(sid, addr, flags);    // Attach SMS (get the SMS address)
     char *p;        // Actual SMS address
     void addr;      // (Recommended)desired address ( 0 for any)

sts = shmdt(p);      // Detach (unmap) SMS (doesn't destroy the SMS)

sts = shmctl(sid,cmd,sbuf);    // Control SMS

     int cmd;        // Command, IPC_RMID for destruction
```

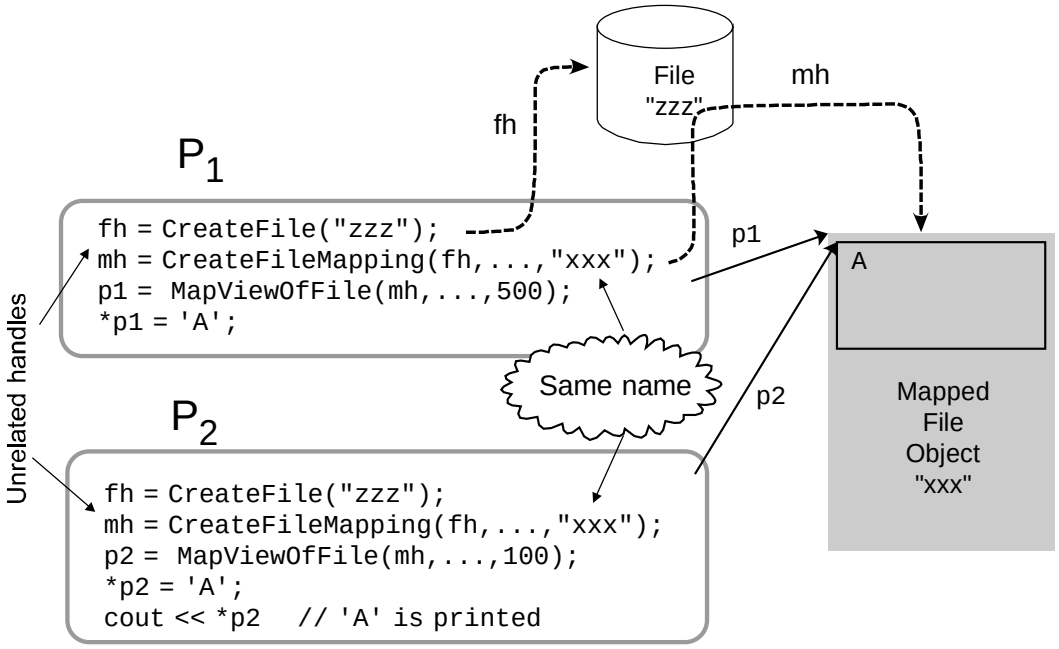


SHARED MEMORY (Cont.)

Win32 Shared Memory

In Windows 95 and Windows NT shared memory is implemented through file mapping

```
fh = CreateFile("fff",...);     // Create a file "fff"

     HANDLE fh;                 // File handle

                                // Create file-mapping object "xxx"
mh = CreateFileMapping(fh,flags,size,"xxx");

     HANDLE mh;                 // Handle of the file-mapping object
     int flags;                 // Access permission bits
     int size;                  // Size of the file-mapping object

                                // Map view of file into user space
p =  MapViewOfFile(mh,flags,size);

