

Sistemas de Arquivos: Implementação do SimpleFS

José Peteck Junior e Vitor Diego Dias Engers
Universidade Federal de Santa Catarina
Centro Tecnológico de Joinville - CTJ
Joinville-SC

Resumo

A finalidade deste relatório consiste em implementar o sistema de arquivo denominado **SimpleFS**, expondo como é composta sua organização de dados. A noção de um sistema de arquivo simples como o *SimpleFS* serve de base para o entendimento de casos mais complexos presente no *Unix*. Acerca da implementação, também será discutido sobre a recuperação de sistema de arquivo, implementando uma varredura de *bitmap* de blocos livres com a utilização extensiva dos recursos da linguagem C++ sob um viés estruturado.

Palavras-chave

Sistemas de arquivo, SimpleFS, Sistemas Operacionais, Programação.

I. INTRODUÇÃO

Todas as aplicações que um computador utiliza, necessitam **armazenar** e **recuperar** as suas informações. A armazenagem das aplicações antigamente, era realizado em discos magnéticos, porém sua estrutura de armazenagem ainda se assemelha como a de hoje em dia. Essa estrutura pode ser traduzida em sequências lineares de blocos de tamanho fixo e que suportam as operações de: **Ler** e **escrever** em um determinado bloco.

Com a abordagem desses blocos de dados, é necessário introduzir o conceito de **arquivo** já que o intuito é reunir um quantidade desses blocos para se compor um arquivo propriamente dito. Os arquivos são unidades lógicas de informação criada pelos processos. Eles são independentes e devem sempre existir (a menos que um usuário ou processo intencionalmente o remova). Além disso o sistema que controla a organização dos arquivos é o **Sistema Operacional**, a essa tarefa é atribuída a conotação de Sistema de arquivos.

O *SimpleFS* é um sistema de arquivo similar à camada de inodos do *Unix* e que contém 3 camadas principais: a *shell*, o disco emulado e o sistemas de arquivos propriamente dito. No que se refere ao escopo do relatório, cabe apenas aos procedimentos relativos ao Sistema de arquivos, que fará leituras e escritas no disco emulado virtualmente. As leituras e escritas do disco emulado, possibilitam funções disponibilizadas ao usuários como a de: *debug*, *format*, *mount*, *create*, *delete*, *getsize*, *read* e *write* que serão explicadas ao longo do texto.

II. DESENVOLVIMENTO

Primeiramente, no projeto de implementar o *SimpleFS* como um todo, é necessário pontuar as condições fornecidas previamente pela proposta do trabalho. Os recursos de auxilio fornecidos para a implementação foram:

- A *shell* em linguagem C
- As imagens de disco
- As operações realizadas no disco em linguagem C
- As estruturas básica dos blocos inodos em linguagem C
- O cabeçalho da funções do usuário em linguagem C.

Reunindo todos os recursos necessários para a implementação, a primeira tarefa consistiu em alterar os códigos que estavam em linguagem C, e converte-los para linguagem C++. As estruturas de dados bases foram mantidas para manter a compactabilidade de futuras análises.

