T2 - INE5412

Thiago Kenzo Takahashi (222215402)
Pedro Costa Casotti (22200374)
Marcos Vinicius

1. Introdução

O SimpleFS é um sistema de arquivos simplificado desenvolvido para demonstrar conceitos fundamentais de gerenciamento de arquivos em sistemas operacionais. Ele suporta operações básicas, como formatação, montagem, leitura, escrita, criação e exclusão de arquivos, utilizando estruturas como superbloco, inodes e blocos diretos/indiretos.

2. Estrutura de Dados e Organização do Sistema

2.1 Superbloco

```
class fs_superblock {
public:
    unsigned int magic;
    int nblocks;
    int ninodeblocks;
    int ninodes;
};
```

2.2 Inodes

```
class fs_inode {
public:
    int isvalid;
    int size;
    int direct[POINTERS_PER_INODE];
    int indirect;
};
```

2.3 Bitmap de Blocos Livres

```
// Inicializa o bitmap para rastrear os blocos usados
block_bitmap = new bool[block.super.nblocks];
std::fill(block_bitmap, block_bitmap + block.super.nblocks, false);

// Marca os blocos reservados (superbloco e inodes) como usados
for (int i = 0; i <= block.super.ninodeblocks; ++i) {
    block_bitmap[i] = true;
}

// Verifica os inodes e marca os blocos associados como usados
for (int i = 0; i < block.super.ninodeblocks; ++i) {
    disk->read(i + 1, block.data);
    for (int j = 0; j < INODES_PER_BLOCK; ++j) {
        fs_inode &inode = block.inode[j];
        if (inode.isvalid) {
            // Marca blocos diretos como usados
            for (int k = 0; k < POINTERS_PER_INODE; ++k) {
                if (inode.direct[k]) {
                    block_bitmap[inode.direct[k]] = true;
                }
            }
            // Marca blocos indiretos como usados
            if (inode.indirect) {
                block_bitmap[inode.indirect] = true;

                fs_block indirect_block;
                disk->read(inode.indirect, indirect_block.data);
                for (int k = 0; k < POINTERS_PER_BLOCK; ++k) {
                    if (indirect_block.pointers[k]) {
                        block_bitmap[indirect_block.pointers[k]] = true;
                    }
                }
            }
        }
    }
}
```

O bitmap de blocos livres é inicializado dentro da função fs_mount( ), ele é inicia com todos os bits marcados como falso, e depois coloca como true os blocos usados pelos inodes e superblocos, e em seguida confere os inodes, e coloca true nos blocos diretos e indiretos usados.

2.4 Blocos Diretos e Indiretos

```
// Escreve os dados enquanto houver bytes a serem processados
while (remaining_length > 0) {
    // Calcula o índice do bloco e a posição dentro do bloco
    int block_index = current_offset / Disk::DISK_BLOCK_SIZE;
    int block_offset = current_offset % Disk::DISK_BLOCK_SIZE;

    int block_num = 0; // Número do bloco onde escreveremos os dados

    // Lida com blocos diretos
    if (block_index < POINTERS_PER_INODE) {
        // Se o bloco direto ainda não foi alocado, aloque-o
        if (inode.direct[block_index] == 0) {
            inode.direct[block_index] = allocate_block();
            if (inode.direct[block_index] == 0) return total_bytes_written; // Falha ao alocar bloco
        }
        block_num = inode.direct[block_index];
    } else {
        // Caso os blocos diretos acabem, usa blocos indiretos
        if (inode.indirect == 0) { // Se o bloco indireto não existe, aloque-o
            inode.indirect = allocate_block();
            if (inode.indirect == 0) return total_bytes_written; // Falha ao alocar bloco indireto

            // Inicializa o bloco indireto com 0
            fs_block indirect_block = {};
            disk->write(inode.indirect, indirect_block.data);
        }

        // Lê o bloco indireto do disco
        fs_block indirect_block;
        disk->read(inode.indirect, indirect_block.data);

        // Calcula o índice dentro do bloco indireto
        int indirect_index = block_index - POINTERS_PER_INODE;

        // Aloca um novo bloco de dados, se necessário
        if (indirect_block.pointers[indirect_index] == 0) {
            indirect_block.pointers[indirect_index] = allocate_block();
            if (indirect_block.pointers[indirect_index] == 0) return total_bytes_written; // Falha ao alocar

            // Atualiza o bloco indireto no disco
            disk->write(inode.indirect, indirect_block.data);
        }
        block_num = indirect_block.pointers[indirect_index];
    }
```

A função fs_write( ) tenta escrever os dados nos blocos diretos, caso o tamanho exceda a quantidade(5) de ponteiros diretos para armazenar os blocos, é necessário a utilização de blocos indiretos.

