



**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 MODULARIZAÇÃO

O fattac 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.

## 4 MÉTODOS

### 4.1 Parte 1: Organização geral do sistema de arquivos.

#### 4.1.1 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 1.

```
20 typedef struct{
21     char tipo[8]; //8b
22     //fat
23     int fatPos; //4b
24     //diretorio raiz
25     int rootPos; //4b
26     //blocos
27     int blockPos; //4b
28     int blockQtde; //4b
29     int blockSize; //4b
30 } Superblock; //28bytes
```

Figura 1 – Struct Superblock

### 4.1.2 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 2, enquanto seu conteúdo pode ser observado na 2ª figura 3.

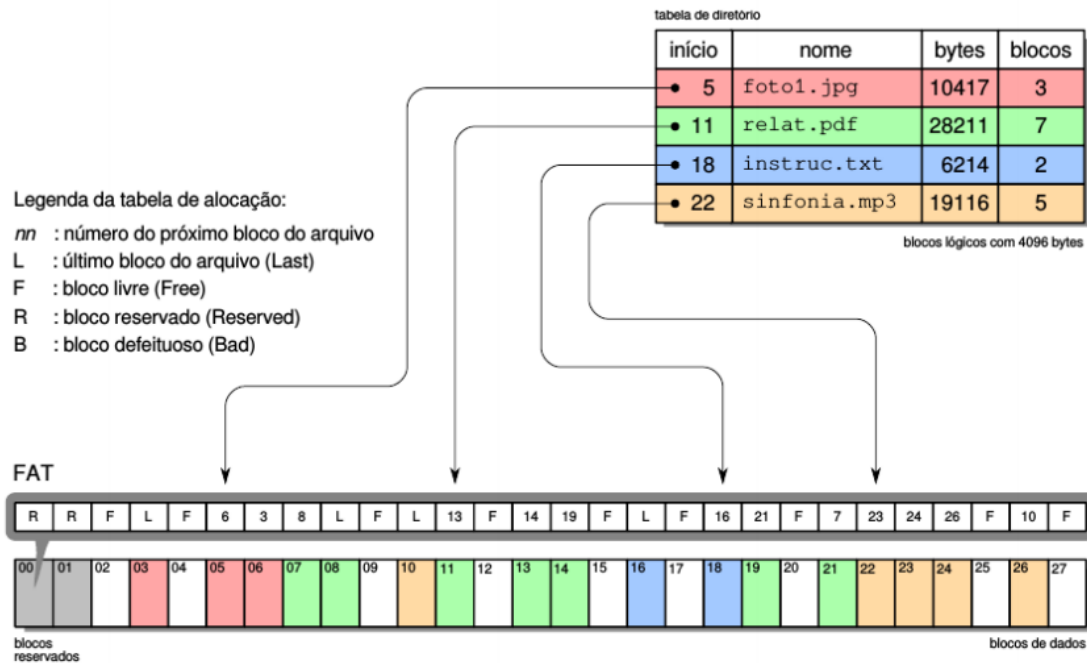


Figura 2 – Funcionamento do FAT

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

Figura 3 – Estrutura do FAT

### 4.1.3 File chunk

O file chunk foi criado para possibilitar o gerenciamento 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 4.

```

40 // Blocos: Arquivos
41 typedef char file_t;
42 caiomiglioli, a day ago | 1 author (caiomiglioli)
43 typedef struct{
44     file_t* file;
45     int bytes;
46 } file_chunk;

```

Figura 4 – Struct file chunk

#### 4.1.4 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 5 pode-se observar sua estrutura.

```

49 typedef struct{
50     char name[8]; //8b
51     char type[4]; //4b extensao (DIR se for diretorio)
52     int bytes; //4b tamanho do arquivo
53     int firstBlock; //4b primeiro bloco onde se encontra o arquivo
54     char reserved[12]; //12b para completar 32b e ser potencia de 2
55 } Entry; //32bytes

```

Figura 5 – Struct Entry

#### 4.1.5 Meta

A struct DirMeta estará presente em todos os diretórios do disco, é ela que guarda todas as informações pertinentes para as manipulações dos diretórios. A figura 6 mostra o conteúdo da estrutura.

```

57 typedef struct{
58     char name[8]; //8b
59     int entryQtde; //4b
60     int bytes; //4b tamanho do arquivo
61     int firstBlock; //4b primeiro bloco onde se encontra o arquivo
62     char reserved[12]; //12b
63 } DirMeta; //32bytes, deve ser igual ou maior que Entry

```

Figura 6 – Struct DirMeta

#### 4.1.6 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 7 mostra seu conteúdo.

```

65 typedef struct{
66     DirMeta meta;
67     Entry** entries;
68 } DirChunk;

