Breno Farias
Felipe Archanjo da Cunha Mendes
Pamella Lissa Sato Tamura

Projeto 02: Implementação de estruturas e operações para manipulação de sistemas de arquivos FAT32

Relatório técnico de atividade prática solicitado pelo professor Rodrigo Campiolo na disciplina de Sistemas Operacionais do 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

Sumário

1 Introdução

O relatório trata da implementação de um FAT32, o qual está dividido descrição da atividade, métodos, resultados e discussão, bugs conhecidos, divisão das atividades, conclusões e referências. Além disso, foi utilizado materiais fornecidos durante a disciplina, assim como as definições de comandos e conceitos que foram obtidos por pesquisas sequenciais por cada membro da equipe (também referenciadas ao final do trabalho).

O FAT - ***File Allocation Table*** ou, na tradução direita, Tabela de Alocação de Arquivos - é um sistema de gerenciamento de arquivos desenvolvido pela *Microsoft* tendo várias versões, o FAT, FAT12, FAT16, FAT32 e o exFAT. O nome foi atribuído pelo conceito de que ele consiste em mapear, por meio de tabelas, onde estão os dados de cada arquivo.

Assim, o uso dessa estrutura torna-se óbvia pois o armazenamento é dividido em bloco, ou seja, dentro de cada bloco podem existir um ou mais arquivos, além de que um arquivo pode ocupar vários blocos todavia, não obrigatoriamente, de forma sequencial. Cada bloco consiste em 512 *bytes* ou também denominados de ***cluster***, sendo assim, se houver um arquivo de tamanho, por exemplo, 800 *bytes*, ele irá consumir 2 *clusters* de 512 *bytes* cada, o que implica em fragmentação interna, contudo o tamanho de um *cluster* pode variar. A fragmentação interna corresponde ao cenário em que um bloco não é usado por completo e o espaço não utilizado é perdido, pois não pode ser aproveitado por outro arquivo, logo o uso de blocos menores permite diminuir o percentual de armazenamento perdido devido a fatores como a fragmentação.

As principais diferenças entre o FAT16 e o FAT32, estão relacionadas a quantidade de *bits* utilizadas para endereçamento de dados, ou seja, o FAT16 utiliza 16 *bits* para endereçar dados e o FAT32 aprimora essa quantidade é emprega 32 *bits*. Dessa forma, com 16 *bits* é possível endereçar até 65536 *clusters* e com 32 *bits* tem-se mais de 4 bilhões de *clusters* que podem ser endereçados. Por fim, é relevante mencionar que os sistemas de arquivos FAT estão ultrapassados, visto que a *Microsoft* apresentou o NTFS ou ***New Technology File System***.

2 Objetivos

2.1 Objetivo Geral

O objetivo desse trabalho consiste em, primeiramente, aprimorar os conhecimentos estudados em sala de aula, ou seja, compreender a estrutura do FAT32 para, por fim, implementar um sistema de gerenciamento de arquivos FAT32.

2.2 Objetivos específicos

- Listar as operações do sistema de gerenciamento de arquivos FAT32 que manipule um arquivo no formato .img por meio de um prompt
- Analisar a utilidade e função de cada operação
- Implementar as respectivas operações listadas
- Apresentar o resultado e/ou saída das operações implementadas

3 Fundamentação

Ao implementar o FAT32, é indispensável o conhecimento de comandos que, cada qual com suas funções, retornam um valor de saída. As operações implementadas, executadas e testadas são: `info`, `cluster <num>`, `pwd`, `attr <file/dir>`, `cd <path>`, `touch <file>`, `mkdir <dir>`, `rm <file>`, `rmdir <dir>`, `rename <file> <newfilename>` e `ls`. Diante disso, o algoritmo desenvolvido foi fundamentada e baseada por documentações apresentadas e referenciadas ao final do documento além das aulas ministradas durante esse período de estudos.

