Universidade Federal de
São João del Rei
Campus Tancredo de Almeida Neves
Departamento Ciência da Computação

FAT-16

Aluno: Gabriel Carneiro
Aluno: Felipe Samuel
Aluno: André Luiz
Professor: Rafael Sachetto

28 Novembro
2020

Conteúdo

1 Introdução

O desafio proposto neste trabalho, foi implementar um simulador de sistema de arquivos simples, fundamentado em trabela de alocação de 16 bits (FAT), anexado um shell para efetuar as operações deste sistema de arquivos. A implementação se baseia no modelo:

- Partição virtual.
  - 512 bytes por setor.
  - Cluster de 1024 bytes (2 setores por cluster).
  - 4096 clusters.
- O sistema de arquivos é preenchido dos seguintes setores
- Boot Block (1 cluster = 1024 bytes preenchido com 0xBBBB)
- FAT (4096 entradas de 16 bits)
- Root dir (1 cluster = 1024 bytes e 32 entradas de diretório)
- Demais entradas(4086 clusters)

1.1 Como compilar e executar o programa

Para compilar e executar o programa basta usar estes comandos num terminal

```
Para compilar:
$ make

Para executar:
$ ./fat_shell

Para ler o manual:
$ man ./fat_shell.1
```

2 Funcionalidades

O programa basicamente simula um sistema de arquivos, criando ou carregando primeiramente um arquivo binário, e usando operações em um shell criado especificamente para a FAT. A execução dos comandos fica por conta do shell

2.1 Shell

2.1.1 Comandos

O shell recebe os comandos e controla a execução das funções, como dito anteriormente, estão disponíveis 13 comandos:

Comando	Descrição
init	inicializa o sistema de arquivos em branco
load	carrega um sistema de arquivos do disco
ls [path]	lista diretório
mkdir [path]	cria um diretório
create [path]	cria um arquivo
unlink [path]	exclui arquivo ou diretório
write "string"[path]	sobrescreve dados em um arquivo
append "string"[path]	anexa dados em um arquivo
read [path]	le conteúdo de um arquivo
clear	limpa a tela
cd [path]	vai ao diretório especificado
help	imprime uma tabela com os comandos
quit	sai do programa

- **init**: Inicializa o sistema de arquivos em branco, semelhante a formatação de um disco, ele escreve no arquivo de texto e carrega a tabela FAT e o diretório raiz para a memoria.
- **load**: Carrega a tabela FAT e o diretório raiz para a memoria, caso não seja feita essa operação, a tabela fica desatualizada no disco e os valores se tornam então, irrelevantes.

- **ls**: Lista as entradas de diretório do diretório atual ou do diretório especificado como argumento do comando, diretórios são representados com a cor azul e os arquivos são representados com a cor branca, pode ser omitido o caminho para o diretório, nesse caso ele usa o diretório atual como caminho
- **mkdir**: Cria um novo diretório dentro do diretório especificado no comando, não pode haver espaço no nome do diretório.
- **create**: Cria um novo arquivo de texto dentro do diretório especificado no comando, não pode haver espaço no nome do arquivo.
- **unlink**: Deleta um arquivo ou diretório, apagando sua referência na tabela FAT, mas deixado os dados intactos no cluster pois quando se perde a referência para o cluster, é o mesmo que caso ele não existisse.
- **write**: Escreve (sobrescreve) uma String em um arquivo de texto, cada cluster de dados pode guardar até 1024 caracteres, os demais caracteres são armazenados em outros clusters.
- **append**: Concatena uma String em um arquivo de texto, cada cluster de dados pode guardar até 1024 caracteres, os demais caracteres são armazenados em outros clusters.
- **read**: Lê um arquivo de texto e imprime seu conteúdo na tela
- **clear**: Limpa a terminal.
- **cd**: Define o diretório atual, para que quando seja omitido a o no começo de um caminho e use o diretório atual como inicio de pesquisa.
- **help**: Lista uma tabela com todos os comandos e uma breve descrição do funcionamento.
- **quit**: Encerra o shell e libera a memória alocada.

