Marius Seiceanu, Marius Oancea, Macarie Breazu

SISTEME DE OPERARE
INDRUMAR DE LABORATOR

Editura Universitatii Lucian Blaga
Sibiu 2007

Referenti stiintifici:

Prof. Univ. Dr. Ing. Gavril Toderean
Universitatea Tehnica din Cluj-Napoca

Prof. Univ. Dr. Dorel Duca
Universitatea Babeş-Bolyai din Cluj-Napoca

Descrierea CIP a Bibliotecii Nationale a Romaniei
SEICEANU, MARIUS MIHAIL
Sisteme de operare : indrumar de laborator/
Seiceanu Marius Mihail, Oancea Marius Virgil, Breazu Macarie. – Sibiu : Editura Universitatii “Lucian Blaga” din Sibiu, 2007
Bibliogr.
ISBN 978-973-739-449-1

I. Oancea, Marius Virgil
II. Breazu, Macarie

004.42

SISTEME DE OPERARE

Indrumar de laborator

1 In loc de introducere. Nivele de acces la resurse

1.1 Introducere

Deoarece o definitie generala (completa) a unui **sistem de operare** (SO) este dificil de dat, vom spune ca un sistem de operare este un pachet de programe (e un produs software, nu ?) care **gestioneaza resursele** unui sistem de calcul si le **pune la dispozitia utilizatorului**.

Din aceasta definitie rezulta principalele sarcini ale unui sistem de operare:

1.1.1 Gestiunea resurselor

Este functia de baza deoarece ea este cea care “da viata” hardware-ului. Ea este responsabila de gestiunea tuturor resurselor sistemului de calcul. Principalele resurse care fac obiectul gestiunii SO sunt:

- **Memoria principala**
- **Spatiul de intrare-iesire** (cel mai important: memoria de masa = unitatile de disc)
- **Unitatea centrala** – si ea trebuie gestionata, in special in SO multitasking.

Gestiunea fiecareia din aceste resurse ridica probleme specifice si presupune folosirea unor algoritmi diferiti. (Intr-un fel se gestioneaza spatiul pe disc, in alt fel memoria interna si in cu totul alt fel unitatea centrala). Aceasta functie (si importanta ei deosebita) este aceea care duce si la numirea SO “**gestionar de resurse**”. De cele mai multe ori insa algoritmii folositi, desi specifici, sunt simpli (gestionari de liste si tabele, cautari in ele, etc.), fara a presupune calcule matematice prea complicate.

Oricum aceasta gestiune a resurselor ar fi inutila in lipsa celei de a doua sarcini majore si anume:

1.1.2 Punerea resurselor la dispozitia utilizatorilor

Poate fi considerata, intr-un fel, scopul final al sistemului de operare. In acest context prin utlizatori intelegem programele de aplicatii / programatorii de aplicatii. Daca gestiunea resurselor este o problema “interna” a SO si priveste mai ales proiectantii de SO problema punerii la dispozitie a resurselor este problema principala a utilizatorilor unui SO. Din acest punct de vedere nu ne intereseaza structura interna a SO ci doar interfata acestuia cu programele de aplicatii (sistemul de operare este privit ca si “cutie neagra” descrisa doar de comportarea “la borne”). O anumita functie sistem se executa “pur si simplu” fara a sti / fara a ne interesa cum anume se realizeaza acest lucru.

1.2 Nivele de acces la resursele unui sistem de calcul

Una si aceiasi resursa poate fi accesata la diferite nivele de abstractizare. Se pot identifica urmatoarele nivele de acces:

Nivelul fizic

acest mod presupune cunoasterea foarte amanuntita a hardware-ului si programarea la nivel de adrese de memorie si de porturi. Nu ne bazam deloc pe SO ci doar pe resursele hardware. Avantajele acestui mod de acces sunt lipsa oricarei restrictii si viteza maxima de executie. Dezavantajele sunt evidente: dificultatea deosebita in programare (productivitate foarte redusa) si lipsa portabilitatii.

Nivelul serviciilor BIOS

pentru a mai atenua din dezavantajele nivelului fizic s-a introdus acest nivel (BIOS = Basic Input - Output System). El consta in servicii implementate in ROM-BIOS (deci implementarea e specifica masinii respective) si asigura functii specifice pentru accesul la fiecare resursa. Cu pretul reducerii avantajelor nivelului fizic se atenueaza semnificativ dezavantajele, principalul cistig fiind obtinerea portabilitatii.

Nivelul serviciilor (D)OS

reprezinta de fapt interfata SO cu programele de aplicatii. Este cunoscut si ca si **API** (***Application Program Interface*** - numele spune aproape totul – interfata cu programele de aplicatie) si reprezinta notatia generica pentru **toate serviciile SO disponibile**. Din punct de vedere al programatorului cunoasterea unui SO echivaleaza cu cunoasterea API-ului respectivului SO.

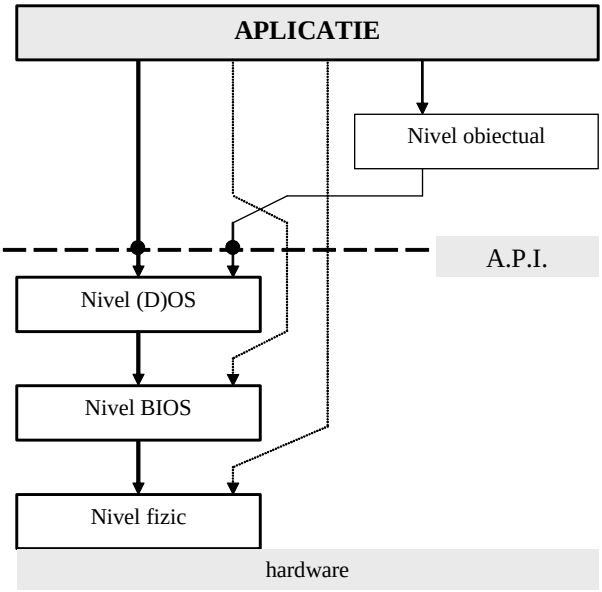
Avantajele si dezavantajele acestui nivel sunt complementare accesului la nivel fizic: programare usoara, cu productivitate mare, portabilitate buna in schimb exista anumite limitari (nu exista functii pe interfata pentru absolut orice situatie/cerinta).

In mod traditional acestea ar fi nivelele de acces la resurse. In ultimul timp evolutia dinspre nivelul fizic spre A.P.I.-uri din ce in ce mai evoluate continua si se mai poate identifica (discutabil) inca un nivel:

Nivelul obiectual asigurat de ierarhiile de obiecte specializate care insotesc compilatoarele actuale. In acest fel se ascunde programatorului A.P.I.-ul SO (devenit oricum din ce in ce mai complex) si se apeleaza metode ale obiectelor disponibile. Din punct de vedere al avantajelor/dezavantajelor se continua linia nivel fizic -> API de crestere a productivitatii.

In mod evident implementarea serviciilor unui anumit nivel se bazeaza pe serviciile nivelului urmator. La fiecare nivel exista posibilitatea de a ne lovi de limitari ale functiilor oferite si atunci solutia consta in a “cobori” la nivelul inferior. In aceasta libertate de alegere se pune problema: la ce nivel abordam realizarea unei anumite aplicatii ? Recomandarea ar fi: ***la cel mai inalt nivel*** la care putem realiza obiectivul cerut.

Tendinta actuala este aceea de a restringe accesul programatorilor la nivelele inferioare (sub Windows acest lucru este, pentru o abordare uzuala, interzis) si de a dezvolta continuu API-ul (orice DLL face de fapt - prin functiile pe care le pune la dispozitie - acest lucru).



1.3 Exemplificari ale accesului la diferite nivele (sub SO DOS).

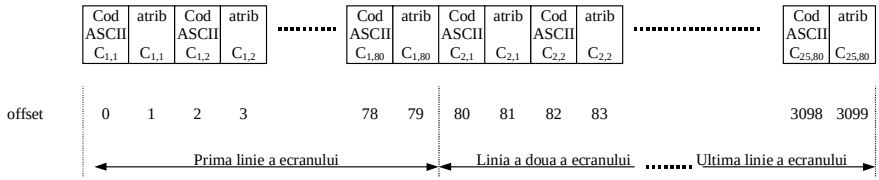
Accesul la (hard/floppy)disc

- La nivel DOS – functiile traditionale de lucru cu fisiere prin servicii ale intreruperii 21h (deschidere, citire, scriere, pozitionare, inchidere).
- La nivel BIOS – serviciile int 13h care permit accesarea intregului disc (deci si a tabelei de partitii) la nivelul adresei [cilindru:cap:sector]. Prin abuz de limbaj doar pentru disc acest mod se mai numeste (incorect) si “acces fizic la disc”.
- La nivel fizic – mult prea greoi pentru abordare la nivel de aplicatii uzuale.

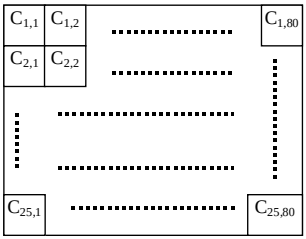
Accesul la cuplorul video (in mod text)

- La nivel DOS – functiile de scriere sir prin servicii ale intreruperii 21h. Avantajul lor este simplitatea dar nu se poate scrie in orice pozitie si cu orice culoare.
- La nivel BIOS – serviciile int 10h care permit scrierea in orice pozitie, cu orice culoare dar doar la nivel de caracter.
- La nivel fizic – descris sumar in cele ce urmeaza.

Pentru cuploarele (S)VGA in mod text memoria video este localizata la adresa B800:0 si are urmatorul continut (prezentat si in figura urmatoare):



Codul ASCII al caracterului de pe ecran urmat de atributul caracterului (culoarea sa) succesiv pentru toate cele 25*80 = 2000 caracterele de pe ecran (prezentate in figura urmatoare).



Dezavantajul metodei consta in necesitatea cunoasterii amanuntelor anterioare si in afectarea portabilitatii (nu functioneaza decit pe cuploare (S)VGA). Avantajul este lipsa oricei restrictii si viteza maxima posibila (stergerea ecranului poate fi facuta cu o singura instructiune REP STOSW).

1.4 Desfasurarea lucrarii

4.1 Se studiaza functiile de acces la disc la nivel BIOS si DOS

4.2 Se studiaza programul FBD.ASM prin prisma avantajelor si dezavantajelor celor 3 metode de scriere pe ecran.

4.3 Se modifica programul astfel incat sa existe si o functie scrie_dos care sa utilizeze variabile pozx_dos si pozy_dos pentru pozitia de inceput a textului.

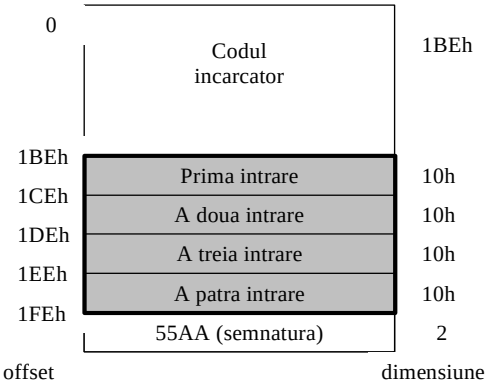
4.4 Sa se analizeze posibilitatea de realizare a unor functii de salvare si restaurare a continutului ecranului la cele 3 nivele (fara a schimba pagina video).

2 Sisteme de fisiere. Partitionarea hdd

2.1 Introducere

Hard disk-ul (HDD) reprezinta cel mai utilizat mijloc de stocare a datelor unui PC, urmand apoi mediile de tip DVD, CD, TAPE, etc. Dintre avantajele HDD fata de celelalte medii de stocare putem enumera capacitatea, viteza si mai nou chiar si mobilitatea. Referitor la capacitate, in ultimii 20 de ani ea a crescut de la 10MB si costuri de 100$/1MB la 100GB si costuri de 0.1$/1MB. Bineinteles ca si viteza de access si rata de transfer au avut o evolutie pozitiva. Putem astfel sa comparam dezvoltarea HDD-urilor cu cea a microprocesoarelor.

Necesitatea partitionarii HDD provine din necesitatea de a avea, pe acelasi HDD, mai multe sisteme de operare. Un alt avantaj este acela ca se pot gestiona mai eficient partitii mai mici.
Tabela partitiilor este o tabela ce se gaseste pe sectorul avind adresa fizica [cilindru=0, cap=0, sector=1] deci primul sector de pe HDD.

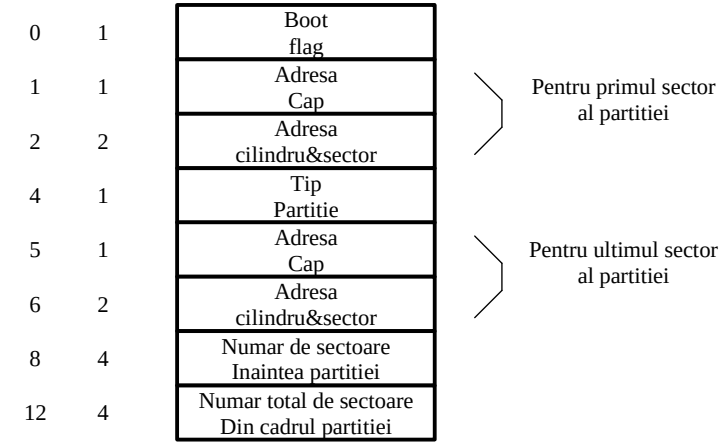


Structura sectorului cu tabela partitiilor
(Partea hasurata este **tabela partitiilor)**

2.2 Tabela de partitii propriu-zisa

Desi riguros tabela partitiilor este doar o tabela in cadrul sectorului [0,0,1] prin abuz de limbaj se foloseste uneori pentru acest sector notatia “tabela partitiilor” in loc de “sectorul tabelei partitiilor” (pierzind din rigoarea exprimarii).

O partitie este descrisa de o **intrare** in tabela partitiilor avind urmatorul format:



Structura unei intrari in tabela partitiilor

- **Boot flag** – este un cimp care indica daca partitia respectiva ete bootabila sau nu (80h – bootabila, 00h – nebootabila).
- **Adresa cap** – indica adresa capului pentru inceputul si sfarsitul partitiei.
- **Adresa cilindru&sector** indica adresa fizica a primului/ultimului sector al partitiei. Din cei 16 biti folositi 10 sunt pentru adresa cilindrului, iar 6 pentru adresa sector.

$$C_1C_2C_3C_4C_5C_6C_7C_8\ C_9C_{10}S_1S_2S_3S_4S_5S_6$$

- **Tip Partitie** indica tipul partitiei respective (Vezi Anexa 1). Campul este esential deoarece un sistem de operare nu acceseaza decat partitiile al caror continut il poate interpreta (il suporta) conform acestui tip.

- **Numar de sectoare inaintea partitiei** – Indica numarul de sectoare aflate inaintea partitiei.
- **Numar total de sectoare in partitie** – Indica numarul total de sectoare pe care il ocupa partitia.

La prima vedere faptul ca se pastreaza inceputul si sfarsitul partitiei folosind adresa cap/cilindru/sector precum si numarul de sectoare inaintea partitiei si numarul de sectoare pe care il ocupa partitia poate parea informatie redundanta. Pentru HDD mici de pana la 4GB este adevarat. Problema apare insa la HDD cu o capacitate mai mare de 4GB unde adresa de tip cap/cilindru/sector caruia ii sunt alocati 3 octeti nu mai poate pastra corect inceputul si sfarsitul partitiilor aflate dupa primii 4GB de HDD.
Pentru a putea utiliza eficient HDD cu capacitate mai mare de 4GB se foloseste tehnologia **LBA** (Adresare logica a blocurilor), sectoarele HDD nemaifiind accesate pe baza adresei cap/cilindru/sector, ci folosind un singur numar care identifica un sector HDD. Se foloseste astfel numarul de sectoare inaintea partitiei, numar caruia ii sunt rezervati 4 octeti si deci se vor putea aloca partitii pe HDD cu capacitati de pana la 2TB.

2.3 Codul incarcator

Pe sectorul tabelei partitiilor se gaseste, pe linga tabela partitiilor propriu-ziza si un cod incarcator. Acesta este incarcat de catre ROM-BIOS la adresa 7C00:0000 si este lansat in executie.
In mare, el executa urmatorii pasi:

- se muta (isi muta propriul cod) la alta adresa (oricum, in acest moment, cu exceptia tablului vectorilor de intrerupere, toata memoria este libera) si face un salt in copie, urmand a se executa in continuare de la aceasta adresa ;
- analizeaza intrarile din tabela partitiilor si cauta o partitie activa (bootabila)
- verifica sa nu mai fie inca o partitie activa
- incarca la 7C00:0000 primul sector al partitiei active (pe baza adresei sale fizice din intrarea corespunzatoare) numit **sector de boot** al partitiei
- verifica semnatura (de sistem) 55AAh din finalul sectorului

- preda controlul codului incarcator de pe sectorul de boot tocmai incarcat printr-un salt :

JMP FAR 7C00:0000

In functie de diferite situatii intilnite se pot genera urmatoarele mesaje:
“Invalid partition table” – tabela este invalida in sensul ca nu este nici o partitie bootabila sau sint mai multe asemenea partitii
“Error loading operating system” – la tentativa de a incarca sectorul de boot al sistemului de operare activ s-a intilnit eroare de citire
“Missing operating sistem” – sectorul de boot citit nu contine semnatura de sistem (55AAh).
Oricare din aceste erori este eroare fatala incarcarea singura alternativa fiind *resetarea* si *incarcarea sistemului de pe discheta* sau *cd* urmata de corectarea situatiei din tabela partitiilor.

