Universidade Tecnológica Federal do Paraná - UTFPR
Coordenação do Curso de Bacharelado em Ciência da Computação - COCIC

Caio Miglioli, Caio Eduardo Theodoro, Alexandre Scrocaro

Simulador de um Sistema de Arquivos

Projeto final

SISTEMAS OPERACIONAIS

Campo Mourão, Paraná, Brasil
2021

1 INTRODUÇÃO

O presente trabalho teve como objetivo realizar a implementação de um sistema de arquivos, simulando o comportamento do bash. A implementação contempla a utilização de funções para a manipulação de diretórios, arquivos, e até do próprio disco simulado.

2 DESCRIÇÃO DA ATIVIDADE

Implemente um sistema de arquivos simulado que será completamente contido em um único arquivo regular armazenado no disco. Esse arquivo de disco conterá diretórios, inodes, informações de blocos livres, blocos de dados de arquivos etc. Escolha algoritmos adequados para manter informações sobre blocos livres e para alocar blocos de dados (contíguos, indexados, encadeados). Seu programa aceitará comandos de sistema do usuário para criar/remover diretórios, criar/remover/abrir arquivos, ler/escrever de/para um arquivo selecionado e listar conteúdos de diretórios.

3 UTILIZAÇÃO DO SISTEMA

O sistema sempre irá iniciar no diretório root, e o manuseio do sistema é feito a partir de uma interface shell utilizando dos seguintes comandos:

3.1 Comandos Padrão:

help: Mostra quais os comandos disponíveis no sistema.
Utilização: 'help'

cd: Entra no diretório especificado.
Utilização: 'cd nomeDir', o nome deve ser exato, só é possível entrar em um diretório por vez. É possível verificar os diretórios disponíveis através do comando 'ls'.

ls: Mostra os arquivos e diretórios alocados no diretório atual.
Utilização: 'ls' / 'ls nome' / 'ls prefixo'. Quando utilizado sem um argumento listará todos os arquivos no diretório. Quanto utilizado com um argumento listará o diretório cujo nome é exatamente o do argumento. Quando utilizado com um prefixo seguido de um '*' o comando listará todos os arquivos cujo nome se inicia com o prefixo.*

pwd: Mostra qual o caminho percorrido pelo usuário do sistema.
Utilização: 'pwd'

mkdir: Cria um novo diretório no diretório atual.
Utilização: 'mkdir nome'. O nome deve ser o que será introduzido como nome do diretorio, nao é possível utilizar caminhos antes do nome para indicar a criação dentro de um diretório que

não é o atual.

cp: Copia um arquivo no diretorio atual para um novo diretório.
Utilização: 'cp nome novoNome' / 'cp nome dir1/dir2/novoNome' / 'cp nome dsc/novoNome.ext' / 'cp nome dsc/dir1/novoNome.ext'. O primeiro argumento (nome) nao permite a utilização de caminhos, portanto é necessário estar no diretório em que o arquivo se encontra. Caso o primeiro caminho do segundo argumento seja 'dsc', o comando buscará um arquivo externo com nome 'nome' e alocará no disco simulado na pasta 'dir1' com nome 'novoNome' e tipo 'ext'. Caso não haja uma extensão no nome, o tipo por padrão é inserido como 'file'. Não é possível copiar diretórios não vazios.

mv: Move um arquivo para o caminho selecionado.
Utilização: 'mv nome dir1/dir2/novoNome' / 'mv nome novoNome'. O primeiro argumento (nome) não permite a utilização de caminhos, portanto é necessário estar no diretório em que o arquivo se encontra. Caso não haja nenhum diretório especificado por '/' no segundo argumento, o comando renomeará o arquivo para 'novoNome'.
Não é possível utilizar este comando para exportar arquivos do disco para o sistema real, para isso utilize o 'exp'

rm: Remove um arquivo do diretório atual.
Utilização: 'rm nome'. O argumento 'nome' deve ser o mesmo do arquivo que deseja ser deletado. Não é possível utilizar caminhos.

format dsc: Formata o disco simulado com uma quantidade X de blocos de 4096 bytes.
Utilização: 'format dsc numDeBlocos'. O argumento 'numDeBlocos' deve ser um número inteiro.

3.2 Comandos extras:

exp: Exporta um arquivo de dentro do sistema simulado para o diretório em que se encontra o programa.
Utilização: 'exp nome novoNome.ext'. O argumento 'nome' deve ser o mesmo do arquivo que deseja ser exportado, é necessário também estar no diretório do arquivo pois o comando não aceita caminho. A extensão no disco simulado não é adicionada ao novoNome do arquivo, ficando a cargo do usuário especificá-la.

print fat: Mostra no shell o vetor da FAT com os valores associados de cada bloco.
Utilização: 'print fat'.

4 CONCEITO

Antes de começarmos a implementar o sistema, foi necessário pensar em um conceito, já que a forma como lidamos com os arquivos depende totalmente de como a alocação foi planejada.

4.1 Disco

Em um disco real, os espaços de memória já existem, um HDD de 1TB terá sempre 1TB independente se existe algo alocado ali ou não. O que muda é quais bytes são válidos ou não, mas isto fica a cargo do gerenciador de arquivos, não do hardware.

Por isso idealizamos um arquivo **disco.dsc** que sempre conterá *BLOCKQTDE x BLOCKSIZE* bytes, e o controle sobre quais dados são válidos fica a cargo do Gerenciador de Arquivos.

Para que seja feita uma divisão para o gerenciamento dos arquivos válidos, é feita uma abstração sobre o *disco.dsc* que chamamos de blocos. Por definição, o tamanho de cada bloco é de **512 Bytes** (BLOCKSIZE), e todos os blocos tem o mesmo tamanho.

O arquivo em si contem apenas bytes em sequência, sem uma divisão física do que é um bloco e do que é outro, portanto se tentarmos alocar um arquivo começando na metade do bloco 1 e terminando na metade do bloco 2, é totalmente possível de ser realizado (se feito utilizando comandos que não são do Gerenciador de Arquivo implementado).

4.2 Gerenciamento de Arquivo

Antes de pensar em como guardar os arquivos no disco, primeiro definimos o que é um arquivo, já que isto definiu qual algoritmo implementamos.

Decidimos que um arquivo é um vetor de bytes com tamanho N e com uma extensão, que por sua vez ditará como lidamos com este arquivo ao ler. Com isso definimos que tudo será um arquivo em nosso sistema, e ao ler, utilizamos a extensão nos metadados para saber como lidar com o arquivo. Um diretório, por exemplo, tem sua extensão como "dir", e para que um arquivo "dir"possa ser lido, foram criadas funções específicas para lidar com arquivos "dir".

Então para garantir cada arquivo possa ser guardado, mantido e lido sem que seja sobreescrito, implementamos um algoritmo baseado no **FAT**, que por sua vez é o responsável por modelar e manusear o disco.

A FAT foi implementada como um vetor de inteiros que armazena para cada índice de seu vetor o valor correspondente à aquele bloco. Por exemplo, na FAT o índice '0' está como 'R' pois o bloco '0' do sistema é um bloco reservado. Já um arquivo que utiliza multiplos blocos terá seu primeiro bloco apontando pro segundo e assim por diante até seu ultimo bloco, que por sua vez tem valor 'L' (Last).

A FAT por sua vez necessita de valores auxiliares para saber quantos blocos existem no disco, quando começa e quando termina um bloco, por exemplo. E para isso foi criado um arquivo, armazenado no bloco 0, chamado de **Superbloco**, que guarda as informações referentes ao disco em si e auxilia nas funções da FAT para a localização de cada espaço.

A organização inicial do disco, ou quando ele é formatado está ilustrada na figura 1.

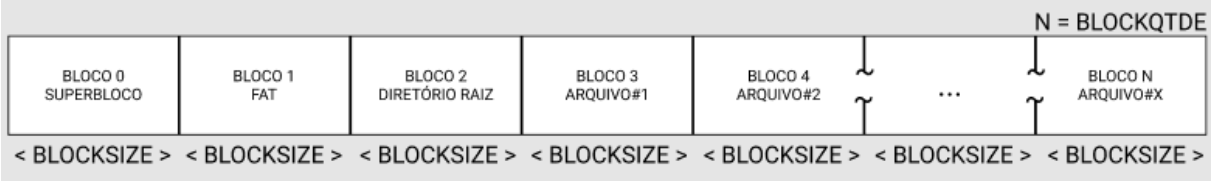


Figura 1 – Organização inicial do disco

4.3 Diretórios

Como, logicamente, os arquivos necessitam de organização para que sejam acessados de forma organizada pelo usuário, foi implementado um sistema hierárquico de diretórios, onde um diretório pode armazenar arquivos ou outros diretórios.

Um diretório foi idealizado como um arquivo que contém um tipo construído de dados chamado **"Meta"** que possui as informações referentes ao próprio diretório e um **vetor de struct Entradas**, onde cada uma delas referencia um arquivo no sistema, possuindo seus **metadados**.

É interessante notar que, se por algum motivo, um diretório se corromper, tudo que está dentro dele será perdido, pois mesmo o arquivo estando no disco, não será possível recuperá-lo sem seus metadados. Isso deve-se ao fato de que, por estarmos usando um algoritmo baseado em FAT, os metadados de cada arquivo devem ser guardados em um diretório, não em um i-node como em outro tipo de implementação.

Na figura 2 é ilustrado como foi idealizado um diretório.

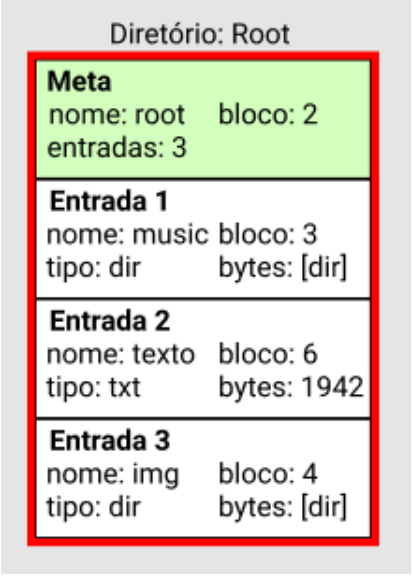


Figura 2 – Idealização de um diretório e suas entradas.

4.4 Inicialização do Sistema

Outra questão para ser solucionada é o manuseio do sistema, pois para que se possa criar, guardar, recuperar, mover ou copiar os arquivos é necessário que se garanta algumas condições, e para garantí-las é necessário haver uma **"Sessão"**.

Ao iniciar o programa, é executado algumas funções que são pré-requisitos pro funcionamento do sistema. A primeira delas é se existe um disco. Caso não exista é necessário criar um antes de continuar. A segunda função é carregar o arquivo do diretório 0, o **Superbloco**, e alocar suas informações na memória, pois é nele que estará guardada as informações cruciais do disco para que se consiga executar o programa.

Com o Superbloco na memória, o próximo passo é carregar o arquivo em que o **FAT** está alocado, conseguimos saber o endereço a partir das informações gravadas no Superbloco. Logo em seguida, de novo utilizando destas informações, carregamos na memória o diretório raíz e o alocamos em uma variável que guardará o diretório em que o usuário estiver. Esta variável é o que garante a navegação por meio dos arquivos, já que ao mudar de diretório, o diretório atual é desalocado e o novo é lido do disco e carregado naquela mesma variável.

Com estas 3 informações guardadas na memória a todo instante, é possível gerar uma interface para o manuseio do nosso disco simulado.

IMPLEMENTAÇÃO

5 MODULARIZAÇÃO

O programa foi implementado em 3 módulos: O Main, com a implementação do Bash e a leitura dos comandos digitados, o Commands que faz todo o processo lógico do que o comando chamado deve executar, e o FATACC, que manipula os dados dos arquivos no disco.

O FATACC além de conter suas funções internas, contém também todas as funções relacionadas ao funcionamento do programa como um todo, são elas que manipulam o disco. Já em commands, tem-se as funções principais e que serão chamadas pelo usuário, tais fazem uso de funções do FATACC em sua implementação. Por fim, no main, teremos o shell que receberá as entradas do usuário, além de conter funções auxiliares para manipular essas entradas.

6 MÉTODOS

6.1 Parte 1: Organização geral do sistema de arquivos.

6.1.1 Superblock

No superbloco, temos as informações principais acerca de posições e tamanhos de blocos. Tais informações serão utilizadas em funções que precisam de manipular a fat ou o diretório raiz, assim como quando se precisa de tamanho, posição ou quantidade de blocos que o disco possui. A estrutura de dados criada para o superbloco pode ser observada na figura 3.

```
typedef struct{
    char tipo[8];   //8b
    //fat
    int fatPos;     //4b
    //diretorio raiz
    int rootPos;    //4b
    //blocos
    int blockPos;   //4b
    int blockQtde;  //4b
    int blockSize;  //4b
} Superblock;       //28bytes
```

Figura 3 – Struct Superblock

6.1.2 **Fat**

O algoritmo escolhido para controlar a alocação de blocos em determinada posição do disco foi o FAT, com o qual temos uma alocação encadeada, que nos permite marcar os blocos livres, reservados, defeituosos e se é o último de um arquivo específico. Para tanto utilizamos a biblioteca limits.h. O funcionamento da FAT está exemplificado na figura 4, enquanto seu conteúdo pode ser observado na [2]figura 5.

