INSTITUTO FEDERAL
São Paulo
Câmpus Presidente Epitácio

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO - IFSP CAMPUS PRESIDENTE EPITÁCIO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ALEXANDRE FERREIRA PEREIRA DE OLIVEIRA
PEDRO HENRIQUE DA SILVA LOPES

SISTEMA DE ARQUIVOS PEALFS

Este documento apresenta a documentação técnica completa do PEALFS (PEAL FileSystem), um sistema de arquivos implementado em linguagem C como requisito da disciplina de Sistemas Operacionais 2.

Discente: _____
Alexandre Ferreira Pereira de Oliveira

Discente: _____
Pedro Henrique da Silva Lopes

PRESIDENTE EPITÁCIO

2025.2

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo – IFSP
Rua Pedro Vicente, 625 – Canindé – São Paulo – SP
CEP: 01109-010
Telefone: (11) 3775-4570
E-mail: prp@ifsp.edu.br

SUMÁRIO

1 INTRODUÇÃO..... 15
1.1.1 Objetivo Geral..... 15
1.1.2 Objetivos Específicos..... 15
1.2 Justificativa..... 15
2 FUNDAMENTAÇÃO TEÓRICA..... 15
2.1 Sistemas de Arquivos..... 15
2.2 Estrutura de Inodes..... 16
2.3 Gerenciamento de Blocos de Dados..... 16
2.4 Sistemas de Arquivos Unix-like..... 16
2.5 Persistência de Dados..... 17
3 ARQUITETURA DO SISTEMA..... 17
3.1 Visão Geral da Arquitetura..... 17
3.2 Estruturas de Dados Principais..... 17
3.2.1 Superbloco - (Superblock)..... 17
3.2.2 Inode..... 18
3.2.3 Entrada de Diretório - (DirectoryEntry)..... 19
3.3 Módulos do Sistema..... 20
3.3.1 Módulo fs.h..... 20
3.3.2 Módulo disk_ops..... 21
3.3.3 Módulo commands..... 22
3.3.4 Módulo shell..... 23
3.5 Programa mkfs..... 23
4 GERENCIAMENTO DE RECURSOS..... 24
4.1 Gerenciamento de Espaço Livre..... 24
4.1.1 Estrutura do Bitmap..... 24
4.1.2 Operações sobre o Bitmap..... 24
4.1.3 Justificativa Técnica..... 25
4.2 Sistema de Busca em Diretórios..... 25
4.2.1 Estratégia Implementada..... 25
4.2.2 Justificativa da Abordagem..... 26
5 IMPLEMENTAÇÃO DOS COMANDOS..... 26
5.1 Comando ls - Listar Diretório..... 26
5.2 Comando mkdir - Criar Diretório..... 27
5.3 Comando cd - Mudar Diretório..... 27
5.4 Comando pwd - Exibir Diretório Atual..... 27
5.5 Comando touch - Criar Arquivo..... 27
5.6 Comando cat - Exibir Conteúdo..... 28
5.7 Comando rm - Remove Recurso..... 28
5.8 Comando stat - Status do Sistema..... 28
6 CONSIDERAÇÕES FINAIS..... 28
6.1 Conclusões..... 28
BIBLIOGRAFIA CONSULTADA..... 29

1 INTRODUÇÃO

1.1.1 Objetivo Geral

Desenvolver um sistema de arquivos educacional, em linguagem C, que simule de forma funcional os principais mecanismos de um sistema Unix-like, possibilitando o aprendizado prático dos conceitos de armazenamento, alocação, leitura e escrita de dados em nível de bloco.

1.1.2 Objetivos Específicos

- Desenvolver um shell interativo para manipulação de arquivos e diretórios com comandos básicos.

- Facilitar a compreensão didática dos conceitos de sistemas de arquivos e gestão de recursos.

1.2 Justificativa

O estudo e a implementação prática de sistemas de arquivos são essenciais para a compreensão das camadas internas dos sistemas operacionais. Entretanto, muitos desses conceitos permanecem abstratos no ensino teórico. O desenvolvimento do PEALFS busca preencher essa lacuna ao oferecer uma ferramenta prática que simula, em escala reduzida, o funcionamento real de um sistema de arquivos Unix-like.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Sistemas de Arquivos

Um sistema de arquivos é uma estrutura lógica que permite ao sistema operacional controlar como os dados são armazenados e recuperados em dispositivos de armazenamento. Sem um sistema de arquivos, os dados seriam armazenados como um grande bloco contínuo sem forma de identificar onde um arquivo termina e outro começa.