```

Figura 7 – struct DirChunk

## 4.2 Parte 2: Funções auxiliares.

### 4.2.1 `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 8 mostra o código, bem como o explica linha-a-linha.

```
160 char** split(char* command){
161
162     command = strtok(command, "\\n"); //remove o \\n do final da string
163
164     char** result = 0; //ponteiro para o resultado (string como array de char)
165     size_t count = 0; //contador de palavras
166     char* tmp = command; //ponteiro temporario para percorrer a string
167     char* last_comma = 0;
168     char delimiter = ' ', delim[2]; //delimitador
169     delim[0] = delimiter, delim[1] = 0;
170
171     while (*tmp){ //enquanto nao chegar no fim da string
172         if (delimiter == *tmp){ //se encontrar um delimitador
173             count++; //incrementa o contador
174             last_comma = tmp; //salva o ponteiro para o ultimo espaço encontrado
175         }
176         tmp++; //incrementa o ponteiro
177     }
178     count += last_comma < (command + strlen(command) - 1); //se o ultimo espaço nao for
179     // o fim da string, incrementa o contador
180     count++;
181     result = malloc(sizeof(char*) * count); //aloca o tamanho do contador
182     if (result){ //se a alocação foi bem sucedida
183         size_t idx = 0; //índice do resultado
184         char* token = strtok(command, delim); //pega a primeira palavra
185
186         while (token){ //enquanto nao chegar no fim da string
187             assert(idx < count); //se o índice for maior que o contador, algo de errado aconteceu
188             *(result + idx++) = strdup(token); //copia a palavra para o resultado
189             token = strtok(0, delim); //pega a proxima palavra
190         }
191         assert(idx == count - 1); //se o índice for diferente do contador, algo de errado aconteceu
192         *(result + idx) = 0; //faz o fim da string
193     }
194     return result; //retorna o resultado
195 }
```

Figura 8 – Função split

### 4.2.2 `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 9 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 9 – Função de criação do pathing

#### 4.2.3 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 requerida (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 10 e 11 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 10 – Função de escuta

```

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

```

Figura 11 – Função de escuta

### 4.3 Parte 3: Funções principais.

#### 4.3.1 Carregar Superbloco: Superblock\* facc\_loadSuperblock()

Função usada para criar/formatar/carregar o superbloco. É alocado o superbloco e atribuído 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 12 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 12 – Função para carregar superbloco

#### 4.3.2 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 13 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); // lê 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 13 – Função para carregar o fat

#### 4.3.3 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 diretório do disco
99     DirChunk* directory = (DirChunk*)malloc(sizeof(DirChunk)); // cria o diretório
100
101     //abrir arquivo
102     file_chunk* fc = opendir(sb, fat, firstBlock); // abre o diretório
103     if(fc == NULL){ // se não conseguiu abrir
104         printf("Erro ao abrir diretório\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 diretório 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 diretório
123 }
```

Figura 14 – Função para carregar o diretório



#### 4.3.4 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 15 mostra seu conteúdo.

```
126 void facc_unloadDirectory(DirChunk* dir){ // desaloca o diretório
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 diretório
132 }
```

Figura 15 – Função para desalocar o diretório

#### 4.3.5 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 16 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 posição
138         }
139     }
140     return -1; // não achou
141 }
```

Figura 16 – Função para encontrar bloco livre

#### 4.3.6 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 17 e 18 mostra seu conteúdo.

```

21 // Formata o disco, deixando-o com um superbloco, fat e um diretorio root
22 void facc_format(int blockSize, int blockQtde){
23     /* criar disco */
24     FILE* dsc = fopen(DISCO, "w"); // cria o disco
25     char* tmp2 = (char*)calloc(blockSize, sizeof(char)); // cria um buffer temporario
26     for(int i=0; i<blockQtde; i++) fwrite(tmp2, 1, blockSize, dsc); // escreve o buffer no disco
27     free(tmp2); // libera o buffer
28     fclose(dsc); // fecha o disco
29
30     /* criar superbloco */
31     Superblock sb;
32     strcpy(sb.tipo, "fatacc"); // tipo do superbloco
33
34     sb.fatPos = 1; // posicao do fat
35     sb.rootPos = 2; // posicao do diretorio root
36     sb.blockPos = 3; // posicao do primeiro bloco de dados
37     sb.blockSize = blockSize; // tamanho do bloco
38     sb.blockQtde = blockQtde; // quantidade de blocos
39
40     writeBlock(&sb, 0, &sb, sizeof(Superblock)); // escreve o superbloco no disco
41     /* fat */
42     Fat* fat = (Fat*)malloc(sizeof(Fat) * sb.blockQtde); // cria o fat
43
44     for(int i=0; i<3; i++){ // preenche o fat
45         fat[i] = FAT_R; //reservados
46     }
47     for(int i=3; i<blockQtde; i++){
48         fat[i] = FAT_F; //livres
49     }

