

Amanda Ferrari
Catarine Soares Cruz
Gustavo Zanzin Guerreiro Martins

Implementação de Sistema de Arquivos EXT2

Relatório técnico do projeto prático
solicitado pelo professor Rodrigo Campiolo
na disciplina de Sistemas Operacionais do
Bacharelado em Ciência da Computação da
Universidade Tecnológica Federal do
Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR
Departamento Acadêmico de Computação – DACOM
Bacharelado em Ciência da Computação – BCC

Campo Mourão

Dezembro / 2022

Resumo

O presente relatório tem como objetivo descrever os passos utilizados para realizar a implementação do sistema de arquivos EXT2, sua respectiva configuração e execução. Dessa forma, para melhor compreender sua estrutura básica foram propostos alguns comandos a serem realizados e analisar suas respectivas saídas. Nesse sentido, foram obtidos os resultados esperados dessa experiência e portanto, assimilou-se os motivos e a melhor compreensão de alguns comandos, como também entendeu-se a definição e a operação de um sistema de arquivos.

Palavras-chave: EXT2. sistema de arquivos. implementação.

Sumário

1. Introdução	4
2. Fundamentação	4
3. Descrição da Atividade	4
4. Materiais	4
5. Métodos e Divisão da Atividade	5
6. Resultados e Discussão	11
7. Bugs Conhecidos	11
9. Conclusões	12
10. Referências	12

1. Introdução

O sistema operacional (SO) Linux foi desenvolvido de forma cruzada sob outro SO, o Minix. Nesse viés, foi pensado que seria mais fácil compartilhar discos entre dois sistemas do que projetar um novo sistema de arquivos. Todavia, não foi o ocorrido, as restrições no projeto do sistema de arquivo do Minix limitavam bastante. Dessa forma, começou-se a implementação de um novo sistema de arquivos no Linux, surgindo, assim, após algum tempo o EXT2.

2. Fundamentação

O sistema de arquivos EXT2 contém alguns conceitos básicos derivados do sistema operacional Unix. Contém conceitos como: inodes, diretórios, links, entre outros.

Nesse sentido, os inodes, importantes estruturas, são uma forma de representar cada arquivo, contendo a descrição (tipo do arquivo, direitos de acessos, proprietários...) e endereços dos blocos de dados alocados. Já os diretórios, estão em um formato de árvore hierárquica, podendo conter arquivos e subdiretórios. Os diretórios são nada mais que arquivos com uma lista de entrada, que contém, por sua vez, um número de inode e um nome de arquivo. Além disso, adicionar um link consiste em criar uma entrada de diretório, no qual o número do inode aponta para o inode.

Portanto, é possível perceber que o EXT2 não é um sistema trivial e será utilizado a teoria a prática para compreendê-lo melhor.

3. Descrição da Atividade

A partir da imagem (.iso) fornecida houve a implementação de diversas estruturas de dados, a fim de manipular a imagem do sistema de arquivos EXT2. Dessa forma, utilizando um prompt (shell) para a invocação dos comandos implementados (info, cat, attr, cd, ls, pwd e cp), será executado as operações desejadas, a partir da referência do diretório corrente, e mostrará a resposta obtida.

4. Materiais

Os materiais utilizados para a realização deste trabalho foram: um editor de código e uma imagem EXT2. Mostrados a seguir:

- Editor de código: Visual Studio Code 1.70;
- Imagem ext2 (64 MiB).

Todos em suas últimas versões estáveis até o presente momento da elaboração deste relatório; a saber 06/12/2022.

5. Métodos e Divisão da Atividade

Foi implementado as funções *info*, *cat*, *attr*, *cd*, *ls*, *pwd* e *cp*. Elas serão descritas e detalhadas a seguir.

