Sistemas Operacionais - T2FS - Relatório

Camila Haas Primieri (00172662) Isadora Pedrini Possebon (00228551)

2016/2

1 Implementação de primitivas

1.1 Considerações gerais

Todas as primitivas implementadas verificam se as estruturas de dados da biblioteca já foram inicializadas através da leitura do superbloco do disco (função *initialize_data*). Caminhos e nomes de arquivos inválidos geram uma mensagem e retorno de erro na função, bem como chamadas de função para o tipo de arquivo errado (chamar *rmdir2* para remover um arquivo, por exemplo).

1.2 create2

A primitiva create2 foi implementada de forma que só é possível criar um novo arquivo regular se as condições abaixo são respeitadas.

- Há menos de 20 arquivos abertos no momento;
- O nome do arquivo contém apenas letras, números e o caractere '.';
- Todos os diretórios do seu caminho absoluto existem;
- Não existe arquivo com mesmo nome no diretório.

Respeitando a definição da primitiva, um i-node livre é alocado para o arquivo, um registro (*t2fs_record*) é criado e o bloco de dados ocupado pelo i-node é marcado como ocupado no bitmap correspondente. Essas estruturas de dados são, então, escritas no disco.

Um estrutura do tipo *file_descriptor* (implementada pelo grupo e definida abaixo) é criada. Esta estrutura tem o objetivo de guardar a posição atual do ponteiro dentro do arquivo (*current_pointer*), o registro do arquivo (*record*), o setor onde esse registro está localizado (*sector_record*) e sua posição dentro do setor (*record_index_in_sector*).

```
struct file_descriptor{
    struct t2fs_record record;
    int current_pointer;
    int sector_record;
    int record_index_in_sector;
};
```

Finalmente, um ponteiro para o *current_pointer* do arquivo é retornado, em caso de sucesso, e utilizado como handler do arquivo em questão (estrutura *FILE2*). Em caso de erro, a função retorna -1.

De acordo com os testes realizados, definidos na seção seguinte, a primitiva funciona conforme o esperado.

1.3 delete2

Para apagar um arquivo regular, procura-se pelo seu registro associado, de forma que se o caminho informado for inválido ou o não existir, um erro é retornado. Quando encontrado, o registro é invalidado, sua posição de memória é liberada e o correspondente i-node é marcado como livre no bitmap de i-nodes. Ainda, os blocos que o arquivo ocupava são desalocados, marcando cada um deles como livre no bitmap de dados.

A função funciona como o esperado, retornando 0 se executou corretamente e -1, em caso de erro.

1.4 open2

Mantém-se uma contador de arquivos abertos atualmente, para garantir que no máximo 20 arquivos possam estar abertos simultaneamente. Dessa forma, antes de abrir um arquivo, verifica-se se este número foi ultrapassado, retornando um erro em caso positivo. Também se verifica se armazena o handle do arquivo em uma lista, para que posteriormente se possa conferir que pertence a um arquivo aberto válido.

Para abrir o arquivo em questão, procuramos, com o caminho absoluto informado, pelo registro associado. Com o registro recuperado, cria-se um *file_descriptor* para este arquivo, posicionando o *current_pointer* em 0.

Em caso de sucesso, o handle deste arquivo é retornado e a lista de arquivos abertos é incrementada. Em caso de erro, retorna o valor -1.

1.5 close2

A função close2 procura pelo handle informado na lista dos arquivos abertos. Caso o handle informado não seja localizado, retorna -1. Senão, apaga a estrutura *file_descriptor* referente ao arquivo a ser fechado e decrementa o contador de arquivos abertos.

1.6 read2

Primeiro, a função read2 confere se o handle informado do arquivo é válido e se o arquivo a ser lido é do tipo regular. Após, recebe o *current_pointer* armazenado no *file_descriptor* e lê o i-node do arquivo. Por fim, executa um laço que vai percorrer todos os blocos que contém bytes a serem lidos.

Esta função assume que os dados do arquivo sejam escritos de forma sequencial nos blocos, iniciando pelos ponteiros diretos, em ordem, e depois pelo de indireção simples, até esgotada sua capacidade e, por último, pelo ponteiro de indireção dupla. Dentro de um bloco, são acessados os setores, sendo copiados os bytes solicitados para o *buffer* fornecido a partir da posição do *current_pointer*. Quando todos os bytes solicitados são lidos ou é atingido o final do arquivo, a função retorna o número de bytes efetivamente lidos.

1.7 write2

Inicia pela conferência da validade do handle informado e de que o arquivo é do tipo regular. Após, recebe o *current_pointer* armazenado no *file_descriptor* e lê o i-node do arquivo. Em seguida, localiza o bloco no qual está o *current_pointer* do arquivo e inicia a escrita no disco a partir deste ponto. Caso ainda existam bytes a serem escritos e foi atingido o final do bloco atual, é alocado um novo bloco para o arquivo.