Em segundo momento, foi então implementado a função "*fs_debug()*" que, segundo a descrição do trabalho, é a função que serve de base para o entendimento do projeto e para as demais funções. Código fonte da função *fs_debug()* é descrito a seguir, no Listing 1.

```
1 void fs_debug () {
2     if (MOUNTED == false) {
3         std::cout << "Error: please mount first!" << std::endl;
4         return;
5     }
6     union fs_block block;
7     disk_read (0, block.data);
8     // SUPERBLOCK
9     if (block.super.magic != static_cast <int> (FS_MAGIC)) {
10         std::cout << " magic number is not valid" << std::endl;
11         return;
12     }
13     std::cout << "superblock: " << std::endl;
14     std::cout << " magic number is valid" << std::endl;
```

```

15     std::cout << "    " << block.super.nblocks << " blocks" << std::endl;
16     std::cout << "    " << block.super.ninodeblocks << " inode blocks" << std::endl;
17     std::cout << "    " << block.super.ninodes << " inodes" << std::endl;
18
19     // INODE BLOCK
20     for(int n_inode_block = 1; n_inode_block <= block.super.ninodeblocks; n_inode_block++)
21     {
22         union fs_block _inode;
23         disk_read(n_inode_block, _inode.data);
24
25         for (unsigned int _n_inodes = 0; _n_inodes < INODES_PER_BLOCK; _n_inodes++) {
26
27             if (_inode.inode[_n_inodes].isvalid) {
28                 std::cout << "inode " << _n_inodes << ":" << std::endl;
29                 std::cout << "    " << "size: " << _inode.inode[_n_inodes].
30                     size << " bytes" << std::endl;
31                 std::cout << "    " << "direct blocks: ";
32
33                 for (unsigned int _direct = 0; _direct < POINTERS_PER_INODE;
34                     _direct++) {
35
36                     if (_inode.inode[_n_inodes].direct[_direct])    std:::
37                         cout << _inode.inode[_n_inodes].direct[_direct] <<
38                             " ";
39
40                 }
41                 std::cout << std::endl;
42
43                 if (_inode.inode[_n_inodes].indirect) {
44                     std::cout << "    indirect block: " << _inode.inode[
45                         _n_inodes].indirect << std::endl;
46                     std::cout << "    indirect data blocks: ";
47
48                     union fs_block _indirect_block;
49                     disk_read(_inode.inode[_n_inodes].indirect,
50                         _indirect_block.data);
51
52                     for (unsigned int _indirect = 0; _indirect <
53                         POINTERS_PER_BLOCK; _indirect++) {
54                         if (_indirect_block.pointers[_indirect])
55                             std::cout << _indirect_block.
56                                 pointers[_indirect] << " ";
57
58                     }
59                     std::cout << std::endl;
60                 }
61             }
62         }
63     }
64 }

```

Listing 1. Implementação de fs_debug() em fs.c

A função debug, começa verificando se a imagem foi montada, caso contrario, ela relata um erro. Em seguida, uma *union* é criada, e irá receber os dados (realizado na leitura "disk_read()"), ao receber o superbloco então, é realizado a verificação se o numero mágico é o esperado, caso não seja, é informado o erro. Se o numero mágico for válido, então as informações do superbloco são impressas. Após isso, o programa faz a varredura nos blocos inodos e por conseguinte nos inodos dentro dos blocos. Para a varredura ser efetivada, o inodo deve conter ser valido (valor da variável isvalid = 1). Em seguida a varredura também é realizada nos blocos indiretos apontados (se houver) realizando a sua impressão par o usuário.

A função *fs_format()* é responsável por criar um novo sistema de arquivos no disco, destruindo qualquer dado presente no disca anterior, sua implementação pode ser observada no Listing 2 a seguir.

```

1 int fs_format () {
2     if (bitmap_is_valid ()) return 0;
3
4     //LIMPANDO OS INODE BLOCKS
5
6     //Varredura sobre blocos de dados inodos.

```

```

7      union fs_block block;
8      disk_read (0, block.data);
9      for (int n_inode_block = 1; n_inode_block <= block.super.ninodeblocks; n_inode_block
10         ++ ) {
11          union fs_block _inode;
12          disk_read(n_inode_block, _inode.data);
13
14          // varredura de inodo
15          for (unsigned int _n_inodes = 0; _n_inodes < INODES_PER_BLOCK; _n_inodes++) {
16
17              if (_inode.inode[_n_inodes].isvalid){
18                  _inode.inode[_n_inodes].isvalid = 0;
19                  _inode.inode[_n_inodes].size = 0;
20
21                  for (unsigned int _direct = 0; _direct < POINTERS_PER_INODE; _direct++)
22                      {
23                          _inode.inode[_n_inodes].direct[_direct] = 0;
24                      }
25                  _inode.inode[_n_inodes].indirect = 0;
26              }
27          }
28          disk_write (n_inode_block, _inode.data);
29      }
30      std::cout << "after format:" << std::endl;
31      for (unsigned int i = 0 ; i < bitmap.size(); i++)          std::cout << "block :" << i <<
32          ". state: " << bitmap[i] << std::endl;
33      return 1;
34  }

```

Listing 2. Implementação de fs_format() em fs.c

A implementação de *fs_format()* primeiramente acessa o superbloco (da mesma maneira que acessa em *fs_debug*). Com o acesso ao superbloco, se tem as informações dos inodos (quantidade, tamanho, etc). A formatação consistirá então em invalidar o inodo (*isvalid = 0*), configurar seu tamanho como zero e desreferenciar os blocos diretos e indiretos. Fazendo com que os inodos percam a referencias dos outros blocos. Os dados que foram apontados anteriormente serão vistos como lixo de memória e serão sobrescritos.

