

# Implementação do sistema de arquivos SimpleSF

André Luigi Bonote  
Filipe Ernesto Siegrist Gonçalves  
Engenharia mecatrônica, Universidade Federal de Santa Catarina,  
centro tecnológico de joinville

2018/02

## 1 Introdução

Como encerramento da disciplina de sistemas operacionais (EMB5632) foi proposto aos discentes a implementação de um sistema simples de arquivos muito similar ao usado no *Unix*, o *simplefs*, com o objetivo de aprofundar os conteúdos vistos durante o curso. Para isto foram fornecidos:

- Protótipos das funções a serem implementadas;
- Um *shell* que se encarrega de fazer a interface do sistema de arquivos com o usuário;
- Um emulador de disco rígido;
- Três imagens virtuais de disco já contendo alguns arquivos.

## 2 Solução

Para o desenvolvimento da solução as funções foram desenvolvidas uma a uma, sequencialmente, conforme sugerido pelo enunciado disponibilizado. Além do desenvolvimento das funções principais também foi necessário a criação e o uso de novas funções auxiliares e algumas estruturas e variáveis globais.

Para fins de implementação optou-se pela manutenção do formato das estruturas de *inodes* e blocos. Essas decisões de projeto foram tomadas para manter a simplicidade e a agilidade do desenvolvimento além de manter a compatibilidade com a arquitetura original. Assim, os seguintes vetores (e seus respectivos tamanhos): `direct[ POINTERS_PER_INODE ]`; e `pointers[ POINTERS_PER_BLOCK]` foram mantidos estáticos e as constantes não foram alteradas.

Outra alteração importante foi o uso da linguagem *C++* ao invés da original, *C*, devido a sua maior gama de ferramentas disponíveis e também de sua compatibilidade com a arquitetura original.

Durante a criação do código, que deveria ser realizada em equipe, optou-se pelo uso da ferramenta *Git*, através do *GitHub*, que oferece poderosos recursos já familiares graças ao seu uso em matérias anteriores.

## 2.1 Funções auxiliares

As seguintes funções foram desenvolvidas com o intuito de auxiliar no desenvolvimento prático do sistema:

### 2.1.1 *int espacoLivre()*:

Esta função retorna o número de bytes desocupados no sistema de arquivos. Foi utilizada na função *fs\_write()*.

### 2.1.2 *int blocoDisponivel()*:

Percorre o *bitmap* e retorna o primeiro numero de bloco disponivel. Foi utilizada na alocação de blocos pela função *fs\_write()*.

### 2.1.3 *void printinodemap()*:

Mostra na tela o mapa de inodos. Foi chamada dentro da função *fs\_debug()*.

### 2.1.4 *void printbitmap()*:

Mostra na tela o mapa de blocos (*bitmap*). Também foi chamada dentro da função *fs\_debug()*.

### 2.1.5 *void inode\_load( int inumber, struct fs\_inode \*inode\_ler )*:

Carrega o “inumber”-ésimo inodo da memória e armazena-o na estrutura apontada por *inode\_ler*. Foi chamada dentro das funções *fs\_read()* e *fs\_write()*.

### 2.1.6 *void inode\_save( int inumber, struct fs\_inode inode\_esc )*:

Armazena o inodo da estrutura apontada por *inode\_ler* no “inumber”-ésimo inodo da memória. Foi chamada dentro da função *fs\_write()*.

## 2.2 Estruturas e variáveis globais

### 2.2.4 *volatile bool \_mounted*:

Variável booleana que armazena a informação que diz se o disco está montado ou não. Quase todas as funções leem seu valor para prosseguir mas apenas a função *fs\_mount()* a altera.

### 2.2.5 *vector<bool> bitmap*:

Essa é uma estrutura global em formato de vetor do tipo booleano também. Cada posição representa um bloco do disco. Se o bloco está livre ele contém *false*. Se o bloco está ocupado a posição guardará *true*. Foi utilizada intensamente para uma alocação (assim como a liberação) rápida de blocos da memória. É inicializada quando a função *fs\_mount()* é chamada.

### 2.2.6 *vector<im\_elem> inodemap*:

Essa outra estrutura global também está em formato de vetor mas o tipo é uma estrutura especial do tipo *im\_elem* descrita abaixo:

```
struct im_elem {
    int bloco_im;
    bool im_valid;
};
```

Figura 1 - Estrutura *im\_elem*. Guarda o bloco em que o inodo se encontra e se esse inodo é válido.

Essa estrutura é o mapa dos inodos. Serve para facilitar a criação destes oferecendo um acesso rápido e simples. É inicializada quando a função *fs\_mount()* é chamada.