```

Figura 17 – Função de formatar disco

```

50     writeBlock(&sb, sb.fatPos, fat, sizeof(Fat)*sb.blockQtde); // escreve o fat no disco
51     /* dirRoot */
52     //cria o que seria o meta do diretorio pai para inserir na entry ".."
53     DirMeta fatherMeta;
54     fatherMeta.firstBlock = sb.rootPos; // posicao do primeiro bloco do diretorio pai
55
56     // 1 - encontrar posicao
57     // -> por ser root, usara a posicao superbloco.rootPos
58
59     // 2 - criar diretorio filho e guardar no disco
60     file_chunk* file = createDirectory("root", sb.rootPos, &fatherMeta); // cria o diretorio root
61     writeBlock(&sb, sb.rootPos, file->file, file->bytes); // escreve o diretorio root no disco
62     updateFat(&sb, fat, sb.rootPos, FAT_L); // atualiza o fat
63
64     // 3 - criar entry no diretorio pai e guardar no disco
65     // -> nao ha diretorio pai pois este eh o root
66
67     /* free */
68     free(fat); // libera o fat
69     free(file->file); // libera o file_chunk
70     free(file); // libera o file_chunk
71 }

```

Figura 18 – Função de formatar disco

#### 4.3.7 Atualizar diretório(adição): void facc\_updateDirAdd(Superblock\* sb, Fat\* fat, DirChunk\* diretorioAtual, Entry\* ref)

Função usada para atualizar o diretório 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 19 mostra seu conteúdo.

```
143 void facc_updateDirAdd(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, Entry* ref){ // atualiza o diretorio apos adicionar um novo entry
144     //criando vetor de entries
145     Entry** newEntries = (Entry**)malloc(sizeof(Entry*) * (diretorioAtual->meta.entryQtde+1)); // aloca o vetor de entries
146
147     //adiciono as entradas do dir->entries para o newEntries
148     int i = 0;
149     while(i<diretorioAtual->meta.entryQtde){ // para cada entry
150         newEntries[i] = diretorioAtual->entries[i]; // copia o entry
151         i++; // incrementa
152     }
153
154     //adiciono a nova entrada no newEntries
155     newEntries[i] = ref;
156
157     //coloco o NewEntries no lugar do antigo entries
158     free(diretorioAtual->entries);
159     diretorioAtual->entries = newEntries;
160
161     //coloco no meta que foi adicionado mais um diretorio
162     diretorioAtual->meta.entryQtde++;
163
164     updateDirectory(sb, fat, diretorioAtual); // atualiza o diretorio
165 }
```

Figura 19 – Função para adicionar uma entry

#### 4.3.8 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 diminuída a quantidade de entries. Por fim, se atualiza o diretório com o diretório atual alterado.

A figura 20 mostra seu conteúdo.

```
168 void facc_updateDirDel(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, int index){ // atualiza o diretorio apos deletar um entry
169     if(index < 2 || index > diretorioAtual->meta.entryQtde){ // se o index for invalido
170         printf("Erro ao remover arquivo.\n"); // erro
171         return;
172     }
173
174     //excluindo o arquivo.
175     deleteFile(sb, fat, diretorioAtual->entries[index]->firstBlock); // exclui o arquivo
176
177     //removendo a entrada.
178     free(diretorioAtual->entries[index]); // libera o entry
179     int ultimoEntry = diretorioAtual->meta.entryQtde - 1; // pega o ultimo entry
180
181     diretorioAtual->entries[index] = diretorioAtual->entries[ultimoEntry]; // copia o ultimo entry para o index
182     diretorioAtual->entries[ultimoEntry] = NULL; // libera o ultimo entry
183
184     diretorioAtual->meta.entryQtde--; // diminui o entryQtde
185
186     updateDirectory(sb, fat, diretorioAtual); // atualiza o diretorio
187 }
188 }
```

Figura 20 – Função para remover uma entry

#### 4.3.9 Atualizar diretório: void updateDirectory(Superblock\* sb, Fat\* fat, DirChunk\* diretorio)

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 21 mostra seu conteúdo.

```
197 void updateDirectory(Superblock* sb, Fat* fat, DirChunk* directory){ // atualiza o diretório
198     //remove o antigo da memória
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 diretório 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 21 – Função para atualizar o diretório

#### 4.3.10 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 22 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 memória
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 memória
226
227         // Faço 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         //faço 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 22 – Função para salvar o file chunk

#### 4.3.11 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 23 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 23 – Função para salvar o file chunk

#### 4.3.12 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 24 mostra seu conteúdo.

```

260 file_chunk* openDir(Superblock* sb, Fat* fat, int firstBlock){
261     if(firstBlock < 0 || firstBlock >= sb->blockQtde) return NULL; // se o primeiro bloco for invalido
262
263     int blocks_qtde = 0;
264     int block = firstBlock;
265
266     //guarda em blocks_qtde quantos blocos o arquivo utiliza
267     do{ // enquanto o bloco nao for o ultimo
268         if(fat[block] == FAT_B || fat[block] == FAT_F) return NULL; // erro
269         blocks_qtde++; // incrementa o contador
270         block = fat[block]; // pega o proximo bloco
271     }while(block != FAT_L);
272
273     //cria o file_chunk com a stream do arquivo e seus bytes para retornar
274     file_chunk* fc = (file_chunk*)malloc(sizeof(file_chunk)); // cria o file_chunk
275     fc->bytes = blocks_qtde * sb->blockSize; // define o tamanho do file_chunk
276     fc->file = (file_t*)malloc(fc->bytes); // aloca o file_t
277
278     file_t* tmp; // auxiliar
279     int i = 0;
280     block = firstBlock; // primeiro bloco
281
282     //faz a leitura de cada bloco e adiciona na stream "file_t fc->file"
283     do{ // enquanto o bloco nao for o ultimo
284         if(fat[block] == FAT_B || fat[block] == FAT_F) return NULL; // erro
285
286         tmp = readBlock(sb, block, sb->blockSize); // le o bloco
287         memcpy(fc->file + (i*sb->blockSize), tmp, sb->blockSize); // copia o bloco para o file_t
288         free(tmp); // libera o bloco temporario
289
290         i++; // incrementa o contador
291         block = fat[block]; // pega o proximo bloco
292     }while(block != FAT_L);
293
294     //retorno o file_chunk
295     return fc;
296 }

```

Figura 24 – Função para abrir diretório

#### 4.3.13 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 25 mostra seu conteúdo.

