The NTFS File System

1. Introduction

Purpose

This chapter explains briefly the NTFS implementation, the file system of the operating system Windows 2000 and presents the main API Win32 functions related to file management.

Objectives

The chapter has the following main objectives:

1. understanding the NTFS file system;
2. understanding the types of files and the access rights from NTFS;
3. understanding the NTFS functions which work with files;
4. understanding the creation process of alternate data streams files.

General presentation

The NTFS (**NT F**ile **S**ystem) is a file system especially developed for Windows NT and upgraded for Windows 2000. NTFS4 is used for Windows NT, while the file system of Windows 2000 is NTFS5. Windows XP Microsoft uses a slightly upgraded version of NTFS5.

The main features of this file system are the following:

- it uses disk addresses on 64 bits and can support partitions up to 2^{64} bytes;
- Unicode characters can be included in the name of the files;
- it allows file names composed of up to 255 characters, including blanks and dots;
- it allows general indexation of the files;
- it offers the possibility of managing dynamically the sectors
- because of the POSIX compatibility, it allows *hard-links* creation, it is case sensitive when it comes to file names and it stores time related information regarding files;
- it allows using alternate data streams files.

2. The NTFS Disk Structure

When formatting a partition (volume) which has a NTFS file system, some system files are created. The most important is the **Master File Table (MFT)**, which stores information about all the files and directories from the NTFS partition.

The first bit of information on a NTFS partition is the Boot Sector, which is the 0 sector of the partition and stores a program (code) used for booting the system. Other information needed by the booting program (e.g.: information needed to access the partition) can be stored in the sectors 1 up to 16, reserved for this purpose. Figure 1 presents a NTFS partition at the end of the format operation.

The first file on a NTFS partition is **the MFT** file. Every file stored on a NTFS partition has at least one entry in the MFT; this is also true for the MFT file. All the information about a file (name, dimension, time related information, access rights, effective data) is kept in the MFT or in an area found outside the MFT, which also keeps entries for the MFT. The file attributes are stored in the MFT if their size allows for them to be kept in the MFT entries; if not, they are stored in auxiliary areas of the HDD outside the MFT file, but associated to the file's entry in the MFT.

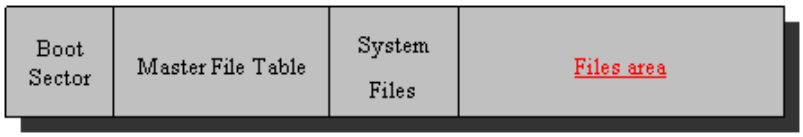

Figure 1.

Figure 1. The structure of a NTFS partition

The table below presents all the attribute types defined by the NTFS file system. These are used internally by the NTFS; the user has no direct access to the attributes and is not allowed to define new types. This list is expandable, meaning that in the future new file attributes will be defined.

Table 1. Types of file attributes in NTFS

Attribute type	Description
Standard information	Includes information related to time and number of links.
Attribute Lists	Lists the locations of all attribute records that do not fit in the MFT record.
File Name	A repeatable attribute for both short and long file names. The long file names can have up to 255 Unicode characters. The short file names are of 8.3 format. The additional names of the hard links needed by the POSIX can be included as additional file name attributes.
Security Descriptor	Stores the file's owner and the users which have the right to access the file.
Data	Stores the file's data. The NTFS allows multiple data attributes per file. Usually, every file has one unnamed data attribute. A file can also have one or more named data attributes, each having a particular syntax.
Object ID	A unique file identifier on the partition, used by the distributed link tracking service. Not all the files have this attribute.
Logged Tool Stream	Similar to a data stream, although the operations are recorded in the log file of NTFS, just like NTFS metadata changes. This is used by EFS.
Reparse Point	Used for volume mount points. It is also used by IFS (Installable File System) filter drivers to mark certain files as special to that driver.
Index Root	Used to implement folders and other indexes.
Index Allocation	Used to implement folders and other indexes.
Bitmap	Used to implement folders and other indexes (for very large directories).
Volume Information	Used only in the $Volume file system. Contains the volume version.
Volume Name	Used only in the $Volume file system. Contains the volume label.

