Win32 Interprocess Communication

- We will examine some of the low-level solutions for getting processes to communicate in the Win32 environment.

- Windows NT applications can send messages to each other, but passing pointers is pointless, because Windows NT runs processes in separate address spaces.

- A pointer that's valid in the context of one application is meaningless in the context of another. Unless you're using shared memory, letting process B use a pointer passed to it by process A will almost certainly trigger an access violation--or worse, to corrupt process B's data without realizing it. (The same holds true for Windows 95.)

Message Queues

- Win32 provides a simple method for moving data between two **threads** within the same process using a **message queue**.

- The **WM_COPYDATA** message was added to Windows NT so that applications could pass pointers across process boundaries and use them for transferring data.

- The **WM_COPYDATA** message is designed to move data between threads, regardless of whether the two threads are in the same process.

- Before sending a **WM_COPYDA**TA message, we initialize a **COPYDATASTRUCT** structure that contains data to be passed to another application, including a pointer to the block of data.

- We then use **SendMessage** to send the target application a **WM_COPYDATA** message with **wParam** holding the application's window handle and **lParam** holding the address of the COPYDATASTRUCT.

- The target application extracts the pointer from the structure and uses it to read the data. The operating system ensures that the pointer is valid in both processes.

- The following is a description of the **COPYDATASTRUCT**:

```
typedef struct tagCOPYDATASTRUCT {  // cds

   DWORD dwData;
   DWORD cbData;
   PVOID lpData;
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

- Where:

  dwData

  A user defined value that **optional** field, with which you can pass an **application-defined** 32-bit value. For example, you can specify an "action code" that describes what to do with the contents of **lpData**.

  cbData

  Specifies the size, in bytes, of the data pointed to by the **lpData** member.

  lpData

  Points to data to be passed to the receiving application. This member can be NULL.

- Suppose **pBuffer** is a pointer to a **block of bytes** that you want to send to another process, **nSize** is the **length** of the block in bytes, **hWnd** is the application's **window handle**, and **hWndTarget** is the handle of the **receiving process's main window.**

- The following code fragment creates a **COPYDATASTRUCT** structure, initializes the structure's **cbData** and **lpData** members with the block's size and address, and then sends a **WM_COPYDATA** message:

```
COPYDATASTRUCT cds;
cds.cbData = (DWORD) nSize;
cds.lpData = (PVOID) pBuffer;
SendMessage (hWndTarget, WM_COPYDATA, (WPARAM) hWnd, (LPARAM) &cds);
```

- When the target application's **WM_COPYDATA** handler receives the message, it can copy the pointer out of the **COPYDATASTRUCT** and use it as it would any other pointer, in the following manner.