Os blocos são sempre alocados em sequência, primeiramente os apontados pelos ponteiros diretos do i-node e, após, os blocos de indireção simples. Apenas após esgotada a capacidade do ponteiro de indireção simples, são alocados os blocos utilizando o ponteiro de indireção dupla. A escrita prossegue até atingir o final do buffer ou o tamanho máximo do arquivo. Se finalizada sem erros, será gravado no registro do arquivo em disco (*t2fs_record*) seu novo tamanho em blocos e bytes. A função retorna o número de bytes efetivamente escritos.

1.8 truncate2

Em primeiro lugar, a função confere se o arquivo é do tipo regular e se não é vazio. Em caso negativo, retorna erro. Em seguida, localiza-se o i-node do arquivo e o correspondente registro.

Para determinar quais blocos podem ser desalocados, verifica-se em qual bloco o *current_pointer* está e qual é o tamanho do arquivo. Com base no novo tamanho definido para o arquivo, desaloca-se os blocos necessários, atualiza-se o novo tamanho em bytes e blocos, e o registro do arquivo é escrito de volta no disco. Retorna 0 se executou corretamente; em caso de erro, -1.

1.9 seek2

A função seek2 apenas pode ser aplicada sobre arquivos abertos. Portanto, este é o primeiro teste realizado. Caso o arquivo não seja um arquivo regular, retorna-se um erro.

De acordo com a especificação, se o valor do offset enviado for -1, o *current_pointer* é posicionado no final do arquivo. Caso contrário, adiciona-se o valor do offset à posição do *current_pointer* e a função retorna 0. Em caso de erro, a função retorna -1.

1.10 mkdir2

A criação de um diretório requer que o seu nome seja válido, que um diretório com o mesmo nome não exista no caminho informado e que todos os diretórios presentes no seu caminho absoluto também existam.

Sabendo que o diretório raiz não possui um registro, testa-se se o diretório criado está em um subdiretório. Se não estiver, cria-se um registro temporário para o diretório raiz.

Em seguida, procura-se por um i-node livre; o seu valor, no bitmap de i-nodes, é alterado para ocupado. Este i-node é, então, formatado como um diretório; isto é, aloca novos blocos para o i-node, e todas as entradas *TypeVal* de registros recebem o valor 0x00. Depois de formatado, o i-node é escrito no disco.

O passo seguinte consiste em criar um registro para o novo diretório, inicializando-o com o tamanho de 1 bloco e 0 bytes, por escolha de implementação. Com este registro, uma estrutura do tipo *file_descriptor* é criada, inicializando o *current_pointer* em 0 e passando o números de setor e posição dentro do setor do diretório criado. A estrutura utilizada para referenciar estas posições é a *record_location*, definida abaixo e utilizada na implementação de todas as outras primitivas.

```
1 struct record_location {
2     int sector;
3     int position;
4 };
```

Com o processo finalizado, a função retorna 0 em caso de sucesso. Caso ocorra algum erro, a função retorna -1.

1.11 opendir2

Para ler um diretório, deve-se procurar pelo registro do diretório, passando o seu caminho absoluto. Quando o registro é encontrado, recupera-o e adiciona-o ao *file_descriptor* correspondente. Ainda, o handle deste diretório é adicionado à lista de diretórios abertos e o tamanho desta lista, incrementado.

Em caso de sucesso, retorna-se o handle deste diretório. Em caso de erro, -1.

1.12 readdir2

Para ler um diretório, verifica-se se o mesmo consta na lista de diretórios abertos. Em caso negativo, retorna o valor -1.

Diretórios vazios não podem ser lidos e, portanto, geram o retorno -1. Em relação aos diretórios que possuem pelo menos um arquivo, recupera-se o i-node do diretório, e realiza a leitura dos n bytes solicitados. Para definir onde a leitura deve ser iniciada, devemos descobrir em qual bloco o *current_pointer* está; para isso, testamos o seu valor.

Com o registro do diretório recuperado, utiliza-se a função auxiliar *readNthEntry*, que lê a entrada indicada pelo *current_pointer* do descritor do arquivo. Após finalizar a leitura, posiciona *current_pointer* no registro seguinte e retorna 0. Em caso de erro, retorna -1.

1.13 closedir2

Recebendo o handle do arquivo, que corresponde ao endereço onde o seu *file_descriptor* está, recupera-se o seu registro e, caso o arquivo seja do tipo diretório, o seu handle é retirado da lista de diretórios abertos. Caso contrário, retorna -1.

Após retirado da lista de diretórios abertos, libera-se o espaço de memória deste diretório e retorna 0.