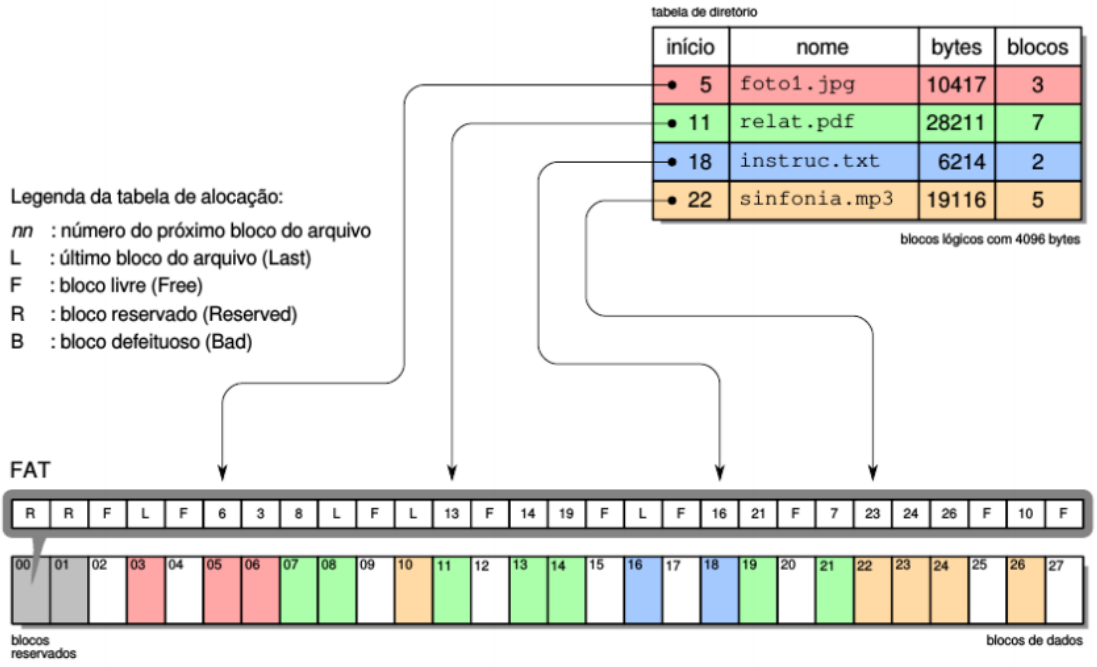


Figura 4 – Funcionamento do FAT

```
// Fat
typedef int Fat;            //bytes = 4 * Superbloco.blockQtde
#define FAT_L (INT_MAX)     // Ultimo bloco do arquivo (Last)
#define FAT_F (INT_MAX-1)   // Bloco livre (Free)
#define FAT_R (INT_MAX-2)   // Bloco reservado (Reserved)
#define FAT_B (INT_MAX-3)   // Bloco defeituoso (Bad)
```

Figura 5 – Estrutura do FAT

6.1.3 File chunk

O file chunk foi criado para possibilitar o manejamento e manipulação de blocos de arquivos, assim como o controle de tamanho de um arquivo através de sua quantidade de bytes. Sua utilização pode ser observada em funções de load, createDirectory, openFile, entre outras. Sua estrutura pode ser observada na figura 6.

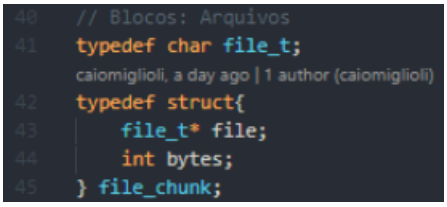


Figura 6 – Struct file chunk

6.1.4 Entry

A Entry consiste em uma estrutura que referencia um arquivo na memória, ela contém as informações necessárias para sua manipulação. Dentro de um diretório há diversas entries, sendo elas arquivos ou diretórios. Na figura 7 pode-se observar sua estrutura.

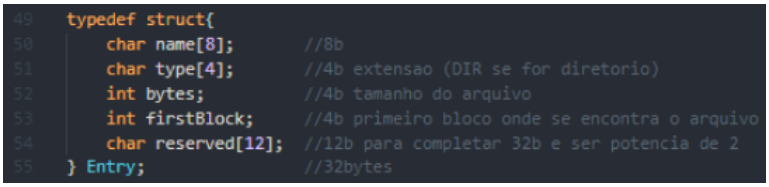


Figura 7 – Struct Entry

6.1.5 Meta

A struct DirMeta estará presente em todos os diretórios do disco, é ela que guarda todas informações pertinentes para as manipulações dos diretórios. A figura 8 mostra o conteúdo da estrutura.

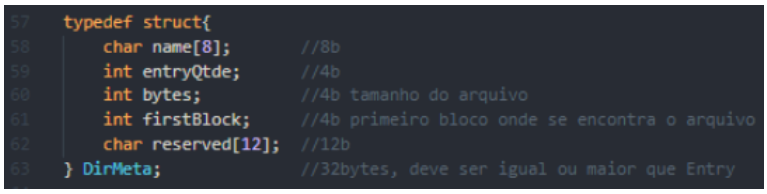
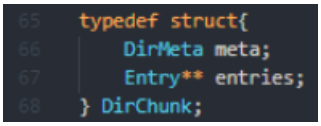


Figura 8 – Struct DirMeta

6.1.6 Dir Chunk

A struct DirChunk foi criada para facilitar a manipulação dos diretórios, com ela é possível navegar entre as entradas de cada diretório, além de consultar seus metadados. A figura 9 mostra seu conteúdo.



```
typedef struct{
    DirMeta meta;
    Entry** entries;
} DirChunk;
```

Figura 9 – struct DirChunk

6.2 Parte 2: Funções auxiliares.

6.2.1 char** split(char* command);

Dada uma string, tem como função retirar os espaços e separá-la em um vetor de strings (usada para a lógica dos comando de manipulação). É percorrida uma string contando seus delimitadores (no caso, utilizamos um espaço - ' ') e com um strtok, são inseridas palavras para cada delimitador encontrado. A figura 10 mostra o código, bem como o explica linha-a-linha.

```
char** split(char* command){

    command = strtok(command, "\n"); //remove o \n do final da string

    char** result    = 0; //ponteiro para o resultado (string como array de char)
    size_t count     = 0; //contador de palavras
    char* tmp  = command; //ponteiro temporario para percorrer a string
    char* last_comma = 0;
    char delimiter = ' ', delim[2]; //delimitador
    delim[0] = delimiter, delim[1] = 0;

    while (*tmp){ //enquanto nao chegar no fim da string
        if (delimiter == *tmp){ //se encontrar um delimitador
            count++; //incrementa o contador
            last_comma = tmp; //salva o ponteiro para o ultimo espaço encontrado
        }
        tmp++; //incrementa o ponteiro
    }
    count += last_comma < (command + strlen(command) - 1); //se o ultimo espaço nao for
                                                            // o fim da string, incrementa o contador
    count++;
    result = malloc(sizeof(char*) * count); //aloca o tamanho do contador
    if (result){ //se a alocacao foi bem sucedida
        size_t idx  = 0; //indice do resultado
        char* token = strtok(command, delim); //pega a primeira palavra

        while (token){ //enquanto nao chegar no fim da string
            assert(idx < count); //se o indice for maior que o contador, algo de errado aconteceu
            *(result + idx++) = strdup(token); //copia a palavra para o resultado
            token = strtok(0, delim); //pega a proxima palavra
        }
        assert(idx == count - 1); //se o indice for diferente do contador, algo de errado aconteceu
        *(result + idx) = 0; //faz o fim da string
    }
    return result; //retorna o resultado
}
```

Figura 10 – Função split

6.2.2 void listenCommand(Superblock** sb, Fat** fat, DirChunk** diretorioAtual, char* command,char* pathing)

Função de "escuta" para o terminal. Dado qualquer comando no simulador, trata ele de acordo com a manipulação requirida (exemplo: cd, mkdir, etc.). Sua implementação é simples, se tratando somente de comparações de string com as de comando (utilizando a função split

para pegar palavra a palavra da string), bem como mandando os valores do bash simulado para as funções que os tratam. As figuras 11 e 12 mostra o código e o explica linha-a-linha.

```
80  void listenCommand(Superblock** sb, Fat** fat, DirChunk** diretorioAtual, char* command,char* pathing){
81      //pegar a linha de comando e dividir em palavras
82      char** vals = split(command); //cria um vetor de strings com as palavras da linha de comando
83      char *cmd, *source, *destination; //cria variaveis para guardar os valores de cada palavra
84      int cmdCount = 0; //contador de palavras
85      if(*(vals + 0) != NULL){ //se a primeira palavra nao for nula
86          cmd = *(vals + 0); //
87          cmdCount++;
88      }
89      if(*(vals + 1) != NULL){ //se a segunda palavra nao for nula
90          source = *(vals + 1); //pega o valor da segunda palavra
91          cmdCount++;
92      }
93      if(*(vals + 2) != NULL){ //se a terceira palavra nao for nula
94          destination = *(vals + 2); //pega o valor da terceira palavra
95          cmdCount++;
96      }
97
98      //==============================
99      //**********************COMANDOS
100     if(strcmp(cmd, "help") == 0){ //se a primeira palavra for help
101         help();
102
103
104     }else if(strcmp(cmd,"cd") == 0){ //se a primeira palavra for cd
105         if(!source || cmdCount < 2){
106             printf("cd <dirName>\n");
107             return;
108         }
109         *diretorioAtual = changeDirectory(*sb, *fat, *diretorioAtual, source, pathing); //muda o diretorio atual
110
111
112     }else if(strcmp(cmd,"mkdir") == 0){ //se a primeira palavra for mkdir
113         if(!source || cmdCount < 2){ //se nao tiver o nome do diretorio
114             printf("mkdir <dirName>\n");
115             return;
116         }
117         makeDirectory(*sb, *fat, *diretorioAtual, source); //cria o diretorio
118
119
120     }else if(strcmp(cmd,"rm") == 0){ //se a primeira palavra for rm
121         if(!source || cmdCount < 2){ //se nao tiver o nome do item (diretorio ou arquivo)
122             printf("rm <Name>\n");
123             return;
124         }
125         removeItem(*sb, *fat, *diretorioAtual, source);
126
127
128     }else if(strcmp(cmd,"cp") == 0){
129         if(!source || !destination || cmdCount < 3){ //se nao tiver o nome do item (diretorio ou arquivo)
130             printf("cp <source> <destination>\n");
131             return;
132         }
133         copyItem(*sb, *fat, *diretorioAtual, source, destination);
134
```

Figura 11 – Função de escuta

```
137     }else if(strcmp(cmd,"exp") == 0){
138         if(!source || !destination || cmdCount < 3){ //se nao tiver o nome do item (diretorio ou arquivo)
139             printf("exp <source> <newname>\n");
140             return;
141         }
142         exportItem(*sb, *fat, *diretorioAtual, source, destination);
143 
144     }else if(strcmp(cmd,"mv") == 0){ //se a primeira palavra for mv
145         if(!source || !destination || cmdCount < 3){ //se nao tiver o nome do item (diretorio ou arquivo)
146             printf("mv <source> <destination>\n");
147             return;
148         }
149         moveItem(*sb, *fat, *diretorioAtual, source, destination); //move o item (diretorio ou arquivo)
150 
151 
152     }else if(strcmp(cmd, "ls") == 0){ //se a primeira palavra for ls
153         listDirectory(*diretorioAtual, source); //lista o diretorio atual
154 
155 
156     }else if(strcmp(cmd, "pwd") == 0){ //se a primeira palavra for pwd
157         showPath(pathing); //mostra o caminho atual
158 
159 
160     }else if(strcmp(cmd, "format") == 0 && strcmp(source, "dsc") == 0){ //se a primeira palavra for format e o segundo for dsc
161         int x = format_dsc(BLOCKSIZE, atoi(destination)); //formata o disco com a quantidade de blocos especificada
162         if(x != 1){
163             //atualiza a memoria
164             //load superblock
165             free(*sb); //libera a memoria do superblock
166             *sb = facc_loadSuperblock(); //carrega o superblock
167 
168             //load fat
169             free(*fat); //libera a memoria do fat
170             *fat = facc_loadFat(*sb); //carrega o fat
171 
172             // READ_ROOT
173             facc_unloadDirectory(*diretorioAtual); //libera a memoria do diretorio atual
174             *diretorioAtual = facc_loadDir(*sb, *fat, (*sb)->rootPos); //carrega o diretorio raiz
175         }
176 
177     }else if(strcmp(cmd, "print") == 0 && strcmp(source, "fat") == 0){
178         printFat(*sb, *fat);
179     }else{ //se nao for nenhuma das opcoes
180         printf("Comando '%s' não encontrado, digite 'help' para ver os comandos disponíveis.\n", cmd);
181     }
182 }
```

Figura 12 – Função de escuta pt. 2

6.2.3 void createPathing()

Função usada no main para criar o pathing inicial. O path foi criado como uma string que será atualizada a cada mudança de diretório que ocorrer. A figura 13 mostra seu conteúdo.

```
190 char* createPathing(){
191     char* path = (char*) malloc (sizeof(char)*250); //aloca memoria para o path
192     strcpy(path,"~root"); //copia o path inicial
193 
194     return path; //retorna o path
195 }
```

