

Introdução

Os sistemas de arquivos são essenciais para a organização e gerenciamento dos dados em mídias físicas. Eles estruturam os dados de forma lógica, permitindo que o sistema operacional realize operações de leitura e escrita sobre eles. Cada sistema operacional popular possui seu próprio sistema de arquivos: HFS+ para macOS, NTFS para Windows e ext4 para Linux. Apesar das diferenças, todos esses sistemas têm elementos comuns, como o superbloco, que contém informações fundamentais e aponta para as partes principais do sistema de arquivos. Para garantir a integridade, o superbloco é replicado em várias localizações ao longo do disco, prevenindo a perda de dados em caso de corrupção de setores.

Devido às diferenças entre sistemas de arquivos, a compatibilidade entre diferentes sistemas operacionais pode ser limitada. Por exemplo, no macOS, pode não ser possível escrever em arquivos formatados com NTFS sem o uso de um driver externo que atue como intermediário. No entanto, existem sistemas de arquivos multiplataforma, como FAT32 e exFAT, que são amplamente utilizados em dispositivos portáteis, como pendrives, e são reconhecidos e manipulados por todos os principais sistemas operacionais.

Este trabalho tem como objetivo implementar as primitivas de um sistema de arquivos chamado DCC605FS. Esse sistema será implementado dentro de um arquivo do computador e simulará as principais políticas de um sistema de arquivos, como a hierarquia de arquivos e diretórios e o gerenciamento de espaço livre.

Sumário

Introdução	1
Sumário	1
Modelagem e Implementação	3
get_file_size	3
fs_write_data	3
fs_read_data	3
fs_find_dir_info	3
fs_find_link	4
fs_create_child	4
fs_add_link	5
fs_remove_link	5
fs_has_links	5
Organização	5
Organização das páginas de blocos livres	6
fs_get_block	6
fs_put_block	6

Modelagem e Implementação

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`

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`.

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`.

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;  
    uint64_t nodeblock;  
    char *nodename;  
};
```

- `uint64_t dirnode` é o bloco relativo ao diretório pai do inode pedido
- `uint64_t nodeblock` é o bloco do inode(arquivo ou diretório) pedido
- `char *nodename` é 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`.

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 seus possíveis filhos, procurando por uma referência a **linkvalue**, e retorna uma struct do tipo **link**, descrita como:



```
struct link {
    uint64_t inode;
    int index;
};
```

Onde

- `uint64_t inode` é o bloco do inode que contém o link */
- `int index` é o índice 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.

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.

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**.

fs_remove_link

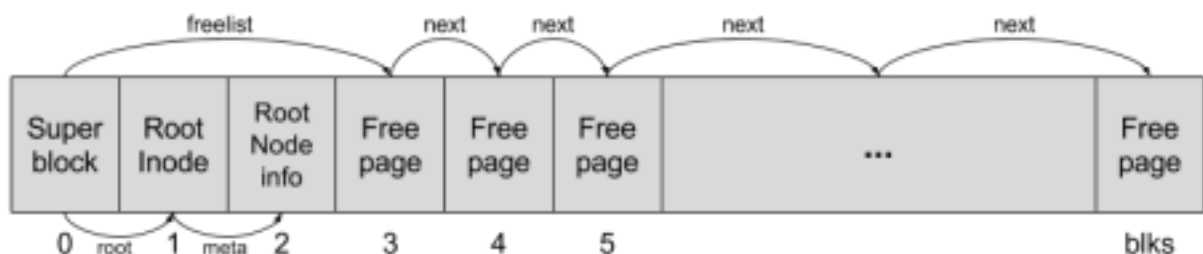
- Cabeçalho: void fs_remove_link(struct superblock *sb, uint64_t parentblk, int linkindex)
- Funcionalidade: Remove o link do índice **linkindex** da lista de links do inode contido em **parentblk**.

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.

Organização

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.



Organização das páginas de blocos livres

Como dito na seção anterior, utilizamos a estrutura de dados **freepage**, declarada no header file **fs.h**, para preencher 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.

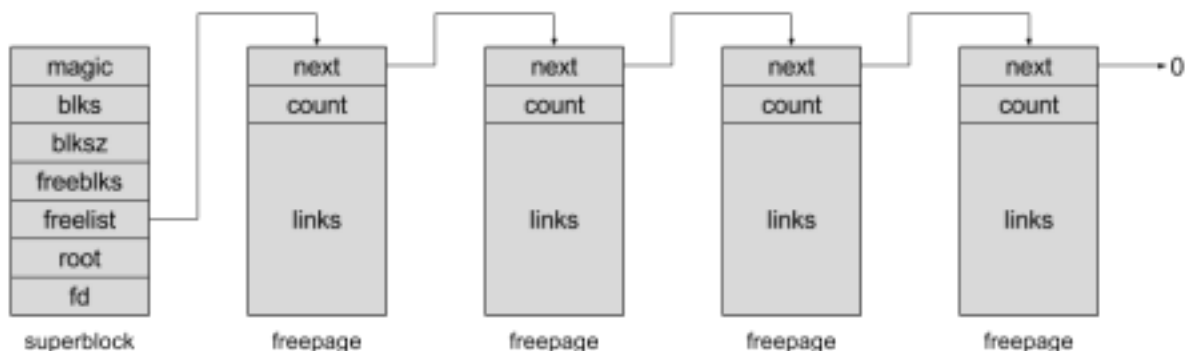


Figura 2 - Lista encadeada de freepages representando os blocos livres

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.

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**.