Universidade Federal de Uberlândia
Faculdade de Computação
Curso de Bacharelado em Ciência da Computação

Felipe Nunes
Igor Blanco Toneti
Lucas Guimarães Mendes

Sistema de arquivos - ufuFS

Relatório Técnico

Uberlândia
2019

1 - Introdução

O presente relatório técnico descreve e detalha a implementação de um sistema de arquivos nomeado ufuFS, proposto pelo professor Dr. Rivalino Matias Júnior como trabalho de conclusão da disciplina de Sistemas Operacionais oferecida pela Faculdade de Computação (FACOM) da Universidade Federal de Uberlândia (UFU). Esse sistema de arquivos foi baseado no *Unix File System* porém de forma simplificada e sem muitas otimizações.

Segundo a descrição do trabalho, o Sistema de Arquivos deverá ser projetado para ser utilizado em pen drives, contendo três partes principais que são: a estrutura do sistema de arquivo em si (ufuFS), o formatador (ufuFS_format) e por fim um Shell capaz de interagir com o sistema de arquivos (ufuFS_shell).

2 - Conceitos

Alguns conceitos de Sistemas Operacionais, mais especificamente sobre Sistema de Arquivos são importantes para a compreensão do trabalho. Alguns deles são:

2.1 - Trilhas

Uma trilha do disco é um caminho circular na superfície onde a informação é armazenada ou lida magneticamente.

2.2 - Setores

Em armazenamento de discos setores são subdivisões de uma trilha em um disco magnético. Cada setor armazena uma quantidade fixa de dados acessíveis pelo usuário. Tradicionalmente 512 bytes para discos magnéticos mas esse valor pode ser diferente pois é decidido pelo fornecedor.

2.3 - Blocos/Clusters

Blocos são grupos de setores em que o sistema operacional consegue referenciar. São as menores unidades as quais um sistema de arquivo consegue operar pois são definidos como múltiplos de setores.

2.4 - Super Bloco

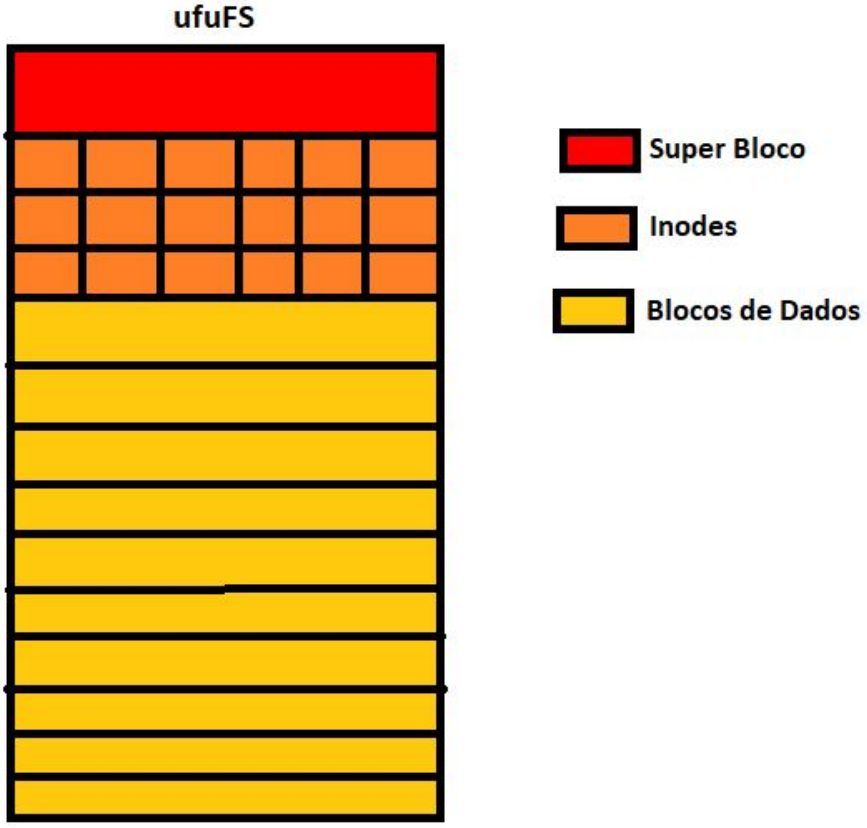
O superbloco é uma estrutura que registra as características do sistema de arquivos, incluindo seu tamanho, tamanho do bloco, número de reconhecimento do sistema de arquivo, tamanho e localização dos inodes como também dos bloco de dados.

2.5 - Inodes

Os inodes nos sistemas de arquivos são estruturas que armazenam todas as informações dos arquivos localizados nos blocos de dados. Os inodes mostram informações como nome, tamanho, data de acesso, permissões, entre outros. Eles também servem para referenciar o arquivo em questão pois armazenam seu endereço na memória.

3 - Estrutura do ufuFS

A estrutura do sistema de arquivo ufuFS é basicamente dividida em três partes, super bloco, tabela de inodes e os blocos de dados como demonstrado na figura abaixo.