Figura 13 – Função de criação do pathing

6.3 Parte 3: Funções principais.

6.3.1 Carregar diretório do disco: DirChunk* facc_loadDir(Superblock* sb, Fat* fat, int firstBlock)

Função usada para carregar o diretório do disco para a memória (RAM). É aberto o arquivo passando o superbloco, fat e o bloco em que se encontra o diretório. Após isso é alocado

um diretório “directory” (que será retornado - carregado - pela função), abre o diretório a ser carregado e o passa para um file_chunk “fc” a partir da função openDir. Se o chunk (file_chunk) recebido for NULL, ou seja, ocorreu um erro ao abrir o arquivo, a função retorna NULL. Após isso copia todos os dados do diretório aberto (fc) para o que será retornado (directory). E então podemos desalocar o fc e retornar o directory. A figura 14 mostra seu conteúdo.

```
98  DirChunk* facc_loadDir(Superblock* sb, Fat* fat, int firstBlock){ // carrega o diretorio do disco
99      DirChunk* directory = (DirChunk*)malloc(sizeof(DirChunk)); // cria o diretorio
100
101     //abrir arquivo
102     file_chunk* fc = openDir(sb, fat, firstBlock); // abre o diretorio
103     if(fc == NULL){ // se nao conseguiu abrir
104         printf("Erro ao abrir diretorio\n"); // erro
105         return NULL;
106     }
107
108     //alocando meta a partir do arquivo
109     memcpy(&(directory->meta), fc->file, sizeof(DirMeta)); // copia o meta do diretorio para o retorno
110
111     //criando vetor de entries
112     directory->entries = (Entry**)malloc(directory->meta.entryQtde * sizeof(Entry*)); // aloca o vetor de entries
113
114     //alocando entries a partir do arquivo
115     for(int i=0; i<directory->meta.entryQtde; i++){ // para cada entry
116         directory->entries[i] = (Entry*)malloc(sizeof(Entry)); // aloca o entry
117         memcpy(directory->entries[i], fc->file + sizeof(DirMeta) + i*sizeof(Entry), sizeof(Entry)); // copia o entry para o retorno
118     }
119
120     free(fc->file); // libera o file_chunk
121     free(fc); // libera o file_chunk
122     return directory; // retorna o diretorio
123 }
```

Figura 14 – Função para carregar o diretório

6.3.2 Carregar Fat: Fat* facc_loadFat(Superblock* sb)

Função usada para carregar o fat do disco. É alocado o fat de acordo com a quantidade de blocos do superbloco, então se lê o fat do disco usando a função readBlock e é retornado o fat pela função. A figura 15 mostra seu conteúdo.

```
86  Fat* facc_loadFat(Superblock* sb){ // carrega o fat do disco
87      //fat
88      Fat* fat = (Fat*)calloc(sb->blockQtde, sizeof(Fat)); // cria o fat
89
90      // Fat fat2[sb2.blockQtde];
91      file_t* tmp = readBlock(sb, sb->fatPos, sizeof(Fat)*sb->blockQtde); // le o fat do disco
92      memcpy(fat, tmp, sizeof(Fat)*sb->blockQtde); // copia o fat para o retorno
93
94      free(tmp); // libera o file_t
95      return fat; // retorna o fat
96  }
```

Figura 15 – Função para carregar o fat

6.3.3 Carregar Superbloco: Superblock* facc_loadSuperblock()

Função usada para criar/formatar/carregar o superbloco. É alocado o superbloco e atribuido seus valores de tamanho e quantidade, feito isso se lê o superbloco do arquivo, passando o mesmo superbloco alocado anteriormente, e então é retornado o superbloco pela função. A figura 16 mostra seu conteúdo.

```
71  Superblock* facc_loadSuperblock(){
72      //superblock que sera retornado
73      Superblock* sb = (Superblock*)malloc(sizeof(Superblock)); // cria o superbloco
74      sb->blockSize = 512; //valores estaticos para poder usar no readBlock
75      sb->blockQtde = 1;
76
77      //read superblock do arquivo
78      file_t* tmp = readBlock(sb, 0, sizeof(Superblock)); // le o superbloco do disco
79      memcpy(sb, tmp, sizeof(Superblock)); // copia o superbloco para o retorno
80
81      free(tmp); // libera o file_t
82      return sb; // retorna o superbloco
83  }
```

Figura 16 – Função para carregar superbloco

6.3.4 Desalocar diretório: void facc_unloadDirectory(DirChunk* dir)

Função usada para desalocar diretório da memória. É passado o diretório por parâmetro e para cada entry presente, é liberado seu espaço. Por fim, se libera o vetor de entries e seu diretório. A figura 17 mostra seu conteúdo.

```
126 void facc_unloadDirectory(DirChunk* dir){ // desaloca o diretorio
127     for(int i=0; i<dir->meta.entryQtde; i++){ // para cada entry
128         free(dir->entries[i]); // libera o entry
129     }
130     free(dir->entries); // libera o vetor de entries
131     free(dir); // libera o diretorio
132 }
```

Figura 17 – Função para desalocar o diretório

6.3.5 Encontrar bloco livre: int facc_findFreeBlock(Superblock* sb, Fat* fat)

Função usada para encontrar um bloco livre no fat. É percorrido o fat comparando cada bloco com FAT_L (que indica que o bloco está livre) e então se retorna a primeira posição livre encontrada. A figura 18 mostra seu conteúdo.

```
134 int facc_findFreeBlock(Superblock* sb, Fat* fat){ // encontra um bloco livre
135     for(int i=3; i < sb->blockQtde; i++){ // para cada bloco
136         if(fat[i] == FAT_F){ // se for livre
137             return i; // retorna a posicao
138         }
139     }
140     return -1; // nao achou
141 }
```

Figura 18 – Função para encontrar bloco livre

6.3.6 Formatar disco: void facc_format(int blockSize, int blockQtde)

Função usada para formatar todo o disco. É criado o disco usando o comando fopen, então é escrita a quantidade de blocos no disco usando um buffer temporário. Após isso se fecha o disco e cria um superbloco para ele, atribuindo posições para o fat e um diretório raiz (root), eles recebem as posições respectivas 1 e 2. Também é feita a atribuição de quantidade de blocos e tamanho do bloco. Feito isso o superbloco é escrito no disco. No próximo passo, se cria um fat,

preenchendo suas 3 primeiras posições como reservadas (superbloco, fat e root), e as restantes como livres, então se escreve o fat no disco. Por ultimo, se cria o meta do diretório pai, para ser inserido na entry "..", e então é criado um diretório (root) usando a função createDirectory, que é escrita no disco e atualizada no fat. Se desaloca as variáveis usadas e o código é finalizado. As figuras 19 e 20 mostra seu conteúdo.

```
// Formata o disco, deixando-o com um superbloco, fat e um diretorio root
void facc_format(int blockSize, int blockQtde){
    /* criar disco */
    FILE* dsc = fopen(DISCO, "w"); // cria o disco
    char* tmp2 = (char*)calloc(blockSize, sizeof(char)); // cria um buffer temporario
    for(int i=0; i<blockQtde; i++) fwrite(tmp2, 1, blockSize, dsc); // escreve o buffer no disco
    free(tmp2); // libera o buffer
    fclose(dsc);     // fecha o disco

    /* criar superbloco */
    Superblock sb;
    strcpy(sb.tipo, "fatacc"); // tipo do superbloco

    sb.fatPos = 1; // posicao do fat
    sb.rootPos = 2; // posicao do diretorio root
    sb.blockPos = 3; // posicao do primeiro bloco de dados
    sb.blockSize = blockSize; // tamanho do bloco
    sb.blockQtde = blockQtde; // quantidade de blocos

    writeBlock(&sb, 0, &sb, sizeof(Superblock)); // escreve o superbloco no disco
    /* fat */
    Fat* fat = (Fat*)malloc(sizeof(Fat) * sb.blockQtde); // cria o fat

    for(int i=0; i<3; i++){ // preenche o fat
        fat[i] = FAT_R; //reservados
    }
    for(int i=3; i<blockQtde; i++){
        fat[i] = FAT_F;     //livres
    }
```

Figura 19 – Função de formatar disco

```
    writeBlock(&sb, sb.fatPos, fat, sizeof(Fat)*sb.blockQtde); // escreve o fat no disco
    /* dirRoot */
    //cria o que seria o meta do diretorio pai para inserir na entry ".."
    DirMeta fatherMeta;
    fatherMeta.firstBlock = sb.rootPos; // posicao do primeiro bloco do diretorio pai

    // 1 - encontrar posicao
    // -> por ser root, usara a posicao superbloco.rootPos

    // 2 - criar diretorio filho e guardar no disco
    file_chunk* file = createDirectory("root", sb.rootPos, &fatherMeta); // cria o diretorio root
    writeBlock(&sb, sb.rootPos, file->file, file->bytes); // escreve o diretorio root no disco
    updateFat(&sb, fat, sb.rootPos, FAT_L); // atualiza o fat

    // 3 - criar entry no diretorio pai e guardar no disco
    // -> nao ha diretorio pai pois este eh o root

    /* free */
    free(fat);     // libera o fat
    free(file->file); // libera o file_chunk
    free(file); // libera o file_chunk
}
```

Figura 20 – Função de formatar disco pt. 2

6.3.7 Atualizar diretório(adição): void facc_updateDirAdd(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, Entry* ref)

Função usada para atualizar o diretorio após adicionar um novo entry. É criado um vetor de entries contendo quantidades +1. Após isso, copio cada entry existente para o novo vetor, e por fim coloco o novo vetor no lugar do vetor original, desalocando o antigo e atualizando o diretório atual. A figura 21 mostra seu conteúdo.

```
void facc_updateDirAdd(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, Entry* ref){ // atualiza o diretorio apos adicionar um novo entry
    //criando vetor de entries
    Entry** newEntries = (Entry**)malloc(sizeof(Entry*) * (diretorioAtual->meta.entryQtde+1)); // aloca o vetor de entries

    //adiciono as entradas do dir->entries para o newEntries
    int i = 0;
    while(i<diretorioAtual->meta.entryQtde){ // para cada entry
        newEntries[i] = diretorioAtual->entries[i]; // copia o entry
        i++; // incrementa
    }

    //adiciono a nova entrada no newEntries
    newEntries[i] = ref;

    //coloco o NewEntries no lugar do antigo entries
    free(diretorioAtual->entries);
    diretorioAtual->entries = newEntries;

    //coloco no meta que foi adicionado mais um diretorio
    diretorioAtual->meta.entryQtde++;

    updateDirectory(sb, fat, diretorioAtual); // atualiza o diretorio
}
```

Figura 21 – Função para adicionar uma entry

6.3.8 Atualizar diretório(remoção): void facc_updateDirDel(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, int index)

Função usada para atualizar o diretório após remover um novo entry. É excluído o arquivo usando a função deleteFile com a posição do arquivo a ser deletado, então é removida a entrada na posição do arquivo, copiado o último arquivo para a posição do arquivo removido e diminuida a quantidade de entries. Por fim, se atualiza o diretório com o diretório atual alterado. A figura 22 mostra seu conteúdo.

```
void facc_updateDirDel(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, int index){ // atualiza o diretorio apos deletar um entry
    if(index < 2 || index > diretorioAtual->meta.entryQtde){ // se o index for invalido
        printf("Erro ao remover arquivo.\n"); // erro
        return;
    }

    //excluindo o arquivo.
    deleteFile(sb, fat, diretorioAtual->entries[index]->firstBlock); // exclui o arquivo

    //removendo a entrada.
    free(diretorioAtual->entries[index]); // libera o entry
    int ultimoEntry = diretorioAtual->meta.entryQtde - 1; // pega o ultimo entry


    diretorioAtual->entries[index] = diretorioAtual->entries[ultimoEntry]; // copia o ultimo entry para o index
    diretorioAtual->entries[ultimoEntry] = NULL; // libera o ultimo entry

    diretorioAtual->meta.entryQtde--; // diminui o entryQtde

    updateDirectory(sb, fat, diretorioAtual); // atualiza o diretorio
}
```

Figura 22 – Função para remover uma entry

6.3.9 Atualizar diretório: void updateDirectory(Superblock* sb, Fat* fat, DirChunk* directory)

Função usada para atualizar o diretório. É removido o antigo diretório da memória e criado um chunk com a quantidade em bytes de cada entry, então se cria um file_t contendo as informações do chunk e para cada entry, copia para o file_t. por fim salvo o file_t no disco, no lugar do diretório anterior. A figura 23 mostra seu conteúdo.

```
197 void updateDirectory(Superblock* sb, Fat* fat, DirChunk* directory){ // atualiza o diretorio
198     //removo o antigo da memoria
199     int firstBlock = directory->meta.firstBlock; // pega o primeiro bloco
200     deleteFile(sb, fat, firstBlock); // exclui o arquivo
201 
202     //Criando chunk com o file_t e controle de bytes
203     file_chunk* fc = (file_chunk*)malloc(sizeof(file_chunk)); // cria o file_chunk
204     fc->bytes = sizeof(DirMeta) + sizeof(Entry)*directory->meta.entryQtde; // define o tamanho do file_chunk
205 
206     //crio um file_t contendo as informacoes do DirChunk
207     fc->file = (file_t*)malloc(fc->bytes * sizeof(char)); // aloca o file_t
208     memcpy(fc->file, &directory->meta, sizeof(DirMeta)); // copia o meta do diretorio para o file_t
209     for(int i=0; i<directory->meta.entryQtde; i++){ // para cada entry
210         memcpy(fc->file+sizeof(DirMeta)+(i*sizeof(Entry)), directory->entries[i], sizeof(Entry)); // copia o entry para o file_t
211     }
212 
213     //coloco no disco o file_t
214     saveFile(sb, fat, fc, firstBlock); // salva o file_t no disco
215 }
```