```
416 file_chunk* openFile(Superblock* sb, Fat* fat, Entry* meta){
417     if(meta->firstBlock < 0 || meta->firstBlock >= sb->blockQtde) return NULL; // se o primeiro bloco for invalido
418
419     //cria o file_chunk com a stream do arquivo e seus bytes para retornar
420     file_chunk* fc = (file_chunk*)malloc(sizeof(file_chunk)); // cria o file_chunk
421     fc->bytes = meta->bytes; // define o tamanho do file_chunk
422     fc->file = (file_t*)malloc(fc->bytes); // aloca o file_t
423
424     file_t* tmp; // auxiliar
425     int i = 0;
426     int block = meta->firstBlock;
427     int bytes = meta->bytes;
428
429     //faz a leitura de cada bloco e adiciona na stream "file_t fc->file"
430     do{ // enquanto o bloco nao for o ultimo
431         if(fat[block] == FAT_B || fat[block] == FAT_F) return NULL; // erro
432
433         int readingBytes = (bytes > sb->blockSize) ? sb->blockSize : bytes;
434         tmp = readBlock(sb, block, readingBytes); // le o bloco
435         memcpy(fc->file + (i*sb->blockSize), tmp, readingBytes); // copia o bloco para o file_t
436         free(tmp); // libera o bloco temporario
437
438         i++; // incrementa o contador
439         block = fat[block]; // pega o proximo bloco
440         bytes -= readingBytes;
441     }while(block != FAT_L);
442
443     if(bytes != 0){
444         printf("Warning: O arquivo foi lido com mais ou menos bytes que o devido.\n");
445     }
446
447     //retorno o file_chunk
448     return fc;
449 }
```

Figura 25 – Função para abrir arquivo

#### 4.3.14 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 26 mostra seu conteúdo.

```
299 void updateFat(Superblock* sb, Fat* fat, int pos, int nextPos){
300     if(pos < 0 || pos >= sb->blockQtde) return; // se o primeiro bloco for invalido
301     if(nextPos == FAT_R || nextPos == FAT_B) return; // se o proximo bloco for invalido
302     if(pos == nextPos) return; // se o proximo bloco for o mesmo que o atual
303
304     fat[pos] = nextPos;
305     writeBlock(sb, sb->fatPos, fat, sizeof(Fat)*sb->blockQtde); // salva o fat no disco
306 }
```

Figura 26 – Função para atualizar a fat

#### 4.3.15 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 próprio 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 27 mostra seu conteúdo.

```
308 file_chunk* createDirectory(char* name, int freeBlock, DirMeta* fatherMeta){
309     DirMeta meta; // cria o meta do diretorio
310     Entry entries[2]; // cria o array de entries do diretorio
311
312     //meta
313     int nameSize = strlen(name)>7 ? 7:strlen(name); //verifica se name tem mais que 7 caracteres
314     strncpy(meta.name, name, nameSize); //copia no maximo 7 caracteres
315     meta.name[nameSize] = '\0'; //adiciona final de string ao final do nome.
316
317     meta.entryQtde = 2; // define a quantidade de entradas
318     meta.bytes = sizeof(DirMeta) + meta.entryQtde*sizeof(Entry); // define o tamanho do meta
319     meta.firstBlock = freeBlock; // define o primeiro bloco
320
321     // .
322     strcpy(entries[0].name, "."); // atual
323     strcpy(entries[0].type, "dir"); // define como diretorio
324     entries[0].firstBlock = meta.firstBlock;
325
326     // ..
327     strcpy(entries[1].name, ".."); // bloco do pai
328     strcpy(entries[1].type, "dir"); // define como diretorio
329     entries[1].firstBlock = fatherMeta->firstBlock;
330
331     //Criando chunk com o file_t e controle de bytes
332     file_chunk* fc = (file_chunk*)malloc(sizeof(file_chunk)); // cria o file_chunk
333     fc->bytes = meta.bytes;
334
335     //Criando arquivo file_t
336     fc->file = (file_t*)malloc(meta.bytes*sizeof(char)); // aloca o file_t
337     memcpy(fc->file, &meta, sizeof(DirMeta)); // copia o meta para o file_t
338     memcpy(fc->file+sizeof(DirMeta), &entries, sizeof(Entry)*meta.entryQtde); // copia o array de entries para o file_t
339
340     return fc;
341 }
```

Figura 27 – Função para criar diretório

#### 4.3.16 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 28 mostra seu conteúdo.

