# Virtual file system 6 WINDOWS

martedì 11 novembre 2025 09:28

Andiamo ora a vedere come è fatto il Virtual file system di windows:

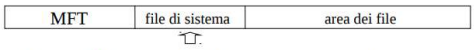
Hard-drive divisi in volumi (partizioni), e organizzati in cluster fino a 64K byte

per ogni volume si ha una **MFT (Master File Table)**

ad ogni file corrisponde almeno un elemento nella MFT

l'elemento contiene:

- nome del file: fino a 255 char Unicode
- informazioni sulla sicurezza
- nome DOS del file: 8+3 caratteri
- i dati del file, o puntatori per il loro accesso – organizzazione stile extent




Bitmap dei cluster + logfile (recupero in caso di crash)

Dove il **master file table è il duale** del vettore degli i-node in UNIX. Possibile che in una MFT vi sono più entry collegate allo stesso file (si usa questa tecnica per i file relativamente piccoli / medi). Invece per i file grandi c'è comunque l'indicizzazione. Vediamo in dettaglio :

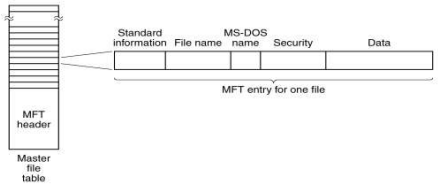
| | |
|---|---|
| Informazioni standard | **attributi di accesso, timestamp** |
| Lista di attributi | **dove localizzare nella MFT attributi che non entrano in un singolo record** |
| Descrittore di sicurezza | **permessi di accesso** per proprietario ed altri utenti |
| Nome | **identificazione nel sistema** |
| Dati | **visti tipicamente come attributo o come indici per l'accesso** |

Quindi nel caso di file immediati (piccoli/medi) si ha una situazione del genere :

per piccoli file i dati sono direttamente nella parte dati nell'elemento della MFT (file immediati)

per file grandi la parte dati dell'elemento della MFT contiene gli indirizzi di cluster o di gruppi di cluster

se un elemento della MFT non basta si aggregano altri




Quindi dopo questo paragone andiamo a vedere l'**architettura del sistema operativo windows**:


- scrittura pigra: gli aggiornamenti vanno su hard-drive su base periodica
- sostituzione cache LRU
- swapping tramite thread di sistema (su file)

Per quanto riguarda il gestore della cache , ricordiamoci la gestione con il buffer cache ( blocchi già presenti con swap in/out) , portando così a lettura anticipata e scrittura ritardata . Andiamo ora a vedere le API principali :

1. **Creazione di un file**
   a. Cambia struttura della MFT
   b. Permette apertura del file
   c. Torna una maniglia

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

d.

**Descrizione**

• invoca la creazione di un file

**Restituzione**

• un handle al nuovo file in caso di successo

e. Dove il quinto parametro rappresenta come deve aprire il file ( creazione, troncamento ecc ecc)

f. Il terzo parametro rappresenta la possibilità di riapertura del file , se lo stesso è aperto

g. In dettaglio :

h.

`lpFileName`: puntatore alla stringa di caratteri che definisce il nome del file da creare

`dwDesiredAccess`: specifica la modalità di accesso al file da creare (GENERIC_READ, GENERIC_WRITE)

`dwShareMode`: specifica se e quando il file può essere nuovamente aperto prima di essere stato chiuso (FILE_SHARE_READ, FILE_SHARE_WRITE)

`lpSecurityAttributes`: specifica il descritore della sicurezza del file

`dwCreationDisposition`: specifica l'azione da fare se il file esiste e quella da fare se il file non esiste (CREATE_NEW, CREATE_ALWAYS, OPEN_EXISTING, TRUNCATE_EXISTING)

`dwFlagsAndAttributes`: specifica varie caratteristiche del file (FILE_ATTRIBUTES_NORMAL, … HIDDEN, ... DELETE_ON_CLOSE)