1.14 rmdir2

Considerando que apenas diretórios vazios podem ser apagados, recupera-se o registro do diretório para acessar o seu i-node e, então, fazer essa verificação. Caso o diretório seja vazio, apaga o registro do disco, libera os blocos que estava utilizando e marca-os como livre no bitmap de dados. Caso o diretório não seja vazio, retorna -1.

2 Testes

Para cada primitiva implementa, foi desenvolvido, em média, um programa de teste. Cada programa visa avaliar o funcionamento da função em pelo menos um caso de erro e no seu caso de sucesso. Os arquivos de teste estão descritos abaixo.

- **test_cmd.c** Interpretador de comandos fornecido pelos professores da disciplina;
- **test_create2.c** Cria um arquivo com o nome informado pelo usuário e, em seguida, fecha-o.
- **test_delete.c** Apaga um arquivo já existente no disco, a partir do caminho absoluto informado.
- **test_initialize.c** Cria um arquivo qualquer, para verificar se a inicialização, a partir dos dados do superbloco, é realizada.
- **test_mkDir.c** Cria um novo diretório no disco.
- **test_open_close_dir.c** Abre um diretório já existente no disco e, em seguida, fecha-o.
- **test_open_close.c** Abre um arquivo regular já existente no disco e, em seguida, fecha-o.
- **test_read_duplo.c** Abre um arquivo regular já existente no disco e realiza a leitura de 5 bytes. Em seguida, realiza a leitura de mais 5 bytes deste mesmo arquivo.
- **test_readdir2.c** Abre um diretório já existente no disco e realiza a leitura de sua primeira entrada.
- **test_rmdir2.c** Cria dois arquivos novos no disco e dois novos arquivos regulares em um dos diretórios. Primeiro, tenta apagar um diretório ocupado, sem sucesso. Em seguida, apaga o diretório vazio.
- **test_seek2.c** Posiciona o *current_pointer* de um arquivo já existente no disco (informado pelo seu caminho absoluto) no byte 16. Em seguida, realiza a leitura.
- **test_truncate2.c** Cria um novo arquivo regular e escreve 64 bytes nele. Realiza a leitura e fecha este mesmo arquivo. Em seguida, abre o mesmo arquivo, posiciona o *current_pointer* no byte 4 e trunca-o. Para comparar o resultado, lê de volta do arquivo e torna a fechá-lo.
- **test_write.c** Cria um novo arquivo com o caminho absoluto informado e escreve nele. Em seguida, lê deste mesmo arquivo. Este teste foi feito para verificar o comportamento do programa com arquivos maiores.

- **test_write2_delete.c** Cria um novo arquivo e apaga-o, em seguida. Este teste foi desenvolvido para testar a remoção de arquivos grandes.
- **test_write2.c** É o teste mais complexo desenvolvido, visa verificar o comportamento do sistema ao escrever e ler arquivos muito grandes (utilizando ponteiros de indireção).
- **teste1.c** Cria um novo diretório pré-definido e, em seguida, um arquivo regular dentro deste diretório.
- **teste2.c** Cria um novo diretório, abre-o e cria um arquivo regular dentro deste mesmo diretório. Em seguida, lê a primeira entrada desde diretório.

3 Dificuldades

O trabalho foi desafiador e exigiu muitas horas de dedicação, gerando mais de 3 mil linhas de código. Enumeramos as principais dificuldades enfrentadas:

- Entender o funcionamento das funções fornecidas para ler e escrever no disco, assim com o uso de buffers de *unsigned char* para armazenar os dados. Apesar de já termos lidado com escritas e leituras em arquivos, foi completamente novo o conceito de ler e escrever num disco, mesmo que simulado;
- Transformação de tipos de variáveis, que deviam ser lidas em um buffer de *unsigned char*, transformadas para inteiros, a fim de serem manipuladas e passadas novamente para *unsigned char* a fim de que fossem gravadas no disco;
- Na especificação do trabalho, a estrutura *t2fs_record* contém a variável *name*, informando que esta possui 31 bytes. Contudo, no arquivo *t2fs.h*, a variável *t2fs_record̀name* possui 32 bytes, assim como na leitura dos arquivos e diretórios no disco fornecido. Essa discrepância de informação causou vários erros e teve uma detecção muito difícil, fazendo com que os valores lidos fossem incorretos, apesar da lógica e do código estar correto;
- Lidar com os ponteiros indiretos: além de ter uma lógica complexa, a implementação de ponteiros indiretos se apresentava difícil de testar, por não possuirmos, no disco, arquivos com o tamanho necessário. Apenas após a impĺementação da função *write2*, conseguimos testar o funcionamento dos ponteiros indiretos.
- Código complexo e muito extenso. A dificuldade e o grau de complexidade do trabalho resultou em um código cheio de detalhes e muito longo. Tentamos ao máximo modularizá-lo para poder reaproveitar as funções já implementadas, contudo, as próprias limitações da linguagem c em relação ao polimorfismo fizeram com que o código final fosse muito extenso.