



UNIVERSITATEA DIN
BUCUREȘTI

FACULTATEA DE
MATEMATICĂ ȘI
INFORMATICĂ



SPECIALIZAREA INFORMATICĂ

Lucrare de licență

FILE SYSTEMS

Absolvent

Dana Mihai-Răzvan

Coordonator științific

Rusu Cristian

București, iunie 2024

Rezumat

Sistemele de fișiere reprezintă o componentă esențială în cadrul sistemelor de calcul, fără de care acestea nu ar putea funcționa. Ele constituie cadrul fundamental de stocare, organizare dar și gestionare a datelor utilizate de către un computer. Acestea reprezintă structurile de bază care le permit utilizatorilor scrierea, accesarea, modificarea și ștergerea, într-un mod eficient, a informațiilor aflate într-un mediu de stocare. Astfel, importanța sistemelor de fișiere nu se limitează doar la mediul informatic, ci are un impact semnificativ și în cadrul utilizării cotidiene a calculatoarelor.

În acest scop, în cadrul acestei lucrări, ne vom concentra pe explorarea detaliată a mai multor sisteme de fișiere, cum ar fi FAT32, Ext2 și Veritas, dar și a unui sistem modern de fișiere distribuit. Ne vom concentra pe aspectele distinctive ale fiecăruia dintre cele 3 sisteme de fișiere, evidențiind funcționalitățile principale ale acestora, cum ar fi crearea, modificarea, citirea și ștergerea datelor. Scopul este acela de a înțelege modul în care datele sunt organizate în cadrul unui mediu de stocare, și în special modul în care acestea sunt interpretate sub forma unor foldere și fișiere, astfel încât utilizarea lor să poată fi posibilă.

Abstract

File systems are an essential component within computing systems, without which they could not function. They constitute the fundamental framework for storing, organizing and managing the data used by a computer. These are the basic structures that allow users to efficiently write, access, modify, and delete information on a storage medium. Thus, the importance of file systems is not only limited to the computer environment, but also has a significant impact in the daily use of computers.

To this end, in this paper, we will focus on the detailed exploration of several file systems such as FAT32, Ext2 and Veritas, but also a modern distributed file system. We will focus on the distinctive aspects of each of the 3 file systems, highlighting their main functionalities such as creating, modifying, reading and deleting data. The goal is to understand how data is organized within a storage medium, and in particular how it is interpreted as folders and files so that its use can be made possible.

Cuprins

1	Introducere	4
1.1	Moțivatie	4
1.2	Domenii abordate	4
1.3	Structura lucrării	5
1.3.1	Implementarea Sistemelor de Fișiere	5
1.3.2	Compararea eficienței	5
2	Implementarea Sistemelor de Fișiere	6
2.1	Simularea disk-ului	6
2.1.1	Inițierea disk-ului	6
2.1.2	Scrierea si citirea sectoarelor	7
2.1.3	Probleme întâmpinate	8
2.2	FAT32	11
2.2.1	Privire de ansamblu asupra FAT32	11
2.2.2	Modul de implementare	12
2.2.3	Interacțiunea cu utilizatorul	17
2.2.4	Concluzii FAT32	19
2.3	Ext2	20
3	Compararea Eficienței	21
4	Concluzii	22

Capitolul 1

Introducere

1.1 Motivație

Dezvoltarea și implementarea sistemelor de fișiere a fost mereu o provocare în domeniul informaticii, iar pentru mine a reprezentat întotdeauna un subiect de interes. Am ales să mă concentrez pe această temă pentru că vreau să înțeleg mai bine cum funcționează stocarea, și în special, interpretarea, datelor de către sistemele de calcul.

Motivația din spatele acestei alegeri constă în dorința de a înțelege mai bine modul în care aceste sisteme funcționează, implementarea acestora, precum și impactul lor asupra gestionării, eficienței și prelucrării datelor în cadrul calculatoarelor. Prin implementarea și, mai apoi, analiza acestor sisteme, doresc să obțin o perspectivă mai clară asupra avantajelor și limitărilor fiecăruia, și să înțeleg situațiile în care un sistem de fișiere poate fi mai potrivit decât altul, și de ce.

1.2 Domenii abordate

Această lucrare va include următoarele 3 domenii principale:

- **Structura Sistemelor de Fișiere**, unde ne focusăm pe modul în care sunt implementate o serie de sisteme de fișiere, structura fiecăruia dintre acestea, și încercăm să înțelegem avantajele aduse de fiecare model în parte.
- **Gestiunea Datelor și Redundanța**, ne axăm pe organizarea datelor, implementând diverși algoritmi cu scopul reducerii redundanței, dar și a fragmentării, atât interne cât și externe.
- **Eficiența și Performanța Sistemelor de Fișiere**, aici ne concentrăm pe optimizarea operațiunilor de stocare, manipulare și ștergere a datelor, astfel încât să obținem timpi cât mai buni de răspuns.