Remarcam ca mecanismul de partitionare este **in afara unui sistem de operare anume**, partitularizarea unuia sau altuia din sistemele de operare incepe doar cu sectorul de boot al partitiei respective. Bineinteles tot acest mecanism de partitionare se bazeaza pe “proprietatea” sistemelor de operare *de a nu accesa zone de disc din afara partitiilor proprii* sau pe care nu le cunosc/suporta (desi nu exista nici un mecanism de evitare a acestui lucru).
In cazul programelor de gen BootManager care au o interfata grafica prietenoasa si pun la dispozitie o serie de utilitare este nevoie de mai mult spatiu pe HDD pentru stocarea aplicatiei. Deoarece spatiul alocat codului incarcator este relativ mic, 446 octeti, acestea pun pe primul sector al hdd doar un cod de initializare, restul aplicatiei aflandu-se in cadrul unei partitii si fiind incarcat si lansat in executie de catre codul de incarcare.

2.4 Particularitati ale partitionarii

Dupa cum s-a vazut mai sus, tabela de partitii poate sa contina maxim patru intrari. In majoritatea cazurilor divizarea HDD in patru partitii este suficienta.

Pentru ca totusi numarul de partitii sa nu fie limitat la patru, s-au introdus conceptele de **partitie primara** si **partitie extinsa**.

Partitia primara – este o partitie “normala” care ocupa o intrare in tabela partitiilor si care poate fi facuta bootabila.

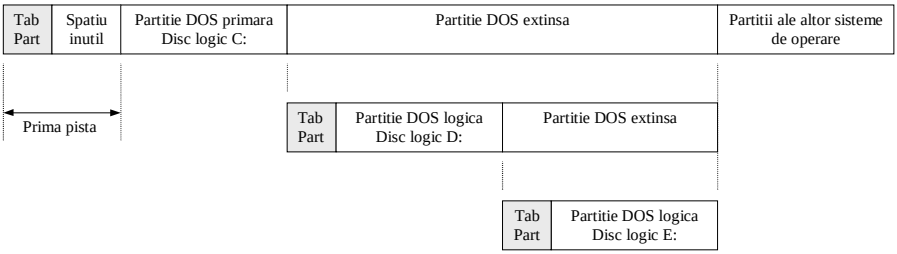
Partitia extinsa – este o partitie care contine la randul ei alte **partitii logice** si care vor fi folosite de catre utilizatori pentru stocarea fisierelor.

Sub sistemul de operare MS-DOS partitionarea prezinta unele particuritati. In acest sens utilitarul **fdisk.exe** destinat partitionarii permite crearea urmatoarelor tipuri de partitii:

Partitie DOS extinsa – in acest caz partitia defineste o zona de disc care va fi tratata in continuare ca si un disc “intreg” avand, pe primul sector, o noua tabela de partitii (dar fara codul incarcator).

Partitie logica in partitia extinsa – se defineste o noua partitie in cadrul spatiului partitiei extinse.

O alta particularitate consta in aceea ca chiar si intr-o partitie extinsa se poate crea o singura partitie logica pe spatiul ramas se construieste (automat) o noua partitie extinsa in care se creaza o partitie logica s.a.m.d. Un exemplu de partitionare a discului cu o partitie DOS primara (accesibila ca si drive C:) si cu doua partitii logice in partitia extinsa (accesibile ca si drive D: si E:) este prezentata in figura.



2.5 Anexa 1 – Identificatori pentru sistemele de fisiere

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 Mb)
05h	Extended Partition (CHS address mode)
06h	FAT16 (CHS address mode, and partition size is larger than 32 Mb)
07h	NTFS
08h	AIX
09h	AIX Bootable
0Ah	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	SFS
50h	Disk Manager
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 incarcator

```
       .MODEL SMALL
SegPart       SEGMENT AT 0
       ASSUME CS:SegPart,DS:NOTHING
;             INCARCATORUL DE PARTITII
;-            ------------------------
;      Prima actiune ce o efectueaza incarcatorul de partitii ste 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 Probleme

Realizati un program care sa afiseze sub forma de tabel informatiile aflate in tabela de partitii. Deoarece pe sistemele de operare Windows NT/2000 accesul la nivel de sector pe HDD nu este permis, se va folosi ca si intrare a programului fisierul Mbr.bin.

Mbr.bin este un fisier in care a fost copiat continutul primului sector al unui HDD cu mai multe partitii.

Pentru o decodificare mai usoara a informatiilor din tabela de partitii se pot folosi urmatoarele 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: In 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 comtaratie cu numarul de octeti aferenti fiecarui membru al structurii. In caz de neconcordanta se vor modifica tipurile membrilor structurii.

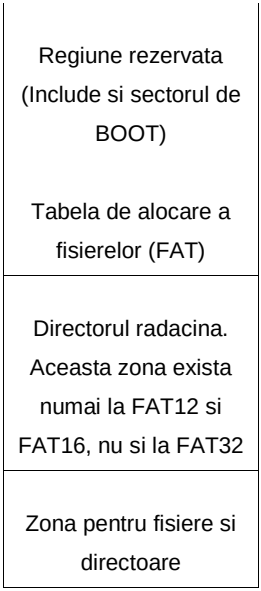
3 Sisteme de fisiere. FAT32

3.1 Introducere

Sistemul de fisiere FAT (File Allocation Table) a fost creat la inceputul anilor ’80 si a fost utilizat in multiplele versiuni ale sistemului de operare MS-DOS. Initial el a fost creat pentru a mentine fisierele aflate pe floppy dik-uri a caror capacitate nu depasea 500K. Bininteles ca in decursul anilor sistemul a fost dezvoltat pentru a suporta discuri de capacitati din ce in ce mai mari, aparand versiunile FAT12, FAT16 si FAT32. Numerele 12, 16 sau 32 reprezinta dimensiunea in biti alocata pentru memorarea adresei unui cluster, dupa cum se va vedea in descrierea de mai jos.

FAT32 a fost introdus si utilizat odata cu lansarea sistemului de operare Windows 95 OSR2. FAT32 a fost creat datorita unei limitari importante a FAT16, si anume crearea de partitii a caror dimensiune nu poate sa depaseasca 2GB.

Structura unei partitii FAT este urmatoarea:



Sectorul de Boot (Bootsector)

Sectorul de boot se afla pe sectorul 0 al partitiei, iar codul continut de acesta este lansat in executie la incarcarea sistemului, in cazul in care partitia este activa si bootabila. Tot in cadrul acestui sector se afla si o zona speciala **BPB** (BIOS Parameter Block), precum si informatii detaliate despre partitie, dupa cum se observa in structura de mai jos

Se considera urmatoarea conventie : numele campurilor care incep cu BPB_ se considera ca facand parte din BIOS Parameter Block, iar cele care incep cu BS_ din boot sector.

Nume camp	Offset	Marime	Descriere
BS_JmpBoot	0	3	Contine codul unei instructiuni de jump neconditionat catre codul de incarcare aflat in sectorul de boot.
BS_OEMName	3	8	Contine numele sistemului de operare care a formatat partitia respectiva ex. (MSWIN4.1). Acest nume nu are o semnificatie aparte.
BPB_BytesPerSector	11	2	Specifica numarul de octeti pe care ii contine un sector al mediului fizic pe care este prezenta partitia. In majoritatea cazurilor valoarea este 512, dar pot fi intalnite si valorile 1024, 2048 sau 4096.
BPB_SecPerClus	13	1	Specifica numarul de sectoare per cluste(Cluster-ul este unitatea de alocare a spatiului in cadrul unei partitii FAT)r. Acest numar poate fi 1, 2, 4, 8, 16, 32, 64, 128. Dimensiunea unui cluster va fi deci: BPB_BytesPerSector * BPB_SecPerClus. Se recomanda ca aceasta dimensiune sa nu depaseasca 32K.

Nume camp	Offset	Marime	Descriere
BPB_RsvdSecCnt	14	2	Specifica numarul de sectoare pe care il ocupa regiunea rezervata (care include si sectorul de boot). In general se intalneste valoarea 32 pentru sistemul de fisiere FAT32, iar pentru FAT12 si FAT16 valoarea este intotdeauna 1.
BPB_NumFATs	16	1	Din motive de securitate a datelor, pentru tabela de alocare a fisierelor se mentine una sau mai multe copii. BPB_NumFATs specifica numarul de tabele. De obicei se intalneste valoarea 2, adica tabela propriu-zisa si o copie a acesteia care va putea fi consultata in cazul in care tabela originala contine date invalide sau nu poate fi citita.
BPB_RootEntCnt	17	2	Contine intotdeauna valoarea 0. Acest camp a fost folosit pentru sistemele FAT12 si FAT16 si specifica numarul de intrari in directorul radacina.
BPB_TotSec16	19	2	Contine intotdeauna valoarea 0. Campul a fost utilizat pentru sistemele FAT12 si FAT16 si continea numarul total de sectoare al partitiei.
BPB_Media	21	1	Specifica tipul de mediu pe care se afla partitia. 0xF8 – pentru discuri fixe (non-removable media) ; 0xF0 – pentru medii mobile (removable media); Alte valori intalnite 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF.

<table>
<tr><th>Nume camp</th><th>Offset</th><th>Marime</th><th>Descriere</th></tr>
<tr><td>BPB_FATSz16</td><td>22</td><td>2</td><td>Contine intotdeauna valoarea 0.
Pentru FAT12 si FAT16 specifica numarul de sectoare ocupate de o tabela de alocare FAT.</td></tr>
<tr><td>BPB_SecPerTrk</td><td>24</td><td>2</td><td>Specifica numarul de sectoare pe pista.</td></tr>
<tr><td>BPB_NumHeads</td><td>26</td><td>2</td><td>Specifica numarul de capete.</td></tr>
<tr><td>BPB_HiddSec</td><td>28</td><td>4</td><td>Specifica numarul de sectoare aflate inaintea partitiei. In cazul in care exista o singura partitie, iar aceasta ocupa tot spatiul disponibil, valoarea ar trebui sa fie 0.</td></tr>
<tr><td>BPB_TotSec32</td><td>32</td><td>4</td><td>Specifica numarul de sectoare pe care le ocupa intreaga partitie.</td></tr>
<tr><td>BPB_FATSz32</td><td>36</td><td>4</td><td>Specifica numarul de sectoare pe care le ocupa o tabela de alocare a fisierelor.</td></tr>
<tr><td>BPB_ExtFlags</td><td>40</td><td>2</td><td>Acesti 16 biti au urmatoarele semnificatii:
<table>
<tr><th>Biti</th><th>Semnificatie</th></tr>
<tr><td>0-3</td><td>Indica numarul tabelei de alocare active. Numerotarea incepe de la 0. Acest numar este luat in considerare numai in cazul in care bitul 7 are valoarea 1.</td></tr>
<tr><td>4-6</td><td>Rezervati.</td></tr>
<tr><td>7</td><td>0 inseamna ca toate tabelele de alocare aferente partitiei sunt updatate la runtime. 1 inseamna ca numai tabela activa este updatata, si anume tabela care este indicata de bitii 0-3.</td></tr>
<tr><td>8-25</td><td>Rezervati.</td></tr>
</table></td></tr>
</table>

Nume camp	Offset	Marime	Descriere
BPB_FSVer	42	2	Specifica veriunea de sistem FAT32. Contine valoarea 0:0, pentru versiunea actuala.
BPB_RootClus	44	4	Specifica numrul cluster-ului unde incepe directorul radacina, si in general are valoarea 2.
BPB_FSInfo	48	2	Specifica numarul sectorului unde se afla structura FSINFO, si are in general valoarea 1.
BPB_BkBootSec	50	2	Specifica numarul sectorului unde se afla o copie a sectorului de boot, doar in cazul in care nu este 0. Se recomanda doar folosirea valorii 6.
BPB_Reserved	52	12	Zona rezervata pentru versiunile ulterioare de FAT32. Ar trebui sa contina in toti octetii valoarea 0.
BS_DrvNum	64	1	Specifica numarul drive-ului si este folosit de catreInt 0x13.
BS_Reserved1	65	1	Octet rezervat. Este utilizat de carte Windows NT. Dupa formatarea pertitiei, ar trebui sa contina valoarea 0.
BS_BootSig	66	1	Semnatura extinsa a sectorului de boot, contine valoarea 0x29, si arata ca urmatoarele 3 campuri sunt prezente.
BS_VolID	67	4	Indica *serial number-ul* partitiei si impreuna cu campul urmator, BS_VolLab ajuta la identificarea mediilor *removable*.
BS_VolLab	71	11	Indica numele partitiei. Acest sir este acelasi cu cel aflat in intrarea de tip volum din directorul radacina.

Nume camp	Offset	Marime	Descriere
BS_FilSysType	82	8	Contine intotdeauna sirul "**FAT32** ". Acest camp nu trebuie folosit pentru a determina versiunea de sistem FAT.
BS_BootCode	90	420	Contine un mic program care este lansat in executie de carte programul aflat pe MBR in cazul in care partitia este selectata ca fiind activa. De aici incepe practic incarcarea sistemului de operare. Acest cod este diferit pentru fiecare sistem de operare in parte.
BS_Signature	510	2	Reprezinta o semnatura a sectorului de boot. Contine valorile 55h in octetul 510 si AAh in octetul 511. Daca incarcam cei doi octeti intr-un registru de 32 biti, orbinem valoarea AA55h.
Spatiu disponibil in cazul in care sectorul fizic are dimensiune mai mare 512 octeti.			

3.2 Tabela de alocare a fisierelor FAT (File Allocation Table)

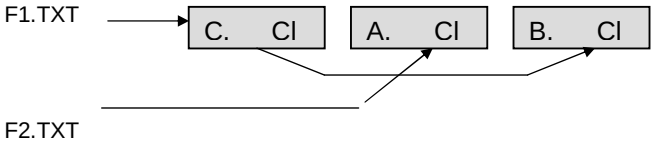
Aceasta tabela este utilizata de catre sistemul de operare pentru a determina pozitia exacta a partilor unui fisier cadrul partitiei. Putem privi aceasta tabela de alocare ca pe un vector de dimensiune ClusterCount.

```
ClusterCount = BPB_TotSec32 / BPB_SecPerClus
```

Fiecare element al acestui vector arata starea efectiva a cluster-ului de pe disc pe care il reprezinta, existand o legarura unu la unu intre un astfel de element al vectorului si cluster-ul de date de pe disc. Dimensiunea exacta a unui cluster este :