3.1 - Super Bloco

É a estrutura responsável por armazenar as características do sistema de arquivos em si.

O super bloco ufuFS_super_bloco apesar de ser simples possui 9 informações importantes sobre nosso sistema. Nela guardamos:

- **Tipo** = É representado por um número em hexadecimal usado para reconhecer o sistema ufuFS.

- **Tam_bloco** = Aqui armazenamos quantidade de setores que um bloco possui em bytes. No caso do ufuFS, há um setor por bloco.

- **Qtd_bloco_pen** = Variável que armazena o número de blocos que possui no pendrive que queremos utilizar.

- **Tam_inode** = Armazena-se o número de bytes que cada estrutura inode terá reservada para si.
- **Tam_tabela_inode** = Tamanho em blocos destinados as structs dos inodes. Ou seja, blocos em que serão colocados todos os metadados dos arquivos.

- **Bloco_ini_inode** = Serve de indicação para o file descriptor onde começa as tabelas de inodes. É medido em blocos.

- **Qtd_inode** = Inteiro que representa a quantidade de inodes do sistema ufuFS. Seria a quantidade máxima de arquivos suportados.

- **Ini_data_block** = Bloco em que começa os arquivos. É medido em blocos.

- **Reservado** = É um vetor criado com o intuito de ocupar o restante do superbloco. Deste modo completamos 512 bytes por struct ufuFS_super_bloco, suprindo inteiramente o tamanho de um bloco.

3.2 - Inodes

Estrutura onde armazenam-se todas as informações referentes aos arquivos do ufuFS. UfuFSInode possui 64 bytes reservados para:

- **Nome** = Vetor de chars com capacidade para 12 caracteres contando o caractere nulo “\0”.

- **Tamanho** = Inteiro com a propriedade de armazenar o tamanho do arquivo referenciado.

- **HoraCriacao** = Variável do tipo long long reservada para o armazenamento da hora de criação do arquivo.

- **HoraModificacao** = Possui as mesmas propriedades da variável horaCriacao, única diferença está no seu propósito pois horaModificacao armazena o horário em que tal arquivo foi lido ou escrito.

- **Blocos** = Vetor que referencia os blocos do pen drive utilizados pelo arquivo.

3.3 - Blocos de Dados

Parte reservada no sistema de arquivos onde armazena-se os arquivos. Os blocos de dados do ufuFS são representados nos inodes pelo vetor blocos. Cada posição do vetor faz referência a um 1 bloco no nosso sistema, então cada arquivo ocupa no máximo 8 blocos.

4 - Implementação

A implementação do presente trabalho foi realizado em linguagem C para o sistema operacional Linux. A seguir, explicaremos os detalhes técnicos das três principais partes do nosso sistema de arquivos, ufuFS, ufuFS_format, ufuFS_shell, incluindo o suporte às operações de criação de arquivos e as operações de open, read, write, close e seek.

4.1 - ufuFS

As definições da estrutura do sistema de arquivo estão implementadas no arquivo “ufuFS.h”. Neste código-fonte estão definidas algumas constantes que representam as características do nosso sistema de arquivos, e facilita a implementação e manutenção do projeto. Essas constantes são:

- **ufuFS_TYPE** recebe uma constante escolhida aleatoriamente para identificar o sistema de arquivos, e assim poder realizar uma checagem ao ser inicializado.
- **ufuFS_BLOCK_SIZE** recebe o tamanho do bloco em bytes. No caso do nosso sistema de arquivos definimos como 512 bytes, ou seja, o bloco possui exatamente um setor do dispositivo.
- **ufuFS_INODE_SIZE** recebe o tamanho em bytes da área de metadados do inode. Definimos como 64 bytes.
- **ufuFS_FILENAME_LEN** recebe o tamanho máximo (número de caracteres) para os nomes dos arquivos. Definimos o mesmo como 11.
- **ufuFS_DATA_BLOCK** recebe a quantidade de blocos de dados reservado para cada inode. O cálculo é feito pela seguinte fórmula:
  ((ufuFS_INODE_SIZE - ((ufuFS_FILENAME_LEN + 1) + 4 + 2*8)) / 4).
  A parte em destaque representa o tamanho total em bytes dos dados pré alocados no inode (nome,tamanho do arquivo e mais 2 variáveis reservadas para data e hora). Logo após, subtrai-se a parte destacada do tamanho total do inode. E por fim divide-se por 4 (tamanho de um inteiro em bytes) a fim de obter a quantidade de blocos de dados. Esse cálculo é feito com objetivo de tornar a estrutura do sistema de arquivos mais genérica, facilitando uma possível mudança na configuração do inode.

```c
#define ufuFS_TYPE 0x13090D15  /*Reconhecimento do sistema de arquivos */
#define ufuFS_BLOCK_SIZE 512 /*Tamanho de cada bloco em Bytes*/
#define ufuFS_INODE_SIZE 64 /*Tamanho de cada inode em Byts*/
#define ufuFS_FILENAME_LEN 11 /*Tamanho maximo p/ nome de arquivo*/
#define ufuFS_DATA_BLOCK ((ufuFS_INODE_SIZE - ((ufuFS_FILENAME_LEN + 1) + 4 + 2*8)) / 4)
```

Além das constantes este código possui duas structs importantes, a ufuFS_super_bloco e a ufuFSInode que são respectivamente a implementação das funções que descrevem os metadados do super bloco e dos inodes.