The *metadata* files are the data structures used by NTFS for accessing and managing files. This file system is based on the principle *everything is a file*. That is why the volume descriptor, the booting information, the records of the ..defecte...sectors etc. are all stored in files.

The files which store the metadata information of the NTFS are described in the table below:

Table 2. Metadata Stored in the Master File Table

File Name	MFT Record	Purpose of the File
$MFT	0	MFT
$MFTmirr	1	File stored at the logical center of the disk. It is a duplicate image of the first 16 records of the MFT.
$LogFile	2	Log file record.
$Volume	3	Contains information about the volume: volume label, volume version etc.
$AttrDef	4	The standard file attributes on the volume.
$.	5	The root directory.
$Bitmap	6	The bitmap of the volume's unallocated space.
$Boot	7	The boot sector (bootable volume).
$BadClus	8	The list of bad clusters.
$Secure	9	Security descriptor for all the files.
$Upcase		File – a table which stores the equivalence between lowercase characters and Unicode uppercase characters which are found in the file names on the volume. This file is necessary because the NTFS file names are memorized in Unicode, which has 65.000 distinct characters and it is not easy to search the equivalent for a

	10	lowercase or an uppercase.
$Quota	11	The file in which the access rights of the users with respect to the disk space are recorded (it is functional only starting with NTFS5).

3. NTFS file types and access rights

In NTFS we can identify the following file types:

- *system files*: the files presented in the table above; they contain information (metadata) which is used only by the operating system.
- *Alternate Data Streams (ADS) files*: files which besides the main data set also contain other distinct sets of data. All these data sets are represented by attributes of *Data* type. Chapter 5 describes how to create and use these auxiliary data sets.
- *compressed files*: NTFS can compress and uncompress files *on-the-fly* (when writing or reading data on or from them). This mechanism is not seen by the applications which use such files.
- *encrypted files:* EFS (Encrypted File System) offers support for storing encrypted files on a NTFS volume. Encryption is transparent to the user who encrypted the file. The other users cannot access these files.
- *sparse files:* files which do not store the information into a contiguous area; the written areas alternate with big, not written areas (spars). NTFS allows setting a special attribute for this file in order to indicate for the I/O system to allocate area on the disk only for the written areas of the file
- *files of type "hard-link":* files especially introduced by NTFS5. These files allow for a file to be accessed through more paths, without duplicating the effective data. If we delete a file which has more than one link, the data will not be deleted from the disk until all links are destroyed. A *hard-link* file can be created by using the function *CreateHardLink* or the command "`fsutil hardlink create`" (in Windows XP).

In NTFS, the access rights are managed through În ceea ce priveşte drepturile de acces, în NTFS ele sunt gestionate prin *access control lists (ACL)*. These lists contain information regarding the access rights of every user or group of users with respect to a file. The access rights are called *permissions.*
NTFS defines 6 main permissions called *special permissions.* The table below describes these permissions and explains their effect upon files and directories.

Table 3. NTFS permissions

Permission	Character	Allowed Access for Files	Allowed Access for Directories
Read	R	Read file content	Read directory content
Write	W	Modify file content	Modify directory content (create files or subdirectories)
Execute	X	Execute program	Traverse subdirectories
Delete	D	Delete file	Delete directory
Change Permissions	P	Change access rights for file	Change access rights for directory
Take Ownership	O	Change owner	Change owner

In order to allow for more "fine-tuned" control over different kinds of access, starting with Windows 2000 some groups of permissions were introduced; they are called *permission components*. Each of them groups one or more special permissions:

- Traverse Folder / Execute File: X
- List Folder / Read Data: R

- Read Attributes: R + X
- Read Extended Attributes: R
- Create Files / Write Data: W
- Create Folders / Append Data: W
- Write Attributes: W
- Write Extended Attributes: W
- Delete Subfolders and Files: D
- Delete: D
- Read Permissions: R + W + X
- Change Permissions: P
- Take Ownership: O

By using the graphic interface to set access rights, we can come across other groups of permissions.