```
ClusterSize = BPB_BytesPerSector * BPB_SecPerClus
```

Spatiul alocat continutului fisierelor este impartit pe cluster-e. Cluster-ul este unitatea de alocare a spatiului unei partitii. Un fisier ocupa unul sau mai multe clustere in functie de marimea sa. De remarcat ca un fisier care contine doar un octet ocupa un cluster si nu un octet din spatiul disponibil. In general fisierele nu detin cluster-e continue in cadrul partitiei, cluster-ele ocupate de un fisier fiind intrepatrunse cu ale altui fisier, responsabilitatea integritatii fisierelor ramanand la nivelul informatiilor din FAT.



Fiecarui cluster de pe disc ii corespunde o intrare in FAT. Intrarea corespunzatoare inceputului unui fisier pe disc contine numarul clusterul-ui urmator, formand astfel un lant ce contine clusterele aferente unui fisier, precum si ordinea acestora.

Trebuie retinut ca doar zona de date este impartita logic in cluster-e, deci pozitia primului cluster coincide cu inceputul zonei de date alocate fisierelor si directoarelor.
Mai este deasemenea foarte important faptul ca primele doua intrari in FAT nu au cluster corespondent in zona de date, adica aceasta (zona de date) incepe cu cluster-ul doi.

O intrare in FAT ocupa 32 biti, deci putem intalni valori cuprinse intre 00000000h si FFFFFFFFh, insa o parte dintre aceste valori au semnificatie speciala, si anume :

Valoare	Descriere
00000000h	Cluster liber.
000000001h - FFFFFFF5h	Poate reprezenta numarul urmatorului cluster in lantul de clustere ce descrie un fisier.
FFFFFFF6h - FFFFFFF7h	Specifica faptul ca clusterul respectiv contine unul sau mai multe sectoare defectete.

Valoare	Descriere
FFFFFFFFh	Indica sfarsitul unui fisier in cluster-ul respectiv.

Trebuie mentionat faptul ca primele doua cluster-e sunt rezervate si sunt marcate in FAT cu valorile FFFFFFF8h si FFFFFFFFh. Zona de date pentru fisiere incepe deci de la cluster-ul cu numarul 2.

3.3 Structura de directoare

Un director este un fisier al carui continut este tratat in mod special. Acest continut este impartit logic in parti de 32 octeti, numite intrari in director. Un director ocupa in zona de date unul sau mai multe cluster-e, continue sau discontinue, la fel ca orice alt fisier.

O intrare in director are urmatoarea structura :

Nume	Offset	Marime	Descriere
DIR_Name	0	11	Specifica numele (scurt) al fisierului sau directorului.
DIR_Attr	11	1	Specifica atributele fisierului: • 0x01 - ATTR_READ_ONLY • 0x02 - ATTR_HIDDEN • 0x04 - ATTR_SYSTEM • 0x08 - ATTR_VOLUME_ID • 0x10 - ATTR_DIRECTORY • 0x20 - ATTR_ARCHIVE • Toate - ATTR_LONG_NAME
DIR_NTRes	12	1	Rezervat pentru utilizare de catre Windows NT
DIR_CrtTimeTenht	13	1	Deoarece campul DIR_CrtTime are precizie de 2 secunde, se specifica aici numarul de sutimi de secunda la care a fost creat fisierul. Valorile admise sunt intre 0 si 199.

Nume	Offset	Marime	Descriere
DIR_CrtTime	14	2	Specifica timpul crearii fisierului.
DIR_CrtDate DIR_LstAccDate	16 18	2 2	Specifica data crearii fisierului. Specifica ultima data cand a fost accesat fisierul, in mod scriere sau citire. Nu se specifica si timpul.
DIR_FstClustHI	20	2	Specifica partea mai semnificativa a cuvantului de 32 biti care indica cluster-ul de start al fisierului.
DIR_WrtTime	22	2	Specifica timpul la care s-a scris in fisier ultima data.
DIR_WrtDate	24	2	Specifica data la care s-a scris in fisier ultima oara.
DIR_FstClustLO	26	2	Specifica partea mai putin semnificativa a cuvantului de 32 biti care indica cluster-ul de start al fisierului.
DIR_FileSize	28	4	Infica lungimea in octeti a fisierului.

Campurile de tip data ocupa 16 biti si au urmatorul format:

- 0–4: Ziua (1-31);
- 5–8: Luna (1-12);
- 9–15: Numarul de ani relativ la 1980. Valorile valide sunt de la 0 la 127, reprezentand anii 1980 pana la 2107.

Campurile de tip timp ocupa 16 biti si au urmatorul format:

- 0–4: indica secundele. Deoarece se pot stoca valorile de la 0 la 29, valoarea se va inmulti cu 2. Obtinem deci valori cuprinse intre 0 si 58, deci avem o precizie de 2 secunde pentru acest camp.
- 5–10: arata minutele (0–59).
- 11–15: arata ora (0-23).

Primul octet din campul DIR_Name are semnificatii speciale in cazul in care valoarea lui este:

- 5Eh – ne arata ca intrarea respectiva in director este libera;

- 00h – ne arata ca intrarea in director este libera si ca toate intrarile in director care urmeaza sunt libere.

Citirea fisierelor unei partitii incepe cu localizarea si analizarea continutului directorului radacina. Pentru localizare se foloseste din FAT campul BPB_RootClus care contine numarul cluster-ului de inceput al directorului radacina.

In tabelul de mai jos se observa structura sumara a unei partitii fat, iar in directorul radacina trei intrari, si anume doua fisiere (file1.txt si file2.txt) si un director dir1.

Boot Sector

FAT #1

		3	4	5	FFF	8	10	FFF	FFF	000
000	000	000	000	000	000	000	000	000	000	000
000	000	000	000	FF7	000	000	000	000	000	000
…	…	…	…	…	…	…	…	…	…	…
000	000	000	000	000	000	000	000	000	000	000
000	000	000	000	000	000	000	000	000	000	000

FAT #2

		3	4	5	FFF	8	10	FFF	FFF	000
000	000	000	000	000	000	000	000	000	000	000
000	000	000	000	FF7	000	000	000	000	000	000
…	…	…	…	…	…	…	…	…	…	…
000	000	000	000	000	000	000	000	000	000	000
000	000	000	000	000	000	000	000	000	000	000

Root Dir

f	i	l	e	1				t	x	t	a										t	t	d	d	0	3	0	0	79	0
f	i	l	e	2				t	x	t	a										t	t	d	d	0	6	0	0	3	E8
d	i	r									a										t	t	d	d	0	3	0	0	0	0

Data

3.4 Structura FSINFO

Aceasta structura contine un sumar al informatiilor despre partitie, si este detaliata in tabelul de mai jos :

Nume camp	Offset	Marime	Descriere
FSI_LeadSig	0	4	Contine valoarea 41615252h care arata ca sectorul contine o structura FSINFO.
FSI_Reserved1	4	480	Rezervati. Ar trebui ca toti octetii acestui camp sa contina valoarea 0.
FSI_StrucSig	484	4	Contine valoarea 61417272h. Este o a doua semnatura pentru sectorul ce contine structura FSINFO.
FSI_Free_Count	488	4	Specifica numarul de cluster-e libere ale partitiei. In cazul in care contine valoarea FFFFFFFFh, numarul de cluster-e libere nu este actualizat si trebuie calculat pe baza informatiilor din FAT.
FSI_Nxt_Free	492	4	Specifica numarul cluster-ului de unde trebuie inceputa cautarea pentru un cluster liber. In cazul in care contine valoarea 0xFFFFFFFF, cautarea trebuie inceputa de la cluster-ul 2.
FSI_Reserved2	496	12	Rezervati. Ar trebui ca toti octetii acestui camp sa contina valoarea 0.
FSI_TrailSig	508	4	Contine valoarea 0xAA550000. Semnatura specifica pentru terminarea sectorului FSINFO.

3.5 Probleme

1. Creati structuri de date capabile sa mentina informatiile aferente :
    a. Sectorului de boot ;
    b. BPB ;
    c. FAT ;
    d. Intrare in director .
2. Folosind structurile create anterior afisati continutul:
    a. directorului radacina aferent partitiei FAT32 ;
    b. unui fisier din directorul radacina ;
    c. unui director aflat in directorul radacina.
3. Realizati o aplicatie care implementeaza pentru partitii de tip FAT32 comenzile:
    a. Dir – afiseaza continutul unui director dupa interpretarea intrarilor acestuia;
    b. Cd – schimba directorul curent;
    c. Type – afiseaza continutul unui fisier primit ca parametru, fisier aflat in directorul curent.

Se va folosi ca si intrare in program un fisier ce contine imaginea (doar partea de inceput) a unei partitii valide de tip FAT32.

4 Sisteme de fisiere. EXT2

4.1 Introducere

Extended File System Version 2 (EXT2) este in momentul de fata sistemul de fisiere cel mai utilizat de sistemele de operare Linux si Unix. Acest sistem de fisiere a fost creat in ideea obtinerii unei securitati mai mari a datelor si a unor viteze mai mari in prelcrarea acestora.

In cadrul sistemului de fisiere EXT2 trebuiesc descrisi urmatorii termeni :

- **Bloc –** este un grup de sectoare, de obicei 8, ceea ce inseamna blocuri de 4KO. Cea mai mica dimensiune intalnita este 1KO.
- **Superbloc –** este o zona care contine informatii generale despre partitie.
- **Inod –** este o zona aferenta fiecarui director sau fisier si care descrie pozitia fizica a acestuia pe disc.
- **Grup –** constituie cea mai mare subdiviziune a unei partitii ext2. Fiecare grup contine o copie a superblocului, o tabela de inoduri precum si alte informatii, dupa cum va fi detaliat mai jos.
- **Fragment –** este o subdivizune a unui bloc. In principiu, se folosesc in cazul fisierelor mici, pentru a nu irosi spatiu.

In tabelul de mai jos este prezentata structura unei dischete cu format ext2 :

Nume camp	Offset	Marime (blocuri)	Descriere
Sector de boot	0	1	Zona care poate sa contina codul incarcator al sistemului de operare
Superbloc	1	1	Acest bloc cantine informatiile aferente structurii Superbloc ce va fi descrisa mai jos
Descriptor de grup	2	1	
Bitmap de blocuri	3	1	Harta care indica blocurile libere
Bitmap de inoduri	4	1	Harta care indica inod-urile libere
Tabela de	5	23	Tabela care contine informatii referitoare la

inoduri			toate inodurile grupului.
Zona blocuri de date	28	1412	Zona de blocuri pentru continutul fisierelor si directoarelor.

In tabelul de mai jos este prezentata structura unei partitii de 20MB cu format ext2 :

Nume camp	Offset	Marime (blocuri)	Descriere
Grupul 0			
Sector de boot	0	1	Zona care poate sa contina codul incarcator al sistemului de operare
Superbloc	1	1	Acest bloc cantine informatiile aferente structurii Superbloc ce va fi descrisa mai jos
Descriptor de grup	2	1	Structura descrisa in continuare.
Bitmap de blocuri	3	1	Harta care indica blocurile libere, 1-8192
Bitmap de inoduri	4	1	Harta care indica inod-urile libere, 1-1712
Tabela de inoduri	5	214	Tabela care contine informatii referitoare la toate inodurile grupului.
Zona blocuri de date	219	7974	Zona de blocuri pentru continutul fisierelor si directoarelor.
Grupul 1			
Superbloc	8193	1	Acest bloc cantine informatiile aferente structurii Superbloc ce va fi descrisa mai jos
Descriptor de grup	8194	1	Structura descrisa in continuare.
Bitmap de blocuri	8195	1	Harta care indica blocurile libere, 1-8192
Bitmap de inoduri	8196	1	Harta care indica inod-urile libere, 1-1712

Tabela de inoduri	8197	214	Tabela care contine informatii referitoare la toate inodurile grupului.
Zona blocuri de date	8411	7974	Zona de blocuri pentru continutul fisierelor si directoarelor.
Grupul 2			
…	…	…	…

4.2 Organizare interna

Unitatea principala de clustering a sistemului de fisiere EXT2 este ***grupul***. Divizarea unei partitii in grupuri este facuta in momentul formatarii si poate fi schimbata doar printr-o reformatare.

Structura unui astfel de grup este urmatoarea:

4.2.1 Superbloc

Contine informatii generale despre sistemul de fisiere si nu despre un anumit grup sau fisier. Aceste informatii includ numarul total de blocuri din sistem, ultima data cand a fost verificat pentru erori, etc. Primul superbloc este cel mai important, deoarece acesta este citit prima data cand sistemul de fisiere va fi montat. Deoarece informatia continuta este cruciala in procesul de montare a partitiei, o copie a acestuia este mentinuta la nivelul fiecarui grup existent, astfel ca in momentul in care informatia este corupta, sa poata fi restaurata din una dintre aceste copii redundante (prin comanda *e2fsck pe sistemele Linux*).

Structura superbloc :

Nume camp	Offset	Marime (octeti)	Descriere
s_inodes_count	0	4	Contine numarul total de inoduri
s_blocks_count	4	4	Contine numarul total de blocuri
s_r_blocks_count	8	4	Contine numarul de blocuri rezervate
s_free_blocks_count	12	4	Contine numarul de blocuri libere
s_free_inodes_count	16	4	Contine numarul de inoduri libere
s_first_data_block	20	4	Contine numarul primului bloc liber care

Nume camp	Offset	Marime (octeti)	Descriere
			poate fi folosit pentru a memora date.
s_log_block_size	24	4	Contine o valoare care ajuta la calcularea lugimii unui bloc, adica: s_block_size = 1024 << s_log_block_size
s_log_frag_size	28	4	Contine o valoare care ajuta la calcularea lugimii unui fragment, adica: s_frag_size = 1024 << s_log_frag_size daca s_log_frag_size este pozitiv, iar daca nu: s_frag_size = 1024 >> -s_log_frag_size
s_blocks_per_group	32	4	Contine numarul de blocuri ale unui grup
s_frags_per_group	36	4	Contine numerul de fragmente ale unui grup
s_inodes_per_group	40	4	Contine numarul de inoduri ale unui grup
s_mtime	44	4	Specifica data cand a fost montata ultima data partitia. Contine numarul de secunde trecute de la 1 ianuarie 1970.
s_wtime	48	4	Specifica data cand a fost scrisa ultima data partitia. Contine numarul de secunde trecute de la 1 ianuarie 1970.
s_mnt_count	52	2	Contine un counter care se incrementeaza de fiecare data cand partitia este montata. Cand valoarea ajunge la s_max_mnt_count, se face o verificare a partitiei, iar counterului i se atribuie valoarea 0.
s_max_mnt_count	54	2	Specifica numarul maxim de montari ale unei partitii, dupa care se face o verificare a acesteia.
s_magic	56	2	Contine semnatura: 0xEF53
s_state	58	2	In momentul montarii unei astfel de

Nume camp	Offset	Marime (octeti)	Descriere
			partitii, se scrie aici valoarea 1, iar cand se demonteaza se inscrie valoarea 2. In cazul unui eveniment neprevazut (reset) valoarea ramane 2, si deci se poate determina la urmatoarea montare faptul ca partitia trebuie verificata deoarece nu a fost demontata corect.
s_errors	60	2	Specifica comportamentul in caz de eroare la montare: - 1 – erorile se ignora; - 2 – partitia se monteaza read-only; - 3 – incarcarea sistemului se opreste cu mesajul “kernel panic”.
s_minor_rev_level	62	2	Numarul minor al reviziei
s_lastcheck	64	4	Specifica data cand a fost verificata partitia ultima data. Contine numarul de secunde trecute de la 1 ianuarie 1970.
s_checkinterval	68	4	Specifica timpul maxim intre doua verificari ale partitiei. Contine numarul de secunde trecute de la 1 ianuarie 1970.
s_creator_os	72	4	Contine un identificator al sistemului de operare care a creat partitia.
s_rev_level	76	4	Numarul reviziei.
s_def_resuid	80	2	Specifica id-ul implicit al utilizatorului care are drepturi asupra blocurilor rezervate. De obicei este id-ul utilizatorului root.
s_def_resgid	82	2	Specifica id-ul implicit al grupului care are drepturi asupra blocurilor rezervate. De obicei este id-ul grupului root.
s_first_ino	84	4	Contine numarul primului inod nerezervat.

Nume camp	Offset	Marime (octeti)	Descriere
s_inode_size	88	2	Marimea tabelei de inoduri.
s_block_group_nr	90	2	Numarul blocului care contine acest superbloc
s_feature_compat	92	4	
s_feature_incompat	96	4	
s_feature_ro_compat	100	4	
s_uuid	104	16	
s_volume_name	120	16	Contine numele partitiei
s_last_mounted	136	64	Contine calea unde a fost montata ultima data partitia.
s_algo_bitmap	200	4	Specifica modul de compresie al partitiei.
s_prealloc_blocks	204	1	Specifica numarul de blocuri prealocate in momentul crearii unui fisier.
s_prealloc_dir_blocks	205	1	Specifica numarul de blocuri prealocate in momentul crearii unui director.
-padding-	206	2	Aliniere pentru cuvinte de 32biti.
s_journal_uuid	208	16	
s_journal_inum	224	4	
s_journal_dev	228	4	

s_last_orphan• 232• 4• • • -reserved-• 236• 788• • •

4			236
		-reserved-	788

4.2.2 Descriptorul de grup

Urmatorul bloc din structura unui grup este alocat unui descriptor de grup care contine informatii referitoare la grupul respectiv. In cadrul fiecarui grup exista un pointer la ***tabela de inoduri*** si harta de biti pentru alocarea inodurilor si cea a blocurilor de date. O harta de alocare este o simpla lista de biti prin care se marcheaza cu 1 inodurile/blocurile de date ocupate si cu 0 inodurile/blocurile de date libere. Se pot determina astfel cu usurinta inodurile/blocurile de date care mai pot fi alocate in sistemul de fisiere.

Structura descriptor de grup :

Nume camp	Offset	Marime (octeti)	Descriere
Bg_block_bitmap	0	4	Specifica numarul primului bloc unde se afla harta de biti pentru alocarea blocurilor
Bg_inode_bitmap	4	4	Specifica numarul primului bloc unde se afla harta de biti pentru alocarea inodurilor
bg_inode_table	8	4	Specifica numarul primului bloc unde se afla tabela de inoduri.
bg_free_blocks_count	12	2	Indica numarul total de blocuri libere din cadrul grupului.
bg_free_inodes_count	14	2	Indica numarul total de inoduri libere din cadrul grupului.
bg_used_dirs_count	16	2	Indica numarul total de inoduri alocate directoarelor in cadrul grupului.
bg_pad	18	2	Aliniere pentru cuvinte de 32biti.
bg_reserved	20	12	Rezervati pentru implementari ulterioare.

4.2.3 Harta de biti pentru blocuri (Block bitmap)

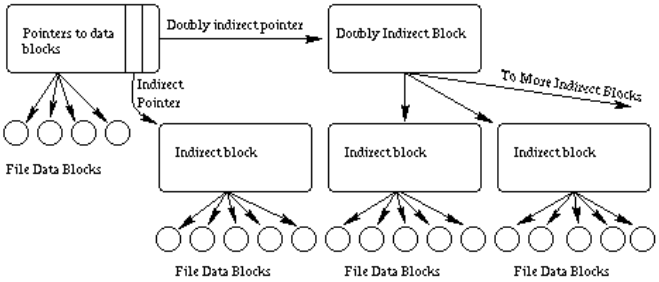
Este o structura care incepe in blocul bg_block_bitmap si fiecare bit specifica starea blocului corespunzator in cadrul grupului, valoarea 0 marcand un bloc liber, iar valoarea 1 un bloc ocupat.

4.2.4 Harta de biti pentru inoduri (Inode bitmap)

Este o structura care incepe in blocul bg_inode_bitmap si fiecare bit specifica starea inodului corespunzator in cadrul grupului, valoarea 0 marcand un inod liber, iar valoarea 1 un inod ocupat.

4.2.5 Tabela de inoduri

Fiecare fisier de pe disc are asociat un singur **inod**. Inodul contine informatii importante despre fisierul in cauza - data creerii si modificarii, permisiuni asupra fisierului, tipul acestuia (fisier normal, director sau un *device*), proprietarul (*owner*-ul) si unde anume este acesta localizat pe disc. Datele continute de fisierul asociat nu sunt stocate in inod, ci in blocuri pe disc. Blocurile de date au marimi fixe, specificate la momentul formatarii, putand avea marimi de 1024, 2048, 4096 sau 8192 octeti. Astfel in fiecare inod sunt 15 pointeri catre blocuri de date; ceea ce nu inseamna insa ca marimea unui fisier este restrictionata la 15 blocuri fizice. Acest lucru se realizeaza prin *indirectare*. Primii 13 pointeri referentiaza blocuri fizice de date, al 14-lea pointer este numit *pointer de indirectare*, deoarece prin el se referentiaza un bloc de pointeri catre blocuri fizice de date. Al 15-lea pointer este numit *pointer de dublu - indirectare* si prin el se referentiaza un bloc de indirectare ce contine pointeri catre blocuri ce referentiaza blocuri de date.



Aceasta schema de organizare permite un acces direct si rapid la fisiere mici (continute in 13-14 blocuri) si in acelasi timp existenta unor fisiere extrem de mari cu doar cateva accese suplimentare.

Tabela de inoduri este o inlantuire de structuri de cate 128 de octeti (marimea unui inod).

Structura unui inod este prezentata in tabelul de mai jos :

Nume camp	Offset	Marime (octeti)	Descriere
i_mode	0	2	Specifica tipul fisierului.
i_uid	2	2	Primii 16 biti mai nesemnificativi din identificatorul detinatorului (owner) fisierului.
i_size	4	4	Specifica marimea fisierului in octeti.
i_atime	8	4	Data cand a fost accesat fisierul.
i_ctime	12	4	Data crearii fisierului.
i_mtime	16	4	Data modificarii fisierului.
i_dtime	20	4	Data stergerii fisierului.
i_gid	24	2	Primii 16 biti mai nesemnificativi din identificatorul grupului (group) fisierului.
i_links_count	26	2	Valoarea acestui camp este incrementata de fiecare data cand o intrare a unui director pointeaza spre acest inod. Cand o astfel de intrare este stearsa, valoarea campului este decrementata. Astfel mai multe fisiere pot pointa spre aceeasi zona de continut.
i_blocks	28	4	Specifica numarul de blocuri aferente fisierului.
i_flags	32	4	Specifica modul de acces asupra fisierului.
osd1	36	4	Valoare specifica sistemului de operare care a formatat partitia.
i_block	40	60	Acest camp contine valorile celor 15 pointer a cate 4 octeti fiecare, catre blocuri de date, dupa cum a fost descris la inceputul subcapitolului.
i_generation	100	4	Specifica versiunea fisierului.
i_file_acl	104	4	Specifica lista de control a aceeselor spre fisier.

Nume camp	Offset	Marime (octeti)	Descriere
i_dir_acl	108	4	Specifica lista de control a aceeselor spre director.
i_faddr	112	4	Specifica numarul fragmenutului din ultimul bloc alocat fisierului. Numai in cazul in care dimensiunea fragmentului nu este aceeasi cu dimensiunea blocului.
osd2	116	12	Valoare specifica sistemului de operare care a formatat partitia.

Marime fisier (octeti)	0-768	769-1.5K	1.5K - 3K	3K - 6K	6K-12K	12K si peste
Aparitie (%)	38.3	19.8	14.2	9.4	7.1	10.1

Frecventa de aparitie a diferitelor marimi de fisiere.

Inodurile sunt stocate in **tabela de inoduri** referentiata de descriptor-ul de grup. Marimea tabelei de inod-uri poate fi specificata doar la formatare schimbarea acesteia fiind posibila doar printr-o reformatare. Cu alte cuvinte, numarul maxim de fisiere din sistem este setat tot la formatare.

Dupa cum se poate observa din structura de mai sus, continutul fisierelor ar putea fi citit corect folosind numai tabela de inoduri, fara a folosi tabela de directoare. Aceasta modalitate este folosita (prin acces manual la informatiile din partitie) in cazul recuperarii datelor de pe o partitie defecta.

4.2.6 Structura de directoare

Directoarele sunt niste fisiere cu un format special, in care sunt memorate numele fisierelor continute. Fiecare director contine o lista de intrari (*directory entries*) in care se asociaza un nume de fisier cu un inod si contin: numarul inodului, lungimea numelui fisierului si numele efectiv. Numele unui fisier poate avea pana la 255 caractere. Directorul ***root*** (***/***) este intotdeauna referentiat de

inodul 2, putand fi astfel citit de sistem la montare. Subdirectoarele sunt implementate prin stocarea numelui in *campul de nume* si a inod-ului subdirectorului in campul *inod*. Link-urile sunt implementate prin stocarea **aceluiasi** inod in intrari diferite. Directoarele speciale "**.**" si "**..**" sunt implementate prin stocarea numelor "**.**" si "**..**" in campurile de nume aferente si a inodurilor directorului curent si a directorului parinte in campurile de inod-uri.

Structura unei intrari in director este urmatoarea :

Nume camp	Offset	Marime (octeti)	Descriere
inode	0	4	Numarul inod-ului aferent fisierului sau directorului.
rec_len	4	2	Specifica lungimea intrarii in director.
name_len	6	1	Specifica lungimea numelui fisierului.
file_type	7	1	Specifica tipul fisierului.
name	8	1-255	Numele fisierului. Lungimea acestui camp este cea specificata in name_len.

Se observa ca intrarile intr-un director nu au toate aceeasi lungime ca la sistemul de fisiere prezentat anterior (FAT32).

4.2.7 Accesarea unui fisier

Sa presupunem ca un proces doreste sa acceseze un fisier de pe disc (/tmp/readme.txt). Fiecarui proces ii este asociat un director de lucru curent. Orice cale de fisier care nu incepe cu **/** este rezolvata pornindu-se de la acest director curent. Caile care insa incep cu **/** sunt rezolvate incepand cu directorul *root*. Numele primului subdirector, *tmp* este cautat in directorul radacina si ii este citit inodul corespunzator. In pasul urmator se verifica permisiunile asupra subdirectorului respectiv. Daca procesul are drepturi de acces, subdirectorul devine director cuent si este cautat numele fisierului dorit. Daca acesta este gasit, i se citeste inodul corespunzator si sunt clarificate din nou drepturile de acces de data aceasta asupra fisierului. Daca este permisa accesarea, procesul va putea incepe accesarea fisierului. Acest proces nu va trebui reluat in cazul in care se

lucreaza cu datele fisierului, ci informatiile despre acesta se iau in acest moment direct din inod.

4.3 Probleme

4. Creati structuri de date capabile sa mentina informatiile aferente unei partitii ext2 :
    a. Superbloc
    b. Descriptor de grup
    c. Bitmap de blocuri
    d. Bitmap de inoduri
    e. Inod
    f. Tabela de inoduri
5. Realizati o aplicatie care implementeaza pentru partitii de tip Ext2 comenzile:
    a. ls – afiseaza continutul unui director dupa interpretarea intrarilor acestuia;
    b. cd – schimba directorul curent;
    c. cat – afiseaza continutul unui fisier primit ca parametru, fisier aflat in directorul curent.

Se va folosi ca si intrare in program un fisier ce contine imaginea unei partitii valide de tip Ext2.

5 Comutare de task-uri. Comutare nonpreemptiva

5.1 Introducere

In functie de algoritmul de planificare a executiei proceselor o prima clasificare imparte sistemele de operare multitasking in doua categorii: nonpreemptive si preemptive.

In sistemele **nonpreemptive** posibilitatea intrarii in rulare a unui proces apare doar in momentul in care procesul curent cedeaza controlul UC sistemului de operare. Aceasta politica de alocare a timpului procesor implica simplificarea gestiunii resurselor partajate de mai multe procese, relaxarea anumitor conditii impuse codului sistemului de operare (reentranta nu este necesara) dar are ca efect si o diminuare a stabilitatii si robustetei sistemului de care sunt raspunzatoare procesele utilizator.

Sistemele de operare **preemptive** se implica direct in dispecerizarea proceselor. Acestora li se acorda controlul UC in functie de starea in care se afla (suspendat, in asteptarea eliberarii unei resurse sau producerii unui eveniment, gata de executie etc.), de prioritate, de timpul de utilizare recenta a procesorului etc. Avantajele rezultate sunt legate de eficienta utilizarii CPU si de cresterea robustetei sistemului.

5.2 Gestiunea taskurilor

Programul prezentat modeleaza gestionarea taskurilor intr-un sistem de operare multitasking nonpreemptiv. Taskurile sunt reprezentate prin functii C, iar suportul pentru multitasking este asigurat de lista dublu inlantuita de obiecte de tip *Task*. Acestea contin date referitoare la contextul taskului precum si informatii necesare comutarii (vezi *task.h*).

Din punctul de vedere al sistemului modelat taskurile trec prin trei faze:

A. Initializare

Functia *Task* fork(void (*f))* (din *task.cpp*) face legatura dintre functia utilizator *void f()* si mecanismul de comutare implementat in program prin crearea dinamica a unui nou obiect de tipul *Task* si inserarea lui in lista de taskuri. Constructorul acestui obiect va aloca o stiva proprie care va contine intregul context al taskului reprezentat aici de:

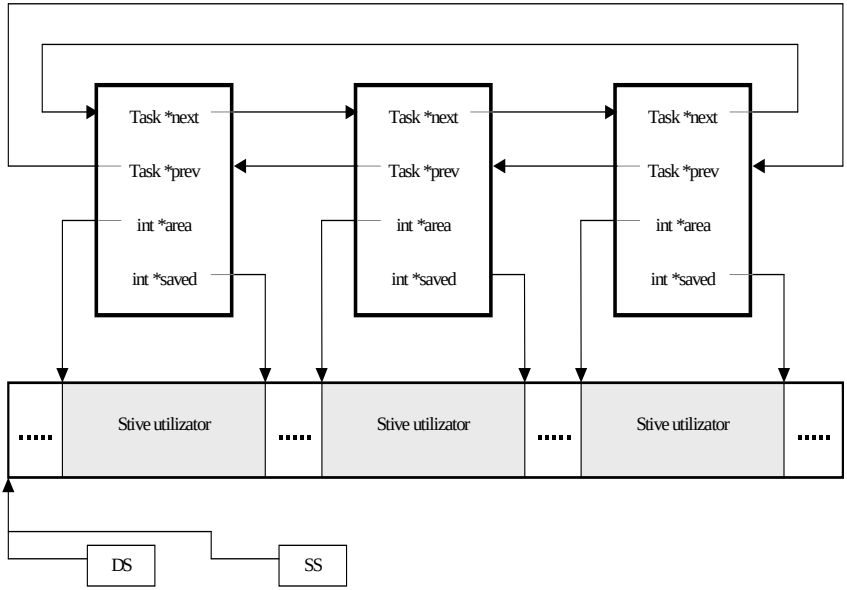
- adresa functiei de terminare a taskului (*kill()*) apelata la incheierea functiei asociate (*f()*); asigura eliberarea structurilor de date ale sistemului alocate taskului care se termina.
- adresa instructiunii care se va executa la primirea controlului de catre task. Initial este impusa prima instructiune a functiei *f()*, apoi adresa este completata / regasita prin mecanismul de apel / revenire din procedura.
- registrul BP - salvat / restaurat automat de coompilatorul C la intrarea / revenirea dintr-o functie.
- registrii SI si DI - compilatorul C presupune ca un apel de functie nu modifica registrii SI, DI si va genera cod pentru salvarea / restaurarea pe stiva daca sesizeaza alterarea lor intr-o functie.

B. Comutare

Se face atunci cand functia asociata unui task cedeaza controlul prin apelarea (directa sau indirecta) a functiei *pause()*. Aceasta realizeaza transferul controlului altui task prin comutarea contextului. Noua stiva va contine adresa de revenire in noul task.

C. Terminare

La terminarea executiei functiei asociate unui task adresa de revenire va fi in functia *kill()*: dezleaga taskul din lista, elibereaza memoria (stiva) si asigura preluarea controlului de alt task.



Observatie:

Intr-un sistem de operare nonpreemptiv nu se pun probleme de reentranta a apelurilor sistem (aici functiile de biblioteca C): transferul controlului se face in puncte stabilite de sistem.

5.3 Probleme

1. Analizati sursele programului: *TASK.H*, *WINDOW.H*, *TASK.CPP*, *WINDOW.CPP*, *MAIN.CPP*. Remarcati obligativitatea cedarii controlului de catre taskuri si modalitatea in care intra in executie un alt task.
2. Scrieti un task care sa afiseze permanent timpul (in secunde) de cand a fost lansat. Remarcati (unica) diferenta fata de un program similar scris pentru un sistem monotasking.
3. Scrieti un task care la fiecare 10 secunde lanseaza un task ce ruleaza 5 secunde si se termina.

6 Comutare de task-uri. Comutare preemptiva

6.1 Introducere

Sistemele de operare preemptive folosesc mecanismul de intreruperi si ceasul de timp real pentru a rula periodic o rutina ***dispecer*** care are rolul de a planifica executia proceselor conform unui algoritm propriu. Acest algoritm tine seama de starea in care se afla procesele, de timpul de utilizare recenta a procesorului, de faptul ca proceselor le poate fi asociata o anumita prioritate.

Dispecerul forteaza intreruperea executiei unui proces urmand a-i reda controlul ulterior. Pentru ca procesul intrerupt sa poata evolua corect (transparent mecanismului de dispecerizare) la relansarea sa in executie ***trebuie refacut contextul*** din momentul intreruperii. Contextul procesului este reprezentat deci de acele informatii de care are nevoie pentru continuarea executiei in momentul planificarii:

- informatii referitoare la resursele masinii inaintea cedarii controlului - formeaza ***contextul fizic*** : registrii procesorului, indicatorii de conditii.
- ***contextul logic***: informatii necesare gestionarii resurselor locale procesului - de exemplu : tabele de pagini, descriptori de fisiere, coada de mesaje asociata procesului, variabile de mediu, director curent etc. Tot aici se pot include si zonele de cod, date si stiva ale procesului.

Sistemul de operare manipuleaza aceste informatii in cadrul rutinei de planificare fara a se baza pe colaborarea proceselor; eventualele erori de programare din acestea nu afecteaza stabilitatea sistemului.

6.2 Gestiunea taskurilor

In *KERNEL.C* rutina de tratare a intreruperii 8 (corespunzatoare IRQ 0 - ceasul de timp real) este rescrisa pentru a apela un dispecer care cedeaza pe rand controlul procesorului unor functii utilizator inscrise intr-o lista circulara (care include si functia *main()*). “Contextul fizic” al acestor functii este continut in cadrele

de stiva tipice declaratiei unei functii C de tip *interrupt*. Codul generat de compilator pentru acest tip de functii debuteaza cu succesiunea de instructiuni:

```
PUSH AX
PUSH BX
PUSH CX
PUSH DX
PUSH ES
PUSH DS
PUSH SI
PUSH DI
PUSH BP
MOV  BP, DGROUP
MOV  DS, BP
```

(aici DGROUP este grupul format din segmentele _DATA - date initializate - si _BSS - date neinitializate). Evident, la iesirea din corpul functiei registrii vor fi restaurati prin introducerea automata de instructiuni POP corespunzatoare.

Acest cadru de stiva obtinut prin salvarea explicita a registrilor generali (AX, BX, CX, DX), de index si pointer (SI, DI, BP) si de segment (DS, ES) este precedat de valorile registrilor pointer de instructiuni (IP), de segment de cod (CS) si de indicatori de conditie (FLAGS) din momentul producerii intreruperii. Ultimii trei registrii sunt salvati pe stiva prin mecanismul de acceptare a intreruperilor. Contextul functiei intrerupte fiind salvat in cadrul de stiva, rolul dispecerului implementat in rutina de tratare a intreruperii se reduce la comutarea ciclica a stivelor asociate functiilor prin salvarea / restaurarea registrilor SS, SP in / din lista *STKF[]*.

6.3 Probleme

3.1. Analizati sursa programului *KERNEL.C*.

3.2. Justificati folosirea secventei :

sprintf(...);

afisVIDEO(...); in functiile *funX()*.

Ce se intampla daca se inlocuieste cu un apel *printf()* ?

Observati efectul declararii globale a variabilei *offset*.

3.3. Implementati un mecanism simplu de prioritati care sa lase controlul unei functii un numar de cicli egal cu prioritatea asociata.

7 Procese

7.1 Introducere

Un process poate fi vazut ca o instanta a unei aplicatii. El este alcatuit din doua parti principale:

- Un *obiect kernel*, pe care sistemul de operare il va folosi pentru a controla procesul, si in care vor fi pastrate informatii statistice despre proces;
- Un *spatiu de adrese*, care va contine intregul cod si datele aferente procesului. Acest spatiu de adrese va contine si memoria dinamica alocata.

Deoarece procesele sunt inerte pentru a putea executa cod, un proces trebuie sa contina cel putin un fir de executie (*thread*). Acest fir, numit si *fir principal de executie* al programului, are responsabilitatea executarii codului continut in spatiul de adresare aferent procesului. Firul principal al aplicatiei este creat de catre sistemul de operare la lansarea in executie a aplicatiei.

Nu este o regula ca un proces sa detina un singur fir de executie, de fapt fiind posibil ca un proces sa lanseze in executie mai multe astfel de fire, acestea ruland “simultan”.

In momentul in care toate firele unui proces se termina, procesul va fi automat terminat de catre sistemul de operare.

O aplicatie Windows, pentru a putea fi lansata in executie, trebuie sa contina ca si punct de start una din urmatoarele patru functii:

```
int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow
);
```

```
int WINAPI wWinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LWPSTR lpCmdLine,
  int nCmdShow
);
```