5.1 info

O comando *info* consiste em exibir informações do disco e do sistema de arquivos. Para realizar essa função é necessário ter conhecimento da estrutura do superbloco, que abrange todas as informações do sistema de arquivos, tais como as informações que serão mostradas ao executar o comando *info*, conforme a Figura 1. Compreende-se também que ao guardar as informações do EXT2 é uma estrutura extremamente importante.

```
void read_super_block() {

    printf(
        "Volume name.....: %s\n"
        "Image size.....: %u bytes\n"
        "Free space.....: %u KiB\n"
        "Free inodes .....: %u\n"
        "Free blocks .....: %u\n"
        "Block size.....: %u bytes\n"
        "Inode size.....: %hu bytes\n"
        "Groups count.....: %u\n"
        "Groups size.....: %u blocks\n"
        "Groups inodes.....: %u inodes\n"
        "Inodetable size.....: %lu blocks\n"
        ,
        super.s_volume_name,
        (super.s_blocks_count * block_size),
        ((super.s_free_blocks_count - super.s_r_blocks_count) * block_size) / 1024,
        super.s_free_inodes_count,
        super.s_free_blocks_count,
        block_size,
        super.s_inode_size,
        (super.s_blocks_count/super.s_blocks_per_group),
        super.s_blocks_per_group,
        super.s_inodes_per_group,
        (super.s_inodes_per_group/(block_size/sizeof(struct ext2_inode)))
    );
}
```

Figura 1 - comando *info*.

5.2 cat

O comando cat tem a função de exibir o conteúdo de um arquivo no formato texto. Para realizar essa função é necessário, como dito anteriormente, estruturas chamadas de inode. No EXT2, eles estão organizados da seguinte forma: 12 ponteiros diretos, 1 ponteiro indireto, 1 ponteiro duplamente indireto e 1 ponteiro triplamente indireto - entretanto, nesta implementação específica, o nível de indireção tripla foi desconsiderado. Esses ponteiros são lidos conforme a necessidade do arquivo, ou seja, o arquivo hello.txt, por exemplo, foi armazenado nos ponteiros diretos, não necessitando realizar a leitura com indireção, de acordo com a imagem 2.

```
// Percorrendo pelos blocos de dados sem indireção
for(int i = 0; i < 12; i++) {

    lseek(fd, BLOCK_OFFSET(inodeEntryTemp->i_block[i]), SEEK_SET);
    read(fd, block, block_size); // Lê bloco i em block

    // Exibindo conteúdo do primeiro bloco
    for(int i = 0; i < 1024; i++) {
        printf("%c", block[i]);

        arqSize = arqSize - sizeof(char); // Quantidade de dados restantes

        if(arqSize <= 0) {
            break;
        }
    }
    if(arqSize <= 0) {
        break;
    }
}
```

Figura 2 - Leitura dos ponteiros diretos.

Nesse viés, ao realizar a leitura de um arquivo maior e mais complexo como o Biblia.txt, será necessário a leitura dos ponteiros com indireção, conforme a ilustração 3.

```

// Se após os blocos sem indireção ainda existirem dados,
// percorre o bloco 12 (uma indireção)
if(arqSize > 0) {

    lseek(fd, BLOCK_OFFSET(inodeEntryTemp->i_block[12]), SEEK_SET);
    read(fd, singleIndirection, block_size);

    for(int i = 0; i < 256; i++) {

        lseek(fd, BLOCK_OFFSET(singleIndirection[i]), SEEK_SET);
        read(fd, block, block_size);

        for(int j = 0; j < 1024; j++) {

            printf("%c", block[j]);
            arqSize = arqSize - 1;

            if (arqSize <= 0) {
                break;
            }
        }
        if (arqSize <= 0) {
            break;
        }
    }
}

```

Figura 3 - Leitura dos ponteiros indiretos.

Para a leitura tanto dos ponteiros com uma, duas ou três indireções (não implementado), é preciso percorrer 256 ponteiros que apontam para outros 1024 ponteiros, cada. Dessa forma o código da figura 3 pode ser replicado para as outras indireções. Além disso, é importante realizar a navegação entre grupos, pois o conteúdo de um arquivo pode começar em um grupo e terminar em outro, por exemplo. Assim, necessita-se de uma função auxiliar, *change_group*, que será explicada mais adiante, para realizar a navegação entre grupos. Logo, será possível ler o conteúdo do arquivo, não importando a sua localização em relação aos níveis de inode e exibir toda a informação na tela do prompt, executando a função do cat e se, caso o arquivo não exista, retorna uma mensagem de arquivo não encontrado.