     char *p;                   // Address of mapped file in user space
```



If the file mapping is used only to implement shared memory, it is not necessary to associate a physical file with the shared memory. This can be achieved with a special handle (fh = 0xFFFFFFFF) which is used instead of the handle of an existing file.

Another way to share memory in Win95/NT/2000 is through global variables of dynamic link libraries. This is a less efficient way, because the access to shared memory is achieved through function calls.

MESSAGES

IPC via messages is essentially the same as messages discussed in the previous chapter: a data structure (message `m`) is copied from the space of the sender process into a message buffer in system space, then copied again from the buffer in system space to the structure in space of the receiving process.

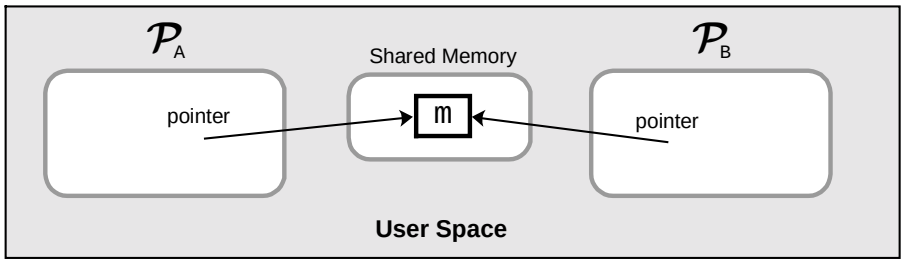
In order to make messages work accross the process boundary, the message buffers have to be named: each process which creates a message buffer has to refer to the same message buffer name (kernel will either create a new message object and return a handle, or will just return the handle if the message buffer has already been created by another process.) The handles returned by the kernel are local for each process.

The message transfer is normally synchronized, i.e. the receiving process is blocked if the message buffer is empty, and the sending process is blocked if the message buffer is full. Usually OS support nonblocking option too (see next page).

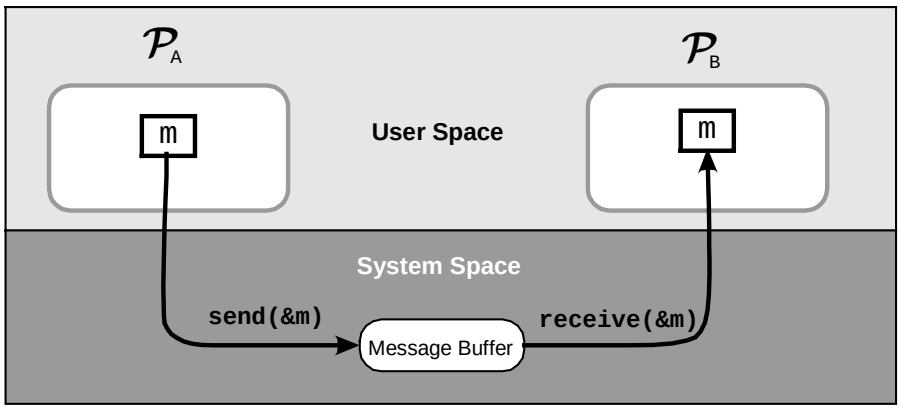
The messages can be of fixed or variable size. In the latter case, the message buffer is organized as a queue of message headers, while the space for the actual message is dynamically allocated.

Messages are less efficient than shared memory (require buffering and synchronization), but sometimes are more suitable due to the built-in synchronization.

Shared Memory Approach



Message Passing Approach



MESSAGES (Cont.)

UNIX System V Messages

Messages are originally implemented in UNIX System V as an additional means for IPC. (Windows 95/NT use named pipes, which are made even more powerful than System V messages.)

System calls:

```
mid = msgget(key, flags);            // Create a message buffer

      int mid;        // Message descriptor
      long key;       // Message buffer ID (used as name, both
                      // processes must use the same key number)
      int flags;      // Access permission bits
                      // (can be OR-ed with IPC_CREAT for a new mes.)
```

Each message queue is associated with the following information:

```
      typedef struct {
           Owners's and creator's user and group ID;
           Access modes;
           Usage sequence number;
           Key;
           Pointer to the first message on queue:
           Pointer to the last message on queue;
           Current number of bytes on queue;
           Number of messages on queue;
           Maximum number of bytes on queue;
           PID of last msgsnd();
           PID of last msgrcv();
           Time of last msgsnd();
           Time of last msgrcv();
           Time of last change of queue info;
      } queue;


msgctl(mid, cmd, &q)           // Operations on message queue

      int mid;        // Message descriptor
      int cmd;        // Command to be performed on the queue:
                      // IPC_STAT - Fetch mes. queue info into q
                      // IPC_SET  - Set some fields of the queue info
                      // IPC_RMID - Remove (destroy) the message queue
      queue q;        // Buffer to accept queue information
```

MESSAGES (Cont.)

The message format is as follows:

```
typedef struct {
     long type;      // Message type (a positive integer)
     char text[512]; // Message data
} message;
```

Messages are sent and received by the following system calls:

