Task 2 - File System

Henrique Plácido - 209822 e Roberto O. Cordoni - 192670

9 de dezembro de 2013

1 Introdução

Sistema de arquivos é a forma de organização de dados em algum meio de armazenamento de dados em massa frequentemente feito em discos magnéticos. Sabendo interpretar o sistema de arquivos de um determinado disco, o sistema operacional pode decodificar os dados armazenados e lê-los ou gravá-los.

Com base nos conceitos aprendidos na disciplina, o objetivo deste trabalho é a implementação de um Sistema de arquivos capaz de gerenciar a ocupação de um disco físico, usando alocação indexada.

2 Descrição do Programa

2.1 Estrutura de Dados

Foi criada uma estrutura básica para o gerenciamento de arquivos abertos, a *FileHandle*. Ela contém o *filePointer*, que é o apontador para a próxima posição do arquivo a ser escrita/lida e o *handleNumber*, que é justamente o identificador do arquivo aberto. Além dessas duas informações básicas para a manipualção de um arquivo aberto, a estrutura contém também uma outra estrutura, a *FileDescriptor*, que armazena as mesmas informações contidas em um registro na área de root e, além delas, provê a informação em qual dos blocos do root e em qual posição relativa desse bloco se encontra o registro do arquivo. Com essas duas informações, o mecanismo de atualizar o registro no disco se torna simples, visto que é só fazer uma cópia direta do descritor em memória para o disco, já que é conhecida a posição do registro em disco.

Toda vez que um arquivo é aberto, é criada uma instancia da estrutura FileHandle e, então, ela é adicionada a uma lista encadeada de arquivos abertos. No momento em que um arquivo é fechado, a instância do FileHandle é apenas removida da lista, visto que as atualizações necessárias no disco são realizadas no momento das escrita de dados em um arquivo.

2.2 Funcionamento

- *t2fs_identify:* Retorna os nomes e o número dos cartões dos implementadores.
- *t2fs_create:* Para a criação de um arquivo, primeiro a função verifica se já existe um arquivo em disco com o mesmo nome do novo. Se houver, o arquivo em disco será removido (será indicado no bit 7 do primeiro

caractere do nome que é um arquivo e inválido e os blocos alocados para esse arquivo serão setados como desocupados no bitmap) e então, será feita uma pesquisa no bitmap para verificar qual bloco do disco pode ser alocado para esse novo arquivo (cada arquivo, ao ser criado, recebe um bloco de dados). Por fim, é adicionado na área de root o registro do novo arquivo.

- *t2fs_delete:* Para remover um arquivo, o bit 7 do primeiro caractere do nome do arquivo é setado como "0" e todos os blocos de dados alocados para esse arquivo serão indicados no bitmap como livres para uso posterior.

- *t2fs_open:* Essa função faz uma pesquisa na área de root para verificar se existe um registro de um arquivo válido com o nome fornecido. Caso exista, é instanciado uma estrutura FileHandle, que contém o apontador de posição corrente dentro do arquivo (é setado como zero), o campo de identificador do arquivo aberto e é instanciada uma outra estrutura, a FileDescriptor, que contém uma cópia do registro do arquivo para manipulação em memória. A estrutura FileDescriptor é um atributo da estrutura FileHandle, portanto, uma instancia de FileHandle provê todas as informações necessárias para manipular o arquivo aberto. Por fim, a instância de FileHandle é adicionada em uma lista de arquivos abertos. Então, toda vez que é utilizada uma função que necessita informar o identificador de um arquivo, é, primeiramente, verificado na lista de arquivos abertos se existe um arquivo aberto para o identificador fornecido.

- *t2fs_close:* Essa função apenas remove da lista de arquivos abertos a instancia de FileHandle, removendo-a em seguida, da memória.

- *t2fs_read:* Essa função, inicialmente, verifica o valor do apontador de posição corrente para determinar em qual bloco do arquivo a sequencia de bytes desejada se encontra. Após o cálculo, ela carrega o bloco correspondente e preenche o buffer de entrada com os bytes do arquivo retornando o número de bytes lidos. Se o apontador de posição tiver excedido o tamanho do arquivo, nenhuma leitura é realizada, retornando zero, o que indica que foi atingido o fim do arquivo.

- *t2fs_write:* Da mesma forma que a função t2fs_read, essa função inicialmente analisa o valor do apontador de posição corrente para determinar em qual bloco de dados do arquivo a escrita será realizada. Após realizar o cálculo, o bloco é carregado do disco e a escrita é realizada em memória. O bloco será transferido para o disco quando a sequência de bytes a ser escrita acabar, ou quando o bloco tiver sido todo preenchido. Caso o bloco seja totalmente preenchido antes da sequência de bytes acabar, um novo bloco é alocado para continuar a escrita. Da mesma forma que o bloco inicialmente carregado, o novo bloco é escrito em memória e depois gravado no disco. A função retorna o número de bytes efetivamente gravados, ou zero bytes, caso todos os blocos de dados possíveis estiverem cheios.