3. Implementação das Funcionalidades do INE5412_FS

3.1 fs_format( )

```
// Formata o disco inicializando o sistema de arquivos
int INE5412_FS::fs_format() {
    fs_block block = {};
    block.super.magic = FS_MAGIC;
    block.super.nblocks = disk->size();
    block.super.ninodeblocks = (block.super.nblocks / 10) + 1;
    block.super.ninodes = block.super.ninodeblocks * INODES_PER_BLOCK;

    disk->write(0, block.data); // Escreve o superbloco no bloco 0

    // Inicializa blocos de inodes
    for (int i = 1; i <= block.super.ninodeblocks; ++i) {
        memset(block.data, 0, Disk::DISK_BLOCK_SIZE); // Zera o conteúdo do bloco
        disk->write(i, block.data);
    }

    // mounted = 0;

    return 1;

}
```

O disco é inicializado, configurando o superbloco e zerando os blocos de inodes. O superbloco é escrito no bloco 0 e os blocos dos inodes nos blocos seguintes.

3.2 fs_mount ( )

```
int INE5412_FS::fs_mount() {
    fs_block block;
    disk->read(0, block.data); // Lê o superbloco

    if (block.super.magic != FS_MAGIC) return 0;

    // Inicializa o bitmap para rastrear os blocos usados
    block_bitmap = new bool[block.super.nblocks];
    std::fill(block_bitmap, block_bitmap + block.super.nblocks, false);

    // Marca os blocos reservados (superbloco e inodes) como usados
    for (int i = 0; i <= block.super.ninodeblocks; ++i) {
        block_bitmap[i] = true;
    }

    // Verifica os inodes e marca os blocos associados como usados
    for (int i = 0; i < block.super.ninodeblocks; ++i) {
        disk->read(i + 1, block.data);
        for (int j = 0; j < INODES_PER_BLOCK; ++j) {
            fs_inode &inode = block.inode[j];
            if (inode.isvalid) {
                // Marca blocos diretos como usados
                for (int k = 0; k < POINTERS_PER_INODE; ++k) {
                    if (inode.direct[k]) {
                        block_bitmap[inode.direct[k]] = true;
                    }
                }
                // Marca blocos indiretos como usados
                if (inode.indirect) {
                    block_bitmap[inode.indirect] = true;

                    fs_block indirect_block;
                    disk->read(inode.indirect, indirect_block.data);
                    for (int k = 0; k < POINTERS_PER_BLOCK; ++k) {
                        if (indirect_block.pointers[k]) {
                            block_bitmap[indirect_block.pointers[k]] = true;
                        }
                    }
                }
            }
        }
    }

    mounted = true; // Marca o sistema como montado
    return 1;
}
```

Verifica se o disco contém um sistema de arquivos válido e inicializa o bitmap de blocos livres.

3.3 fs_create( )

```
// Cria um novo inode
int INE5412_FS::fs_create() {
    if (!mounted) return -1;

    // Ler o superbloco
    fs_block block;
    disk->read(0, block.data);
    int ninodeblocks = block.super.ninodeblocks;  // Pega a quantidade de blocos para inodes

    for (int i = 0; i < ninodeblocks; ++i) {
        disk->read(i + 1, block.data); // Le a partir do bloco 1
        for (int j = 0; j < INODES_PER_BLOCK; ++j) {
            if (!block.inode[j].isvalid) {
                block.inode[j].isvalid = 1; // Marca o inode como válido
                block.inode[j].size = 0; // Inicializa o tamanho como 0
                memset(block.inode[j].direct, 0, sizeof(block.inode[j].direct)); // Zera os ponteiros diretos
                block.inode[j].indirect = 0; // Zera o ponteiro indireto

                disk->write(i + 1, block.data); // Atualiza o bloco de inodes no disco
                return i * INODES_PER_BLOCK + j;
            }
        }
    }

    return -1;
}
```

Busca por um inode livre e o inicializa, com os valores padrão de: isvalid = 1, tamanho como 0 e zera os ponteiros diretos e indiretos.