A função *fs_mount()* analisa o disco para um sistema de arquivo. Se o disco esta presente, ela irá ler seu superbloco, construir um *bitmap* de blocos livres e preparar o sistema para uso. A implementação de *fs_mount()* é descrita no Listing 3 a seguir.

```

1  int fs_mount () {
2      union fs_block block;
3      disk_read (0, block.data);
4      if (block.super.magic != static_cast <int> (FS_MAGIC)) {
5          std::cout << " magic number is not valid" << std::endl;
6          return 0;
7      }
8      if (!bitmap_is_valid ()) set_bitmap ();
9
10     MOUNTED = true;
11     return 1;
12 }

```

Listing 3. Implementação de fs_mount() em fs.c

A função *fs_mount* faz a leitura do superbloco no começo, para verificar o numero mágico. Caso o numero mágico for valido, começa a configurar o *bitmap*. O *bitmap* será criado apenas se ele não for valido (não foi criado anteriormente). O *bitmap* é um vetor que leva a informação se um bloco do disco é valido ou não, é uma variável global do fs.cpp.

A função *fs_create()* que pode ser observada no Listing 4, tem como objetivo criar um inodo de comprimento zero.

```

1  int fs_create () {
2      if (MOUNTED == false) {
3          std::cout << "Error: please mount first!" << std::endl;
4          return -1;
5      }
6      union fs_block block;
7
8      disk_read (0, block.data);
9
10     //Varredura sobre blocos de dados inodos.

```

```

11     for(int n_inode_block = 1; n_inode_block <= block.super.ninodeblocks; n_inode_block++)
12     {
13         union fs_block _inode;
14         disk_read (n_inode_block, _inode.data);
15
16         for (unsigned int _n_inodes = 1; _n_inodes < INODES_PER_BLOCK; _n_inodes++) {
17
18             //verifica se o inodo esta vazio( nao for valido)
19             if (!_inode.inode[_n_inodes].isvalid) {
20                 _inode.inode[_n_inodes].isvalid = 1;
21                 _inode.inode[_n_inodes].size = 0;
22
23                 for (unsigned int i = 0; i < POINTERS_PER_INODE; i++) _inode.
24                     inode[_n_inodes].direct[i] = 0;
25                 _inode.inode[_n_inodes].indirect = 0;
26
27                 disk_write (n_inode_block, _inode.data);
28
29                 return _n_inodes;
30             }
31         }
32         // todos os inodos ocupados ou falha.
33         return 0;
34     }
35 }

```

Listing 4. Implementação de `fs_create()` em `fs.c`

O que é realizado em `fs_create` consiste em varrer os inodos, e verificando quais estão disponíveis para serem criados (conter o `isvalid = 0`), caso consiga encontrar, a função configura o `isvalid = 1`, e atribui o tamanho de zero unidades. Além disso, são configurados as informações iniciais dos inodos diretos e indiretos como zero. Feito isso, a função retorna o numero do inodo que foi criado, se houver espaço para criar um novo inodo.

