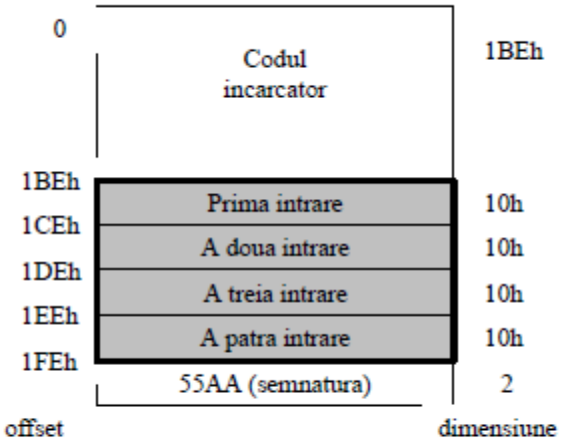
# 2 Sisteme de fișiere. Partiționarea HDD

## 2.1 Introducere

Hard disk-ul (HDD) reprezintă cel mai utilizat mijloc de stocare a datelor unui PC, urmând apoi mediile de tip memory stick, DVD, CD, TAPE, etc. Dintre avantajele HDD față de celelalte medii de stocare putem enumera capacitatea, viteza, mobilitatea. Referitor la capacitate, în ultimii 30 de ani ea a crescut de la 100MB și costuri de 100$/1MB la 10TB și costuri de 0.01$/1MB. Bineînțeles ca și viteza de acces și rata de transfer au avut o evoluție pozitiva. Putem astfel sa comparam dezvoltarea HDD-urilor cu cea a microprocesoarelor.

Orice sistem de operare înainte de a se instala pe un HDD necesită partiționarea acestuia (chiar și o singură partiție). Astfel partiționarea HDD oferă posibilitatea de a avea, pe același HDD, mai multe sisteme de operare. Un alt avantaj este acela că se pot gestiona mult mai eficient partițiile dacă sunt mai mici față de cele de dimensiuni mari (mai puțin spațiu pierdut pentru salvarea fișierelor mici).

La începutul HDD pe primul sector având adresa fizică 0 [cilindru=0, cap=0, sector=1] se creează o tabelă în care sunt păstrate informații despre partițiile care există pe acel HDD. Acel sector poartă numele de MBR (Master Boot Record). În acea tabelă se pot păstra informațiile despre maxim 4 partiții. Structura sectorului cu tabela de partiții este prezentată în figura următoare. Acest sector pornește cu un program încărcător, program care este copiat de BIOS de pe acest sector și care este lansat de acesta.

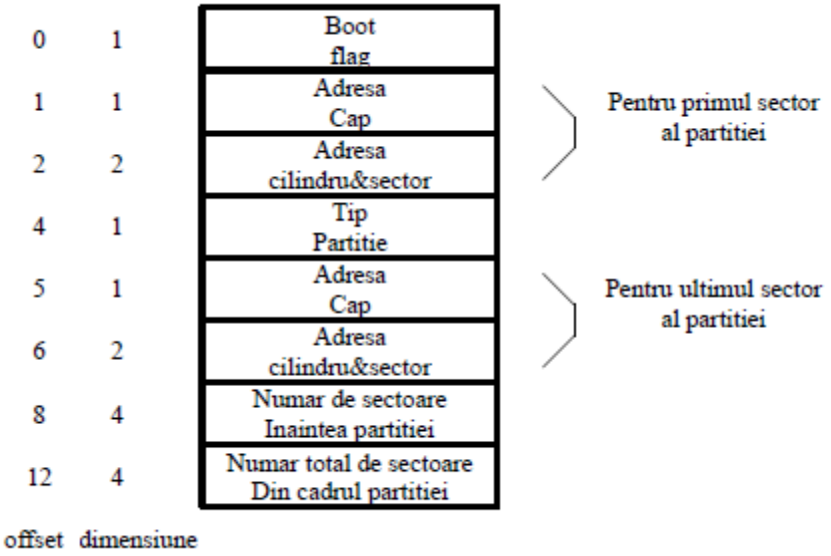



Structura sectorului cu tabela partitiilor
(Partea hasurata este **tabela partitiilor**)

## 2.2 Tabela de partiții propriu-zisă

Deși riguros tabela partițiilor este doar o tabelă în cadrul sectorului [0,0,1] prin abuz de limbaj se folosește uneori pentru acest sector notația "tabela partițiilor" în loc de "sectorul tabelei partițiilor" (pierzând din rigoarea exprimării).

