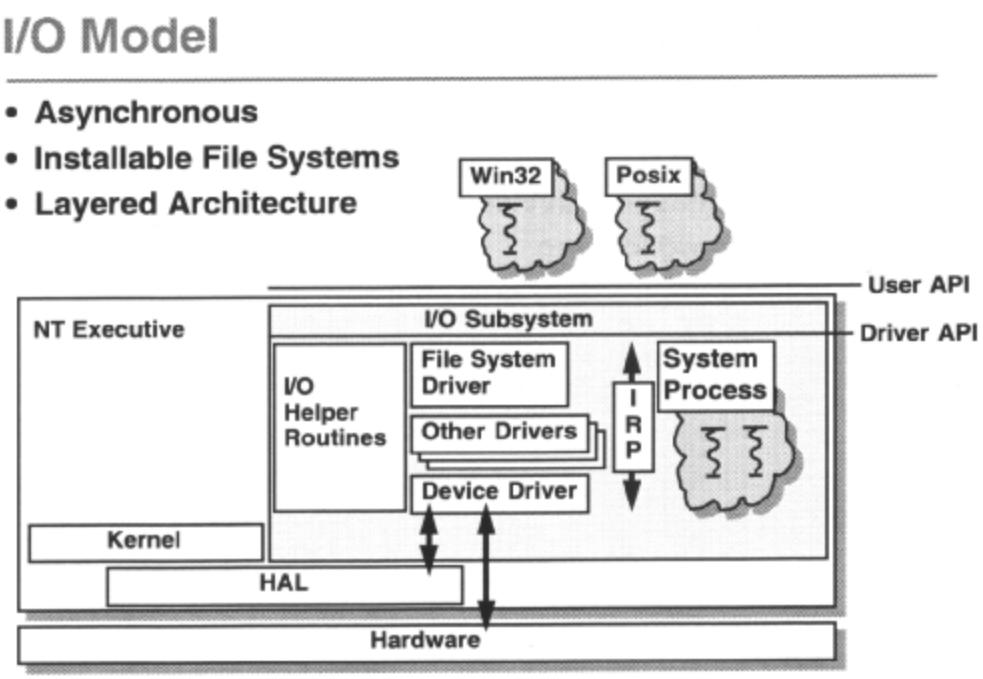


Input and output in Windows NT is done by the I/O system; a group of components responsible for processing input from, and passing output to, a variety of devices. The Design goals of the Windows NT I/O system were:

- Easy driver development; Well-defined interfaces allowing driver layering. Generic helper routines.
- Portability; I/O system and drivers written in C. Layering isolates platform changes.
- Security; Data from one process is protected against access or corruption by others.
- Multi-user support; multi-threads, multi-processors to implement asynchronous I/O.
- Installable file system support.
- Layered driver model.
- Object-oriented; all knowledge of a driver is confined to objects exposed to I/O system.
- Fast; Disk caching. Operations are streamlined in kernel mode.
- Multi-thread and multi-processor safe.
- Support for all envisaged subsystems.

NT environment subsystems use the facilities of the NT kernel I/O system via a standard system services interface. This implements the device-independent I/O facilities and establishes an asynchronous I/O model.

A File System Driver (FSD) represents each file system in NT. The I/O system communicates with each file system through a dynamic link interface, issuing logical requests for file access, which the file system translates into physical I/O requests for a particular device.

The low-level device drivers directly manipulate hardware and issue physical I/O requests to the device. NT allows dynamic loading and unloading of device drivers and file systems.

The I/O system uses an asynchronous I/O model but the I/O system services offer both asynchronous and synchronous I/O. Asynchronous I/O means that the caller makes an I/O request, and then gets notified later when the device has finished the data transfer. Because I/O devices are usually slower than processors, the caller can do other processing while waiting for I/O to complete.

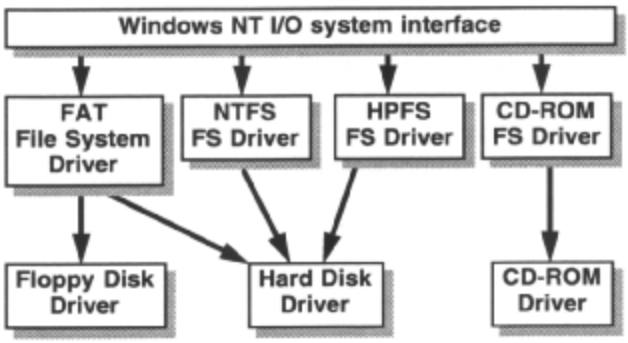
Asynchronous I/O is made possible because each file system also has associated with it a kernel process whose threads are able to perform I/O asynchronously from calling threads.

The drivers are modular with well-defined standard interfaces between them, which allow different file systems to use the same device drivers, and allow a layered hierarchy of drivers to implement intermediate processing. For instance, layered between the file system driver and the device driver could be a mirroring driver and/or a UPS driver and/or a disk-striping driver. I/O request packets (IRPJ) are passed up and down the layers; they contain a ‘stack area’ per driver layer, containing specific driver parameters, and a header accessible to all layers. Standard I/O ‘helper’ routines are available for each FSD and device driver and for the system services.