O ufuFS_super_bloco guardará informações como: tipo do sistema de arquivo, tamanho do bloco, quantidade de blocos do pendrive, tamanho do inode, tamanho da tabela de inodes, bloco de início da tabela de inodes, quantidade de inodes, início do bloco de dados e um array chamado reservado que irá ocupar o tamanho restante para que o super bloco ocupe exatamente um bloco.

```c
typedef struct ufuFS_super_bloco{
    unsigned int tipo;
    unsigned int tam_bloco;
    unsigned int qtd_bloco_pen;
    unsigned int tam_inode;
    unsigned int tam_tabela_inode;
    unsigned int bloco_ini_inode;
    unsigned int qtd_inode;
    unsigned int ini_data_block;
    unsigned int reservado[ufuFS_BLOCK_SIZE / 4 - 8];
} ufuFS_superBloco;
```

O ufuFSInode guardará informações como: nome do arquivo (array de 11 posições mais 1 byte para o \0), tamanho do arquivo, data e hora de criação e modificação, e os espaços reservados para os dados do arquivo representado por um vetor de 8 posições, sendo que cada posição indica um bloco, ou seja, pode ocupar até 8 blocos, que totaliza 4096 bytes.

```c
typedef struct ufuFSInode{
    char nome[ufuFS_FILENAME_LEN + 1];
    unsigned int tamanho;
    unsigned long long horaCriacao;
    unsigned long long horaModificacao;
    unsigned int blocos[ufuFS_DATA_BLOCK];
} ufuFS_INODE;
```

Além da estrutura definida no ufuFS.h, há a implementação de funções básicas de um sistema de arquivos no código-fonte ufuFS.c. Esse código-fonte inclui também o shell e os comandos suportados por ele, os quais iremos detalhar mais para frente neste relatório.

4.2 - ufuFS_format

Na parte da formatação, incluímos a biblioteca ufuFS.h que contém as estruturas já prontas. Assim, inicializamos a estrutura global do super bloco com o nome “sb” e atribuímos às constantes também já definidas no ufuFS.h que são: o tipo, tamanho do bloco, tamanho do inode e o bloco de início dos inodes.

Logo após a inicialização do super bloco, partimos para a inicialização dos inodes, a qual demos o nome de “ufuInode”.

Para que se possa realizar a formatação é preciso ter acesso à localização da partição onde o pendrive se encontra. E para tal é necessário que se passe o caminho do mesmo como parâmetro para a função main, que guardará na variável argv[].

Após ter o caminho já definido e guardado na variável argv[1], cria-se uma variável int chamada “ufuFD”. Essa variável receberá o retorno da system call open, que será um file descriptor referente ao pendrive. A partir disso, toda modificação será feita referenciando-se o file descriptor “ufuFD”.

A system call open recebe como parâmetro de entrada o caminho da partição do pendrive e a permissão O_RDWR que possibilita fazer tanto escrita como leitura no arquivo.

Em seguida é utilizado a system call ioctl para obter o tamanho total do pendrive. Essa informação fica armazenada na variável size criada anteriormente. Caso haja alguma falha ao obter o tamanho, a system call retorna -1.

```
ufuFD = open(argv[1], O_RDWR);
if (ufuFD == -1)
{
    fprintf(stderr, "Erro ao formatar %s: %s\n", argv[1], strerror(errno));
    return 2;
}
if (ioctl(ufuFD, BLKGETSIZE64, &size) == -1) /*size recebe o tamanho total do dispositivo
{
    fprintf(stderr, "Erro ao obter o tamanho de %s: %s\n", argv[1], strerror(errno));
    return 3;
}
```

Se tudo ocorrer sem nenhum erro, inicia-se o preenchimento dos campos da estrutura do super bloco, que são:

- **sb.qtd_bloco_pen** = (tamanho em bytes do pendrive) / (tamanho em bytes do bloco)
    - Resulta na quantidade total de blocos do pendrive.
- **sb.tam_tabela_inode** = (quantidade de blocos do pendrive) * 0.10

    - Reserva 10% do total de blocos do pendrive para armazenar a tabela de inodes.
- **sb.qtd_inode =** (quantidade de blocos reservados para tabela de inodes) * (tamanho de cada bloco) / (número de bytes que cada estrutura inode possui)
    - Resulta na quantidade de arquivos que podemos criar no sistema de arquivos.
- **sb.ini_data_block =** (bloco de início da tabela de inodes) + (quantidade de blocos ocupados pela tabela de inodes)
    - Resulta no endereço do primeiro bloco de dados;

Por fim, a estrutura é gravada no pendrive através da função "writeSuperBloco" que por sua vez utiliza a system call write. Após a escrita, chama-se a função "resetaInode" que preenche todos os blocos de dados com 0, indicando que os blocos estão disponíveis para serem utilizados.