```
PCOPYDATASTRUCT pcds =  (PCOPYDATASTRUCT) lParam;
PBYTE pBuffer =  (PBYTE) pcds->lpData;
```

- Note that the data in the receiving thread is owned by the system. The data block is temporary and will disappear as soon as the message has been processed.

- Therefore, if the receiving thread needs to modify the data or needs to store a permanent copy, then it must make its own local copy.

- One nuance of which to be aware when using WM_COPYDATA is that WM_COPYDATA messages must be sent (**SendMessage()**), not posted (**PostThreadMessage()**).

- This is because the system needs to manage the lifetime of the buffer used to move the data.

- If you use **PostThreadMessage(),** the buffer will be cleaned up and the system will free the memory it allocated in the target process's address space.

- One of the main disadvantages of this method is that it is very slow and therefore not an effective solution if performance is an issue. This is mainly due to the fact that all data passed has to be copied into the receiving process.

- Also note that the receiving thread must have a message queue and an associated window. If the thread does not have a window, you cannot use WM_COPYDATA.

- Lastly, SendMessage() is a synchronous call. The sender cannot continue until the receiver has processed the message. This forces the sender and receiver to be synchronized.

Anonymous Pipes

- An anonymous pipe is an **unnamed**, **unidirectional** pipe that transfers data between a parent process and a child process or between two child processes of **the same parent process**.
- Anonymous pipes are only used for point-to-point communication, and are most useful when one process creates another.
- Anonymous pipes don't have names. An application creates an anonymous pipe by calling **CreatePipe**.
- If another application wishes to connect to the anonymous pipe, the first application must pass it a pipe handle.
- Because Win32 pipe handles are process-relative (another way of saying that, like pointers, pipe handles are valid only in the context of a given process), one of the processes must create a copy of the handle that the client can use.
- This involves just enough extra work that most programmers prefer to use named pipes, unless the client process is a child process, which can obtain a pipe handle through inheritance.
- Since anonymous pipes cannot reach out across a network, applications that communicate using them must be running on the same PC.

- The **CreatePipe()** function creates an anonymous pipe and returns **two handles**, one to the **read end** and one to the **write end** of the pipe. The read handle has only read access to the pipe, and the write handle has only write access to the pipe.
- The function is defined as follows:

BOOL CreatePipe (PHANDLE hReadPipe, PHANDLE hWritePipe,
LPSECURITY_ATTRIBUTES lpPipeAttributes, DWORD nSize);

- where

hReadPipe

Points to the variable that receives the read handle for the pipe.

hWritePipe

Points to the variable that receives the write handle for the pipe.

lpPipeAttributes

Points to a **SECURITY_ATTRIBUTES** structure that specifies the security attributes for the pipe. The file system must support this parameter for it to have an effect.

If lpPipeAttributes is NULL, the pipe is created with a default security descriptor, and the resulting handle is not inherited.

nSize

Specifies the buffer size for the pipe. The size is only a suggestion; the system uses the value to calculate an appropriate buffering mechanism. If this parameter is zero, the system uses the default buffer size.

- Return Value

  If the function succeeds, the return value is TRUE.
  If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

- To **read** from the pipe, a process uses the **read handle** in a call to the **ReadFile** function. Unless an error occurs, ReadFile does not return until the specified number of bytes has been read or until all handles to the write end of the pipe have been closed.

- To **write** to the pipe, a process uses the write handle in a call to the **WriteFile** function. WriteFile does not return until the specified number of bytes has been written or an error occurs.

- If the pipe's buffer is full and bytes remain to be written, WriteFile does not return until some other process or thread reads from the pipe, making more buffer space available.

- **Asynchronous** (overlapped) read and write operations are not supported for anonymous pipes.
- The **lpOverLapped** parameter of the **ReadFile** and **WriteFile** functions is ignored when used with an anonymous pipe.

Named Pipes

- Named pipes have names, like filenames and they are much more flexible than anonymous pipes.

- A named pipe can be unidirectional or bidirectional and it will also work over a network. Unrelated processes can attach to it.

- Named pipe can beset up for overlapped (asynchrnous) operations.

- The **CreateNamedPipe** function creates a named pipe and returns a handle to it. Once a named pipe is created, a client process can connect to it by passing the pipe name to **CreateFile**.

- Named pipes may be used to connect two processes running on the same PC or one process running on a network server and another running on a network workstation.

- The basic syntax of the CreateNamedPipe function is as follows:

```
HANDLE CreateNamedPipe(

   LPCTSTR  lpName, // address of pipe name
   DWORD  dwOpenMode,    // pipe open mode
   DWORD  dwPipeMode,     // pipe-specific modes
   DWORD  nMaxInstances,  // maximum number of instances
   DWORD  nOutBufferSize, // output buffer size, in bytes
   DWORD  nInBufferSize,    // input buffer size, in bytes
   DWORD  nDefaultTimeOut,         // time-out time, in milliseconds
   LPSECURITY_ATTRIBUTES  lpSecurityAttributes          // address of security
attributes structure
  );
```

- The following code fragment the use of this function in a server program:

```
//
   // Create a named pipe.
   //
   m_hPipe = CreateNamedPipe (
         "\\\\.\\pipe\\ipcdemo",          // Pipe name
         PIPE_ACCESS_OUTBOUND,            // Write access only
         PIPE_TYPE_BYTE | PIPE_NOWAIT,    // Write bytes, no waiting
         1,                               // One instance at a time, please
         0,                               // Output buffer size (bytes)
         0,                               // Input buffer size (bytes)
         0,                               // Timeout value (milliseconds)
         NULL                             // Use default security descriptor
   );
   //

   // Check for failures.
   //
   if (m_hPipe == INVALID_HANDLE_VALUE)
   {
         MessageBox ("Unable to create a named pipe.", "Error",
               MB_OK | MB_ICONSTOP);
         return -1;
   }
   if (m_hPipe == NULL)
   {
         MessageBox ("Older Windows versions don't support named pipe servers. " \
               "This application must be run under at least Windows NT.", "Error",
               MB_OK | MB_ICONSTOP);
         return -1;
   }
```

- The following code fragment illustrates how a client would establish a connection to the named pipe:

```
……..
…….
LPTSTR lpszPipename = "\\\\.\\pipe\\ipcdemo";