Os sistemas de arquivos modernos implementam funcionalidades como organização hierárquica de diretórios, controle de acesso, metadados de arquivos e mecanismos de recuperação de falhas. O PEALFS implementa conceitos fundamentais presentes em sistemas de arquivos Unix-like, incluindo a separação entre metadados (inodes) e dados propriamente ditos (blocos).

2.2 Estrutura de Inodes

O conceito de inode (index node) foi introduzido nos primeiros sistemas Unix e representa uma estrutura de dados que armazena informações sobre um arquivo ou diretório, exceto seu nome e conteúdo. Cada inode contém metadados como tipo do arquivo, tamanho, número de blocos alocados e ponteiros para os blocos de dados.
No PEALFS, cada inode possui aproximadamente 16-24 bytes, dependendo do alinhamento de memória (padding) aplicado pelo compilador.

- **Tipo:** Identifica se é arquivo **('f')**, diretório **('d'**) ou não utilizado **('u')**
- **Tamanho:** Quantidade de bytes de dados válidos
- **Contagem de blocos**: Número de blocos de dados alocados
- **Ponteiros diretos:** Array de 4 ponteiros para blocos de dados

Esta estrutura permite acesso direto aos dados sem necessidade de percorrer estruturas intermediárias, garantindo eficiência nas operações de leitura e escrita

2.3 Gerenciamento de Blocos de Dados

Os blocos de dados são as unidades básicas de alocação de espaço em disco. O PEALFS utiliza blocos de tamanho fixo de 128 bytes, o que simplifica o gerenciamento e permite cálculos diretos de endereçamento. A alocação de blocos pode seguir diferentes estratégias:

- **Contígua:** Blocos sequenciais (rápida mas sujeita a fragmentação)
- **Encadeada:** Cada bloco aponta para o próximo (lenta para acesso aleatório)
- **Indexada**: Uso de índices (inodes) que apontam para blocos (utilizada pelo PEALFS)

O PEALFS implementa alocação indexada através dos ponteiros diretos nos inodes, permitindo acesso direto a qualquer bloco do arquivo sem necessidade de percorrer estruturas intermediárias.

2.4 Sistemas de Arquivos Unix-like

O PEALFS é inspirado na arquitetura de sistemas de arquivos Unix, especialmente o ext2 (Second Extended Filesystem). Características compartilhadas incluem:

- **Separação de metadados e dados:** Inodes armazenam metadados enquanto blocos armazenam conteúdo
- **Diretórios como arquivos especiais:** Diretórios são arquivos cujo conteúdo são entradas de diretório
- **Entradas especiais [.] e [..]:** Todo diretório contém referências para si mesmo e seu pai
- **Identificação por número de inode:** Arquivos são identificados internamente por números, não nomes

2.5 Persistência de Dados

A persistência é garantida através do armazenamento de todas as estruturas de dados em arquivos regulares do sistema operacional hospedeiro. O PEALFS mantém quatro arquivos principais:

- **Superblock.dat:** Metadados do sistema de arquivos (texto plano)
- **Inodes.dat:** Tabela sequencial de inodes (binário)
- **Blocks.dat:** Sequência de blocos de dados (binário)
- **Freespace.dat:** Bitmap de blocos livres (binário)

Todas as operações de escrita são realizadas imediatamente nos arquivos correspondentes, garantindo que os dados estejam sempre sincronizados com o disco.

3 ARQUITETURA DO SISTEMA

3.1 Visão Geral da Arquitetura

O **PEALFS** possui uma arquitetura modular desenvolvida em linguagem **C**, que simula o funcionamento de um sistema de arquivos Unix-like de forma didática e funcional.
O sistema é composto por cinco módulos principais:

- **fs.h**: define as estruturas de dados e constantes globais;
- **disk_ops**: executa operações de leitura e escrita simulando o disco;
- **commands**: implementa os comandos de manipulação de arquivos e diretórios;
- **shell**: fornece a interface interativa com o usuário;
- **mkfs**: inicializa e formata o sistema de arquivos.

Essa organização garante separação de responsabilidades, facilita a manutenção do código e torna o sistema extensível para futuras melhorias.