3 Estruturas e funções

3.1 Estruturas de dados

3.1.1 dir_entry_t

```
typedef struct {
    uint8_t filename[18];
    uint8_t attributes;
    uint8_t reserved[7];
    uint16_t first_block;
    uint32_t size;
} dir_entry_t;
```

Representa uma entrada de diretório, podendo ser um diretório ou um arquivo, o qual ocupa 32 bytes. Cada diretório é um vetor com 32 entradas, então um diretório ocupa 1024 bytes = um cluster.

3.1.2 data_cluster

```
union data_cluster{
    dir_entry_t dir[CLUSTER_SIZE / sizeof(dir_entry_t)];
    uint8_t data[CLUSTER_SIZE];
};
```

Um data cluster pode armazenar tanto um diretório quanto um cluster de dados, 32 entradas de 32 bytes para cada diretório, totalizando 1024 bytes, um cluster de dados também ocupa 1024 bytes.

3.2 Macros e variáveis globais

3.2.1 Macros de tamanho

Representa o tamanho total de alguma estrutura de dados

```
Tamanho de um cluster de dados
#define CLUSTER_SIZE    1024

Tamanho máximo de um comando
#define CMD_SIZE        4096
```

3.2.2 Macros de retorno da função de navegação de diretório

A função dir_nav() retorna essas macros para sinalizar o resultado da busca

```
retornado quando se encontra no diretório raiz
#define ROOT_DIR      0

retornado quando um diretório no caminho não foi encontrado
#define DIR_NOT_FOUND 1

retornado quando o diretório procurado foi encontrado
#define DIR_EXIST     2

retornado quando o diretório requisitado encontra-se cheio
#define DIR_FULL      3

retornado quando quando o diretório pai requisitado existe
#define DIR_READY     4

retornado quando é encontrado um arquivo
#define NOT_A_DIR     5

retornado quando é encontrado um diretório
#define NOT_A_FILE    6
```

3.2.3 Pedidos de navegação de arquivo

Usado para informar o navegador de arquivos se necessita que retorne o diretório pai ou o diretório requisitado

```
Quando se necessita que retorne o próprio arquivo
#define WANT_CLUSTER  0

Quando se necessita que retorne o diretório pai
#define WANT_PARENT   1
```

3.2.4 Variáveis globais

```
Define se o sistema de arquivos foi inicializado
int fs_loaded = 0;

Define o diretório atual para que seja possível omitir parte do caminho
char cur_dir[256] = "/";

Representa o símbolo que aparece antes da leitura do comando
const char *PS1 = VERDE(USER_SYMBLE);

Um vetor com os comandos disponíveis
const char *comandos_disponiveis[] = {
"init",
"load",
"ls",
"mkdir",
"create",
"unlink",
"write",
"append",
"read",
"clear",
"cd",
"help",
"quit",
};
```

3.3 Funções

3.3.1 int init()

Inicializa o sistema de arquivos, o primeiro cluster é o boot block, nele escrevemos 0xBBBB em todas as posições desse cluster, em seguida inicializamos a tabela FAT, com os 10 primeiros blocos reservados para o boot block (1 entrada), a propria tabela FAT (8 blocos) e o diretório raiz (1 bloco), após isso é escrito o restantes dos 4086 blocos com o valor 0 e carrega a tabela FAT e o diretório raiz para a memória principal.

3.3.2 int load()

Lê o sistema de arquivos e carrega a tabela FAT e o diretório raiz para a memória principal.

3.3.3 int ls(char *dir)

Recebe uma string com o caminho para o diretório, com cada diretório separado por / (exemplo: /home/rafael) e imprime na tela o nome dos diretórios e arquivos, sendo que diretórios são imprimidos com a cor azul e arquivos com a cor branca.

3.3.4 int mkdir(char *dir)

Cria um novo diretório no caminho especificado, primeiramente ele chama a função de navegação de diretórios com o argumento WANT_PARENT e caso o caminho exista, ele checa se existe um diretório ou um arquivo com o nome requisitado, caso não haja, ele cria o diretório, atualiza a tabela FAT e grava o arquivo do disco.