```
int __cdecl main(
        int argc,
        char_t *argv[ ],
        char_t *envp[ ]
        );
int __cdecl wmain(
       int argc,
       wchar_t *argv[ ],
       wchar_t *envp[ ]
       );
```

7.2 Tipuri de aplicatii (procese)

Sistemele de operare Windows suporta doua tipuri de aplicatii:

- Aplicatii cu o interfata grafica (GUI)
  Acestea sunt aplicatii care contin elemente grafice in definirea interfetei cu utilizatorul (ferestre, butoane, etc.)
- Aplicatii de consola (CUI)
  Sunt aplicatii de mod text, care ruleaza in consola sistemului de operare.

7.3 Crearea unui proces

Pentru a crea un proces se foloseste functia:

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Unde:

- `lpApplicationName` – numele aplicatiei care va fi lansata in executie;
- `lpCommandLine` – sir care contine linia de comanda;
- `lpProcessAttributes` – atribute de secutitare pentru procesul creat. Poate lua si valoarea NULL;
- `lpThreadAttributes` – atribute de secutitare pentru firul principal al procesului creat. Poate lua si valoarea NULL;
- `dwCreationFlags` - specifica modul cum va fi creat procesul, poate o combinatie (OR) dintre valorile: CREATE_SUSPENDED, CREATE_NEW_CONSOLE, CREATE_NO_WINDOW, ...
- `lpStartupInfo` – pointeaza catre o structura de tip STARTUPINFO in care se poate specifica printre altele dimensiunea si pozitia ferestrei aplicatiei, precum si daca aceasta va fi sau nu vizibila pe ecran.
- `lpProcessInformation` – pointer catre o structura de tip PROCESS_INFORMATION unde functia CreateProcess va specifica informatii despre procesul creat.

7.4 Terminarea unui proces

O modalitate de a termina un proces este apelarea functiei:

VOID ExitProcess(UINT exitCode);

Aceasta functie poate fi apelata din interiorul procesului, de oricare dintre firele acestuia.

A doua posibiliate de a termina un proces este apelul functiei

BOOL TerminateProcess(HANDLE hProcess, UINT fuExitCode);

Functia poate fi apelata de orice process, daca cunoaste handler-ul procesului pe care doreste sa il termine.

Orice proces se va termina prin „moarte" naturala in momentul in care toate firele lui se termina.

7.5 Handlerul instantei de proces

La fiecare creare a unui nou proces, sistemul de operare va crea un identificator unic, numit *handler* sau *handle*, pe care sistemul il va asigna acestuia. Acest identificator unic va fi transmis ca si prim argument, functiei principale a aplicatiei *(w)WinMain.* De obicei acest identiificator este folosit pentru incarcarea diferitelor resurse necesare aplicatiei.

Documentatia platformei Windows precizeaza ca anumite functii necesita ca si parametru de tip *HMODULE:*

```
DWORD GetModulefileName(
        HMODULE hinstModule,
        PTSTR pszPath,
        DWORD cchPath
);
```

In realitate cele doua tipuri sunt aceleasi putand fi folosite oricand unul in locul celuilalt. Sunt totusi doua tipuri diferite, deoarece in cazul sistemelor Windows pe 16 biti, cele doua tipuri identifica lucruri diferite.

Valoarea efectiva a parametrului *hinstExe* este adresa de baza de memorie de la care a fost incarcata imaginea fisierului executabil, in spatial de adresare al procesului. Aceasta adresa de baza este determinate de catre link-editor.

Functia *GetModuleHandle* returneaza handlerul/adresa de baza de unde fisierul executabil a fost incarcat in spatial de adresare al procesului.

HMODULE GetModuleHandle(PCTSTR pszModule);

Parametrul acestei functii va fi un sir de caractere terminat cu 0, ce va contine numele fisierului executabil sau dll ce urmeaza a fi executat. Daca sistemul va gasi fisierul cu numele respectiv, functia va returna adresa de baza de unde imaginea fisierului va fi incarcata, altfel *NULL*.

7.6 Linia de comanda a procesului

La momentul creerii unui proces, acestuia ii este asignata o linie de comanda ce va contine numele fisierului ce urmeaza a fi executat si un sir de caractere terminat cu 0. La momentul rularii in executie a programului, este citit numele fisierului dupa care pointerul este deplasat la ceea ce a mai ramas din linia de comanda (*pszCmdLine*). Este important de notat faptul ca *pszCmdLine* este intotdeauna un ANSI string.

Obtinerea pointerului catre intreaga linie de comanda se face prin apelarea functiei G*etCommandLine*, ce va intoarce un pointer catre un buffer ce va contine intreaga linie de comanda.

7.7 Variabilele de mediu ale unui process

Fiecare proces are un bloc de variabile de mediu asociat in spatial de adresare al procesului. Fiecare astfel de bloc va contine intrari de forma:

VarName1=VarValue1\0

VarName2=VarValue2\0

VarName3=VarValue3\0

.

.

VarNameX=VarValueX\0

\0

Prima parte a fiecarui sir de caractere o constituie numele variabilei de mediu, urmat de caracterul = si de valoarea variabilei respective. Toate intrarile in blocul de variabile de mediu trebuie sa fie ordonate dupa numele variabilelor.

7.8 Probleme

1. Scrieti un program care sa lanseze in executie ca si proces separat, programul primit ca si parametru prin numele sau. Se poate folosi si o fereastra de dialog unde utilizatorul introduce intr-un camp de editare numele aplicatiei pe care doreste sa o ruleze.

Se vor utiliza functiile :

- `CreateProcess`
- `GetStartupInfo`
- `GetLastError`

2. Scrieti un program care sa afiseze informatii utile despre toate procesele active la un moment dat in sistemul de operare, asemanator cu programul TaskManager. Programul va trebui sa puna la dispozitia utilizatorului o modalitate de a termina unul dintre procesele afisate.

Se vor utiliza functiile :

- `Process32First`
- `Process32Next`
- `CreateToolhelp32Snapshot`
- `TerminateProcess`

8 Fire de executie

8.1 Notiuni introductive

In cadrul unui sistem de operare exista la un moment dat o serie de procese care sunt definite de obicei ca instante ale unor programe. Procesele sunt inerte, adica ele nu au capabilitatea de e executa cod si de aceea este nevoie de crearea cel putin a unui fir pentru executia codului aplicatiei. Putem privi deci procesul ca un container de fire de executie care ruleaza in paralel*, sistemul de operare atribuind controlul unitatii centrale tuturor firelor existente, in functie de anumite prioritati si numarul procesoarelor existente.

Conform celor amintite anterior, in realizarea unei aplicatii, chiar daca programatorul nu creaza explicit fire de executie, in momentul rularii aplicatia respectiva va avea cel putin un fir creat de catre sistemul de operare, si anume firul principal de executie fara de care aplicatia nu ar primi niciodata controlul asupra CPU, deci nu ar putea executa instructiuni.

In general firele de executie se creaza cand mai multe sarcini ale aceleiasi aplicatii pot fi rezolvate in paralel. Spre exemplu, in cadrul unui editor de texte unul dintre fire se poate ocupa de interactiunea cu utilizatorul, in timp ce altul verifica textul din punct de vedere grammatical, astfel incat nu mai este nevoie ca utilizatorul sa foloseasca optiunea respectiva din meniu.

Totusi, in anumite situatii crearea mai multor fire atrage dupa sine anumite probleme. In cazul editorului de texte daca am realiza inca un fir care s-ar ocupa de tiparirea documentului pentru ca utilizatorul sa nu astepte terminarea acestei operatii ar putea aparea probleme deoarece textul se modifica in timp ce este tiparit. Problema se poate rezolva prin salvarea temporara a documentului iar firul de tiparire va folosi aceasta copie.

Este de dorit ca o aplicatie sa aiba cat mai multe fire deoarece in general procesorul unui calculator nu este ocupat 100% tot timpul. Totusi acest lucru nu poate fi considerat ca o regula, ramanand la latitudinea programatorului numarul de fire si utilitatea acestora in cadrul propriei aplicatii.

8.2 Crearea firelor de executie

Fiecare fir de executie are un punct de pornire care este o functie cu urmatorul prototip:

```
DWORD WINAPI ThreadFunc(LPVOID pvParam) {
.
.
.
return dwResult;
}
```

Functia contine codul aferent firului respectiv (bineinteles ca se pot face din aceasta functie apeluri carte alte functii) si in momentul in care aceasta se termina se spune ca firul s-a terminat prin “moarte naturala”.

Pentru crearea unui fir de executie exista in API-ul WIN32 functia:

```
HANDLE CreateThread(
      PSECURITY_ATTRIBUTES psa,
      DWORD cbStack,
      PTHREAD_START_ROUTINE pfnStartAddr,
      PVOID pvParam,
      DWORD fwdCreate,
      PDWORD pdwThreadID);
```

psa – este un pointer catre o structura de tipul PSECURITY_ATTRIBUTES. De obicei se paseaza valoarea NULL pentru acest parametru, caz in care o valoare implicita va fi folosita.

cbStack – specifica dimensiunea stivei care va putea fi folosita de catre firul respectiv. O valoare de zero pentru acest parametru va determina folosirea valorii implicite.

pfnStartRoutine – valoarea acestui parametru indica adresa functiei firului, adica punctual de start al acestuia. Acest parametru poate fi considerat cel mai important si valoarea lui nu mai poate fi una implicita ca in cazul celorlalti parametri.

pvParameter – pointer catre o zona in care se pot afla parametrii pe care firul ii va primi in cadrul functiei de executie. In cazul in care nu dorim sa transmitem nici un parametru firului folosim valoarea NULL.

fwdCreate – specifica modul in care va fi creat firul respectiv. Valoarea 0 indica faptul ca firul va considerat activ imediat dupa ce a fost creat, iar valoarea CREATE_SUSPENDED va duce la crearea firului in stare suspendata, oferind posibilitatea de a realiza modificari inainte de a porni efectiv firul. Pentru un fir creat in starea suspendat va trebui sa folosim functia ResumeThread pentru a-l porni.

pdwThreadID – reprezinta adresa unei variabile de tip DWORD unde se va stoca valoarea ID-ului firului respectiv, valoare care va fi atribuita de catre sistemul de operare. Pentru sistemele de tip Windows NT, se poate transmite valoarea NULL pentru acest parametru, insa pentru cele de tip Windows 95 acest lucru va duce la generarea unei erori.

Trebuie remarcat faptul ca functia CreateThread intoarce o valoare de tip DWORD care este numita si “exit code”. Aceasta valoare ar trebui sa indice daca firul a executat cu succes ceea ce avea de facut sau au aparut erori in executie. Prin convetie, valoarea 0 ne arata faptul ca firul s-a terminat cu succes, iar o valoare diferita de 0 semnifica un insucces.

8.3 Terminarea firelor de executie

Un fir de executie se poate termina in patru moduri, si anume:

a. functia firului ajunge la sfarsit si returneaza o valoare. Acest mod denumit si moarte naturala este cel mai indicat;
b. in functia firului se apeleaza ExitThread, ceea ce duce la “sinuciderea” firului de executie;
c. functia TerminateThread este apelata avand ca parametru firul respectiv;
d. procesul din care facea parte firul se termina.

Din punct de vedere al elegantei si claritatii programului doar prima metoda este indicata, insa si celelalte trei metode reprezinta solutii valide pentru anumite cazuri speciale. In cazul folosirii metodelor b, c si d se elibereaza resursele asociate firului de catre sistemul de operare, insa destructorii obiectelor create de catre fir nu vor fi apelati.

8.4 Suspendarea si reluarea executiei firelor

In cadrul duratei de viata a unui fir, acesta se poate afla si in starea suspendat in care poate ajunge din doua motive:

a. a fost folosit parametrul CREATE_SUSPENDED pentru functia CreateThread;
b. se apeleaza functia DWORD SuspendThread(HANDLE thread);

Spre deosebire de functia TerminateThread, la apelul SuspendThread se vor salva toate informatiile de care firul respectiv are nevoie, astfel incat la apelul functiei ResumeThread firul sa isi poata continua executia din punctul in care a fost trecut in suspendare. Functia de reluare a executiei unui fir este de forma:

```
DWORD ResumeThread(HANDLE Thread);
```

Daca pentru un fir se apeleaza functia SuspendThread de n ori, functia pereche ResumeThread va trebui apelata de acelasi numar de ori n pentru ca firul sa isi reia activitatea.
Un fir de executie poate transmite sistemului de operare faptul ca doreste sa nu mai fie programat pentru executie pentru o anumita perioada de timp prin apelul functiei:

```
void Sleep(DWORD dwMilliseconds);
```

Totusi parametrul mai sus amintit trebuie considerat aproximativ, sistemul de operare nu ofera garantia ca firul va fi reluat dupa numarul exact de milisecunde. De obicei reluarea executie se face dupa numarul de milisecunde specificat, dar sunt cazuri in care se poate prelungi durata cu cateva secunde sau chiar minute.

8.5 Comutarea catre un alt fir

Daca un anumit fir doreste sa cedeze controlul asupra unitatii centrale unui alt fir poate sa apeleze functia:

```
BOOL SwitchToThread();
```

moment in care sistemul de operare ofera controlul unui alt fir programat pentru executie (daca exista).
Aceasta functie nu este apelata doar din politete, exista cazuri in care un fir cu prioritate mare apeleaza aceasta functie deoarece asteapta eliberarea unei resurse care este detinuta de catre un fir cu prioritate mai mica.

Functia SwitchToThread are effect similar cu apelul functiei Sleep la care ii transmitem ca si numar de milisecunde 0. Deosebirea consta in faptul ca SwitchToThread poate duce la predarea controlului asupra CPU unui thread cu prioritate mai mica, in timp ce la apelul Sleep(0) daca nu exista un fir cu aceeasi prioritate, controlul va reveni firului care a apelat Sleep, chiar daca exista fire cu prioritate mai mica care ar dori sa se execute.

8.6 Timpul de executie al unui fir

In anumite cazuri este necesar sa stim cat timp ii este necesar unui fir pentru a executa o anume operatie. O solutie deseori utilizata de multi programatori ar fi:

```
// Se citeste timpul curent (timpul de start)
DWORD dwStartTime = GetTickCount();
……..
// Implementarea unui algoritm complex
……..
// Se calculeaza durata ca fiind diferenta dintre timpul
curent si timpul de start
DWORD dwElapsedTime = GetTickCount() - dwStartTime;
```

Aceasta metoda de calcul pleaca de la o presupunere simpla si anume aceea ca firul nu va fi intrerupt pe parcursul executiei. Aceasta presupunere nu mai este insa valida in cazul sistemelor de operare preemptive actuale, deoarece nu o sa stim niciodata in ce moment sistemul va ceda controlul procesorului firului nostru. Mai mult, cand firului ii este luat controlul de catre sistem (pentru a da posibilitatea executarii si a altor fire), timpul necesar terminarii anumitor sarcini devine mult mai greu de determinat. Pentru a se putea avea totusi acces la astfel de date, sistemul de operare Windows pune la dispozitie o functie numita GetThreadTimes:

```
BOOL GetThreadTimes(
   HANDLE hThread,
   PFILETIME pftCreationTime,
   PFILETIME pftExitTime,
   PFILETIME pftKernelTime,
   PFILETIME pftUserTime);
```

Functia returneaza patru valori diferite de timp, dupa cum urmeaza:

- **Creation time**: - valoare absoluta in care se indica timpul creeri firului.
- **Exit time**: - valoare absoluta in care se indica timpul “mortii” firului. Daca functia este apelata in timp ce firul se executa (nu este distrus) valoarea returnata nu este definita.
- **Kernel time**: - valoare relativa care indica timpul cat firul a folosit procesorul pentru executarea de cod al sistemului de operare.
- **User time**: - valoare relativa care indica timpul cat firul a folosit procesorul pentru executarea de cod al aplicatiei.

Similara acestei functii exista definita una echivalenta pentru procese, GetProcessTimes.

```
BOOL GetProcessTimes(
   HANDLE hProcess,
   PFILETIME pftCreationTime,
   PFILETIME pftExitTime,
   PFILETIME pftKernelTime,
   PFILETIME pftUserTime);
```

Functia returneaza unitatiile de timp ce se aplica tuturor firelor ce apartin procesului specificat.

8.7 Prioritatile firelor de executie

Fiecare fir are asignat un numar, reprezentand prioritatea acestuia, luand valori intre 0 (prioritatea cea mai mica) si 31 (prioritatea cea mai mare). In momentul in care sistemul trebuie sa decida carui fir sa-i permita acesul la procesor, intr-un prim pas examineaza toate firele care au prioritatea 31 si le permite acesul folosind tehnica round-rubin. La sfarsitul perioadei de timp alocate unui fir, sistemul verifica daca mai exista un fir cu prioritatea 31, care asteapta sa primeasca controlul procesorului; daca exista, este executat. Atata timp cat exista fire de executie cu prioritatea 31 sistemul de operare nu va da accesul la procesor nici unui alt fir cu o prioritate intre 0 si 30. Aceasta este asa numita conditie de infometare (starvation) in care firele cu prioritati mari folosesc atat de mult procesorul, incat celelalte nu mai pot sa se execute. In sistemele multiprocesor insa, aceasta conditie este foarte greu de atins, deoarece sisetemul de operare incearca pe cat posibil ca procesoarele sa fie ocupate cat mai mult timp. Intr-un astfel de sistem, doua fire cu prioritati diferite 31 respectiv 30 pot fi executate simultan.

Pentru a preintampina eventualele probleme, sistemul de operare Windows ofera un suport si acces limitat programatorului la scheduler-ul sistemului (programul care se ocupa efectiv de asignarea microprocesorului diferitelor fire din sistem). Astfel sunt puse la dispozitia programatorului sase clase de prioritati, dupa cum urmeaza:

Clasa	**Identificatori**
Real-time	REALTIME_PRIORITY_CLASS
High	HIGH_PRIORITY_CLASS
Above normal	ABOVE_NORMAL_PRIORITY_CLASS
Normal	NORMAL_PRIORITY_CLASS
Below normal	BELLOW_NORMAL_PRIORITY_CLASS
Idle	IDLE_PRIORITY_CLASS

Clasa Real-time: firele dintr-un astfel de process trebuie sa raspunda imediat la evenimente pentru a putea fi capabile sa execute sarcini de maxima importanta. De asemenea, scheduler-ul intrerupe firele cu prioritati mai mici, chiar daca acestea sunt in mujlocul unitatii de timp allocate, pentru a da controlul celor cu

prioritate mare. Acest tip de prioritate este recomandat a se folosi cu foarte mare grija.

Clasa High: firele dintr-un astfel de process trebuie sa raspunda imediat la evenimente pentru a putea fi capabile sa execute sarcini de maxima importanta. Programul Task Manager ruleaza la o astfel de prioritate, pentru a putea intrerupe la nevoie, in conditii de maxima siguranta, anumite procese din sistem.

Clasa Above normal: este o clasa de prioritati nou introdusa (Windows 2000). Firele procesului din aceasta clasa ruleaza cu prioritati intre clasele High si Normal.

Clasa Normal: este clasa cea mai comuna, folosita de majoritatea aplicatiilor.

Clasa Below normal: este o clasa de prioritati nou introdusa (Windows 2000). Firele procesului din aceasta clasa ruleaza cu prioritati intre clasele Normal si Idle.

Clasa Idle: firele din pocesele cu aceasta clasa de prioritati ruleaza doar cand in system nu ruleaza alte procese cu prioritati mai mari. Aceasta clasa de prioritati este de obicei folosita de aplicatii de tip screensaver sau programe utilitare de background.

Clasa de prioritati a procesului

Prioritatea relativa a firului	Idle	Blow normal	Normal	Above normal	High	Real-time
Time-critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

De notat, ca in tabelul de mai sus nu exista nici un fir cu prioritatea 0. Aceasta se intampla deoarece singurul fir din sistem cu aceasta prioritate este firul numit “zero page”, instantiat in procesul de boot-are si care se ocupa cu evacuarea paginilor din memoria RAM. De asemenea urmatoarele nivele de prioritate nu pot fi obtinute, decat de programe care ruleaza in modul kernel (drivere): 17, 18, 19,

20, 21, 27, 28, 29, 30. Conceptul de clase de prioritati, desi poate confuz, este folosit doar ca si un concept abstract pentru a izola utilizatorul de conceptele si modurile interne de lucru ale scheduler-ului.

Functia pusa la dispozitie pentru schimbarea clasei de prioritate a unui process este SetPriorityClass.