3.2 Estruturas de Dados Principais

O núcleo do PEALFS é composto por três estruturas de dados fundamentais, definidas no módulo **fs.h**: **Superblock**, **Inode** e **DirectoryEntry**. Cada uma desempenha um papel específico na organização e gerenciamento das informações armazenadas.

3.2.1 Superbloco - (Superblock)

O superbloco é a estrutura de dados que contém metadados essenciais sobre o sistema de arquivos como um todo. No PealFS, o superbloco é representado pela estrutura Superblock definida em fs.h e armazenado persistentemente no arquivo fs/superblock.dat. A estrutura Superblock em memória contém os seguintes campos:

```
typedef struct {
    char filesystem[8];    // Nome/identificação do sistema de arquivos
    int block_size;        // Tamanho de cada bloco em bytes
    int partition_size;    // Tamanho total da partição em bytes
    int num_blocks;        // Número total de blocos
    int num_inodes;        // Número máximo de inodes
} Superblock;
```

O campo filesystem armazena a string "pealfs" que identifica o tipo de sistema de arquivos, permitindo que programas verifiquem se estão operando sobre um sistema PealFS válido antes de tentar acessá-lo. O campo block_size define a unidade mínima de alocação (128 bytes), usado para calcular posições de blocos no arquivo blocks.dat. O campo partition_size especifica o tamanho total disponível (10240 bytes), usado para validar que operações não excedam os limites do sistema.

Os campos num_blocks e num_inodes armazenam respectivamente o número total de blocos (80) e o número máximo de inodes (256), servindo como limites superiores para validações durante a alocação de recursos. Estes valores são calculados a partir de constantes em tempo de compilação mas são armazenados no superbloco para permitir que sistemas com diferentes configurações coexistem. No arquivo fs/superblock.dat, o superbloco é armazenado em formato texto legível:

```
filesystem=pealfs
blocksize=128
partitionsize=10240
```

Esta escolha de formato texto (em vez de binário) facilita inspeção manual e depuração, permitindo que desenvolvedores verifiquem a configuração do sistema usando editores de texto comuns. O formato chave-valor também permite extensibilidade futura, onde novos parâmetros podem ser adicionados sem quebrar compatibilidade com versões anteriores, bastando que o código de leitura ignore parâmetros desconhecidos.

3.2.2 Inode

O inode é a estrutura central que armazena metadados sobre cada arquivo ou diretório no sistema. No PealFS, cada inode é representado pela estrutura Inode definida em fs.h:

```
typedef struct {
    char type;                    // 'f' para arquivo, 'd' para diretório, 'u' para não utilizado
    uint32_t size;                // Tamanho do arquivo em bytes
    uint16_t block_count;         // Quantidade de blocos utilizados
   uint16_t direct_blocks[NUM_DIRECT_POINTERS];   // Ponteiros diretos para os blocos de dados
    // Podemos adicionar ponteiros indiretos aqui no futuro, se necessário.
} Inode;
```

O campo type diferencia arquivos regulares ('f') de diretórios ('d'), determinando como o

conteúdo dos blocos de dados deve ser interpretado. Para arquivos, os blocos contêm dados arbitrários fornecidos pelo usuário. Para diretórios, os blocos contêm um array de estruturas DirectoryEntry que mapeiam nomes para números de inodes.

O campo size armazena o número de bytes efetivamente utilizados pelo arquivo ou diretório. Para arquivos, este é o tamanho do conteúdo fornecido pelo usuário (limitado a 512 bytes). Para diretórios, este é o número de bytes ocupados pelas entradas de diretório, calculado como:

$$\text{num\_entries} \times \text{sizeof(DirectoryEntry)}$$

O campo direct_blocks é um array de 4 inteiros, onde cada posição contém o número de um bloco de dados ou -1 se aquela posição não está em uso. Quando um arquivo ou diretório é criado, pelo menos o direct_blocks[0] é alocado. Arquivos maiores que 128 bytes utilizam blocos adicionais (direct_blocks[1], direct_blocks[2], etc.) até o limite de 4 blocos. A ausência de ponteiros indiretos limita arquivos a 512 bytes mas simplifica drasticamente a implementação.

Os inodes são armazenados sequencialmente no arquivo fs/inodes.dat, onde o inode número i está localizado na posição i×sizeof(Inode) bytes do início do arquivo. Este layout permite acesso direto em tempo constante O(1) a qualquer inode dado seu número, simplesmente usando a operação de seek do sistema de arquivos hospedeiro. O tamanho da estrutura Inode é tipicamente 21-24 bytes dependendo do alinhamento de memória da arquitetura, resultando em uma tabela de inodes de aproximadamente 6 KB para 256 inodes.