Finaliza-se a formatação utilizando a system call close.

```
sb.qtd_bloco_pen = size / ufuFS_BLOCK_SIZE;
sb.tam_tabela_inode = sb.qtd_bloco_pen * ufuRazaoInode;
sb.qtd_inode = sb.tam_tabela_inode * (sb.tam_bloco / sb.tam_inode);
sb.ini_data_block = ufuFSComecoDoBlocoInode +  sb.tam_tabela_inode;

printf("Partitioning %Ld byte sized %s ... ", size, argv[1]);
fflush(stdout);
writeSuperBloco(ufuFD, &sb);
resetaInode(ufuFD, &sb);

close(ufuFD);
```

4.3 - ufuFS_shell

Antes de o shell entrar em ação, a função main realiza o open e a checagem do sistema de arquivo. A função main recebe como parâmetro de entrada o caminho do pendrive já formatado. Esse caminho é armazenado em uma variável chamada ufuFS_file.

Também é criada uma outra variável chamada ufuFD na qual é guardado o descritor de arquivo retornado pela função open. Nesta função open passa-se o ufuFS_file e a permissão para leitura e gravação (O_RDWR) do referido dispositivo.

Em seguida, usa-se a system call read para ler o superbloco que está armazenado no pendrive e salvar na estrutura global do super bloco criado no código-fonte do shell chamada de "sb". Com a estrutura já carregada na memória, as modificações serão feitas sobre ela e ao finalizar as operações serão salvas no pendrive.

Após essa leitura do superbloco é feito uma checagem do tipo do sistema de arquivos, se não for o esperado (ufuFS_TYPE), o programa é finalizado. Caso contrário chama-se a função ufuFS_shell.

```
ufuFS_file = argv[1];
ufuFD = open(ufuFS_file, O_RDWR);

if(ufuFD == -1)
{
    fprintf(stderr,"Não foi possível inicializar o shell sobre %s\n",ufuFS_file);
    return 2;
}

read(ufuFD, &sb, sizeof(ufuFS_superBloco));

if(sb.tipo != ufuFS_TYPE)
{
    fprintf(stderr, "Sistema de Arquivo inválido. Finalizando...\n");
    close(ufuFD);
    return 3;
}

ufuFS_shell(ufuFD);
close(ufuFD);
```

Ao iniciar a função ufuFS_shell, é exibido uma mensagem de boas vindas e um menu com informações sobre algumas características do sistema de arquivos. Em seguida é chamado uma outra função com nome inicializa_shell, que tem como parâmetro de entrada o file descriptor.

```
void ufuFS_shell(int ufuFD)
{
    int finalizado;
    char cmd[256], *fn;
    int ret;

    finalizado = 0;

    printf("Bem vindo ao Shell de navegação ufuFS\n\n");
    printf("Tamanho do bloco            : %d bytes\n", sb.tam_bloco);
    printf("Tamanho da partição         : %d blocos\n", sb.qtd_bloco_pen);
    printf("Tamanho do Inode            : %d bytes\n", sb.tam_inode);
    printf("Tamanho da tabela de Inodes : %d blocos\n",sb.tam_tabela_inode);
    printf("Quantidade de Inodes        : %d\n", sb.qtd_inode);
    printf("\n");

    inicializa_shell(ufuFD);
```

Na função inicializa_shell, a estrutura criada na memória é formatada de acordo com a estrutura que está no pendrive, marcando os blocos que estão sendo utilizados através de uma lista criada globalmente.

Após a finalização da função inicializa_shell, é iniciado o loop que recebe os comandos realizados em linha de comando e encaminha para cada função responsável por realizar as operações.

```
if(strcmp(cmd, "help") == 0)
{
    ufuFS_help();
    continue;
}
else if(strcmp(cmd, "exit") == 0)
{
    finalizado = 1;
    continue;
}
else if(strcmp(cmd, "list") == 0)
{
    ufuFS_list(ufuFD);
    continue;
}
else if(strncmp(cmd, "create", 6) == 0)
{
    if(cmd[6] == ' ')
    {
        fn = cmd + 7;
        while (*fn == ' ') fn++;
        if(*fn != '\0')
```

Após a finalização do loop, a função finaliza_shell é chamada. Essa função libera a lista de blocos usados, que foi criada de forma global para ser usada durante as operações do sistema de arquivos a fim de saber os blocos disponíveis para utilização.

Por fim a função main fecha o arquivo através da system call close, utilizando como parâmetro de entrada o file descriptor criado no início da função.