4 Materiais

- VirtualBox 6.1.34 com Ubuntu 20.04 LTS (ISO)
- Windows 11 (64 bits)
- Intel Pentium Gold G5400
- 8GB de RAM
- Python 3.8.2
- Visual Studio Code 1.68
- 010 Editor
- AccessData FTK Imager 3.1.2.0

5 Descrição da Atividade

Implementar um sistema de gerenciamento de arquivos FAT32 que manipule um arquivo no formato .img por meio*prompt* (*shell*) com as seguintes operações:

- `info`: exibe informações do disco e da FAT.
- `cluster <num>`: exibe o conteúdo do bloco num no formato texto.
- `pwd`: exibe o diretório corrente (caminho absoluto).
- `attr <file / dir>`: exibe os atributos de um arquivo (file) ou diretório (dir).
- `cd <path>`: altera o diretório corrente para o definido como path.
- `touch <file>`: cria o arquivo file com conteúdo vazio.
- `mkdir <dir>`: cria o diretório dir vazio.
- `rm <file>`: remove o arquivo file do sistema.
- `rmdir <dir>`: remove o diretório dir, se estiver vazio.
- `rename <file> <newfilename>`: renomeia arquivo file para newfilename.
- `ls`: listar os arquivos e diretórios do diretório corrente.

6 Métodos

Para a criação desse projeto foi utilizado a linguagem de programação Python que, com a modularização, responsável pela organização da estrutura do projeto e do paradigma de programação orientada a objetos, possibilitando a criação de classes e métodos específicos, foi possível criar estruturas que auxiliassem no gerenciamento do FAT32.

6.1 Estrutura do Projeto

A implementação do projeto esta estruturada de forma organizada para melhor compreensão das classes e de seus métodos. O diretório **imagem** contém a imagem do sistema FAT32 a ser manipulada pelo usuário. Já o diretório **src**, contém todas as classes (organizadas em arquivos distintos) utilizadas para representar as instâncias de um sistema de arquivos FAT32. Por fim, o arquivo **main.py** contém a função principal do programa.

6.2 Principais Classes

Para a melhor representação das estruturas do FAT32 foi necessário a utilização de classes. Dentre elas, tais quais FatShell, Fat32, BootSector, Fat32Table e RootDirectory.

6.2.1 Classe Fatshell

A classe Fatshell representa o Shell, é responsável por instanciar a classe Fat32, sendo ela a principal classe do sistema utilizado para a interação entre o usuário e sistema de gerenciamento de arquivos. Nele o usuário solicitará as principais operações para a manipulação de arquivos, tais como *info*, *cluster*, *pwd*, *attr*, *cd*, *touch*, *mkdir*, *rm*, *rmdir*, *rename* e *ls*, alem de outras auxiliares, tais como *help*, *quit* e *clear*.

Em seu método *run* a classe irá verificar se o comando escrito pelo usuário existe para poder tratar ela de forma adequada, executando os métodos da classe Fat32, instanciada por ela.

6.2.2 Classe BootSector

Esta classe representa o *BootSector* do sistema de arquivos. Em sua inicialização ela lê todo o setor do *BootSector* da imagem e armazena seus valores nos atributos da classe. Com isso, ao acessar essa classe é possivel ter acesso a todos esses valores.

6.2.3 Classe Fat32Table

Essa classe representa as tabelas FAT's do sistema. Nela é armazenada como principal atributo sua posição no arquivo imagem.

6.2.4 Classe RootDirectory

Essa classe representa o *RootDirectory* do sistema de arquivos. Nela são armazenados como principais atributos:

root_pos - Posição do *RootDirectory*

root_path - O caminho padrão do diretório Root (/)

current_pos - A posição atual na árvore de diretórios

current_path - O caminho atual na árvore de diretórios

dir_hierarchy - Representando uma lista com os diretórios anteriores do atual

6.2.5 Classe Fat32

Esta classe representa todo o sistema de gerenciamento de arquivos, tendo como principais atributos:

filename: recebe o caminho do arquivo no formato .img para eventual leitura ou manipulação.

boot_sector: instancia uma classe chamada BootSector, com as informações do sistema, para representar o Boot Sector do sistema FAT32.