3.2.3 Entrada de Diretório - (DirectoryEntry)

Uma entrada de diretório (directory entry) é a estrutura que mapeia um nome de arquivo ou subdiretório para seu número de inode correspondente. Esta indireção permite que múltiplos nomes referenciam o mesmo inode (hard links em sistemas mais avançados) e que arquivos sejam renomeados simplesmente alterando entradas de diretório sem mover dados.

No PealFS, cada entrada é representada pela estrutura DirectoryEntry:

```
typedef struct {
    char name[MAX_FILENAME];   // Nome do arquivo/diretório
    uint8_t inode_number;      // Número do inode correspondente (1 byte = 0-255)
} DirectoryEntry;
```

O campo name armazena o nome do arquivo ou diretório como uma string C terminada em nulo. O tamanho máximo é 14 caracteres úteis mais o terminador, totalizando 15 bytes. Esta limitação segue convenções históricas de sistemas Unix antigos

e garante que cada entrada tenha tamanho fixo, facilitando cálculos de posição dentro de blocos de diretório.

O campo inode_number é um inteiro unsigned de 8 bits (0-255) que identifica qual inode contém os metadados do arquivo. O uso de 8 bits limita o sistema a 256 inodes totais, o que é aceitável para um sistema educacional. O valor 0 é reservado para o diretório raiz, e valores de 1 a 255 estão disponíveis para outros arquivos e diretórios.

O tamanho total de cada DirectoryEntry é exatamente 15 bytes (14 bytes para o nome + 1 byte para o inode), permitindo que 8 entradas caibam perfeitamente em um bloco de 128 bytes (128÷15=8.53). Esta escolha otimiza o uso de espaço e permite que diretórios contenham até 8 entradas, incluindo as obrigatórias "." e "..", deixando espaço para 6 arquivos ou subdiretórios adicionais.

As entradas de diretório são armazenadas sequencialmente no bloco de dados do diretório. Para localizar uma entrada específica, o sistema calcula sua posição como

$$\text{entry\_index} \times \text{sizeof(DirectoryEntry)}$$

dentro do buffer do bloco, permitindo acesso direto. Entradas não utilizadas ou removidas são identificadas por terem inode_number igual a 0 (após remoção) ou estarem além do tamanho efetivo do diretório.

3.3 Módulos do Sistema

A arquitetura do PEALFS é modularizada em diferentes componentes de *software* para separar as responsabilidades e facilitar a manutenção e o desenvolvimento.

3.3.1 Módulo fs.h

O módulo fs.h é o arquivo de cabeçalho principal do sistema, responsável por definir todas as estruturas de dados, constantes e tipos utilizados pelo PEALFS. Este módulo estabelece a arquitetura fundamental do sistema de arquivos através de três componentes essenciais.

O módulo define as seguintes constantes que determinam as limitações e características do filesystem:

BLOCK_SIZE	Define o tamanho de cada bloco em 128 bytes
PARTITION_SIZE	Estabelece o tamanho total da partição em 10.240 bytes
NUM_BLOCKS	Calcula automaticamente 80 blocos totais (10240 ÷ 128)
MAX_INODES	Limita a 256 inodes o número máximo de arquivos e diretórios
MAX_FILENAME	Define 14 caracteres como tamanho máximo para nomes de arquivos

BLOCK_SIZE	Define o tamanho de cada bloco em 128 bytes
NUM_DIRECT_POINTERS	Estabelece 4 ponteiros diretos por inode, limitando arquivos a 512 bytes

Estruturas principais:

- A estrutura Superblock armazena metadados globais incluindo o nome do filesystem, tamanho do bloco, tamanho da partição, número de blocos e número de inodes. É armazenada em formato texto no arquivo superblock.dat.

- A estrutura Inode representa cada arquivo ou diretório, contendo: campo type ('f' para arquivo, 'd' para diretório, 'u' para não utilizado), campo size (tamanho em bytes do conteúdo), campo block_count (número de blocos utilizados) e array direct_blocks com 4 ponteiros para blocos de dados. Cada inode ocupa aproximadamente 21-24 bytes dependendo do alinhamento.