a função `fs_delete()`, que é mostrada no Listing 5, irá deletar um inodo que foi passado como parâmetro.

```

1 int fs_delete(int inumber) {
2
3     if (MOUNTED == false) {
4         std::cout << "Error: please mount first!" << std::endl;
5         return -1;
6     }
7
8     union fs_block block;
9
10    disk_read (0, block.data);
11
12    union fs_block _inode;
13    //leitura de disco a partir do bloco informado (multiplo de INODES_PER_BLOCK)
14    disk_read(1 + (inumber / static_cast <int> (INODES_PER_BLOCK)), _inode.data);
15    // recebe o bloco de inodos.
16
17    //verifica se o inodo esta vazio( nao for valido)
18    if (_inode.inode[inumber % INODES_PER_BLOCK].isvalid) {
19
20        _inode.inode[inumber % INODES_PER_BLOCK].isvalid = 0;
21        _inode.inode[inumber % INODES_PER_BLOCK].size = 0;
22
23        for (unsigned int i = 0; i < POINTERS_PER_INODE; i++) _inode.inode[inumber %
24            INODES_PER_BLOCK].direct[i] = 0;
25        _inode.inode[inumber % INODES_PER_BLOCK].indirect = 0;
26
27        disk_write (1 + (inumber / static_cast <int> (INODES_PER_BLOCK)), _inode.data);
28        // retorna o numero do inodo criado.
29        return 1;
30    }
31    return 0;
32 }

```

Listing 5. Implementação de `fs_delete()` em `fs.c`

A função *fs_delete()* possui o comportamento similar ao da *fs_create()*. É feito a varredura nos inodos até que se encontre o inodo a ser removido. Ao encontrar o inodo a ser removido, configura seus atributos (*isvalid* e *size*) como zero, e desreferencia os ponteiros diretos e indiretos. Feito isso, é escrito no disco as modificações que foram realizadas. O caso de falha é quando a função não encontra o inodo informado como parâmetro.

A função *fs_getsize* possui como retorno o tamanho lógico do inodo especificado, em bytes. A sua implementação pode ser observada no listing 6 a seguir.

```

1 int fs_getsize (int inumber) {
2
3     //retorna -1 caso nao tenha montado a imagem
4     if (MOUNTED == false) {
5         std::cout << "Error: please mount first!" << std::endl;
6         return -1;
7     }
8
9     //recebe o bloco de dados
10    union fs_block block;
11    disk_read (0, block.data);
12
13    unsigned int _n_blocks = 0;
14
15    //se o inumber eh valido
16    if (inumber < block.super.ninodes && block.inode[inumber/INODES_PER_BLOCK].isvalid) {
17        union fs_block _inode;
18
19        disk_read (inumber/INODES_PER_BLOCK + 1, _inode.data);
20
21        for (unsigned int _direct = 0; _direct < POINTERS_PER_INODE; _direct++) {
22            if (_inode.inode [inumber%INODES_PER_BLOCK].direct[_direct]) _n_blocks
23                ++;
24        }
25
26        if (_inode.inode[(inumber % static_cast <int> (INODES_PER_BLOCK))].indirect !=
27            0) {
28            std::cout << "   piahedaoqujshfdoiashdoiashd" << std::endl;
29            union fs_block _indirect;
30            disk_read (_inode.inode[(inumber % static_cast <int> (INODES_PER_BLOCK))
31                ].indirect, _indirect.data);
32            for (unsigned int n_indirect = 0; n_indirect < POINTERS_PER_BLOCK;
33                n_indirect++) {
34                if (_indirect.pointers[n_indirect]) _n_blocks++;
35            }
36        }
37        return _n_blocks*4096;
38    }
39    return -1;
40 }

```

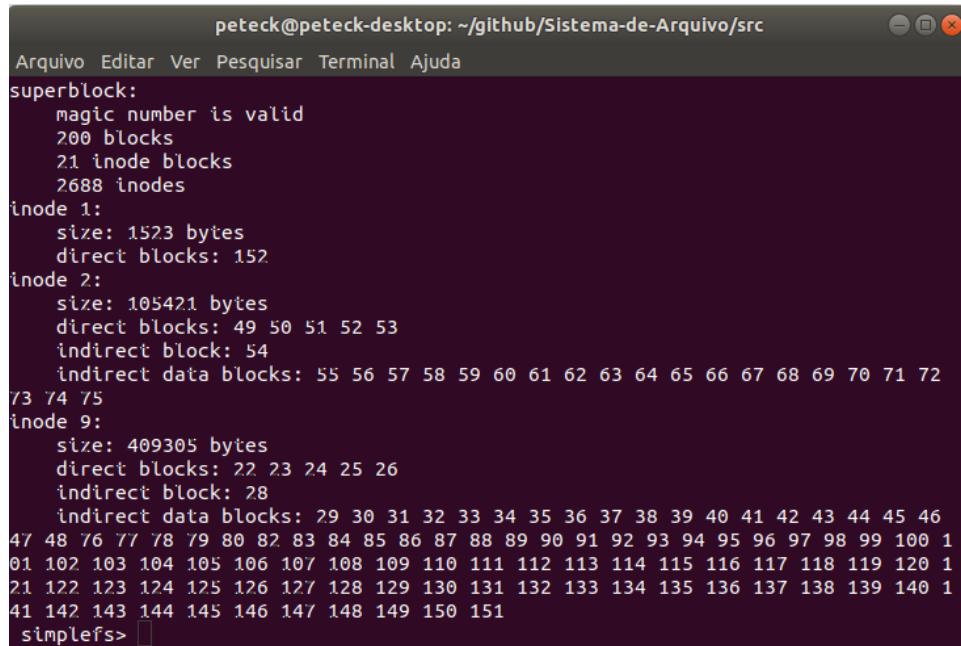
Listing 6. Implementação de *fs_getsize()* em *fs.c*