pos_fat0: Posição da primeira tabela fat, sendo obtida a partir do seguinte cálculo: ReservedSectors * BytesPerSector

fat32_table_0: Instancia uma classe chamada Fat32Table para representar a primeira tabela Fat32 do sistema.

pos_fat1: Posição da segunda tabela fat, sendo obtida a partir do seguinte cálculo: pos_fat0 + SectorsPerFAT32 * BytesPerSector

fat32_table_1: Instancia uma classe chamada Fat32Table para representar a segunda tabela Fat32 do sistema.

pos_root_dir: Posição do RootDirectory, sendo obtido a partir do seguinte cálculo: pos_fat1 + SectorsPerFAT32 * BytesPerSector

root_directory: Instancia de uma classe chamada RootDirectory para representar o RootDirectory do sistema.

Com a utilização dos dados recebidos pelos atributos foi possível criar as principais funções do sistema para a manipulação de arquivos.

Info: A função `info` apenas exibe no terminal as informações coletadas do sistema pela classe *BootSector* em seu processo de leitura no arquivo *myimagefat32* e armazenada em seus atributos. O *BootSector* é o primeiro bloco do disco do FAT. Dentro dele está armazenadas informações como o nome da FAT, o número de *bytes* por setor, número de setores por *cluster*, *RootCluster* que informa o *cluster* do diretório *root*, entre outros.

Cluster: Para exibir o conteúdo do bloco através de seu valor do *cluster* foi necessário abrir o arquivo para leitura binária (rb) e atualizar o ponteiro para o *cluster* buscado pelo usuário a partir da seguinte expressão: `pos\_root\_dir - RootCluster * BytesPerSector +` Nesse sentido, é lido todo o setor e exibido seu conteúdo no terminal.

Pwd: A função `pwd()` exibe no terminal o atributo *current_path* da classe RootDirectory, ou seja, a string que armazena o caminho atual na árvore de diretórios. Conforme o *current_path* for se alterando durante o processo de manipulação de arquivos, quando o usuário digitar `$ pwd`, será exibido o caminho atual atualizado na árvore de diretórios.

Attr: A função attr() utiliza de uma função auxiliar *SearchFile* para encontrar a posição de determinado arquivo pedido pelo usuário. Diante disso o ponteiro no arquivo aponta para essa determinada posição e começa seu processo de leitura da imagem para exibir ao usuário as informações de determinado arquivo, respeitando o tamanho, em

bytes, de cada atributo.

Cd: A função *cd()* é dividida em 3 casos:

1. Se for solicitado `$ cd .` o comando será ignorado, fazendo com que o usuário permaneça no diretório atual.

2. Caso for solicitado `$ cd ..` e o caminho atual for o root (/), o comando será ignorado. Caso contrário será retirado o último diretório da lista *dir_hierarchy*, o que implica na atualização da posição do diretório corrente (*current_pos*) para o diretório anterior ao removido e remoção do nome do diretório da string *current_path*.

3. Se for solicitado a operação de `$ cd` para qualquer outro argumento (nome do diretório), será verificado a existência de algum diretório com esse nome a partir do diretório atual e, em caso positivo, esse caminho é adicionado na lista de hierarquia de diretórios (*dir_hierarchy*), o que implica na atualização do caminho corrente (*dir_hierarchy*) e atualização da posição do diretório corrente (*current_pos*).

Touch: Inicialmente é encontrado uma posição vazia onde o arquivo poderá ser criado, por meio da função *find_pos_empty_entry*, para que o arquivo possa ser criado sem que ocorra corrupção da imagem. O processo de escrita é relativamente simples pois, uma vez atualizado o ponteiro do arquivo para a posição correta, basta escrever as informações uma seguida da outra, respeitando a quantidade de bytes que cada campo exige.