Figura 23 – Função para atualizar o diretório

6.3.10 Salvar o file chunk: void saveFile(Superblock* sb, Fat* fat, file$_{chunk} * fc, int block$)

Função usada para salvar o file_chunk no disco. É pego o tamanho do file_chunk enviado por parâmetro e enquanto houver bytes se escreve o bloco na memória. Para cada bloco se atualiza o fat, atribuindo posição ocupada. A última posição ocupada é atribuída como última. A figura 24 mostra seu conteúdo.

```
217 void saveFile(Superblock* sb, Fat* fat, file_chunk* fc, int block){ // salva o file_chunk no disco
218     int bytes = fc->bytes; // pega o tamanho do file_chunk
219     int i = 0; // contador
220     int lastBlock = -1; // ultimo bloco
221 
222     while(bytes > 0){
223         //escrevo o bloco na memoria
224         int bufferSize = (bytes > sb->blockSize) ? sb->blockSize : bytes; // define o tamanho do buffer
225         writeBlock(sb, block, fc->file + (i * sb->blockSize), bufferSize); // escreve o bloco na memoria
226 
227         // Faco com que o bloco anterior aponte para este bloco
228         if(lastBlock != -1){
229             updateFat(sb, fat, lastBlock, block); // atualiza o fat
230         }
231 
232         //faco com que este bloco esteja ocupado e receba status de ultimo
233         updateFat(sb, fat, block, FAT_L);
234 
235         //atualizo o ultimo bloco pra proxima iteracao
236         lastBlock = block;
237 
238         //update das variaveis de controle
239         i++;
240         block = facc_findFreeBlock(sb, fat);
241         bytes = bytes - sb->blockSize;
242     }
243 }
```

Figura 24 – Função para salvar o file chunk

6.3.11 Deletar arquivo: void deleteFile(Superblock* sb, Fat* fat, int firstBlock)

Função usada para deletar um arquivo. Ele percorre a Fat na posição passada por parâmetro e para cada bloco relacionado, libera o bloco. A figura 25 mostra seu conteúdo.

```
245  void deleteFile(Superblock* sb, Fat* fat, int firstBlock){
246
247      int block = firstBlock; // primeiro bloco
248      int aux = block; // auxiliar
249
250      do{ // enquanto o bloco nao for o ultimo
251          if(fat[block] == FAT_B || fat[block] == FAT_F) return; // erro
252
253          block = fat[block]; // pega o proximo bloco
254
255          fat[aux] = FAT_F; // libera o bloco
256          aux = block; // atualiza o auxiliar
257      } while(block != FAT_L);
258  }
```

Figura 25 – Função para salvar o file chunk

6.3.12 Ler bloco: file_t* readBlock(Superblock* sb, int blockNum, int bufferSize)

Função usada para ler um bloco. É aberto o disco no modo de leitura, movido o ponteiro com fseek até o bloco selecionado e então se faz a leitura dos bits na posição usando fread. Feito isso é fechado o arquivo de disco e retornado o buffer onde foi alocado os bytes lidos. A figura 26 mostra seu conteúdo.

```
359  file_t* readBlock(Superblock* sb, int blockNum, int bufferSize){
360      if(blockNum >= sb->blockQtde) return NULL; // se o bloco for invalido
361      if(bufferSize <= 0 || bufferSize > sb->blockSize) return NULL; // se o buffer for invalido
362
363      FILE* dsc = fopen(DISCO, "r"); // abre o disco
364      file_t* buffer = (file_t*)malloc(sizeof(char)*bufferSize); // aloca o buffer
365
366      //ir para o bloco selecionado e ler os bytes
367      fseek(dsc, blockNum*sb->blockSize, SEEK_SET); // ir para o bloco selecionado
368      fread(buffer, sizeof(char), bufferSize, dsc); // ler os bytes
369
370      fclose(dsc); // fechar o disco
371      return buffer; // retorna o buffer
372  }
```

Figura 26 – Função para ler bloco

6.3.13 Abrir diretório: file_chunk* openDir(Superblock* sb, Fat* fat, int firstBlock)

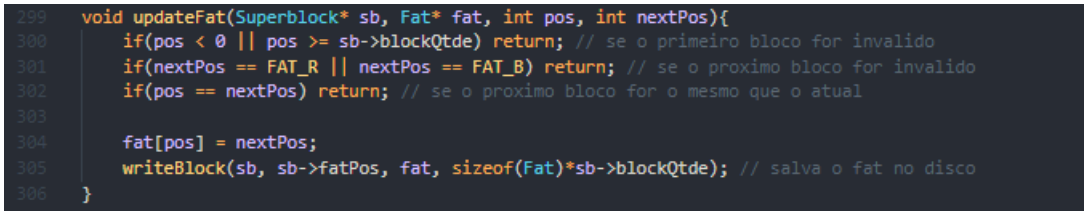
Função usada para abrir um diretório. Verifica se o primeiro bloco do diretório é válido, caso contrário retorna NULL. Em seguida faz um laço do while para descobrir a quantidade de blocos que o diretório utiliza, tendo isso pode-se alocar um file chunk baseado na quantidade de blocos do diretório. E então faz-se outro laço do while lendo cada bloco do diretório e adicionando-os ao file chunk. Com isso feito, retorna o file chunk (abre o diretório em memória). A figura 27 mostra seu conteúdo.

```
file_chunk* openDir(Superblock* sb, Fat* fat, int firstBlock){
    if(firstBlock < 0 || firstBlock >= sb->blockQtde) return NULL; // se o primeiro bloco for invalido

    int blocks_qtde = 0;
    int block = firstBlock;

    //guarda em blocks_qtde quantos blocos o arquivo utiliza
    do{  // enquanto o bloco nao for o ultimo
        if(fat[block] == FAT_B || fat[block] == FAT_F) return NULL; // erro
        blocks_qtde++; // incrementa o contador
        block = fat[block]; // pega o proximo bloco
    }while(block != FAT_L);

    //cria o file_chunk com a stream do arquivo e seus bytes para retornar
    file_chunk* fc = (file_chunk*)malloc(sizeof(file_chunk)); // cria o file_chunk
    fc->bytes = blocks_qtde * sb->blockSize; // define o tamanho do file_chunk
    fc->file = (file_t*)malloc(fc->bytes);  // aloca o file_t

    file_t* tmp; // auxiliar
    int i = 0;
    block = firstBlock; // primeiro bloco

    //faz a leitura de cada bloco e adiciona na stream "file_t fc->file"
    do{ // enquanto o bloco nao for o ultimo
        if(fat[block] == FAT_B || fat[block] == FAT_F) return NULL; // erro

        tmp = readBlock(sb, block, sb->blockSize); // le o bloco
        memcpy(fc->file + (i*sb->blockSize), tmp, sb->blockSize); // copia o bloco para o file_t
        free(tmp); // libera o bloco temporario

        i++; // incrementa o contador
        block = fat[block]; // pega o proximo bloco
    }while(block != FAT_L);

    //retorno o file_chunk
    return fc;
}
```

Figura 27 – Função para abrir diretório

6.3.14 Atualizar fat: void updateFat(Superblock* sb, Fat* fat, int pos, int nextPos)

Função usada para atualizar o fat. É passada a posição e a próxima posição por parâmetro, se o primeiro e o próximo bloco não forem inválidos, nem o mesmo bloco, é atribuído ao fat na posição passada o próximo bloco, e então é salvo o fat no disco utilizando a função writeBlock. A figura 28 mostra seu conteúdo.



```
void updateFat(Superblock* sb, Fat* fat, int pos, int nextPos){
    if(pos < 0 || pos >= sb->blockQtde) return; // se o primeiro bloco for invalido
    if(nextPos == FAT_R || nextPos == FAT_B) return; // se o proximo bloco for invalido
    if(pos == nextPos) return; // se o proximo bloco for o mesmo que o atual

    fat[pos] = nextPos;
    writeBlock(sb, sb->fatPos, fat, sizeof(Fat)*sb->blockQtde); // salva o fat no disco
}
```

Figura 28 – Função para atualizar a fat

6.3.15 Abrir arquivo: file_chunk* openFile(Superblock* sb, Fat* fat, Entry* meta)

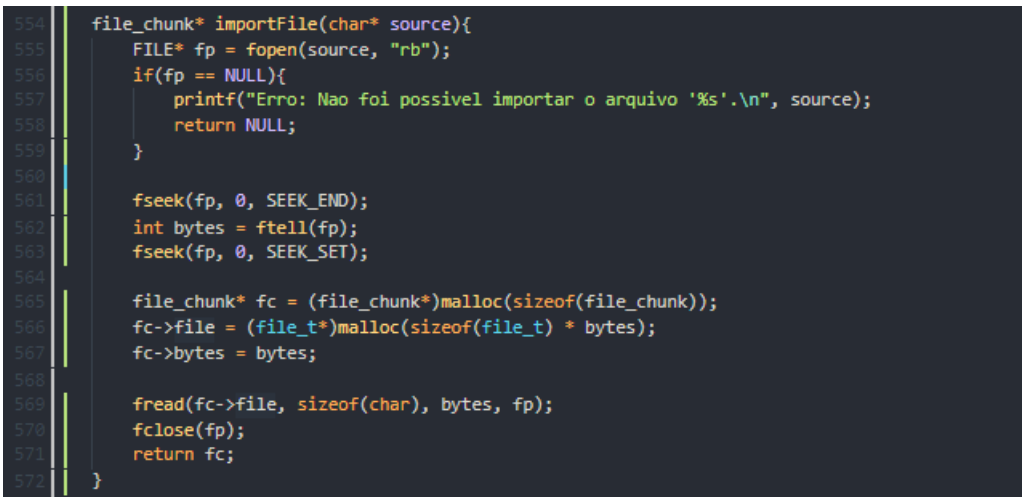
Função usada para abrir um arquivo. Verifica se o primeiro bloco do arquivo é válido, caso contrário retorna NULL. Em seguida aloca um file chunk com os dados do arquivo (entry), após isso faz um laço do while lendo cada bloco do arquivo e adicionando-os ao file chunk. Com isso feito, retorna o file chunk (abre o arquivo em memória). A figura 29 mostra seu conteúdo.

```c
file_chunk* openFile(Superblock* sb, Fat* fat, Entry* meta){
    if(meta->firstBlock < 0 || meta->firstBlock >= sb->blockQtde) return NULL; // se o primeiro bloco for invalido

    //cria o file_chunk com a stream do arquivo e seus bytes para retornar
    file_chunk* fc = (file_chunk*)malloc(sizeof(file_chunk)); // cria o file_chunk
    fc->bytes = meta->bytes; // define o tamanho do file_chunk
    fc->file = (file_t*)malloc(fc->bytes);  // aloca o file_t

    file_t* tmp; // auxiliar
    int i = 0;
    int block = meta->firstBlock;
    int bytes = meta->bytes;

    //faz a leitura de cada bloco e adiciona na stream "file_t fc->file"
    do{ // enquanto o bloco nao for o ultimo
        if(fat[block] == FAT_B || fat[block] == FAT_F) return NULL; // erro

        int readingBytes = (bytes > sb->blockSize) ? sb->blockSize : bytes;
        tmp = readBlock(sb, block, readingBytes); // le o bloco
        memcpy(fc->file + (i*sb->blockSize), tmp, readingBytes); // copia o bloco para o file_t
        free(tmp); // libera o bloco temporario

        i++; // incrementa o contador
        block = fat[block]; // pega o proximo bloco
        bytes -= readingBytes;
    }while(block != FAT_L);

    if(bytes != 0){
        printf("Warning: O arquivo foi lido com mais ou menos bytes que o devido.\n");
    }

    //retorno o file_chunk
    return fc;
}
```

Figura 29 – Função para abrir arquivo

6.3.16 Importar arquivo: file_chunk* importFile(char* source)

Função usada para importar um arquivo. É aberto o arquivo, então ele é percorrido e seu tamanho é alocado em uma variável “bytes”, feito isso se retorna novamente para o começo do arquivo usando fseek SEEK_SET. Então se aloca memória para um file_t com o tamanho do arquivo, que é lido e salvo na memória do file_chunk. Por fim, é fechado o arquivo e retornado o file_chunk. A figura 30 mostra seu conteúdo.



```c
file_chunk* importFile(char* source){
    FILE* fp = fopen(source, "rb");
    if(fp == NULL){
        printf("Erro: Nao foi possivel importar o arquivo '%s'.\n", source);
        return NULL;
    }

    fseek(fp, 0, SEEK_END);
    int bytes = ftell(fp);
    fseek(fp, 0, SEEK_SET);

    file_chunk* fc = (file_chunk*)malloc(sizeof(file_chunk));
    fc->file = (file_t*)malloc(sizeof(file_t) * bytes);
    fc->bytes = bytes;

    fread(fc->file, sizeof(char), bytes, fp);
    fclose(fp);
    return fc;
}
```