4. API calls for the NTFS file system

All the resources (files, processes) of the operating systems based on Windows NT are identified by *handlers*. A *handler* is a token which allows us to identify the access of a program with respect to a resource. It is similar to the file descriptors used in Unix. Hereby, when a file is created or opened such a handler is returned; by using this handler the file can be accessed for reading and writing operations.

The *CreateFile* function

The function is used for creating a new file or for opening an existing file. The syntax of the function is the following:

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
        DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile);
```

Parameters:

lpFileName – pointer to a null-terminated string that specifies the name of the file to be created or opened

dwDesiredAccess – the type of access to the file. An application can obtain read, write, read/write or query devices access right. The most important values of this parameter are:

0 – device query access to the file.

GENERIC_READ – read access to the file. Data can be read from the file and the file pointer can be moved. It is combined with GENERIC_WRITE for read-write access.

GENERIC_WRITE – write access to the file. Data can be written to the file and the file pointer can be moved.

DELETE – the right to delete the file.

READ_CONTROL – the right to read information from the security descriptor of the file.

WRITE_OWNER – the right to change the owner in the security descriptor of the file.

SYNCHRONIZE – the right to use the file for synchronization. In this way a thread can wait until the file is in the marked (marcată) state.

GENERIC_EXECUTE – execution right.

GENERIC_ALL – read, write and execution right.

dwShareMode – specifies the way in which the file can be shared among more users. If *dwShareMode* is 0 and **CreateFile** is successful, the file cannot be shared and cannot be opened again until the handler has not been closed. In order to share the file among more users, one of the following combinations of values can be used:

FILE_SHARE_DELETE – the next opening operations on the file will succeed only if the deleting right is requested.

FILE_SHARE_READ – the next opening operations on the file will succeed only if the reading right is requested.

FILE_SHARE_WRITE – the next opening operations on the file will succeed only if the writing right is requested.

lpSecurityAttributes – pointer to a structure *SECURITY_ATTRIBUTES,* which determines if the handler can be inherited by the children processes. If the attribute *lpSecurityAttributes* is NULL, then the handler cannot be inherited.

dwCreationDisposition – specifies the action which will be undergone by the existing or new created file. It needs to take one of the following values:

CREATE_NEW – creates a new file. The function fails if the file already exists.

CREATE_ALWAYS – creates a new file. If the file already exists, the function overwrites the file, deletes the existing attributes and combines the file attributes and the flags specified by *dwFlagsAndAttributes* with FILE_ATTRIBUTE_ARCHIVE.

OPEN_EXISTING – opens a file. The function fails if the file does not exist.

OPEN_ALWAYS – opens the file if it exists. If the file does not exist, the function creates the file as if the *dwCreationDisposition* had the value CREATE_NEW.

TRUNCATE_EXISTING – opens the file. Once opened, the file is truncated such that its size equals 0 bytes. The process which called the function has to open the file with at least GENERIC_WRITE access. The function fails if the file does not exist.

dwFlagsAndAttributes – specifies the attributes and the flags of the file. A file can have the following attributes: *archive, encrypted, hidden, normal, not content indexed, offline, read-only, system, temporary*. A file can have the following flags: *write through, overlapped, no buffering, random access, sequential scan, delete on close, backup semantics, POSIX semantics, open reparse point, open no recall*.

hTemplateFile – specifies a handler with GENERIC_READ access to a template file. The template file provides the attributes for the file being created.

If the function is successful it returns a value which is the handler used to access the file. If the function fails, the returned value INVALID_HANDLE_VALUE. In order to obtain more detailed error information *GetLastError* needs to be called.

The *DeleteFile* function

The function deletes an existing file and has the following syntax:

```
BOOL DeleteFile(
  LPCTSTR lpFileName);   // the name of the file
```

If the function succeeds it returns a non-zero value. If the function fails, it returns 0.

The *CloseHandle* function

The function closes an open file handle.