1.3 Structura lucrării

Lucrarea de licență este împărțită în 2 capitole principale:

- **Implementarea Sistemelor de Fișiere:** Presupune implementarea a 3 sisteme de fișiere locale, și anume FAT32, Ext2 și Veritas, dar și a unui sistem de fișiere distribuit.
- **Compararea eficienței:** Compararea eficienței celor 3 sisteme de fișiere locale pentru diverse cazuri de utilizare.

1.3.1 Implementarea Sistemelor de Fișiere

Pentru a realiza implementarea celor 3 sisteme de fișiere locale, vom crea o bibliotecă care să simuleze operațiile de bază ale unui hard disk, cum ar fi scrierea și citirea sectoarelor. Acest hard disk simulat va fi reprezentat de un folder pe sistemul de operare local. Implementarea sistemelor de fișiere va utiliza această bibliotecă ca interfață între sistemul de fișiere și hard disk-ul simulat.

Pentru implementarea sistemului de fișiere distribuit, ne vom concentra pe crearea unui API pentru comunicarea între un server dedicat pentru stocarea fișierelor și clienții care doresc să acceseze aceste fișiere. Acest lucru va permite distribuirea datelor între mai multe calculatoare.

1.3.2 Compararea eficienței

Pentru a compara eficiența între sistemele de fișiere FAT32, Ext2 și Veritas, vom evalua mai multe aspecte, inclusiv viteza operațiunilor de scriere, modificare, accesare și ștergere pentru fișiere de diferite dimensiuni. De asemenea, ne vom concentra pe gestionarea fragmentării spațiului de stocare a datelor, analizând modul în care fiecare sistem de fișiere abordează acest aspect.

Vom efectua teste pentru a măsura timpul necesar executării operațiilor menționate mai sus, utilizând seturi de date variate, atât pentru fișiere de dimensiuni reduse, cât și pentru cele mai ample. Aceste teste vor fi realizate în condiții similare pentru fiecare sistem de fișiere, asigurând o comparație corectă.

De asemenea, va fi analizată și fragmentarea, atât internă, cât și externă, și modul în care aceasta este gestionată de către fiecare dintre cele 3 sisteme de fișiere.

Capitolul 2


Implementarea Sistemelor de Fișiere

2.1 Simularea disk-ului

2.1.1 Inițierea disk-ului

Pentru a putea simula disk-ul, vom crea un folder în cadrul sistemului de fișiere oferit de sistemul de operare pe care vor fi rulate mai apoi cele 3 sisteme de fișiere. Acest folder conține 2 fișiere simple, fără extensie: un fișier 'Metadata', care oferă informațiile necesare inițierii simulării (cum ar fi numărul de sectoare ale hard disk-ului și dimensiunea în bytes a fiecărui sector), odată ce aceasta a fost deja creată într-o dată anterioară, și un alt fișier 'Data', care reprezintă disk-ul simulat. Acest fișier este în esență doar un șir de caractere, care vor reprezenta byții din cadrul disk-ului nostru.

Pentru crearea celor două fișiere, precum și pentru scrierea și citirea acestora, a fost necesară utilizarea API-ului oferit de sistemul de operare în cadrul căruia este dezvoltată această lucrare de licență, și anume Windows 10. Acesta ne pune la dispoziție o serie de metode pentru manipularea fișierelor, prin intermediul bibliotecii "windows.h".





 Metadata Type: File	Date modified: 05/04/2024 09:19 Size: 6 bytes
 Data Type: File	Date modified: 05/04/2024 09:27 Size: 15.9 MB

Figura 2.1: Disk-ul simulat

În momentul în care este creat disk-ul, acesta nu conține nicio informație, toți byții fiind setați la 0. Pentru a realiza acest lucru în cadrul simulării, după crearea celor două fișiere, vom umple fișierul 'Data' cu caracterul null (adică valoare 0), în conformitate cu dimensiunea disk-ului, și anume (numărul de sectoare) * (dimensiunea unui sector). Aceste valori vor fi oferite ca și input de către utilizator la crearea disk-ului, împreună cu locația unde se dorește salvarea acestuia.