Figura 30 – Função para importar arquivos

6.3.17 Criar diretório: file_chunk* createDirectory(char* name, int freeBlock, DirMeta* fatherMeta)

Função usada para criar um diretório. Cria os dados (meta e entries) do diretório. Limita o nome do diretório a sete caracteres. Define a quantidade de entradas do diretório (sempre que criar um diretório ele terá duas entradas "." e ".."), define a quantidade de bytes do diretório (tamanho do proprio meta + tamanho das duas entradas) e define o primeiro bloco do diretório. Nomeia as duas primeiras entradas, já explicadas, do diretório e seus respectivos blocos. Aloca um file chunk e seta seus bytes conforme o meta.bytes. Por fim cria o arquivo file_t, o aloca e cria a referência do meta e das entries. Por fim retorna o file chunk (diretorio criado). A figura 31 mostra seu conteúdo.

```
file_chunk* createDirectory(char* name, int freeBlock, DirMeta* fatherMeta){
    DirMeta meta; // cria o meta do diretorio
    Entry entries[2]; // cria o array de entries do diretorio

    //meta
    int nameSize = strlen(name)>7 ? 7:strlen(name); //verifica se name tem mais que 7 caracteres
    strncpy(meta.name, name, nameSize); //copia no maximo 7 caracteres
    meta.name[nameSize] = '\0'; //adiciona final de string ao final do nome.

    meta.entryQtde = 2;     // define a quantidade de entradas
    meta.bytes = sizeof(DirMeta) + meta.entryQtde*sizeof(Entry); // define o tamanho do meta
    meta.firstBlock = freeBlock; // define o primeiro bloco

    // .
    strcpy(entries[0].name, "."); // atual
    strcpy(entries[0].type, "dir"); // define como diretorio
    entries[0].firstBlock = meta.firstBlock;

    // ..
    strcpy(entries[1].name, ".."); // bloco do pai
    strcpy(entries[1].type, "dir"); // define como diretorio
    entries[1].firstBlock = fatherMeta->firstBlock;

    //Criando chunk com o file_t e controle de bytes
    file_chunk* fc = (file_chunk*)malloc(sizeof(file_chunk)); // cria o file_chunk
    fc->bytes = meta.bytes;

    //Criando arquivo file_t
    fc->file = (file_t*)malloc(meta.bytes*sizeof(char)); // aloca o file_t
    memcpy(fc->file, &meta, sizeof(DirMeta)); // copia o meta para o file_t
    memcpy(fc->file+sizeof(DirMeta), &entries, sizeof(Entry)*meta.entryQtde); // copia o array de entries para o file_t

    return fc;
}
```

Figura 31 – Função para criar diretório

6.3.18 Escrever bloco: void writeBlock(Superblock* sb, int blockNum, void* buffer, int bufferSize)

Função usada para escrever um bloco. É aberto o disco no modo de escrita, movido o ponteiro com fseek até o bloco selecionado e então gravado o bloco no disco usando fwrite. Feito isso é fechado o arquivo de disco. A figura 32 mostra seu conteúdo.

```c
void writeBlock(Superblock* sb, int blockNum, void* buffer, int bufferSize){
    if(blockNum >= sb->blockQtde) return;
    if(bufferSize <= 0 || bufferSize > sb->blockSize) return;

    FILE* dsc = fopen(DISCO, "r+"); // abre o disco

    //ir para o bloco selecionado
    fseek(dsc, blockNum*sb->blockSize, SEEK_SET);

    //gravar no disco
    fwrite(buffer, sizeof(char), bufferSize, dsc);
    fclose(dsc);
}
```

Figura 32 – Função para criar bloco

6.4 Parte 4: Comandos de manipulação.

6.4.1 cd: DirChunk* changeDirectory(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* name, char* pathing)

Função usada no comando “cd” para mudar de diretório (pathing) de navegação. A função percorre o diretório atual procurando um diretório com o nome mandado pelo usuário, e então cria um firstBlock com a entrie da posição encontrada, feito isso ocorre uma troca do diretório atual com o firstBlock. Se não foi encontrado retorna, caso contrário inclui o nome na variável do pathing. Dependendo da string enviada, é feito um tratamento diferente na inclusão do pathing, como por exemplo: voltar para o diretório root ou avançar para outro diretório. A figura 33 documenta linha-a-linha sua implementação.

```c
DirChunk* changeDirectory(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* name, char* pathing){
    DirChunk* novoDir;
    int notFound = 1;
    for(int i=0; i<diretorioAtual->meta.entryQtde; i++){  // Percorre o diretorio
        if(strcmp(diretorioAtual->entries[i]->name, name) == 0){ // Se encontrar o arquivo
            if(strcmp(diretorioAtual->entries[i]->type, "dir") == 0){ // Se encontrar um diretorio (tipo dir)
                int firstBlock = diretorioAtual->entries[i]->firstBlock; // pega o primeiro bloco do diretorio
                facc_unloadDirectory(diretorioAtual); // desaloca o diretorio atual
                novoDir = facc_loadDir(sb, fat, firstBlock); // carrega o diretorio
                notFound = 0;
                break;
            }
        }
    }
    if(notFound == 1){ // Se nao encontrar o diretorio
        printf("Diretorio nao encontrado\n"); // imprime mensagem de erro
        return diretorioAtual; // retorna o diretorio atual (proprio diretorio)
    }
    if(strcmp(name,"..")== 0){ // Se o diretorio for o diretorio pai
        int size = strlen(pathing); // pega o tamanho do caminho
        while(size>=0){ // percorre o caminho
            if(pathing[size] == '/'){ // se encontrar um '/'
                pathing[size] = '\0'; // corta o caminho
                break; // para de percorrer o caminho
            }
            size--;
        }
    }else if(strcmp(name, ".") == 0){ // Se o diretorio for o diretorio atual
        //faz nada
    }else{
        strcat(pathing, "/"); // concatena o caminho com "/"
        strcat(pathing, novoDir->meta.name); // concatena o caminho com o nome do diretorio
    }
    //pwd path
    return novoDir; // retorna o novo diretorio
}
```

Figura 33 – Função de cd

6.4.2 mkdir: makeDirectory(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* name)

Função usada no comando "mkdir" para criar um diretório. São percorridas as entries buscando um conflito de nome, se não houve, pega-se o primeiro bloco livre no fat para o diretório. Feito isso, é chamada a função createDirectory passando-se o bloco livre, então é escrito o bloco no disco e atualizado o fat. Por fim é criada a referência para o diretório, e criada a referência do novo diretório dentro do diretório atual. A figura 34 mostra sua implementação.

```
134 void makeDirectory(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* name){
135         //checar se ja existe diretorio com este nome
136         for(int i=0; i<diretorioAtual->meta.entryQtde; i++){ // Percorre o diretorio
137             if(strcmp(diretorioAtual->entries[i]->name, name) == 0){ // Se encontrar o arquivo
138                 printf("Erro, ja existe um diretorio com este nome\n");
139                 return;
140             }
141         }
142 
143         //Encontro um bloco livre
144         int blockNum = facc_findFreeBlock(sb, fat); // pega o primeiro bloco livre
145         if(blockNum < 0){ // Se nao encontrar bloco livre
146             printf("Erro, o disco esta cheio!\n");
147             return;
148         }
149 
150         //Crio um diretorio e guardo no disco
151         file_chunk* newDir = createDirectory(name, blockNum, &diretorioAtual->meta); // cria um diretorio
152         writeBlock(sb, blockNum, newDir->file, newDir->bytes); // escreve o diretorio no disco
153         updateFat(sb, fat, blockNum, FAT_L); // atualiza o FAT
154 
155         //criando a referencia pro diretorio
156         Entry* ref = (Entry*)malloc(sizeof(Entry)); // cria um novo entry
157 
158         int nameSize = strlen(name)>7 ? 7:strlen(name); //verifica se name tem mais que 7 caracteres
159         strncpy(ref->name, name, nameSize); //copia no maximo 7 caracteres
160         ref->name[nameSize] = '\0'; //adiciona final de string ao final do nome.
161 
162         ref->firstBlock = blockNum; // atribui o bloco do diretorio
163         strcpy(ref->type, "dir");        // atribui o tipo do diretorio
164 
165         //crio a referencia (entry) do novoDir dentro do diretorioatual
166         facc_updateDirAdd(sb, fat, diretorioAtual, ref);
167 }
```

Figura 34 – Função de mkdir

6.4.3 pwd: void showPath(char* pathing)

Função usada no comando "pwd" para mostrar o caminho atual. Apenas mostra na tela o pathing, variável que contém o caminho no qual o usuário se encontra. A figura 35 mostra seu conteúdo.

```
257 void showPath(char* pathing){
258     printf("%s\n", pathing); //mostra o path
259 }
```

Figura 35 – Função de pwd

6.4.4 rm: removeItem(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* name)

Função usada no comando "rm" para remover um diretório. Primeiramente, faz-se uma verificação para garantir que o diretório pode ser removido (não é diretório de referência, como "." e ".."). Em seguida, é feito um laço para percorrer todo o diretório, a fim de encontrar

o arquivo especificado pelo usuário, se encontrar o arquivo e ele for um diretório não vazio, é removido através da função facc_updateDirDel, caso for um arquivo, é removido sem condições. Já se não for encontrado no diretório em questão, apenas retorna. A figura 36 mostra sua implementação.

```
51  void removeItem(Superblock* sb, Fat* fat, DirChunk* diretorioAtual,  char* name){
52      if(strcmp(name, ".") == 0 || strcmp(name, "..") == 0 ){
53          printf("Nao e possivel excluir o diretorio de referencia '%s'.\n", name);
54          return;
55      }
56
57      int notFound = 1;
58      for(int i=0; i<diretorioAtual->meta.entryQtde; i++){// Percorre o diretorio
59          if(strcmp(diretorioAtual->entries[i]->name, name) == 0){// Se encontrar o arquivo
60
61              //se encontrei arquivo e eh um dir
62              if(strcmp(diretorioAtual->entries[i]->type, "dir") == 0){
63
64                  int firstBlock = diretorioAtual->entries[i]->firstBlock;// pega o primeiro bloco do diretorio
65                  DirChunk* auxDir = facc_loadDir(sb, fat, firstBlock); // carrega o diretorio
66
67                  if(auxDir->meta.entryQtde > 2){ // Se o diretorio nao esta vazio
68                      printf("Nao eh possivel deletar '%s': Diretorio nao vazio.\n", name);
69
70                  }else{ // Se o diretorio esta vazio
71                      facc_updateDirDel(sb, fat, diretorioAtual, i); // remove o diretorio do diretorio atual
72                  }
73
74                  facc_unloadDirectory(auxDir); // desaloca o diretorio auxiliar
75                  notFound = 0;
76                  break;
77
78              //se encontrei arquivo e eh type qualquer (.txt, etc...)
79              }else{
80                  facc_updateDirDel(sb, fat, diretorioAtual, i);
81                  break;
82              }
83          }
84      }
85
86      if(notFound == 1){
87          printf("Arquivo ou diretorio '%s' nao encontrado\n", name);
88          return;
89      }
90  }
```

Figura 36 – Função de rmItem

6.4.5 ls: listDirectory(DirChunk* diretorioAtual, char* name)

Função usada no comando “ls” para listar o conteúdo dentro de um diretório ou a informação de um arquivo. A figura 37 mostra sua implementação.

- **1º caso:** compara se há segundo parâmetro(exemplo: ls <dir>), se não há, é percorrido o diretório e imprimido todos os nomes presentes.
- **2º caso:** compara o último caracter, se for “*”(estrela) é percorrido todo o diretório e imprimindo todos com as inicias mandadas pelo usuário.
- **3º caso:** caso não seja nenhuma das duas opções, trata-se de um arquivo específico. Então é percorrido o diretório procurando o arquivo com o nome enviado pelo usuário.

```
196  void listDirectory(DirChunk* diretorioAtual, char* name){
197
198      if(name == NULL){   //se nao ha segundo parametro
199          for(int i=0; i<diretorioAtual->meta.entryQtde; i++){
200              if(strcmp(diretorioAtual->entries[i]->type, "dir") == 0){
201                  printf("%s\t", diretorioAtual->entries[i]->name);
202              }else{
203                  printf("%s.%s\t", diretorioAtual->entries[i]->name, diretorioAtual->entries[i]->type);
204              }
205          }
206
207      }else if(name[strlen(name)-1] == '*') {    //se o ultimo elemento for * mostra todos os arquivos com as iniciais
208          for(int i=0; i<diretorioAtual->meta.entryQtde; i++){
209              if(strncmp(diretorioAtual->entries[i]->name, name, strlen(name)-1) == 0){
210                  if(strcmp(diretorioAtual->entries[i]->type, "dir") == 0){
211                      printf("%s\t", diretorioAtual->entries[i]->name);
212                  }else{
213                      printf("%s.%s\t", diretorioAtual->entries[i]->name, diretorioAtual->entries[i]->type);
214                  }
215              }
216          }
217
218
219      }else{    //se esta procurando um arquivo especifico
220          int notFound = 1;
221
222          for(int i=0; i<diretorioAtual->meta.entryQtde; i++){
223
224              if(strcmp(diretorioAtual->entries[i]->name, name) == 0){
225                  if(strcmp(diretorioAtual->entries[i]->type, "dir") == 0){    //se encontrou o arquivo, checar se eh dir
226                      printf("%s\t", diretorioAtual->entries[i]->name);
227                  }else{    //se não eh diretorio, lista as informacoes do arquivo (nome)
228                      printf("%s.%s\t", diretorioAtual->entries[i]->name, diretorioAtual->entries[i]->type);
229                  }
230                  notFound = 0;
231              }
232          }
233
234          if(notFound == 1){
235              printf("Arquivo ou Diretorio inexistente.");
236          }
237      }
238      printf("\n");
239  }
```

Figura 37 – Função de ls

6.4.6 format dsc: format_dsc(int blockSize, int blockQtde)

Função usada no comando “format dsc” para formatar o disco que está sendo usado no momento. Utilizamos o valor 4096 como o padrão para o tamanho dos blocos, podendo ser modificada apenas a quantidade de blocos no disco, que não deve ser maior do que o sistema suporta (exemplificado no segundo if da função). Faz uso da função facc_format para fazer a formatação. A figura 38 mostra seu conteúdo.

```
434  int format_dsc(int blockSize, int blockQtde){
435      if(blockQtde < 4){ //se o numero de blocos for menor que 4, nao eh possivel formatar
436          printf("Entrada inválida, a quantidade de blocos deve possuir ao menos 4 blocos (sb, fat, root, data#1)\n");
437          return 1;
438      }
439      if((blockQtde * sizeof(Fat)) > blockSize){
440          printf("Nao eh possivel armazenar uma fat de %d posicoes em um bloco de %d bytes. (Max: %d)\n", blockQtde, blockSize, blockSize/sizeof(Fat));
441          return 1;
442      }
443
444      facc_format(blockSize, blockQtde); //formata o disco
445      return 0;
446  }
```

Figura 38 – Função de format dsc

6.4.7 mv: moveItem(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* source, char* destination)

Função usada no comando “mv” para mover um diretório ou arquivo. Primeiro é percorrido o diretório procurando a posição do arquivo a ser movido, quando achado é guardada a posição em entryFound e sourceEntry. Se entryFound não tiver a posição o código finaliza. Após isso, é percorrido o caminho especificado e na primeira ocorrência de “/” se corta o caminho. Esse passo é feito para identificar se deve ser feita a movimentação do arquivo ou a renomeação.