5.3 attr

Para a implementação do comando attr, que exibe os atributos de um arquivo ou diretório, foi necessário, também, da função auxiliar *change_group*, para realizar, se preciso, a troca de grupo. Ao encontrar, pelo nome, o inode correspondente, será lido os atributos desses arquivo, tais como: as permissões, o identificador de usuário (uid), o identificador de grupo (gid), tamanho e quando foi modificado.

As permissões podem ser de leitura, escrita ou execução. Além disso, ela está relacionada a quem tem essa permissão, podendo ser, em ordem, do usuário, do grupo ou de outros. É mostrado, também, o tipo, se é um arquivo ou um diretório. Para mostrar as permissões, com o formato de letras, foi necessário utilizar a tabela dos valores do *i_mode* definidos, direitos de acesso, para decodificar da forma correta. O tamanho é fornecido em bytes, portanto, para ser mostrado em kibibyte (KiB), foi utilizado um tratamento. Não diferente, o período de modificação necessitou de um tratamento, pois o valor fornecido é em segundos. Foi utilizado a biblioteca *time.h* para essa função. Após isso, é possível concretizar a função do *attr*.

5.4 cd

O comando *cd* permite ao usuário navegar pelos diretórios do sistema de arquivos. Para tanto, sua implementação consistiu em exibir, caso encontre o diretório desejado, alguns atributos sobre o mesmo, como o nome, o número do inode, o espaço gravado desse registro, entre outros. Além disso, o *change directory* (*cd*) foi efetuado utilizando a troca de grupo; o cálculo do index real do inode do novo grupo - a saber, inode da entrada de diretório encontrada módulo a quantidade de inodes por grupo; a atualização do inode do novo grupo - realizada por meio da função *read_inode*. Com isso, finalmente, terá navegado para outro diretório. Entretanto, caso a entrada de diretório não seja compatível com o nome desejado pelo usuário (por razões como a pasta não existir, por exemplo), o sistema de tratamento de erros retorna uma mensagem informando que o diretório não foi encontrado.

5.5 ls

O comando *ls*, quando invocado, percorre por todas as entradas de diretório do diretório corrente, listando os atributos de arquivos e subdiretórios. Para isto, ao ler o bloco da imagem, é calculada a primeira entrada do diretório em questão e para a iteração por cada entrada de diretório (*struct ext2_dir_entry_2*) que se faz presente, é exibido na tela atributos como o nome, o número do inode, o espaço gravado desse registro, entre outros.

5.6 pwd

O comando *pwd* ao ser invocado exibirá o diretório corrente. Dessa forma, foi utilizado uma estrutura auxiliar, pilha, para poder empilhar os diretórios. Para acontecer esse empilhamento, foi utilizado uma condição dentro da implementação do comando *cd*, como a figura 4 abaixo. Se o usuário entrar em um diretório será empilhado na pilha, utilizando a função *PUSH* (insere na última posição do vetor), se o usuário quiser voltar um diretório, utilizando o *..*, será desempilhado, com a função *POP* (decrementa o vetor em 1). Posteriormente, quando a função for executada, será mostrado o conteúdo da pilha, usando a função *mostra*.


```

if((strcmp(entry->name, "..") != 0) && (strcmp(entry->name, ".") != 0)){
    PUSH(entry->name, stack);
}
else if(strcmp(entry->name, "..") == 0){
    POP(stack);
}

```

Figura 4 - condição de empilhamento ou desempilhamento.

5.7 cp

O comando `cp` possibilita que o usuário copie o conteúdo de um determinado arquivo de texto (`.txt`) para outro arquivo de texto. A ideia inicial era usá-lo com a seguinte sintaxe: `cp origem.txt destino.txt`. Entretanto, foi constatado um problema de corretude devido a forma de implementação nos casos em que os arquivos informados no momento da inserção do comando tivessem o caractere “espaço” (na tabela ASCII, 32 em decimal), conforme também será abordado na seção Bugs Conhecidos. Para solucionar tal entrave, o caminho adotado foi utilizar aspas simples (‘’) envolvendo o nome dos arquivos que estarão envolvidos na operação de cópia, da seguinte forma: `cp 'origem.txt' 'destino.txt'`. Caso a sintaxe utilizada para tal comando não seja essa, o seu comportamento esperado não será realizado e, com isso, terá uma mensagem emitida pelo sistema de tratamento de erros informando que a sintaxe utilizada é inválida.