3.3.5 int create(char *dir)

Cria um novo arquivo no caminho especificado, primeiramente ele chama a função de navegação de diretórios com argumento de WANT_PARENT e caso o caminho exista, ele checa se existe um diretório ou um arquivo com o nome requisitado, caso não haja, ele cria o arquivo, atualiza a tabela FAT e grava o arquivo do disco.

3.3.6 int unlink(char *dir)

Deleta o diretório ou arquivo especificado no caminho, para isso ele chama a função de navegação de diretório com o argumento WANT_CLUSTER e

caso o arquivo ou diretório exista, ele passa para a próxima etapa da exclusão. Caso seja um arquivo, ele apaga a referência para o arquivo e deleta suas entradas na tabela FAT, mas não zera os clusters de dados pois em uma eventual sobrescrita, os dados que já estão lá serão irrelevantes. Caso seja um diretório, primeiro será feita uma verificação para saber se o diretório está vazio, e no caso de estar vazio ele é excluído.

3.3.7 int write(char *string, char *dir)

Escreve uma string em um arquivo, primeiramente chama a função de navegação de diretório para saber se o arquivo realmente existe, caso existe é retornado o endereço do arquivo na memória, em seguida todos os clusters extras são removidos pois não são mais necessários já que é uma sobrescrita. Em seguida é chamada a função break_str_into_clusters(), que recebe uma string e retorna um vetor com as strings quebradas em tamanhos de 1024 bytes cada, e em seguida escreve esses clusters de dados no disco.

3.3.8 int append(char *string, char *dir)

Concatena uma string em um arquivo, primeiramente chama a função de navegação de diretórios para saber se o arquivo realmente existe, caso exista, é retornado o endereço para ele. Em seguida procuramos o ultimo cluster que contém o fim da string para podermos concatenar. Em seguida ele concatena a string no ultimo cluster de forma que ele ocupe o cluster inteiro. A função break_str_into_clusters() é chamada e retorna um vetor de data_clusters, que são então gravados no disco.

3.3.9 int read(char *dir)

Lê um arquivo no disco e imprime na tela, para isso ele primeiramente chama a função de navegação de diretórios para saber se o arquivo realmente existe, caso exista, é retornado o endereço do arquivo e é feita a leitura de cada cluster e imprime eles na tela.

3.3.10 data_cluster read_data_cluster(unsigned index)

Procura um cluster de dados na memória a partir do endereço especificado como parâmetro e retorna.

3.3.11 void write_data_cluster(unsigned index, data_cluster cluster)

Grava um cluster de dados no disco a partir do endereço especificado no parâmetro da função.

3.3.12 int dir_nav(char **dir_list, int dir_num, int *index, int want)

Função de navegação no sistema de arquivos, essa função funciona de forma que o usuário define o caminho que ela deve seguir e caso algum problema ocorra, ela retorna as macros definidas na seção 3.2.4 para a função chamadora defina o que fazer. O parâmetro want define se será retornado o diretório pai o o arquivo/diretório do fim do caminho, want = 0 retorna o fim do caminho, e 1 retorna o diretório pai.

3.3.13 int break_dir(char *dir, char ***dir_list)

Recebe uma string com um caminho e retorna um vetor de strings contendo em cada posição, o nome dos diretórios no caminho especificado.

3.3.14 int save_fat()

Grava a tabela FAT no disco.

3.3.15 int break_str_into_clusters(char *string, data_cluster**buffer)

Recebe uma string com o texto e divide essa string em clusters de 1024 bytes, e retorna um vetor desses clusters para que possam ser gravados em disco.

3.3.16 int cd(char *dir)

Define o diretório atual para que possa ser omitido o caminho completo no outros comandos, ela basicamente define o caminho atual (variável global cur_dir) pelo parâmetro dir substituindo-o, sendo possível navegar mais livremente pelo sistema de arquivos, sem que seja necessário escrever todo o caminho quando for chamar as funções.