Figura 2.2: Vizualizare a disk-ului după creare

2.1.2 Scrierea si citirea sectoarelor

Pe lângă crearea și inițierea disk-ului, biblioteca pe care o creăm pune la dispoziție, în mod evident și o serie de funcții care ajută la scrierea și citirea datelor aflate pe disk. Scrierea și citirea byților a fost realizată într-un mod care să simuleze felul în care datele sunt extrase și modificate de pe un disk real, bineînțeles cu limitările pe care ni le impune faptul că în realitate lucrăm cu un fișier aflat în cadrul sistemului de operare Windows 10, și nu cu o componentă hardware adevărată. Astfel, fiecare operațiune de scriere sau citire afectează întregul conținut al unui sector, sau a mai multor sectoare consecutive.

Să luăm ca și exemplu, un disk care are dimensiunea sectoarelor de 512 bytes. În cazul în care dorim să citim byții aflați între adresele 800 și 1200, noi vom citi de fapt toți byții de la adresa 512 până la 1535, adică sectoarele 1 și 2 în totalitate (indexarea sectoarelor făcându-se de la 0), același lucru întâmplându-se și în cazul operației de scriere.

```

1 static int readSector(DiskInfo *diskInfo, uint32_t sector, char *buffer)
2 {
3     char *fullFilePath = buildFilePath(diskInfo->diskDirectory);
4
5     HANDLE fileHandle = CreateFile(fullFilePath, OFN_READONLY, 0, nullptr,
6                                     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, nullptr);
7
8     if(fileHandle == INVALID_HANDLE_VALUE)
9     {
10         CloseHandle(fileHandle);
11         delete[] fullFilePath;
12         return SECTOR_READ_FAILED;
13     }
14
15     OVERLAPPED overlapped;
16     memset(&overlapped, 0, sizeof(OVERLAPPED));
17     overlapped.Offset = diskInfo->diskParameters.sectorSizeBytes * sector;
18     overlapped.hEvent = nullptr;
19
20     DWORD dwBytesRead = 0;
21     bool readFileResult = ReadFile(fileHandle, buffer,
22                                     diskInfo->diskParameters.sectorSizeBytes, &dwBytesRead, &overlapped
23                                     );
24
25     if(!readFileResult || dwBytesRead < diskInfo->diskParameters.
26         sectorSizeBytes)
27     {

```

```

27     CloseHandle(fileHandle);
28     delete[] fullFilePath;
29     return SECTOR_READ_FAILED;
30 }
31
32 CloseHandle(fileHandle);
33 delete[] fullFilePath;
34
35 return SECTOR_READ_SUCCESS;
36 }

```

2.1.3 Probleme întâmpinate

Ideea din spatele acestei simulări a fost încă de la început aceea de a putea fi folosită, într-un stil similar, de toate cele 3 sisteme de fișiere care vor fi implementate folosindu-se de funcționalitățile oferite de aceasta. Astfel, în momentul finalizării librăriei, aceasta a trebuit să pună la dispoziție o interfață care urma să fie folosită în cadrul celor 3 proiecte, fără a mai fi schimbată ulterior, pentru a nu trebui să facem modificări în cadrul sistemelor de fișiere, datorită faptului că simularea disk-ului a suferit schimbări (adică s-a dorit ca toate modificările ulterioare să fie backwards compatible). Mai jos se poate vedea API-ul pus la dispoziție de către librărie, acesta nesuferind nicio modificare ulterioară de-a lungul dezvoltării acestei lucrări de licență.

```

1 DiskInfo* initializeDisk(const char* diskDirectory, uint32_t sectorsNumber,
    uint16_t sectorSize);
2
3 int fillDiskInitialMemory(DiskInfo *diskInfo, uint32_t batchSize);
4
5 DiskInfo* getDisk(const char* diskDirectory);
6
7 int getDiskStatus(DiskInfo *diskInfo);
8
9 int readDiskSectors(DiskInfo *diskInfo, uint32_t numOfSectorsToRead,
    uint32_t sector, char* buffer, uint32_t &numOfSectorsRead);
10
11 int writeDiskSectors(DiskInfo *diskInfo, uint32_t numOfSectorsToWrite,
    uint32_t sector, char buffer[], uint32_t &numOfSectorsWritten);

```

Deși această interfață a fost gândită astfel încât să nu necesite modificări ulterioare, asta nu înseamnă că implementarea sa nu a trecut prin anumite schimbări de-a lungul timpului. Pe lângă mici bug-uri care au fost reparate, cum ar fi ștergerea unui pointer null, sau dealocarea la momentul inoportun a unui buffer folosit la scrierea/citirea datelor, a fost necesară și o regândire totală a modului în care sectoarele sunt definite în cadrul simulării.