```
msgsnd(mid, &m, n, flags);           // Send a message

     int mid;        // Message descriptor returned by msgget()
     message m;      // Message buffer in sender's space
     int n;          // Number of bytes used in m.text[]
     int flags;      // Can specify IPC_NOWAIT: function returns an
                     // error message instead of blocking the caller

msgrcv(mid, &m, n, type, flags);     // Receive message

     int mid;        // Message descriptor returned by msgget()
message m;      // Message buffer in receiver's space
     int n;          // Number of bytes used in m.text[]
     int type;       // type = 0 - retrieve  first message from buf.
                     // type > 0 - retr. 1st mes. with m.type = type
                     // type < 0 - retr. 1st mess. with m.type < type
     int flags;      // Can specify IPC_NOWAIT: function returns an
                     // error message instead of blocking the caller
```

Both functions return 0 in case of success, or a negative error code at failure.

PIPES

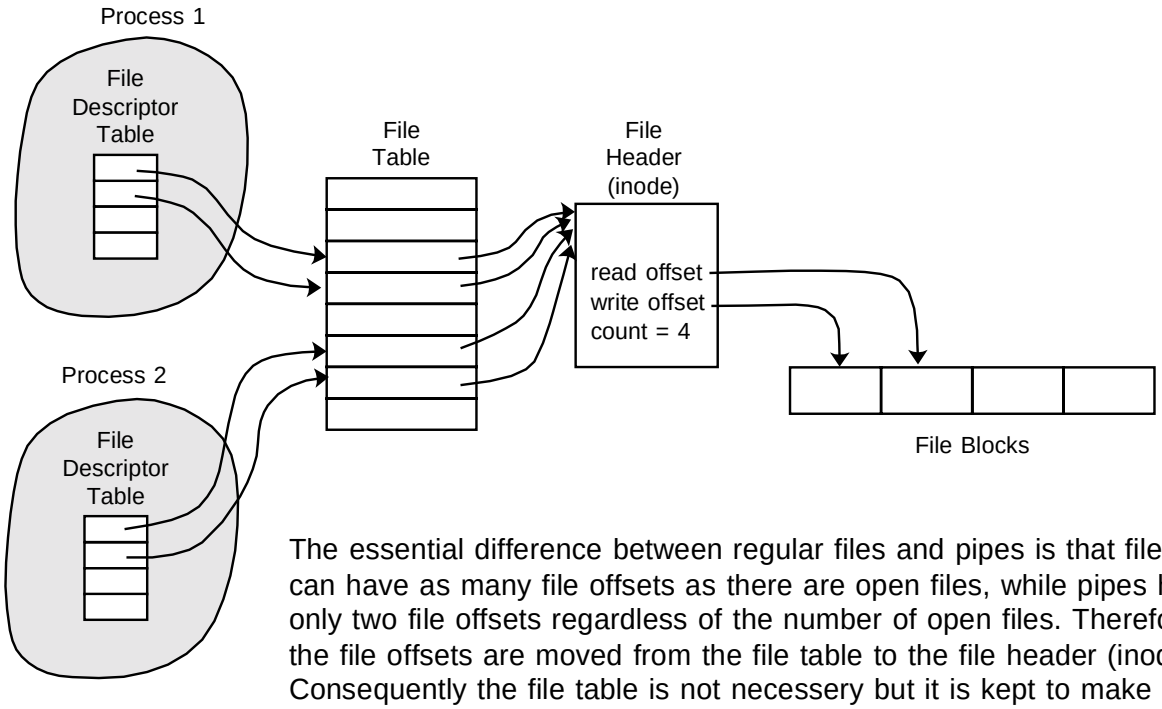
Pipes are originally used in UNIX and are made even more powerful in Windows 95/NT/2000.

Pipes are implemented in file system. Pipes are basically files with only two file offsets: one for reading another for writing. Writing to a pipe and reading from a pipe is strictly in FIFO manner. (Therefore pipes are also called FIFOs).

For efficiency, pipes are in-core files, i.e. they reside in memory instead on disk, as any other global data structure. Therefore pipes must be restricted in size, i.e. number of pipe blocks must be limited. (In UNIX the limitation is that pipes use only direct blocks.)