Mkdir: O processo de criação do comando *mkdir* é relativamente parecido com o *touch*, uma vez que, de início, deve ser calculado o endereço de uma posição vazia para a criação do diretório. À partir disso, é necessário preencher os atributos do diretório, como seu nome e extensão. Não só isso mas foi necessário criar o subdiretório dentro da região data com dois outros diretórios: o . (aponta para o diretório atual) e o .. (aponta para o diretório anterior).

Rm: Para criar a lógica de funcionamento do comando *rm* foi necessário apontar para o endereço onde o arquivo se encontrava e apagar seu nome e extensão. Além disso, em seu atributo foi escrito 0xE5 para que, caso algum processo de leitura (como o *ls*) for ler, constatar que ali havia um arquivo que foi apagado. Ademais, foi necessário escrever, em ambas as tabelas FAT32, que os *clusters* referente ao arquivo deletado estão livres. Por fim, os dados foram apagados da região data a partir do valor dos *clusters* utilizados por esse arquivo.

Rmdir: O processo de criação do comando relativamente parecido com o rm, porém com algumas singelas diferenças. Foi necessário criar um sistema de verificação para constatar se o diretório estava de fato vazio ou não. Por fim, o resto do processo foi

idêntico ao do rm, sendo necessário apagar o nome do diretório e escrever 0xE5 em seu atributo.

Rename: Inicialmente, é necessário verificar se o arquivo solicitado para renomeação existe e se realmente é um arquivo. Em caso positivo, o ponteiro interno no arquivo será atualizado diretamente para a posição onde é armazenado o nome do arquivo para poder atualizar seu valor para o solicitado pelo usuário.

Ls: Para a realização da listagem de arquivos do diretório atual, foi necessário realizar uma varredura dos arquivos a partir da posição do diretório atual *current_pos*, o que implica na exibição o nome dos arquivos até não encontrar mais nenhum dado.

Help, Quit e Clear: Para melhor representação de um shell real foram criados os comandos `$ Help`, `$ Quit e Clear` que lista os comandos que podem ser utilizados para a manipulação de arquivos, saída do sistema e limpeza do terminal, respectivamente.

7 Discussão e Resultados

À princípio, foi decidido que seriam desenvolvidas as operações menos complexas e, estruturalmente, mais simples. Para que, no decorrer do desenvolvimento e progresso das aplicações fossem produzidas as demais operações gradualmente de acordo com as respetivas complexidades.

Inicialmente, quando a ideia do projeto ainda não estava totalmente clara e formada, foram encontrados alguns obstáculos no desenvolvimento do comando `cluster <num>` para exibir o conteúdo de um bloco *num* em formato texto. Mas, com a ajuda e atendimento - PAluno remoto - do professor Rodrigo Campiolo o problema foi esclarecido e norteado.

As aplicações mais complexas foram dos comandos `cd <path>` e `touch <file>`. Em resumo, houve a dificuldade no entendimento de como e onde os novos dados referente a um novo arquivo seriam escritos em relação ao `touch <file>` sem que houvesse a possibilidade de corromper a imagem. Já no `cd <path>`, o empecilho encontrado foi relacionado ao gerenciamento das mudanças dos diretórios. Considerando que, nesse último, a posição e o nome do(s) diretório(s) do caminho atual deveriam ser armazenados.

Apesar de ter sido desenvolvido grande parte dos comandos de manipulação de arquivos para simular o sistema de arquivos FAT32, não foi possível implementar as operações *cp* e *mv*. No entanto, todas as operações até então desenvolvidas no projeto foram executadas sem erros de compilação.

Após as divisões das atividades definidas na seção 8 serem realizadas, para cada operação foi obtido um *output*. Dessa forma, podemos observar na Figura 1 a saída do primeiro comando, `make run`, utilizado para executar a *main.py* do programa.

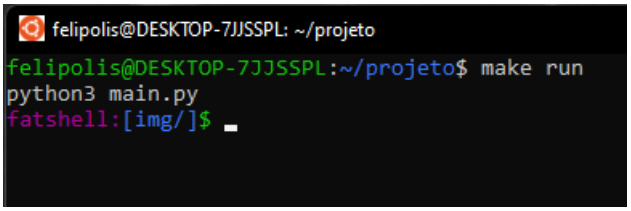

Figura 1 – Execução do comando make run

A próxima operação a ser executada é o comando `info` do qual exibe, como já explicitado anteriormente na subseção 6.1, as informações do primeiro bloco do disco do FAT, *BootSector*. O resultado dessa ação pode ser observado na Figura 2.

