

Getúlio Coimbra Regis  
Igor Lara de Oliveira

## **Projeto 2 de Sistemas Operacionais Implementação FAT32**

Relatório técnico de atividade prática solicitado pelo professor Rodrigo Campiolo na disciplina de Sistemas Operacionais do curso Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR

Departamento Acadêmico de Computação – DACOM

Bacharelado em Ciência da Computação – BCC

Campo Mourão

Junho / 2022

# Resumo

O trabalho tem como objetivo implementar uma Tabela de Alocação de Arquivos (FAT) em sua versão de 32 bits (FAT32) em um programa em C, utilizando bibliotecas da linguagem como auxílio e obtendo um resultado satisfatório para o grupo.

**Palavras-chave:** Sistema de Arquivo, FAT, FAT32, C.

# Sumário

1	Introdução . . . . .	4
2	Descrição de Atividades . . . . .	4
3	Métodos . . . . .	5
3.1	Organização do Código . . . . .	5
3.2	Decisão de Projeto . . . . .	7
4	Resultados e Discussão . . . . .	7
4.1	Comandos . . . . .	7
4.1.1	Info . . . . .	7
4.1.2	Cluster . . . . .	8
4.1.3	Pwd . . . . .	9
4.1.4	Attr . . . . .	9
4.1.5	Cd . . . . .	9
4.1.6	Touch . . . . .	10
4.1.7	Mkdir . . . . .	10
4.1.8	Rm . . . . .	10
4.1.9	Rmdir . . . . .	11
4.1.10	Rename . . . . .	11
4.1.11	Ls . . . . .	12
4.2	Bugs conhecidos . . . . .	12
4.3	Divisão de tarefas . . . . .	12
5	Conclusão . . . . .	12
6	Referências . . . . .	13

## 1 Introdução

Para o segundo projeto da matéria de Sistemas Operacionais, nosso professor Rodrigo Campiolo solicitou a implementação de estrutura de dados de um sistema de arquivos FAT32.

Neste trabalho apresentaremos como implementamos uma estrutura de dados e um programa na linguagem C para manipular a imagem de um sistema de arquivos de tabela de alocação de arquivos (FAT32) através de operações em um prompt de comandos (shell).

## 2 Descrição de Atividades

Para este projeto foi solicitado um shell que pudesse executar operações, que fossem implementadas utilizando funções e estrutura de dados, com objetivo de manipular a imagem de uma FAT32. Sendo essas as funções requisitadas:

- **info** : Operação que exibe informações da FAT.
- **cluster** : Operação que exibe o conteúdo de um bloco passado por parâmetro.
- **pwd** : Operação que exibe o diretório atual do usuário no shell.
- **attr** : Operação que exibe informações sobre o arquivo/diretório passado por parâmetro.
- **cd** : Operação que altera o diretório atual para o diretório passado por parâmetro.
- **touch** : Operação que cria arquivo com conteúdo vazio com nome passado por parâmetro.
- **mkdir** : Operação que cria pasta vazia com nome passado por parâmetro.
- **rm** : Operação que remove o arquivo de nome passado por parâmetro.
- **rmdir** : Operação que remove a pasta de nome passado por parâmetro.
- **cp** : Operação que copia arquivo de um diretório de origem para o diretório de destino, ambos com caminhos passadas por parâmetros, podendo também passar arquivos da partição atual para imagem e vice versa.
- **mv** : Operação que move arquivo de um diretório de origem para o diretório de destino, ambos com caminhos passados por parâmetros, podendo também passar arquivos da partição atual para imagem e vice versa.
- **rename** : Operação que renomeia arquivo/diretório com passagem de parâmetros do nome novo e nome antigo.
- **ls** : Operação que exibe uma lista dos diretórios e arquivos existentes no diretório atual.

Porém não conseguimos implementar as operações *cp* e *mv*.

## 3 Métodos

De forma a começar a fazer o projeto, precisamos pegar materiais de referência, dentre eles utilizamos a documentação feita pela Microsoft ([MICROSOFT... , 2000](#)) sobre a FAT32, e também os slides de aulas sobre FAT ([FRANKEL, \)](#) da universidade Harvard.

### 3.1 Organização do Código

Dentro da implementação da FAT utilizamos as structs definidas no documento da Microsoft ([MICROSOFT... , 2000](#)) e colocamos em dois arquivos principais, *fat32.c* e *fat32.h*, onde o *fat32.h* serve para referenciar a biblioteca.

Também criamos um arquivo *main.c* que implementava o shell que executava as operações dependendo do comando passado pelo mesmo.