There is also a cache manager, which improves I/O performance by caching recently read disk information. It uses the NT virtual memory management to implement asynchronous read-ahead and write-behind.

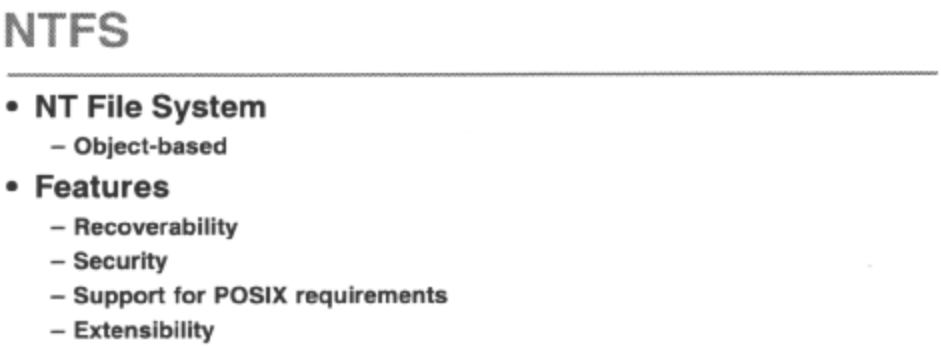
Installable File Systems

- **Multiple active file systems**
  - **FAT, HPFS, NTFS, CD-ROM, Named Pipes, Mailslots, Sockets, LAN Manager Redirector**
- **Automatic mounting and verification**



Because the file system for Windows NT, it is not ‘embedded’ in the operating system kernel, but is kept logically distinct, this allows Windows NT to support multiple Installable File Systems (IFS). The default file system is the File Allocation Table (FAT) file system inherited from DOS and does not need to be installed. Other installable file systems are OS/2 High Performance File System (HPFS), the new NT File System (NTFS) and a CD-ROM File System. These need to be installed at set-up time by specifying the desired file-system dynamic-link libraries and device drivers; the DRIVERS.INF file describes the driver to the setup procedure. Local named pipes and mail slots are referenced by names, which also exists in their own file system.

The first time a volume is accessed, the appropriate File System Driver DLL and device drivers are established and used for all subsequent I/O on that volume. This is all transparent to the calling application, which just uses the appropriate subsystem open/close/read/write type API, regardless of the file system type.



In addition to providing support for FAT and HPFS file systems, Windows NT has a new file system called Windows NT File System (NTFS). It's main features are that it has no artificial limitations, good recoverability and is secure.

Another file system was required for NT because of security and reliability requirements of the Windows NT system, plus new functionality is needed to meet the requirements of the POSIX subsystem. The goals of NTFS arc:

Recoverability - Full recoverability after system failure - checking a volume in seconds. This is achieved by logging file system changes to a log file which time-stamps changes to the disk structure. On reboot from a crash, the log file is replayed to make appropriate changes to the disk structure, keeping it consistent. This does not assure data integrity within files, but assures file system integrity.

Support for POSIX requirements - case-sensitive names, links; hard and symbolic (only hard-links are supported currently), sparse files.

Security - NTFS is object-based and like other objects in NT, files can have security attached to them. Access control lists, which specify users and their corresponding access permissions, can be attached to each file and directory. NTFS is the only secure file system.

Extensibility - Multiple data streams per-file, popular file server support; e.g. Novell, AppleShare, SunNFS

Removal of limitations - 64-bit file addressing - large storage media up to 2^64 bytes, Unicode for file and volume names, aiding localization of files, hardware sector size independence, MS-DOS name space and name generation

Utilities are provided to convert existing FAT and HPFS partitions to NTFS, but not in the other direction.

Creating and Opening Files

- `CreateFile()`
  - creates new or opens existing file by name
- **Other parameters**
  - read/write access and share options
  - open/create action
  - many attributes and flags
  - security descriptor
- **Returns handle**
  - valid until file closed; `CloseHandle()`
  - used for all API calls
  - can be inherited

Calling `CreateFile()` not only creates a new tile, or opens or truncates an existing one, it also specifies way in which the file will be accessed by other API functions. It accepts a name and returns a valid open handle if successful, used for all further access to the file, or INVALID_HANDLE_VALUE if not.

The *'access mode'* parameter specifies whether data is allowed to be read/written and the file pointer modified, and is a combination of the following. GENERIC_READ allows reading and GENERIC_WRITE allows writing.

The *'share mode'* parameter specifies how the file is shared if it is subsequently opened one or more times, and is a combination of the following; 0 means no sharing, FILE SHARE_READ means other open operations may be performed for read access and FILE_SHARE_WRITE means other open operations may be performed for write access.

