UNIVERSIDADE FEDERAL DE UBERLÂNDIA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

PROGRAMA UNDELETE PARA SISTEMAS DE ARQUIVOS FAT32

Arthur do Prado Labaki
Marco Túlio Flores Melo
Vinnicius Pereira da Silva

Uberlândia - MG
2022

Sumário

1 Introdução

O objetivo deste trabalho é criar um programa capaz de recuperar arquivos em sistema de arquivos *FAT32*. Para desenvolver o programa, são requisitados conhecimentos de sistema de arquivos *FAT32* (*File Allocation Table*) e algumas bibliotecas que permitirão a criação do código-fonte.

FAT32 é um sistema de arquivos que gerencia o acesso a arquivos em *HDs* e outras mídias. Ele é um sistema de arquivos legado que é simples e robusto, que oferece bom desempenho mesmo em implementações muito leves, mas não pode oferecer o mesmo desempenho, confiabilidade e escalabilidade de alguns sistemas de arquivos modernos. É, no entanto, suportado por razões de compatibilidade por quase todos os sistemas operacionais atualmente desenvolvidos para computadores pessoais e muitos computadores domésticos, dispositivos móveis e sistemas embarcados e, portanto, é um sistema de arquivos adequado para troca de dados entre computadores e dispositivos de quase qualquer tipo e idade de 1981 até o presente (Wikipedia, 2021).

1.1 Estrutura do sistema de arquivos *FAT32*

O layout da *FAT32* é relativamente simples, o primeiro setor é sempre o ID do Volume, que é seguido por algum espaço não utilizado chamado de setores reservados. Após os setores reservados estão duas cópias da *FAT32*. O restante do sistema de arquivos são dados organizados em *clusters*, com talvez um pouco de espaço não utilizado após o último *cluster* (PJRC: Electronic Projects, 2005). *Cluster* (também chamado de bloco) é uma unidade de alocação de espaço em disco para arquivos e diretórios que não iremos aprofundar pois o mesmo já foi discutido em aula.

Dos setores reservados, o primeiro é o setor de inicialização (também chamado de *Volume Boot Record* ou simplesmente VBR). Ele inclui uma área chamada *BIOS Parameter Block* (BPB) que contém algumas informações básicas do sistema de arquivos. Os setores reservados ainda incluem um setor de informações do sistema de arquivos e um setor de inicialização de backup, além de poderem conter outros setores mais reservados (Wikipedia, 2021).

Na região da *FAT*, normalmente contém duas cópias da Tabela de Alocação de Arquivos para verificação de redundância, embora raramente usada. Esses são mapas da Região de Dados, indicando quais *clusters* são usados por arquivos e diretórios que pode ser visto na figura 2.

Por fim, temos a região de dados, em que os dados reais do arquivo e do diretório são armazenados e ocupam a maior parte da partição. Normalmente os *cluster* têm pelo menos 4k (8 setores), e os tamanhos de 8k, 16k e 32k também são amplamente utilizados. Algumas versões posteriores do *Microsoft Windows* permitem o uso de tamanhos de *cluster* ainda maiores,

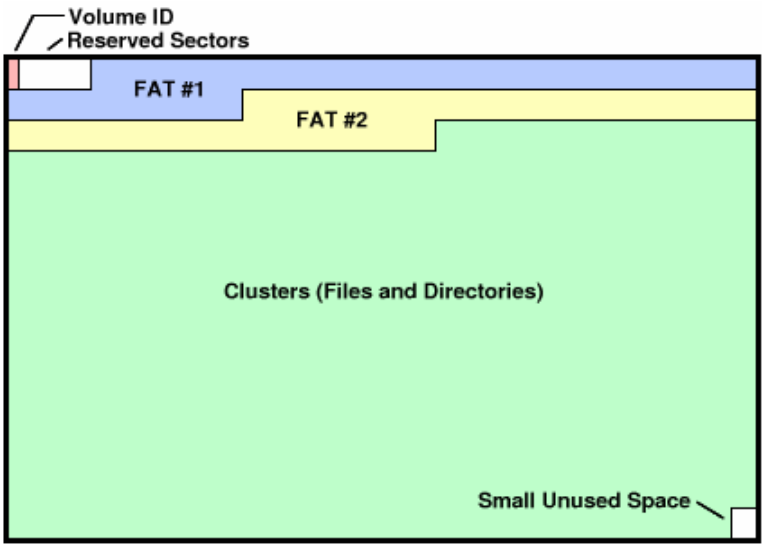

Figura 1 – Layout geral do sistema de arquivos *FAT32*

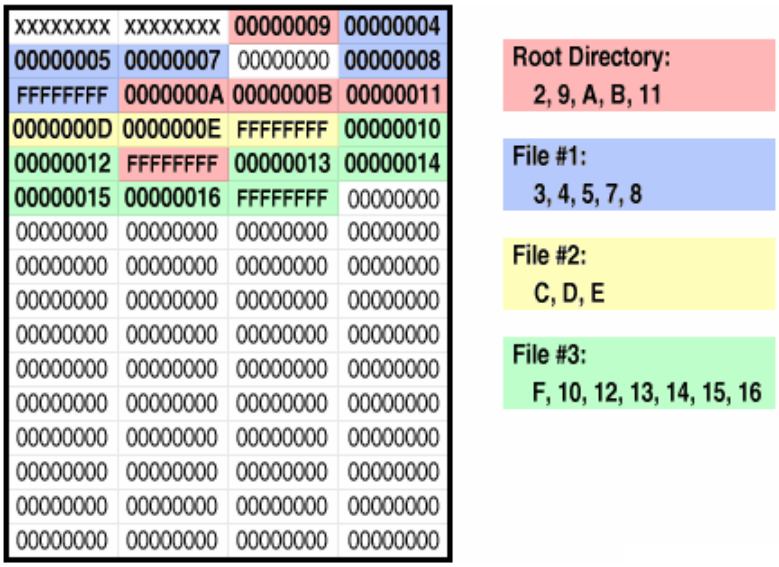

Figura 2 – Exemplo de um setor *FAT32*, com cadeias de *cluster* para diretório raiz e três arquivos

considerando efetivamente o tamanho do setor como um múltiplo de 512 bytes (MICROSOFT, 2014). A especificação *FAT32* da Microsoft afirma que 32k é o tamanho máximo do *cluster*.