```
BOOL CloseHandle(
  HANDLE hObject);   //the object's handler
```

If the function succeeds it returns a non-zero value. If the function fails, it returns 0.

The *ReadFile* function

The **ReadFile** function reads data from a file, starting at the position indicated by the file pointer. After the read operation has been completed, the file pointer is adjusted by the number of bytes actually read, unless the file handler is created with the overlapped attribute. If the file's handler is created for overlapped I/O, the application has to adjust the file pointer's position after the read operation.

.

```
BOOL ReadFile(
  HANDLE hFile,                // the file's handler
  LPVOID lpBuffer,             // data buffer
  DWORD nNumberOfBytesToRead,  // nr of bytes to be read
  LPDWORD lpNumberOfBytesRead, // nr of bytes read
  LPOVERLAPPED lpOverlapped);  // overlapped buffer
```

Parameters:

hFile – handler to the file to be read. The file's handler had to be created with the GENERIC_READ access to the file.
lpBuffer – pointer to the buffer which receives the data read from the file.
nNumberOfBytesToRead – the number of bytes which have to read from the file.
lpNumberOfBytesToRead – pointer to the variable which receives the number of read bytes.
lpOverlapped – pointer to an OVERLAPPED structure. This structure Această structură is required if hFile was created with FILE_FLAG_OVERLAPPED.

The function returns if the number of bytes requested has been read or if an error occurred. If the function succeeds the returned value is non-zero.

The *WriteFile* function

This function writes data to a file and is designed for both synchronous and asynchronous operations. The function begins writing data to the file at the position indicated by the file pointer. After the write operation has been completed, the file pointer is adjusted by the number of bytes actually written, except when the file is opened with FILE_FLAG_OVERLAPPED.

```
BOOL WriteFile(
 HANDLE hFile,                // the file's handler
 LPCVOID lpBuffer,            // data buffer
 DWORD nNumberOfBytesToWrite, // nr of bytes to be written
 LPDWORD lpNumberOfBytesWritten,  // nr of bytes written
 LPOVERLAPPED lpOverlapped);      // overlapped buffer
```

The parameters have a similar meaning to the parameters of the *ReadFile* function.

If the function succeeds the returned value is non-zero. If the function fails, the returned value is 0.

The *SetFilePointer* function

The function moves the pointer of an opened file.

```
DWORD SetFilePointer(
  HANDLE hFile,
  LONG lDistanceToMove,
  PLONG lpDistanceToMoveHigh,
  DWORD dwMoveMethod);
```

Parameters:
hFile – handler to the file whose pointer is going to be moved. The file's handler had to be created with one or both of the following access types to a file: GENERIC_READ or/and GENERIC_WRITE.
lDistanceToMove – the least significant 32 bits of the signed value which specifies the number of bytes with which the pointer will be adjusted.
lpDistanceToMoveHigh – pointer to the most significant 32 bits of the distance expressed in a signed value on 64 bits. If the most significant 32 bits are not needed, this pointer has to be set to NULL.
dwMoveMethod – the pointer's starting point. This parameter can have one of the following values:
FILE_BEGIN – the starting point is 0, i.e. the beginning of the file.
FILE_CURRENT – the starting point is the current value of the file's pointer.
FILE_END – the starting point is the current EOF (end of file).

If the function **SetFilePointer** is successful and *lpDistanceToMoveHigh* is NULL, the returned value is the least significant double-word **DWORD** of the file's new pointer. If *lpDistanceToMoveHigh* is not NULL, then the function returns the least significant double-word **DWORD** of the file's new pointer; it also stores the most significant double-word **DWORD** of the file's new pointer in the value LONG to which the parameter points to. If the function fails and *lpDistanceToMoveHigh* is NULL, the returned value is INVALID_SET_FILE_POINTER.

The *GetFileAttributes* function

This function retrieves a set of FAT file system attributes for a specified file or directory.