Așa cum am văzut mai sus, în momentul actual disk-ul în sine, este simulat cu ajutorul unui singur fișier 'Data', care conține toți byții din cadrul său, sectoarele nefiind separate în mod 'fizic' în interiorul acestui fișier, distincția dintre ele făcându-se doar la nivel logic. Acesta nu a fost însă cazul încă de la început, implementarea inițială presupunând existența unui fișier separat pentru fiecare sector în parte.



















 sector_0 Type: File	Date modified: 20/01/2024 13:25 Size: 512 bytes
 sector_1 Type: File	Date modified: 20/01/2024 13:25 Size: 512 bytes
 sector_2 Type: File	Date modified: 20/01/2024 13:25 Size: 297 bytes
 sector_3 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_4 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_5 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_6 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_7 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_8 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_9 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_10 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_11 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_12 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_13 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_14 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_15 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector_16 Type: File	Date modified: 20/01/2024 13:11 Size: 0 bytes
 sector 17	Date modified: 20/01/2024 13:11

Figura 2.3: Implementarea inițială a disk-ului

De ce am făcut această schimbare drastică? Ei bine, în timpul dezvoltării primului dintre cele 3 sisteme de fișiere, și anume FAT32, am observat probleme legate de timpul necesar creării disk-ului. Inițial am ignorat acest lucru, întrucât în cadrul dezvoltării erau suficiente disk-uri de dimensiuni reduse (512 KiB) pentru testele pe care le făceam, astfel și timpul inițierii fiind unul redus, însă odată cu evoluția implementării, teste pe fișiere de dimensiuni mai ample au devenit necesare, astfel și dimensiunea disk-ului trebuind să crească în mod corespunzător. În acest moment, problemele legate de performanța

instanțierii disk-ului au devenit tot mai vizibile, devenind chiar un impediment major în procesul de testare. Spre exemplu, crearea unui disk de 1 GiB dura inițial aproximativ 30 de minute, spre deosebire de câteva secunde, cât durează cu implementarea actuală, care folosește un fișier comun, și nu câte unul pentru fiecare sector.

Problemele de performanță, însă, nu erau cauzate doar de existența unui număr ridicat de fișiere, ci și a modului în care se realiza popularea inițială a datelor din cadrul disk-ului, cu caracterul null (valoarea 0).

```
1 int fillDiskInitialMemory(DiskInfo *diskInfo, uint32_t batchSize);
```

Astfel, au existat 3 moduri de 'umplere' a disk-ului, până să se ajungă la varianta finală, care este bineînțeles și cea mai eficientă.

- **Popularea fiecărui fișier ce constituia un sector**, aceasta fiind varianta folosită în cadrul implementării inițiale, care folosea câte un fișier separat pentru reprezentarea fiecărui sector, și astfel, scrierea acestora făcându-se în mod independent.
- **Popularea independentă a fiecărui sector logic**, aici vorbim deja de cea de-a doua implementare a disk-ului, care folosește un singur fișier, și unde sectoarele sunt separate doar la nivel logic. Cu toate acestea, popularea fiecărui sector se realiza separat, acest lucru presupunând un număr ridicat de operațiuni de deschidere, scriere și apoi închidere asupra fișierului 'Data', ceea ce ducea la o performanță scăzută.
- **Popularea sectoarelor în batch-uri**, această variantă finală presupune exploatarea faptului că sectoarele nu sunt separate în mod fizic, ci sunt poziționate consecutiv în interiorul fișierului 'Data'. Astfel, pentru a reduce numărul operațiunilor asupra acestui fișier, popularea lor se realizează în batch-uri de mai multe sectoare.

2.2 FAT32

2.2.1 Privire de ansamblu asupra FAT32

FAT32 este un sistem de fișiere relativ simplu, care presupune utilizarea unei tabele de indici (File Allocation Table) pentru înlănțuirea unor blocuri de date numite cluster. Din punct de vedere structural, FAT32 se împarte în 4 regiuni:

- **Reserved Area**, această zonă conține date necesare pentru pornirea și funcționarea sistemelor din cadrul calculatorului. În prima parte a acestei regiuni se află 'Boot Record', care conține codul ce trebuie executat inițial pentru pornirea sistemului, totodată aici aflându-se o serie de structuri de date necesare sistemului nostru de fișiere, și anume BIOS Parameter Block (BPB), Extended Boot Record și FSInfo.
- **FAT 1**, regiune ce reprezintă tabela de indici menționată mai sus. Aceasta conține o serie de valori de 32 de biți (dintre care doar ultimii 28 sunt luați în considerare, primii 4 fiind ignorați), fiecare astfel de valoare reprezentând numărul următorului cluster în cadrul înlănțuirii actuale (a directorului curent). Spre exemplu, dacă la indexul 16 (adică byte-ul 64) se află valoarea 783, înseamnă că datele aflate în continuarea celor din cluster-ul 16 se află în cluster-ul 783.
- **FAT 2**, este o copie a FAT 1, servind în special ca și un backup, în cazul posibilelor erori ce pot apărea în tabela principală.
- **Data Area**, această regiune conține datele propriu-zise, înglobând folderele și fișierele stocate. Prima parte a acestei zone este numită 'Root directory', și se comportă ca un director obișnuit, acesta putând conține atât sub-foldere cât și fișiere. Aspectul distinctiv al acestuia însă, este faptul că este singurul director ce nu are un părinte, el reprezentând rădăcina arborelui de directoare stocate în cadrul sistemului de fișiere.