O FAT32 normalmente inicia a tabela de diretório raiz no primeiro *cluster* da região de dados. Essa tabela de diretórios armazena informações sobre os arquivos e diretórios localizados no diretório raiz. Cada arquivo ou subdiretório armazenado dentro dele é representado por uma entrada de 32 *bytes* na tabela. Cada entrada registra o nome, extensão, atributos, o endereço do primeiro *cluster* dos dados do arquivo ou diretório, o tamanho do arquivo ou diretório, além de outras informações, como pode ser analisado na figura 3.

Algumas informações sobre nome e atributos são importantes para o entendimento dos códigos do programa proposto. É importante saber que o primeiro caractere do nome na estrutura de diretórios pode ser modificado. Se ele for **0x2E** ele representa um ponto ("."ou "."ou ".."),

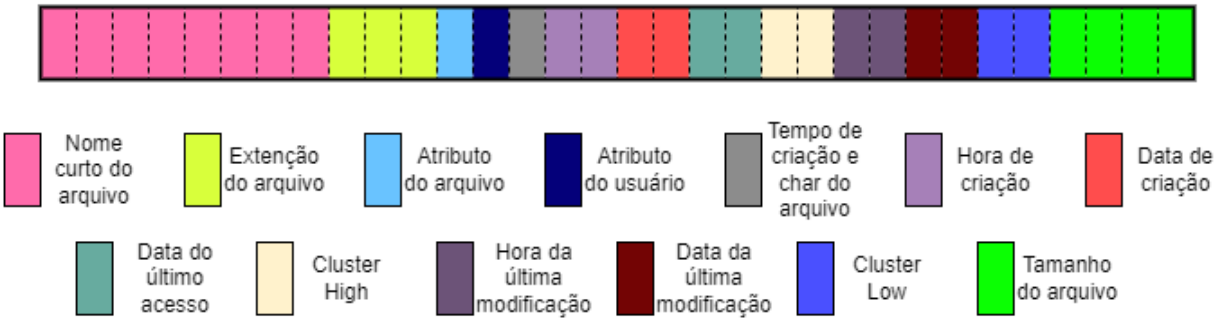

Figura 3 – Estrutura de diretório completa de 32 *bytes*

se ele for **0xE5** representa um arquivo que já foi excluído e se ele for **0x00** ele representa que ele está disponível e nenhuma entrada subsequente está em uso. Também é importante saber o significado do conteúdo do atributo do arquivo, que está representado na tabela.

Bit	Máscara	Função
0	0x01	Somente leitura
1	0x02	Oculto
2	0x04	Sistema
3	0x08	ID do Volume
4	0x10	Subdiretório
5	0x20	Arquivo
6	0x40	Dispositivo
7	0x80	Reservado

Tabela 1 – Atributo do arquivo

1.2 Inserção, remoção e recuperação

A inserção de arquivos em *FAT* não é muito difícil de ser compreendida. Nela ocorre a inserção do arquivo na região de dados, alocando-os em um ou mais *cluster* em cadeias, como demonstrado na figura 2. É feito seu mapeamento na Tabela de Alocação de Arquivos, indicando a localização dos *cluster* usados por aquele arquivo. A escolha dos *clusters* não é feita de forma organizada, podendo fazer com que a cadeia de *clusters* pule alguns (como o file 1 na figura 2) ou até voltando (começando no *cluster* 12 e seu próximo *cluster* ser o 7). Ainda é realizado a inserção de informações do arquivo na tabela de diretório raiz, armazenando em uma entrada de 32 *bytes* as informações vistas na figura 3.

A remoção de arquivos não ocorre da forma com que a maioria das pessoas inexperientes acreditam. Na remoção, o arquivo não é apagado, ele apenas deixa de ser referenciado. Ele ainda é mantido, na região de dados, em seus respectivos *clusters*, além do mapa na tabela de alocação. Porém seu primeiro byte no registro de entrada na tabela de diretório raiz é alterado para **0xE5**, sinalizando para o sistema que o arquivo acessado foi excluído. Posteriormente quando um novo

arquivo for adicionado, ele poderá sobrescrever os dados daquele arquivo excluído, alterando alguns de seus *clusters* (PJRC: Electronic Projects, 2005).

Com isso, a função *undelete*, que significa recuperação, ou também, deixar de ser deletado, consiste em alterar o arquivo para que seu acesso se torne novamente acessível. Em *FAT32*, essa função é possível pois como o arquivo apenas foi deixou de ser referenciado, basta referencia-lo novamente, alterando seu registro de entrada na tabela de diretório raiz, além de analisar seus *clusters*, verificando se algum *cluster* não foi sobrescrito.

2 Programa

2.1 Informações gerais

Para realizar esse trabalho, além de bibliotecas padrões do **C**, utilizamos a biblioteca *OpenSSL*. Ela especifica informações e atributos requeridos para a identificação de uma pessoa ou sistema de computador e é usado para administração de segurança e distribuição certificados assinados digitalmente através de redes de internet seguras (OpenSSL, 2021).

Dentre suas diversas funções, usaremos nesse trabalho o *SHA-1* (*Secure Hash Algorithm* ou algoritmo de dispersão seguro) que é uma função de dispersão criptográfica (ou função *hash* criptográfica). Ela produz um valor de dispersão de 160 *bits* (20 *bytes*) conhecido como resumo da mensagem. Um valor de dispersão *SHA-1* é normalmente tratado como um número hexadecimal de 40 dígitos (The OpenSSL Project Authors, 2021). Usaremos essa função para evitar ambiguidade que será discutido mais à frente.

Também é importante ressaltar que todo o trabalho foi totalmente desenvolvido em *Kali Linux*, que é uma é uma distribuição *Linux* de código aberto, baseada no *Debian* (OffSec Services Limited, 2022). Com isso, as instruções de compilação podem não funcionar corretamente dependendo do tipo de distribuição ou sistema operacional usado, sendo necessário uma pesquisa especifica para verificar os comandos equiparáveis.