4.4 - Funções

4.4.1 - Create

A função create recebe como parâmetros de entrada o file descriptor referente ao pendrive e um nome para o arquivo a ser criado. Em seguida, com a system call lseek, apontamos para o primeiro bloco subsequente ao super bloco, que é a tabela de inodes.

```
void ufuFS_create (int ufuFD, char *fn)
{
    int i;
    ufuFS_INODE i_node;

    lseek(ufuFD, sb.bloco_ini_inode * sb.tam_bloco, SEEK_SET);
```

Em seguida, é realizado uma verificação na tabela de inodes para achar o primeiro inode sem dados escritos ou se tem algum arquivo com o mesmo nome. Caso tenha um arquivo com o mesmo nome é exibido uma mensagem de erro e finaliza-se a função. Caso contrário, incrementa-se o laço de repetição até achar um inode vazio.

```
void ufuFS_create (int ufuFD, char *fn)
{
    int i;
    ufuFS_INODE i_node;

    lseek(ufuFD, sb.bloco_ini_inode * sb.tam_bloco, SEEK_SET);

    for (i = 0; i < sb.qtd_inode; i++){
        read(ufuFD, &i_node, sizeof(ufuFS_INODE));
        if(!i_node.nome[0]) break;
        if(strcmp(i_node.nome, fn) == 0)
        {
            printf("O arquivo %s já existe!\n",fn);
            return;
        }
    }
```

Se o índice do laço de repetição for igual à quantidade de inodes significa que não há mais entradas disponíveis.

```
if(i == sb.qtd_inode)
{
    printf("Não há mais entradas!");
    return;
}
```

Logo após as verificações, o atributo nome é copiado para a estrutura do inode e os demais atributos são zerados. Além disso a data de criação e modificação são atualizadas.

```
strncpy(i_node.nome, fn, ufuFS_FILENAME_LEN);

i_node.nome[ufuFS_FILENAME_LEN] = 0;
i_node.tamanho = 0;
i_node.horaCriacao = time(NULL);
i_node.horaModificacao = time(NULL);

for(i=0; i < ufuFS_DATA_BLOCK; i++)
{
    i_node.blocos[i] = 0;
}
```

Por fim, as modificações são salvas no pendrive utilizando-se a system call write.

4.4.2 - Open

Para a compreensão da função open deve-se entender que o objetivo é pegar a estrutura e carregá-la na memória principal.

No sistema de arquivo ufuFS passamos para a função main, por meio de um argumento, a partição em que se localiza o pendrive. Depois de abrirmos o arquivo com open, obtendo assim seu file descriptor, necessitamos checar o tipo do sistema de arquivo.

Nesse momento utiliza-se a system call read, que armazena o conteúdo do pendrive em uma estrutura criada globalmente (sb). Essa estrutura será usada para fazer as alterações no sistema de arquivos, além de poder comparar seu tipo.

Feito isso, devemos carregar os blocos existentes no ufuFS para a memória principal, a idéia é criar uma “cópia” de toda a estrutura para facilitar as modificações.

Primeiro devemos alocar um vetor com a quantidade de blocos no nosso sistema, o segundo passo é marcar todos os blocos que representam o superbloco e inodes como usados pois não podemos criar, deletar e escrever nesses arquivos. O terceiro e último passo é andar pelos inodes checando se os nomes não estão vazios, caso não estejam, esse inode é marcado no vetor como usado, indicando que existe algum arquivo. As funcionalidades descritas acima se dividem basicamente em duas funções: a main e o inicializa_shell.

4.4.3 - Read

Criada com o nome ufuFS_read, essa funcionalidade recebe como parâmetro um file descriptor que tem como referência o pendrive e um nome de arquivo. UfuFS_read faz primeiramente uma busca pelos inodes usando a função ufuFS_seek para saber em qual inode as informações do arquivo que queremos ler está guardada.

Descoberto o inode precisamos andar pelos blocos de dados a qual esse inode faz referência. Criamos assim uma iteração para passarmos por todos os data blocks desse arquivo. A cada iteração ocorre uma checagem para analisarmos se não há nada escrito, assim sabemos quando pararmos de ler do arquivo.

```
for(block_i = 0; block_i < ufuFS_DATA_BLOCK; block_i++)
{
    if(!i_node.blocos[block_i]) break;

    to_read = (rem_to_read >= sb.tam_bloco) ? sb.tam_bloco : rem_to_read;

    lseek(ufuFD, i_node.blocos[block_i] * sb.tam_bloco, SEEK_SET);
    read(ufuFD, block, to_read);
    write(1, block, to_read);

    already_read += to_read;
    rem_to_read -= to_read;

    if(!rem_to_read) break;
}
```

Essa leitura será exibida no shell através da system call write com o primeiro parâmetro de entrada sendo 1.

Outra versão do read foi implementada, chamada read_to_file. Essa versão foi desenvolvida para ser usada na função copy_to_so que foi detalhada neste relatório. A principal diferença nesta função é o seu retorno, que ao contrário da saída ser exibida no terminal é escrita em um outro arquivo.

4.4.4 - Write

Criada como ufuFS_write, essa função recebe como parâmetro de entrada o file descriptor e o nome do arquivo que será escrito.

Ao inicializar, a função utiliza o ufuFS_seek para verificar se o arquivo a ser escrito existe. Caso exista é retornado o seu endereço. E se não encontrado, retorna-se uma mensagem de erro e a função é finalizada.