3.4 fs_delete( )

```
// Deleta um inode e libera seus blocos
int INE5412_FS::fs_delete(int inumber) {
    if (!mounted) return 0;

    fs_inode inode;
    inode_load(inumber, &inode); // Carrega o inode indicado
    if (!inode.isvalid) return 0;

    // Itera sobre os ponteiros diretos
    for (int i = 0; i < POINTERS_PER_INODE; ++i) {
        if (inode.direct[i]) free_block(inode.direct[i]); // se estiveer alocado, marca o bloco como livre
    }

    if (inode.indirect) {
        fs_block indirect_block;
        disk->read(inode.indirect, indirect_block.data); // Lê o bloco indireto
        for (int i = 0; i < POINTERS_PER_BLOCK; ++i) {
            if (indirect_block.pointers[i]) free_block(indirect_block.pointers[i]); // Marca como livre todos os blocos apontados
        }
        free_block(inode.indirect); // Marca como livre o próprio bloco indireto
    }

    inode.isvalid = 0;
    inode_save(inumber, &inode);
    return 1;
}
```

Libera todos os blocos associados a um inode

3.5 fs_read( )

```
// Lê dados de um inode
int INE5412_FS::fs_read(int inumber, char *data, int length, int offset) {
    if (!mounted) return 0;

    fs_inode inode;
    inode_load(inumber, &inode);
    if (!inode.isvalid) return 0;

    int bytes_read = 0;
    while (length > 0) {
        int block_offset = offset / Disk::DISK_BLOCK_SIZE;
        int block_pos = offset % Disk::DISK_BLOCK_SIZE;
        int to_read = std::min(length, Disk::DISK_BLOCK_SIZE - block_pos);

        fs_block block;
        if (block_offset < POINTERS_PER_INODE && inode.direct[block_offset]) {
            disk->read(inode.direct[block_offset], block.data);
        } else if (inode.indirect) {
            fs_block indirect_block;
            disk->read(inode.indirect, indirect_block.data);
            if (indirect_block.pointers[block_offset - POINTERS_PER_INODE]) {
                disk->read(indirect_block.pointers[block_offset - POINTERS_PER_INODE], block.data);
            }
        } else {
            break;
        }

        memcpy(data + bytes_read, block.data + block_pos, to_read);
        bytes_read += to_read;
        offset += to_read;
        length -= to_read;
    }

    return bytes_read;
}
```

Lê dado de tamanho ‘length’ e posição ‘offset’ do inode requisitado, retorna a quantidade de bytes lidos.

3.6 fs_write( )

```
int INE5412_FS::fs_write(int inumber, const char *data, int length, int offset) {
    if (!mounted) return 0;

    fs_inode inode;
    inode_load(inumber, &inode);

    if (!inode.isvalid) return 0;

    int total_bytes_written = 0;
    int current_offset = offset;
    int remaining_length = length;

    while (remaining_length > 0) {
        int block_index = current_offset / Disk::DISK_BLOCK_SIZE;
        int block_offset = current_offset % Disk::DISK_BLOCK_SIZE;

        int block_num = 0;
        if (block_index < POINTERS_PER_INODE) {
            if (inode.direct[block_index] == 0) {
                inode.direct[block_index] = allocate_block();
                if (inode.direct[block_index] == 0) return total_bytes_written;
            }
            block_num = inode.direct[block_index];
        } else {
            if (inode.indirect == 0) {
                inode.indirect = allocate_block();
                if (inode.indirect == 0) return total_bytes_written;

                fs_block indirect_block = {};
                disk->write(inode.indirect, indirect_block.data);
            }
            fs_block indirect_block;
            disk->read(inode.indirect, indirect_block.data);

            int indirect_index = block_index - POINTERS_PER_INODE;
            if (indirect_block.pointers[indirect_index] == 0) {
                indirect_block.pointers[indirect_index] = allocate_block();
                if (indirect_block.pointers[indirect_index] == 0) return total_bytes_written;

                disk->write(inode.indirect, indirect_block.data);
            }
            block_num = indirect_block.pointers[indirect_index];
        }

        fs_block block;
        disk->read(block_num, block.data);

        int bytes_to_write = std::min(remaining_length, Disk::DISK_BLOCK_SIZE - block_offset);
        memcpy(block.data + block_offset, data + total_bytes_written, bytes_to_write);

        disk->write(block_num, block.data);

        total_bytes_written += bytes_to_write;
        current_offset += bytes_to_write;
        remaining_length -= bytes_to_write;
    }

    inode.size = std::max(inode.size, offset + length);
    inode_save(inumber, &inode);

    return total_bytes_written;
```

Escreve dado de tamanho ‘length’ e posição ‘offset’ no inode requisitado, retorna a quantidade de bytes escritos.