```
DWORD GetFileAttributes(
  LPCTSTR lpFileName);   // the name of the directory/file
```

If the function is successful, the returned value will contain the attributes of the specified file or directory. The attributes can be one or more of the values presented in paragraph 2.2.1. completed with sparse directories or files.

The *LockFile* function

The function locks an area of an opened file in order to ensure mutual exclusion. Locking the area ensures the fact that other processes cannot access it.

The syntax of the LockFile function is presented below:

```
BOOL LockFile(
  HANDLE hFile,
  DWORD dwFileOffsetLow,
  DWORD dwFileOffsetHigh,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh);
```

Parameters:
hFile – handler to the file whose region is going to be locked. The name of the file had to be created with at least one of the following access types to the file: GENERIC_READ or/and GENERIC_WRITE.
dwFileOffsetLow – specifies the least significant word of the starting byte offset where the lock should begin.
dwFileOffsetHigh – specifies the most significant word of the starting byte offset where the lock should begin.
nNumberOfBytesToLockLow – specifies the least significant word of the length of the byte range to be locked.
nNumberOfBytesToLockHigh – specifies the most significant word of the length of the byte range to be locked.

If the function succeeds the returned value is non-zero. If the function fails, the returned value is 0.

The *UnlockFile* function

The function unlocks an area of an opened file. The syntax of this function is similar to the one of the *LockFile* function.

The *CreateDirectory* function

This function creates a new directory. If the existing file system allows security options for directories and files, the function will apply a security descriptor specified for the new directory.

```
BOOL CreateDirectory(
  LPCTSTR lpPathName,     // the name of the directory
  LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

If the function succeeds the returned value is non-zero. If the function fails, the returned value is 0.

The *RemoveDirectory* function

The function deletes an existing empty directory.

```
BOOL RemoveDirectory(
  LPCTSTR lpPathName);   // the name of the directory
```

If the function succeeds the returned value is non-zero. If the function fails, the returned value is 0.

The *FindFirstFile* function

The function searches a directory for a file whose name matches the specified filename. **FindFirstFile** examines subdirectory names as well as filenames.

```
HANDLE FindFirstFile(
  LPCTSTR lpFileName,             // the name of the file
  LPWIN32_FIND_DATA lpFindFileData);  // data buffer which //stores information about the found
        file
```

If the function is successful, the returned value is a search handle used by calls such as *FindNextFile* or *FindClose*. If the function fails, the returned value is INVALID_HANDLE_VALUE.

The *FindNextFile* function

This function continues a file search from a previous call to the **FindFirstFile** function.

```
BOOL FindNextFile(
  HANDLE hFindFile,                  // search handle
  LPWIN32_FIND_DATA lpFindFileData); // data buffer
```

If the function succeeds the returned value is non-zero. If the function fails, the returned value is 0.

The *MoveFile* function

This function moves an existing file or directory, including its copies.

```
BOOL MoveFile(
  LPCTSTR lpExistingFileName,  // the old name of the file
  LPCTSTR lpNewFileName);      // the new name of the file
```

If the function succeeds the returned value is non-zero. If the function fails, the returned value is 0.

The *SetCurrentDirectory* function

The function changes the current directory of the current process.

```
BOOL SetCurrentDirectory(
  LPCTSTR lpPathName);   // the name of the new directory
```

If the function succeeds the returned value is non-zero. If the function fails, the returned value is 0.