```

554 | file_chunk* importFile(char* source){
555 |     FILE* fp = fopen(source, "rb");
556 |     if(fp == NULL){
557 |         printf("Erro: Nao foi possivel importar o arquivo '%s'.\n", source);
558 |         return NULL;
559 |     }
560 |
561 |     fseek(fp, 0, SEEK_END);
562 |     int bytes = ftell(fp);
563 |     fseek(fp, 0, SEEK_SET);
564 |
565 |     file_chunk* fc = (file_chunk*)malloc(sizeof(file_chunk));
566 |     fc->file = (file_t*)malloc(sizeof(file_t) * bytes);
567 |     fc->bytes = bytes;
568 |
569 |     fread(fc->file, sizeof(char), bytes, fp);
570 |     fclose(fp);
571 |     return fc;
572 | }

```

Figura 28 – Função para importar arquivos

#### 4.3.17 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 29 mostra seu conteúdo.

```

344 | void writeBlock(Superblock* sb, int blockNum, void* buffer, int bufferSize){
345 |     if(blockNum >= sb->blockQtde) return;
346 |     if(bufferSize <= 0 || bufferSize > sb->blockSize) return;
347 |
348 |     FILE* dsc = fopen(DISCO, "r+"); // abre o disco
349 |
350 |     //ir para o bloco selecionado
351 |     fseek(dsc, blockNum*sb->blockSize, SEEK_SET);
352 |
353 |     //gravar no disco
354 |     fwrite(buffer, sizeof(char), bufferSize, dsc);
355 |     fclose(dsc);
356 | }

```

Figura 29 – Função para criar bloco

#### 4.3.18 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 30 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 30 – Função para ler bloco

#### 4.4 Parte 4: Comandos de manipulação.

##### 4.4.1 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 31 documenta linha-a-linha sua implementação.

```

117 DirChunk* changeDirectory(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* name, char* pathing){
118     DirChunk* novoDir;
119     int notFound = 1;
120     for(int i=0; i<diretorioAtual->meta.entryQtde; i++){ // Percorre o diretorio
121         if(strcmp(diretorioAtual->entries[i]->name, name) == 0){ // Se encontrar o arquivo
122             if(strcmp(diretorioAtual->entries[i]->type, "dir") == 0){ // Se encontrar um diretorio (tipo dir)
123                 int firstBlock = diretorioAtual->entries[i]->firstBlock; // pega o primeiro bloco do diretorio
124                 facc_unloadDirectory(diretorioAtual); // desaloca o diretorio atual
125                 novoDir = facc_loadDir(sb, fat, firstBlock); // carrega o diretorio
126                 notFound = 0;
127                 break;
128             }
129         }
130     }
131     if(notFound == 1){ // Se nao encontrar o diretorio
132         printf("Diretorio nao encontrado\n"); // imprime mensagem de erro
133         return diretorioAtual; // retorna o diretorio atual (proprio diretorio)
134     }
135     if(strcmp(name, "..") == 0){ // Se o diretorio for o diretorio pai
136         int size = strlen(pathing); // pega o tamanho do caminho
137         while(size>0){ // percorre o caminho
138             if(pathing[size] == '/'){ // se encontrar um '/'
139                 pathing[size] = '\0'; // corta o caminho
140                 break; // para de percorrer o caminho
141             }
142             size--;
143         }
144     }else if(strcmp(name, ".") == 0){ // Se o diretorio for o diretorio atual
145         //faz nada
146     }else{
147         strcat(pathing, "/"); // concatena o caminho com "/"
148         strcat(pathing, novoDir->meta.name); // concatena o caminho com o nome do diretorio
149     }
150     //pwd path
151     return novoDir; // retorna o novo diretorio
152 }

```

Figura 31 – Função de cd

#### 4.4.2 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 32 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 32 – Função de mkdir

#### 4.4.3 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 33 mostra seu conteúdo.

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

Figura 33 – Função de pwd

#### 4.4.4 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 48 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                 }else{ // Se o diretorio esta vazio
70                     facc_updateDirDel(sb, fat, diretorioAtual, i); // remove o diretorio do diretorio atual
71                 }
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 34 – Função de `rmItem`

#### 4.4.5 `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.

- **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 iniciais 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.

. A figura 35 mostra sua implementação.