Após esta primeira verificação, a função libera os blocos caso estejam ocupados, ou seja, o conteúdo anterior do arquivo será sobrescrito. Essa liberação de blocos é feito pela função put_data_block, que basicamente coloca 0’s no array de blocos usados definido de forma global (used_blocks).

```
for(block_i = 0; block_i < ufuFS_DATA_BLOCK; block_i++)
{
    if(!i_node.blocos[block_i])
    {
        break;
    }
    put_data_block(ufuFD, i_node.blocos[block_i]);
}
```

Em seguida, a função entra em um laço de repetição e pega o conteúdo digitado no shell usando a system call read com primeiro parâmetro 0. Além disso, o read retorna 0 quando não houver mais o que ler, assim o laço de repetição é interrompido.

```
while ((cur_read = read(0, block + cur_read_i, to_read)) > 0)
```

Se o arquivo já tiver todos os blocos ocupados ou o sistema de arquivo estiver cheio o laço de repetição é interrompido.

```
if(block_i == ufuFS_DATA_BLOCK) //Tamanho limete do arquivo
    break;

if((free_i = get_data_block(ufuFD)) == -1) //Sistema de arquivos cheio
    break;
```

A variável free_i vai receber o índice do bloco livre para ser gravado, através da função get_data_block, e em seguida, o lseek irá apontar para o endereço onde esse bloco livre está, para que o write em seguida grave o conteúdo.

```
lseek(ufuFD, free_i * sb.tam_bloco, SEEK_SET); /*Aponta para o início do proximo bloco livre*/

write(ufuFD, block, sb.tam_bloco); //Grava o conteúdo lido pelo read
```

Em seguida, o bloco escrito é inserido no vetor dos blocos de dados que o inode possui, incrementa o contador de blocos já lidos e soma o total de bytes lidos.

```
i_node.blocos[block_i] = free_i;
block_i++;
total_size += sb.tam_bloco;
```

Quando somente uma parte do bloco foi preenchido, a escrita é tratada separadamente.

```
if((cur_read <= 0) && (cur_read_i))
{
    //Quando não se usa todo o bloco
    if((block_i != ufuFS_DATA_BLOCK) && ((i_node.blocos[block_i] = get_data_block(ufuFD)) != -1))
    {
        lseek(ufuFD, i_node.blocos[block_i] * sb.tam_bloco, SEEK_SET);
        write(ufuFD, block, cur_read_i);

        total_size += cur_read_i;
    }
}
```

Por fim, a função salva no estrutura do inode o tamanho total, altera a hora de modificação do arquivo, aponta para o arquivo contido no pendrive (lseek) e grava estrutura modificada.

```
i_node.tamanho = total_size;
i_node.horaModificacao = time(NULL);

lseek(ufuFD, sb.bloco_ini_inode * sb.tam_bloco + i * sb.tam_inode, SEEK_SET);
write(ufuFD, &i_node, sizeof(ufuFS_INODE));
```

Outra versão do write foi implementada, chamada write_to_file. Essa versão foi desenvolvida para ser usada na função copy_to_usb que foi detalhada neste relatório. A principal diferença nesta função é o seu parâmetro de entrada, que acrescenta o caminho do arquivo no computador. Além disso, dentro da função é criado um arquivo no pendrive, o qual receberá o conteúdo do arquivo do computador.

4.4.5 - Seek

A função ufuFS_seek recebe como parâmetro o file descriptor, o nome do arquivo e a estrutura inode referenciada. O ufuFS_seek tem como objetivo buscar o arquivo pelo nome e retornar o número do inode deste arquivo.

O primeiro passo da função é apontar o file descriptor para o início do bloco de inodes usando system call lseek, assim pode-se caminhar por eles e compará-lo nome por nome. A cada iteração lemos do pendrive a estrutura inode colocada no buffer pela system call read e assim temos a possibilidade de checar o nome. Caso encontre o arquivo retorna-se o número do inode onde o arquivo começa, se não for encontrado retorna-se -1.

```
int ufuFS_seek(int ufuFD, char *fn, ufuFS_INODE *i_node) //PROCURA O ARQUIVO PELO NOME
{
    int i;

    lseek(ufuFD, sb.bloco_ini_inode*sb.tam_bloco, SEEK_SET);

    for(i = 0; i < sb.qtd_inode; i++)
    {
        read(ufuFD, i_node, sizeof(ufuFS_INODE));
        if(!i_node->nome[0]) continue;
        if (strcmp(i_node->nome, fn) == 0) return i;
    }
    return -1; //Retorna -1 se não achou o arquivo
}
```

4.4.6 - Close

A principal idéia da funcionalidade close é tirar toda a informação auxiliar do sistema de arquivo, que alocamos e carregamos, da memória principal. Depois de usarmos o comando exit no ufuFS_shell passamos por parâmetro na função finaliza_shell o file descriptor carregado pela função open. Na função finaliza_shell usaremos a função free para indicarmos para o sistema operacional que os endereços alocados pelo used_blocks podem ser usados por outros programas. Deste modo tiramos todas as informações carregadas na memória principal pelo inicializa_shell.

```
void finaliza_shell(int ufuFD)
{
    free(used_blocks);
}
```

4.4.7 - Delete

A funcionalidade delete está implementada com o nome ufuFS_remove e ela recebe como parâmetro o file descriptor e o nome do arquivo que gostaríamos de deletar. O objetivo é zerar os metadados e os blocos para quando formos colocar outro arquivo o sistema posse a notar que nada está escrito lá.