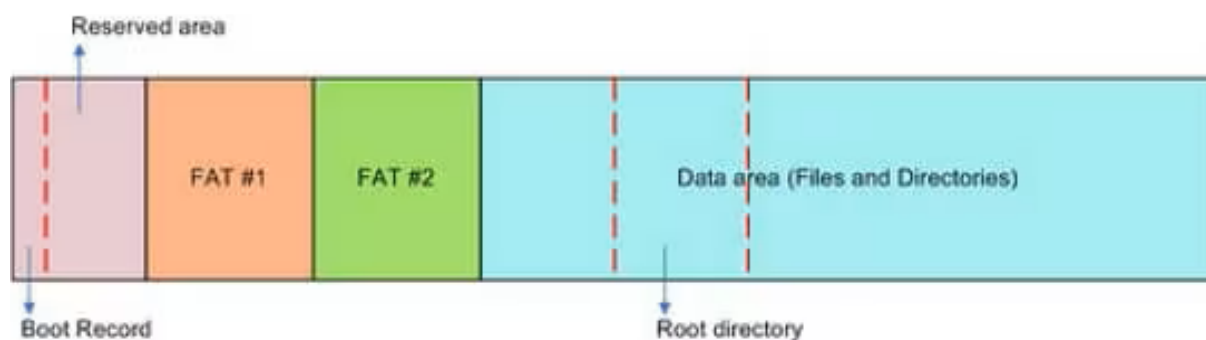


Figura 2.4: Structura FAT32

2.2.2 Modul de implementare

Fiind în esență doar o simulare (la fel ca toate cele 3 sisteme de fișiere din cadrul acestei lucrări), implementarea FAT32 a trebuit adaptată astfel încât să funcționeze fără a avea acces la hardware, ci doar la disk-ul nostru simulat, și fără a face parte dintr-un sistem de operare. Singura interacțiune de acest fel este una indirectă, prin intermediul simulării disk-ului, atunci când se realizează operațiuni asupra datelor stocate de acesta.

Din aceste considerente, simularea de FAT32 rulează sub forma unui program, cu care utilizatorul poate interacționa prin intermediul terminalului. Așa cum am precizat și mai sus, simularea noastră nu are legătură în mod direct cu sistemul de operare, ca în cazul unui sistem de fișiere adevărat. Aceasta duce la imposibilitatea ca structurile de date de care avem nevoie în cadrul FAT32 să se afle în mod permanent în memorie (RAM), și totodată la lipsa unor procese special dedicate care să ruleze în momentul în care se dorește realizarea unei operațiuni asupra disk-ului (cum ar fi scrierea sau citirea datelor). De aceea, am considerat ca rularea programului de simulare este singura soluție viabilă în cazul nostru, pentru a putea interacționa cu sistemul de fișiere.

Inițializarea FAT32

În momentul în care programul este rulat, acesta parcurge mai multe etape în vederea pregătirii disk-ului, cât și a sistemului de fișiere propriu-zis:

1. **Verificarea existenței disk-ului**, aici se verifică dacă disk-ul a fost deja inițializat la locația dată ca și parametru.
2. **Inițializarea/Citirea disk-ului**, această etapă depinde de rezultatul pasului anterior. Dacă simularea disk-ului nu a fost găsită la locația precizată, acesta este instanțiat: se creează fișierele 'Metadata' și 'Data', acesta din urmă fiind populat cu null (valoarea 0), în conformitate cu dimensiunea dorită. Dacă disk-ul a fost deja inițiat, atunci se citesc datele referitoare la acesta și se stochează într-o structură de date, DiskInfo, care ne va fi de folos pe parcurs.
3. **Verificarea inițializării sistemului de fișiere**, acest pas realizează citirea primului sector de pe disk, adică a sectorului de bootare. Dacă la finalul acestui sector se află semnătura specifică sistemului de fișiere FAT32, și anume 0xAA55, înseamnă că sistemul a fost deja inițiat, sau nu, în caz contrar.
4. **Inițializarea sistemului de fișiere**, etapă ce are loc doar în cazul în care verificarea de mai devreme returnează fals. În cadrul acestui pas sunt create datele din cadrul BIOS Parameter Block și a Extended Boot Record, care vor fi stocate împreună în cadrul unei structuri numite BootSector, pentru ca mai apoi să fie salvate în sectorul de bootare (sectorul 0), bineînțeles împreună cu semnătura reprezentativă

FAT32 despre care am vorbit mai sus. Tot aici este creată o altă structură de date, FSInfo, aceasta fiind salvată în sectorul 1. De asemenea, sunt salvate și câteva date esențiale referitoare la Directorul Radăcină (cum ar fi dimensiunea, primul cluster etc.), în cadrul intrării 'dot' a acestuia.