3.7 fs_debug( )

```cpp
// Função de depuração que exibe o estado do sistema de arquivos
void INE5412_FS::fs_debug() {
    fs_block block;
    disk->read(0, block.data); // Lê o superbloco

    std::cout << "superblock:" << std::endl;
    if (block.super.magic == FS_MAGIC) {
        std::cout << "    magic number is valid" << std::endl;
    } else {
        std::cout << "    invalid magic number" << std::endl;
        return;
    }

    // Informações do superbloco
    std::cout << "    " << block.super.nblocks << " blocks on disk" << std::endl;
    std::cout << "    " << block.super.ninodeblocks << " blocks for inodes" << std::endl;
    std::cout << "    " << block.super.ninodes << " inodes total" << std::endl;

    // Itera pelos blocos de inodes
    for (int i = 0; i < block.super.ninodeblocks; ++i) {
        disk->read(i + 1, block.data);
        for (int j = 0; j < INODES_PER_BLOCK; ++j) {
            fs_inode &inode = block.inode[j];
            if (inode.isvalid) {
                std::cout << "inode " << (i * INODES_PER_BLOCK + j) << ":" << std::endl;
                std::cout << "    size: " << inode.size << " bytes" << std::endl;
                std::cout << "    direct blocks:";
                for (int k = 0; k < POINTERS_PER_INODE; ++k) {
                    if (inode.direct[k]) std::cout << " " << inode.direct[k];
                }
                std::cout << std::endl;
                if (inode.indirect) {
                    std::cout << "    indirect block: " << inode.indirect << std::endl;
                    fs_block indirect_block;
                    disk->read(inode.indirect, indirect_block.data);
                    std::cout << "    indirect data blocks:";
                    for (int k = 0; k < POINTERS_PER_BLOCK; ++k) {
                        if (indirect_block.pointers[k]) std::cout << " " << indirect_block.pointers[k];
                    }
                    std::cout << std::endl;
                }
            }
        }
    }
}
```

Imprime informações sobre o estado dos inodes para depuração.

4. Implementação das funcionalidades do Disk

4.1 Disk

```cpp
// Construtor da classe Disk
// Inicializa o arquivo de disco e configura o número de blocos.
Disk::Disk(const char *filename, int nblocks) : nblocks(nblocks) {
    // Abre o arquivo de disco no modo "append + binário"
    diskfile = fopen(filename, "a+b");
    if (!diskfile) throw runtime_error("Falhou para abrir o disk file"); // Erro se o arquivo não for aberto

    // Fecha temporariamente o arquivo
    fclose(diskfile);

    // Ajusta o tamanho do arquivo para o tamanho total necessário (número de blocos * tamanho de um bloco)
    if (truncate(filename, DISK_BLOCK_SIZE * nblocks) != 0) {
        throw runtime_error("Falhou para set file size"); // Erro se o ajuste falhar
    }

    // Reabre o arquivo no modo "leitura + escrita binária"
    diskfile = fopen(filename, "r+b");
    if (!diskfile) throw runtime_error("falhou para reabrir disk file"); // Erro se o arquivo não puder ser reaberto
}
```

Construtor de Disk com os argumentos do arquivo da imagem de disco e o número de blocos.

4.2 size()

```cpp
// Retorna o número de blocos do disco
int Disk::size() { return nblocks; }
```

Retorna o número de blocos.

4.3 sanity_check()

```cpp
void Disk::sanity_check(int blocknum, const void *data) {
    if (!data || blocknum < 0 || blocknum >= nblocks) {
        std::cerr << "Sanity check failed! Blocknum: " << blocknum
                  << ", Data is null: " << (data == nullptr)
                  << ", Total blocks: " << nblocks << std::endl;
        throw std::runtime_error("Disk sanity check falhou");
    }
}
```

Verifica os parâmetros antes de realizar as operações do disco.

4.4 read()

```
// Lê dados de um bloco do disco
void Disk::read(int blocknum, char *data) {
    sanity_check(blocknum, data); // Verifica os parâmetros
    fseek(diskfile, blocknum * DISK_BLOCK_SIZE, SEEK_SET); // Move o ponteiro do arquivo para o bloco correspondente
    fread(data, DISK_BLOCK_SIZE, 1, diskfile); // Lê um bloco de dados do arquivo
}
```

4.5 write()

```
// Escreve dados em um bloco do disco
void Disk::write(int blocknum, const char *data) {
    sanity_check(blocknum, data); // Verifica os parâmetros
    fseek(diskfile, blocknum * DISK_BLOCK_SIZE, SEEK_SET); // Move o ponteiro do arquivo para o bloco correspondente
    fwrite(data, DISK_BLOCK_SIZE, 1, diskfile); // Escreve um bloco de dados no arquivo
}
```