The *'open or create mode'* parameter sets the action that `CreateFile()` should take if the file does or does not exist, and must be one of the following. CREATE_NEW creates a new file but fails if it already exists, CREATE_ALWAYS creates a new file and overwrite it if it already exists OPEN_EXISTING opens a file but fails if it does not exist, OPENALWAYS opens a file and creates a new one if it does not already exist, TRUNCATE_EXISTING opens a file of length 0 but fails if it does not exist (must have write access).

The *'attributes and flags'* parameter marks the file attribute and other interesting options. Attributes can be FILE_ATI'RIBUTE_NORMAL or any combination of FILE_ATTRIBUTE ARCHIVE, FILE_ATITRIBUTE_NORMAL,FILE_ATTRIBUTE_READONLY,FILE_ATTRIBUTE_HIDDEN, FILE_ATTRIBUTE_SYSTEM,FILE ATTRIBUTE_TEMPORARY.

Other flags, amongst many, can be combined with this to

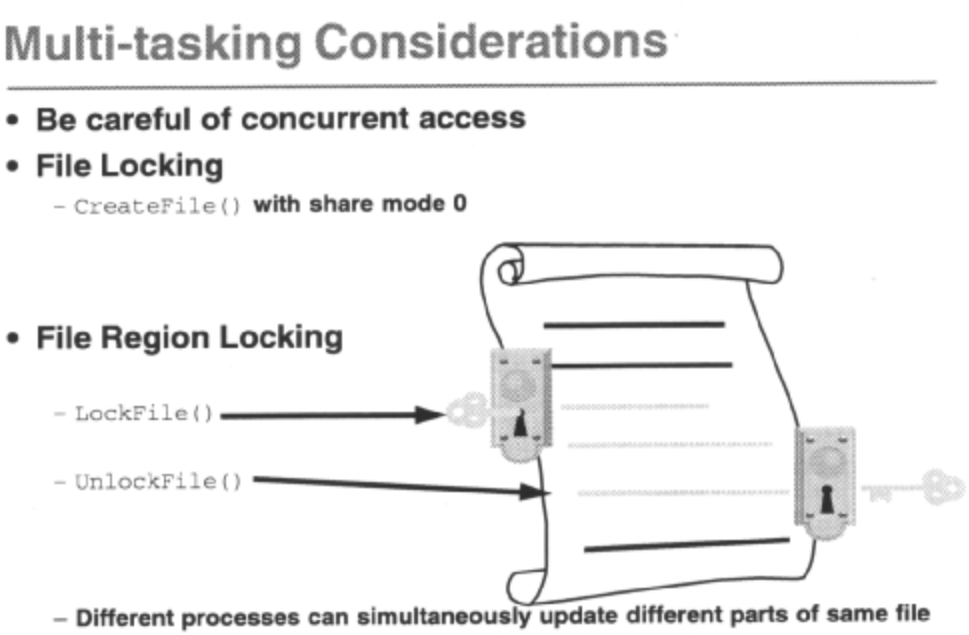
disabling caching and lazy-writing (FILE_FLAG_NO_BUFFERING, FILE_FLAG_WRITE_THROUGH)

allowing asynchronous I/O for lengthy operations (FILE_FLAG_OVERLAPPED)

specifying whether sequential/non-sequential access is required to help optimize caching (FILE FLAG_RANDOM_ACCESS, FILE_FLAG_SEQUENTIALSCAN)

specifying if the file is deleted when all open handles to it are closed
(FILE FLAG_DELETE_ON_CLOSE)

Note that the Windows 3.1 `OpenFile()` is still supported but `CreateFile()` is preferred.
`GetTempPath()` and GetTempFileName() are also still supported to aid the creation of unique *temporary* files in a multi-tasking environment.



A multi-threaded application must be careful when manipulating files, for example, setting a file pointer. Threads of the same process sharing a file handle and updating the file pointer must protect their access with a *Mutex* or *Critical Section*. Processes that have inherited handles to objects or duplicated them, have the handles mapped into their local handle table; the handles reference the same object and other processes see changes to the object made in one process via one handle. Threads of different processes using handles, which reference the same object, should protect their access with a Mutex, or file locks.

File locking is supported by the `CreateFile()` share modes, which enable one process to exclude all others while it accesses a file, or to allow read-only access by other processes.

File region locking permits a process to lock only that part of the file it is working with, excluding other processes from reading or writing, but giving them free access to the rest of the file. Any 2^64 range may be locked. Processes should obtain an open file handle with the access mode set to GENERIC_READ GENERIC_WRITE and share mode set to FILE_SHARE_READ | FILE_SHARE_WRITE. Each process can then call `LockFile()` just before it is about to read or write to the file, specifying the file handle, and the range to be locked. These ranges are expressed in terms of an offset from the start of the file, and the number of bytes to be locked or unlocked. A process can move its file pointer to the start of the record it has designated for locking with `SetFilePointer()`, and read or write data with `ReadFile()` and `WriteFile()`. Any other process attempting to perform a read or write which overlaps the locked area will fail. A process should unlock a record after use with a call to `UnlockFile()`, specifying the file handle, and the range to be unlocked. `CloseHandle()` also remove any file locks still in place. Locking beyond the end of a file is not an error, although overlapping locks is. File locks are not inherited.