Em primeiro lugar devemos checar se tal arquivo que gostaríamos de deletar realmente existe usando a função ufuFS_seek e assim obtermos o inode do arquivo.

Depois de obtido o inode andamos pelos seus blocos zerando cada espaço ocupado. Quando terminado a operação devemos zerar uma variável struct inode auxiliar com a função memset e depois copiarmos ela no pendrive por meio do file descriptor para que deste modo sobrescrever os metadados do arquivo desejado.

```
for(block_i = 0; block_i < ufuFS_DATA_BLOCK; block_i++)
{
    if(!i_node.blocos[block_i])
    {
        break;
    }
    put_data_block(ufuFD, i_node.blocos[block_i]); //Libera os blocos usados pelo inode
}

memset(&i_node, 0, sizeof(ufuFS_INODE)); //Libera o inode

lseek(ufuFD, sb.bloco_ini_inode * sb.tam_bloco + i * sb.tam_inode,SEEK_SET);

write(ufuFD, &i_node, sizeof(ufuFS_INODE)); //salva as alterações
```

4.4.8 - copy_to_so

A função copy_to_so recebe como parâmetro de entrada o file descriptor, e o nome do arquivo do pendrive a ser copiado para o computador. Depois de verificar se o arquivo existe no pendrive, é requerido o endereço do local onde deve ser copiado no computador, incluindo no final do endereço o novo nome do arquivo.

Feito isso, chama-se a função ufuFS_read_file. Essa função recebe como parâmetro de entrada o file descriptor, o nome do arquivo e o caminho onde o arquivo deve ser gravado no computador.

Dentro da função, usa-se a system call open passando como parâmetro o caminho referenciado no computador, e mais três parâmetros, o O_RDWR, O_CREAT e 0777.

- O O_RDWR significa que poderá ser feito tanto leitura como gravação no arquivo.
- O O_CREAT significa que caso o arquivo especificado no primeiro parâmetro não exista o mesmo é criado.
- O 0777 são permissões de usuário.

```
int file;
file = open(path, O_RDWR|O_CREAT, 0777); //Cria arquivo no SO
if(file == -1){
    printf("Erro na criação do arquivo!\n");
    return;
}
```

Em seguida, o arquivo a ser copiado é buscado no pendrive e é feito o processo de leitura e escrita similar ao ufuFS_read descrito acima, porém ao invés do write ser direcionado para o shell o write é direcionado para o arquivo no computador.

```
write(file, block, to_read);
```

Por fim, utiliza-se a system call close para fechar o arquivo do computador.

4.4.9 - copy_to_usb

A função copy_to_usb recebe como parâmetro de entrada o file descriptor, e o caminho do arquivo a ser copiado do computador. Em seguida, pede-se ao usuário para renomear o arquivo que deseja salvar no pendrive. Depois disso, chama-se a função ufuFS_write_to_usb, que recebe o file descriptor, o novo nome e o caminho do arquivo. Essa função implementa a funcionalidade principal deste comando.

```c
void ufuFS_copy_to_usb(int ufuFD, char *path)
{
    char new_name[ufuFS_FILENAME_LEN];

    printf("Digite o (novo)nome do arquivo: ");
    scanf("%s",new_name);

    ufuFS_write_to_file(ufuFD, new_name, path);
}
```

A função ufuFS_write_to_usb recebe como parâmetro, o file descriptor, o novo nome do arquivo a ser copiado no pendrive e o caminho referenciado no computador. Em seguida abre-se este arquivo com a system call open com permissão para leitura e escrita. Após a abertura do arquivo do computador, é necessário criar o arquivo no pendrive utilizando a função ufuFS_create com o nome definido pelo usuário.

Tendo os dois arquivos já prontos, faz-se a leitura do arquivo do computador com a system call read e logo após faz-se a gravação no arquivo do pendrive com a system call write.

```c
void ufuFS_write_to_file(int ufuFD, char *fn, char *path)
{
    int i, cur_read_i, to_read, cur_read, total_size, block_i, free_i;
    ufuFS_INODE i_node;

    int file;
    file = open(path, O_RDONLY); /*Abre o arquivo do computador*/

    if(file == -1){
        printf("Erro ao abrir arquivo!\n");
        return;
    }

    ufuFS_create(ufuFD, fn);
```

4.4.10 - List

Na funcionalidade List devemos iterar por todos os metadados dos arquivos e assim apresentar as informações de cada arquivo para o usuário.