```
BOOL SetPriorityClass(HANDLE hProcess,
                      DWORD  fdwPriority);
```

Prioritatea unui fir poate fi setata folosinduse functia SetThreadPriority

```
BOOL SetThreadPriority(
   HANDLE hThread,
   int    nPriority);
```

Prioritatea relativa a firului	Constanta simbolica
Time-critical	THREAD_PRIORITY_TIME_CRITICAL
Highest	THREAD_PRIORITY_HIGHEST
Above normal	THREAD_PRIORITY_ABOVE_NORMAL
Normal	THREAD_PRIORITY_NORMAL
Below normal	THREAD_PRIORITY_BELOW_NORMAL
Lowest	THREAD_PRIORITY_LOWEST
Idle	THREAD_PRIORITY_IDLE

8.8 Probleme

Aplicatia aferenta lucrarii creaza patru fire care modifica cele patru progressbar-uri ale aplicatiei. Fiecare fir are definita cate o functie proprie si este creat in starea suspendat. Firele vor porni in momentul in care utilizatorul apasa unul din cele patru butoane “Resume” si isi vor incheia executia la apasarea butonului “Terminate”.

1. Folosind aplicatia Task Manager a sistemului de operare, urmariti numarul de fire existente in sistem inainte de rularea aplicatiei si dupa lansarea in executie.
2. Modificati programul astfel incat butonul “Resume” sa indeplineasca doua functii:

   - daca textul afisat este “Resume” sa reia executia firului si sa isi modifice textul afisat in “Suspend”;

   - daca textul afisat este “Suspend” sa suspende executia firului si sa isi modifice textul afisat in “Resume”.

3. Modificati programul astfel incat la apasarea butonului “Resume” sa se citeasca din combobox-ul aferent valoarea prioritatii si sa se atribuie firului respectiv inainte ca acesta sa isi reia activitatea.

4. Pentru fiecare dintre cele patru fire, adaugati cate un buton si o zona de text. La apasarea butonului, in zona de text se vor afisa informatiile obtinute prin apelul functiei GetThreadTimes.

5. Modificati programul astfel incat toate cele patru fire sa foloseasca ca si punct de pornire aceeasi functie. Observati daca exista modificari in functionare.

9 Sincronizarea firelor de executie. Sectiuni critice

9.1 Introducere

Intr-un sistem de operare, toate firele de executie trebuie sa aiba acces la resursele acestuia (porturi seriale, fisiere, ferestre, etc.). Daca un fir cere acces exclusiv sistemului asupra unei anumite resurse, celelalte fire nu o vor putea folosi. Pe de alta parte, sistemul nu poate permite oricarui fir sa acceseze o anumita resursa la orice moment de timp. Imaginati-va ca un fir ar scrie o anumita zona de memorie, in timp ce altul citeste aceeasi zona. Situatia ar fi analoaga cu aceea in care o persoana citeste o carte in timp ce alta modifica continutul paginilor.

Firele pot si este necesar sa comunice intre ele, cand:

- Mai multe fire acceseaza resurse partajate, fara ca acestea sa devina corupte (inconsistente);
- Un fir trebuie sa notifice un alt fir sau mai multe, ca o anumita operatie a fost terminata.
- Pentru ca aceasta comunicare sa se poata realiza intr-un mod sigur, fara a fi periclitate resursele comune sau celelalte fire coexistente, apare necesitatea definirii si implementarii de mecanisme de sincronizare.

Lucrarea de fata isi propune sa prezinte mecanismele de sincronizare “sectiune critica” si “mutex”.

9.2 Sectiuni critice

O sectiune critica este o zona de cod ce necesita acces exclusiv la o resursa partajata a sistemului, inainte ca si codul din zona respectiva sa poata fi executat. Portiunea respectiva de cod va manipula *atomic* resursa sistemului, existand siguranta ca nici un alt fir nu o va putea accesa. Cu alte cuvinte, orice fir ce doreste sa foloseasca resursa respectiva, nu va primi controlul asupra ei atata timp cat firul ce detine controlul, nu va parasi sectiunea critica. In continuare este prezentat un exemplu simplu, ce demonstreaza ce s-ar putea intampla daca nu sunt folosite sectiuni critice.

```
const int MAX = 1000;
int index = 0;
DWORD array_timp[MAX];

DWORD WINAPI FunctieThread1(PVOID pvParam) {
     while(index < MAX) {
          array_timp[index] = GetTickCount();
          index++;
     }
     return(0);
}

DWORD WINAPI FunctieThread2(PVOID pvParam) {
     while(index < MAX) {
          index++;
          array_timp[index - 1] = GetTickCount();
     }
     return(0);
}
```

Luate independent, cele doua functii din exemplu produc acelasi rezultat si anume vor umple sirul global *array_timp* cu valori (esantioane) temporale crescatoare. Daca insa sunt privite ca si functii a doua fire concurente, rezultatele nu sunt garantate a fi aceleasi. Sa presupunem ca cele doua fire tocmai au fost startate, pe un calculator uniprocesor. Sistemul de operare incepe rularea firului 2, care incrementeaza variabila globala *index* la valoarea 1. Sistemul de operare fiind preemptiv, presupunem ca in acest moment va transfera controlul primului fir. Acesta seteaza *array_timp[1]* cu timpul sistemului, dupa care sistemul reda controlul celui de-al doilea fir. Acesta, isi va relua executia cu instructiunea urmatoare si va seta continutul *array_timp[0]* cu timpul sistemului. Aceasta operatie fiind insa efectuata la un moment urmator in timp, celei similare din primul fir, va avea ca rezultat o inconsistenta a continutului lui *array_timp*, acesta nemaifiind garantat a contine valori temporale crescatoare. Versiunea corecta a programului, folosind sectiuni critice, este urmatorea:

```
const int MAX = 1000;
int index = 0;
DWORD array_timp[MAX];
CRITICAL_SECTION sectiune_critica;

DWORD WINAPI FunctieThread1(PVOID pvParam) {
     while(index < MAX) {
          EnterCriticalSection(&sectiune_critica);
          array_timp[index] = GetTickCount();
          index++;
          LeaveCriticalSection(&sectiune_critica);
     }
     return(0);
}

DWORD WINAPI FunctieThread2(PVOID pvParam) {
     while(index < MAX) {
          EnterCriticalSection(&sectiune_critica);
          index++;
          array_timp[index - 1] = GetTickCount();
          LeaveCriticalSection(&sectiune_critica);
     }
     return(0);
}
```

In exemplul anterior, este initializata o structura de tip CRITICAL_SECTION, *sectiune_critica*, dupa care orice secventa de cod care “atinge” cele doua variabile globale *index* si *array_timp* este gardata de cele doua apeluri *EnterCriticalSection* si *LeaveCriticalSection.*

De cate ori un fir acceseaza o resursa partajata, accesul trebuie facut dupa un apel prealabil al metodei *EnterCriticalSection*, in care este transmisa ca si parametru sectiunea critica prin care va fi identificata resursa respectiva. Daca functia *EnterCriticalSection* detecteaza ca nici un alt fir nu a blocat sectiunea

respectiva, va permite blocajul pentru firul apelant. Cand acesta nu mai are nevoie de resursa respectiva, va executa un apel al functiei *LeaveCriticalSection*, prin care este resetat blocajul asupra resursei si implicit este redata posibilitatea altor fire de a accesa resursa comuna.

In cazul in care se uita apelul functiilor *EnterCriticalSection* si *LeaveCriticalSection* la folosirea de resurse globale (accesibile si folosite si de alte fire), implementarea programului respectiv devine una nesigura si inconsistenta, cu o mare deschidere pentru erori.

Principalul dezavantaj al sectiunilor critice este acela ca nu pot fi folosite la sincronizarea firelor din procese diferite.

In general, structurile de tip CRITICAL_SECTION trebuie sa fie alocate global, pentru a putea fi accesate de toate firele programului. Ele pot fi alocate insa si local sau dinamic, o prima cerinta fiind necesitatea ca toate firele care le folosesc sa cunoasca adresele lor. O a doua cerinta este aceea ca structura sa fie initializata inainte ca orice fir sa incerce accesarea resursei protejate. Initializarea se face printr-un apel al functiei :

```
VOID InitializeCriticalSection(PCRITICAL_SECTION pcs);
```

Aceasta functie initializeaza membrii unei structuri CRITICAL_SECTION (pointata de pcs). Aceasta functie trebuie chemata inainte ca orice fir sa faca apeluri ale functiei EnterCriticalSection, altfel rezultatele fiind nedefinite daca se incearca intrarea intr-o sectiune criitica neinitializata.

In momentul in care firele procesului nu mai folosesc resursa respectiva si deci nu mai au nevoie de sectiunea critica anterior creata, aceasta poate fi stearsa (resetata) printr-un apel al functiei :

```
VOID DeleteCriticalSection(PCRITICAL_SECTION pcs);
```

Functia reseteaza variabilele membru din interiorul structurii. In mod natural, sectiunea critica nu trebuie stearsa daca mai exista fire care o folosesc.

Odata sectiunea critica initializata, orice secventa de cod care acceseaza o resursa partajata trebuie precedata de un apel al functiei *EnterCriticalSection.*

Functia inspecteaza continutul variabilelor sectiunii critice primite ca si parametru. Acestea indica ce fir, daca exista unul, acceseaza la un moment dat resursa.

Functia va efectua urmatoarele teste:

- Daca nici un fir nu acceseaza resursa, *EnterriticalSection* actualizeaza variabilele membru pentru a indica faptul ca firul apelant a primit accesul in sectiunea respectiva si se reda controlul imediat acestuia pentru a-si putea continua activitatea.
- Daca variabilele membru indica faptul ca firul apelant deja a primit accesul, sunt actualizate variabilele structurii pentru a indica de cate ori i-a fost acceptat accesul firului apelant. Aceasta sitautie apare rar, doar in cazul in care se fac doua apeluri succesive de *EnterCriticalSection* fara a apela *LeaveCriticalSection.*
- Daca variabilele membru ale sectiunii critice indica faptul ca deja un fir (altul decat cel apelant) detine dreptul de acces, firul apelant este pus in stare de asteptare. In starea de asteptate, firul nu va folosi deloc timp al CPU. Sistemul memoreaza faptul ca firul doreste sa acceseze resursa si automat actualizeaza continutul variabilelor membru, permitandu-se astfel ca firului sa-I fie permis accesul imediat dupa ce firul ce detine controlul apeleaza functia *LeaveCriticalSection*.

Implementarea functiei *EnterCriticalSection* nu este complicata, in ea se executa doar niste simple teste. Ceea ce face ca aceasata functie sa fie valoroasa, este faptul ca daca doua fire separate, intr-un sistem multiprocesor apeleaza functia in exact acelasi moment, aceasta se va comporta corect si anume, va permite accesul la resursa unuia din fire, trecandul pe celalalt in stare de asteptare.

Firele care sunt in stare de asteptare nu ajung niciodata la *conditia de infometare* (*starvation* – nu mai primesc acces la CPU), apelurile functiei *EnterCriticalSection* eventual ajungand sa expire si sistemul sa genereze o exceptie. Perioada de timp dupa care apelurile trebuie sa expire este determinata de continutul variabilei *CriticalSectionTimeout* din intrarea in registrii sistemului:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager
```

Continutul variabilei este setat in mod implicit la 2.592.000 secunde, aproximativ 30 zile.
O functie asemanatoare este:

```
BOOL TryEnterCriticalSection(PCRITICAL_SECTION pcs);
```

Apelul acestei functii nu va pune niciodata firul apelant in stare de asteptare, returnand in schimb o valoare booleana prin care se poate determina daca acesta a obtinut sau nu accesul.

In momentul in care firul ce detine controlul sectiunii critice si-a indeplinit sarcinile, aceasata trebuie eliberata, folosindu-se functia:

```
VOID LeaveCriticalSection(PCRITICAL_SECTION pcs);
```

LeaveCriticalSection examineaza variabilele structurii sectiunii critice. Va decrementa cu 1 contorul in care este memorat numarul de ori cat firul apelant a primit accesul la resursa partajata. Daca valoarea contorului este mai mare decat 0, functia reda controlul firului apelant, fara a mai face altceva. Daca valoarea devine 0, se verifica daca nu mai exista alte fire care sunt trecute in stare de asteptare in urma unui apel *EnterCriticalSection* pentru aceeasi sectiune critica. Daca exista cel putin un fir in asteptare, variabilele structurii sunt actualizate si este “trezit” unul dintre firele in asteptare.

Sectiuni critice si spinlock - uri – in momentul in care un fir incearca sa intre intr-o sectiune critica controlata de un alt fir, este trecut in stare de asteptare. Aceasta inseamna ca firul trebuie sa faca o tranzitie din *modul user* in *modul kernel*, care de obicei dureaza aproximativ 1000 cicli CPU. Aceasta tranzitie este extrem de costisitoare. Mai mult, intr-un calculator multiprocesor, firul care acceseaza resursa partajata, poate fi executata pe un procesor, si chiar poate sa o elibereze inainte ca cel apelant sa-si fi incheiat perioada de tranzitie si sa preia efectiv controlul asupra sectiunii critice si implicit asupra resursei eliberate. Pentru a imbunatatii performanta sectiunilor critice, Microsoft a introdus conceptul de *spinlock.* Mai precis, cand este apelata functia *EnterCriticalSection,* se incearca

intr-un ciclu accesarea resursei, inainte de efectuarea propriuzisa a tranzitiei firului in modul kernel, in starea de asteptare. Pentru a folosi un spinlock cu o sectiune critica, sectiunea critica trebuie initializata cu functia:

```
BOOL InitializeCriticalSectionAndSpinCount(
     PCRITICAL_SECTION pcs,
     DWORD dwSpinCount);
```

Ca si in cazul functiei *InitializeCriticalSection,* primul parametru reprezinta adresa sectiunii critice. Al doilea parametru, *dwSpinCount*, reprezinta numarul de iteratii pe care in care ciclul spinlock incearca sa preia controlul asupra resursei, inainte de trecerea in starea de asteptare. Aceasta valoare poate sa varieze intre 0 si 0x00FFFFFF. Daca aceasta functie este apelata intr-un program ce ruleaza pe un calculator cu un singur procesor, al doilea parametru este ignorat (pus pe 0). Numarul de spin-uri al unei sectiuni critice poate fi schimbat foloosindu-se functia

```
DWORD SetCriticalSectionSpinCount(
     PCRITICAL_SECTION pcs,
     DWORD dwSpinCount);
```

Din nou, al doilea parametru este ignorat in cazul rularii pe un calculator cu un singur procesor.

Sugestii in folosirea sectiunilor critice:

- Se recomanda folosirea unei variabile de tip CRITICAL_SECTION pentru fiecare resursa partajata;
- Accesul la mai multe resurse partajate esete bine sa fie facut, daca este posibil, simultan;
- Nu este recomandat ca un fir sa pastreze un timp indelungat controlul asupra sectiunii critice; performanta aplicatiei ar putea avea de suferit, daca celelalte fire concurente asteapta prea mult timp pentru eliberarea aceseteia;

9.3 Probleme

1. Creati o aplicatie care are doua fire de executie, fiecare dinre ele executand codul urmator:

```
while(true){
     for(int i = 0 ; i < 10 ; i++){
          globalCounter++ ;
     }
}
```

Variabila globalCounter este definita la nivel global si ambele fire au access la ea.
Dupa fiecare incrementare a lui globalCounter afisati intr-o lista id-ul firului si valoarea variabilei.
Folosind mecanismul de sectiune critica, modificati programul astfel incat in momentul in care unul dintre fire incepe incrementarea, el sa detina controlul asupra variabilei pana la terminarea ciclului for.
Lista va trebui sa contina inregistrari de forma :

```
Thread1 - 0     1501
Thread1 - 1     1502
Thread1 - 2     1503
Thread1 - 3     1504
Thread1 - 4     1505
Thread1 - 5     1506
Thread1 - 6     1507
Thread1 - 7     1508
Thread1 - 8     1509
Thread1 - 9     1510
Thread1 - 10     1511
…….
Thread_2 - 0          1551
Thread_2 - 1          1552
Thread_2 - 2          1553
Thread_2 - 3          1554
Thread_2 - 4          1555
Thread_2 - 5          1556
Thread_2 - 6          1557
Thread_2 - 7          1558
Thread_2 - 8          1559
Thread_2 - 9          1560
Thread_2 -10          1561
```

Observati continutul listei si in cazul in care nu se foloseste sectiunea critica.

