Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Sistema de Arquivos: DCC605FS

Aluno: João Francisco Barreto da Silva Martins
Disciplina: DCC605: Sistemas Operacionais
Professor: Ítalo Fernando Scotá Cunha

1. Introdução

Sistemas de arquivos são ferramentas de suma importância para organizar e gerenciar os dados armazenados em uma mídia física. Ao separar os dados em estruturas lógicas, o sistema de arquivos permite ao sistema operacional realizar operações sobre eles, como leitura e escrita.

Os sistemas operacionais mais utilizados no mundo possuem seus próprios sistemas de arquivo. São eles: HFS+ (macOS), NTFS (Windows) e ext4 (Linux). Apesar possuírem algumas particularidades eles funcionam de forma muito similar, todos requerendo um superbloco que contém as informações base do sistema, e apontam para partes chave dele. O superbloco e replicado em diversos ao longo do disco para prevenir que a corrupção de um setor deixe o sistema de arquivos ilegível.

Por causa das diferenças entre sistemas, se você estiver operando um Mac por exemplo, não haverá compatibilidade para escrita em arquivos que utilizam alguns outros sistemas de arquivos, como NTFS. Para que pudéssemos realizar essa operação seria necessário um driver externo que trabalharia como intermediário entre os dois. Porém também existem sistemas que são multiplataforma, como o FAT32 e exFAT, muito usados em drives portáteis, como pendrives, que são reconhecidos e manipulados por qualquer sistema operacional.

Este trabalho visa implementar as primitivas de um sistema de arquivos. Nomeado DCC605FS, o nosso sistema existe dentro de um arquivo do computador e simula as políticas básicas de um sistema de arquivo, como hierarquia de arquivos e diretórios e gerenciamento de espaço livre.

2. Modelagem e Implementação

Foram implementadas diversas funções além das descritas no cabeçalho(fs.h) provido com a especificação do trabalho, de forma que o código ficasse mais legível e otimizado. As funções auxiliares criadas serão descritas nas subseções abaixo.

2.1. get_file_size

- **Cabeçalho**: int get_file_size(const char *fname)
- **Funcionalidade**: Abre o arquivo em **fname** e retorna o seu tamanho em bytes.

2.2. fs_write_data

- **Cabeçalho**: void fs_write_data(struct superblock *sb, uint64_t pos, void *data)
- **Funcionalidade**: Escreve os dados de **data** na posição **pos** de **sb->fd**.

2.3. fs_read_data

- **Cabeçalho**: void fs_read_data(struct superblock *sb, uint64_t pos, void *data)
- **Funcionalidade**: Lê os dados da posição **pos** de **sb->fd** e os grava em **data**.

2.4. fs_find_dir_info

- **Cabeçalho**: struct dir * fs_find_dir_info(struct superblock *sb, const char *dpath)
- **Funcionalidade**: Faz um parsing no caminho em **dpath**, percorre a hierarquia de arquivos e retorna uma struct do tipo **dir**, descrita como:

  ```
  struct dir {
      uint64_t dirnode; /* O bloco relativo ao diretorio pai do inode pedido */
      uint64_t nodeblock; /* O bloco do inode(arquivo ou diretorio) pedido */
      char *nodename; /* O nome do inode pedido */
  };
  ```

  Caso dpath esteja referenciando apenas o diretório raiz("/"), **dirnode** e **nodeblock** assumem o valor 1 e **nodename** é a string vazia **""**. Caso contrário, dirnode assume o valor do bloco relativo ao diretório pai do inode pedido, que pode ser tanto um diretório como um arquivo; nodeblock assume o valor do bloco do inode caso ele já exista, ou -1 caso contrário; e nodename assume o nome do inode, que é o último nome em dpath. Se o caminho dpath não existir, a função atribui ENOENT a errno e retorna NULL.

2.5. fs_find_link

- **Cabeçalho**: struct link * fs_find_link(struct superblock *sb, uint64_t inodeblk, uint64_t linkvalue)
- **Funcionalidade**: Percorre os links de **inodeblk**, e de seu possíveis filhos, procurando por uma referência a **linkvalue**, e retorna uma struct do tipo **link**, descrita como:

  ```
  struct link {
      uint64_t inode; /* Bloco do inode que contem o link */
      int index; /* Indice do link em inode */
  };
  ```

  Em **inode** armazenamos o bloco do inode que contém o link a linkvalue e a **index** atribuímos o seu índice dentro do arranjo de links do inode. Se o link não existir, retorna o último inode da lista de inodes relacionados a inodeblk e -1 como index. Uma chamada de fs_find_link com linkvalue igual a zero busca por um link vazio.

2.6. fs_create_child

- **Cabeçalho**: uint64_t fs_create_child(struct superblock *sb, uint64_t thisblk, uint64_t parentblk)
- **Funcionalidade**: Cria um **inode** filho(IMCHILD) com parent igual a **parentblk** e meta(inode anterior na lista) igual a **thisblk** e retorna o valor do bloco aonde ele foi salvo.

2.7. fs_add_link

- **Cabeçalho**: void fs_add_link(struct superblock *sb, uint64_t parentblk, int linkindex, uint64_t newlink)

- **Funcionalidade**: Adiciona **newlink** no índice **linkindex** da lista de links do inode contido em **parentblk**.