4 Execução do programa

Criamos um manual do nosso sistemas de arquivo, podendo ser acessado com o seguinte comando no terminal:

```
$ man ./fat_shell.1
```

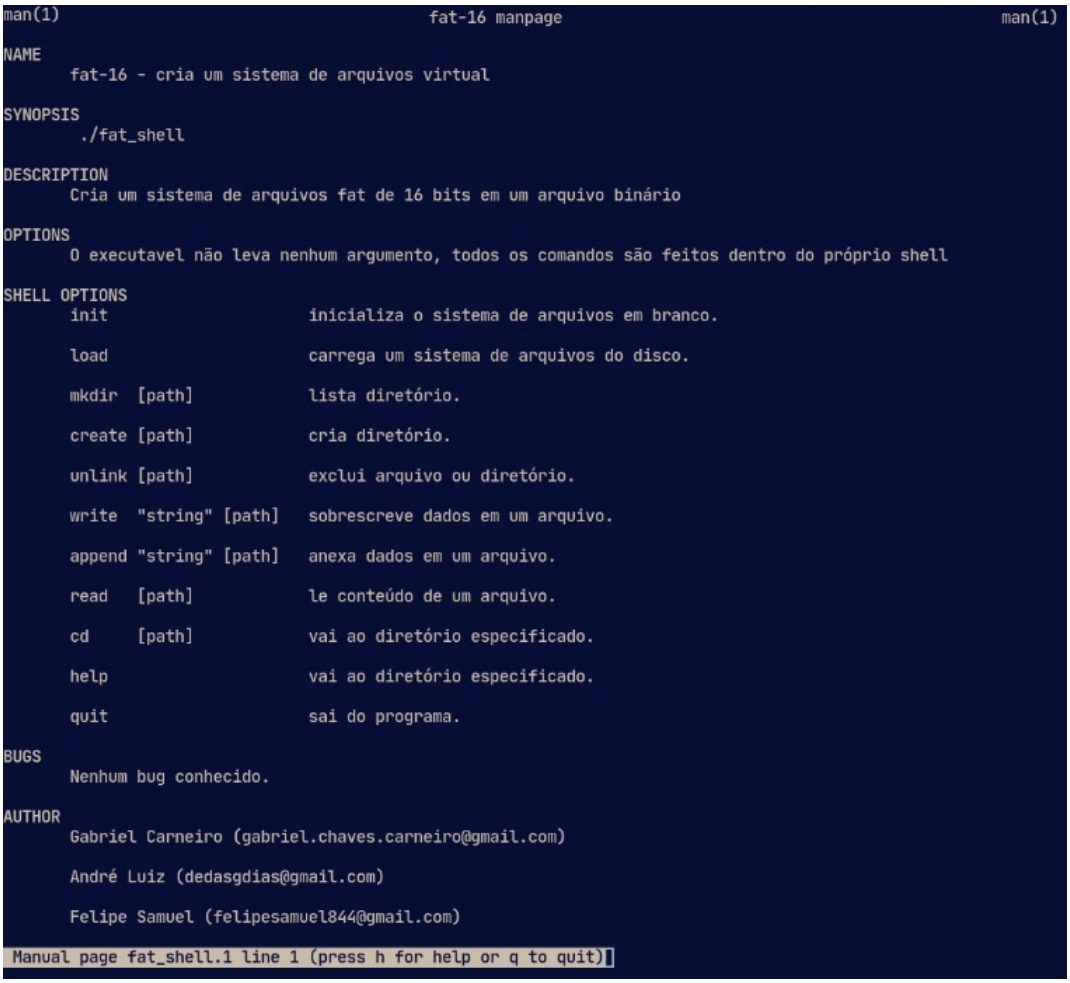

Figura 1: Entrada de manual

Podemos observar a seguir a funcionalidade de todos os comandos, agindo corretamente com suas devidas condições. A seguir podemos observar a implementação do comando CD, navegando entre diretórios.