Here is an alternative example of appending to the end of a shared file:

```
char  szBuf[50]
DWORD dwbytesWritten, dwPos;
HANDLE hFile = CreateFile(    “shared.cxc”,GENERIC_WRITE,
                              FILE_SHARE_READ | FILE_SHARE WRITE
                              ,NULL,OPEN_EXISTING,
                              FILE_ATTRIBUTE_NORMAL,NULL );
if ( hFile != INVALID_HANDLE_VALUE )
```

```
{
      dwPos = SetFilePointer( hFile, 0, NULL FILE_END );
      if ( LockFile( fFile, dwPos, CL, SQL, CL ) )
      {
      if (  WriteFile( hFile, szBuf, SQL, &dwBytesWritten, NULL ) &&
            dwBytesWritten SQL ) /*success*/
      {
      }
      UnlockFile( fFile, dwPos, CL, SQL, CL);
      }
      CloseHandle( hFile);
}
```

File Information

- **Attributes**
  - SetFileAttributes()
  - GetFileAttributes()
- **Type**
  - GetFileType()
  - **disk, character, pipe**
- **Size**
  - GetFileSize()
- **Path**
  - GetFullPathName()
- **Time**
  - **Create, last accessed, last modified time**
  - SetFileTime()
  - GetFileTime()
- **Version**
  - GetFileVersionInfo()
- **Security**
  - SetFileSecurity()
  - GetFileSecurity()

There are seven file attributes, six of which can be combined:

FILE_ATTRIBUTE_ARCHIVE	marked for backup or removal
FILE_ATTRIBUTE_NORMAL	normal file, can only be used on its own
FILE_ATTRIBUTE_READONLY	cannot be deleted or opened for writing
FILE_ATTRIBUTE_HIDDEN	cannot be displayed in normal directory listing
FILE_ATTRIBUTE_SYSTEM	excluded from normal directory search
FILE_ATTRIBUTE_DIRECTORY	directory
FILE_ATTRIBUTE_VOLUME_LABEL	volume label

File attributes can be set or queried using `SetFileAttributes()`/`GetFileAttributes()`. Obviously the directory attribute or the volume label attribute cannot be set.

Because there are different types of file systems that share `the` same file I/O API, e.g. named pipes, console I/O, then there is a `GetFileType()` function that returns the type based on an open handle. Returned types are pipe, disk, character or unknown.

File time stamps for last modification, last access and creation can be set and queried with `SetFileTime()`/`GetFileTime()`. The `CompareFileTime()` compares file times thus obtained. There is also an API for converting between a system time and a file time; `FileTimeToSystemTime()` and `SystemTimeToFileTime()`.

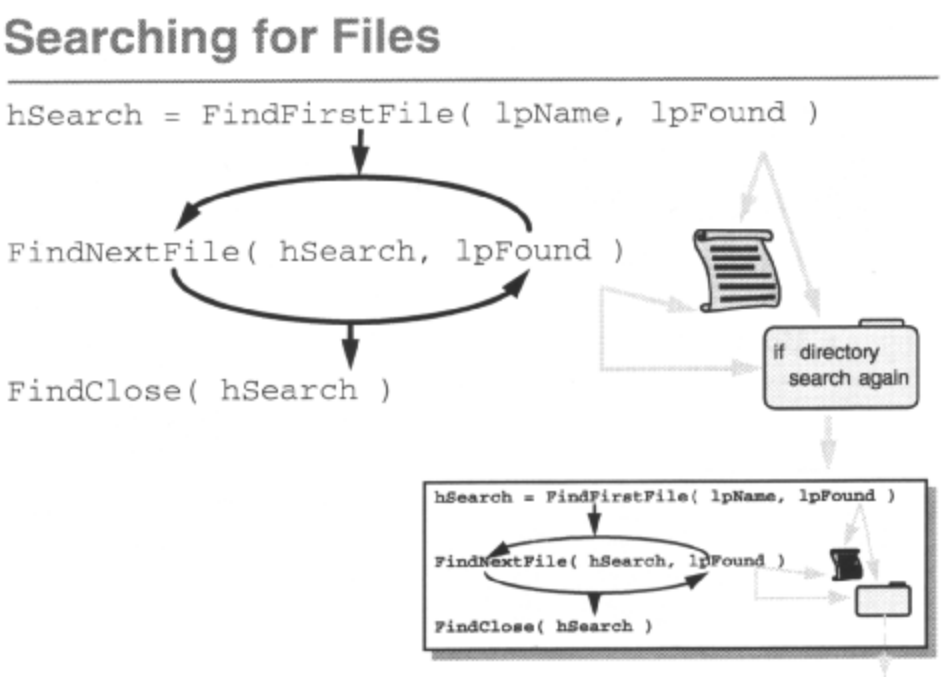
Version information can be retrieved using `GetFileVersionlnfo()` and then `VerQueryValue()` there is a whole version control API, `VerXXX()` in Win32. Version information is stored as a resource in an executable file, and so retrieving version information only makes sense for .EXEs or .DLLs.