2.8. fs_remove_link

- **Cabeçalho**: void fs_remove_link(struct superblock *sb, uint64_t parentblk, int linkindex)
- **Funcionalidade**: Remove o link do índice **linkindex** da lista de links do inode contido em **parentblk**.

2.9. fs_has_links

- **Cabeçalho**: int fs_has_links(struct superblock *sb, uint64_t thisblk)
- **Funcionalidade**: Checa se o inode contido em **thisblk** possui algum link.

3. Organização Inicial do Sistema

Na chamada de **fs_format(const char *fname, uint64_t blocksize)** nós construímos o sistema de arquivos em cima do arquivo **fname**. O número de blocos deste sistema é calculado como o tamanho do arquivo fname dividido por **blocksize**, e então é armazenado em **sb->blks**. No primeiro bloco (bloco 0) gravaremos o superbloco. Em seguida gravaremos o inode do diretório raiz no bloco de número 1 e seus metadados, apontados pelo campo meta, logo na sequência no bloco 2. A partir daí populamos todos os blocos restantes com freepages que formam uma lista encadeada, com o primeiro elemento apontado por **sb->freelist**. Essas freepages serão então substituídas por outros tipos de dado a medida que os blocos forem sendo requisitados.

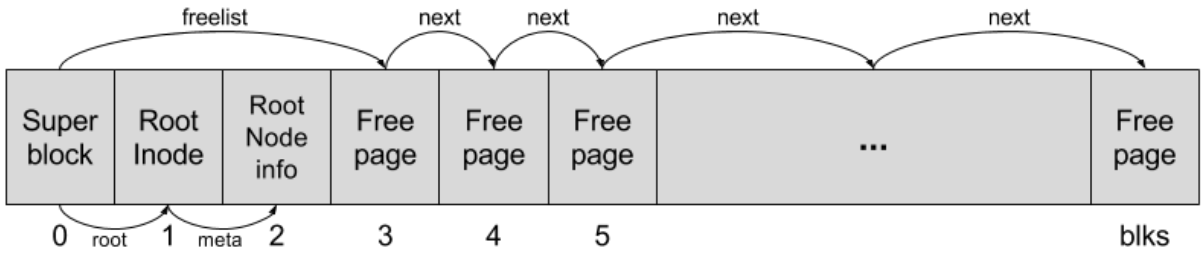

Figura 1 - Sistema de arquivos DCC605FS logo após formatação

4. Organização das páginas de blocos livres

Como dito na seção anterior, utilizamos a estrutura de dados **freepage**, declarada no header file **fs.h**, para preenchermos os blocos livres. Essa estrutura possui um campo **next** que utilizamos para criar a lista encadeada de blocos livres. O bloco apontado por **sb->freelist** é a cabeça da lista, e aponta para o próximo. A última freepage aponta para o endereço 0 para simbolizar o fim da lista. O campo **links** não foi utilizado, porém ele é inicializado assim com o campo **counts** para caso o sistema operacional queira interagir com essa parte da estrutura. Para gerenciarmos o espaço de armazenamento do sistema de arquivos utilizamos as funções **fs_get_block** e **fs_put_block**. Elas foram desenvolvidas de forma que ambas as funções operassem em O(1), pois essas são as operações que mais serão utilizadas pelo sistema de arquivos.

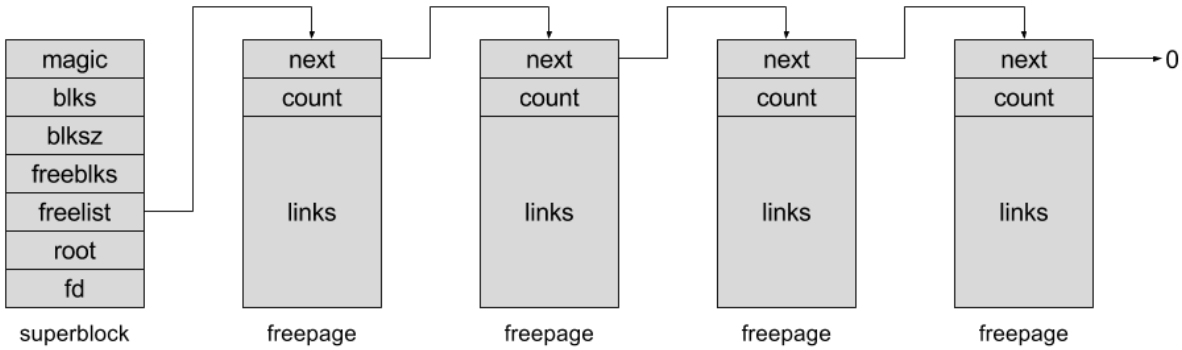

Figura 2 - Lista encadeada de freepages representando os blocos livres

4.1. fs_get_block

- **Cabeçalho**: uint64_t fs_get_block(struct superblock *sb);
- **Funcionalidade**: Pega o bloco apontado por sb->freelist e retorna seu endereço relativo dentro do sistema de arquivos. A nova cabeça da lista se torna o campo **next** da antiga cabeça.

4.2. fs_put_block

- **Cabeçalho**: int fs_put_block(struct superblock *sb, uint64_t block);
- **Funcionalidade**: Cria uma freepage, faz seu campo **next** apontar para sb->freelist e faz com que sb->freelist aponte ela, assim tornando-a a cabeça da lista de blocos livres. Por fim grava os seus dados no bloco apontado por **block**.