A seguir algumas das estruturas que nós implementamos.

---

```
struct FSInfo {
    uint32_t FSI_LeadSig;
    uint8_t FSI_Reserved1[480];
    uint32_t FSI_StrucSig;
    uint32_t FSI_Free_Count;
    uint32_t FSI_Nxt_Free;
    uint8_t FSI_Reserved2[12];
    uint32_t FSI_TrailSig;
}__attribute__((packed));
```

---

**Código 1** : Struct de FSINFO da documentação

---

```
struct ShortDirEntry {
    char DIR_Name[8];
    char DIR_Extension[3];
    uint8_t DIR_Attr;
    uint8_t DIR_NTRes;
    uint8_t DIR_CrtTimeTenth;
    uint16_t DIR_CrtTime;
    uint16_t DIR_CrtDate;
    uint16_t DIR_LstAccDate;
    uint16_t DIR_FstClusHI;
    uint16_t DIR_WrtTime;
    uint16_t DIR_WrtDate;
    uint16_t DIR_FstClusLO;
    uint32_t DIR_FileSize;
}__attribute__((packed));
```

---

**Código 2** : Struct de Short Dir Entry da documentação

Para cada operação solicitada fizemos ao menos uma função dentro do código no arquivo *fat32.c*, além dessas funções também criamos funções adicionais para o facilitar o desenvolvimento da função principal. Também tivemos que fazer algumas vezes, funções recursivas, como por exemplo, para alocar espaço na FAT, ou imprimir o caminho do diretório atual, pois dessa forma parecia mais natural o desenvolvimento.

---

```

void info();
void read_dir();
void ls();
void cluster(int i);
void cd(char* folder);
void pwd();
void attr(char* entry_name);
void rename_dir_entry(char* entry_name, char* new_name);
void rm(char* entry_name);
void touch(char* file_name);
void mkdir(char* entry_name);
void rmdir(char* entry_name);

void create_formated_name(char* name, char* unformatted_name);
void print_name(char* name);

```

---

Figura 1 – Funções criadas para as operações

Para navegar nos repositórios criamos uma estrutura de dados *directory\_t* que funciona como uma pilha, ao navegar para um diretório mais profundo este era adicionado ao topo da pilha junto com suas características: entradas no diretório, quantidade de entradas, *cluster* que se encontra, o nome do diretório e um ponteiro que aponta para o diretório passado.

---

```

typedef struct directory {
    DirEntry* entries;
    uint32_t quantity;
    struct directory* previous;
    char name[260];
    uint32_t cluster;
} directory_t;

extern directory_t* directory_stack;
extern uint32_t directory_stack_count;

```

---

Figura 2 – Estrutura de dados da pilha de diretório

A variável *directory\_stack* mantém o ultimo elemento da pilha e a variável *directory\_stack\_count* a altura.

## 3.2 Decisão de Projeto

Na especificação do projeto que foi passada pelo nosso professor, existiam algumas simplificações permitidas para a implementação, dentre elas a não necessidade de tratar vários diretórios em uma operação (ex : `cd img/pasta1/pasta2/pasta3`), logo escolhemos usar essa simplificação para evitar erros.

Outra decisão importante de citar, é a não implementação de entradas de nome de *long entry*, apenas a de *short entry*, apesar de não implementados eles são lidos, mas não são tratados, apenas tratamos das *short entry*.

Também decidimos que os comandos iam sempre converter os parâmetros para letras maiúsculas, uma vez que é algo necessário para o funcionamento da *short entry* da FAT.

## 4 Resultados e Discussão

Nossos resultados com o que conseguimos testar estão satisfatórios, testamos todas as funções que fizemos e todas funcionaram bem para nossos testes, e após manusear a imagem no nosso prompt de comando e montar a imagem, a imagem continuava funcionando sem estar corrompida.

Tentamos implementar os comandos *cp* e *mv*, mas como sempre estava acontecendo bugs decidimos não implementá-los.

### 4.1 Comandos

Como comentado anteriormente implementamos todos os comandos exceto os comandos *mv* e *cp*, nesta seção iremos mostrar o funcionamento dos comandos.

#### 4.1.1 Info

Exibe informações do sistema de arquivo, sendo que a maioria dos seus dados são obtidos através do *bootsector*, sendo que alguns desses são obtidos através de cálculos como é o caso dos endereços da FAT e do início da área de dados, nenhum desses dados estão explícitos no setor inicial.