```

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     }else{ //se esta procurando um arquivo especifico
219         int notFound = 1;
220
221         for(int i=0; i<diretorioAtual->meta.entryQtde; i++){
222
223             if(strcmp(diretorioAtual->entries[i]->name, name) == 0){
224                 if(strcmp(diretorioAtual->entries[i]->type, "dir") == 0){ //se encontrou o arquivo, checar se eh dir
225                     printf("%s\t", diretorioAtual->entries[i]->name);
226                 }else{ //se não eh diretorio, lista as informacoes do arquivo (nome)
227                     printf("%s.%s\t", diretorioAtual->entries[i]->name, diretorioAtual->entries[i]->type);
228                 }
229                 notFound = 0;
230             }
231         }
232
233         if(notFound == 1){
234             printf("Arquivo ou Diretorio inexistente.");
235         }
236     }
237     printf("\n");
238 }
239

```

Figura 35 – Função de ls

#### 4.4.6 format dsc: format\_dsc(int blockQtde)

Função usada no comando “format dsc” para formatar o disco que está sendo usado no momento. Utilizamos o valor 512 como o padrão para o tamanho dos blocos, podendo ser modificada apenas a quantidade de blocos no disco (para modificar o tamanho do bloco, deve-se mudar o define BLOCKSIZE do arquivo main.c). Faz uso da função facc\_format para fazer a formatação. A figura 36 mostra seu conteúdo.

```

438 void format_dsc(int blockSize, int blockQtde){
439     if(blockQtde < 4){ //se o numero de blocos for menor que 4, nao eh possivel formatar
440         printf("Entrada inválida, a quantidade de blocos deve possuir ao menos 4 blocos (sb, fat, root, data#1)\n");
441         return;
442     }
443
444     facc_format(blockSize, blockQtde); //formata o disco
445 }

```

Figura 36 – Função de format dsc

#### 4.4.7 mv: moveltem(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 atribuímos 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 37, 38 e 39 mostra sua implementação.

```
211 void moveItem(Superblock* sb, Fat* fat, DirChunk* diretorioAtual, char* source, char* destination){
212     int sourceEntry = -1;
213     //checar se existe uma entrada com este nome
214     int entryFound = -1;
215     for(int i=0; i<diretorioAtual->meta.entryQtde; i++){ // Percorre o diretorio
216         if(strcmp(diretorioAtual->entries[i]->name, source) == 0){ // Se encontrar o arquivo
217             entryFound = i;
218             sourceEntry = i;
219             break;
220         }
221     }
222     //se nao existir uma entrada retorna erro
223     if(entryFound == -1){
224         printf("Erro, nao existe um arquivo com este nome\n");
225         return;
226     }
227     //Separar em novo nome e encontrar o proximo diretorio
228     char path[250];
229     char newName[8];
230     //encontrar novo path
231     strcpy(path, destination);
232     int size = strlen(path);
233     while(size>0){ // percorre o caminho
234         if(path[size] == '/'){ // se encontrar um '/'
235             path[size] = '\0'; // corta o caminho
236             break; // para de percorrer o caminho
237         }
238         size--;
239     }
240     if(strlen(path) == strlen(destination)){ //se nao houver nenhum '/' eh pra renomear, portanto path eh vazio
241         path[0] = '\0';
242     }
243     int nameIndex = 0;
244     for(int i=size+1; i<strlen(destination); i++){ //encontrar o nome do arquivo
245         newName[nameIndex] = destination[i];
246         nameIndex++;
247     }
```



```

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 39 – Função mv

#### 4.4.8 cp: cpyItem(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 makeDirectory; 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 diretório 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 40, 41, 42, 43, 44, 45, 46 e 47 mostra sua

[illegible][illegible]



```

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 42 – 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 se arquivo 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 43 – Função copy

```

606     }
607 }
608 }
609 if(entryFound == -1){
610     printf("Erro, diretorio de destino '%s' nao existe.\n", token);
611     return;
612 }
613
614 //se o diretorio existir:
615 firstBlock = dir->entries[entryFound]->firstBlock;
616 if(dir != diretorioAtual) facc_unloadDirectory(dir);
617 dir = facc_loadDir(sb, fat, firstBlock);
618
619 token = strtok(NULL, "/");
620 }
621
622 //verificar se ja existe um arquivo com o mesmo nome no diretorio destino
623 for(int i=0; i<dir->meta.entryQtde; i++){ // Percorre o diretorio
624     if(strcmp(dir->entries[i]->name, newName) == 0){ // Se encontrar um arquivo
625         printf("Erro, ja existe um arquivo com o nome '%s' no diretorio destino '%s'\n", newName, dir->meta.name);
626         return;
627     }
628 }
629
630 //#####
631 //carregar arquivo e fazer um file_chunk* dele.
632 file_chunk* fc = importFile(source);
633
634 //se deu certo
635 if(fc != NULL){
636     //guardar arquivo em um bloco
637     int freeBlock = facc_findFreeBlock(sb, fat);
638     saveFile(sb, fat, fc, freeBlock);
639
640     //criar entrada no diratual;
641     Entry* ref = (Entry*)malloc(sizeof(Entry));
642     strcpy(ref->name, newName);
643     if(strlen(fileType) > 0) strcpy(ref->type, fileType);
644     else strcpy(ref->type, "file");

```

Figura 44 – Função copy

