BRENO WILLE BEZERRA CORREIA

PE3022447

ELABORAÇÃO DE SISTEMA DE ARQUIVOS:
BrenimFS

PRESIDENTE EPITÁCIO - SP

2025

ELABORAÇÃO DE SISTEMA DE ARQUIVOS:

BrenimFS

Trabalho apresentado à disciplina de Estrutura de Dados 2 do Instituto Federal de Educação, Ciência e Tecnologia Campus Presidente Epitácio como nota parcial para aprovação na disciplina do curso Bacharelado em Ciência da Computação, turma 2023.
Professor: Marcelo Roberto Zorzan.

PRESIDENTE EPITÁCIO - SP

2025

SUMÁRIO

1. INTRODUÇÃO

Este documento tem como objetivo detalhar o projeto e implementação do BrenimFS, um sistema de arquivos simulado em C. O sistema simula um EXT3/EXT4 e portanto gerencia uma estrutura de inodes, blocos de dados e diretórios de forma persistente em disco e é acessado através de uma shell interativa com os comandos básicos do Sistema Operacional Linux.

2. ESTRUTURAS DE DADOS EM DISCO

Além dos arquivos contendo toda lógica do Sistema de Arquivos, o projeto consiste em 4 arquivos binários no formato .dat que organizam (em disco) a forma como os dados são tratados e guardados pelo Sistema Operacional, encontrado na pasta “FS” (os arquivos são gerados após a montagem do sistema).

2.1. SuperBlock.dat

O arquivo “SuperBlock.dat” atua como o cabeçalho mestre do sistema de arquivos. É um arquivo que armazena uma única *struct* representando o super bloco. Ele é lido durante a montagem (através da função “montar_fs”) e contém metadados globais que definem as características da partição, incluindo o nome do sistema de arquivos, o tamanho total da partição, o tamanho de cada bloco, o número total de inodes, o número total de bloco de dados e o índice do inode raiz.

2.2. Inodes.dat

O arquivo “Inode.dat” é o índice central de todos os arquivos e diretórios. Ele é implementado como um array contíguo de structs INODE de tamanho fixo, onde cada INODE representa um arquivo ou diretório.

O índice de um inode neste array (de 0 a “total_inodes” - 1) é o “num_inode” usado pelas entradas de diretório para referenciá-lo. Esta estrutura de array permite acesso direto (aleatório) a qualquer inode em tempo O(1) usando “fseek” para a posição “num_inode * SIZEOF(inode)”.

Cada Inode guarda os metadados de um arquivo, contendo: o tipo do arquivo, o tamanho do arquivo, a quantidade de blocos que aquele arquivo ocupa e os blocos (indicados por ponteiros) onde o arquivo está localizado.

2.3. Blocks.dat

Este arquivo representa o disco bruto. É um arquivo único, pré-alocado no tamanho total da partição, que é tratado como um array contíguo de blocos de dados. Cada bloco é acessado pelo seu índice (de 0 a “total_blocos” - 1), e a posição de qualquer bloco N no arquivo é calculado como “N*tamanho_bloco”. O conteúdo deste arquivo é o dado bruto dos arquivos e as estruturas de entrada de diretórios.

2.4. Bitmap.dat

Este arquivo implementa o gerenciamento de espaço livre. Ele armazena um Bitmap (Mapa de Bits), onde cada bit corresponde diretamente a um bloco de dados no Blocks.dat. Um bit 0 indica que o bloco está livre e 1 que o mesmo está em uso. A estrutura é carregada inteiramente na memória RAM durante a montagem para garantir operações de alocação e liberação de forma eficiente.

3. ARQUITETURA DE CÓDIGO-FONTE E MÓDULOS

O projeto foi dividido em cinco cabeçalhos (.h) principais, que contêm toda a lógica do sistema de arquivos, e um único arquivo.c, que atua como o ponto de entrada da shell.

3.1. Estruturas.h

Este módulo guarda as definições de dados que são compartilhados entre todos os outros módulos. Temos as Estruturas de Disco (Super_Bloco, Inode e Entrada_Diretorio) e uma estrutura de memória (Sistema_Arquivos), que age como o contexto do sistema de arquivos enquanto ele está montado.

3.2. Bitmap.h

Este módulo contém funções para manipulação *bitwise* do mapa de espaço livre. Ele não tem conhecimento sobre inodes ou arquivos, apenas opera em um array de *bytes.*

3.3. Fs_nucleo.h

Este módulo implementa a camada de I/O e a lógica de baixo nível para interagir com os arquivos.dat. Inclui estruturas.h e bitmap.h e implementa as funções principais do sistema de arquivos (desde sua montagem, desmontagem e funções de alocação e liberação da memória).