- A estrutura DirectoryEntry mapeia nomes de arquivos para inodes, com um campo name de 14 caracteres e um campo inode_number de 8 bits (0-255). Cada entrada ocupa exatamente 15 bytes (14 + 1), permitindo 8 entradas por bloco (128 / 15 ≈ 8,53)

3.3.2 Módulo disk_ops

O módulo disk_ops fornece a camada de abstração para todas as operações de I/O do filesystem, simulando acesso a disco através de arquivos binários.

Operações com blocos:

- read_block(): Abre blocks.dat, posiciona o ponteiro em block_num * BLOCK_SIZE usando fseek() e lê 128 bytes para o buffer

- write_block(): Escreve 128 bytes do buffer na posição calculada do arquivo blocks.dat

Operações com inodes:

- read_inode(): Acessa inodes.dat na posição inode_num * sizeof(Inode) e lê a estrutura completa

- write_inode(): Grava a estrutura Inode na posição correspondente em inodes.dat

Gerenciamento de recursos:

- find_free_inode(): Percorre sequencialmente inodes.dat procurando o primeiro inode com type == 'u', retornando seu índice ou -1

- find_free_block(): Lê o bitmap de freespace.dat e examina cada bit em ordem MSB-first (bit mais significativo primeiro), retornando o número do primeiro bloco livre ou -1

- alloc_block(): Marca um bloco como ocupado setando o bit correspondente para 1 no bitmap usando operação OR (|=)

- free_block(): Libera um bloco zerando seu bit no bitmap usando operação AND com máscara negada (&= ~)

3.3.3 Módulo commands

O módulo commands implementa os oito comandos do shell interativo, manipulando arquivos e diretórios através das funções do disk_ops

Comandos de navegação:

- do_ls(): Lê o inode do diretório atual, interpreta seu primeiro bloco como array de DirectoryEntry, itera pelas entradas exibindo tipo, número de inode, nome e tamanho

- do_cd(): Busca o nome fornecido nas entradas do diretório atual, verifica se é um diretório (type == 'd') e atualiza current_directory_inode

- do_pwd(): Reconstrói o caminho absoluto percorrendo a hierarquia de diretórios usando a entrada '..' até alcançar a raiz (inode 0), construindo o caminho de trás para frente

Comandos de criação:

- **do_mkdir():** Valida o nome, verifica duplicatas, aloca inode e bloco livres, configura o novo inode com type 'd', inicializa o bloco com entradas '.' (próprio inode) e '..' (inode pai), adiciona uma entrada no diretório pai e persiste as mudanças

- **do_touch():** Solicita entrada do usuário via stdin até EOF, calcula blocos necessários (máximo 4), aloca recursos, divide o conteúdo em chunks de 128 bytes, escreve cada chunk em seu bloco correspondente e adiciona entrada no diretório pai

Comandos de leitura e remoção:

- do_cat(): Localiza o arquivo, verifica type 'f', lê blocos sequencialmente e imprime apenas bytes válidos do último bloco usando fwrite()

- do_rm(): Localiza a entrada, proíbe remoção de '.' e '..', verifica se diretórios estão vazios, libera blocos com free_block(), marca inode como 'u' e remove entrada do diretório pai

- do_stat(): Lê bitmap completo, conta bits zerados (blocos livres), calcula espaço livre em bytes e exibe estatísticas

3.3.4 Módulo shell

O módulo shell (implementado em shell.c) fornece a interface de linha de comando que permite ao usuário interagir com o PealFS

Mantém a variável global current_directory_inode inicializada em 0, representando o diretório raiz. A função main() exibe o prompt "peal:/ >", lê entrada com fgets(), remove newline usando strcspn(), utiliza strtok() para separar o comando de argumentos.

O mapeamento de comandos utiliza cadeia if-else: comandos sem argumentos (ls, pwd, stat, exit) são chamados diretamente; comandos com argumentos (mkdir, cd, touch, cat, rm) validam a presença do argumento antes da execução.

3.5 Programa mkfs

O programa `mkfs` (make filesystem) formata e inicializa o PEALFS, criando toda a estrutura necessária antes do primeiro uso.