`hTemplateFile`: specifica un handle ad un file di template

i. Questa è la versione generica , senza specificare il tipo di codifica , quindi dobbiamo usare la variante ASCII createFileA o unicode createFileW

j. Vediamo ora in dettaglio la famosa struttura lpSecurityAttributes

i.

```
typedef struct _SECURITY_ATTRIBUTES {
        DWORD nLength;
        LPVOID lpSecurityDescriptor;
        BOOL bInheritHandle;
} SECURITY_ATTRIBUTES
```

Implementa la logica ACL

**Descrizione**

• struttura dati che specifica permessi

**Campi**

• nLength: va settato SEMPRE alla dimensione della struttura
• lpSecurityDescriptor: puntatore a una struttura SECURITY_DESCRIPTOR
• bInheritHandle: se uguale a TRUE un nuovo processo puo' ereditare l'handle a cui fa riferimento questa struttura

ii. Il secondo parametro permette di puntare ad una ACL (access control list) , specificando i permessi di accesso

iii. Quindi se andiamo a vedere in dettaglio come è fatta una ACL si arriva alla seguente bipartizione

iv.

specifica la descrizione della sicurezza di oggetti, quindi anche di file

è formata da una lista di ACE (Access Control Entry)

nel caso di generazione di oggetti in cui il descrittore di sicurezza non è specificato, ACL viene popolata a partire dall'*access token* del processo chiamante

in tal caso si ha una ACL di default

ACL è formata da DACL (Discretionary ACL) e SACL (System ACL)

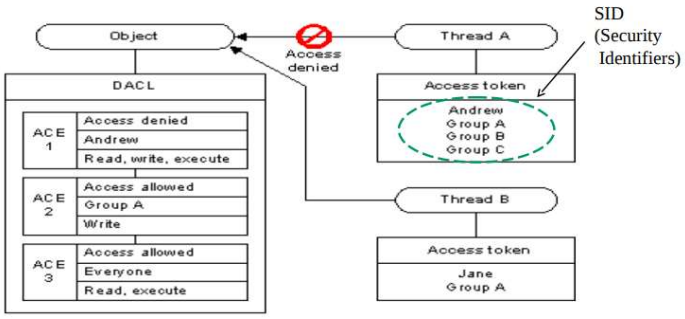
DACL specifica i permessi

SACL specifica azioni di log da eseguire in base agli accessi

sono entrambe parti di un security descriptor

v. Quindi **ACL (insieme di ACE) =DACL+SACL**

vi. Vediamo un esempio

1. 


2. Quindi in una ACL conta l'ordine

vii. Riepilogando:

Ogni processo ha dei Security IDentifier (come utente e come gruppo)

I SID costituiscono l'access token che può direttamente garantire speciali privilegi (ad esempio l'accesso a qualsiasi file)

1. Quando un processo cerca di accedere a un oggetto viene utilizzato l'access token per controllare se il processo ha diritti incondizionati sull'oggetto

   Altrimenti il kernel "scandisce" l'ACL controllando le singole ACE

   La prima ACE che specificamente garantisce o nega l'accesso richiesto è decisiva nell'esito dell'accesso

2. La cui gestione viene fatto nei registri :

   Tramite registry attraverso il file

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\
   CurrentVersion\ProfileList
   ```

3. Tramite (power)shell command

   ```
   wmic useraccount get name,sid
   whoami /user
   ```

k. **Gestione ACL**

i. The **GetSecurityInfo** function retrieves a copy of the *security descriptor* for an object specified by a handle.

Syntax

C++

```
DWORD WINAPI GetSecurityInfo(
  _In_      HANDLE               handle,
  _In_      SE_OBJECT_TYPE       ObjectType,
  _In_      SECURITY_INFORMATION SecurityInfo,
  _Out_opt_ PSID                 *ppsidOwner,
  _Out_opt_ PSID                 *ppsidGroup,
  _Out_opt_ PACL                 *ppDacl,
  _Out_opt_ PACL                 *ppSacl,
  _Out_opt_ PSECURITY_DESCRIPTOR *ppSecurityDescriptor
);
```

field pointers into the descriptor

need free after usage

ii. L'ultimo è un puntatore a puntatore

iii.