- **1º caso, renomeação:** Comparamos se o arquivo a ser renomeado é um diretório. Caso seja, é carregado o diretório com a função loadDir, então é mudado o nome do diretório com o nome do parâmetro e atualizado com a função updateDirectory. Caso não seja um diretório, simplesmente mudamos o nome do arquivo na posição encontrada anteriormente e atualizamos o diretório.

- **2º caso, movimentação:** Carregamos o diretório e atribuimos ao token o caminho que será movido o arquivo. Feito isso, percorremos o caminho e verificamos se o diretório existe. Caso exista armazenamos a posição em uma variável entryFound. Logo após, guardamos a posição do novo diretório em uma variável firstBlock e desalocamos o diretório atual. Carregamos então o novo diretório usando facc_loadDir e pegamos o próximo token (será pego caso haja mais caminhos a percorrer, exemplo: /documentos/so/projetofinal). Após entrarmos no diretório desejado, mudamos o nome da entry do diretório pai e atualizamos o diretório atual(seu nome e bloco). Por fim o nome da entry no diretório pai é atualizado e é criada uma referência para a nova entry no diretorio pai e atribui a entrada ao novo diretório. A figura abaixo e as figuras 40, 41 e 42 mostra sua implementação.

```
void moveItem(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* source, char* destination){
    int sourceEntry = -1;
    //checar se existe uma entrada com este nome
    int entryFound = -1;
    for(int i=0; i<diretorioAtual->meta.entryQtde; i++){ // Percorre o diretorio
        if(strcmp(diretorioAtual->entries[i]->name, source) == 0){ // Se encontrar o arquivo
            entryFound = i;
            sourceEntry = i;
            break;
        }
    }
    //se nao existir uma entrada retorna erro
    if(entryFound == -1){
        printf("Erro, nao existe um arquivo com este nome\n");
        return;
    }
    //Separar em novo nome e encontrar o proximo diretorio
    char path[250];
    char newName[8];
    //encontrar novo path
    strcpy(path, destination);
    int size = strlen(path);
    while(size>=0){ // percorre o caminho
        if(path[size] == '/'){ // se encontrar um '/'
            path[size] = '\0'; // corta o caminho
            break; // para de percorrer o caminho
        }
        size--;
    }
    if(strlen(path) == strlen(destination)){ //se nao houver nenhum '/' eh pra renomear, portanto path eh vazio
        path[0] = '\0';
    }
    int nameIndex = 0;
    for(int i=size+1; i<strlen(destination); i++){  //encontrar o nome do arquivo
        newName[nameIndex] = destination[i];
        nameIndex++;
    }
```

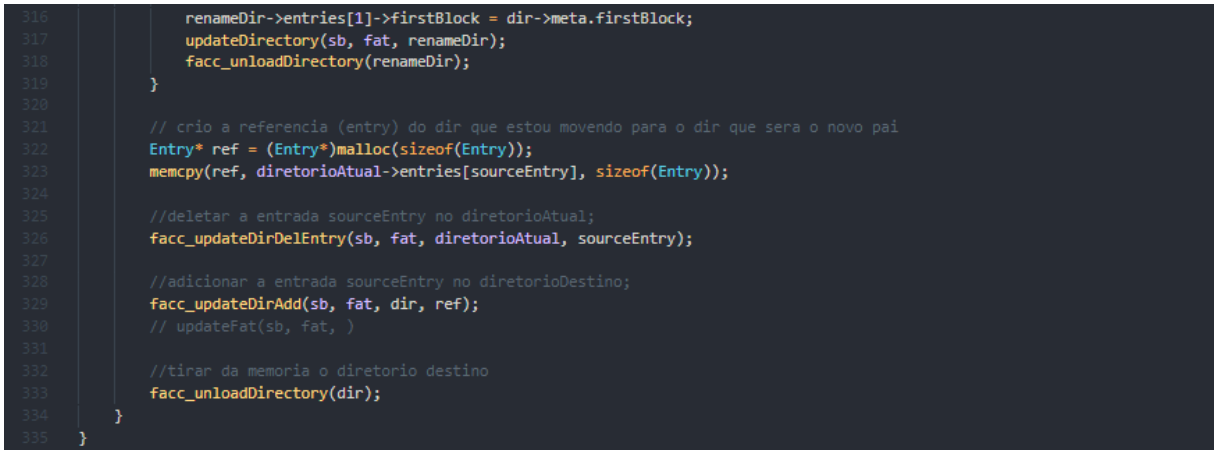
Figura 39 – Função mv

```
248     newName[nameIndex] = '\0';
249     newName[7] = '\0'; //limitando em 8 caracteres.
250
251     // >>>>>>>>>>>>>>>>>> passo 2 <<<<<<<<<<<<<<<<<<
252     // >>>>>>>>>>>>>>>>>> passo 2 <<<<<<<<<<<<<<<<<<
253     // se nao existir path, ele entra aqui e renomeia
254     if(strlen(path) < 1){
255
256         // > mudando nome no meta do arquivo
257         if(strcmp(diretorioAtual->entries[sourceEntry]->type, "dir") == 0){
258             DirChunk* renameDir = facc_loadDir(sb, fat, diretorioAtual->entries[sourceEntry]->firstBlock);
259             strcpy(renameDir->meta.name, newName);
260             updateDirectory(sb, fat, renameDir);
261             facc_unloadDirectory(renameDir);
262         }
263
264         // > mudando o nome na entry
265         strcpy(diretorioAtual->entries[sourceEntry]->name, newName);
266         updateDirectory(sb, fat, diretorioAtual);
267
268
269
270     // se existir path, ele percorre o path procurando o ultimo diretorio
271     }else{
272         DirChunk* dir = facc_loadDir(sb, fat, diretorioAtual->meta.firstBlock);
273         int firstBlock;
274
275         char* token = strtok(path, "/");
276
277         if(strcmp(token, source) == 0){
278             printf("Erro, nao eh possivel mover um arquivo para dentro dele mesmo. (%s > %s)\n", source, token);
279             return;
```

Figura 40 – Função mv pt. 2

```
280         }
281         while(token != NULL){
282             //verificar se o diretorio existe
283             entryFound = -1;
284             for(int i=0; i<dir->meta.entryQtde; i++){ // Percorre o diretorio
285                 if(strcmp(dir->entries[i]->name, token) == 0){ // Se encontrar um arquivo
286                     if(strcmp(dir->entries[i]->type, "dir") == 0){  //se o arquivo for dir
287                         entryFound = i;
288                     }
289                 }
290             }
291             if(entryFound == -1){
292                 printf("Erro, diretorio de destino '%s' nao existe.\n", token);
293                 return;
294             }
295             //se o diretorio existir:
296             firstBlock = dir->entries[entryFound]->firstBlock;
297             if(dir != diretorioAtual) facc_unloadDirectory(dir);
298             dir = facc_loadDir(sb, fat, firstBlock);
299
300             token = strtok(NULL, "/");
301         }
302         //verificar se ja existe um arquivo com o mesmo nome no diretorio destino
303         for(int i=0; i<dir->meta.entryQtde; i++){ // Percorre o diretorio
304             if(strcmp(dir->entries[i]->name, newName) == 0){ // Se encontrar um arquivo
305                 printf("Erro, ja existe um arquivo com o nome '%s' no diretorio destino '%s'\n", newName, dir->entries[i]->name);
306                 return;
307             }
308         }
309         //mudar o diretorio para o novo nome
310         // > mudando na entry diretorio pai
311         strcpy(diretorioAtual->entries[sourceEntry]->name, newName);
312         // > mudando nome no meta do arquivo
313         if(strcmp(diretorioAtual->entries[sourceEntry]->type, "dir") == 0){
314             DirChunk* renameDir = facc_loadDir(sb, fat, diretorioAtual->entries[sourceEntry]->firstBlock);
315             strcpy(renameDir->meta.name, newName);
```

Figura 41 – Função mv pt. 3



```
316             renameDir->entries[1]->firstBlock = dir->meta.firstBlock;
317             updateDirectory(sb, fat, renameDir);
318             facc_unloadDirectory(renameDir);
319         }
320
321         // crio a referencia (entry) do dir que estou movendo para o dir que sera o novo pai
322         Entry* ref = (Entry*)malloc(sizeof(Entry));
323         memcpy(ref, diretorioAtual->entries[sourceEntry], sizeof(Entry));
324
325         //deletar a entrada sourceEntry no diretorioAtual;
326         facc_updateDirDelEntry(sb, fat, diretorioAtual, sourceEntry);
327
328         //adicionar a entrada sourceEntry no diretorioDestino;
329         facc_updateDirAdd(sb, fat, dir, ref);
330         // updateFat(sb, fat, )
331
332         //tirar da memoria o diretorio destino
333         facc_unloadDirectory(dir);
334     }
335 }
```

Figura 42 – Função mv pt. 4

6.4.8 cp: copyItem(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* source, char* destination)

Função usada no comando “cp” para copiar um diretório ou arquivo. Se o arquivo for interno ao disco simulado, percorre as entradas para verificar se o parâmetro recebido consta no diretório, caso encontre atribui a posição do arquivo às variáveis entryFound e sourceFound. Caso o arquivo não exista, retorna erro.

Encontra o novo path (se não houver um path significa que deve-se renomear o arquivo), salva o nome do arquivo em newName através do path, além de também salvar a extensão do arquivo na variável fileType. Caso o path não exista, verifica se é um diretório, se for verificará se contém elementos, caso não, retorna erro, caso sim, cria um novo diretório através da função makeDiretory; se não for um diretório, acha um bloco livre na memória, e aloca um file_chunk nele, após isso cria os metadados do arquivo copiando os dados antes obtidos e atualiza o diretório adicionando o novo arquivo pela função facc_updateDirAdd.

Caso exista o path, verifica se o arquivo é externo ao disco simulado, então percorre o path para encontrar o diretório requerido. Caso o diretório exista, o carrega para a memória, senão continua no loop (para percorrer o path). Caso não exista um arquivo no destino com o mesmo nome passado por parâmetro, irá carregar o arquivo para um file_chunk, se o arquivo for carregado corretamente, ele é alocado em um bloco da memória, cria sua entrada em seu diretório pai, copiando todos os dados e utiliza a função facc_updateDirAdd para adicionar o arquivo copiado ao diretório. Então verifica se o diretório inserido é o diretório atual, caso sim o carrega na memória.