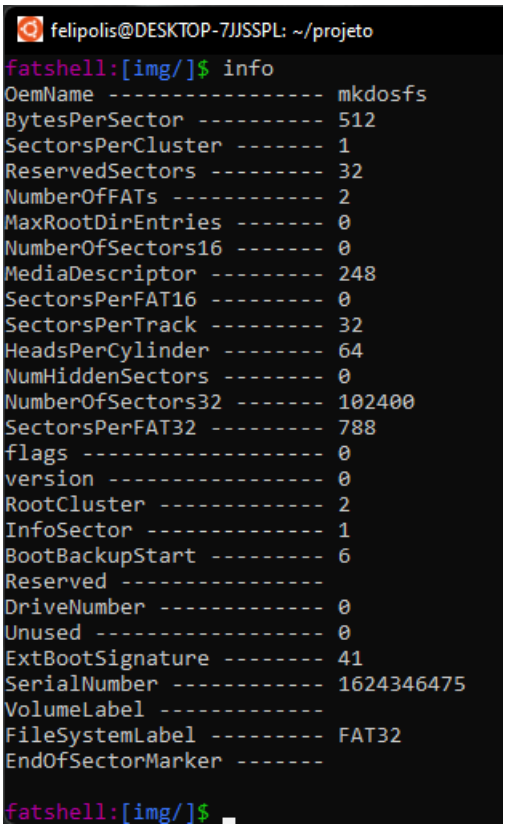

Figura 2 – Execução do comando info

O comando conseguinte executado `cluster <num>`, também apresentado na subseção da Estrutura do Projeto gera uma saída exibindo o conteúdo do bloco do *cluster*, pode ser analisada pela Figura 3 tendo como exemplo o *cluster* 3.

Figura 3 – Execução do comando cluster <num>

Já na Figura 4, referencia o resultado da operação `ls`. O comando que, assim como os anteriores e os posteriores, foi descrito na subseção 6.1.

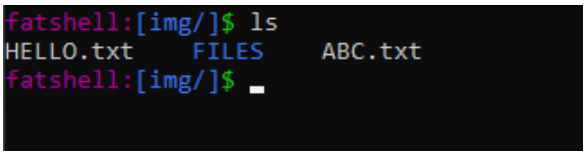

Figura 4 – Execução do comando pwd

Para mostrar os atributos de um arquivo (*file*) ou diretório (*dir*), efetua-se o comando `attr <file / dir>`. Assim, o resultado é notado na Figura 5. Nesse caso, optamos por exibir os atributos de um arquivo (*HELLO.txt*).

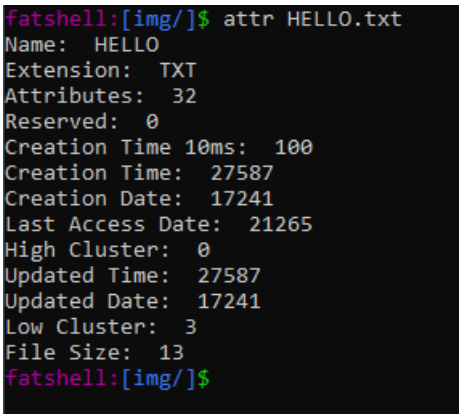

Figura 5 – Execução do comando attr <file | dir>

Quando `cd <path>` é operado, altera o diretório atual para o definido caminho definido como *path*. Após ele ser efetuado, caso desejável, o comando `ls` é executado para exibir os arquivos e diretórios do novo diretório corrente - ilustrada na Figura 6 - e o usuário confirmar a mudança ocorrida.