```
typedef enum _SE_OBJECT_TYPE {
  SE_UNKNOWN_OBJECT_TYPE       = 0,
  SE_FILE_OBJECT,
  SE_SERVICE,
  SE_PRINTER,
  SE_REGISTRY_KEY,
  SE_LMSHARE,
  SE_KERNEL_OBJECT,
  SE_WINDOW_OBJECT,
  SE_DS_OBJECT,
  SE_DS_OBJECT_ALL,
  SE_PROVIDER_DEFINED_OBJECT,
  SE_WMIGUID_OBJECT,
  SE_REGISTRY_WOW64_32KEY
} SE_OBJECT_TYPE;
```

iv.

The **LookupAccountSid** function accepts a *security identifier* (SID) as input. It retrieves the name of the account for this SID and the name of the first domain on which this SID is found.

Syntax

C++

```
BOOL WINAPI LookupAccountSid(
  _In_opt_  LPCTSTR       lpSystemName,
  _In_      PSID          lpSid,
  _Out_opt_ LPTSTR        lpName,
  _Inout_   LPDWORD       cchName,
  _Out_opt_ LPTSTR        lpReferencedDomainName,
  _Inout_   LPDWORD       cchReferencedDomainName,
  _Out_     PSID_NAME_USE peUse
);
```

actual buffers

account type

buffer sizes

v. Quindi in generale :

Servizi WinAPI da usare e sequenza di uso

1. `InitializeSecurityDescriptor` ← Reset della struttura dati
2. `SetSecurityDescriptorOwner` — Scrittura dei SID nella struttura dati
3. `SetSecurityDescriptorGroup`
4. `InitializeAcl` ← Inizializzazione di un buffer ACL
5. `AddAccessDeniedAce....` — Inserimento di ACE in ACL
6. `AddAccessAllowedAce...`
7. `SetSecurityDescriptorDacl` ← Collegamento di ACL al security descriptor

vi. Vediamo un esempio

1.

```c
// ACL-access.c : Defines the entry point for the console application.
// you can compile with either ASCII or UNICODE
//
#
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#include <tchar.h>
#include <Sddl.h>
#include "accctrl.h"
#include "aclapi.h"
#pragma comment(lib, "advapi32.lib")

int main(void)
{
	DWORD dwRtnCode = 0;
	PSID pSidOwner = NULL;
	BOOL bRtnBool = TRUE;
	LPTSTR AcctName = NULL;
	LPTSTR DomainName = NULL;
	DWORD dwAcctName = 1, dwDomainName = 1;
	SID_NAME_USE eUse = SidTypeUnknown;
	HANDLE hFile;
	PSECURITY_DESCRIPTOR pSD = NULL;
	TCHAR *stringSID;
	// Get the handle of the file object.
	hFile = CreateFile(TEXT("myfile.txt"),GENERIC_READ,FILE_SHARE_READ,NULL,CREATE_ALWAYS, //|OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
	// Check GetLastError for CreateFile error code.
	if (hFile == INVALID_HANDLE_VALUE) {
		DWORD dwErrorCode = 0;
		dwErrorCode = GetLastError();
		_tprintf(TEXT("CreateFile error = %d\n"), dwErrorCode);
		return -1;
	}
	// Get the owner SID of the file.
	dwRtnCode = GetSecurityInfo(hFile,SE_FILE_OBJECT,OWNER_SECURITY_INFORMATION,&pSidOwner,NULL,NULL,NULL,&pSD);
	printf("this is the owner address %p\nthis is the structure address %p\n", pSidOwner, pSD);
	// Check GetLastError for GetSecurityInfo error condition.
	if (dwRtnCode != ERROR_SUCCESS) {
		DWORD dwErrorCode = 0;
		dwErrorCode = GetLastError();
		_tprintf(TEXT("GetSecurityInfo error = %d\n"), dwErrorCode);
		return -1;
	}
	ConvertSidToStringSid(pSidOwner, &stringSID);
	_tprintf(TEXT("this is the owner SID %s\n"), stringSID);
	AcctName = (LPTSTR)malloc(4096);
	DomainName = (LPTSTR)malloc(4096);
	dwAcctName = 4096;
	dwDomainName = 4096;
	bRtnBool = LookupAccountSid(
		NULL,			// name of local or remote computer
		pSidOwner,		// security identifier
		AcctName,		// account name buffer
		(LPDWORD)&dwAcctName,	// size of account name buffer
		DomainName,		// domain name
		(LPDWORD)&dwDomainName, // size of domain name buffer
		&eUse);			// SID type