Caso seja apenas para copiar um arquivo interno, verifica se o diretório é ele mesmo (retorna se for) e percorre o path. A cada diretorio encontrado ao percorrer o path, verifica se é existente, se for o carrega na memória através da função facc_loadDir e continua para o próximo diretório do path. Ao encontrar o diretório destino, verifica se já existe um arquivo com o mesmo nome lá, se sim retorna, senão verifica se o arquivo é um diretório e não está vazio, então copia o diretório. Caso não seja um diretório, encontra um bloco livre para o arquivo, aloca um file_chunk para o mesmo e o salva. Após isso cria a referência da entrada e atualiza o diretório (através da função facc_updateDirAdd) no qual será inserido. As figuras 43, 44, 45, 46, 47, 48, 49 e 50 mostra sua

implementação.

```
void copyItem(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* source, char* destination){
    int entryFound = -1;
    int sourceEntry = 0;

    //se nao estiver buscando arquivos externos:
    if(strncmp(destination, "dsc", 3) != 0){
        //checar se existe uma entrada com este nome
        for(int i=0; i<diretorioAtual->meta.entryQtde; i++){ // Percorre o diretorio
            if(strcmp(diretorioAtual->entries[i]->name, source) == 0){ // Se encontrar o arquivo
                entryFound = i;
                sourceEntry = i;
                break;
            }
        }

        //se nao existir uma entrada retorna erro
        if(entryFound == -1){
            printf("Erro, nao existe um arquivo com este nome\n");
            return;
        }
    }

    // >>>>>>>>>>>>>>>>>> passo 1 <<<<<<<<<<<<<<<<<<
    // >>>>>>>>>>>>>>>>>> passo 1 <<<<<<<<<<<<<<<<<<
    //Separar em novo nome e encontrar o proximo diretorio
    char path[250];
    char newName[50];
    char fileType[50];

    //encontrar novo path
    strcpy(path, destination);
    int size = strlen(path);
    while(size>=0){ // percorre o caminho
        if(path[size] == '/'){ // se encontrar um '/'
            path[size] = '\0'; // corta o caminho
            break; // para de percorrer o caminho
        }
        size--;
    }
```

Figura 43 – Função copy

```
    if(strlen(path) == strlen(destination)){ //se nao houver nenhum '/' eh pra renomear, portanto path eh vazio
        path[0] = '\0';
    }

    //encontrar o nome do arquivo
    int nameIndex = 0;
    for(int i=size+1; i<strlen(destination); i++){ //percorre o caminho
        newName[nameIndex] = destination[i]; //copia o nome do arquivo
        nameIndex++; //incrementa o indice do nome
    }

    //encontrar a extensao do arquivo
    int flag = 0;
    int ext = 0;
    for(int i=0; i<strlen(newName); i++){ //percorre o nome do arquivo
        if(flag == 1){ //se flag for 1, eh pra pegar a extensao
            fileType[ext] = newName[i]; //copia a extensao do arquivo
            ext++; //incrementa o indice da extensao
        }
        if(newName[i] == '.' && i != 0){ //se encontrar um ponto e nao for o primeiro caractere
            nameIndex = i; //salva o indice do ponto
            flag = 1;
        }
    }

    fileType[ext] = '\0'; //termina a extensao do arquivo
    fileType[3] = '\0'; //termina a extensao do arquivo

    newName[nameIndex] = '\0'; //termina o nome do arquivo
    newName[7] = '\0'; //limitando em 8 caracteres.

    // >>>>>>>>>>>>>>>>>> passo 2 <<<<<<<<<<<<<<<<<<
    // >>>>>>>>>>>>>>>>>> passo 2 <<<<<<<<<<<<<<<<<<

    // se nao existir path, ele entra aqui e duplica
    if(strlen(path) < 1){

        //verificar se ja existe um arquivo com o mesmo nome no diretorio destino
```

Figura 44 – Função copy

```
530         for(int i=0; i<diretorioAtual->meta.entryQtde; i++){ // Percorre o diretorio
531             if(strcmp(diretorioAtual->entries[i]->name, newName) == 0){ // Se encontrar um arquivo
532                 printf("Erro, ja existe um arquivo com o nome '%s' no diretorio destino '%s'\n", newName, diretorioAtual->meta.name);
533                 return;
534             }
535         }
536 
537         //verificar se o arquivo que esta sendo copiado eh um diretorio e esta vazio
538         if(strcmp(diretorioAtual->entries[sourceEntry]->type, "dir") == 0){
539             DirChunk* auxdir = facc_loadDir(sb, fat, diretorioAtual->entries[sourceEntry]->firstBlock);
540 
541             //verifica se o diretorio ta vazio
542             if(auxdir->meta.entryQtde > 2){
543                 printf("Erro: Nao eh possivel copiar um diretorio nao vazio.\n");
544                 facc_unloadDirectory(auxdir);
545                 return;
546             }
547 
548             //caso nao esteja vazio
549             makeDirectory(sb, fat, diretorioAtual, newName);
550             facc_unloadDirectory(auxdir);
551 
552         //caso nao seja um diretorio
553         }else{
554             //openFile;
555             int freeBlock = facc_findFreeBlock(sb, fat);
556             file_chunk* fc = openFile(sb, fat, diretorioAtual->entries[sourceEntry]);
557             saveFile(sb, fat, fc, freeBlock);
558 
559             //createMeta
560             Entry* ref = (Entry*)malloc(sizeof(Entry));
561             strcpy(ref->name, newName);
562             strcpy(ref->type, diretorioAtual->entries[sourceEntry]->type);
563             ref->firstBlock = freeBlock;
564             ref->bytes = fc->bytes;
565             facc_updateDirAdd(sb, fat, diretorioAtual, ref);
566 
567             //atualizar o diretorio atual
```

Figura 45 – Função copy

```
568             // int firstBlock = diretorioAtual->meta.firstBlock;
569             // facc_unloadDirectory(diretorioAtual);
570             // DirChunk* dir = facc_loadDir(sb, fat, firstBlock);
571             // *diretorioAtual = *dir;
572 
573             free(fc->file);
574             free(fc);
575         }
576 
577 
578     // se existir path, ele percorre o path procurando o ultimo diretorio
579     }else{
580         DirChunk* dir = facc_loadDir(sb, fat, diretorioAtual->meta.firstBlock);
581         int firstBlock;
582 
583         char* token = strtok(path, "/");
584 
585         //verificar se eh pra trazer um arquivo externo
586         //verificar se eh pra trazer um arquivo externo
587         //verificar se eh pra trazer um arquivo externo
588         //verificar se eh pra trazer um arquivo externo
589         if(strcmp(token, "dsc") == 0){
590             //verificar searquivo existe
591             if(access(source, F_OK) != 0){
592                 printf("Erro, nao foi possivel encontrar no um arquivo '%s' no sistema real de arquivos.\n", source);
593                 return;
594             }
595 
596             //remover 'dsc'
597             token = strtok(NULL, "/");
598 
599             while(token != NULL){
600                 //verificar se o diretorio existe
601                 entryFound = -1;
602                 for(int i=0; i<dir->meta.entryQtde; i++){ // Percorre o diretorio
603                     if(strcmp(dir->entries[i]->name, token) == 0){ // Se encontrar um arquivo
604                         if(strcmp(dir->entries[i]->type, "dir") == 0){  //se o arquivo for dir
605                             entryFound = i;
```

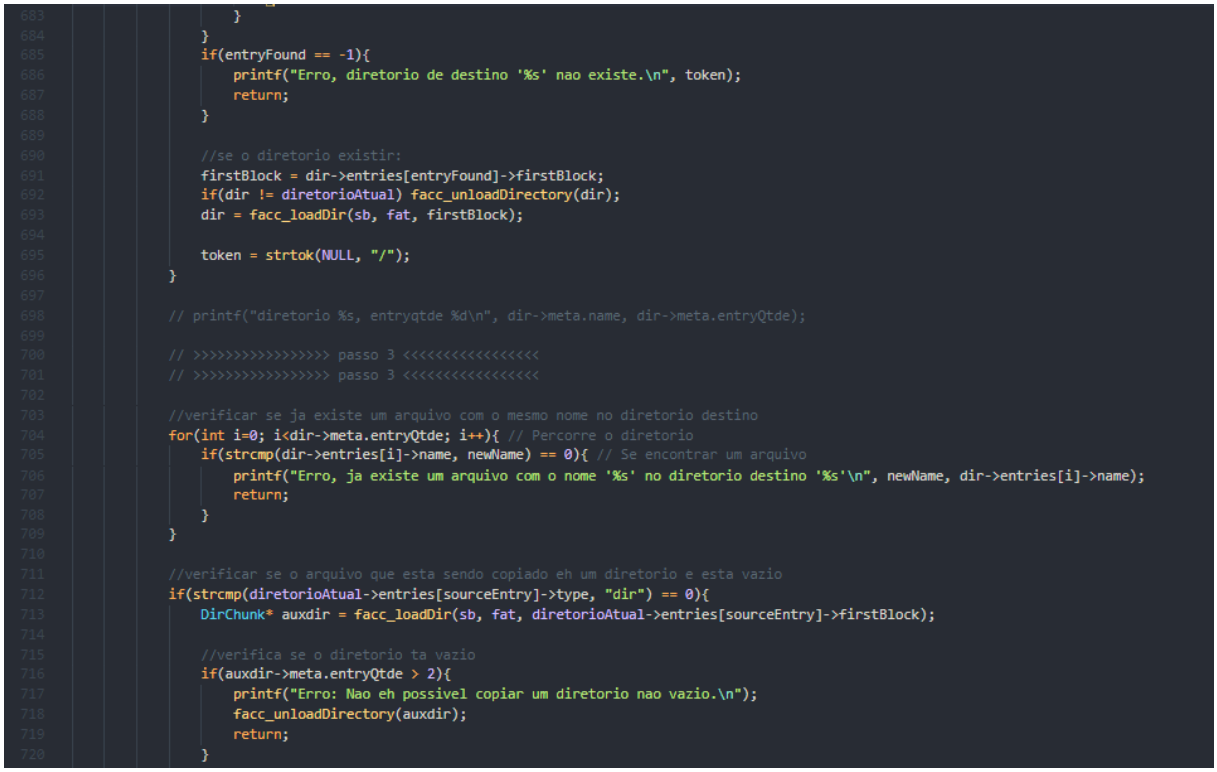
Figura 46 – Função copy

```
            }
        }
    }
    if(entryFound == -1){
        printf("Erro, diretorio de destino '%s' nao existe.\n", token);
        return;
    }

    //se o diretorio existir:
    firstBlock = dir->entries[entryFound]->firstBlock;
    if(dir != diretorioAtual) facc_unloadDirectory(dir);
    dir = facc_loadDir(sb, fat, firstBlock);

    token = strtok(NULL, "/");
}

//verificar se ja existe um arquivo com o mesmo nome no diretorio destino
for(int i=0; i<dir->meta.entryQtde; i++){ // Percorre o diretorio
    if(strcmp(dir->entries[i]->name, newName) == 0){ // Se encontrar um arquivo
        printf("Erro, ja existe um arquivo com o nome '%s' no diretorio destino '%s'\n", newName, dir->meta.name);
        return;
    }
}

//######
//carregar arquivo e fazer um file_chunk* dele.
file_chunk* fc = importFile(source);

//se deu certo
if(fc != NULL){
    //guardar arquivo em um bloco
    int freeBlock = facc_findFreeBlock(sb, fat);
    saveFile(sb, fat, fc, freeBlock);

    //criar entrada no diratual;
    Entry* ref = (Entry*)malloc(sizeof(Entry));
    strcpy(ref->name, newName);
    if(strlen(fileType) > 0) strcpy(ref->type, fileType);
    else strcpy(ref->type, "file");
```

Figura 47 – Função copy

```
            ref->bytes = fc->bytes;
            ref->firstBlock = freeBlock;

            //adicionar entrada no dir atual;
            facc_updateDirAdd(sb, fat, dir, ref);

            if(dir->meta.firstBlock == diretorioAtual->meta.firstBlock){

                int dirBlock = dir->meta.firstBlock;
                facc_unloadDirectory(diretorioAtual);
                diretorioAtual = facc_loadDir(sb, fat, dirBlock);
            }

            //del file_chunk
            facc_unloadDirectory(dir);
            free(fc->file);
            free(fc);
        }


    //caso seja apenas pra copiar um arquivo interno
    //caso seja apenas pra copiar um arquivo interno
    //caso seja apenas pra copiar um arquivo interno
    //caso seja apenas pra copiar um arquivo interno
    }else{
        if(strcmp(token, source) == 0){
            printf("Erro, nao eh possivel copiar um arquivo para dentro dele mesmo. (%s > %s)\n", source, token);
            return;
        }

        while(token != NULL){
            //verificar se o diretorio existe
            entryFound = -1;
            for(int i=0; i<dir->meta.entryQtde; i++){ // Percorre o diretorio
                if(strcmp(dir->entries[i]->name, token) == 0){ // Se encontrar um arquivo
                    if(strcmp(dir->entries[i]->type, "dir") == 0){  //se o arquivo for dir
                        entryFound = i;
                    }
```

Figura 48 – Função copy



```
            }
        }
        if(entryFound == -1){
            printf("Erro, diretorio de destino '%s' nao existe.\n", token);
            return;
        }

        //se o diretorio existir:
        firstBlock = dir->entries[entryFound]->firstBlock;
        if(dir != diretorioAtual) facc_unloadDirectory(dir);
        dir = facc_loadDir(sb, fat, firstBlock);

        token = strtok(NULL, "/");
    }

    // printf("diretorio %s, entryqtde %d\n", dir->meta.name, dir->meta.entryQtde);

    // >>>>>>>>>>>>>>>>>> passo 3 <<<<<<<<<<<<<<<<<<
    // >>>>>>>>>>>>>>>>>> passo 3 <<<<<<<<<<<<<<<<<<

    //verificar se ja existe um arquivo com o mesmo nome no diretorio destino
    for(int i=0; i<dir->meta.entryQtde; i++){ // Percorre o diretorio
        if(strcmp(dir->entries[i]->name, newName) == 0){ // Se encontrar um arquivo
            printf("Erro, ja existe um arquivo com o nome '%s' no diretorio destino '%s'\n", newName, dir->entries[i]->name);
            return;
        }
    }

    //verificar se o arquivo que esta sendo copiado eh um diretorio e esta vazio
    if(strcmp(diretorioAtual->entries[sourceEntry]->type, "dir") == 0){
        DirChunk* auxdir = facc_loadDir(sb, fat, diretorioAtual->entries[sourceEntry]->firstBlock);

        //verifica se o diretorio ta vazio
        if(auxdir->meta.entryQtde > 2){
            printf("Erro: Nao eh possivel copiar um diretorio nao vazio.\n");
            facc_unloadDirectory(auxdir);
            return;
        }
```

Figura 49 – Função copy

```
            //caso nao esteja vazio
            makeDirectory(sb, fat, dir, newName);
            facc_unloadDirectory(auxdir);

        //caso nao seja um diretorio
        }else{
            //openFile;
            int freeBlock = facc_findFreeBlock(sb, fat);
            file_chunk* fc = openFile(sb, fat, dir->entries[sourceEntry]);
            saveFile(sb, fat, fc, freeBlock);

            //createMeta
            Entry* ref = (Entry*)malloc(sizeof(Entry));
            strcpy(ref->name, newName);
            strcpy(ref->type, dir->entries[sourceEntry]->type);
            ref->firstBlock = freeBlock;
            ref->bytes = fc->bytes;
            facc_updateDirAdd(sb, fat, dir, ref);

            //tirar da memoria o diretorio destino
            facc_unloadDirectory(dir);
            free(fc->file);
            free(fc);
        }
    }
}
```

Figura 50 – Função copy

6.4.9 export: exportItem(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* source, char* destination)

Para um melhor entendimento da função cp, decidimos fazer a funcionalidade de exportar o arquivo do disco simulado para o disco real em uma função separada, chamada por linha de comando via “exp”.