2. Creati o aplicatie care pe un fir incarca si afiseaza continutul unui fisier de dimensiune mare, iar pe un altul adauga la sfarsitul aceluiasi fisier textul “thread2“. Pe interfata vor exista doua butoane pentru pornirea celor doua fire. Realizati aplicatia astfel incat, indiferent de timpul de pornire al celor doua fire, continutul fisierului afisat nu va contine textul “thread2“. Se presupune ca firul care citeste se va porni intotdeauna inaintea celui care adauga la sfarsitul fisierului.

10 Sincronizarea firelor de executie. Evenimente.

10.1 Introducere

In lucrarea “Fire de executie” s-a putut observa modul in care decurge executia in timp a firelor, dar si faptul ca acestea nu interactionau intre ele. In majoritatea cazurilor insa, prin natura aplicatiilor firele trebuie sa interactioneze intre ele prin diverse metode, sa astepte anumite rezultate de la alte fire, sa semnalizeze faptul ca au terminat de executat anumite operatii, etc.

Din punct de vedere al sistemului de operare urmatoarele obiecte se pot afla in starea semnalizat sau nesemnalizat:

- Processes;
- Threads;
- Jobs;
- Files;
- Console input;
- File change notification;
- Events;
- Waitable timers;
- Semaphores;
- Mutexes.

Toate aceste obiecte au in interiorul un membru boolean care indica starea unui obiect mai sus amintit, astfel: valoarea TRUE ne indica faptul ca obiectul este semnalizat, iar FALSE ne arata faptul ca obiectul se afla in starea nesemnalizat.

In momentul in care sistemul de operare creaza un astfel de obiect atribuie valoarea TRUE sau FALSE, dupa caz, pentru membrul responsabil pentru mentinerea starii obiectului.

In continuare ne vom ocupa de tratarea obiectelor Event, cu mentiunea ca punctul urmator, “2” este valabil si pentru celelalte obiecte amintite anterior.

10.2 Functii de asteptare

Functiile de asteptare dau posibilitatea firelor de executie sa astepte pana cand anumite obiecte devin semnalizate.

Trebuie amintit aici faptul ca unui fir care asteapta ca un obiect sa fie semnalizat nu i se va mai da controlul asupra procesorului, indiferent de prioritatea acestuia.

Exista urmatoarele functii pentru asteptare:

```
DWORD WaitForSingleObject(
          HANDLE hObject,
          DWORD dwMilliseconds);
```

Unde hObject este obiectul pe care firul asteapta sa fie pus in starea semnalizat, iar dwMilliseconds este numarul maxim de milisecunde asteptate de catre fir.

De obicei pentru al doilea parametru al functiei se foloseste valoarea INFINITE, firul asteptand pana cand obiectul devine semnalizat, fara limita de timp.

```
WaitForSingleObject( myHandle, INFINITE);
```

Pentru a verifica modul in care s-a terminat apelul functiei WaitForSingleObject putem testa valoarea returnata, care poate fi:

- WAIT_OBJECT_0 – semnifica faptul ca obiectul asteptat a fost semnalizat;
- WAIT_TIMEOUT – ne arata ca a trecut timpul maxim de asteptare fara ca obiectul sa fie pus in starea semnalizat;
- WAIT_FAILED – ne arata ca functia nu a putut fi executata, cel mai probabil datorita faptului ca hObject-ul transmis a fost invalid.

```
DWORD WaitForMultipleObjects(
          DWORD dwCount,
          CONST HANDLE* phObjects,
          BOOL fWaitAll,
          DWORD dwMilliSeconds);
```

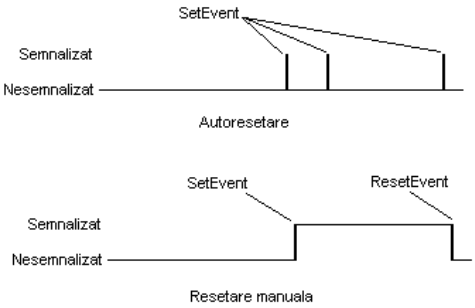
Unde dwCount indica numarul de obiecte pentru care se asteapta, valoarea acestui parametru fiind cuprinsa intre 1 si MAXIMUM_WAIT_OBJECTS (definita ca egala cu 64), phObjects este un pointer catre un array care contine obiectele pentru care se asteapta, fWaitAll specifica daca se asteapta ca toate obiectele trimise sa fie trecute in starea semnalizat, sau doarun obiect din cele trimise este suficient pentru ca asteptarea sa ia sfarsit, iar dwMilliseconds are acelasi rol ca si la functia WaitForSingleObject.
Pentru a verifica modul in care s-a terminat apelul functiei WaitForMultipleObjects putem testa valoarea returnata, care poate fi:

- WAIT_TIMEOUT – ne arata ca a trecut timpul maxim de asteptare fara ca obiectul sa fie pus in starea semnalizat;
- WAIT_FAILED – ne arata ca functia nu a putut fi executata, cel mai probabil datorita faptului ca hObject-ul transmis a fost invalid;
- WAIT_OBJECT_0 – semnifica faptul ca toate obiectele asteptate au fost semnalizate, bineinteles numai daca pentru parametrul fWaitAll a fost transmisa valoarea TRUE;
- [WAIT_OBJECT_0, WAIT_OBJECT_0 + dwCount - 1] – una dintre valorile acestui interval va fi returnata daca paramertul fWaitAll a avut valoarea FALSE si unul dintre obiecte a fost trecut in starea semnalizat. Pentru a afla care dintre obiecte a fost semnalizat, scadem din valoarea returnata valoarea WAIT_OBJECT_0.

10.3 Functii pentru managementul evenimentelor

In principiu un eveniment este o structura din care face parte, pe langa alte informatii, o valoare booleana care ne indica daca evenimentul este semnalizat sau nu, si o alta care ne arata tipul de eveniment, cu resetare manuala sau cu resetare automata.
De obicei, un eveniment semnalizat ne arata ca o anumita operatie a fosta realizata. In cazul evenimentelor cu resetare manuala, toate firele care asteptau dupa acel eveniment vor fi trezite, iar evenimentul va ramane semnalizat pana la apelul functiei ResetEvent. Pentru evenimentele cu resetare automata, un singur

fir din cele care asteptau dupa acel eveniment va fi trezit, iar evenimentul va fi trecut automat in starea nesemnalizat, celelalte fire care il asteptau ramanand in adormire pana la noi semnalizari ale evenimentului.



Timpul cat un eveniment se afla in starea semnalizat

Trebuie mentionat ca in diagrama de mai sus, pentru cazul evenimentelor cu resetare automata, daca nu exista nici un fir care sa astepte acel eveniment, el va ramane in starea semnalizat pana cand cineva va solicita asteptarea pentru evenimentul in cauza.

Pentru crearea unui eveniment avem la dispozitie functia:

```
HANDLE CreateEvent(
          PSECURITY_ATTRIBUTES psa,
          BOOL fManualReset,
          BOOL fInitialState,
          PCTSTR pszName);
```

Unde amintim ca fManualReset indica daca obiectul este cu resetare manuala (TRUE), sau cu resetare automata (FALSE), fInitialState ne arata starea in care va fi creat evenimentul (semnalizat=TRUE sau nesemnalizat=FALSE), iar pszName este un pointer catre un sir de caractere reprezentand numele evenimentului.

```
HANDLE OpenEvent(
          DWORD fdwAccess,
          BOOL fInherit,
```

```
        PCTSTR pszName);
```

Cu ajutorul functiei OpenEvent fire ale altui proces pot primi un maner la un anumit eveniment daca cunosc numele acestui si il transmit ca paramentu la apelul functiei mai sus amintite.

```
BOOL SetEvent(HANDLE hEvent);
```
Trece evenimentul transmis ca parametru in starea semnalizat.

```
BOOL ResetEvent(HANDLE hEvent);
```
Trece evenimentul transmis ca parametru in starea nesemnalizat.

```
BOOL PulseEvent(HANDLE hEvent);
```
Trece evenimentul transmis ca parametru in starea semnalizat, iar imediat dupa aceea evenimentul va fi trecut in starea nesemnalizat. La apelul acestei functii pentru un eveniment cu resetare manuala doar unul dintre firele care asteptau acel eveniment va fi trezit, iar pentru apelul cu parametru de tip eveniment cu resetare automata se va trezi la fel ca in cazul anterior, un fir din cele care asteapta, insa daca nu exista nici un fir care sa astepte, evenimentul respectiv va fi trecut in starea nesemnalizat.

10.4 Probleme

1. Realizati un program de rezolvare a ecuatiei de gradul al doilea, astfel: de pe firul principal al aplicatiei creati trei fire, unul pentru calcularea discriminantului delta, fir care asteapta initializarea valorilor coeficientilor a, b, c lucru semnalizat prin evenimentul “a b c Ready”. Al doilea fir calculeaza valoarea primei solutii x1, imediat ce are delta disponibil, adica dupa ce firul pentru delta semnalizeaza prin evenimentul “delta Ready”, iar al treilea fir calculeaza x2, dupa ce delta este calculat, adica dupa ce firul delta anunta acest lucru prin semnalizarea “delta Ready”.

Firele care calculeaza x1 si x2 for semnaliza terminarea calculelor prin evenimentul “x1 Ready”, respectiv “x2 Ready”, aceste doua evenimente fiind asteptate de catre firul principal al aplicatiei, care acum poate sa afiseze solutiile.

2. Creati o aplicatie care sa simuleze urmatoarea situatie : se considera un grup de patru persoane (fire de executie). Persoana p1 se afla la Sibiu, p2 si p3 la Brasov, iar p4 la Bucuresti. La momentul t0 persoana p1 pleaca spre Brasov. P2 si p3 asteapta sa vina p1 un timp t1 (relativ la t0) dupa care pleaca (cu sau fara p1) spre Bucuresti. P4 care se afla la Bucuresti asteapta un timp t2 (relativ la t0) dupa p1, p2 si p3 dupa care pleaca cu sau fara acestea spre Constanta. Daca p1 ajunge la Brasov dupa plecarea lui p2 si p3, se va opri aici.
Daca p2 si p3 (cu sau fara p1) ajung la Bucuresti dupa plecarea lui p4 se vor opri aici.
Pentru simularea drumului celor patru persoane se vor folosi cortroale de tip progressbar.
Aplicatia va avea capabilitati de setare a timpilor t1 si t2 precum si a duratei (sau vitezei) pentru deplasarea persoanei p1 de la Sibiu la Brasov, respectiv p2 si p3 (cu sau fara p1) de la Brasov la Bucuresti.

11 Sincronizarea firelor de executie. Mutex. Semaphore. Waitable Timer

11.1 Mutex

Mutex-urile sunt obiecte kernel care asigura accesul mutual exclusiv asupra unei resurse. Mutex-urile se comporta asemanator sectiunilor critice, dar sunt obiecte kernel, in timp ce sectiunile critice sunt obiecte de mod utilizator.
Spre deosebire de sectiunile critice, mutex-ul poate fi folosit si la sincronizarea firelor din procese diferite.
Structura unui mutex contine un id de fir care ne arata ce fir detine mutex-ul respectiv la un anumit moment. Daca id-ul firului este zero inseamna ca nici un fir nu detine mutex-ul respectiv si acesta se afla in starea semnalizat, iar daca id-ul firului este diferit de zero mutex-ul este detinut de catre firul cu valoarea ‘id’ si se afla in starea nesemnalizat.

Un mutex se poate crea prin apelul functiei :

```
HANDLE CreateMutex (
          PSECURITY_ATTRIBUTES psa,
          BOOL fInitialOwner,
          PCSTR pszName);
```

- pszName este numele mutex-ului si el va putea fi folosit de catre fire ale altui proces decat cel din cadrul caruia a fost creat mutex-ul pentru a avea acces la acesta, dupa cum va fi descris in continuare, folosind functia OpenMutex.
- fInitialOwner este parametrul cu ajutorul caruia putem stabili starea initiala a mutexului. Valoarea false va stabili pentru id-ul firului care detine mutexul valoarea zero si va pune mutexul in starea semnalizat. Valoarea true va pune mutex-ul in starea nesemnalizat si pentru id-ul firului va stabili valoarea id-ului firului care a creat mutex-ul. Mutex-ul va ramane in starea nesemnalizat pana la apelul functiei ReleaseMutex care va fi descrisa mai jos.

```
HANDLE OpenMutex (
          DWORD fwdAccess,
          BOOL bInheritHandle,
          PCSTR pszName);
```

Functia OpenMutex se apeleaza in cazul in care un fir care doreste acces la un anumit mutex nu are handle-ul acestuia, dar cunoaste numele lui. Dupa cum am amintit mai sus, acest caz este intalnit la sincronizarea firelor din procese diferite.

```
BOOL ReleaseMutex (HANDLE hMutex);
```

Cu ajutorul acestei functii un fir poate sa elibereze un mutex pe care il detine. In momentul eliberarii id-ul intern al mutex-ului primeste valoarea 0 si deci el va putea sa fie detinut in continuare de catre an alt fir.

Pentru a obtine controlul asupra unui mutex se folosesc functiile de asteptare descrise in laboratorul precedent, si anume :

```
DWORD WaitForSingleObject(
          HANDLE hObject,
          DWORD dwMilliseconds);
```

sau,

```
DWORD WaitForMultipleObjects(
          DWORD dwCount,
          CONST HANDLE* phObjects,
          BOOL fWaitAll,
          DWORD dwMilliSeconds);
```

11.2 Semafoare

Semafoarele servesc la controlul asupra unui numar limitat de resurse. Ele pot fi considerate o generalizare a sectiunilor critice sau a mutex-urilor, astfel nu doar un fir ci mai multe pot sa detina un semafor. Numarul maxim de fire care pot detine in acelasi timp un semafor este mentinut in structura acestuia, impreuna cu un counter care ne arata cate fire detin la un anumit moment semaforul.

Pentru a intelege mai bine cum functioneaza un semafor vom descrie valorile pe care le poate avea counter-ul ce ne arata numarul de fire care detin semaforul la un moment dat si modul cum aces counter isi schimba valorile :

- Counter-ul nu va avea niciodata valori negative, iar valoarea lui nu va putea fi mai mare decat valoarea specificata la momentul creerii semaforului ca fiind numarul maxim de fire ce pot detine controlul asupra semaforului, dupa cum vom vedea in prototipul functiei CreateSemaphore.

- Daca counter-ul are valoarea 0 semaforul nu este semnalizat, si deci nici un fir nu va mai putea obtine acces asupra lui.

- Daca counter-ul are o valoare diferita de 0 un fir poate obtine control asupra semaforului prin functiile de asteptare WaitForSingleObject sau WaitForMultipleObjects. In momentul in care un fir primeste control asupra semaforului, valoare counter-ului se decrementeaza, iar in momentul in care ajunge la 0 nici un alt fir nu mai poate obtine control asupra semaforului.

- La apelul functiei ReleaseSemaphore de catre un fir care are control asupra semaforului, counter-ul se incrementeaza cu unu (de obicei), dupa cum vom vedea in descrierea prototipului functiei.

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG lInitialCount,
  LONG lMaximumCount,
  LPCTSTR lpName
);
```

Aceasta functie se utilizeaza pentru a crea un semafor, iar parametrii au urmatoarele semnificatii :

- `lInitialCount` – ne arata cate dintre resursele semaforului creat sunt disponibile initial. De obicei lInitialCount are valoarea numarului maxim de resurse controlate.
- `lMaximumCount` – specifica numarul maxim de resurse pe care semaforul le va putea controla.

```
HANDLE OpenSemaphore(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName
);
```

Se foloseste pentru a obtine un handle la un semafor al carui nume este cunoscut. Amintim aici ca semaforul este obiect care se poate folosi la sincronizare intre procese.

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,
  LONG lReleaseCount,
  LPLONG lpPreviousCount
);
```

Prin apelul acestei functii indicam faptul ca se elibereaza un anumit numar de resurse, si anume `lReleaseCount` resurse.
In paramentrul `lpPreviousCount` functia intoarce valoarea numarului de resurse controlate inainte de apelul functiei.

Pentru a obtine controlul asupra unui semafor un fir poate folosi functiile WaitForSingleObject sau WaitForMultipleObjects.

11.3 Waitable Timer

Aceste obiecte sunt folosite in situatii in care dorim sa efectuam anumite operatii la un moment specificat sau in cazul in care operatiile respective se repeta la un anumit interval de timp.