2.2 Instrução para compilação

Para conseguir compilar o código sem problemas, é importante manter todos os pacotes instalados sempre atualizados. Para isso, utilizaremos dois comandos opcionais:

- *sudo apt-get update*
- *sudo apr-get upgrade*

Também utilizaremos um comando para instalar o pacote *OpenSSL* já apresentado:

- *sudo apt-get install openssl*

Para instalar o nosso programa, é necessária uma sequência de comandos para compilar todos os arquivos. Porém foi criado um arquivo *Makefile* para facilitar essa parte. Arquivos *Makefile* são usados para automatizar o processo de construção de aplicações chamando o compilador, linkeditor, executando testes, e até mesmo fazendo o *deploy*, entre outras operações possíveis. No nosso caso, será necessário apenas compilar todos os arquivos com um simples e único comando:

- *make*

Também é necessário um arquivo binário do disco *FAT32* para executar o programa, no qual será ensinado a realizar essa criação na próxima seção. Com isso, será ensinado montar essa imagem de disco em um diretório.

2.3 Formatando em *FAT32*

Para criar a imagem de disco em *FAT32* ou até para montar essa imagem em um diretório, siga os passos a seguir:

1. Crie um arquivo de imagem vazio que contenha zeros.

   - *dd if=/dev/zero of=fat32.disk bs=256k count=1*

   O comando **dd if=/dev/zero** cria o arquivo de imagem contendo zeros, em seguida o nome do arquivo, seu tamanho e quantidade (respectivamente **of, bs, count**).

2. Converta o arquivo criado para um arquivo de imagem em *FAT32*.

   - *mkfs.fat -F 32 -f 2 -S 512 -s 1 -R 32 fat32.disk*

   mkfs.fat: gera uma imagem de disco de *FAT*;
   -F: relaciona o tipo de *FAT* (*FAT12, FAT16, FAT32*);
   -f: o número de *FATs*;
   -S: o numero de bytes por setor;
   -s: o numero de setores por *cluster*;
   -R número de setores reservados.

3. Ainda é possível usar o comando abaixo para verificar os detalhes do sistema de arquivos *FAT* criado.

   - *fsck.fat -v fat32.disk*

4. Caso for necessário, crie uma pasta vazia e inclua o sistema de arquivos criado nessa pasta.

   - *sudo mkdir /mnt/disk*
   - *sudo mount -o umask=0 fat32.disk /mnt/disk*

2.4 Demonstração de uso

Ao executar o arquivo principal (**file**) por linha de comando (./file), será imprimido todas as funções possíveis do programa.

```
$ ./file
Para usar: ./file fat32 <opções>
  -i                    Mostra informações do sistema de arquivos.
  -l                    Lista o diretorio raiz.
  -r arquivo [-s sha1]  Recupera o arquivo.
```

Ao executar o arquivo colocando como argumento a imagem de disco e o comando **-i**, será imprimido algumas informações básicas do sistema de arquivos criado.

```
$ ./file fat32.disk -i
Número de FATs = 2
Número de bytes por setor= 512
Número de setores por cluster = 1
Número de setores reservados= 32
```

Também é possível listar todo o conteúdo do diretório raiz do sistema de arquivos, bem como seu tamanho e *cluster* inicial, além da quantidade total de componentes.

```
$ ./file fat32.disk -l
HELLO.TXT (Tamanho = 14, Primeiro cluster = 3)
DIR/ (Tamanho = 0, Primeiro cluster = 4)
MELLO.TXT (Tamanho = 21, Primeiro cluster = 5)
EMPTY (Tamanho = 0, Primeiro cluster = 0)
Número total componentes = 4
```

É importante ressaltar que foi criado alguns arquivos e diretórios no disco. Para isso, foi necessário utilizar o comando *touch* para arquivos ou *mkdir* para diretórios.

A fim de teste, foi excluído o arquivo **EMPTY**, utilizando o comando abaixo, para testar o comando *undelete* do nosso programa.

- *rm /mnt/disk/EMPTY*

```
$ ./file fat32.disk -l
HELLO.TXT (Tamanho = 14, Primeiro cluster = 3)
DIR/ (Tamanho = 0, Primeiro cluster = 4)
MELLO.TXT (Tamanho = 21, Primeiro cluster = 5)
Número total componentes = 3
```

```
$ ./file fat32.disk -r EMPTY
EMPTY: Undelete com sucesso

$ ./file fat32.disk -l
HELLO.TXT (Tamanho = 14, Primeiro cluster = 3)
DIR/ (Tamanho = 0, Primeiro cluster = 4)
MELLO.TXT (Tamanho = 21, Primeiro cluster = 5)
EMPTY (Tamanho = 0, Primeiro cluster = 0)
Número total componentes = 4
```

Caso o nome do arquivo fornecido não existir na tabela de alocação de arquivos, o programa imprimira uma mensagem erro.

```
$ ./file fat32.disk -r XMPTY
XMPTY: arquivo não encontrado
```

Já foi visto que quando um arquivo é excluído, seu primeiro caractere é alterado para **0xE5**. Agora, se houver vários arquivos cujos nomes diferem apenas pelo primeiro caractere, torna-se complicado para o programa recuperar esse arquivo. Um exemplo seria algo como **HELLO.TXT** e **MELLO.TXT**. Nesses casos, a ferramenta retornará um erro de ambiguidade, avisando ao usuário que não foi possível recupera-lo.

```
$ ./file fat32.disk -r HELLO.TXT
HELLO.TXT: Erro de ambiguidade. Múltiplos candidatos encontrados.
```

Para lidar com esse tipo de erro, pode ser fornecido o *SHA-1* do arquivo, que é exclusivo para cada arquivo, dependendo de seu conteúdo. Para obter o *hash SHA-1* do arquivo, utilizamos o comando *sha1sum*.

```
$ sha1sum /mnt/disk/HELLO.TXT
01ba98b3c90126f14577d5b1fdb1ffe9d3364469  /mnt/disk/HELLO.TXT
```