O partiție este descrisa de o **intrare** în tabela partițiilor având următorul format:




Structura unei intrări în tabela de partiții

**Boot flag** – este un câmp care indică dacă partiția respectiva este boot-abila sau nu, adică dacă există un SO care pornește de pe acea partiție sau nu (80h – boot-abila, 00h – neboot-abila).

**Adresa cap** – indică adresa capului pentru începutul și sfârșitul partiției. **Adresa cilindru&sector** indică adresa fizică a primului/ ultimului sector al partiției. Din cei 16 biți folosiți 10 sunt pentru adresa cilindrului, iar 6 pentru adresa sector așezați astfel.

$$C_1C_2C_3C_4C_5C_6C_7C_8 \quad C_9C_{10}S_1S_2S_3S_4S_5S_6$$

**Tip Partiție** indică tipul partiției respective (Vezi secțiunea 2.5). Câmpul este esențial deoarece un sistem de operare nu accesează decât partițiile al căror conținut îl poate interpreta (îl suportă) conform acestui tip.

**Număr de sectoare înaintea partiției** – Indică numărul de sectoare

aflate înaintea partiției respective.

**Număr total de sectoare în partiție** – Indică numărul total de sectoare pe care îl ocupă partiția.

La prima vedere faptul că se păstrează începutul și sfârșitul partiției folosind adresa cap/cilindru/sector precum și numărul de sectoare înaintea partiției și numărul de sectoare pe care îl ocupă partiția poate părea informație redundantă. Pentru HDD mici de pana la 4GB este adevărat. Problema apare însă la HDD cu o capacitate mai mare de 4GB unde adresa de tip cap/cilindru/sector căruia îi sunt alocați 3 octeți nu mai poate păstra corect începutul și sfârșitul partițiilor aflate după primii 4GB de HDD.

Pentru a putea utiliza eficient HDD cu capacitate mai mare de 4GB se folosește tehnologia **LBA** (Logical Block Address - Adresare logică a blocurilor), sectoarele HDD nemaifiind accesate pe baza adresei cap/cilindru/sector, ci folosind un singur număr care identifică un sector de pe HDD. Se folosește astfel numărul de sectoare înaintea partiției, număr căruia îi sunt rezervați 4 octeți și deci se vor putea aloca partiții pe HDD cu capacități de pana la 2TB.

## 2.3 Codul încărcător

Pe sectorul tabelei partițiilor se găsește, pe lângă tabela partițiilor propriu-zisă și un cod încărcător. Acesta este încărcat de către ROM-BIOS în memorie la adresa 7C00:0000 și este lansat în execuție de fiecare dată la pornirea sistemului. În mare, el execută următorii pași:

- se mută (își mută propriul cod) la altă adresă (oricum, în acest moment, cu excepția tabelei vectorilor de întrerupere, toată memoria este liberă) și face un salt la copie, urmând a se executa în continuare de la această adresă;

- analizează intrările din tabela partițiilor și caută o partiție activa (boot-abilă);
- verifică să nu mai fie încă o partiție activă;
- încarcă în memorie la adresa 7C00:0000 primul sector al partiției active (pe baza adresei sale fizice din intrarea corespunzătoare) numit **sector de boot** al partiției active;
- verifică semnătura (de sistem) 55AAh din finalul sectorului;
- predă controlul codului încărcător de pe sectorul de boot tocmai încărcat printr-un salt de tipul :

```
JMP FAR 7C00:0000
```

În funcție de diferite situații întâlnite se pot genera următoarele mesaje:

**“Invalid partition table”** – tabela este invalidă în sensul că nu este nici o partiție boot-abilă sau sunt mai multe asemenea partiții;

**“Error loading operating system”** – la tentativa de a încărca sectorul de boot al sistemului de operare activ s-a întâlnit eroare de citire;

**“Missing operating sistem”** – sectorul de boot citit nu conține semnătura de sistem (55AAh).

Oricare din aceste erori este eroare fatală, soluția alternativă fiind *resetarea* și *încărcarea sistemului de pe CD* sau *dispozitiv de stocare* urmată de corectarea situației din tabela partițiilor.

Remarcăm că mecanismul de partiționare este **în afara unui sistem de operare anume**, particularizarea unuia sau altuia din sistemele de operare începe doar cu sectorul de boot al partiției respective. Bineînțeles, tot acest mecanism de partiționare se bazează pe “proprietatea” sistemelor de operare *de a nu accesa zone de disc din afara partițiilor proprii* sau pe care nu le cunosc/suportă (deși nu exista nici un mecanism de evitare a acestui lucru).

În cazul programelor de tip BootManager care au o interfață grafică prietenoasă și pun la dispoziție o serie de utilitare este nevoie de mai mult spațiu pe HDD pentru stocarea aplicației. Deoarece spațiul alocat codului încărcător este relativ mic, 446 octeți, acestea pun pe primul sector al HDD doar un cod de inițializare, restul aplicației aflându-se în cadrul unei partiții și fiind încărcat și lansat în execuție de către codul de încărcare.

## 2.4 Particularități ale partiționării

După cum s-a văzut mai sus, tabela de partiții poate să conțină maxim patru intrări. În majoritatea cazurilor divizarea HDD în patru partiții este suficientă.

Pentru ca totuși numărul de partiții să nu fie limitat la patru, s-au introdus conceptele de **partiție primară** și **partiție extinsă**.

**Partiția primară** – este o partiție "normală" care ocupă o intrare în tabela partițiilor și care poate fi făcută boot-abilă.

**Partiția extinsă** – este o partiție care conține la rândul ei alte **partiții logice** și care vor fi folosite de către utilizatori pentru stocarea fișierelor.

## 2.5 Anexa 1 – Identificatori pentru sistemele de fișiere

| Valoare hexa | Descriere |
|---|---|
| 00h | Partition Not Used |
| 01h | FAT12 |
| 02h | XENIX Root |
| 03h | XENIX User |

| | |
|---|---|
| 04h | FAT16 (CHS address mode, and partition size is less than 32 |
| 05h | Extended Partition (CHS address mode) |
| 06h | FAT16 (CHS address mode, and partition size is larger than 32 Mb) |
| 07h | NTFS, HPFS, IFS, exFAT |
| 08h | AIX, OS/2, QNX |
| 09h | AIX Bootable |
| 0Ah | OS/2 Bootmanager |
| 0Bh | FAT32 (CHS address mode) |
| 0Ch | FAT32 (LBA address mode) |
| 0Eh | FAT16 (LBA address mode) |
| 0Fh | Extended Partition (LBA address mode) |
| 10h | OPUS |
| 12h | CPQ Diagnostics |
| 14h | Omega FS (The file system designed for the Maverick OS) |
| 15h | Swap Partition |
| 16h | Hidden big FAT16 (Not official I think) |
| 24h | NEC MS-DOS 3.x |
| 40h | VENIX 80286 |
| 42h | Secure File System |
| 50h | LynxOS |
| 51h | Novell |
| 52h | CP/M |
| 56h | GoldenBow |
| 61h | SpeedStor |
| 63h | Unix SysV386 |
| 64h | Novell Netware |
| 65h | Novell Netware |
| 75h | PC/IX |
| 80h | Old MINIX |
| 81h | Linux / MINIX |

| | |
|---|---|
| 82h | Linux Swap |
| 83h | Linux Nature |
| 93h | Amoeba |
| 94h | Amoeba BBT |
| A5h | BSD/386 |
| B7h | BSDI file system |
| B8h | BSDI swap |
| C7h | Cyrnix |
| DBh | CP/M |
| E1h | DOS Access |
| E3h | DOS R/O |
| F2h | DOS Secondary |
| F4h | SpeedStor |
| FEh | LANstep |
| FFh | BBT |