Crearea unui timer se face prin simplul apel al functiei CreateWaitableTimer, dar el nu va putea fi utilizat inainte de apelul functiei SetWaitableTimer care il va defini complet.

```
HANDLE CreateWaitableTimer(
  LPSECURITY_ATTRIBUTES lpTimerAttributes,
  BOOL bManualReset,
  LPCTSTR lpTimerName
);
```

Ca si in cazul evenimentelor, un timer poate fi cu reset manual sau automat, dupa cum se specifica prin parametrul `bManualReset`. Parametrul `lpTimerName` este numele timer-ului si el va putea fi folosit la apelul OpenWaitableTimer.

```
HANDLE OpenWaitableTimer(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpTimerName
);

BOOL SetWaitableTimer(
  HANDLE hTimer,
  const LARGE_INTEGER* pDueTime,
  LONG lPeriod,
  PTIMERAPCROUTINE pfnCompletionRoutine,
  LPVOID lpArgToCompletionRoutine,
  BOOL fResume
);
```

Prin intermediul acestei functii se stabileste data cand timer-ul va trece in starea semnalizat si perioada de timp dupa care se vor efectua urmatoarele semnalizari.

`pDueTime` – spefica timpul cand timer-ul va trece in starea semnalizat.

`lPeriod` – ne arata perioada de timp dupa care timer-ul va trece din nou in starea semnalizat.

`pfnCompletionRoutine` - specifica o functie care se va executa in momentul in care timer-ul trece in starea semnalizat. Aceasta metoda este folosita din ce in ce mai putin.

`lpArgToCompletionRoutine` – parametrii pentru functia amintita mai sus.

`fResume` – este folosit pentru cazul in care calculatorul trece in starea de suspend. In cazul in care valoarea acestui parametru este TRUE, si timpul pentru care este setat timer-ul s-a scurs, calculatorul va reveni din starea suspend, iar firelor care asteptau timer-ul respectiv li se va da controlul.

Codul urmator creaza si seteaza un timer sa devina semnalizat dupa 7 secunde, iar dupa acee la interval de 2 minute :

```
HANDLE hTimer = NULL;
LARGE_INTEGER liDueTime;

//se inmulteste valoare dorita in secunde cu 10000000
//deoarece timpul specificat este in unitati de 100
//de nanosecunde
//timpul este specificat ca valoare negativa pentru
//a arata faptul ca valoarea data este relativa la
//momentul curent si nu este o data absoluta.
liDueTime.QuadPart= -7 * 10000000;

hTimer = CreateWaitableTimer(NULL, FALSE, "WaitableTimer");
SetWaitableTimer(hTimer, &liDueTime, 2 * 60 * 1000, NULL,
NULL, FALSE) ;

while(‘conditie’){
```

```
// Asteapta ca timer-ul sa fie semnalizat
if (WaitForSingleObject(hTimer, INFINITE) != WAIT_OBJECT_0)
{
     //aici asteptarea de 7 secunde s-a terminat.
}else{
     // a aparut o eroare
     GetLastError());
}
}
```

Pentru a crea un timer care sa fie semnalizat la data de 16.12.2003 ora 09 :35 putem folosi un cod de genul :

```
HANDLE hTimer = NULL;
LARGE_INTEGER liUTC;
SYSTEMTIME st;
FILETIME ftLocal, ftUTC;
hTimer = CreateWaitableTimer(NULL, FALSE, "WaitableTimer");
st.wYear = 2003 ;
st.wMonth = 12 ;
st.wDay = 16 ;
st.wHour = 9 ;
st.wMinute = 35 ;
SystemTimeToFileTime(&st, &ftLocal) ;
LocalFileTimeToFileTime(&ftLocal, &ftUTC) ;
liUTC.LowPart = ftUTC.dwLowDateTime ;
liUTC.HighPart = ftUTC.dwHighDateTime ;
SetWaitableTimer(hTimer, &liUTC, 0, NULL, NULL, FALSE) ;
.....
```

```
BOOL CancelWaitableTimer(
  HANDLE hTimer
);
```

Aceasta functie anuleaza timer-ul primit ca parametru, astfel incat acesta nu va mai trece in starea semnalizat la data stabilita anterior.

11.4 Probleme

1. Creati o aplicatie care pe un fir al unui proces p1 incarca si afiseaza continutul unui fisier de dimensiune mare, iar un fir al procesului p2 adauga la sfarsitul aceluiasi fisier textul “thread2“. Realizati aplicatia astfel incat, indiferent de timpul de pornire al celor doua fire, continutul fisierului afisat nu va contine textul “thread2“. Se presupune ca firul care citeste se va porni intotdeauna inaintea celui care adauga la sfarsitul fisierului.

Problema este asemanatoare cu cea din lucrarea ‘Sectiuni critice’, cu deosebirea ca cele doua fire care trebuie sa se sincronizeze se afla in procese diferite.

2. Realizati o aplicatie care la fiecare apasare a unui buton deschide cate o fereastra. Folosind un semafor limitati numarul maxim de ferestre ce pot fi deschise la un moment dat este 10. De fiecare data cand o astfel de fereastra este inchisa se va apela functia ReleaseSemaphore pentru a decrementa numarul de resurse ocupate.

La incercarea de deschidere a unei noi ferestre, pentru a verifica daca toate resursele semaforului sunt ocupate se poate utiliza codul urmator :

```
DWORD dwWaitResult;
dwWaitResult = WaitForSingleObject(hSemaphore, 0L);
switch (dwWaitResult)
{
    case WAIT_OBJECT_0:
        // Puneti aici codul care deschide fereastra
        break;
    case WAIT_TIMEOUT:
        // Numarul de ferestre este maxim.
        break;
}
```

3. a. Folosind un Waitable Timer realizati o aplicatie care deschide la interval de 10 secunde cate un mesaj de dialog care contine data si timpul deschiderii ferestrei.

b. Asemanator cu problema precedenta, limitati numarul de ferestre deschise la 5. Astfel chiar daca intervalul de 10 secunde s-a scurs, o noua fereastra se deschide daca nu exista deja 5 ferestre deja pe ecran.

12 Servicii Windows

12.1 Introducere

Un serviciu windows este un program care nu necesita ca un utilizator sa fie autentificat in system pentru a rula. Serviciile pornesc fie la bootarea calcuratorului, fie cand un utilizator il porneste prin intermediul managerului de control al serviciilor (SCM - Service Control Manager) fie cand o alta aplicatie porneste explicit serviciul. De remarcat ca anumite servicii pot porni la randul lor alte servicii de care acestea au nevoie.

In proictarea serviciilor, trebuie sa avem in vedere comunicarea cu SCM si sa le proiectam in conformitate cu necesitatile ce cimunicare ale acestuia.

SCM are o structura interna de date in care tine toate serviciile instalate si pune la dispozitie un mijloc unificat si sigur de a le controla. SCM este un server RPC ce porneste automat imediat dupa bootare.

12.2 Anatomia unui serviciu

Facand abstractie de partea de instalare si dezinstalare a serviciului care sunt triviale (si vor fi explicate mai tarziu in acesta lucrare), un serviciu este compus din doua functii. O functie principala a serviciului (ServiceMain) care este de fapt inima serviciului, si o functie care este apelata de catre Service Control Manager si care asigura comunicatia inapoi, dinspre SCM si serviciu insusi.

Pe scurt, un serviciu trebuie sa-i spuna SCM-ului care este functia ce trebuie apelata cand SCM schimba starea serviciului dupa care adduce serviciul in starea RUNNING. In acesta stare, ServiceMain face ce are de facut (si eventual asteapta dupa un eveniment trimis de catre functia apelata de SCM) si apoi isi termina executia ajungand in starea STOPPED.

12.3 Constrangeri ale implementarii

12.3.1 Interfata grafica sau linia de comanda

Serviciile pot avea o intrefata cu utilizatorul in linia de comanda sau chiar o intrefata grafica darn u este recomandata utilizarea unei interfate grafice din

motive lesne de inteles. Daca serviciul are nevoie de o interfata mai complexa pentru configurarea sa, se recomanda createa unei aplicatii separate in acst scop, aplicatie ce va comunica cu seviciul prin una din metodele de comunicare interproces existente. Cea mai utilizata tehnica este comunicarea prin socket-uri deoarece in acest fel se asigura o mobilitate in ce priveste locul de acces la serviciu (serviciul se poate configure de oriunde din Internet). Se mai folosesc si fisiere mapate dar in aces fel serviciul putand fi configurat de pe acelasi calculator. Se pot folosi si pipe-uri numite, in acst fel asigurandu-se o mobilitate ceva mai mare, interfata de configarare putand rula oriunde in reateaua locala in care ruleaza si serviciul.

12.3.2 Versiunea de sistem de operare suportata

De remarcat ca serviciile nu pot rula pe sistemul de operare Windows 95, doar NT/2000/XP oferind support in acest sens.

12.3.3 Securitate

In mod normal un serviciu este executat in contul lui Local System. Un serviciu nu poate executa in mod normal cod care necesita un user logat in system (ex. : nu poate folosi spre exemplu cheia din registry HKEY_CURRENT_USER).

12.4 Implementare

Pentru a crea un serviciu putem urma urmatorii pasi:

- Facem un program “Win32/Console Application”.
- Facem o functie ServiceMain – acesta fiind punctual de intrare al serviciului (un fel de main)
- Cream un maner pentru accesul SCM – functie ce comunica cu SCM numita si “Service Control Hadler”
- Cream un installer/unistaller ce inregistreaza un EXE ca un serviciu.

Sa incepem prin a analiza urmatorul exemplu:

```
#include “Winsvc.h”
main() {
```

```
    SERVICE_TABLE_ENTRY services[] = {{“Primul”,
ServiceMain}, {NULL,NULL}};
    StartServiceCtrlDispatcher(services);
}
```

Acest exemplu nu face altceva decat a umple un array SERVICE_TABLE_ENTRY. “Primul” este numele serviciului asa cum se regaseste in SCM. “ServiceMain” este numele functiei principale a serviciului. Cei doi NULL sunt doar pentru a marca finalul listei si nu sunt neaparat necesari. Daca dorim putem specifica mai multe servicii in arrayul mai sus mentionat astfel fiind posiblila existenta mai multor servicii in acelasi EXE.

12.4.1 ServiceMain

ServiceMain trebuie sa aiba urmatoarea declaratie:

```
void WINAPI ServiceMain(DWORD argc, LPTSTR *argv)
```

In principal, aceasta functie trebuie sa:

- umple structura SERVICE_STATUS cu valori
- inregistreze functia Service Control Handler (prescurtata SCH)
- cheme functiile actuale de procesare.

Pe parcursul ServiceMain un serviciu trece in mod normal prin urmatoarele stari: START_PENDING, RUNNING, STOP_PENDING, STOPPED.

ServiceMain poate accepta argumente din linia de comanda (ca o functie main). Argv contine cel putin un argument (numele serviciului).
Structura SERVICE_STATUS contine starea curenta a serviciului si trebuie trimisa SCM ului. Pentru a trimite acesta structura la SCM, se utilizeaza o functie API SetServiceStatus().
Cei mai important membri ai structurii SERVICE_STATUS sunt:

- dwServiceType ex.: SERVICE_WIN32;
- dwCurrentState ex.: SERVICE_START_PENDING;
- dwControlsAccepted ex.: SERVICE_ACCEPT_STOP.

ServiceMain la inceput seteaza SERVICE_START_PENDING. Acesta “spune” SCM-ului ca serviciul tocmai se starteaza. Daca intervine o problema, trebuie sa trimitem un SERVICE_STOPPED. In mod implicit SCM monitorizeaza serviciile si daca nu sesizeaza nici o activitate pe parcursul a 2 minute, SCM omoara serviciul.

Pentru inregistrarea functiei SCH se foloseste RegisterServiceControlHandler ce are doi parametrii (numele serviciului si functia SCH).

12.4.2 Functia ServiceControlHandler

Aceasta functie este chemata de catre SCM la orce actiune a utilizatorului (cum ar fi start, stop, pause sau continue). Acesta functie contine de obicei un switch case pentru fiecare din cazurile de mai sus.

Functia SCH primeste un opcode care poate avea valori ca: SERVICE_CONTROL_PAUSE/CONTINUE/STOP/INTEROGATE etc.

```
Void WINAPI aStudentExampleServiceControlHandler(DWORD
Opcode) {
     switch (Opcode) {
          case SERVICE_CONTROL_PAUSE:
               serviceStatus.dwCurrentState = SERVICE_PAUSED;
               break;
          case SERVICE_CONTROL_CONTINUE:
               serviceStatus.dwCurrentState =
SERVICE_RUNNING;
               break;
          case SERVICE_CONTROL_SHUTDOWN:
          case SERVICE_CONTROL_STOP:
               serviceStatus.dwWin32ExitCode = 0;
               serviceStatus.dwCheckPoint = 0;
               serviceStatus.dwWaitHint = 0;
               serviceStatus.dwCurrentState =
SERVICE_STOPPED;
               SetServiceStatus(serviceStatusHandle,
&serviceStatus);
               running = false;
```

```
                break;
          case SERVICE_CONTROL_INTEROGATE:
                break;
     }
}
```

“serviceStatus” este o variabila globala care pastreaza starea curenta a serviciului. Acesta structura este modificata si retrimisa SCM de cate ori starea serviciului se schimba.

“running” este o variabila booleana care este interogata in ServiceMain. ServiceMain trebuie sa se termine cand SCM trimite un opcode SERVICE_CONTROL_STOP. Acest lucru se asigura prin punerea variabilei running pe pozitia “false”.

12.5 Instalarea serviciilor

Pentru a instala un serviciu, trebuiesc facute cateva intrari in registry-ul sistem. Exista functii API pentru manipularea registry-ului dar se prefera utilizarea functiilor API speciale pentru servicii: CreateService() / DeleteService().
Pentru instalarea unui serviciu se deschide baza de date SCM (OpenSCManager) cu drepturile dorite de acces dupa care se lanseaza CreateService cu calea catre fisierul reprezentand serviciul ca parametru.

12.5.1 OpenSCManager

De regula acesta functie se foloseste in forma:

```
schSCManager = OpenSCManager(NULL, NULL,
SC_MANAGER_ALL_ACCESS);
```

Forma generala a aceste functii este:

```
SC_HANDLE OpenSCManager(
  LPCTSTR lpMachineName,
  LPCTSTR lpDatabaseName,
  DWORD dwDesiredAccess
);
```

Parametri:

- lpMachineName - [in] Pointer la un stringZ, care specifica numele calculatorului destinatie. Daca pointerul este NULL sau pointeaza la un sir gol, functia se connecteaza la SCM-ul de pe masina locala.
- lpDatabaseName – [in] Pointer la un stringZ care specifica numele bazei de date SCM ce va fi deschisa. Daca se pune NULL aici, se descige baza de date SERVICES_ACTIVE_DATABASE.
- dwDesiredAccess - [in] Accesul la SCM. Inaintea deschiderii, systemul verifica tokenul de securiate al procesului chemator.

.

12.5.2 CreateService

Cea mai des utilizata forma este:

```
schService=CreateService(schSCManager, “Primul”,
          “Primul exemplu de serviciu”, // numele prezentat
utilizatorului
          SERVICE_ALL_ACCESS, // nivelul de acces dorit
          SERVICE_WIN32_OWN_PROCESS, // tipul de serviciu
          SERVICE_DEMAND_START, // tipul de pornire
          SERVICE_ERROR_NORMAL,  //  tipul  de  control  al
erorilor
          lpszBinaryPathName,
          NULL, // fara grup de ordine a incarcarii
          NULL, // fara identificator de tag
          NULL, // fara dependinte
          NULL, // contul LocalSystem
          NULL); // fara parola
```

Dupa ce serviciul a fost creat este important sa nu uitam sa inchidem baza de date a SCM: CloseServiceHandle(schService).

Dezinstalarea serviciului se face cu :

```
HANDLE schSCManager;
SC_HANDLE hService;
  schSCManager =
OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);

  if (schSCManager == NULL)
    return false;

hService=OpenService(schSCManager,"Primul",SERVICE_ALL_ACCESS
);
  if (hService == NULL)
    return false;
  if(DeleteService(hService)==0)
    return false;
  if(CloseServiceHandle(hService)==0)
    return false;
```

12.6 Probleme

Creati un serviciu care sa porneasca odata cu initializarea sistemului de operare si care sa emita un beep de 1khz de lumgime 100ms la fiecare 5 secunde. Instalarea, dezinstalare si serviciu insusi vor fi in acelasi fisier exe chemat cu diversi parametri (ex: -I instalare, -D dezinstalare). Se prefera o implementare in care SCH comunica cu ServiceMain prin evenimente dar se accepta si o solutie prin polling.

13 Bibliografie

- J. Richter, Programming Applications for Microsoft Windows, Microsoft Press, 1999
- Microsoft Extensible Firmware Initiative, FAT32 File System Specification, 2000, Microsoft Corporation
- Silberschatz, P. B. Galvin, G. Gagne, Operating System Concepts, John Wiley & Sons, 2003

- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/service_security_and_access_rights.asp
- http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=96
- http://osdever.net/tutorials/chs_lba.php?the_id=87
- http://www.maverick-os.dk/FileSystemFormats/FileSystemIdentifiers.html
- http://www.maverick-os.dk/FileSystemFormats/Standard_DiskFormat.html
- http://www.pcguide.com/ref/hdd/file/struct-c.html
- http://www.pcguide.com/ref/hdd/file/structPartitions-c.html
- http://www.pctechguide.com/04disks.htm
- http://www.maverick-os.dk/FileSystemFormats/VFAT_LongFileNames.html
- http://www.maverick-os.dk/FileSystemFormats/FAT32_FileSystem.html
- http://hjem.get2net.dk/rune_moeller_barnkob/filesystems/vfat.html
- http://hjem.get2net.dk/rune_moeller_barnkob/filesystems/fat.html
- http://www.nondot.org/sabre/os/files/FileSystems/ext2fs/ext2fs_11.html
- http://www.nondot.org/sabre/os/files/FileSystems/ext2fs/
- http://www.nongnu.org/ext2-doc/ext2.html
- http://uuu.sourceforge.net/docs/ext2.php
- http://msdn.microsoft.com/