Com o *hash SHA-1* do arquivo, a recuperação dele se torna possível, pois o programa não usara o nome do arquivo, e sim o *hash*, impedindo a ambiguidade demonstrada.

```
$ ./file fat32.disk -l
DIR/ (Tamanho = 0, Primeiro cluster = 4)
EMPTY (Tamanho = 0, Primeiro cluster = 0)
Número total componentes = 2

$ ./file fat32.disk -r HELLO.TXT
-s 01ba98b3c90126f14577d5b1fdb1ffe9d3364469
HELLO.TXT: Undelete com sucesso usando SHA-1
```

```
$ ./file fat32.disk -l
HELLO.TXT (Tamanho = 14, Primeiro cluster = 3)
DIR/ (Tamanho = 0, Primeiro cluster = 4)
EMPTY (Tamanho = 0, Primeiro cluster = 0)
Número total componentes = 3
```

3 Desenvolvimento

O código-fonte do programa foi desenvolvido em 4 arquivos:

- ***Fat32disk****: Funções para o acesso do programa no sistema de arquivos;*
- ***Directory****: Operações de diretório em FAT32;*
- ***Recover****: Analise e recuperação do arquivo desejado;*
- ***File****: O arquivo principal, que comanda as operações do programa.*

Será analisado cada um de seus principais aspectos e funções separadamente, após um breve estudo de algumas estruturas e funções disponíveis na biblioteca do **C**.

3.1 Código fonte

O código-fonte do programa desse trabalho está disponível em um repositório publico do *GitHub*. Seu link de acesso está disponível abaixo:
Repositório do projeto Undelete em FAT32

3.2 Aspectos importantes

Para conseguir manipular o sistema de arquivos, foi necessário a criação de uma estrutura (*struct*), para conter elementos importantes para execução do programa, mais detalhes na figura 4. O mesmo foi feito para os elementos da tabela do diretório raiz, conforma a figura 5. Vale ressaltar que as duas estruturas foram baseadas completamente nas informações respectivas do Documento Técnico de Hardware *"Microsoft Extensible Firmware Initiative FAT32 File System Specification"* em Microsoft (2000).

A estrutura *BootEntry* é uma representação de alguns elementos da estrutura do Setor de inicialização (BS) e dos Bloco de parâmetros do BIOS (BPB). Ela é necessária, pois será nela que existiram algumas informações importantes do sistema de arquivos *FAT* que iremos utilizar para a realização do *undelete*. Dentre os mais de 20 dados da estrutura *BootEntry*, todos estão comentados com sua respectiva função na figura 4.

Já a estrutura *DirEntry* é a representação da estrutura de entrada de diretório raiz de 32 bytes. Ele é imprescindível para o *undelete*, pois será nele que encontraremos o nome do arquivo e seu primeiro *cluster*, sendo possível sua restauração. Na estrutura, contem todas as informações do arquivo no diretório, como já visto na figura 3.

```c
#pragma pack(push,1)  // Struct para o sistema de arquivos
typedef struct BootEntry {
  unsigned char  BS_jmpBoot[3];     // Instruções em assembly para pular para o código de inicialização
  unsigned char  BS_OEMName[8];     // OEM nome em ASCII
  unsigned short BPB_BytsPerSec;    // Bytes por setor. Os valores permitidos incluem 512, 1024, 2048 e 4096
  unsigned char  BPB_SecPerClus;    // Setores por cluster (unidade de dados).
  unsigned short BPB_RsvdSecCnt;    // Tamanho em setores da área reservada
  unsigned char  BPB_NumFATs;       // Número de FATs
  unsigned short BPB_RootEntCnt;    // Número máximo de arquivos no diretório raiz para FAT12 e FAT16.
  unsigned short BPB_TotSec16;      // Valor de 16 bits do número de setores no sistema de arquivos
  unsigned char  BPB_Media;         // Tipo de mídia
  unsigned short BPB_FATSz16;       // Tamanho de 16 bits em setores de cada FAT para FAT12 e FAT16.
  unsigned short BPB_SecPerTrk;     // Setores por faixa do dispositivo de armazenamento
  unsigned short BPB_NumHeads;      // Número de Heads no dispositivo de armazenamento
  unsigned int   BPB_HiddSec;       // Número de setores antes do início da partição
  unsigned int   BPB_TotSec32;      // Valor de 32 bits do número de setores no sistema de arquivos.
  unsigned int   BPB_FATSz32;       // Tamanho de 32 bits em setores de uma FAT
  unsigned short BPB_ExtFlags;      // Uma flag para FAT
  unsigned short BPB_FSVer;         // O número da versão principal e secundária
  unsigned int   BPB_RootClus;      // Cluster onde o diretório raiz pode ser encontrado
  unsigned short BPB_FSInfo;        // Setor onde a estrutura FSINFO pode ser encontrada
  unsigned short BPB_BkBootSec;     // Setor onde a cópia de backup do setor de inicialização está localizada
  unsigned char  BPB_Reserved[12];  // Reservado
  unsigned char  BS_DrvNum;         // Número da unidade BIOS INT13h
  unsigned char  BS_Reserved1;      // Não usado
  unsigned char  BS_BootSig;        // Assinatura de inicialização estendida
  unsigned int   BS_VolID;          // Número de série do volume
  unsigned char  BS_VolLab[11];     // Label de volume em ASCII.
  unsigned char  BS_FilSysType[8];  // Label do tipo de sistema de arquivos em ASCII
} BootEntry;
#pragma pack(pop)
```