/* Try to open a named pipe in read/write mode; wait for it, if necessary. */

while (1)
{
        hPipe = CreateFile (lpszPipename, GENERIC_READ | GENERIC_WRITE,
                0, NULL, OPEN_EXISTING, 0, NULL);

        /* Break if the pipe handle is valid. */

        if (hPipe != INVALID_HANDLE_VALUE)
        {
                MessageBox (strErrMsg, "Error", MB_OK |
                        MB_ICONEXCLAMATION);
                return;
        }
        ……
        ……
```

- The simplest server process can use the CreateNamedPipe function to 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 must communicate with multiple client processes. A server process can use a single pipe instance by connecting to and disconnecting from each client in sequence, but performance would be poor.

- To handle multiple clients simultaneously, the server process must create multiple pipe instances.

- Examples of a multithreaded server and a server using asynchronous operations are provided to you. These have been taken directly from the Windows API help system. Study them carefully.

Shared Memory

- A shared memory area is a region of memory that is explicitly designated to be visible to several processes at the same time.

- The first step is to create a file-mapping kernel object that specifies the size of the shared area using the **CreateFileMapping()** function.

- The next step is to map the shared area into your process's address space using the **MapViewOfFile()** function.

- The **CreateFileMapping** function creates a named or unnamed file-mapping object for the specified file.

```
HANDLE CreateFileMapping (

HANDLE  hFile,    // handle of file to map
LPSECURITY_ATTRIBUTES  lpFileMappingAttributes,  // optional security
                                                    attributes
DWORD  flProtect,          // protection for mapping object
DWORD  dwMaximumSizeHigh, // high-order 32 bits of object size
DWORD  dwMaximumSizeLow,  // low-order 32 bits of object size
LPCTSTR  lpName            // name of file-mapping object
);
```

- We now have a kernel object, but we still do not have pointer to memory that can be used.

- To obtain a pointer from the file-mapping kernel object, we use the **MapViewOfFile()** function that will map a view of a file into the address space of the calling process :

```
LPVOID MapViewOfFile(
   HANDLE  hFileMappingObject,          // file-mapping object to map into address
                                           space
   DWORD  dwDesiredAccess,    // access mode
   DWORD  dwFileOffsetHigh,    // high-order 32 bits of file offset
   DWORD  dwFileOffsetLow,     // low-order 32 bits of file offset
   DWORD  dwNumberOfBytesToMap   // number of bytes to map
);
```

- The following code fragment illustrates the use of the functions just described:

```
HANDLE hFilemapping;
LPDWORD pcounter;

hFilemapping = CreateFileMapping (
    (HANDLE) 0Xffffffff,        // File handle
    NULL,                       // Security attributes
    PAGE_READWRITE,             // Protection
    0,                          // Size - high 32 bits
    sizeof(DWORD),              // Size - low 32 bits
    "Server Thread ID");        // Name

pCounter = (LPDWORD) MapViewOfFile (
    hFilemapping,               // File mapping object
    FILE_MAP_ALL_ACCESS,        // Read/Write
    0,                          // Offset - high 32 bits
    0,                          // Offset - low 32 bits
    0);                         // map the whole thing
```

- The above code fragment creates a shared memory object that is large enough to hold a **DWORD**.

- A very useful application of this would be to store the ID of a server thread responsible for processing some sort of request.

- We would store the thread ID instead of a thread handle because handles are only meaningful in the context of their own process.

- So far have seen how to create a shared memory kernel object that other process can find by name and then map a view into their address space.

- Let us assume that the shared memory will be used in a client/server model. Only the server process should create and initialize the shared memory.

- All client processes will then use the **OpenFileMapping()** function to get a handle to the file-mapping kernel object already created:

```
HANDLE OpenFileMapping (
    DWORD  dwDesiredAccess,  // access mode
    BOOL  bInheritHandle,    // inherit flag
    LPCTSTR  lpName      // Points to a string that names the file-mapping object
                                to be opened
);
```

- Now all the thread has to do to access the shared memory is to use pointer operations such as **strcpy** to read data from the shared memory buffer or store data into the buffer.

- Note that you can also access the mapped file by means of the input and output (I/O) functions (ReadFile and WriteFile).

- The final step is to release the shared memory object by calling **UnmapViewOfFile()** with the pointer returned by **MapViewOfFile()** and calling **CloseHandle()** with the handle to the file-mapping kernel object:

```
BOOL UnmapViewOfFile (
   LPVOID  lpBaseAddress    // Pointer to a shared memory area. This must
                            match what was returned by MapViewOfFile().
);
```