```

645     ref->bytes = fc->bytes;
646     ref->firstBlock = freeBlock;
647
648     //adicionar entrada no dir atual;
649     facc_updateDirAdd(sb, fat, dir, ref);
650
651     if(dir->meta.firstBlock == diretorioAtual->meta.firstBlock){
652
653         int dirBlock = dir->meta.firstBlock;
654         facc_unloadDirectory(diretorioAtual);
655         diretorioAtual = facc_loadDir(sb, fat, dirBlock);
656     }
657
658     //del file_chunk
659     facc_unloadDirectory(dir);
660     free(fc->file);
661     free(fc);
662 }
663
664
665 //caso seja apenas pra copiar um arquivo interno
666 //caso seja apenas pra copiar um arquivo interno
667 //caso seja apenas pra copiar um arquivo interno
668 //caso seja apenas pra copiar um arquivo interno
669 }else{
670     if(strcmp(token, source) == 0){
671         printf("Erro, nao eh possivel copiar um arquivo para dentro dele mesmo. (%s > %s)\n", source, token);
672         return;
673     }
674
675     while(token != NULL){
676         //verificar se o diretorio existe
677         entryFound = -1;
678         for(int i=0; i<dir->meta.entryQtde; i++){ // Percorre o diretorio
679             if(strcmp(dir->entries[i]->name, token) == 0){ // Se encontrar um arquivo
680                 if(strcmp(dir->entries[i]->type, "dir") == 0){ //se o arquivo for dir
681                     entryFound = i;
682                 }

```

Figura 45 – Função copy

[illegible]

Figura 46 – Função copy

```

721
722 //caso nao esteja vazio
723 makeDirectory(sb, fat, dir, newName);
724 facc_unloadDirectory(auxdir);
725
726 //caso nao seja um diretorio
727 }else{
728     //openFile;
729     int freeBlock = facc_findFreeBlock(sb, fat);
730     file_chunk* fc = openFile(sb, fat, dir->entries[sourceEntry]);
731     saveFile(sb, fat, fc, freeBlock);
732
733     //createMeta
734     Entry* ref = (Entry*)malloc(sizeof(Entry));
735     strcpy(ref->name, newName);
736     strcpy(ref->type, dir->entries[sourceEntry]->type);
737     ref->firstBlock = freeBlock;
738     ref->bytes = fc->bytes;
739     facc_updateDirAdd(sb, fat, dir, ref);
740
741     //tirar da memoria o diretorio destino
742     facc_unloadDirectory(dir);
743     free(fc->file);
744     free(fc);
745 }
746 }
747 }
748 }

```

Figura 47 – Função copy

```
4.4.9 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 48 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                 }else{ // Se o diretorio esta vazio
70                     facc_updateDirDel(sb, fat, diretorioAtual, i); // remove o diretorio do diretorio atual
71                 }
72
73                 facc_unloadDirectory(auxDir); // desaloca o diretorio auxiliar
74                 notFound = 0;
75                 break;
76             }
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 48 – Função de rmItem

## 4.5 Parte 4.1: Comandos adicionais de manipulação.

### 4.5.1 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 49 mostra seu conteúdo.

```

22 void help(){ // Função que imprime a ajuda
23     printf("Descricao dos comandos:\n\n");
24
25     printf("cd - Acessa um diretorio\n");
26     printf("\tcd <dir>\n\tcd /caminho/nome\n\n");
27
28     printf("mkdir - Cria um diretorio\n");
29     printf("\tmkdir <dir>\n\tmkdir /caminho/nome\n\n");
30
31     printf("rm - Remove um diretorio/arquivo\n");
32     printf("\trm <dir | arquivo>\n\t rm /caminho/nome\n\n");
33
34     printf("cp - Copia um diretorio/arquivo\n");
35     printf("\tcp <dir | arquivo>\n\tcp /caminho/nome\n\n");
36
37     printf("mv - Move Diretório/Arquivo.\n");
38     printf("\tmv <dir | arquivo>\n\n");
39
40     printf("ls - Lista o conteudo de um diretorio ou mostra as informacoes ddo arquivo\n");
41     printf("\tls <dir | arquivo>\n\n");
42
43     printf("pwd - Mostra o caminho do diretorio atual\n");
44     printf("\tpwd\n\n");
45
46     printf("format dsc - Formata o disco simulado\n");
47     printf("\tformat dsc <num blocks>\n\n");
48 }

```

Figura 49 – Função de ajuda

#### 4.5.2 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 51 mostra seu conteúdo.

```

804 void printFat(Superblock* sb, Fat* fat){
805     printf("FAT => ");
806     for(int i = 0; i < sb->blockQtde; i++){
807         if(fat[i] == FAT_L)         printf("L");
808         else if(fat[i] == FAT_F)    printf("F");
809         else if(fat[i] == FAT_R)    printf("R");
810         else if(fat[i] == FAT_B)    printf("B");
811         else                        printf("%d", fat[i]);
812         if(i != sb->blockQtde-1)    printf(", ");
813     }
814     printf("\n");
815 }