## 2.6 Anexa 2 – Cod încărcător

```
        .MODEL SMALL
SegPart         SEGMENT AT 0
        ASSUME CS:SegPart,DS:NOTHING
;       INCARCATORUL DE PARTITII
;-      ------------------------
;       Prima actiune ce o efectueaza incarcatorul de partitii este,
;de a transfera articolul de partitionare de la adresa 0000:7C00H
la
;adresa 0000:0600H, adresa 0000:7C00H fiind folosita in continuare
;pentru a citi sectorul de BOOT al partitiei active daca aceasta
este
;gasita. Se transfera intreg articolul (512 octeti)cu alte cuvinte
;incarcatorul de partitii, tabela de partitii si marcajul de
;sistem.
        ORG     7C00H
STARTLDR:                       ;Adresa (0000:7C00H)
        CLI                     ;Invalideaza intreruperile
        XOR AX,AX               ;Sterge AX (AX=0000)
        MOV SS,AX               ;Initializeaza registrul SS (SS=0000)
        MOV SP,offset STARTLDR ;Initializeaza registrul SP (SP=7C00H)
                                ;Pregateste transferul articolului
```

```
                              ;de partitionare
        MOV SI,SP             ;Registrul index sursa SI=7C00H
        PUSH    AX
        POP ES                ;ES=0000
        PUSH AX
        POP DS                ;DS=0000
        STI                   ;Valideaza intreruperile
        CLD                   ;Sterge indicatorul de directie
        MOV     DI,0600H      ;Registrul index destinatie DI=0600H
        MOV     CX,0100H      ;Se muta 256 cuvinte (512 octeti)
        REPNZ   MOVSW         ;Transfera articolul de partitionare
        JMP     FAR PTR LOADPART
                              ;si preda controlul incarcatorului
                              ;de partitii (0000:061DH).
;       Incarcatorul de partitii analizeaza fiecare intrare din
tabela
;de partitii cu scopul de a detecta o partitie avind octetul
BOOT
;FLAG 80H adica o partitie activa. Daca esta gasita o astfel de
;partitie  se preiau din tabela adresa de inceput a acestei,
;partitii care corespunde sectorului de BOOT se testeaza ca
restul
;de intrari din tabela sa aiba BOOT FLAG = 00H si se face salt la
;rutina care citeste secorul de BOOT al partitiei.
        ORG 061DH
LOADPART:                     ;Incarcatorul 0000:061DH
        MOV SI,0600H+01BEH    ;SI pe prima intrare in tabela de
                              ;partitii 0000:07BEH
        MOV BL,04             ;BL=4 numar intrari in tabela de
partitii
NEXTENTR:
        CMP BYTE PTR [SI],80H ;test BOOT FLAG partitie activa
        JZ INITREAD           ;salt daca activa
        CMP BYTE PTR [SI],0   ;test BOOT FLAG
        JNZ ERRORPTB          ;Salt afisare mesaj tabela de partitii
                              ;invalida.
        ADD SI,10H            ;SI = Adresa urmatoarea intrare
        DEC BL                ;Decrementeaza numar intrari
        JNZ NEXTENTR          ;Salt analiza urmatoarea intrare
        INT 18H               ;Apel interpretor BASIC (ROM). Acesta
                              ;se gaseste doar la microcalculatoarele
                              ;IBM, clonele folosind interpretorul
                              ;GWBASIC livrat impreuna cu sistemul de
                              ;operare ca o comanda externa.
INITREAD:                     ;Pregateste citire BOOT sector
        MOV DX,[SI]           ;DL=80H drive DH=adresa HEAD
        MOV CX,[SI+2]         ;CX= Adresa cilindru,sector in format
                              ;BIOS DX si CX se preiau direct din
                              ;tabela
```

```
        MOV     BP,SI           ;Salveaza registrul SI
TSTNEXT:                        ;Test pentru restul de intrari
                                ;BOOT FLAG = 0 ?
ADD     SI,10H                  ;Urmatoarea intrare in tabela de
                                ;partitii