	// Print the account name.
	_tprintf(TEXT("Account owner = %s\n"), AcctName);
	return 0;
}
```

2. Ritorna una entry della ACL

2. **Chiusura del file**

BOOL CloseHandle(HANDLE hObject)

chiude un oggetto

a.

**Descrizione**

• invoca la chiusura di un oggetto (ad esempio un file)

**Restituzione**

• 0 in caso di fallimento

3. **Lettura da file**
   a. Leggo una sequenza di byte in modo diretto

```
BOOL ReadFile(HANDLE hFile,
              LPVOID lpBuffer,
              DWORD nNumberOfBytesToRead,
              LPDWORD lpNumberOfBytesRead,
              LPOVERLAPPED lpOverlapped)
```

   b.

**Descrizione**

• invoca la lettura di una certa quantità di byte da un file

**Restituzione**

• 0 in caso di fallimento

   c. Dove

   i.
- hFile: handle valido al file da cui si vuole leggere
- lpBuffer: puntatore all'area di memoria nella quale i caratteri letti devono essere bufferizzati
- tnNumberOfBytesToRead: definisce il numero di caratteri (byte) che si vogliono leggere
- lpNumberOfBytesRead: puntatore a un intero positivo corrispondente al numero di caratteri effettivamente letti in caso di successo
- lpOverlapped: puntatore a una struttura OVERLAPPED da usarsi per I/O asincrono

4. **Scrittura su file**

```
BOOL WriteFile(HANDLE hFile,
               LPCVOID lpBuffer,
               DWORD nNumberOfBytesToWrite,
               LPDWORD lpNumberOfBytesWritten,
               LPOVERLAPPED lpOverlapped)
```

   a. **Descrizione**

   • invoca la scrittura di una certa quantità di byte su un file

   **Restituzione**

   • 0 in caso di fallimento

   b.
- hFile: handle valido al file su cui si vuole scrivere
- lpBuffer: puntatore all'area di memoria che contiene i caratteri da scrivere
- tnNumberOfBytesToWrite: definisce il numero di caratteri (byte) che si vogliono scrivere
- lpNumberOfBytesWritten: puntatore a un intero positivo corrispondente al numero di caratteri effettivamente scritti in caso di successo
- lpOverlapped: puntatore a una struttura OVERLAPPED da usarsi per I/O asincrono

   c. Esempio : copia di un file

i.

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

#define BUFSIZE 1024

int main(int argc,char **argv)
{
        HANDLE sd,dd;
        DWORD size,result;
        char buffer[BUFSIZE];

        if(argc!=3) /* controllo numero argomenti */
        {
                printf("sage: copia source target\n");
                return -1;
        }
        /*Apertura file sola lettura */
        sd=CreateFile(argv[1],GENERIC_READ,
0,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
        if(sd==INVALID_HANDLE_VALUE)
        {
                printf("cannot open file \n");
                return -1;
        }
        /*creazione del file di destinazione */
        dd=CreateFile(argv[2],GENERIC_WRITE,
0,NULL,CREATE_ALWAYS,FILE_ATTRIBURE_NORMAL,NULL);
        if(dd==INVALID_HANDLE_VALUE)
        {
                printf("cannot open destination file \n");
                return -1;
        }
        do // qui iniziamo operazioni copia
        {
                if(ReadFile(sd,buffer,BUFSIZE,&size,NULL)==0)
                {
                        printf("cannot read from source file\n");
                        return -1;
                }
                if(WriteFile(dd,buffer,size,&result,NULL)==0)
                {
                        printf("cannot write to destination \n");
                        return -1;
                }
        }while(size>0);
        CloseHandle(sd);
        CloseHandle(dd);
}
```