Since the pipes have a limited size and the FIFO access discipline, the reading and writing processes are synchronized in a similar manner as in case of message buffers. The access functions for pipes are the same as for files: `WriteFile()` and `ReadFile()`.

There are two types of pipes: anonymous (unnamed) pipes and named pipes.



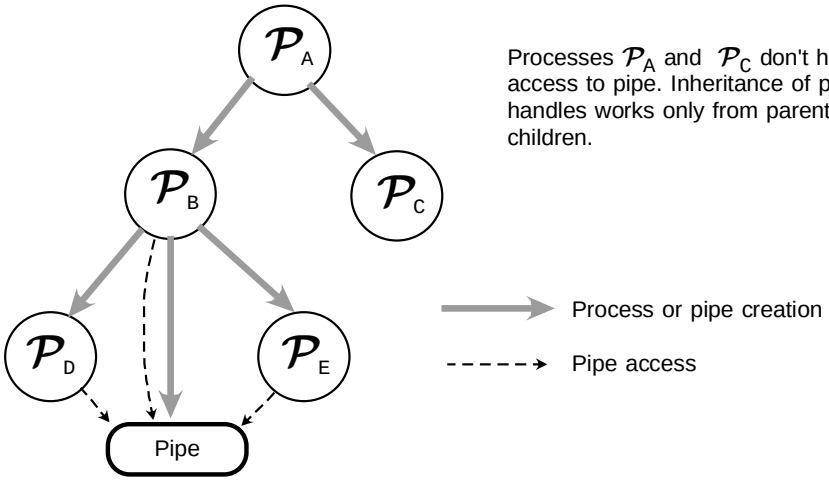
The essential difference between regular files and pipes is that files can have as many file offsets as there are open files, while pipes have only two file offsets regardless of the number of open files. Therefore, the file offsets are moved from the file table to the file header (inode). Consequently the file table is not necessery but it is kept to make file access consistent with all types of files.

PIPES (Cont.)

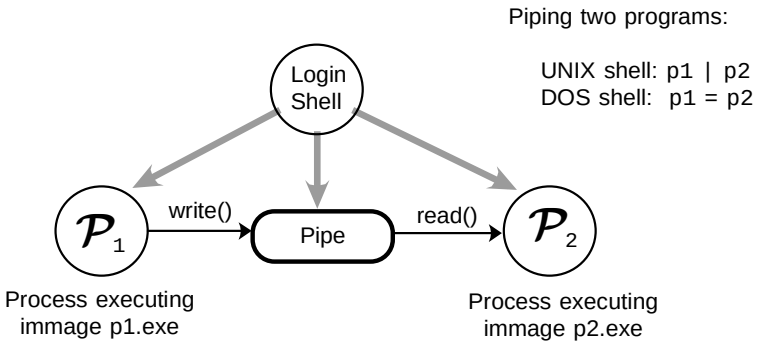
Anonymous Pipes

Anonymous pipes don't have names, therefore they can be used only between related processes which can inherit the file handles (file descriptors).

The inheritance of the file handles of anonymous pipes:



Anonymous pipes are typically used to "pipe" two programs: standard output from one pro gram is redirected to the pipe input (write handle), while the standard input of the second pro-gram is redirected to from the pipe output (read handle). The pipe is created by the parent (usually the login shell), and the pipe handles are passed to children through the inheritance mechanism (see example on the next page).



Anonymous pipes cannot be used across a network. Also, anonymous pipes are unidirectional - in order to communicate two related processes in both directions, two anonymous pipes must be created.

PIPES (Cont.)

Example of Win32 anonymous pipes used for program piping:

```
//*********************************************************************
//   This program implements piping of programs p1.exe and p2.exe
//   through an anonymous pipe. The program creates two child processes
//   (which execute images p1.exe and p2.exe) and a pipe, then passes
//   the pipe handles to the children.
//
//   The program is invoked as: pipe p1 p2  (no command line arguments)
//*********************************************************************

#include <windows.h>
#include <iostream.h>

int main(int argc, char *argv[])
{
                    // Create anonymous (unnamed) pipe
     SECURITY_ATTRIBUTES sa;
     sa.nLength = sizeof(SECURITY_ATTRIBUTES);
     sa.lpSecurityDescriptor = 0;
     sa.bInheritHandle = TRUE; // Handles are inheritable (default is FALSE)
     HANDLE rh,wh;             // Read and write handles of the pipe
     if(!CreatePipe(&rh,&wh,&sa,0))
     {
          cout << "Couldn't create pipe " << GetLastError()<< endl;
          return (1);
     }
                    // Create the first child process p1

     PROCESS_INFORMATION pi1;
     STARTUPINFO si1;
     GetStartupInfo(&si1);     // Get default startup structure
     si1.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
     si1.hStdOutput = wh;      // Std output of p1 is input to the pipe
     si1.dwFlags = STARTF_USESTDHANDLES;

     CreateProcess( argv[1],   // Name of the p1's image (without ".exe."
                    0,0,0,
                    TRUE,      // Each open inheritable handle of the
                               // parent will be inherited by the child
                    0,0,0,
                    &si1,&pi1);

     CloseHandle(wh);          // Pipe handle no longer needed
```

PIPES (Cont.)

Example of Win32 anonymous pipes used for program piping (continued):

```
                        // Create the second child process p2

     PROCESS_INFORMATION pi2;
     STARTUPINFO si2;
     GetStartupInfo(&si2);     // Get default startup structure
     si2.hStdInput = rh;       // Std input of p2 is otput from the pipe
     si2.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

     si2.dwFlags = STARTF_USESTDHANDLES;

     CreateProcess( 0,argv[2], // Name of the p1's image (without ".exe."
                    0,0,
                     TRUE,     // Each open inheritable handle of the
                               // parent will be inherited by the child
                    0,0,0,
                    &si2,&pi2);


     WaitForSingleObject(pi1.hProcess,INFINITE);
     CloseHandle(pi1.hProcess);

     WaitForSingleObject(pi2.hProcess,INFINITE);
     CloseHandle(pi2.hProcess);

     CloseHandle(rh);

     return(0);

}
```

Comment: In order to communicate two processes ($\mathcal{P}_1$ and $\mathcal{P}_2$) through anonymous pipes by redirecting the standard I/O, the processes don't have to be aware of the existence of pipes, i.e. their sources and images don't have to be modified.

PIPES (Cont.)

Named Pipes

Main properties of Win32 named pipes:

- Can be used to communicate unrelated processes
- The processes can be on different machines connected through the network
- They are message-oriented, i.e. can be used to send/receive messages (not only byte streams)
- They are full-duplex, i.e. a process can read and write to the same end of the pipe
- Named pipes can be instantiated, i.e. several independent instances of the same pipe can be created. (This means that a server can communicate independently with several clients.)

Named pipes also exist in UNIX (byte streams), with some restrictions:

UNIX pipes are half-duplex

UNIX pipes are limited to a single machine

UNIX pipes are byte-oriented

The implementation of the named pipes uses directory entry for each pipe (in addition to the file header). This makes pipes visible across the process and the machine boundaries.

PIPES (Cont.)

A pipe must be created first by the server. A typical system call:

```
HANDLE ph = CreateNamedPipe(
     name,                        // Pipe name (see below)
     PIPE_ACCESS_DUPLEX,          // Pipe open mode
     PIPE_TYPE_MESSAGE|           // Pipe operation mode
     PIPE_READMODE_MESSAGE|PIPE_WAIT,
     PIPE_UNLIMITED_INSTANCES,// Maximum number of instances
     0,                           // Output buffer size (use default)
     0,                           // Input buffer size (use default)
     0,                           // Time-out time (typically not used)
     0);                          // Security attributes (standard value)
```

The pipe name is a string with the following format:

\\.\pipe\[*pathname*]*pipename*

The "." stands for the local machine name.

After a pipe has been created, the creating process (server) must wait for client(s) to connect to the pipe:

```
ConnectNamedPipe(ph, (LPOVERLAPPED) NULL);
```

Client process connects to the named pipe by calling `CreateFile()`. After successful connection, the server can read or write into the pipe:

```
ReadFile(ph,                      // Pipe Handle
         &m,                      // Pointer to the message buffer
         sizeof(m),               // Size of the message buffer
         &n,                      // Number of bytes read
         (LPOVERLAPPED) NULL);
```