The size of a file is retrieved using `GetFileSize()`. The size of a file is set by writing to it or calling `SetEndOfFile()`

`GetFullPathName()` retrieves the full path name for a file. Most of this information, and some extra, can be

obtained in one go with `GetFileInformationByHandle() which returns a` BY_HANDLE_FILE_INFORMATION structure.

Note that the Windows `3.1 GetFileResourceO, for` extracting resources from files, is no longer supported.

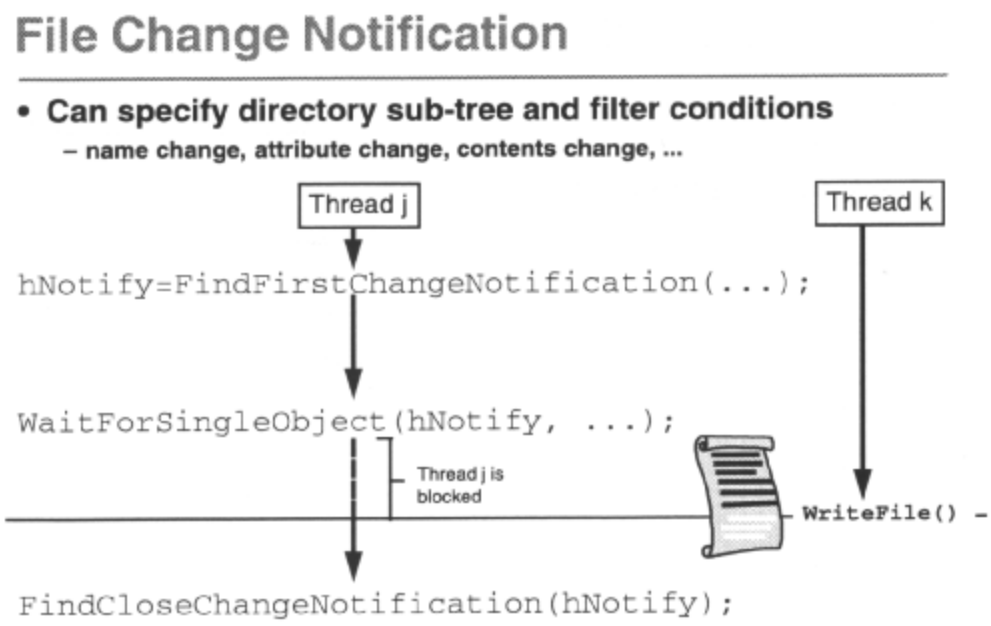


`FindFirstFile()` creates a 'file search' handle and establishes a file pattern to search for in the current directory. Note this can only be used for name-based searches, not attribute-based searches. It returns information about the first matching file, if any. Subsequent files can be found with `FindNextFile()`. When the search is over, the search handle is invalidated with `FindClose()`. An application can search for a single file on a specific path with
`SearchPath()`.

Here is an example of a recursive subdirectory search to find all archived *.txt files under the *\tmp* directory. Assume the existence of some utility functions.

```
Syntax: FindFiles( "\\tmp", *.tmp, FILE ATTRIBUTE_ARCHIVE );

void FindFiles(LPSTR lpStartDir, LPSTR lpFileSpec,DWORD dwAttrib )
W1N32_FIND_DATA fd;
HANDLE      hSearch;
char  szCurrDir[256];

GetCurrentDirectory(sizeof(szCurrDir) ,szCurrDir);
SetCurrentDirectory(lpStartDir);
hSearch=FindFirstFile(*,&fd);
if (hSearch != INVALID_HANDLE VALUE) {
      if (fd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY && fd.cFileName
[0] != 0)
      LPSTR lpNewDir = lpRecurseDirName(lpStartDir, fd.cFileName);
      if (lpNewDir) {
            FindFiles(lpNewDir,lpFileSpec,dWAtitrib)
            free (lpNewDir)
      }

      while (FindNextFile (hSearch, &fd) )
```

```
            if (fd.dwFileAtitributes & FILE_ATTRIBUTE_DIRECTORY &&
      fd.cFileName[0] != '.')

      LPSTR lpNewDir = lpRecurseDirName (lpStartDir, fd.cFileName);
      if (lpNewDir) {
            FindFiles (lpNewDir, ipFileSpec, dwAttrib);
            free(lpNewDir)
      }
      if (bNemeMarchFound(&fd,lpFileSpec) && dWAttrib &
      fd.dWFileAttributes) printf('found %s\\%s\n",lpSrartDir,fd.cFileName)

      FindClose(hSearch);
SetCurrentDirectory(szCurrDir);
}
```



`FindFirstChangeNotification()` creates a 'file change notification handle for directory or subtree based on filter values. The handle references a waitable object and can be used with `WaitForSingleObject()` or `WaitForMultiple0bjects()`.
A wait on this handle is satisfied (the handle is signaled) when a change matching the filter condition is met in the specified directory or subtree. Filter values can be a combination of:

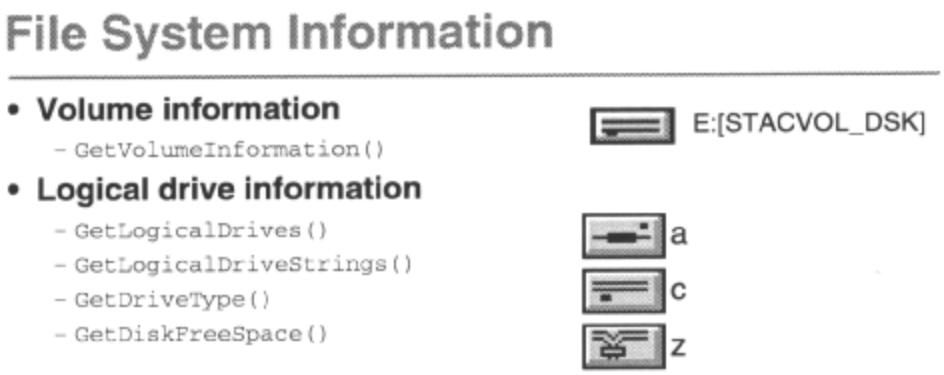
FILE_NOTIFY_CHANGE_FILENAME file renames, creations or deletions
FILE_NOTIFY_CHANGE_DIRNAME directory creations or deletions
FILE_NOTIFY_CHANGE_ATTRIBUTES file attribute changes
FILE_NOTIFY_CHANGE_SIZE file size changes on disk
FILE_NOTIFY_CHANGE_LAST_WRITE file last write time changes on disk
FILE_NOTIFY_CHANGE_SECURITY file security descriptor changes

When a wait is satisfied, `FindNextChangeNotification()` requests the handle to be signaled again the next time the filter condition is met in the specified directory or subtree, and the handle is re-waited on When no longer interested in file change notification, `FindCloseChangeNotification()` should be called to invalidate the handle.

Here is an example to wait for temporary file changes:

```
BOOL  bNoError    TRUE;
HANDLE hNotify FindFirstChangeNotification( “c:\\tmp”, /*subtree root*/
                                                  TRUE, /*watch subtrees*/
                                          FILE_NOTIFY_CHANGE_FILENAME
                                          FILE_NOTIFY_CHANGE_DIRNAME );

/*watch for file and
           directory changes*/
if ( hNotify             INVALID_HANDLE_VALUE )
```

```
while ( bNoError )
      WaitForSingleObject( hNorify, INFINITE ); /“re-read directory and
take action*/
      bNoError    FindNextChangeNotification( hNotify );

FindCloseChangeNotification( hNotify );
```



It is possible to obtain information about a file system.

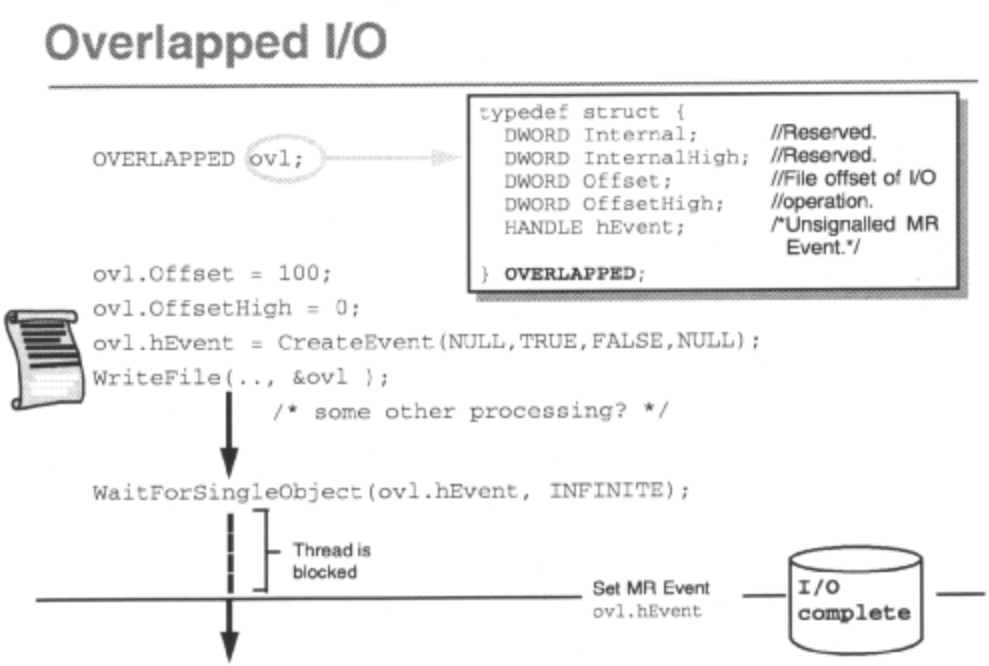
`GetVolumeInformation()` returns information about the file system on a given volume. It yields the following information:

- Volume name and serial number file system root directory file system name, e.g. FAT, HPFS
- File system flags concerning case sensitivity and UniCode maximum length of file names for the file system

It is possible to find which drives are present on the system. `GetLogicalDrives`() returns a bitmask specifying which drives are present (bit 0 = a:, bit I = b: etc.) and `GetLogicalDriveStrings`() returns drive root directory names in a string formatted thus:
`"c:\\0d:\\0\\server\share\0\0"`.