5. Files with alternate data streams

As it has already been mentioned, the NTFS system allows us to associate more data attributes (i.e. data streams) to a file. Every file has one main unnamed data stream associated to it. If necessary, other alternative named data streams can be associated to the file. This ensures for some file data to be accessed as a distinct unit. For example, a graphical application can store the thumbnail for a bitmap image into a different stream included in the NTFS file which stores the image. The alternate data streams have different sizes than the main file, but they have the same permissions.

An alternate data stream can be easily created from the command line. In order to create the main data stream we can type:

```
echo "The main data stream of the file" > Data
```

In this way, we have created the file with the name *Data*. An alternative data stream with the name *ADS* will be added:

```
echo "Alternate data stream" > Data:ADS
```

We can notice that the data stream added by typing the command above does not appear among the files listed in the directory. The alternate data stream does not enlarge the size of the main file. In order to read the content of the main data stream and of the alternate data stream we execute the commands:

```
more  < Data
more  < Data:ADS
```

In order to open an alternate data stream in Notepad, the name of the stream needs to have an extension, for example: *Data:ADS2.txt*. The stream can be edited with the command: "`notepad Data:ADS2.txt`".

The alternate data streams can also store binary data, i.e. executable files. They can be executed with the command: "`start .\Data:fis.exe`".

The Windows system uses alternate data streams when it specifies supplementary data for a file through *Properties->Summary*. The streams offer viruses a good way of hiding, because they cannot be seen in the files list and they do not modify the dimension or the time stamp of the main file. Windows does not offer programs for detecting alternate data streams

6. Examples

1. Copying a file using the API functions of Windows 2000

```
#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 10

void main() {

HANDLE inhandle, outhandle;
char buffer[BUF_SIZE];
int count;
DWORD ocnt;

/* Open the input and the output file */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

/* Copy the file */
do {
   s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
   if (s && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s>0 && count>0);

/* Close the files */
CloseHandle(inhandle);
CloseHandle(outhandle);

}
```

2. Find the .txt files from the current directory and set the attributes as read-only.

```
#include <windows.h>
#include <stdio.h>

WIN32_FIND_DATA FileData;
HANDLE hSearch;
DWORD dwAttrs;
BOOL fFinished = FALSE;

void main() {

// Search .TXT files in the current directory
hSearch = FindFirstFile("*.txt", &FileData);
if (hSearch == INVALID_HANDLE_VALUE)
{
    printf("No .TXT files found.");
    return;
}
```

```
// For each file change the attribute in read-only if it is //not already set as read-only
while (!fFinished)
{
   dwAttrs = GetFileAttributes(FileData.cFileName);
        if (!(dwAttrs & FILE_ATTRIBUTE_READONLY))
        {
            SetFileAttributes(FileData.cFileName,
                dwAttrs | FILE_ATTRIBUTE_READONLY);
        }

    if (!FindNextFile(hSearch, &FileData))
    {
        if (GetLastError() == ERROR_NO_MORE_FILES)
        {
            printf("No more .TXT files. Search completed.");
            fFinished = TRUE;
        }
        else
        {
            printf("Couldn't find next file.");
            return;
        }
    }
}
 // close the search handle
FindClose(hSearch);
}
```

7. Problems

1. How can we find the size of a file by using the function *SetFilePointer* ?
2. Using the API functions presented in this chapter, write a C program which lists in reverse order the lines of a file.
3. Write a program which reads and writes the characters 0, 20, 40… from a previously created file.
4. Write a program which after being launched in the background N times writes into a file the ID of the current process. None of the programs are able to continue their execution while all the processes have not written their unique ID in the file. In the end, every process prints the ID of the next process.
5. Write a program which allows writing strings of characters (taken from the standard input) into a file, starting with a given position. The call of the program has the following format: *rw poz fis*.
6. Write a program which lists all the files from a given directory.
7. Write a program which eliminates the byte number 0, 5, 10, 15, … from a file. Observation: do not use a temporary file.
8. Write a program which demonstrates the way of using the *hard link* type files.
9. Create a file text and associate to it an alternate data stream which should contain the program solitaire (*sol.exe*).