A sintaxe desse comando é: *info*.

```
fatshell:[img/] $ info
FAT Filesystem information

OEM name: mkdosfs
Total sectors: 102400
Jump: 0xEB5890
Sector size: 512
Sectors per cluster: 1
Reserved sectors: 32
Number of fats: 2
Root dir entries: 0
Media: 0xF8
Sectors by FAT: 788
Sectors per track: 32
Number of heads: 64
Hidden sectors: 0
Drive number: 0x00
Current head: 0x00
Boot signature: 0x29
Volume ID: 0x60D18F6B
Volume label:
Filesystem type: FAT32
BS Signature: 0xAA55
FAT1 start address: 0x00000000000004000
FAT2 start address: 0x00000000000006800
Data start address: 0x000000000000C9000
```

Figura 3 – Resultado do comando info. Fonte: Autoria própria.

### 4.1.2 Cluster

Exibe informações do *cluster* em formato hexadecimal e em texto, como usado em vários editores e visualizadores de dados hexadecimais.

A sintaxe desse comando é: *cluster* <número do cluster>.

[illegible]

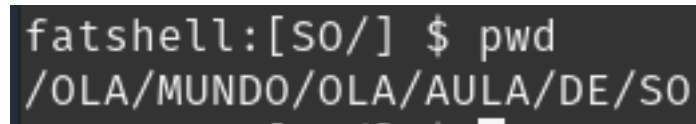
Figura 4 – Resultado do comando cluster. Fonte: Autoria própria.



#### 4.1.3 Pwd

Imprime na tela o caminho atual, para implementar utilizamos a nossa pilha de diretórios, primeiros chegamos na base e depois voltamos ao topo da pilha imprimindo o nome do diretório.

A sintaxe desse comando é: *pwd*.



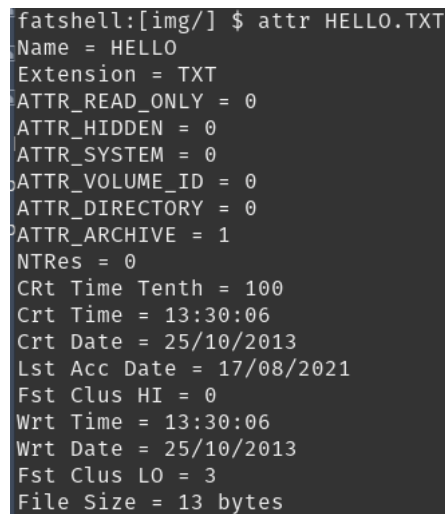
```
fatshell:[SO/] $ pwd
/OLA/MUNDO/OLA/AULA/DE/SO
```

Figura 5 – Resultado do comando pwd. Fonte: Autoria própria

#### 4.1.4 Attr

Imprime na informações sobre uma entrada no diretório, como por exemplo os atributos, nome, tamanho, cluster, datas.

A sintaxe desse comando é: *attr <arquivo ou diretório>*.



```
fatshell:[img/] $ attr HELLO.TXT
Name = HELLO
Extension = TXT
ATTR_READ_ONLY = 0
ATTR_HIDDEN = 0
ATTR_SYSTEM = 0
ATTR_VOLUME_ID = 0
ATTR_DIRECTORY = 0
ATTR_ARCHIVE = 1
NTRes = 0
Crt Time Tenth = 100
Crt Time = 13:30:06
Crt Date = 25/10/2013
Lst Acc Date = 17/08/2021
Fst Clus HI = 0
Wrt Time = 13:30:06
Wrt Date = 25/10/2013
Fst Clus LO = 3
File Size = 13 bytes
```

Figura 6 – Resultado do comando attr. Fonte: Autoria própria

#### 4.1.5 Cd

Utilizado para navegar pelos diretórios, ao entrar em um, é empilhado ou desempilhado a pilha de diretórios.

A sintaxe desse comando é: *cd <nome do diretório>*.



```
fatshell:[img/] $ cd files
fatshell:[FILES/] $
```

Figura 7 – Resultado do comando cd. Fonte: Autoria própria

#### 4.1.6 Touch

Utilizado para criar uma nova entrada no diretório, também é alocado um *cluster* para essa nova entrada, vai ter tamanho 0 a entrada.