3.4. Fs_utils.h

Este módulo implementa as funções auxiliares utilizadas pelas funções implementadas no módulo de Comandos. O propósito deste módulo é deixar apenas as funções principais dos comandos em Comando.

3.5. Comandos.h

Este módulo é responsável por isolar as funções de comandos exercidos pela shell (ls, mkdir, cd, cat, touch, pwd, rm, stat e help) e implementar as lógicas de alto nível para cada comando.

4. ESTRATÉGIAS DE EFICIÊNCIA E DESEMPENHO

Para atender aos requisitos de funcionalidade e alto desempenho, o sistema emprega duas estratégias. A primeira foca na gestão de espaço livre por meio de um Mapa de Bits (Bitmap), e a segunda implementa uma estrutura de diretórios baseada em Tabela Hash para eliminar a busca sequencial O(n).

4.1. Gerenciamento de Espaço Livre (Bitmap)

O arquivo bitmap.dat é um *array* de bits onde cada bit mapeia diretamente um bloco de dados (0=livre, 1=usado). Para garantir a máxima performance, este bitmap é carregado integralmente na memória RAM (mapa_espaco_livre) durante a montagem do sistema de arquivos (montar_fs). Essa decisão otimiza a busca por um novo bloco (encontrar_bloco_livre) para uma varredura rápida em memória. Consequentemente, as operações de alocação (marcar_bit_usado) e liberação (marcar_bit_livre) são computacionalmente triviais, executando em tempo O(1) com manipulação *bitwise*.

4.2. Tabela Hash para Diretórios

O sistema utiliza uma Tabela Hash com Encadeamento Separado para estruturar seus diretórios. Cada bloco de 128 bytes de um diretório (tipo = 'd') é interpretado como uma estrutura de hash, onde os bytes iniciais atuam como "cabeçalhos" das listas encadeadas. O restante do bloco armazena as ENTRADA_DIRETORIO como nós nessas listas.

A função hash_djb2 é utilizada para determinar a lista (ou "cabeça") correta. Para resolver a ambiguidade com o *inode* 0 (raiz), que poderia ser confundido com um *slot* vazio, definiu-se o valor INODE_NULO (0xFFFF) para marcar entradas não utilizadas. Esta arquitetura reduz o tempo médio de busca (buscar_inode_por_nome) para O(1+k) (onde 'k' é o tamanho da cadeia) e a inserção (encontrar_slot_livre) para O(1).

A principal vantagem desta abordagem reside na remoção (rm). Ao contrário da sondagem linear, que pode sofrer com o problema do "buraco", o encadeamento separado permite que a remoção (remover_entrada_diretorio_pai) seja uma operação O(1) limpa e segura, que simplesmente religa ponteiros da lista sem degradar o desempenho futuro do hash.

5. INTERFACE DO SHELL E FUNCIONALIDADES DOS COMANDOS

A interação do usuário com o BrenimFS é feita através de uma shell interativa (implementada em main.c). A shell exibe um prompt no formato usuario@BrenimFS:caminho $ e aceita um conjunto de comandos que se dividem em duas categorias:

5.1. Comandos Embutidos

São comandos que não interagem com o sistema de arquivos, sendo gerenciados diretamente pela main:

- **help**: exibe um menu de ajuda com todos os comandos disponíveis e suas descrições.
- **exit / shutdown :** encerra o loop da shell. Antes de sair, o programa chama desmontar_fs (do fs_nucleo.h) para garantir que o bitmap (bitmap.dat) seja salvo em disco, persistindo todas as alterações da sessão.

5.2. Comandos do Sistema de Arquivos

São comandos que realizam operações de I/O no disco. Sua lógica de alto nível é implementada no comandos.h e foi extensivamente refatorada em sub-funções para maior clareza e manutenibilidade.

5.2.1. Comandos de Navegação e Listagem

- **ls:** Lista o conteúdo do diretório atual. A implementação coleta todas as entradas válidas do diretório (contar_e_coletar_entradas), armazena-as em um buffer temporário, ordena este buffer para exibição e, em seguida, imprime a lista formatada (ordenar_e_imprimir_entradas), garantindo uma saída alfabética para o usuário.
- **cd <caminho>:** Altera o diretório atual. A lógica é encapsulada na função buscar_diretorio, que valida o <caminho> (incluindo .., ., e subdiretórios) e verifica se o alvo é um diretório válido (tipo = 'd'). Em caso de sucesso, atualiza a variável num_inode_diretorio_atual no SISTEMA_ARQUIVOS.
- **pwd:** Imprime o caminho absoluto do diretório atual. Toda a lógica de subida recursiva (seguindo .. até a raiz e empilhando nomes) é gerenciada pela função obter_caminho_absoluto, que retorna uma string alocada dinamicamente contendo o caminho completo. A cmd_pwd então imprime essa string e libera a memória.