Processo de inicialização:

Cria o diretório fs/ usando _mkdir() e gera quatro arquivos essenciais. O arquivo superblock.dat é criado em formato texto com três linhas:

filesystem:	pealfs
blocksize:	128
partitionsize:	10240

O arquivo freespace.dat é inicializado com 10 bytes (80 bits ÷ 8), todos zerados indicando blocos livres. Em seguida, o primeiro bit é setado para 1 usando a operação bitmap |= (1 << 7), marcando o bloco 0 (do diretório raiz) como completamente ocupado.

O arquivo inodes.dat é preenchido com 256 inodes, todos marcados como 'u' (não utilizados). O inode 0 é então reconfigurado como diretório raiz: type 'd', size de 32 bytes (2 × sizeof(DirectoryEntry)), block_count 1, direct_blocks apontando para bloco 0 e demais ponteiros marcados como -1.

O arquivo blocks.dat é criado com 10.240 bytes zerados (80 blocos × 128 bytes). O bloco 0 recebe duas entradas de diretório: entrada '.' com inode_number 0 e entrada '..' também com inode_number 0, pois a raiz é seu próprio pai.

4 GERENCIAMENTO DE RECURSOS

4.1 Gerenciamento de Espaço Livre

O PEALFS implementa o gerenciamento de espaço livre através de um bitmap armazenado no arquivo freespace.dat, que rastreia o estado de alocação de cada um dos 80 blocos da partição.

4.1.1 Estrutura do Bitmap

O bitmap consiste em um arquivo binário de exatamente 10 bytes, onde cada bit representa o estado de um bloco específico. A representação utiliza a convenção: bit 0 (zero) indica bloco livre e disponível para alocação; bit 1 (um) indica bloco ocupado e em uso por arquivo ou diretório.

O sistema adota a ordenação MSB-first (Most Significant Bit first), onde o bit mais significativo de cada byte representa o bloco de menor índice. Por exemplo, no primeiro byte do bitmap: o bit 7 (MSB) representa o bloco 0, o bit 6 representa o bloco 1, e assim sucessivamente até o bit 0 (LSB) representando o bloco 7. Este padrão continua nos bytes subsequentes, com o byte 1 representando blocos 8-15, byte 2 representando blocos 16-23, e assim por diante. O cálculo da posição de um bloco no bitmap é realizado através de duas operações:

Operação 1:

byte_index = block_num / 8	Determina qual byte contém o bit relevante

Operação 2:

bit_offset = 7 - (block_num % 8)	Calcula a posição do bit dentro do byte seguindo a ordem MSB-first. Essa estrutura permite representar 80 blocos usando apenas 10 bytes.

4.1.2 Operações sobre o Bitmap

O módulo disk_ops implementa quatro operações fundamentais sobre o bitmap, todas seguindo o padrão de leitura completa do arquivo, modificação em memória e escrita completa.

find_free_block()	Percorre o bitmap da esquerda para a direita (do byte 0 ao 9, e do bit mais significativo ao menos significativo) em busca do primeiro bit zero encontrado. Retorna o número do bloco livre. Retorna -1 se não houver espaço.
alloc_block(int block_num)	Altera o bit correspondente ao `block_num` para **1** (ocupado), atualizando e salvando o `freespace.dat`.
free_block(int block_num)	Altera o bit correspondente ao `block_num` para **0** (livre), operação usada durante a remoção de recursos (`do_rm`).

do_stat()	lê o bitmap completo e implementa um loop duplo que percorre todos os bytes e todos os bits, incrementando um contador sempre que encontra um bit zerado. O resultado é multiplicado por `BLOCK_SIZE` (128 bytes) para exibir o espaço livre total em bytes.

4.1.3 Justificativa Técnica

A escolha do bitmap como estrutura de gerenciamento de espaço livre se justifica por múltiplas razões técnicas adaptadas ao contexto do PEALFS.

Eficiência de espaço: Com apenas 80 blocos na partição, o bitmap ocupa apenas 10 bytes, representando overhead desprezível (menos de 0,1% do espaço total). Estruturas alternativas como listas encadeadas de blocos livres consumiriam mais espaço e aumentariam a complexidade.

Simplicidade de implementação: Operações sobre bits usando máscaras (&, |, ~, <<) são diretas e não requerem estruturas de dados complexas. O código resultante é compacto e fácil de depurar.

Acesso direto: Verificar o estado de qualquer bloco específico é uma operação O(1), bastando calcular o byte e bit correspondentes. Isto contrasta com listas encadeadas que exigiriam busca sequencial.