Figura 4 – Estrutura para o sistema de arquivos *FAT*

```c
#pragma pack(push,1)  // Struct para os diretorios e arquivos na FAT
typedef struct DirEntry {
  unsigned char  DIR_Name[11];      // Nome do arquivo
  unsigned char  DIR_Attr;          // Atributos do arquivo
  unsigned char  DIR_NTRes;         // Reservado
  unsigned char  DIR_CrtTimeTenth;  // Data de criação (décimos de segundo)
  unsigned short DIR_CrtTime;       // Data de criação (horas, minutos e segundos)
  unsigned short DIR_CrtDate;       // Dia de criação
  unsigned short DIR_LstAccDate;    // Dia de acesso
  unsigned short DIR_FstClusHI;     // 2 bytes mais altos do primeiro endereço do cluster (cluster high)
  unsigned short DIR_WrtTime;       // Data da modificação (horas, minutos e segundos)
  unsigned short DIR_WrtDate;       // Dia da modificação
  unsigned short DIR_FstClusLO;     // 2 bytes mais baixos do primeiro endereço de cluster (cluster low)
  unsigned int   DIR_FileSize;      // Tamanho do arquivo em bytes. (0 para diretórios)
} DirEntry;
#pragma pack(pop)
```

Figura 5 – Estrutura para a tabela de diretório raiz

Essas estruturas são necessárias pois será utilizado uma função para mapear uma região de memoria, com isso, torna-se necessário criar as estruturas da forma mais completa possível, pois a função retorna o endereço no qual o mapeamento foi colocado.

A função que consegue realizar essas ações é o *mmap*. O *mmap* cria um novo mapeamento de memória virtual, podendo realizar o mapeamento de arquivo, onde é necessário um arquivo para poder mapear essa memória, ou o mapeamento anônimo, nesse caso não é necessário um arquivo para realizar o mapeamento. Se dois processos utilizarem o mesmo arquivo para mapear essa memória com o tipo compartilhado, um processo pode enxergar o que o outro está alterando, permitindo assim a troca de mensagem entre eles.

O formato de chamada do *mmap* é da seguinte forma:

- *pa = mmap (addr, len, prot, flags, fildes, off);*

A função *mmap* deve estabelecer um mapeamento entre o espaço de endereçamento do processo em um endereço **pa** para **len** bytes para o objeto de memória representado pelo descritor de arquivo **fildes** em offset **off** para **len** bytes. O valor de **pa** é uma função definida pela implementação do parâmetro **addr** e os valores de **flags**. Uma chamada bem-sucedida deve retornar **pa** como resultado. O intervalo de endereços começando em **pa** e continuando por **len** bytes devem ser legítimos para o espaço de endereço possível (não necessariamente atual) do processo. O intervalo de bytes começando em **off** e continuando para **len** bytes deve ser legítimo para os possíveis deslocamentos (não necessariamente atuais) no objeto de memória representado por **fildes** (The Open Group Base Specifications, 2018).

3.3 Fat32

Como já entendido, nessa parte do código será criada algumas funções para conseguir o acesso de algumas partes importantes do sistema de arquivos *FAT32*.

A função *getFileDirectory* é usada para conseguir o acessar o diretório raiz a partir da imagem de disco.

```
unsigned int getFileDirectory(const char *diskName){
int fd = open(diskName, O_RDWR);
return fd;}
```

Essa função recebe como parâmetro a imagem do disco. Ela tenta abrir o diretório raiz do disco recebido e, se possível, o retorna.

Agora que foi acessado o diretório raiz, pode-se utilizar a função *readFileSystem* para mapear a região da memória, usando a função *mmap*.

```
BootEntry* readFileSystem(int fd){
BootEntry *disk = mmap(NULL, sizeof(BootEntry),
PROT_READ, MAP_PRIVATE, fd, 0);
return disk;}
```

Ela recebe de entrada o diretório raiz de um sistema de arquivos e, com o uso do *mmap*, registra-o na estrutura *BootEntry*, retornando a estrutura.

Ainda com os dados já mapeados na estrutura, pode-se mostrar na tela algumas informações relevantes do sistema de arquivo mapeado, utilizando a função *showDiskInformation* (no caso, seria o argumento **-i** do programa). Também é criado uma mensagem para o usuário demonstrando os usos do programa caso for necessário utilizando a função *showUsage*.

```
void showDiskInformation(BootEntry* disk){
    printf("Número de FATs = %d\n"
           "Número de bytes por setor = %d\n"
           "Número de setores por cluster = %d\n"
           "Número de setores reservados = %d\n",
           disk->BPB_NumFATs, disk->BPB_BytsPerSec,
           disk->BPB_SecPerClus, disk->BPB_RsvdSecCnt
    ); fflush(stdout);}

void showUsage(){
    printf(
    "Para usar: ./file fat32 <opções>\n"
    "  -i     Mostra informações do sistema de arquivos.\n"
    "  -l     Lista o diretorio raiz.\n"
    "  -r arquivo [-s sha1]   Recupera o arquivo.\n"
    );
    fflush(stdout);
    exit(1);}
```

3.4 Diretórios

Diferente do arquivo criado **Fat32disk**, o **Directory** vai realizar operações dentro do diretório raiz (ou atual), que já foi devidamente acessado anteriormente.

A função *getDisk* vai gerar o mapeamento, usando o *mmap*, de um arquivo ou diretório. Ela recebe o diretório raiz e vai retornar o mapeamento de todos os elementos daquele diretório.

```
char* getDisk(unsigned int fd){
struct stat fs;
return mmap(NULL , fs.st_size, PROT_READ, MAP_PRIVATE, fd, 0);}
```

A partir dos diretórios mapeados, a função *getclusterPtr* consegue inseri-los nas estruturas criadas. A função recebe o *mmap* da função anterior, a estrutura do disco e o *cluster* de inicio. São calculados o setor do diretório raiz ou atual e o *offset* do *cluster* do diretório raiz ou atual. Por fim, são adicionados na estrutura *DirEntry* e retornados.

```
DirEntry* getclusterPtr(char* file_content,
BootEntry* disk, unsigned int cluster){

unsigned int rootSector = (disk->BPB_RsvdSecCnt +
disk->BPB_NumFATs*disk->BPB_FATSz32) +
```

```
(cluster- 2)*disk->BPB_SecPerClus;

unsigned int rootClusterOffset=rootSector*disk->BPB_BytsPerSec;
DirEntry* dirEntry=(DirEntry*)(file_content+rootClusterOffset);
return dirEntry; }
```

A função *isDirectory* é uma função verdade que retornara verdadeiro ou falso caso o argumento for um diretório. Ela recebe a estrutura *DirEntry* e, caso seu atributo for **0x10**, será um diretório, retornando verdadeiro.

```
bool isDirectory(DirEntry* dirEntry){
    return dirEntry->DIR_Attr == 0x10;}
```

A função *showRootDirectory* imprime o nome do arquivo ou diretório na tela (colocando uma '/' em diretórios para diferenciar de arquivos). Ela recebe a estrutura *dirEntry* e, caso for um diretório, imprime o seu nome e uma barra no final. Caso contrario, imprime seu nome e extensão, se existir.