Verifica se já existe um arquivo com o nome passado por parâmetro no diretório destino, se existir retorna, senão busca pelo arquivo no diretório do disco simulado através de um laço for.

Se encontrar salva a posição dentro do diretório na variável entryIndex, senão retorna.
Após isso carrega o arquivo na memória através de um openFile. Para exportar o arquivo, abre-se o diretório destino através de um fopen e escreve o arquivo pelo fwrite. A figura 51 mostra sua implementação.

```
void removeItem(Superblock* sb, Fat* fat, DirChunk* diretorioAtual,  char* name){
    if(strcmp(name, ".") == 0 || strcmp(name, "..") == 0 ){
        printf("Nao e possivel excluir o diretorio de referencia '%s'.\n", name);
        return;
    }

    int notFound = 1;
    for(int i=0; i<diretorioAtual->meta.entryQtde; i++){// Percorre o diretorio
        if(strcmp(diretorioAtual->entries[i]->name, name) == 0){// Se encontrar o arquivo

            //se encontrei arquivo e eh um dir
            if(strcmp(diretorioAtual->entries[i]->type, "dir") == 0){

                int firstBlock = diretorioAtual->entries[i]->firstBlock;// pega o primeiro bloco do diretorio
                DirChunk* auxDir = facc_loadDir(sb, fat, firstBlock); // carrega o diretorio

                if(auxDir->meta.entryQtde > 2){ // Se o diretorio nao esta vazio
                    printf("Nao eh possivel deletar '%s': Diretorio nao vazio.\n", name);

                }else{ // Se o diretorio esta vazio
                    facc_updateDirDel(sb, fat, diretorioAtual, i); // remove o diretorio do diretorio atual
                }

                facc_unloadDirectory(auxDir); // desaloca o diretorio auxiliar
                notFound = 0;
                break;

            //se encontrei arquivo e eh type qualquer (.txt, etc...)
            }else{
                facc_updateDirDel(sb, fat, diretorioAtual, i);
                break;
            }
        }
    }

    if(notFound == 1){
        printf("Arquivo ou diretorio '%s' nao encontrado\n", name);
        return;
    }
}
```

Figura 51 – Função de rmItem

6.5 Parte 4.1: Comandos adicionais de manipulação.

6.5.1 void help()

Função que imprime o catálogo de comandos. Sua implementação é apenas vários printf's, especificando o que cada comando faz e como usá-lo. A figura 52 mostra seu conteúdo.

```c
void help(){ // Função que imprime a ajuda
    printf("Descricao dos comandos:\n\n");

    printf("cd - Acessa um diretorio\n");
    printf("\tcd <dir>\n\n");

    printf("mkdir - Cria um diretorio\n");
    printf("\tmkdir <dir>\n\n");

    printf("rm - Remove um diretorio/arquivo\n");
    printf("\trm <dir | arquivo>\n\n");

    printf("cp - Copia um diretorio/arquivo\n");
    printf("\tcp <dir | arquivo>\n\tcp /caminho/nome\n\n");

    printf("mv - Move Diretório/Arquivo.\n");
    printf("\tmv <dir | arquivo>\n\n");

    printf("ls - Lista o conteudo de um diretorio ou mostra as informacoes ddo arquivo\n");
    printf("\tls <dir | arquivo>\n\n");

    printf("pwd - Mostra o caminho do diretorio atual\n");
    printf("\tpwd\n\n");

    printf("format dsc - Formata o disco simulado\n");
    printf("\tformat dsc <num blocks>\n\n");

    printf("exp - Exporta um arquivo para o diretorio do programa.\n");
    printf("\texp <arquivo> <novo nome>\n\n");

    printf("print fat - Imprime a fat\n");
    printf("\tprint fat\n\n");
}
```

Figura 52 – Função de ajuda

6.5.2 void printFat(Superblock* sb, Fat* fat)

Função que imprime o o vetor de informações das posições dos blocos, ou seja, o FAT. Sua implementação trata-se de um laço for que percorre todo o vetor do FAT verificando e mostrando na tela a situação do fat em cada posição. A figura 54 mostra seu conteúdo.

```c
void printFat(Superblock* sb, Fat* fat){
    printf("FAT => ");
    for(int i = 0; i < sb->blockQtde; i++){
        if(fat[i] == FAT_L)         printf("L");
        else if(fat[i] == FAT_F)    printf("F");
        else if(fat[i] == FAT_R)    printf("R");
        else if(fat[i] == FAT_B)    printf("B");
        else                        printf("%d", fat[i]);
        if(i != sb->blockQtde-1)    printf(", ");
    }
    printf("\n");
}
```

Figura 53 – Função para mostrar a FAT

7 RESULTADOS

Nesta seção teremos as saídas obtidas ao executar o programa, bem como o conteúdo encontrado na main, e os testes para verificação do funcionamento das funções principais do simulador. Primeiramente, é consultada a situação do disco e mostrada ao usuário, então o mesmo decide entre formatar ou manter o disco. A figura 54 mostra como isso é feito.

```
20  //arquivo existe
21  if(access(DISCO, F_OK) == 0){                // Verifica se o arquivo existe
22      while(1){                                // Enquanto o usuário não digitar "exit"
23          printf("Já existe um disco, deseja formata-lo? [y, n]\n"); // Pergunta se deseja formatar o disco
24          fgets(command, sizeof(command), stdin);          // Recebe a resposta do usuário
25
26          if(strncmp(command, "y", 1) == 0){            // Se a resposta for "y"
27              //pedir blocksize e blockqtde
28              printf("Quantidade de blocos: ");         // Pergunta quantos blocos o disco deve ter
29              fgets(command, sizeof(command), stdin);   // Recebe a resposta do usuário
30              int x = format_dsc(BLOCKSIZE, atoi(command));          // Chama a função que formata o disco
31              if(x != 1){
32                  printf("Disco criado.\n");            // Informa que o disco foi criado
33                  break;
34              }                                // Sai do loop
35
36          }else if(strncmp(command, "n", 1) == 0) {    // Se a resposta for "n"
37              printf("Disco carregado com sucesso!\n"); // Informa que o disco foi carregado
38              break;
39
40          }else{
41              printf("Opcao invalida.\n");
42          }
43      }
44
45  //arquivo nao existe
46  }else{
47      format_dsc(BLOCKSIZE, BLOCKQTDE);          // Chama a função que cria o disco
48      printf("Disco gerado com %d blocos de %d bytes.\nUtilize o comando 'format dsc <qtdeBlocos>' para formatar\n", BLOCKQTDE, BLOCKSIZE);   // Informa que o disco foi criado
49  }
```

Figura 54 – Verificação da situação do disco.

Após isso, o superbloco, a fat e o diretório raiz são carregados na memória (para possibilitar suas manipulações), como visto na figura 55.

```
49      //load superblock
50      Superblock* sb = facc_loadSuperblock();         // Carrega o superblock na memória
51
52      //load fat
53      Fat* fat = facc_loadFat(sb);                    // Carrega o fat na memória
54
55      // READ_ROOT
56      DirChunk* diretorioAtual = facc_loadDir(sb, fat, sb->rootPos); // Carrega o diretório raiz na memória
```

Figura 55 – Carregando superbloco, fat e root na memória.

E então, tem-se um while(true) que simula o funcionamento de um shell que consegue interpretar o comando, passado pelo usuário, pela função listenCommand ou pelo strncmp (no caso do exit), para finalizar a execução do programa. A figura 56 ilustra o que foi relatado.

```
59      //SHELL
60      while(1){
61          printf("(%s)$ > ", pathing);           // Imprime o caminho atual
62          fgets(command, sizeof(command), stdin); // Recebe a entrada do usuário
63          if(strncmp(command, "exit", 4) == 0) break; // Se a entrada for "exit"
64          listenCommand(&sb, &fat, &diretorioAtual, command, pathing); // Chama a função que escuta a entrada do usuário
65      }
```

Figura 56 – Simulador de shell.

7.1 Resultados em execução

Temos agora as entradas de comandos e saídas obtidas ao executar cada comando.

7.1.1 ls, mkdir, cd, rm e pwd.

Exemplo usando ls, mkdir, cd, rm e pwd na figura 57.

Figura 57 – Saídas de: ls, mkdir, cd, rm e pwd.

7.1.2 ls, cp e cd.

Exemplo usando ls, cp e cd na figura 58.

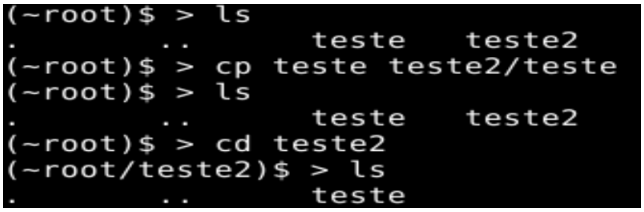


Figura 58 – Saídas de: ls, cp e cd.

7.1.3 mv, ls e cd.

Exemplo usando mv, ls e cd na figura 59.

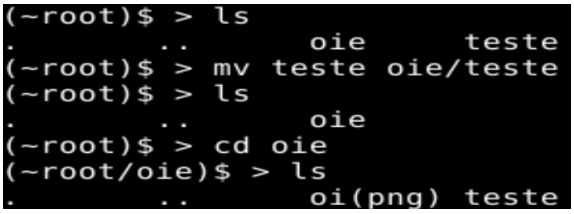


Figura 59 – Saídas de: mv, ls e cd.

7.1.4 ls e exp.

Exemplo usando ls e exp na figura 60 e o resultado na pasta na figura 61.

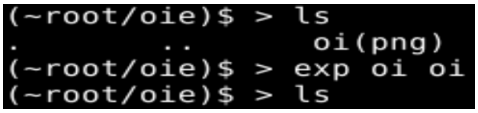


Figura 60 – Saídas de: ls e exp.

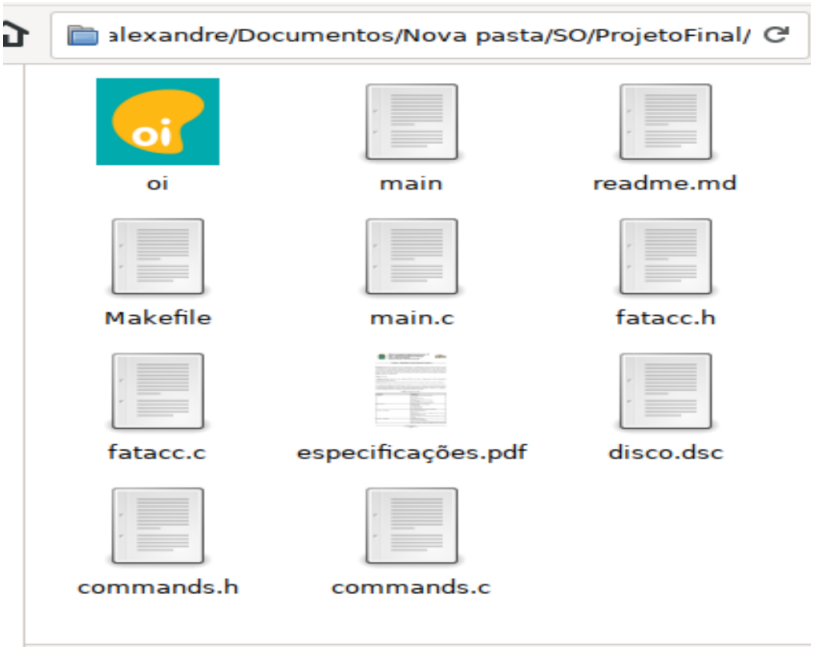


Figura 61 – Saídas de: ls, mkdir, cd, rm e pwd.

8 CONCLUSÃO

Nessa atividade nós realizamos a implentação do sistemas de arquivos, e com isso podemos verificar como o shell é realizado na prática, como o manejamento se comporta em código, e qual a complexidade de um sistema de arquivos. Desse modo, nos familiarizamos com os sistemas de arquivos, podendo então fazer melhor uso do mesmo ou até modificá-lo de uma forma mais específica que atenda nosso interesses.

9 REFERÊNCIAS

[1] MAZIERO, C. Sistemas Operacionais: Conceitos e Mecanismos. Editora da UFPR, 2019. 456 p. ISBN 978-85-7335-340-2.

[2] Slides do professor.

Documentação linguagem C https://www.cplusplus.com/reference/clibrary/