5. **Citirea structurilor de date necesare**, indiferent de rezultatul de la pasul anterior, citim structurile BootSector si FSInfo de pe disk.
6. **Inițializarea tabelelor de indici**, această etapă se realizează doar în cazul în care sistemul de fișiere a fost inițiat anterior (la pasul 4). Specificația FAT32 precizează faptul că Zona de Date începe întotdeauna de la cluster-ul cu numărul 2, de aceea setăm în tabelul de indici primele 2 cluster-uri ca fiind rezervate, iar cel de-al treilea (care aparține Directorului Radăcină) ca fiind cluster final în cadrul înlanțuirii.

```
1 void fat32Startup(char* diskDirectory, DiskInfo** diskInfo,
2                 BootSector** bootSector, FsInfo** fsInfo,
3                 uint32_t sectorsNumber, uint32_t sectorSize)
4 {
5     if(checkDiskInitialization(diskDirectory) == false)
6         initializeDisk(diskDirectory, diskInfo, sectorsNumber, sectorSize);
7     else
8         *diskInfo = getDisk(diskDirectory);
9
10    bool fat32AlreadyInitialized = true;
11    if(checkFat32FileSystemInitialization(*diskInfo) == false)
12    {
13        initializeBootSectors(*diskInfo);
14        fat32AlreadyInitialized = false;
15        std::cout << "Boot sectors initialized\n";
16    }
17
18    *bootSector = readBootSector(*diskInfo);
19    *fsInfo = readFsInfo(*diskInfo, *bootSector);
20
21    if(fat32AlreadyInitialized == false)
22    {
23        initializeFat(*diskInfo, *bootSector);
24        std::cout << "File allocation table initialized\n";
25    }
26 }
```

Structura unui director

Directoarele din cadrul implementării noastre de FAT32 pot fi de 2 tipuri: folder sau fișier. Modul în care acestea sunt stocate în Zona de Date este unul asemănător, însă

diferența o face scopul acestora și tipul de date pe care acestea îl pot îngloba. Un lucru comun pe care îl au ambele tipuri de directoare sunt intrările speciale 'dot' și 'dotdot', având lungimea de 4 bytes și aflându-se la începutul fiecărui director. Printre altele, intrarea 'dot' conține valoarea cluster-ului în care se află, în timp ce intrarea 'dotdot' conține valoarea primului cluster al părintelui, astfel acestea ajutând la traversarea arborelui de directoare ce se formează în cadrul sistemului de fișiere.

Datele conținute într-un folder sunt de tipul DirectoryEntry, o structură care are lungimea de 4 bytes și care conține informații referitoare la un director copil, cum ar fi numele său, dimensiunea, valoarea primului cluster sau date temporale. Pentru fiecare urmaș direct al unui folder, acesta conține o intrare de acest fel, intrările fiind poziționate consecutiv în cadrul clusterelor. Datele aflate în clusterelor corespunzătoare unui folder vor fi interpretate doar sub forma acestor tipuri de intrări, ceea ce înseamnă că un folder nu poate conține și alt tip de date.

```
1 typedef struct
2 {
3     uint8_t FileName[11];
4     uint8_t Attributes;
5     uint8_t Reserved;
6     uint8_t CreationTimeTenths;
7     uint16_t CreationTime;
8     uint16_t CreationDate;
9     uint16_t LastAccessedDate;
10    uint16_t FirstClusterHigh;
11    uint16_t LastWriteTime;
12    uint16_t LastWriteDate;
13    uint16_t FirstClusterLow;
14    uint32_t FileSize;
15 } __attribute__((packed)) DirectoryEntry;
```

Spre deosebire de foldere, un fișier obișnuit conține numai date simple, care nu sunt interpretate sub nicio formă de către sistemul de fișiere. Ele sunt scrise și citite fără a fi analizate sau modificate în vreun fel.

Alocarea de clustere

Fie că dorim să creăm un director sau să extindem unul deja existent, este necesară realizarea unei alocări de clustere, un proces care se desfășoară în mai mulți pași. Alocarea unui nou cluster se realizează atunci când ultimul cluster din cadrul directorului nu mai conține suficienți bytes liberi pentru a cuprinde întreaga cantitate de date care se dorește a fi scrisă.