Porém a função *getRootDirectoryEntries*, que utiliza as outras funções acima, consegue imprimir o diretório completo na tela (seria o argumento **-l** do programa). É recebido o diretório principal e a estrutura *BootEntry*. Em resumo, a função vai, a partir do diretório raiz, ler um arquivo, se for arquivo não excluído ou um diretório, irá imprimi-lo chamando a função anterior, além de imprimir seu tamanho e seu primeiro *cluster* obtido da estrutura *dirEntry*. Caso encontre o final da *FAT*, ele ira encerrar a função, imprimindo no fim o numero de arquivos ou diretórios encontrados.

```
void showRootDirectory(DirEntry* dirEntry){
    if (isDirectory(dirEntry)){
        int idx = 0;
        while(dirEntry->DIR_Name[idx]!=' '){
            printf("%c",dirEntry->DIR_Name[idx]);
            idx++; }
        printf("/"); }
    else{
        for(int i=0;i<8;i++){
            if(dirEntry->DIR_Name[i]==' ')
                break;
            printf("%c",dirEntry->DIR_Name[i]); }
        if(dirEntry->DIR_Name[8]!=' '){
            printf(".");
            for(int i=8;i<12;i++){
                if(dirEntry->DIR_Name[i]==' ')
```

```
                    break;
                printf("%c",dirEntry->DIR_Name[i]); } } }
}

void getRootDirectoryEntries(int fd, BootEntry* disk){
    unsigned int nEntries = 0;
    unsigned int currCluster = disk->BPB_RootClus;
    unsigned int totalPossibleEntry =
(disk->BPB_SecPerClus * disk->BPB_BytsPerSec)/sizeof(DirEntry);
    unsigned char *file_content = getDisk(fd);
    do{
        DirEntry* dirEntry =
getclusterPtr(file_content,disk,currCluster);
        for(int m=0;m<totalPossibleEntry;m++){
            if (dirEntry->DIR_Attr == 0x00){
                break; }
            if(dirEntry->DIR_Name[0] != 0xe5){
                showRootDirectory(dirEntry);
                int startCluster = dirEntry->DIR_FstClusHI << 16
| dirEntry->DIR_FstClusLO;
                printf(" (Tamanho = %d, Primeiro cluster = %d)\n"
,dirEntry->DIR_FileSize, startCluster);
                nEntries++; }
            dirEntry++; }
        unsigned int *fat = (unsigned int*)
(file_content+disk->BPB_RsvdSecCnt*disk->BPB_BytsPerSec+4*currCluster);
        if(*fat >= 0x0ffffff8 || *fat==0x00){
            break; }
        currCluster=*fat;
    } while(1);
    printf("Número total componentes = %d\n",nEntries);
}
```

3.5 Undelete

O arquivo de código **Undelete** se preocupa somente com a recuperação dos arquivos pelo nome, chamando as outras funções criadas para esse fim.

nOfContiguousCluster é uma função que conta quantos *cluster* determinado arquivos utiliza. Ela recebe ambas as estruturas e, com base em informações retiradas delas, como o tamanho do

arquivo, setores por *cluster* e bytes por setor, conta quantos *clusters* o arquivo utilizava.

```
int nOfContiguousCluster(BootEntry *disk, DirEntry *dirEntry){
int fileSize = dirEntry->DIR_FileSize;
int nBytesPerCluster=disk->BPB_SecPerClus*disk->BPB_BytsPerSec;
int n_Clusters = 0;
if (fileSize % nBytesPerCluster)
n_Clusters = fileSize/nBytesPerCluster + 1;
else n_Clusters = fileSize/nBytesPerCluster;
return n_Clusters; }
```

A função *getfilename* é semelhante a função *showRootDirectory*. Começa recebendo a estrutura *DirEntry*, alocando um espaço na memória para o nome do arquivo usando *malloc*,e apos receber todo o nome, retornando-o.

```
unsigned char* getfilename(DirEntry* dirEntry){
unsigned char *ptrFile = malloc(12 * sizeof(unsigned char *));
unsigned int idx=0;
for(int i=0;i<8;i++){
if(dirEntry->DIR_Name[i]==' ')  break;
ptrFile[idx] = dirEntry->DIR_Name[i]; idx++; }
if(dirEntry->DIR_Name[8]!=' '){
ptrFile[idx] = '.'; idx++; }
for(int i=8;i<12;i++){
if(dirEntry->DIR_Name[i]==' ') break;
ptrFile[idx] = dirEntry->DIR_Name[i]; idx++; }
ptrFile[idx] = '\0';
return ptrFile; }
```

O *unmapDisk* é uma função que consegue reverter o mapeamento da região de memoria utilizada pelo *mmap*, utilizando a função *munmap*, para evitar gasto de memória.

```
void unmapDisk(unsigned char* file_content, int fileSize){
munmap(file_content, fileSize); }
```

Apos recuperar o arquivo, é necessário algumas mudanças tanto no sistema de arquivos quanto na tabela de diretórios. Assim a função *updateFat* vai recolocar os *clusters* usados pelo arquivo na *FAT*, enquanto o *updateRootDir* rescreve o primeiro caractere do arquivo.

A *updateRootDir* recebe o ponteiro da estrutura do arquivo mapeada, a estrutura do disco, o nome do arquivo e o numero de entradas (que deve ser 1). Dado isso, ela encontra o primeiro

cluster através do setor e seu *offset* e consegue alterar o primeiro caractere do nome do arquivo, de **0xE5** para seu nome anterior.