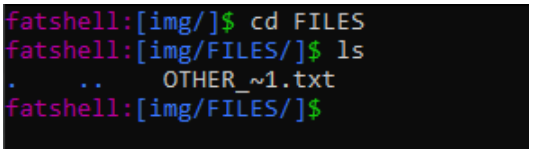

Figura 6 – Execução do comando `cd <path>`

A Figura 7 refere-se ao comando `touch <file>`, ou seja, a operação cria um arquivo *file* com conteúdo vazio. Pode-se observar que antes dele ser executado, com a operação `ls`, apenas os arquivos *HELLO.txt*, *ABC.txt* e o diretório *FILES* existem. Em seguida, após alguns `touch <file>` consecutivos serem executados, os arquivos *snake.py, main.c, fat16.c e fat16.h* são criados podendo, também, ser verificado com `ls`.

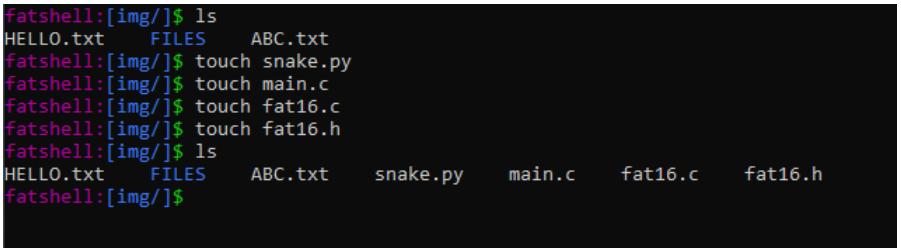

Figura 7 – Execução do comando `touch <file>`

De forma semelhante porém antagônica à ação anterior, para a remoção de um arquivo `rm <file>` deve ser executado. Assim, com `ls` confirmando, visualmente, o que existe antes e depois da operação `rm <file>` é chamado. Nesse caso, os arquivos *ABC.txt* e *HELLO.txt* são removidos podendo, assim, ser observado na Figura 8.

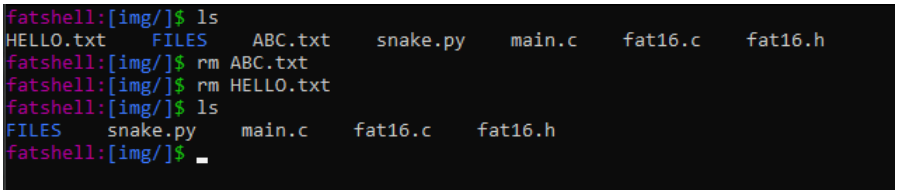

Figura 8 – Execução do comando `rm <file>`

Ao falar sobre criação de um diretório *dir* vazio, estamos referenciando o comando `mkdir <dir>`. Para a demostração dessa ação, o diretório "filmes"é criado (confirmado logo em seguida pelo `ls`. Ainda dentro desse diretório recém criado, também é possível criar outros diretórios. Assim, nota-se que diretórios vazios com nomes terror, aventura, romance e fantasia foram criados. E, não podendo faltar com a confirmação das operações com `ls` onde é evidente, juntamente aos comandos anteriores, na Figura 9.

```
fatshell:[img/]$ ls
FILES    snake.py    main.c    fat16.c    fat16.h
fatshell:[img/]$ mkdir filmes
fatshell:[img/]$ ls
FILES    snake.py    main.c    fat16.c    fat16.h    filmes
fatshell:[img/]$ cd filmes
fatshell:[img/filmes/]$ mkdir terror
fatshell:[img/filmes/]$ mkdir aventura
fatshell:[img/filmes/]$ mkdir romance
fatshell:[img/filmes/]$ mkdir fantasia
fatshell:[img/filmes/]$ ls
.    ..    terror    aventura    romance    fantasia
fatshell:[img/filmes/]$ cd terror
fatshell:[img/filmes/terror/]$ ls
.    ..
fatshell:[img/filmes/terror/]$
```