```
WriteFile(ph,                     // Pipe Handle
         &m,                      // Pointer to the message buffer
         sizeof(m),               // Size of the message buffer
         &n,                      // Number of bytes written
         (LPOVERLAPPED) NULL);
```

NOTICE: Reading and/or writing from/to the pipe is usually done through a special thread created by the server after each connection with a new client. This way the server can create a new instance of the pipe (with the same name) and wait for another client to connect, while the thread is communicating with the previously connected client. Such threads are called client threads. Each instance of the pipe will have its own pipe handle ph, which has to be passed to the client thread.

PIPES (Cont.)

After a pipe has been created at the server side, a client process can connect to the pipe by typically calling:

```
HANDLE ph = CreateFile(
     name,                // Pipe name (see below)
     GENERIC_READ|        // Access mode
     GENERIC_WRITE,
     0,                   // Share mode
     (LPSECURITY_ATTRIBUTES) NULL,
     OPEN_EXISTING,
     FILE_ATTRIBUTE_NORMAL
     (HANDLE)NULL);
```

Pipe name has the following format if the server and the client are on the same machine:

\\.\pipe\[*pathname*]*pipename*

Otherwise the format is:

machinename\pipe\[*pathname*]*pipename*

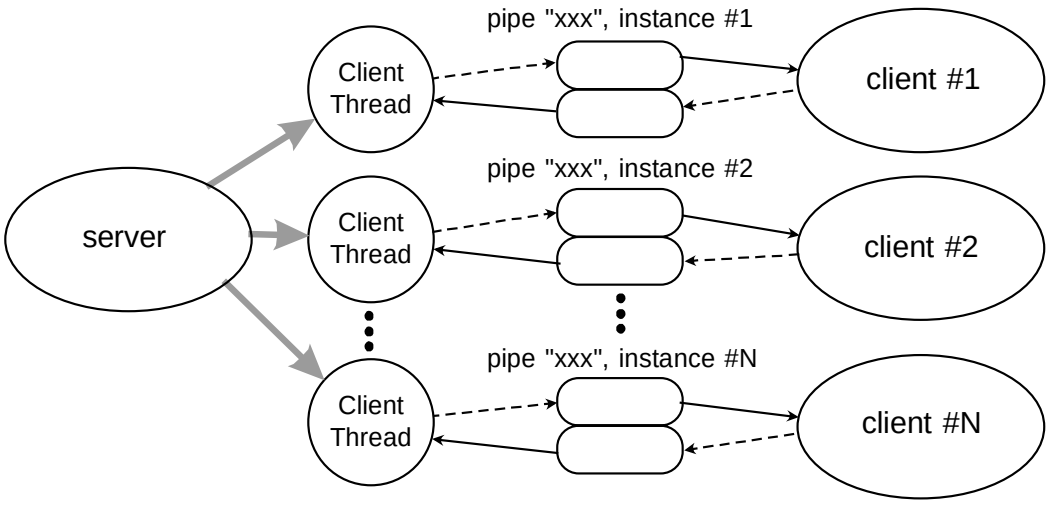
(pipename must be the same as the one specified in `CreateNamedPipe()`)

After successful connection to the pipe, the client can read or write from/to the pipe by using `ReadFile()` and `WriteFile()`.

PIPES (Cont.)

```
void main(){                                                    // SERVER
     HANDLE h;
     char* pipename = "\\\\.\\pipe\\xxx";
     while(1){
          h=CreateNamedPipe(pipename); // Create new pipe instance
          ConnectnamedPipe(h);         // Wait for client to connect
          CreateThread(ClientThread,h);
     }
}

void ClientThread(HANDLE h){
     while(1){
          ReadFile(h,&rm);     // Wait for request from client
          Use request message rm and crete an answer message am;
          WriteFile(h,&am);    // Send answer to the client
     }
```