E o *updateFat* recebe o ponteiro da estrutura do arquivo mapeada, a estrutura do disco, o *cluster* atual e o próximo *cluster*. Ele vai atualizar na tabela de alocação de arquivos do *cluster* atual o próximo *cluster*, como visto na figura 2. Ele faz isso para cada *FAT* existente no sistema de arquivos.

```
void updateRootDir(unsigned char* file_content ,
BootEntry* disk, char *filename, int nEntries){
unsigned int rootSector = (disk->BPB_RsvdSecCnt +
disk->BPB_NumFATs * disk->BPB_FATSz32);
unsigned int rootClusterOffset=rootSector*disk->BPB_BytsPerSec;
file_content[rootClusterOffset + nEntries] =
(unsigned char) filename[0];}

void updateFat(unsigned char* file_content , BootEntry* disk,
int currCluster, unsigned int value){
for (int i=0; i<disk->BPB_NumFATs; i++){
unsigned int *fat = (unsigned int*)(file_content +
(disk->BPB_RsvdSecCnt + i * disk->BPB_FATSz32) *
disk->BPB_BytsPerSec  + 4*currCluster);
*fat = value; } }
```

Como vimos anteriormente, as vezes torna-se necessário o uso do *SHA-1* para evitar ambiguidade. Com isso, a função *getShaOfFileContent* o extrai do arquivo deletado e a função *checkSHA* verifica se os dois *hashes* são iguais.

A função *checkSHA* recebe os dois *hashes*, os transforma em strings e os compara com a função *strcmp*, retornando verdade para caso eles forem iguais.

Já o *getShaOfFileContent* recebe as duas estruturas, além do ponteiro do arquivo excluído. A função *SHA-1* precisa de um arquivo e seu tamanho, logo o *getShaOfFileContent* vai armazenar todos os *cluster* do arquivo excluído em um arquivo alocado, além de seu tamanho, para assim, gerar o seu *hash*.

```
bool checkSHA(unsigned char *shaFile, char *shaUser){
char shaToChar[SHA_DIGEST_LENGTH*2];
for(int i = 0; i < SHA_DIGEST_LENGTH; i++)
sprintf(shaToChar+(i*2), "%02x", shaFile[i]);
if(strcmp(shaToChar, shaUser) == 0) return true;
return false; }
```

```
unsigned char* getShaOfFileContent(BootEntry *disk,
DirEntry *dirEntry, unsigned char* file_content){
    int n_Clusters = nOfContiguousCluster(disk, dirEntry);
    int startCluster = dirEntry->DIR_FstClusHI << 16
| dirEntry->DIR_FstClusLO;
    unsigned char *fileData = (unsigned char*)malloc(
dirEntry->DIR_FileSize * sizeof(unsigned char*));
    unsigned char *sha = (unsigned char*)malloc(
SHA_DIGEST_LENGTH * sizeof(unsigned char*));
    int currLen=0;
    if (n_Clusters>1){ for(int i=0; i<n_Clusters;i++){
            unsigned int rootSector = (disk->BPB_RsvdSecCnt
+ disk->BPB_NumFATs*disk->BPB_FATSz32) +
(startCluster- 2)*disk->BPB_SecPerClus;
            unsigned int rootClusterOffset =
rootSector*disk->BPB_BytsPerSec;
            unsigned int bytesInCluster =
disk->BPB_SecPerClus * disk->BPB_BytsPerSec;
            int currReadLen = i<n_Clusters-1 ? bytesInCluster :
dirEntry->DIR_FileSize - i * bytesInCluster;
            for(int i=0;i<currReadLen;i++){
                fileData[currLen] = file_content[rootClusterOffset+i];
                currLen++; }
            startCluster++; } }
    else{ unsigned int rootSector = (disk->BPB_RsvdSecCnt +
disk->BPB_NumFATs*disk->BPB_FATSz32) +
(startCluster- 2)*disk->BPB_SecPerClus;
        unsigned int rootClusterOffset =
rootSector*disk->BPB_BytsPerSec;
        for(unsigned int i=0;i<dirEntry->DIR_FileSize;i++){
            fileData[i] = file_content[rootClusterOffset+i]; } }
    SHA1(fileData, dirEntry->DIR_FileSize, sha);
    return sha; }
```

Enfim, temos a função *getDeletedDirEntry* que vai buscar os arquivos excluídos com o mesmo nome (com exceção do primeiro caractere), usando as outras funções, e tentar recuperalos se os *clusters* usados não tiverem sidos sobrescritos. Ele também verifica a necessidade do uso do *SHA-1*, em caso de ambiguidade.

Resumindo, ela recebe o diretório raiz, a estrutura *BootEntry*, o nome do arquivo excluído e pode ou não receber seu *SHA-1*. Ela cria a região de memória compartilhada do diretório e,