O funcionamento da função *fs_getsize()* é interrompido, se o inumber passado como referência for inválido. Se for válido, será realizado uma leitura de disco e um laço para somar cada bloco direto e indireto. ao final dos dois laços de repetições o acumulador (*_n_blocks*) é então multiplicado por 4096 (que é o tamanho por bloco), retornando esse valor, desse modo.

A função *fs_read()* tem como proposito, ler um dado de um inodo que seja valido. possui como retorno o numero total de bytes lidos. Não será exposto seu código fonte, devido ao seus extenso comprimento, no entanto, será explicado o seu funcionamento a seguir. A função *fs_read()* só pode ler 16KB por vez, devido a limitação imposta pelo *Shell*. Como cada bloco possui 4KB, ela irá ler 4 blocos por vez. Além disso, ela trabalha com o *offset* passado como argumento. Ela lê os blocos diretos e também os indiretos (se houver).

A função *fs_write()* tem como proposito, escrever um dado para um inodo que seja valido. Ela alocará dados para blocos diretos e/ou indiretos. O seu código fonte também não será exposto neste escopo devido sua grande extensão. A função *fs_write()* então, funcionará da mesma maneira que a função *fs_read()* porém as leituras serão agora escritas no disco. Todos os detalhes dessas duas ultimas funções podem ser observadas diretamente no código fonte, onde estão devidamente comentada, explicando seu passo-a-passo.

Por fim, com todos os métodos descritos implementados, a Figura 1 a seguir demonstra um exemplo de execução do *simpleFS*.



```
peteck@peteck-desktop: ~/github/Sistema-de-Arquivo/src
Arquivo Editar Ver Pesquisar Terminal Ajuda
superblock:
  magic number is valid
  200 blocks
  21 inode blocks
  2688 inodes
inode 1:
  size: 1523 bytes
  direct blocks: 152
inode 2:
  size: 105421 bytes
  direct blocks: 49 50 51 52 53
  indirect block: 54
  indirect data blocks: 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
73 74 75
inode 9:
  size: 409305 bytes
  direct blocks: 22 23 24 25 26
  indirect block: 28
  indirect data blocks: 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
47 48 76 77 78 79 80 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 1
01 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 1
21 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 1
41 142 143 144 145 146 147 148 149 150 151
simplefs>
```

Figura 1. Terminal do Linux executando o programa SimpleFS

A Figura descreve uma execução do programa com alguns comandos na *Shell* do *simpleFS* e os resultados impressos no terminal do usuário, para a verificação.

III. CONCLUSÃO

A implementação do *SimpleFS* demonstrou uma aplicação prática e de um Sistema de Arquivos - ainda que rudimentar - composto de operações básicas presente ao usuário, o que mostra um caráter didático, neste quesito. A implementação ainda serviu de base para consolidar os conteúdos aprendidos na matéria de Sistemas Operacionais, principalmente no que se refere a estrutura dos blocos de dados, seus identificadores e os tipos de blocos presentes no arquivo. Além disso, todos esses conceitos puderam ser aplicados em termos de linguagem de programação C++, consolidando os tópicos que eram largamente teóricos em puramente práticos.