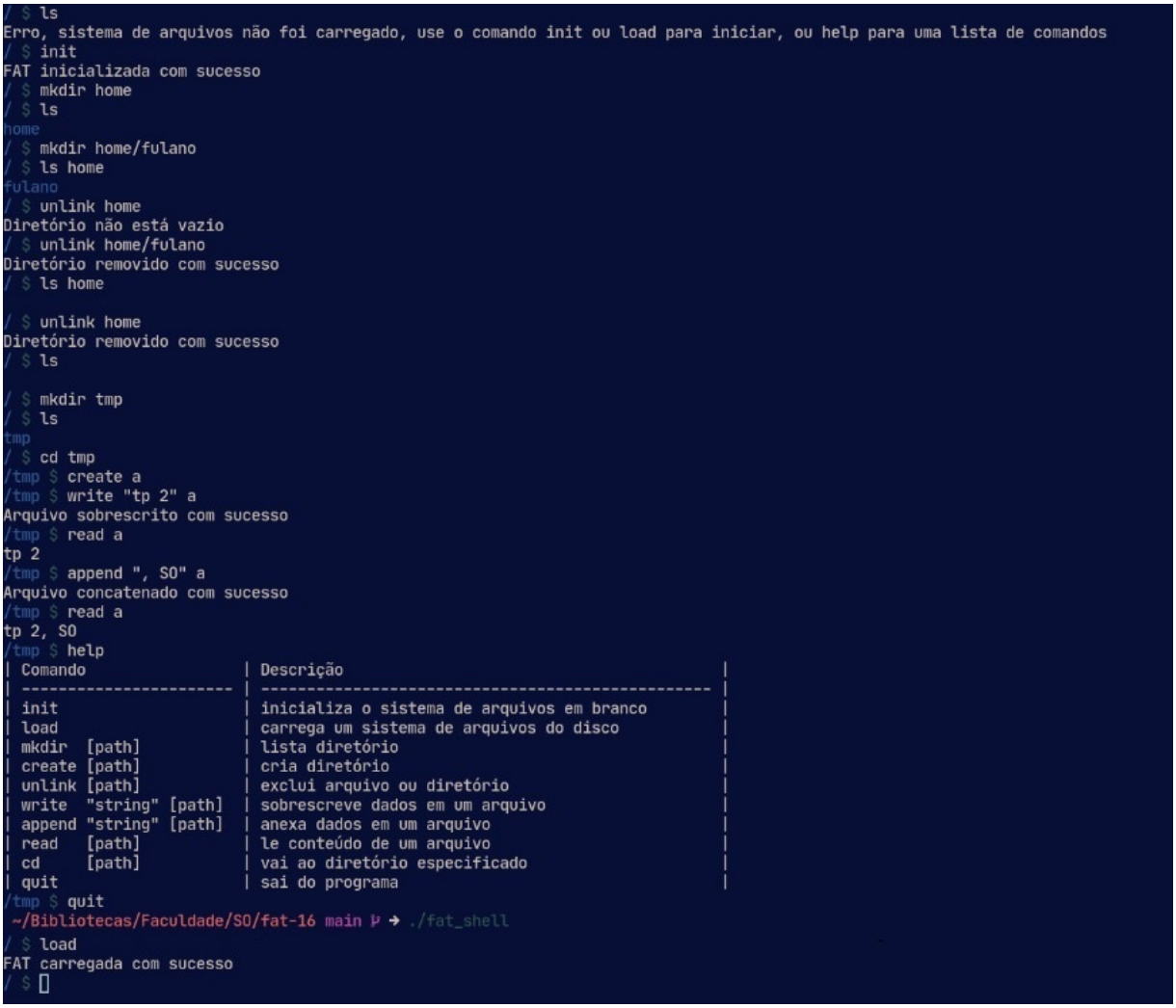

Figura 2: Analise e teste de comandos

5 Conclusão

O objetivo deste trabalho foi desenvolver um algoritmo simulador de sistema de arquivos simples. No decorrer do trabalho a implementação não foi o maior desafio, o obstáculo encontrado foi entender e abstrair informações necessárias para resolução do problema. No decorrer do desenvolvimento conseguimos atingir o objetivo e portanto este trabalho abre uma uma oportunidade de simular e compreender os métodos e sistema de arquivos a partir de um arquivo binário.

6 Bibliografia

Todo material utilizado foi obtido através do manual do LINUX, nenhum outro material foi consultado para o desenvolvimento do trabalho.