4.2 Sistema de Busca em Diretórios

O PEALFS implementa busca linear em diretórios para localizar arquivos e subdiretórios por nome, uma abordagem adequada dado o limite de 8 entradas por diretório.

4.2.1 Estratégia Implementada

A busca em diretórios é realizada através de um algoritmo de três etapas executado por múltiplas funções do módulo commands.

Etapa 1 - Leitura do diretório:

A função lê o inode do diretório atual usando read_inode(current_directory_inode, &dir_inode) e verifica se o campo type é 'd', garantindo que a operação é válida apenas em diretórios. Em seguida, lê o primeiro bloco de dados do diretório usando read_block(dir_inode.direct_blocks, block_buffer)

Etapa 2 - Interpretação das entradas:

O buffer de 128 bytes é convertido em um ponteiro para array de DirectoryEntry através de cast: DirectoryEntry *entries = (DirectoryEntry *)block_buffer. O número de entradas válidas é calculado como num_entries = dir_inode.size / sizeof(DirectoryEntry), utilizando o campo size do inode que armazena exatamente quantos bytes de entradas

estão em uso.

Etapa 3 - Busca sequencial:

Um loop percorre as entradas de 0 até num_entries - 1, comparando o campo name de cada entrada com o nome procurado usando strcmp(). Quando encontra correspondência, a função extrai o inode_number e realiza a operação desejada (navegar para o diretório em do_cd(), ler o arquivo em do_cat(), remover em do_rm(), etc.).

4.2.2 Justificativa da Abordagem

A busca linear foi escolhida como estratégia para localizar arquivos e diretórios no PEALFS principalmente devido ao tamanho reduzido dos diretórios. Como cada diretório suporta no máximo 8 entradas (sendo apenas 6 utilizáveis após reservar espaço para '.' e '..'), usar algoritmos mais sofisticados como árvores balanceadas ou hash tables simplesmente não traria benefícios práticos.

Vale mencionar que essa abordagem segue a mesma linha de sistemas Unix clássicos como o ext2, que também usam busca linear para diretórios pequenos e só adotam estruturas indexadas quando há centenas de entradas.

5 IMPLEMENTAÇÃO DOS COMANDOS

O PEALFS disponibiliza oito comandos fundamentais através de uma interface shell interativa, implementados no módulo commands.c. Cada comando opera diretamente sobre as estruturas de dados do filesystem utilizando as funções fornecidas pelo módulo disk_ops. Essa seção descreve detalhadamente a implementação, algoritmo e tratamento de erros de cada comando.

5.1 Comando ls - Listar Diretório

O comando ls é responsável por listar o conteúdo do diretório atual no sistema de arquivos PEALFS. Ao ser executado, ele acessa o inode do diretório, verifica sua validade como diretório e percorre as entradas armazenadas no bloco de dados, exibindo informações formatadas como tipo, número do inode, nome e tamanho em bytes de cada arquivo ou subdiretório presente. Caso o inode não seja um diretório válido, o comando retorna uma mensagem de erro apropriada. O comando é fundamental para a navegação e organização dos arquivos dentro do sistema, apresentando os resultados em formato tabular com cabeçalho.

5.2 Comando mkdir - Criar Diretório

O comando mkdir permite a criação de um novo subdiretório no diretório atual. O funcionamento envolve a validação do nome fornecido, a verificação de duplicidade e a alocação de recursos necessários, como inode e bloco livre. O comando também verifica se há espaço suficiente no diretório pai para adicionar a nova entrada. Após a configuração do novo inode e inicialização do bloco com as entradas obrigatórias '.' e '..', o bloco é marcado como ocupado no bitmap e o diretório pai é atualizado para incluir o novo subdiretório. O comando garante que apenas nomes válidos e diretórios não duplicados sejam criados, retornando mensagens de erro específicas em caso de falhas ou limitações de recursos.

5.3 Comando cd - Mudar Diretório

O comando cd é utilizado para alterar o diretório de trabalho atual, permitindo a navegação pela hierarquia do sistema de arquivos. Ao receber o nome do diretório desejado, o comando busca a entrada correspondente no diretório atual, valida se o destino é realmente um diretório e atualiza o contexto do shell para o novo inode através da variável global current_directory_inode. O comando implementa um caso especial para navegar diretamente para a raiz através do caminho '/'. Caso o nome não seja encontrado ou não corresponda a um diretório, uma mensagem de erro é exibida, mantendo o usuário no diretório original.