```

Figura 50 – Função para mostrar a FAT

## 5 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 51 mostra como isso é feito.

```

16 int main(int argc, char* argv[]){
17     char command[256];
18     char* pathing = createPathing(); // Cria o caminho atual (root)
19
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 formatar-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                 format_dsc(atol(command)); // Chama a função que formata o disco
31                 printf("Disco criado.\n"); // Informa que o disco foi criado
32                 break; // Sai do loop
33
34             }else if(strncmp(command, "n", 1) == 0){ // Se a resposta for "n"
35                 printf("Disco carregado com sucesso!\n"); // Informa que o disco foi carregado
36                 break;
37
38             }else{
39                 printf("Opcao invalida.\n");
40             }
41         }
42
43         //arquivo nao existe
44     }else{
45         facc_format(BLOCKSIZE, BLOCKQTDE); // Chama a função que cria o disco
46         printf("Disco gerado com %d blocos de %d bytes.\nUtilize o comando format dsc <qtdBLocos> para formatar", BLOCKQTDE, BLOCKSIZE); // Informa que o disco foi criado
47     }

```

Figura 51 – 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 52.

```

49 //load superbloc
50 Superblock* sb = facc_loadSuperblock(); // Carrega o superbloc 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 52 – 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 53 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 53 – Simulador de shell.

## 5.1 Resultados em execução

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

#### 5.1.1 ls, mkdir, cd, rm e pwd.

Exemplo usando ls, mkdir, cd, rm e pwd na figura 54.

```
(~root)$ > ls
.
..
(~root)$ > mkdir teste
(~root)$ > ls
.
..
teste
(~root)$ > cd teste
(~root/teste)$ > ls
.
..
(~root/teste)$ > cd ..
(~root)$ > ls
.
..
teste
(~root)$ > rm teste
(~root)$ > ls
.
..
(~root)$ > mkdir teste2
(~root)$ > cd teste2
(~root/teste2)$ > pwd
~root/teste2
```

Figura 54 – Saídas de: ls, mkdir, cd, rm e pwd.

#### 5.1.2 ls, cp e cd.

Exemplo usando ls, cp e cd na figura 55.

```
(~root)$ > ls
.
..
teste
teste2
(~root)$ > cp teste teste2/teste
(~root)$ > ls
.
..
teste
teste2
(~root)$ > cd teste2
(~root/teste2)$ > ls
.
..
teste
```

Figura 55 – Saídas de: ls, cp e cd.

#### 5.1.3 mv, ls e cd.

Exemplo usando mv, ls e cd na figura 56.

```
(~root)$ > ls
.
..
oie
teste
(~root)$ > mv teste oie/teste
(~root)$ > ls
.
..
oie
(~root)$ > cd oie
(~root/oie)$ > ls
.
..
oi(png)
teste
```

Figura 56 – Saídas de: mv, ls e cd.

#### 5.1.4 ls e exp.

Exemplo usando ls e exp na figura 57 e o resultado na pasta na figura 58.

```
(~root/oie)$ > ls
.          oi(png)
(~root/oie)$ > exp oi oi
(~root/oie)$ > ls
```

Figura 57 – Saídas de: ls e exp.

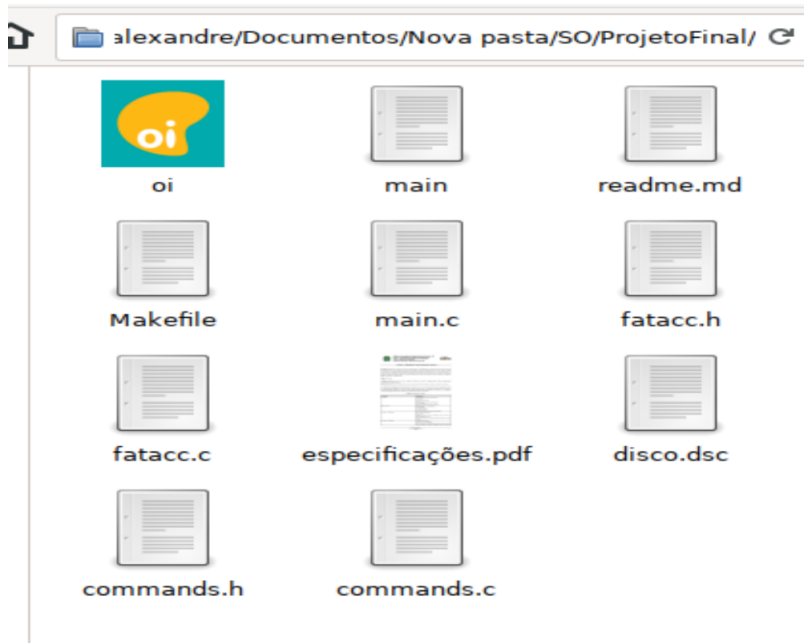


Figura 58 – Saídas de: ls, mkdir, cd, rm e pwd.

## 6 CONCLUSÃO

Nessa atividade nós realizamos a implementação do sistemas de arquivos, e com isso podemos verificar como o shell é realizado na prática, como o gerenciamento 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.

## 7 REFERÊNCIAS

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

<sup>2</sup> Slides do professor.

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