A sintaxe desse comando é: *touch <nome do novo arquivo>*.

```
fatshell:[img/] $ touch projeto.so
fatshell:[img/] $ ls
CREATEDATE CRT_TIME UPDATEDATE UPD_TIME LSTACCDATE SIZE NAME
25/10/2013 13:30:06 25/10/2013 13:30:06 17/08/2021 13 - HELLO.TXT
25/10/2013 13:30:46 25/10/2013 13:30:46 25/10/2013 0 d FILES
17/08/2021 15:00:10 17/08/2021 15:00:10 17/08/2021 1680 - ABC.TXT
30/06/2022 04:03:04 30/06/2022 04:03:04 30/06/2022 0 d OLA
30/06/2022 04:12:24 30/06/2022 04:12:24 30/06/2022 0 - PROJETO.SO
```

Figura 8 – Resultado do comando touch. Fonte: Autoria própria

#### 4.1.7 Mkdir

Funciona como o comando *touch*, mas além de criar uma entrada de diretório é criado duas entradas uma entrada *ponto*, que referencia o proprio diretório, e outra entrada *ponto ponto*, que referencia o diretório pai, dentro do diretório novo.

A sintaxe desse comando é: *mkdir <nome do novo diretório>*.

```
fatshell:[img/] $ mkdir so
fatshell:[img/] $ cd so
fatshell:[so/] $ ls
CREATEDATE CRT_TIME UPDATEDATE UPD_TIME LSTACCDATE SIZE NAME
30/06/2022 04:16:04 30/06/2022 04:16:04 30/06/2022 0 d .
30/06/2022 04:16:04 30/06/2022 04:16:04 30/06/2022 0 d ..
```

Figura 9 – Resultado do comando mkdir. Fonte: Autoria própria

#### 4.1.8 Rm

Utilizado para remover uma entrada no diretório, para isso marcamos o primeiro *byte* da entrada com o valor em hexadecimal 0xE5, que significa que é um arquivo removido, que pode ser sobre escrito, além de também liberarmos os *clusters* alocados pelo arquivo.

A sintaxe desse comando é: *rm <nome do arquivo>*.

```
fatshell:[so/] $ ls
CREATEDATE CRT_TIME UPDATEDATE UPD_TIME LSTACCDATE SIZE NAME
30/06/2022 04:16:04 30/06/2022 04:16:04 30/06/2022 0 d .
30/06/2022 04:16:04 30/06/2022 04:16:04 30/06/2022 0 d ..
30/06/2022 04:20:08 30/06/2022 04:20:08 30/06/2022 0 - PROJETO.02
fatshell:[so/] $ rm projeto.02
fatshell:[so/] $ ls
CREATEDATE CRT_TIME UPDATEDATE UPD_TIME LSTACCDATE SIZE NAME
30/06/2022 04:16:04 30/06/2022 04:16:04 30/06/2022 0 d .
30/06/2022 04:16:04 30/06/2022 04:16:04 30/06/2022 0 d ..
```

Figura 10 – Resultado do comando rm. Fonte: Autoria própria

#### 4.1.9 Rmdir

Funciona como o comando *rm*, só que para diretórios que se encontram vazios, ou seja, só com as duas entradas: *ponto* e *ponto ponto*.

A sintaxe desse comando é: *rmdir <nome do diretório>*.

```

CREATEDATE CRT_TIME UPDATEDATE UPD_TIME LSTACCDATE SIZE      NAME
25/10/2013 13:30:06 25/10/2013 13:30:06 17/08/2021 13      - HELLO.TXT
25/10/2013 13:30:46 25/10/2013 13:30:46 25/10/2013 0        d FILES
17/08/2021 15:00:10 17/08/2021 15:00:10 17/08/2021 1680     - ABC.TXT
30/06/2022 04:03:04 30/06/2022 04:03:04 30/06/2022 0        d OLA
30/06/2022 04:15:46 30/06/2022 04:15:46 30/06/2022 0        d TESTE
30/06/2022 04:16:04 30/06/2022 04:16:04 30/06/2022 0        d SO
fatshell:[img/] $ rmdir so
fatshell:[img/] $ ls
CREATEDATE CRT_TIME UPDATEDATE UPD_TIME LSTACCDATE SIZE      NAME
25/10/2013 13:30:06 25/10/2013 13:30:06 17/08/2021 13      - HELLO.TXT
25/10/2013 13:30:46 25/10/2013 13:30:46 25/10/2013 0        d FILES
17/08/2021 15:00:10 17/08/2021 15:00:10 17/08/2021 1680     - ABC.TXT
30/06/2022 04:03:04 30/06/2022 04:03:04 30/06/2022 0        d OLA
30/06/2022 04:15:46 30/06/2022 04:15:46 30/06/2022 0        d TESTE

```

Figura 11 – Resultado do comando *rmdir*. Fonte: Autoria própria

#### 4.1.10 Rename

O comando *rename* troca os 11 primeiros *bytes* da entrada do diretório para o nome novo do diretório/arquivo, além de tratar também nomes com ponto e caracteres não aceitos nos *short dir entry*, é formatado o nome para que a extensão fique nos três últimos *bytes*, sendo separado o nome da extensão por espaços, código hexadecimal *0x20*, e troca os caracteres não aceitos por "\_".