A função ufuFS_list recebe como parâmetro o file descriptor do arquivo aberto relacionado ao pen drive. Em seguida, o primeiro passo deve ser apontar o ponteiro do file descriptor para o início da tabela de inode. Feito isso criamos uma estrutura de repetição para andar por cada inode mostrando na tela todas suas informações guardadas, caso exista um arquivo referenciado pelo inode.

```c
void ufuFS_list (int ufuFD)
{
    int i;
    ufuFS_INODE i_node;

    lseek(ufuFD, sb.bloco_ini_inode * sb.tam_bloco, SEEK_SET);

    for(i = 0; i < sb.qtd_inode; i++)
    {
        read(ufuFD, &i_node, sizeof(ufuFS_INODE));

        if(!i_node.nome[0]) continue;

            printf("%-15s  %10d bytes Criacao: %s ",
            i_node.nome, i_node.tamanho,
            ctime((time_t *)&i_node.horaCriacao)
            );

            printf("Ultimo acesso: %s",ctime((time_t *)&i_node.horaModificacao));
    }
}
```

4.5 - Bibliotecas utilizadas

Para a realização do trabalho foi necessário a utilização de algumas bibliotecas da linguagem C. Algumas delas foram:

- time.h
    - Arquivo cabeçalho que fornece funções, macros e definições para manipulação de datas e horas. Foi utilizada para obter horário de criação e modificação dos arquivos no ufuFS.

- sys/types.h
    - Contém o número derivado de tipos básicos como off_t que é utilizados por arquivos e offsets para ler os arquivos, como também time_t para ver o tempo em segundos.

- sys/ioctl.h
    - Cabeçalho normalmente utilizado para controlar inputs e outputs para interação de usuário com o hardware. Foi usado para obter informações da entrada do Pen Drive.

4.6 - Diretivas de Compilação

As diretivas de compilação são os comandos utilizados pelo pré-processador e permite que o programador modifique a compilação. As diretivas usadas foram:

- #include
    - A organização de constantes e definições de macros pode ser feita dentro de include files. Depois de feita, necessita da diretiva “#include” para adicionar tais constantes e macros dentro de source files. Include files são úteis para incorporar declaração de variáveis externas e tipos de dados complexos. O uso no trabalho é imprescindível pois precisa-se do suporte de tais estruturas e funções para o funcionamento correto do programa.
- #define
    - A diretiva define leva o compilador a substituir uma token-string (uma token-string é um símbolo, constante ou declaração) para cada ocorrência do identificador no source file. A principal funcionalidade é levar a praticidade ao programador não deixando tarefas com repetições longas e complexas.
- #ifdef / #ifndef
    - Usa-se as diretivas “#ifdef” e “#ifndef”em qualquer lugar que a diretiva “#if” pode ser usada. A declaração do identificador “#ifdef” é equivalente ao “#if 1” quando o identificador foi declarado ou definido. É equivalente ao “#if 0”quando o identificador não foi definido. Essas diretivas checam apenas as presenças ou ausência dos identificadores definidos com “#define”. A diretiva “#ifndef” checa o oposto da diretiva “#ifdef”.

4 - Considerações Finais

4.1 - Prós e Contras

- **Prós:**
    - Fácil implementação (estruturas simples)
    - Blocos de tamanho pequeno (512 bytes), logo, evita desperdício de memória ao alocar um arquivo;
    - Formatação de acordo com a capacidade de armazenamento do pendrive;
- **Contras:**
    - Sem implementação de mecanismos de segurança;
    - Inicialização do Shell e Formatação não otimizada;
    - Não implementa permissões de usuário;
    - Não implementa técnicas de confiabilidade;
    - Fragmentação Externa, pois aloca blocos de forma contígua na memória;

4.2 - Conclusão

O desenvolvimento do presente Trabalho de Conclusão de Disciplina possibilitou um melhor entendimento de conceitos de Sistemas Operacionais, mais especificamente sobre conceitos relacionados à Sistemas de Arquivos.

Além disso, também permitiu um conhecimento mais aprofundado em programação e sua relação com o sistema operacional, principalmente no que se refere à utilização de system calls, acesso a memória, manipulação de arquivos e partições. Com isso, colocamos em prática conceitos vistos em sala de aula, o que ajudou a consolidar o conhecimento adquirido.

Além disso, a implementação do Sistema de Arquivos nos fez perceber características que não devem ser desprezadas, como por exemplo, otimização do código, segurança e confiabilidade.

5 - Bibliografia

- **TANENBAUM, A. S. , Sistemas Operacionais Modernos. Quarta Edição, Pearson Education do Brasil, 2016.**
- **https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/**
- **http://www.ntfs.com/hard-disk-basics.htm**
- **https://docs.microsoft.com/pt-br/cpp/preprocessor/hash-include-directive-c-cpp?view=vs-2019**
- **https://docs.microsoft.com/pt-br/cpp/preprocessor/hash-define-directive-c-cpp?view=vs-2019**
- **https://docs.microsoft.com/pt-br/cpp/preprocessor/hash-ifdef-and-hash-ifndef-directives-c-cpp?view=vs-2019**