        DEC     BL              ;Test sfirsit tabela
        JZ      READBOOT        ;Citeste sectorul de BOOT CMP
        BYTE PTR [SI],0         ;Test BOOT FLAG = 0
        JZ      TSTNEXT         ;Salt urmatoarea intrare
ERRORPTB:
        MOV     SI,offset INVPARTB
;Mesaj de eroare tabela de partitii invalida
;Rutina ERRORMSG afiseaza pe display via INT 10H BIOS (Servicii
;Video),un sir ASCIIZ, reprezentind mesajul de eroare al
;incarcatorului catre utilizator, dupa care se asteapta intr-o
bucla
;infinita din care poate fi scos doar cu RESET.
ERRORMSG:
        LODSB                   ;Incarca octet mesaj
        CMP     AL,0            ;Test sfirsit sir
        JZ      WAITLOOP        ;Daca da salt asteptare in bucla
        PUSH    SI              ;Salveaza SI in stiva
        MOV     BX,7            ;BL=7 culoarea de fond
        MOV     AH,0EH          ;Cod functie BIOS
        INT     10H             ;Apel functie BIOS (Video services)
        POP     SI              ;reface SI
        JMP     ERRORMSG        ;Urmatorul caracter
WAITLOOP:
        JMP     WAITLOOP        ;Bucla infinita (se iese cu RESET)
; Daca exista o partitie activa in sistem, cu registrii DX si CX
;pregatiti anterior, se apeleaza aceasta rutina care face 5
tentative
;de a incarca la adresa 0000:7C00H sectorul de BOOT al partitiei,
;verifica prezenta si corectitudinea marcajului de sistem la
;sfirsitul sectorului
;de BOOT dupa care preda controlul incarcatorului de BOOT.
READBOOT:
        MOV     DI,5            ;DI=5 se fac maxim 5 tentative de
citire
RETRY:                          ;Citeste sectorul de BOOT
        MOV     BX,offset BOOTSTART ;BX=7C00H Adresa de transfer
        MOV     AX,0201H        ;AH=2 Citeste / AL=1 un sector
                                ;vezi parametrii INT 13H BIOS
        PUSH    DI              ;Salveaza registrul DI
        INT     13H             ;Apel BIOS
        POP     DI              ;Rescrie registrul DI
        JNB     READOK          ;Salt daca s-a citit sectorul de BOOT
        XOR     AX,AX           ;AH=0 Recalibrare (RESET CONTROLER)
        INT     13H             ;Apel BIOS
```

```
        DEC     DI              ;Decrementeaza numar tentative de
                                ;citire
        JNZ     RETRY           ;Se face o noua tentativa
        MOV     SI,offset ERRLOASY ;Daca dupa 5 tentative de citire a
                                ;sectorului de BOOT nu s-a reusit
        JMP     ERRORMSG        ;se afiseaza mesaj de eroare de
                        ;incarcare si se asteapta in bucla infinita.
READOK:
        MOV     SI,offset MISOPESY
                                ;Asuma mesaj lipsa sistem operare
        MOV     DI,offset BOOTSTART+1FEH
                                ;DI   adresa   marcaj   sistem   de   pe
                        sectorul
                                ;de BOOT.(DI=0000:7DFEH)
        CMP     WORD PTR [DI],0AA55H
                                ;Test  marcaj sistem AA55H
        JNZ     ERRORMSG        ;Daca lipseste marcajul sistem se
                                ;afiseaza mesajul de eroare si se
                                ;asteapta in bucla infinita.
        MOV     SI,BP           ;reface registrul SI
        JMP     FAR PTR BOOTSTART
                                ;Preda controlul INCARCATORLUI  BOOT
                                ;0000:7C00H
;       Mesaje de eroare
INVPARTB        DB      'Invalid partition table',0
ERRLOASY        DB      'Error loading operating system',0
MISOPESY        DB      'Missing operating system',0
SegPart ENDS
;Aici  se  preda  controlul  incarcatorului  BOOT  dupa  citirea
;sectorului  de boot.
SegBoot        SEGMENT        AT 0
        ASSUME CS:SegBoot
        ORG     7C00H
BOOTSTART:
SegBoot         ENDS
        END STARTLDR
```