```
void main(){                                                    // CLIENT
     HANDLE h;
     char *pipename = "\\\\medusa\\pipe\\xxx";
     h = CreateFile(pipename); // Connect to pipe

     while(1){
          WriteFile(h,&rm);    // Send to server the request message
          ReadFile(h,%am);     // Receive from server the answer
          process the answer;
     }
}
```

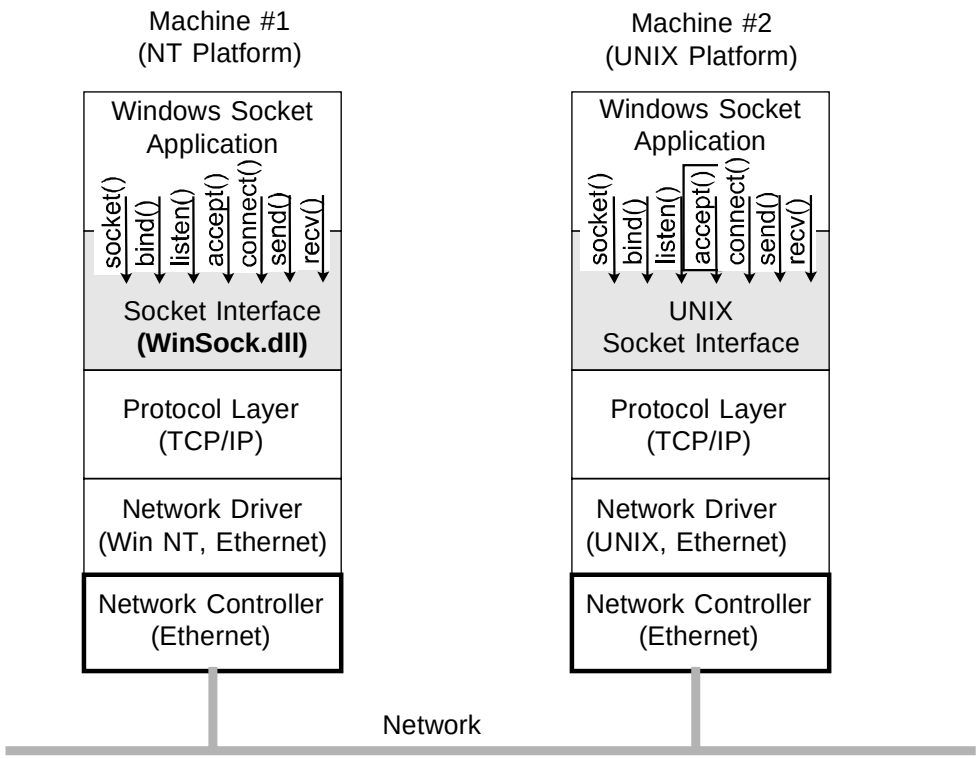


SOCKETS

Windows' named pipes provide synchronized communication between unrelated processes which even can execute on different machines connected through network. However, the machines must have the same operating system (i.e. Windows NT). In order to run applications across the network on heterogeneous platforms (for example Windows NT and UNIX), there are more general, network-supported mechanisms called Berkeley sockets and Remote Procedure Calls (RPC). This section will discuss sockets only, because they are more general, more efficient, and easier to set up than RPC.

Berkeley sockets were originally used in BSD 4.2 (1983) and are by now widely accepted in other operating systems.

Sockets are implemented in a layer of system software called socket layer, which sits on top of the protocol layer (protocol stack). Protocol layer together with the network driver (software) and network controller (hardware) provides networking (see next chapter). Socket layer in Windows 95/NT is implemented as a DLL library, WinSock.dll, which can work with different kinds of network protocols (like: DECnet, XNS (Xerox Network Systems), TCP/IP, IPX/SPX (Internet Packet Exchange, Novell Corp.), NetBEUI, AppleTalk, ISO/TP4)



TCP - Transmission Control Protocol
IP - Internet Protocol

SOCKETS (Cont.)

Sockets are typically used in server/client communication: server creates socket and binds it to a communication port and waits for connection request from a client (passive rolle - waits for client to initiate the connection). Client also creates a socket of the same type, then requests a connection to the known port on the known machine (active rolle - client initiates the connection). Once the server accepts the client's connection request, it will create another socket and bind it to another port, which will be used for further communication with the client. Once this is done, data can flow between the client's socket and the server's second socket. The server's first socket remains open, waiting requests from new clients. Waiting for new requests and communicating with clients with established connection is normally done through different threads (see diagram on the page 9-19).

Server socket calls:

```
SOCKET s = socket(          // Create socket
     AF_INET,               // Internet address family
     SOCK_STREAM,           // Connection-oriented (stream) socket type
                            // which uses TCP packets (reliable, duplex)
                            // Can also be: SOCK_DGRAM - uses UDP packets
                            // (see chapter 10)
     0);                    // Protocol (0 - system picks approp. protocol
                            // Returns socket descriptor (s)
```

Setting server's socket address:

```
SOCKADDR_IN sa;                 // Socket address structure
sa.sin_port = htons(44966);     // Supply the port number (identifies
                                // the application)
sa.sin_family = AF_INET;
sa.sin_addr.s_addr = htonl(INADDR_ANY);
```

Binding the socket to the port:

```
bind(                       // Associate the local address with the socket
     s,                     // Socket descriptor (returned by socket())
     &sa,                   // Pointer to the socket address structure
     sizeof(sa));           // Length of the address structure
```

Creating an empty request queue at the port:

```
listen(                     // Allow socket to take connections
     s,                     // Socket descriptor  (returned by socket())
     n);                    // Maximal number of requests allowed
```

SOCKETS (Cont.)

Creating a connection:

```
SOCKET ns = accept(          // Accept a connection from a client
     s,                      // Socket descriptor (returned by socket())
                             // (s identifies the socket which listens
                             // and accepts connection requests)
     &ca,                    // Socket address structure received from the
                             // connecting entity (client)
     &length);               // Length of ca

                             // Returns new socket descriptor (ns)
```

Accept() waits for connection requests. If such request gets into the port queue, it will be removed from the queue and new socket will be created along with a new port (with system chosen port number). The new socket will be bound to the new port. The new port will be used for further communication between the server and the client, i.e. the client's socket will be connected to the new port.

Communicating with the client:

```
int n = recv(                // Receive data from socket ns
     ns,                     // Connected socket
     &m,                     // Pointer to data buffer (char m[])
     sizeof(m),              // Size of data buffer
     NO_FLAGS_SET);          // Flags and options
                             // Returns number of bytes received (n)
                             // or SOCKET_ERROR

int n = send(                // Send data to socket ns
     ns,                     // Connected socket
     &m,                     // Pointer to data buffer (char m[])
     strlen(m)+1,            // Number of bytes sent (length of data in m[])
     NO_FLAGS_SET);          // Flags and options
                             // Returns number of bytes sent (n)
                             // or SOCKET_ERROR
```

Destroying a socket:

```
int m = closesocket(s);      // Release the socket descriptor s
                             // and delete all underlying structures
                             // (address info, queue and queued requests
                             // if any.)
                             // Returns 0 or SOCKET_ERROR
```

SOCKETS (Cont.)

Client also has to create a socket, which must be of the same type as the server's socket:

```
SOCKET cs = socket(        // Create socket
     AF_INET,              // Internet address family (
     SOCK_STREAM,          // Connection-oriented (stream) socket type
                           // which uses TCP packets (reliable, duplex)
                           // Can also be: SOCK_DGRAM - uses UDP packets
     0);                   // Protocol (0 - system picks approp. protocol
                           // Returns client's socket descriptor (cs)
```

After creating a socket client requests connection to the server:

```
int m = connect(           // Request connection with the server
     cs,                   // Client's socket (returned by socket())
     &sa,                  // Pointer to the server's socket address str.
     sizeof(sa));          // Length of the address structure
                           // Returns 0 (no error) or SOCKET_ERROR
```

The server's address structure:

```
SOCKADDR_IN sa;                // Socket address structure
sa.sin_port = htons(44966);    // Supply the port number
sa.sin_family = AF_INET;
long a = inet_addr("130.191.229.195"); // Destination IP address
memcpy(&sa.sin_addr,&a,sizeof(a));
```

After the connection is accepted by the server, the client can communicate with the server with:

```
send(cs, &m, strlen(m)+1, NO_FLAGS_SET);

recv(cs, &m, sizeofI(m), NO_FLAGS_SET);
```

SOCKETS (Cont.)