Windows 3.1 routines like:

`GetDriveType`() removable, fixed or network
`GetSystemDirectory`()
`GetWindowsDirectory`()
`GetDiskFreeSpace`() bytes per sector, sectors per cluster, free clusters, total clusters etc are still supported.



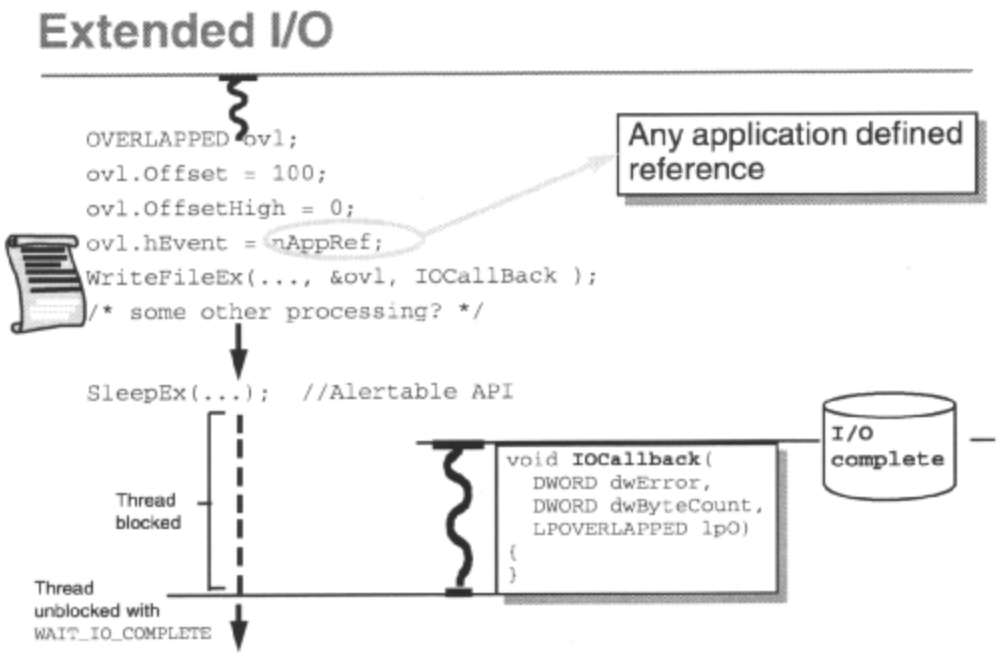
Applications may use ReadFile() or WriteFile() to perform *'synchronous' or 'asynchronous* I/O. Synchronous I/O means that the caller will be blocked until the I/O operation completes. Asynchronous I/O means that the caller may return immediately from instigating the I/O operation, and will be notified at a later date of the I/O completion. Asynchronous I/O allows the caller to continue processing while an I/O operation completes. The caller could perform more asynchronous I/O and have many I/O operations outstanding at any time. This is called *'overlapped'* I/O and is a major benefit of asynchronous I/O operations. Extended I/O, using the ReadEileEx() and WriteFileEx(), allows asynchronous I/O only. This is discussed on the next slide.

To perform asynchronous I/O, the file must have been opened with the FILE_FLAG_OVERLAPPED flag set in the *'attributes'*. If not, an asynchronous read or write will return FALSE and GetLastError() will return ERROR_INVALID_PARAMETER. When an asynchronous read or write is performed, the address of an OVERLAPPED structure is passed. The position of the file pointer is ignored in asynchronous I/O operations because of the possibility of overlapped I/O; the OVERLAPPED structure specifies where to begin the operation, and the file pointer must be updated after the operation completes, if desired.

With an asynchronous read or write performed with ReadFile() or WriteFile(), the function may behave as for a synchronous operation, if the operation can be performed quickly enough. However, if it can't, the function returns FALSE immediately, and `GetLastError()` returns `ERROR_IO_PENDING`. In this case a handle to a Manual Reset Event object, which is specified in the *OVERLAPPED* structure, will be signaled when the operation completes and the caller can wait on this `using WaitForSingleObject() or WaitForMultipleObjects() The event handle must be reset` before each transfer ensues. `GetOverlappedResult()` reports the results of the **I/O** operations for functions that returned FALSE with ERROR_I0_PENDING.

For non-overlapped I/O it would be possible to wait on the file handle, which is set to non-signaled when the I/O operation is started, and signaled when it completes. This wouldn't work for overlapped I/O, as there would be no way of knowing which of a number of simultaneous requests has completed.