Figura 9 – Execução do comando `mkdir <dir>`

O comando `pwd` exibe o caminho absoluto do diretório atual. Assim, a Figura 10 mostra a saída desse comando, de forma exemplar, como */filmes/terror/* indicando o caminho até o diretório corrente em que o usuário se encontra.

```
fatshell:[img/filmes/terror/]$ pwd
/filmes/terror/
fatshell:[img/filmes/terror/]$
```

Figura 10 – Execução do comando `pwd`

De forma contrária à função do `mkdir <dir>`, o `rmdir <dir>` remove um diretório indicado por parâmetro com a condição de não ter conteúdo em seu interior, ou seja, o diretório só é removido se caso estiver vazia caso contrário, não. Assim, a fim de exemplo, os diretórios criados anteriormente com `mkdir <dir>` - terror, aventura, romance e fantasia - são retirados com a operação `rmdir <dir>`. Esse procedimento é ilustrado na Figura 11.

```
fatshell:[img/filmes/]$ ls
.    ..    terror    aventura    romance    fantasia
fatshell:[img/filmes/]$ rmdir terror
fatshell:[img/filmes/]$ rmdir aventura
fatshell:[img/filmes/]$ rmdir romance
fatshell:[img/filmes/]$ rmdir fantasia
fatshell:[img/filmes/]$ ls
.    ..
fatshell:[img/filmes/]$ cd ..
fatshell:[img/]$ ls
FILES    snake.py    main.c    fat16.c    fat16.h    filmes
fatshell:[img/]$ rmdir filmes
fatshell:[img/]$ ls
FILES    snake.py    main.c    fat16.c    fat16.h
fatshell:[img/]$
```

Figura 11 – Execução do comando `rmdir <dir>`

O último comando requerido implementado é `rename <file> <newfilename>` do qual possui o papel de renomear um arquivo *file* para *newfilename*, como já explicitado

na subseção 6.1. A fim de exemplo, o arquivo *snake.py* é renomeado para *cobrinha.js* e, em seguida, *cobrinha.js* para *game.cpp*. E ainda, para cada mudança faz-se chamada da operação `ls` para melhor visualização do ocorrido o que pode ser verificado na Figura 12.

```
fatshell:[img/]$ ls
FILES    snake.py    main.c    fat16.c    fat16.h
fatshell:[img/]$ rename snake.py cobrinha.js
fatshell:[img/]$ ls
FILES    cobrinha.js    main.c    fat16.c    fat16.h
fatshell:[img/]$ rename cobrinha.js game.cpp
fatshell:[img/]$ ls
FILES    game.cpp    main.c    fat16.c    fat16.h
fatshell:[img/]$
```

Figura 12 – Execução do comando `rename <file> <newfilename>`

Por fim, com o *software AccessData FTK Imager* é montada a imagem .img, sendo possível visualizar os arquivos e diretórios criados e modificados a partir dos comandos anteriores. O último `ls` na Figura 12 nos informa que há, no diretório *root*, o diretório *FILES* e os arquivos *game.cpp, main.c, fat16.c e fat16.h*. Da mesma forma, ao se analisar o resultado da montagem da imagem na Figura 13, percebe-se que ela contém todos os arquivos e diretórios informados pelo terminal.