## 2.7 Funcțiile API de acces la disk

Pentru accesul la fișierele de pe disk, și nu numai, sistemul de operare pune la dispoziție o serie de funcții API care permit deschiderea, citire, scrierea de byte pe disk.

Astfel pentru deschiderea/ crearea unui fișier pe disk sistemul de operare pune la dispoziție 2 funcții CreateFile și OpenFile.

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

Funcția CreateFile permite crearea sau deschiderea unui fișier sau a unui dispozitiv de I/O. Cele mai comune dispozitive de I/O sunt fișierele, stream-urile, tastatura și porturile de comunicație.

Parametrii funcției CreateFile sunt:

- `lpFileName` – un pointer la un șir de caractere care conține calea și numele fișierului care se dorește creat/deschis;
- `dwDesireAccess` – accesul care se dorește la acel fișier, cele mai uzuale valori sunt `GENERIC_READ` sau `GENERIC_WRITE`. Se preferă folosirea constantelor simbolice declarate deja in biblioteca sistemului de operare decât folosirea valorilor corespunzătoare (cod mai lizibil);
- `dwShareMode` – se poate specifica dacă fișierul deschis poate fi accesat și de altă aplicații și comenzile permise celeilalte aplicații care accesează fișierul respectiv. Se recomandă valoarea 0;
- `lpSecurityAttributes` – un pointer la o structură în care se descrie drepturile de acces ale handle-rului returnat dacă acesta este moștenit de alte procese. Acest parametru poate fi NULL;
- `dwCreationDisposition` – acțiunea care trebuie făcută de sistemul de operare în cazul în care fișierul există sau nu există;
- `dwFlagsAndAttributes` – atributele fișierului sau

dispozitivului;
- hTemplateFile – acest parametru poate fi NULL;

Dacă funcția este executată cu succes de SO aceasta va deschide/crea fișierul specificat și întoarce un handler (identificator unic în sistemul de operare) prin intermediul căruia se va ”discuta” cu sistemul de operare despre acel fișier. Dacă funcția nu este executată cu succes valoarea returnată este `INVALID_HANDLE_VALUE`. Pentru a vedea exact care este motivul pentru care nu a putut fi deschis/creat fișierul se poate apele funcția GetLastError().

```
DWORD WINAPI GetLastError(void);
```

Această funcție nu are nici un parametru și returnează un număr care reprezintă codul erorii. O listă completă cu toate codurile de eroare pentru Windows se găsește căutând după SystemErrorCodes pe web.

O altă funcție mai simplă pentru accesul doar la fișiere și care nu se mai recomandă a se utiliza în noile sisteme de operare este OpenFile cu prototipul:

```
HFILE OpenFile(
  LPCSTR     lpFileName,
  LPOFSTRUCT lpReOpenBuff,
  UINT       uStyle
);
```

Pentru citire din fișier există la nivelul sistemului de operare funcția ReadFile cu următorul prototip:

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

Parametrii funcției sunt:

- `hFile` – handle-rul fișierului / dispozitivului din care se dorește realizarea citirii;
- `lpBuffer` – un pointer la un buffer în care să se salveze informațiile citite din fișier;
- `nNumberOfBytesToRead` – numărul de bytes care se doresc a fi citiți din fișier;
- `lpNumberOfBytesRead` – un pointer către o variabilă de tip DWORD în care se va scrie după apelul funcție câți bytes au fost citiți din fișier. Acest parametru este scris de SO.
- `lpOverlapped` – un pointer la o structură prin care se specifică dacă sunt permise acțiuni simultane în același fișier. De exemplu citire și scriere simultană din fișier sau dispozitiv (cum ar fi portul serial).