A sintaxe desse comando é: *rename <nome atual> <nome novo>*.

```

fatshell:[img/] $ ls
CREATEDATE CRT_TIME UPDATEDATE UPD_TIME LSTACCDATE SIZE      NAME
25/10/2013 13:30:06 25/10/2013 13:30:06 17/08/2021 13      - HELLO.TXT
25/10/2013 13:30:46 25/10/2013 13:30:46 25/10/2013 0        d FILES
17/08/2021 15:00:10 17/08/2021 15:00:10 17/08/2021 1680     - ABC.TXT
30/06/2022 04:03:04 30/06/2022 04:03:04 30/06/2022 0        d OLA
30/06/2022 04:15:46 30/06/2022 04:15:46 30/06/2022 0        d TESTE
fatshell:[img/] $ rename OLA oi
fatshell:[img/] $ ls
CREATEDATE CRT_TIME UPDATEDATE UPD_TIME LSTACCDATE SIZE      NAME
25/10/2013 13:30:06 25/10/2013 13:30:06 17/08/2021 13      - HELLO.TXT
25/10/2013 13:30:46 25/10/2013 13:30:46 25/10/2013 0        d FILES
17/08/2021 15:00:10 17/08/2021 15:00:10 17/08/2021 1680     - ABC.TXT
30/06/2022 04:03:04 30/06/2022 04:30:56 30/06/2022 0        d OI
30/06/2022 04:15:46 30/06/2022 04:15:46 30/06/2022 0        d TESTE

```

Figura 12 – Resultado do comando *rename*. Fonte: Autoria própria

#### 4.1.11 Ls

Ao abrir um diretório com o comando *cd*, todas as entradas desse são lidas, o comando *ls* lista elas de uma forma legível, mostrando a data e tempo de criação, data e tempo de atualização, data do último acesso, tamanho, um "-" caso seja um arquivo ou "d" caso seja um diretório e por fim o nome dessa entrada no diretório.

A sintaxe desse comando é: *ls*.

```
fatshell:[img/] $ ls
CREATEDATE CRT_TIME UPDATEDATE UPD_TIME LSTACCDATE SIZE NAME
25/10/2013 13:30:06 25/10/2013 13:30:06 17/08/2021 13 - HELLO.TXT
25/10/2013 13:30:46 25/10/2013 13:30:46 25/10/2013 0 d FILES
17/08/2021 15:00:10 17/08/2021 15:00:10 17/08/2021 1680 - ABC.TXT
30/06/2022 04:03:04 30/06/2022 04:30:56 30/06/2022 0 d OI
30/06/2022 04:15:46 30/06/2022 04:15:46 30/06/2022 0 d TESTE
```

Figura 13 – Resultado do comando *ls*. Fonte: Autoria própria

## 4.2 Bugs conhecidos

Em nossos testes não encontramos nenhum bug para relatar, porém acreditamos que possamos ter esquecidos de tratar casos de que exista mais de um setor por *cluster* o que pode acabar atrapalhando no funcionamento do nosso *shell*.

## 4.3 Divisão de tarefas

Nossas divisões de tarefas não foi segregada, nós decidimos entrar em chamadas de áudio no aplicativo Discord e compartilhávamos o código que fazíamos e testávamos através da extensão Live Share do Visual Studio Code.

Apesar de termos feito a maior parte das coisas juntas, temos duas coisas mais notáveis que separamos o trabalho: o prompt de comando foi desenvolvido pelo Igor e a leitura dos dados do setor de boot da FAT foi feita pelo Getúlio.

## 5 Conclusão

Nossa conclusão é que após implementarmos essa estrutura de dados conseguimos entender melhor como a tabela de alocação de arquivos funciona, em sua maior parte, como os arquivos são referenciados dentro dela e como eles estão separados.

## 6 Referências

FRANKEL, J. L. *FAT32 File Structure (slides)*. Harvard. [S.l.]. Disponível em: <https://cscie92.dce.harvard.edu/spring2021/slides/FAT32%20File%20Structure.pdf>. Citado na página 5.

MICROSOFT. Microsoft Extensible Firmware Initiative FAT32 File System Specification. [S.l.], 2000. Disponível em: <https://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc>. Citado na página 5.