Para a implementação de tal comando, utilizou-se a função *copia_arquivo*, na qual aloca cópias do conteúdo do descritor de grupo e do inode recebidos por parâmetro; troca o grupo (quando necessário) para o que contém o novo inode com a função *change_group*; calcula o index real do inode no novo grupo para utilizá-lo na atualização do inode no novo grupo por meio da função *read_inode*. A posteriori, a função *copia_arquivo* também criará um arquivo (se ainda não existir) com o nome do terceiro parâmetro (arquivo de destino) para ser o arquivo de destino. E, então, começa a percorrer a estrutura do inode nos seus blocos diretos, ainda sem indireção, lendo caractere a caractere e escrevendo no arquivo informado, sendo seguido, portanto, pelos níveis simples e duplo de indireção. Ao fim do processo, é invocada a função *read_dir* para obter-se o inode do diretório corrente (.) a fim de realizar sua alteração para o mesmo por meio da *read_inode*.

5.8 Funções Auxiliares

Algumas funções auxiliares foram implementadas para facilitar a construção das outras, as principais serão discutidas aqui.

5.8.1 change_group

A função *change_group* consiste em alterar o grupo que está sendo lido atualmente para o que se deseja ler. Assim, ao receber o inode do arquivo ou diretório desejado é realizado um cálculo

para poder saber, a partir do inode, qual grupo de bloco que ele estará e trocar se for preciso, conforme a figura 5.

```
void change_group(unsigned int* inode, struct ext2_group_desc* groupToGo, int* currentGroup) {
    unsigned int block_group = ((*inode) - 1) / super.s_inodes_per_group; // Cálculo do grupo do Inode

    if (block_group != (*currentGroup))
    {
        *currentGroup = block_group;

        lseek(fd, BASE_OFFSET + block_size + sizeof(struct ext2_group_desc) * block_group, SEEK_SET);
        read(fd, groupToGo, sizeof(struct ext2_group_desc));
    }
}
```

Figura 5 - função que altera o grupo.

5.8.2 read_dir

A função *read_dir* consiste em ler um diretório e suas entradas retornando o valor do inode correspondente ao nome do arquivo passado. Dessa maneira, será lido as entradas desse diretório até que a condição da imagem 6 seja atendida. Quando a condição for verdadeira, não irá mais verificar as entradas e será devolvido o valor do inode do arquivo ou diretório desejado.

```
if((strcmp(nomeArquivo, file_name)) == 0) {
    return entry->inode;
}
```

Figura 6 - comparando o nome e retornando inode.

5.9 Prompt de Comando

O Prompt funciona como um interpretador de linha de comando, a cada comando digitado ele faz uma comparação com diversos comandos em um laço while. Esta comparação mencionada é feita utilizando a função *strcmp* da biblioteca *string.h* e é ela quem define o que será executado, sempre verificando o comando principal com as estruturas condicionais.

Ao executar o comando `cat hello.txt` por exemplo, a estrutura condicional verdadeira será aquela que verifica se o comando principal (`cat`) é igual a “cat”, com isso, inicia-se o protocolo de verificação de arquivo (`hello.txt`) e sintaxe no qual caso o arquivo informado não exista, o sistema de tratamento de erros exibirá uma mensagem informando que a sintaxe é inválida e a função `cat` não será invocada. No entanto, se o arquivo for encontrado, a função desejada será executada corretamente.

O tratamento e separação de cada palavra e/ou parâmetro de um determinado comando digitado no prompt de comando é gerenciado por algumas funções, variáveis e uma estrutura lógica de condicionais. O grupo de funções para realizar esse tratamento é composto por 4 delas - a saber, *catch_principal_param*, *catch_second_param*, *catch_second_param_cp* e *catch_third_param_cp*.