para cada arquivo existente no diretório mapeado, verifica se tem o mesmo nome que o arquivo excluído (sem contar o primeiro caractere), se existirem mais de um, utiliza as funções do *SHA-1* (caso não for passado, retorna o erro de ambiguidade). Após encontrar o arquivo excluído, utilizará as funções desenvolvidas nessa seção para atualizar seu nome (retirar o **0xE5**) e seus *clusters* na Tabela de Alocação de Arquivos. por fim, retorna **1** para sucesso e **-1** pra erro ou fracasso.

```
int getDeletedDirEntry(int fd, BootEntry* disk,
char *filename, char *shaFile){
    unsigned int nEntries=0;
    unsigned int currCluster = disk->BPB_RootClus;
    unsigned int totalPossibleEntry =
(disk->BPB_SecPerClus * disk->BPB_BytsPerSec)/sizeof(DirEntry);
    struct stat fs;
    if(fstat(fd, &fs) == -1)
    { perror("Erro ao ler o stat"); }
    unsigned char* file_content  =
mmap(NULL , fs.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    int fileCount=0;
    int nEntries1=0;
    int currCluster1=0;
    DirEntry* dirEntry1;
    do{
       DirEntry* dirEntry=getclusterPtr(file_content,disk,currCluster);
        for(unsigned int m=0;m<totalPossibleEntry;m++){
            if (dirEntry->DIR_Attr == 0x00){
                break; }
            if(isDirectory(dirEntry)==0&&dirEntry->DIR_Name[0]==0xe5){
                if (dirEntry->DIR_Name[1]==filename[1]){
                    char *recFilename = getfilename(dirEntry);
                    if(strcmp(recFilename+1,filename+1)==0){
                        if (shaFile){
                            bool shaMatched =
checkSHA(getShaOfFileContent(disk, dirEntry, file_content), shaFile);
                            if (shaMatched){
                                fileCount=1;
                                nEntries1=nEntries;
                                dirEntry1=dirEntry;
                                currCluster1=currCluster; } }
                        else{
```

```
                                if (fileCount<1){
                                    nEntries1=nEntries;
                                    dirEntry1=dirEntry;
                                    currCluster1=currCluster;
                                    fileCount++; }
                                else{
                                    printf("%s: Erro de ambiguidade.
Múltiplos candidatos encontrados.\n",filename);
                                    fflush(stdout);
                                    return 1; } } } } }
            dirEntry++;
            nEntries+=sizeof(DirEntry); }
        unsigned int *fat = (unsigned int*)
(file_content+disk->BPB_RsvdSecCnt*disk->BPB_BytsPerSe +4*currCluster);
        if(*fat >= 0x0ffffff8 || *fat==0x00){
            break; }
        currCluster=*fat;
    } while(1);
    if (fileCount==1){
        updateRootDir(file_content, disk, filename, nEntries1);
        int n_Clusters = nOfContiguousCluster(disk, dirEntry1);
        if (n_Clusters>1){
            int startCluster = dirEntry1->DIR_FstClusHI << 16
| dirEntry1->DIR_FstClusLO;
            for(int i=1;i<n_Clusters;i++){
              updateFat(file_content,disk,startCluster,startCluster+1);
                startCluster++; }
            updateFat(file_content , disk, startCluster, 0x0ffffff8);}
        else
            updateFat(file_content , disk, currCluster1+1, 0x0ffffff8);
        unmapDisk(file_content, fs.st_size);
        if (shaFile)
            printf("%s: Undelete com sucesso usando SHA-1\n",filename);
        else
            printf("%s: Undelete com sucesso\n",filename);
        fflush(stdout);
        return 1; }
    return -1; }
```

Por fim, a função *recoverFile* simplesmente utiliza a função *getDeletedDirEntry* para tentar a

recuperação do arquivo. Ela simplesmente facilita o entendimento do código, em que recebe e envia todos os parâmetros para a *getDeletedDirEntry*.

```
void recoverFile(int fd, BootEntry* disk,
char *filename, char *shaFile){
if (getDeletedDirEntry(fd, disk, filename, shaFile) == -1){
 printf("%s: arquivo não encontrado\n",filename);
fflush(stdout); } }
```

3.6 Arquivo principal

O arquivo File é o arquivo principal (*Main*) e é ele que chama os outros arquivos. Nele contem o código que realiza a leitura do comando no console e seleciona qual das funções dos outros arquivos do programa será chamada

```
nt main(int argc, char **argv){
    int flag=0, anyOption=0;
    int information=0, listRootDir=0, recoverFiles=0, sha=0;
    char *recOptarg, *shaOptarg;
    while((flag=getopt(argc, argv, "ilr:s:")) != -1){
        anyOption=1;
        switch(flag){
            case 'i':
                information=1;
                break;
            case 'l':
                listRootDir=1;
                break;
            case 'r':
                recoverFiles=1;
                recOptarg = optarg;
                break;
            case 's':
                sha = 1;
                shaOptarg = optarg;
                break;
            default:
                showUsage();
                break; } }
    if (anyOption==0 || optind==argc)
```

```
        showUsage();
    int fd = getFileDirectory(argv[optind]);
    BootEntry* disk = readFileSystem(fd);
    if (information)
        showDiskInformation(disk);
    else if (listRootDir)
        getRootDirectoryEntries(fd, disk);
    else if(sha){
        if(recoverFiles){
            if (recOptarg == NULL)
            showUsage();
        recoverFile(fd, disk, recOptarg, shaOptarg); }
        else{
            showUsage(); } }
    else if(recoverFiles && !sha){
        if (recOptarg == NULL)
            showUsage();
        recoverFile(fd, disk, recOptarg, NULL); }
}
```

3.7 Observações Importantes

Esse relatório tenta simplificar o entendimento do sistema de arquivos *FAT32* e do programa *Undelete* desenvolvido. Porém caso seja necessário um maior entendimento do código, é indicado que procure o código-fonte do programa no repositório do *GitHub*, pois nele exite alguns comentários, que aqui foram retirados, para facilitar o entendimento de cada linha do programa desenvolvido.

Referências

MICROSOFT, C. *Microsoft Extensible Firmware Initiative FAT32 File System Specification*. [S.l.: s.n.], 2000. v. 1.03. 34 p. Citado na página 11.

MICROSOFT, C. Disks and file systems. Chapter 10, 2014. Citado na página 3.

OffSec Services Limited. *The most advanced Penetration Testing Distribution*. 2022. Disponível em: <https://www.kali.org/>. Citado na página 6.

OpenSSL. *OpenSSL: Cryptography and SSL/TLS Toolki*. 2021. Disponível em: <https://www.openssl.org/>. Citado na página 6.

PJRC: Electronic Projects. *Understanding FAT32 Filesystems*. 2005. Disponível em: <https://www.pjrc.com/tech/8051/ide/fat32.html>. Citado 2 vezes nas páginas 2 e 5.

The Open Group Base Specifications. *mmap - Map pages of memory*. 2018. Disponível em: <https://pubs.opengroup.org/onlinepubs/9699919799/functions/mmap.html>. Citado na página 13.

The OpenSSL Project Authors. *SHA1 - library of OPENSSL*. 2021. Disponível em: <https://www.openssl.org/docs/man3.0/man3/SHA1.html>. Citado na página 6.

Wikipedia. *Design of the FAT file system*. 2021. Disponível em: <https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system>. Citado na página 2.