## 2.3 Funções solicitadas

### 2.3.1 *int fs\_format()*:

Uma das funções primordiais do sistema de arquivos. Só é executada quando o bloco não está montado (verifica a variável global *\_mounted*) e apaga todos os inodos (invalida-os) e também apaga todas as referências que esses inodos contém para blocos (diretos e indiretos). O número mágico e o tamanho do disco são mantidos no processo.

### 2.3.2 *void fs\_debug()*:

Esta função percorre todos os inodos e mostra seus tamanhos, assim como os blocos (diretos e indiretos) para onde estes apontam. Mas antes disso ela chama as funções *printinodemap()* e *printbitmap()* que ajudam o programador a ver como o sistema está se comportando.

### 2.3.3 *int fs\_mount()*:

É responsável pela criação do mapa de inodos e do *bitmap* (mapa de blocos). Os mapas são inicializados com valores padrão (inodos inválidos e blocos livres). Da mesma forma que em *fs\_debug()* esta função percorre todos os inodos para armazenar as informações em *inodemap* assim como percorre todos os blocos para onde estes apontam para marcá-los como ocupados. No fim de sua execução muda a variável global *\_mounted* para dar “sinal verde” às outras funções deste sistema.

### 2.3.4 *int fs\_create()*:

Reserva um inodo e inicializa-o como vazio. Para isso percorre o mapa de inodos até encontrar um inodo inválido e o valida. Também altera o valor no *inodemap* para indicar que o inodo está válido.

Ao se desenvolver esta função percebeu-se um erro de enunciado de trabalho: Os inodos começam da posição zero mas o protótipo foi feito de forma que o retorno zero indica um erro. Para manter a portabilidade do código com o *shell* optou-se por simplesmente considerar que o inodo zero sempre é válido e vazio (e ignorá-lo). Desta função em diante, sempre será necessário verificar se o disco está montado (variável *\_mounted*).

### 2.3.5 *int fs\_delete(int inumber)*:

Esta função carrega o inodo na memória, libera todos os seus blocos (diretos e indiretos) e invalida-o. Além disto a sua posição no mapa de inodos é colocada como inválida e cada bloco liberado tem seu valor alterado no *bitmap*.

#### 2.3.6 `int fs_getsize( int inumber )`:

Esta função simplesmente carrega o inodo na memória e retorna o seu atributo *size*.

#### 2.3.7 `int fs_read( int inumber, char *data, int length, int offset )`:

Esta foi uma das funções mais difíceis de se implementar. A dificuldade estava no fato de a interface *shell* usar um *buffer* de tamanho fixo e ir incrementando o *offset* repetidas vezes e não foi simples conciliar os retornos de erro com os retornos precisos de *bytes* lidos. Além disto o *offset* aumentou a complexidade da função, que foi separada basicamente em duas partes: A leitura dos blocos diretos e a leitura dos blocos indiretos do arquivo. A leitura começava em *offset*, verificava se este *offset* estava em um bloco direto ou indireto e, após isso, começava uma leitura de todos os blocos até que uma das duas condições fosse atingida: fim do arquivo ou número de bytes totais requisitados lidos.

Uma comparação com a função `fs_getsize()` foi suficiente para implementar a primeira condição. Já para a segunda condição uma variável auxiliar *remanescente* foi utilizada. Ela começou com o atributo *length* e ia sendo decrementada a cada leitura até ser zerada.

Para a cópia dos dados foi usada função `std::copy`, disponível na biblioteca *algorithm*, padrão da linguagem C++.

#### 2.3.8 `int fs_write( int inumber, const char *data, int length, int offset )`:

Devida a sua grande complexidade esta foi a função que o grupo teve maior dificuldade em implementar. O gerenciamento do *offset* junto com a *length*, as verificações de necessidade de alocação de memória e a própria alocação de memória tornaram esta a função mais complexa. Além disto, quaisquer erros nesta função corriam um sério risco de danificar os discos virtuais o que, embora fácil de resolver, demandava tempo de substituição.

Também é dividida em duas partes, uma gerenciando os blocos diretos e outra gerenciando os blocos indiretos. Mas antes uma verificação importante foi feita: Se a escrita ultrapassasse os limites do arquivo uma alocação de memória seria necessária (se houvesse espaço no disco, claro). Havendo esta necessidade aqui utilizou-se as funções auxiliares `blocoDisponivel()` e `espacoLivre()` para a alocação dos blocos de memória.

Aqui também foi usada a função `std::copy` para copiar os dados.

## 3 Considerações Finais

O projeto permitiu ampliar e verificar na prática os conceitos apresentados e discutidos de maneira teórica em sala de aula, permitindo que de maneira segura, graças ao uso de discos virtuais, o entendimento do desenvolvimento e funcionamento de um sistema real de arquivos.

Todo o projeto está disponível no seguinte endereço *url*:

<https://github.com/AndreLuigiB/simplefs/>