5.4 Comando pwd - Exibir Diretório Atual

O comando pwd exibe o caminho absoluto do diretório de trabalho atual. Para isso, o comando realiza um rastreamento reverso na hierarquia de diretórios, partindo do inode atual até a raiz. O algoritmo utiliza as entradas '..' para subir na hierarquia, consultando o diretório pai para descobrir o nome do diretório atual em cada nível, reconstruindo assim o caminho completo. Quando o diretório atual é a raiz, o comando simplesmente exibe '/'. O resultado é apresentado ao usuário, facilitando a orientação e localização dentro do sistema de arquivos, especialmente em estruturas mais profundas ou complexas.

5.5 Comando touch - Criar Arquivo

O comando touch é utilizado para criar um novo arquivo regular no diretório atual, solicitando o conteúdo do usuário via entrada padrão através dos comandos Ctrl+Z (Windows) ou Ctrl+D (Linux) para finalizar. O processo envolve a validação do nome, verificação de duplicidade, leitura do conteúdo, alocação dinâmica dos blocos necessários e configuração do inode correspondente. O arquivo criado é limitado a 512 bytes (resultado de 4 ponteiros diretos × 128 bytes por bloco), devido à estrutura do sistema. O comando implementa tratamento de erros robusto, liberando blocos já alocados em caso de falha, e garante que apenas arquivos válidos e não duplicados sejam adicionados ao diretório, retornando mensagens de erro apropriadas em caso de falhas.

5.6 Comando cat - Exibir Conteúdo

O comando cat tem como objetivo exibir o conteúdo completo de um arquivo regular na saída padrão. Ao ser executado, o comando localiza o arquivo pelo nome no diretório atual, valida se é um arquivo regular e lê sequencialmente os blocos de dados associados ao inode. O conteúdo é exibido ao usuário utilizando a função fwrite() para garantir a correta impressão de dados brutos, e uma nova linha é adicionada ao final da saída. Caso o arquivo não exista ou não seja um arquivo regular (por exemplo, se for um diretório), o comando retorna uma mensagem de erro específica, garantindo a integridade da operação.

5.7 Comando rm - Remove Recurso

O comando rm é responsável por remover arquivos regulares ou diretórios vazios do sistema de arquivos. O funcionamento envolve a localização da entrada pelo nome, validação do tipo e, no caso de diretórios, verificação rigorosa de que estejam vazios (contendo apenas as entradas '.' e '..'). Após a liberação dos blocos de dados através da função free_block() que atualiza o bitmap, o inode é marcado como não utilizado (tipo 'u') e a entrada é removida do diretório pai. Para otimizar o processo, a última entrada do diretório é copiada para a posição do item removido. O comando implementa proteções importantes, impedindo a remoção das entradas especiais '.' e '..' e a remoção de diretórios não vazios, retornando mensagens de erro específicas quando necessário.

5.8 Comando stat - Status do Sistema

Por fim, o comando stat exibe estatísticas básicas sobre a utilização de recursos do sistema de arquivos. O comando realiza a leitura do bitmap de blocos livres armazenado em freespace.dat, itera pelos bits para contar quantos blocos estão disponíveis (marcados com 0) e calcula o espaço livre correspondente em bytes. O relatório apresentado ao usuário contém três informações fundamentais: quantidade de blocos livres, espaço livre em bytes e tamanho do bloco (128 bytes). Essas informações são úteis para o monitoramento e gerenciamento básico do espaço disponível no sistema, permitindo ao usuário verificar a capacidade de armazenamento restante.

6 CONSIDERAÇÕES FINAIS

6.1 Conclusões

O desenvolvimento do **PEALFS** permitiu a aplicação prática dos conceitos fundamentais de sistemas de arquivos, consolidando o entendimento sobre a interação entre inodes, blocos de dados e mecanismos de gerenciamento de espaço. A implementação modular e a execução dos comandos via shell demonstraram a viabilidade de simular um sistema Unix-like de forma simplificada, mas funcional.

Além do aprendizado técnico, o projeto proporcionou experiência na análise, modelagem e depuração de estruturas persistentes, reforçando a importância da organização lógica e da consistência entre camadas de software.

BIBLIOGRAFIA CONSULTADA

TANENBAUM, Andrew S.; BOS, Herbert. *Sistemas Operacionais Modernos.* 4. ed. São Paulo: Pearson, 2015.