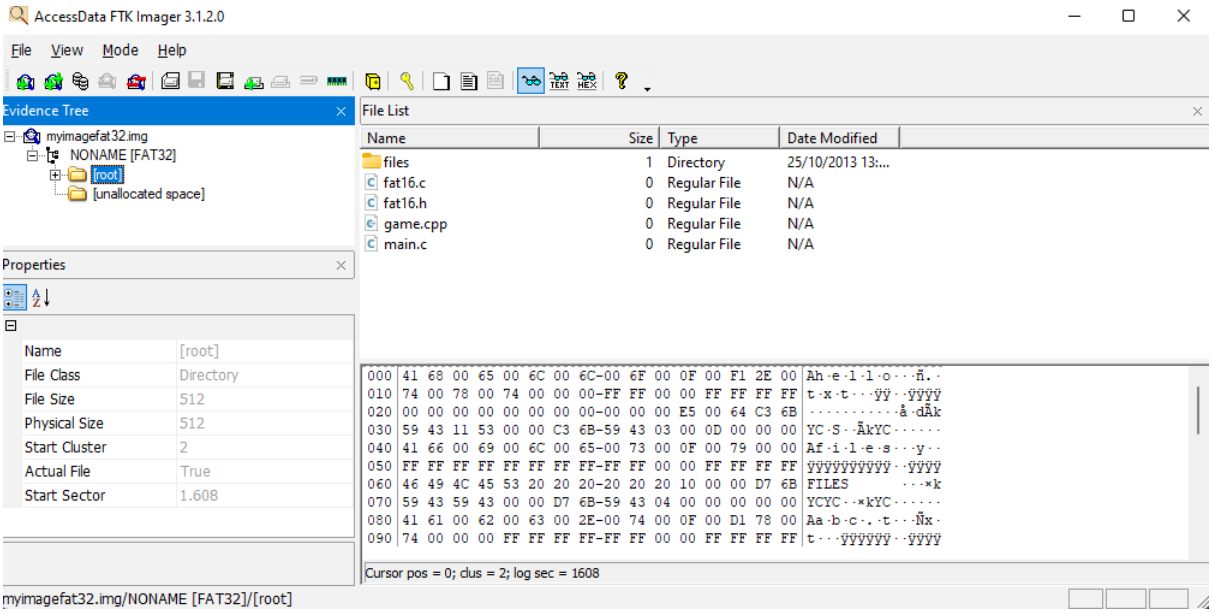

Figura 13 – Resultado final da imagem .img montada

8 Divisão das Atividades

Vale ressaltar que em todo o processo de desenvolvimento do projeto todos os membros da equipe se mostraram dispostos à contribuir um com outro, tirando dúvidas ou realizando pesquisas pontuais refente ao contexto. Portanto, apesar de cada um ficar

responsável por desenvolver uma parte do projeto, todos tiveram participação do trabalho por inteiro.

Felipe: Foi responsável pelo desenvolvimento do *FatShell*, da estrutura das classes, e das seguintes operações:

- `cd <path>`
- `cluster <num>`
- `touch <file>`
- `mkdir <dir>`

Breno: Foi responsável pelo desenvolvimento das seguintes operações:

- `ls`
- `rm <file>`
- `attr  <file / dir>`
- `rename e <file> <newfilename>`

Pamella: Foi responsável pela estruturação inicial do relatório, além do desenvolvimento das seguintes operações:

- `pwd`
- `info`
- `help`
- `quit`
- `rmdir <dir>`

9 Conclusões

Em vista dos procedimentos realizados, portanto, foi desenvolvido um sistema de gerenciamento de arquivo *FAT32* com um *shell* que permite ao usuário realizar a manipulação de arquivos sobre um arquivo no formato .img. Conclui-se, então, que todos os processos até então implementados foram atendidos por inteiro apesar de dois comandos não estarem elaborados, tanto `mv <source\_path> <target\_path>` quanto a operação `cp <source\_path> <target\_path>`).

10 Referências

[1]OS Dev.org. FAT. Disponível em https://wiki.osdev.org/FAT. Acessado em 07/10/2021.

[2] Frankel, James L. FAT32 File Structure (slides). Harvard. Disponível em https://cscie92.dce.harvard.edu/spring2021/slides/FAT32%20File%20Structure.pdf . Acessado em 07/10/2021.

[3] Microsoft. Microsoft Extensible Firmware Initiative FAT32 File System Specification. 2000. Disponível em https://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc .

[4] ___. FAT32 Utility Operations Guide (slides). Florida State University. Disponível em: http://www.cs.fsu.edu/ cop4610t/lectures/project3/Week12/Slides_week12.pdf. Acessado em 07/10/2021.

[5] ___. FAT32 Utility Operations Guide: rm and rmdir (slides). Florida State University. Disponível em: http:// www.cs.fsu.edu/ cop4610t/lectures/project3/Week13/Slides_week13.pdf. Acessado em 07/10/2021.