ii. Il quale copia il contenuto di un file in un altro

5. **Rimozione di un file**

BOOL DeleteFile(LPCTSTR lpFileName)

**Descrizione**

• invoca la cancellazione di un file

a. **Parametri**

• lpFileName: puntatore alla stringa di caratteri che definisce il nome del file che si vuole rimuovere

**Restituzione**

• un valore diverso da zero in caso di successo, 0 in caso di fallimento

6. **Riposizionamento del file pointer**

a. Ci spostiamo in un'altra zone del file

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

**Descrizione**

b. • invoca il riposizionamento del file pointer

**Restituzione**

• `INVALID_SET_FILE_POINTER` in caso di fallimento, i 32 bit meno significativi del nuovo valore del file pointer (valutato in caratteri dall'inizio del file) in caso di successo

c. Dove:

`hFile`: handle di file che identifica il canale di input/output associato al file per il quale si vuole modificare il file pointer

`lDistanceToMove`: i 32 bit meno significativi di un valore intero con segno indicante il numero di caratteri di cui viene spostato il file pointer

d.

`lpDistanceToMoveHigh`: (opzionale) puntatore a un long contenente i 32 bit più significativi del valore in lDistanceToMove

`dwMoveMethod`: tipo di spostamento da effettuare (`FILE_BEGIN`, `FILE_CURRENT`, `FILE_END`)

7. **Gestione degli standard handle**

```
HANDLE WINAPI GetStdHandle( _In_  DWORD nStdHandle );

BOOL WINAPI SetStdHandle( _In_  DWORD nStdHandle, _In_  HANDLE hHandle );
```

a.

| Value | Meaning |
|---|---|
| **STD_INPUT_HANDLE** (DWORD)-10 | The standard input device. |
| **STD_OUTPUT_HANDLE** (DWORD)-11 | The standard output device. |
| **STD_ERROR_HANDLE** (DWORD)-12 | The standard error device. |

b.

```c
// channel-redirection.c : Defines the entry point for the console application.
//

/******* write stdin data to a file by writing to stdout ********/
//please compile with ASCII settings

#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#define BUFFER_SIZE 1024


int main(int argc, char **argv[])
{
	HANDLE hFile, hStdout, hStdin;
	DWORD length, filePointer, dummy;
	char input_buffer[BUFFER_SIZE];
	DWORD res;


	hFile = CreateFileA((LPCSTR)argv[1], GENERIC_READ | GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
	if (hFile == INVALID_HANDLE_VALUE) {
		printf("Cannot open target file\n");
		return -1;
	}

	printf("will redirect stdout-handle to file named: %s - give me the input\n", argv[1]);
	fflush(stdout);

	hStdin = GetStdHandle(STD_INPUT_HANDLE);

	SetStdHandle(STD_OUTPUT_HANDLE, hFile);
	hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

	while (1){
		ReadFile(hStdin, (LPVOID)input_buffer, BUFFER_SIZE, &res, NULL);
		WriteFile(hStdout, (LPVOID)input_buffer, res, &dummy, NULL);
	}

	return 0;
}
```