The OVERLAPPED structure must not be an automatic variable, because otherwise it may become invalid before the I/O operation completes. Each overlapped I/O operation must have its own OVERLAPPED structure.

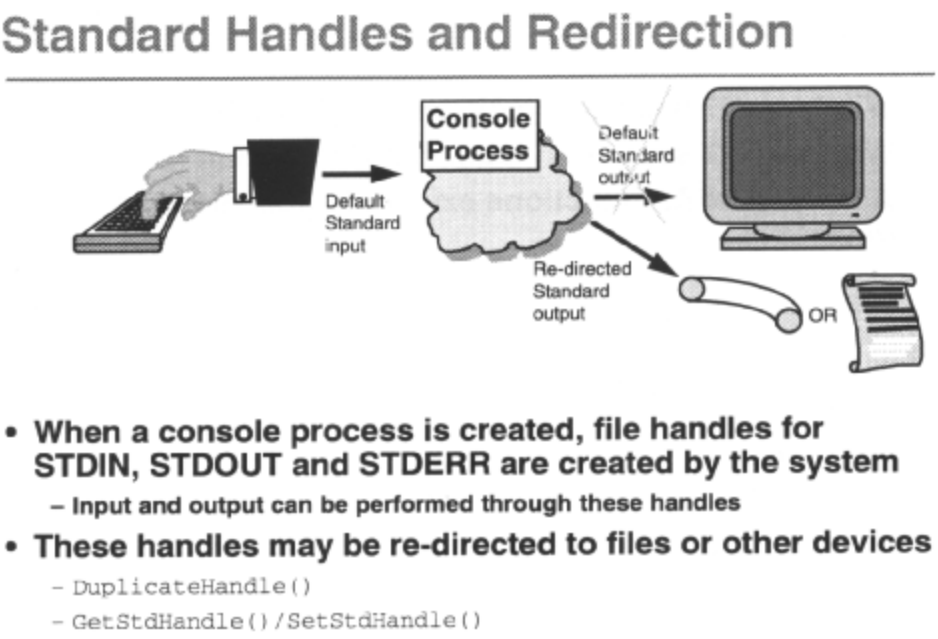


Alternatively, `ReadFileEx()` and `WriteFileEx()` allow a completion callback function to be specified, which will be called after the I/O operation completes when the caller is next in an *'alertable wait'* state. `ReadFileEx()` and `WriteFileEx()` only allow asynchronous I/O and return FALSE if there is a problem (use GetLastError() for more information) or TRUE to indicate that the operation has not yet completed. ReadEileEx() and WriteFileEx() can use the Event handle member of an OVERLAPPED structure to pass information to the completion routine, since an Event object is not needed.

Waiting is done with SleepEx() ,WaitForSingleObjectEx() or WaitForMultipleObjectsEx() which are the same as their non-Ex counterparts, except that they can be set to an *'alertable'* state which means that they can return with WAIT_10_COMPLETE when an I/O completion routine has been called, even though the object(s) being waited on have not become signaled. The I/O completion routine is called on a new thread, started by the Win32 subsystem.

The format of an I/O completion callback function is:

```
void IOCompletion(      DWORD dwErrorCode,
                        DWORD nBytesXferred, LPOVERLAPPED ipOverlapped );
```



Win32 automatically opens three standard files for each process when it is created: standard input *(stdin),* output *(stdout),* and error *(stderr).* These only make sense in a console application, where stdin is usually the keyboard, and stdout and stderr are usually directed to the screen. These standard handles can be queried by using `GetStdHandle()` and redirected to the handle for an open file or device using `SetStdHand1e`(). For instance, setting stdout as a file or device handle causes an application to write to the file or device instead of the screen.

The standard handles, amongst others, can be duplicated with the `DuplicateHandle()` call. If an application wishes to restore any redirected standard handles during the lifetime of the program, it will need to have duplicated them. An unnamed pipe is often used to redirect standard output from a child to a parent process, as detailed in the *"JPC:Pipes and Mailsiots"* chapter.

DuplicateHandle() can also be used to duplicate the following types of handles for passing to unrelated processes:

- console I/O
- event
- file and file mapping objects
- mutex
- pipe (unnamed and named)
- process
- semaphore
- thread

`DuplicateHandle()` requires an open *'source'* handle to be duplicated, and an open handle to the process in whose context the source handle is valid. This requires the source process to pass the handle and its process id to the *'target'* process, which can then call `OpenProcess()` to obtain the source process handle, and then call

DuplicateHandle() A *'duplicate access'* can be requested for the handle, but specifying a *'duplicate option'* of DUPLICATE_SANE_ACCESS is common, which causes the duplicate handle to have the same access as the source. A *'duplicate option'* of
DUPLICATE_CLOSE_SOURCE causes the source handle to be closed when the duplicate handle is created, otherwise it is left open.

Summary

- Windows NT has an asynchronous I/O model
- Windows NT supports installable file systems
- A number of API functions exist to handle files and devices