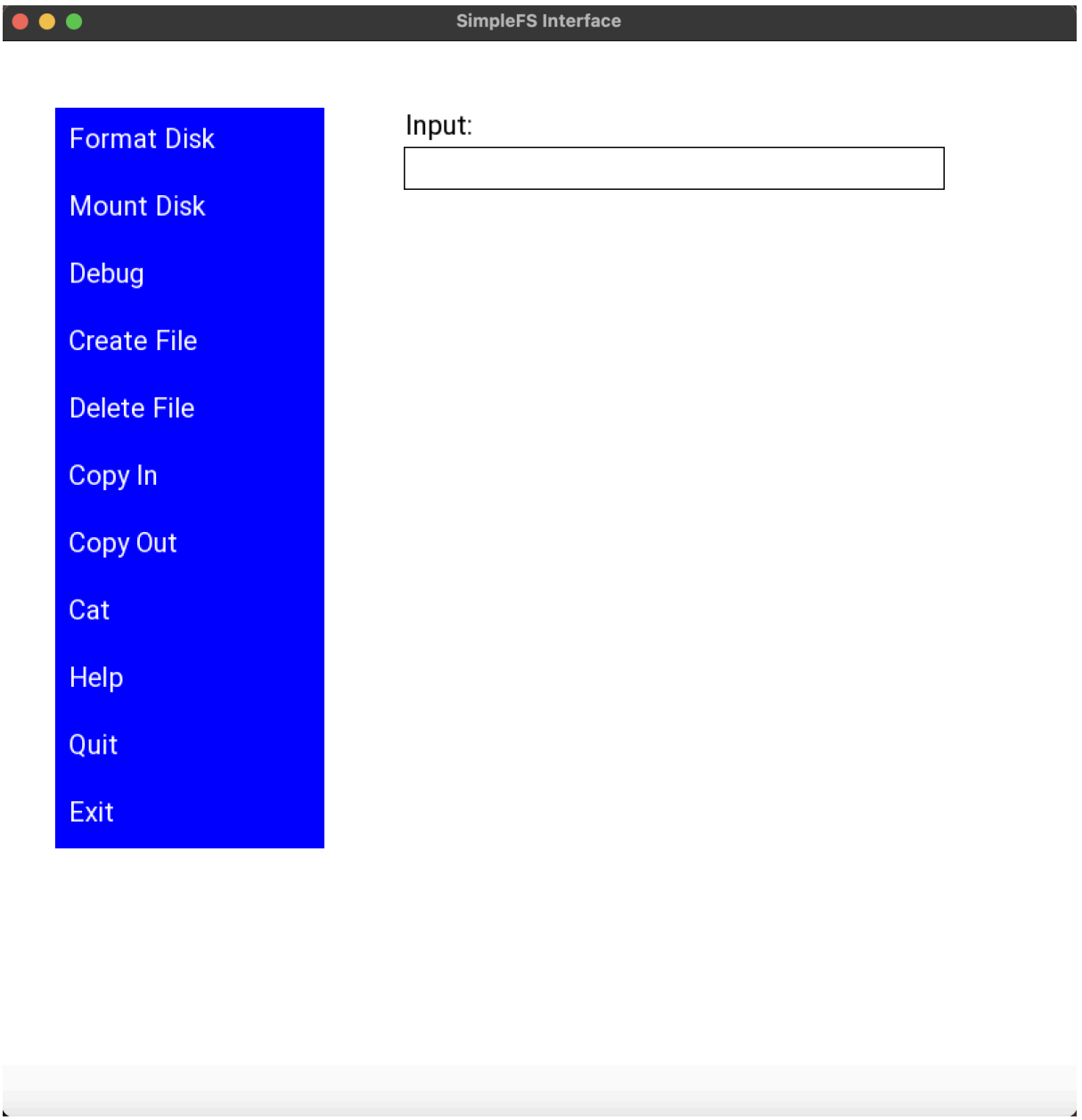
4.6 close()

```
// Fecha o arquivo de disco
void Disk::close() {
    if (diskfile) fclose(diskfile); // Fecha o arquivo, se estiver aberto
}
```

5. Interface

```
thiagotakahashi@Thiagos-MacBook-Air codigo_em_c++ % ./bin/simplefs resources/image.200 200
Select mode of operation:
1. Command Line Interface (CLI)
2. Graphical User Interface (GUI)
Enter your choice (1 or 2): 2
```

Quando rodamos o programa, aparece as opções de escolha entre utilizar o terminal ou a interface



Quando selecionada a interface, aparece essa tela, com os botões para cada função e a caixa de texto dos inputs, quando clicado uma função que necessita input, será necessário colocar os argumentos.