c. Scrive ridirezionando in canale sul file

8. **Creazione di directory**

a.

```
BOOL WINAPI CreateDirectory(
    __in LPCTSTR lpPathName,
    __in_opt LPSECURITY_ATTRIBUTES  lpSecurityAttributes );

BOOL WINAPI RemoveDirectory(
    __in LPCTSTR lpPathName );

HANDLE FindFirstFile(
  [in] LPCSTR          lpFileName,
  [out] LPWIN32_FIND_DATA lpFindFileData
);

BOOL  FindNextFile(
  [in]  HANDLE              hFindFile,
  [out] LPWIN32_FIND_DATA lpFindFileData
);
```

ASCII vs UNICODE

9. **Creazione hard links**

```
BOOL WINAPI CreateHardLink( LPCTSTR lpFileName,
                            LPCTSTR lpExistingFileName,
                            LPSECURITY_ATTRIBUTES lpSecurityAttributes );
```

**a.**

*lpFileName* [in]
- The name of the new file.
- This parameter cannot specify the name of a directory.

*lpExistingFileName* [in]
- The name of the existing file.
- This parameter cannot specify the name of a directory.

*lpSecurityAttributes*
- Reserved; must be NULL.

10. **Creazione simbolyc link**

```
BOOLEAN WINAPI CreateSymbolicLink(
  __in LPTSTR lpSymlinkFileName,
  __in LPTSTR lpTargetFileName,
  __in DWORD dwFlags )
```

a.

*lpSymlinkFileName* [in]
- The symbolic link to be created.

*lpTargetFileName* [in]
- The name of the target for the symbolic link to be created.
- If *lpTargetFileName* has a device name associated with it, the link is treated as an absolute link; otherwise, the link is treated as a relative link

*dwFlags* [in]
- Indicates whether the link target, *lpTargetFileName*, is a directory.

11. Vediamo degli esempi
    a. **Reverse file**
    b. Scrive nel file in ordine inverso

C.

```c
#define BUFFER_SIZE 1024

#define OPTIMIZED_BUFFER_MANAGEMENT


void Error(char * message) {
        puts(message);
        ExitProcess(-1);
}


int main(int argc, char **argv[])
{
        HANDLE hReverseFile;
        DWORD length, filePointer, dummy;
        char buffer[2][BUFFER_SIZE], *input_buffer, *temp_buffer;
        DWORD res;
        int i = 0;

        printf("file name is: %s - give me the input\n", argv[1]);
        hReverseFile = CreateFileA((LPCSTR)argv[1], GENERIC_READ | GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hReverseFile == INVALID_HANDLE_VALUE) Error("Cannot open reverse file\n");

        input_buffer = buffer[i];
        temp_buffer = buffer[i + 1];

        while (1){
                scanf("%s", input_buffer);
                //printf("%s\n",input_buffer);
                //fflush(stdout);
                if (strcmp(input_buffer, ".") == 0) break;
                length = strlen(input_buffer);
                input_buffer[length++] = '\n';
                SetFilePointer(hReverseFile, 0, NULL, FILE_BEGIN);
                do{
                        ReadFile(hReverseFile, (LPVOID)temp_buffer, length, &res, NULL);
                        filePointer = SetFilePointer(hReverseFile, -res, NULL, FILE_CURRENT);
                        //printf("file pointer is %d",filePointer);
                        //fflush(stdout);
                        WriteFile(hReverseFile, (LPVOID)input_buffer, length, &dummy, NULL);
#ifdef OPTIMIZED_BUFFER_MANAGEMENT
                        i = (i++);
                        input_buffer = buffer[i % 2];
                        temp_buffer = buffer[(i + 1) % 2];
#else
                        memcpy(input_buffer, temp_buffer, res);
#endif
                        length = res;
                } while (res>0);

        }

        return 0;
}
```