A primeira delas (*catch_principal_param*) é responsável por separar o primeiro/principal comando da entrada completa digitada pelo usuário. A segunda (*catch_second_param*) só será chamada para fazer a separação do segundo parâmetro para comandos que necessitam de, por exemplo, do nome de um arquivo informado para funcionarem (a exemplo de comandos como `cat` ou `attr`). Por outro lado, a terceira função (*catch_second_param_cp*) fará algo semelhante - capturando o segundo parâmetro digitado pelo usuário ou primeiro parâmetro do comando `cp`, mas para outro caso de uso; quando o parâmetro recebido estiver envolvido por aspas simples, que significa que o comando será o `cp`. E a quarta (*catch_third_param_cp*) também fará algo parecido com a anterior, entretanto tem como papel capturar o último parâmetro. Tais funções retornarão *strings* que serão utilizadas pela maioria das outras funções, portanto, necessitam ser atribuídas à algumas variáveis para que possam ser acessadas posteriormente. Por fim, a aplicação prática de tudo isso se dá na estrutura condicional que verifica se:

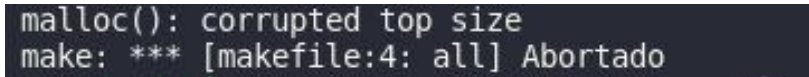
1. A *string* contém aspas simples. Se sim, significa que o comando em questão é um `cp` e portanto devem ser executadas as funções *catch_principal_param*, *catch_second_param_cp* e *catch_third_param_cp* a fim de **separar** o comando completo que o usuário digitou em comando principal, arquivo de origem e arquivo de destino, respectivamente.
2. A *string* contém espaço. Se sim, significa que o comando seja algum que necessite de pelo menos um parâmetro para funcionar e, portanto, deve executar as funções *catch_principal_param* e *catch_second_param_cp* a fim de separar o comando principal e o arquivo informado.
3. Caso contrário, ou seja, se a *string* não contiver nem ao menos um caractere espaço, o comando completo digitado pelo usuário é apenas copiado para a variável “comando” para simplificar a implementação das comparações posteriores com os possíveis comandos.

6. Resultados e Discussão

Na especificação do trabalho foi pedido para ser implementado as seguintes operações: *info*, *cat*, *attr*, *cd*, *ls*, *pwd*, *touch*, *mkdir*, *re*, *redir*, *rename*, *cp* e *mv*. Contudo, as operações implementadas foram as descritas acima. Logo, todas as operações que envolviam leitura foram implementadas e uma operação de cópia também foi feita. Os resultados obtidos são ótimos, pois foi alcançado o objetivo, comportando-se da forma esperada e obtendo os resultados desejados, fazendo ressalva aos bugs conhecidos, próximo tópico.

7. Bugs Conhecidos

Para a execução do comando `cp` é necessário que os nomes dos arquivos estejam em aspas simples (‘’), dessa forma: `cp 'arquivo1.txt' 'arquivo2.txt'`. Ao executar o comando `cp`, em sequência, em arquivos relativamente grandes ocorre o erro mostrado na imagem 7. Todavia, ao ser executado uma vez, ou em sequência com arquivos menores esse erro não ocorre.



```
malloc(): corrupted top size
make: *** [makefile:4: all] Abortado
```

Figura 7 - erro do cp.

9. Conclusões

Em suma, realizar a manipulação da imagem de um sistema de arquivos EXT2 proporcionou uma imersão no conteúdo de sistemas de arquivos. Desse modo, para realizar a prática era extremamente necessário dominar o conteúdo teórico, as estruturas como inode e suas indireções, blocos, grupo de blocos, entre outros foram de suma importância para a elaboração do projeto. Por conseguinte, realizar esse trabalho, trouxe um grande acréscimo intelectual a respeito do assunto.

10. Referências

ALTIERI, Emanuele, HOWE, Nicholas. **The Ext2 Filesystem**. 2002. Disponível em <https://www.science.smith.edu/~nhowe/262/oldlabs/ext2.html>. Acessado em 06/12/2022.

CARD, Rémy, TS'O, Theodore, TWEEDIE, Stephen. 2019. **Design and Implementation of the Second Extended Filesystem**. Disponível em <https://e2fsprogs.sourceforge.net/ext2intro.html>. Acessado em 06/12/2022.

POIRIER, Dave. **The Second Extended File System**. Disponível em <https://www.nongnu.org/ext2-doc/ext2.html>. Acessado em 06/12/2022.