Dacă funcția se execută cu succes valoarea returnată este TRUE altfel întoarce valoarea 0 (FALSE).

Funcția pusă la dispoziție de SO pentru scrierea în fișier este WriteFile și are următorul prototip:

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

Parametrii funcției sunt:

- `hFile` - handle-rul fișierului / dispozitivului în care se dorește scrierea informațiilor;
- `lpBuffer` – un pointer la un buffer în care se găsesc informațiile care se doresc a fi scrise în fișier;
- `nNumberOfBytesToWrite` – numărul de bytes care se dorește a fi scris în fișier;

- `lpNumberOfBytesWritten` – numărul de bytes care au fost scriși în fișier;
- `lpOverlapped` – un pointer la o structură prin care se specifică dacă sunt permise acțiuni simultane în același fișier;

Pentru a închide un fișier SO nu pune la dispoziție o funcție specifică ci se poate folosi funcția prin care îi spunem SO-ului să invalideze un handler din lista lui, să șteargă acel handler și să nu mai poată fi accesat de aplicații ulterior.

```
BOOL CloseHandle(
  HANDLE hObject
);
```

Pentru poziționarea în fișier SO pune la dispoziție funcția SetFilePointer cu următorul prototip:

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

Unde:

- `hFile` – handler-ul fișierului;
- `lDistanceToMove` – o valoare pe 32 biți care reprezintă partea low a numărului de bytes care se dorește a fi sărit în fișier;
- `lpDistanceToMoveHigh` – o valoare pe 32 biți care reprezintă partea high a numărului de bytes care se dorește a fi sărit în fișier;
- `dwMoveMethod` – punctul de start față de care se face saltul. Acesta poate fi `FILE_BEGIN`, `FILE_CURRENT` sau `FILE_END`.

Dacă funcția se execută cu succes întoarce valoarea low a

poziționării în fișier. Dacă nu se execută cu succes întoarce `INVALID_SET_FILE_POINTER`.

## 2.8 Probleme

**Problema 1**. Realizați un program care să afișeze sub formă de tabel informațiile aflate în tabela de partiții. Deoarece pe sistemele de operare Windows accesul la nivel de sector pe HDD nu este permis, se va folosi ca și intrare a programului fișierul MBR.bin.

Mbr.bin este un fișier în care a fost copiat conținutul primului sector al unui HDD cu mai multe partiții.

Pentru o decodificare mai ușoară a informațiilor din tabela de partiții se pot folosi următoarele structuri :

```
typedef struct {
       unsigned char BootFlag;       //flagul boot
       unsigned char BeginHead;      //adresa cap
       unsigned int  BeginSectCyl;   //adresa cilindru & sector
       unsigned char SysIdent;       //identificator sistem
       unsigned char EndHead;        //adresa cap
       unsigned int  EndSectCyl;     //adresa cilindru & sector
       unsigned long RelativeSect;   //numar relativ de //sectoare
       unsigned long TotalSect;      //numar total de sectoare
}ENTRY_PART;

typedef struct SectorZero {
       unsigned char Loader[0x01be]; //incarcatorul
       ENTRY_PART    TabPart[4];     //tabela de partitii
       unsigned int  SysSign;        //semnatura sistem
}SECTOR_ZERO;
```

Important:

- În cazul folosirii unui compilator C++ pentru Windows se va folosi directiva **#pragma pack(1)** pentru o aliniere la 1 octet a membrilor structurii de mai sus.
- Se vor verifica lungimile pentru tipurile de date din

structurile de mai sus prin comparație cu numărul de octeți aferenți fiecărui membru al structurii. În caz de neconcordantă se vor modifica tipurile membrilor structurii. Atenție că tipul de date *int* diferă de la un mediu de programare la altul și depinde de modul de adresare în mediul de programare respectiv.