În prima fază, se caută un cluster liber, fără a avea importanță poziția sau vecinii

acestui pe disk. Pentru a realiza această căutare, se parcurge tabelul de indici până când este găsită valoarea 0, reprezentând un cluster liber.

Dacă această căutare are succes, clusterul găsit este marcat în tabelul de indici cu valoarea 0xFFFFFFFF, reprezentând finalul unei înlănțuiri de cluster (END OF CHAIN). În cazul în care acest nou cluster nu este primul din cadrul unui director, trebuie actualizată și valoarea din tabel a precedentului ultim cluster, astfel încât acesta să indice către cel nou.

Căutarea unui director

Indiferent ce fel de operație dorim să realizăm asupra unui director, primul pas care trebuie făcut este să-i descoperim locația, mai exact să aflăm care este primul său cluster. Pentru asta, trebuie parcurs arborele de directoare, pornind de la Directorul Rădăcină, până când ajungem la destinația dorită. Astfel, este folosită o funcție recursivă, care iterează prin DirectoryEntry-urile folder-ului curent, în căutarea unui director care să aibă denumirea pe care o căutăm. Odată găsit directorul căutat, putem afla informațiile necesare despre acesta, precum primul său cluster sau dimensiunea sa prin intermediul structurii de date menționate mai devreme.

Crearea directoarelor

Crearea unui director, fie că acesta este de tip folder sau fișier obișnuit, este un proces complex, care presupune interacțiunea cu majoritatea funcționalităților din cadrul implementării noastre.

Înainte de începerea instanțierii propriu-zise a acestui nou director, se verifică faptul că datele oferite sunt valide: se verifică dacă numele respectă formatul corespunzător și nu conține caractere interzise, se verifică existența directorului oferit drept părinte, se verifică faptul că acesta este de tipul folder, iar în final ne asigurăm că nu există un alt director cu numele similar celui pe care dorim să-l cream, deoarece duplicarea denumirilor nu este permisă.

După aceea, se caută un cluster disponibil, care este alocat noului director, iar apoi sunt introduse intrările 'dot' și 'dotdot' în prima parte a acestuia. În final, este creat un DirectoryEntry corespunzător și este adăugat în ultimul cluster al părintelui.

Scrierea fișierelor

Când vine vorba despre scrierea datelor în fișiere, există 2 moduri prin care putem face acest lucru în cadrul implementării noastre:

- **Modul TRUNCATE**, care presupune suprascrierea byților deja existenți, adăugarea datelor realizându-se începând cu primul byte disponibil, adică byte-ul 64 din cadrul fișierului (primii 64 bytes fiind ocupați de intrările 'dot' și 'dotdot').

- **Modul APPEND**, în acest mod, datele pe care dorim să le scriem sunt adăugate în continuarea celor existente deja, fără a se produce vreo suprascriere, deci fără a suferi vreo pierdere de informații.

Pentru ca operațiunea de scriere să fie posibilă, fișierul nostru trebuie să aibă alocat un număr suficient de clustere. În cazul scrierii în modul truncate, dacă dimensiunea datelor este mai redusă decât dimensiunea actuală a fișierului, atunci nu mai trebuie adăugate clustere, ci dimpotrivă, este posibil să fie necesară eliberarea unei părți din acestea. În caz contrar, sau în cazul în care operăm în modul append, alocarea de noi clustere este un lucru ce se va întâmpla în cele mai multe situații. După alocarea clusterelor și scrierea datelor, se realizează actualizarea informațiilor despre fișierul actual în DirectoryEntry-ul corespunzător acestuia aflat în părintele său, cât și intrările sale 'dot' și 'dotdot'.

Citirea fișierelor

Citirea datelor din cadrul unui fișier se poate realiza atât în totalitate, cât și parțial. Implementarea noastră permite citirea unui anumit număr de bytes, începând de la o poziție de start. Această operațiune se realizează destul de simplu, fiind necesară doar identificarea fișierului dorit, iar apoi parcurgerea clusterelor sale, începând cu poziția inițială dată, până când numărul dorit de bytes a fost citit (sau până când am ajuns la finalul fișierului).

Eliberarea memoriei fișierelor

În cazul în care dorim să eliminăm o parte dintr-un fișier, cu scopul eliberării spațiului, acest lucru este posibil prin intermediul unei operații de truncate. Aceasta se realizează prin marcarea ca neocupate în cadrul tabelii de indici a clusterelor ce se doresc a fi eliberate, cât și prin reducerea dimensiunii fișierului, în cadrul structurilor ce rețin informațiile referitoare la acesta.