5.2.2. Comandos de Manipulação de Arquivos/Diretórios:

- **mkdir <nome>:** Cria um novo diretório. A operação foi refatorada em várias sub-funções:
    - buscar_duplicata_e_slot é chamada para verificar se o nome já existe e encontrar um local livre na lista encadeada do diretório pai.
    - recursos_novo_diretorio aloca o INODE e o bloco de dados necessários.
    - novo_diretorio_inode_bloco formata o novo bloco de dados com as entradas “.”e “..”.
    - adicionar_entrada_diretorio_pai insere a entrada do novo diretório na lista do pai.
- **touch <nome>:** Cria um novo arquivo. Primeiro, valida a disponibilidade do nome (similar ao mkdir). Em seguida, ler_conteudo_stdin é chamada para ler a entrada do usuário até CTRL+D. A função criar_arquivo_conteudo aloca os blocos de dados necessários e escreve o conteúdo neles, atualizando o INODE do novo arquivo. Finalmente, adicionar_entrada_diretorio_pai o insere no diretório.
- **cat <nome>:** Exibe o conteúdo de um arquivo. A função validar_e_obter_inode_arquivo primeiro garante que o <nome_arq> exista e seja um arquivo (tipo = 'f'). Se for válido, imprimir_conteudo_arquivo é chamada para iterar pelos ponteiros de bloco do INODE, lendo cada bloco e escrevendo seu conteúdo diretamente na saída padrão (stdout).
- **rm <nome>:** Remove um arquivo ou diretório. Operação refatorada em várias sub-funções:
    - encontrar_e_validar_alvo_rm localiza o inode do alvo e, se for um diretório, verifica se está vazio.
    - desalocar_recursos_inode é chamada para liberar todos os blocos de dados (via liberar_bloco_dados) e o próprio INODE (via liberar_inode).
    - remover_entrada_diretorio_pai remove a entrada da lista encadeada do diretório pai, religando os ponteiros da lista.

5.2.3. Comando de Informação:

- **stat:** Exibe o status de utilização do disco. A função varre o mapa_espaco_livre (em memória), conta o número de bits 0 (livres) e reporta o total de blocos livres e o espaço livre em bytes, além do tamanho do bloco.

6. LIMITAÇÕES

O projeto foi implementado para fins acadêmicos e portanto carrega consigo algumas limitações que foram impostas para facilitar seu desenvolvimento, sendo elas:

1. **Tamanho Máximo de Arquivo:** A limitação mais significativa é o tamanho máximo de um arquivo. A struct INODE foi implementada com um array fixo de 8 ponteiros de bloco diretos (PONTEIROS_BLOCO_INODE). Com blocos de 128 bytes, isso restringe qualquer arquivo a um tamanho máximo de 1024 bytes (1 KB).

2. **Tamanho Máximo de Diretório:** Da mesma forma, a implementação da tabela hash de diretório (com encadeamento separado) assume que toda a estrutura (a tabela de cabeçalhos e o pool de nós da lista) cabe dentro de um único bloco de 128 bytes. Isso limita o número de arquivos que um diretório pode conter. Se um diretório ficar "cheio", o sistema não poderá mais criar arquivos nele.

3. **Resolução de Caminho Simplificada:** A shell atual é capaz de lidar com nomes simples (arquivo.txt, breno), nomes relativos (., ..) e o caminho raiz (/). No entanto, ela não implementa um parser de caminho recursivo completo. Comandos como cd /breno/docs ou cat ../pasta/outro.txt não são suportados.

4. **Sistema de Usuário Único e Sem Permissões:** O BrenimFS opera como um sistema de usuário único. Não há conceito de propriedade de arquivo (usuário/grupo) ou permissões (leitura, escrita, execução). Qualquer usuário que executa a shell tem controle total sobre todos os arquivos. Além disso, o sistema não é seguro para concorrência; se duas instâncias da shell fossem executadas simultaneamente, elas poderiam corromper o bitmap.dat ao tentar alocar o mesmo bloco ao mesmo tempo.