Ștergerea directoarelor

Dacă dorim să ștergem în totalitate un director, lucrurile pot deveni mai complicate. La fel ca și în cazul eliberării memoriei, clusterelor trebuie marcate ca ocupate, iar datele referitoare la director trebuie șterse din părintele acestuia. În cazul folderelor, trebuie realizată și ștergerea moștenitorilor direcți, cât și indirecti ai acestuia. Pentru acest lucru, se folosește o metodă recursivă, care utilizează un algoritm breadth-first. Acesta începe cu ștergerea 'frunzelor' din cadrul arborelui de directoare și urcă, până se ajunge la folderul a cărui ștergere a fost cerută.

2.2.3 Interacțiunea cu utilizatorul

Pentru a putea interacționa cu sistemul de fișiere, în timpul rulării simulării acestuia, utilizatorul are posibilitatea de a introduce de la terminal o serie de comenzi ce realizează operațiile descrise în detaliu mai sus, până când acesta introduce comanda 'exit' ce semnalează dorința de a închide programul.

```
1 while(true)
2     {
3         std::cout << '\n';
4         std::getline(std::cin, command);
5
6         if(command == "exit")
7         {
8             std::cout << "Process terminated with exit";
9             break;
10        }
11
12        std::vector<std::string> tokens = splitString(command, ' ');
13
14        if(tokens[0] == "mkdir")
15            commandCreateDirectory(diskInfo, bootSector, tokens);
16        else if(tokens[0] == "ls")
17            commandListSubdirectories(diskInfo, bootSector, tokens);
18        else if(tokens[0] == "write")
19            commandWriteFile(diskInfo, bootSector, tokens);
20        else if(tokens[0] == "read")
21            commandReadFile(diskInfo, bootSector, tokens);
22        else if(tokens[0] == "truncate")
23            commandTruncateFile(diskInfo, bootSector, tokens);
24        else if(tokens[0] == "rmdir")
25            commandDeleteDirectory(diskInfo, bootSector, tokens);
26        else if(tokens[0] == "la")
27            commandShowDirectoryAttributes(diskInfo, bootSector, tokens);
28        else
29            std::cout << "Unknown command \n";
30    }
```

Comenzile ce pot fi introduse de la terminal sunt:

- **ls (Nume folder)** sau **ls -l (Nume folder)**: *ls Root/MyFile* sau *ls -l Root/MyFile* - listează sub-directoarele unui folder, cea de-a doua comandă afișând și dimensiunea acestora.
- **write (Nume fișier) (Număr bytes) (Mod scriere) (Terminator)**: *write Root/MyFile 10000 TRUNCATE EOF* - scrie în fișierul dat un număr maxim de

bytes dat într-unul din cele 2 moduri, TRUNCATE sau APPEND, citirea byților realizandu-se de la terminal până când se întâlnește un rând care conține terminatorul dat.

- **read (Nume fișier) (Poziție start) (Număr bytes):** *read Root/MyFile 100 10000* - citește din fișierul dat un număr maxim dat de bytes, începând cu o anumită poziție dată.
- **truncate (Nume fișier) (Dimensiune):** *truncate Root/MyFile 100* - reduce dimensiunea unui fișier la un număr dat de bytes.
- **rmdir (Nume director):** *rmdir Root/MyFolder* - șterge un director, împreună cu toți descendenții săi, atât direcți cât și indirecti.
- **la (Nume director):** *la Root/MyFile* - arată informații detaliate despre un director.

```
la Root/Dir_1/File_1
Size: 164
Size on disk: 2048
Creation - 2024:4:10 - 22:59:0
Last accessed - 2024:4:10
Last write - 2024:4:10 - 22:59:26

rmdir Root/Dir_1
Successfully deleted this directory!

ls -l Root
Subdirectories for Root:

exit
Process terminated with exit
Process finished with exit code 0
```

Figura 2.5: Interacțiunea cu sistemul de fișiere FAT32

2.2.4 Concluzii FAT32

În încheiere, FAT32 este un sistem de fișiere relativ simplu, care de-a lungul timpului a dat dovadă de versatilitate și stabilitate, acesta fiind o alegere ideală pentru dispozitivele de stocare de dimensiuni medii. Cu toate acestea, din cauza limitărilor sale, cum ar fi dimensiunea maximă a fișierelor, precum și a securității slabe oferite de acesta, FAT32 poate fi neadecvat pentru stocarea unor fișiere de dimensiuni mari, sau pentru gestionarea datelor sensibile. În concluzie, FAT32 este o alegere ideală datorită compatibilității pe care o oferă, însă poate fi limitat de anumite caracteristici ale sale.

2.3 Ext2

Va urma..

Capitolul 3

Compararea Eficienței

Capitolul 4

Concluzii