- *t2fs_seek:* Essa função carrega o FileHandle do arquivo desejado da lista de arquivos abertos e altera o campo filePointer, que é o apontador de posição corrente. Se o valor que se deseja colocar em filePointer for maior que o tamanho do arquivo, um erro é retornado.

- *t2fs_first:* Essa função varre o root em busca do primeiro arquivo válido. Caso encontre algum, ela insere na estrutura t2fs_find as informações necessárias para a função t2fs_next poder carregar o registro desse arquivo encontrado. O método de busca basicamente apenas testa o bit 7 do primeiro caractere do nome enquanto não for encontrado um arquivo válido e enquanto houver blocos de root disponíveis.

- *t2fs_next:* Acessando as informação contidas na estrutura t2fs_find, essa função retorna um registro de um arquivo válido encontrado pela função t2fs_first ou por um uso prévio de t2fs_next. Após carregar o registro do arquivo, essa função varre o root em busca do próximo arquivo válido. Caso encontre algum, carrega em t2fs_find as informações necessárias para que a próxima chamada dessa função retorne o registro do arquivo encontrado.

3 Dificuldades Encontradas

Uma das maiores dificuldades foi a de implementar as funções de gerenciamento da área de bitmap, visto que a linguagem C não fornece funções para o tratamento de bits em uma variável. Além disso, tivemos um pouco de dificuldade para calcular corretamente qual seria o bit que iria representar um determinado bloco da área de dados.

Outra dificuldade considerável encontrada foi a de manipular blocos de dados endereçados por blocos de índices. Como tivemos alguns problemas que não conseguimos resolver, a nossa implementação não permite que em um arquivo, possam ser lidos ou escritos blocos de dados que são acessados através dos dois ponteiros de indireção, ficando restrito a blocos apontados pelos ponteiros diretos.

4 Questionário

1. *Sem alterar a quantidade de ponteiros de alocação indexada do Superbloco, quais outros fatores influenciam no maior tamanho de arquivo T2FS possível? Como esse fatores influenciam nesse tamanho?*

   R.: O tamanho do bloco, pois, com blocos maiores é possível armazenar mais dados por bloco. Além disso, com um bloco maior, é possível armazenar mais ponteiros em blocos de índices, o que permite um arquivo ainda maior.

2. *Se fosse você reprojetar o T2FS para suportar um diretório organizado em árvore, que permite a criação de diretórios dentro de diretórios, que mudanças seriam necessárias na estrutura atual do T2FS? Responda explicando o impacto dessas alterações nas várias estruturas de controle do T2FS (Superbloco, área de bitmap, área de diretório, formato dos blocos de índice, etc.) assim como nas funções da API (tabela seção 3).*

   R.: Para oferecer uma maior flexibilidade através de um sistema de diretório organizado em árvore, cada diretório poderia ser também um arquivo que armazena registros. Para tal, no registro de um arquivo deveria existir um campo que informa a natureza do arquivo (diretório, arquivo comum). Então, como diretórios podem ser criados e excluídos, da mesma

forma que arquivos, não seria necessário alterar a forma como é tradada a área de bitmap, pois, na implementação, as funções de gerenciamento tratam somente dos bits que representam blocos na área de dados. Por fim, as áreas de superbloco e diretório não sofreriam alterações.

3. *Supondo que você desejasse melhorar o T2FS, permitindo a criação de vínculos estritos (hardlinks). Que alterações seriam necessárias no T2FS? Há necessidade da criação de novas funções? Se sim, quais? Se não, porque não.*

   R.: Partindo do pressuposto que foi implementado um sistema de arquivos com diretórios organizados em árvore, conforme o item anterior seria necessário definir um novo tipo de arquivo (seria indicado no mesmo campo de tipo de arquivo proposto anteriormente). Esse arquivo, então, teria o ponteiro para o arquivo desejado. Para permitir essa funcionalidade, seria necessário definir uma função para a criação desse novo tipo de arquivo e outra para acessar as informações nele contidas.

4. *As estruturas de controle do T2FS contêm informações que permitem verificar a consistência de alguns de seus elementos. Isso é possível graças a um certo nível de redundância de informação (por exemplo, no registro de arquivo, nas entradas do diretório, o número total de blocos usados por um arquivo e o tamanho do arquivo – em bytes – permitem uma verificação). Identifique quais outros elementos são redundantes e discuta como seria possível usar essa redundância para aumentar a confiabilidade do T2FS.*

   R.: Há uma redundância em relação à informação do número de blocos do arquivo e os ponteiros para blocos de dados. Por exemplo, se o segundo ponteiro direto for 0, o arquivo não terá mais de um bloco, o que pode ser, também, verificado no campo do número de blocos. Com essa redundância, é possível fazer uma verificação de consistência a fim de descobrir se os campos de tamanho me blocos realmente contém a informação correta.

5. *Como você implementou a atribuição dos identificadores de arquivos (file handler) pelas funções t2fs_create e t2fs_open? Discuta a questão da reutilização dos mesmos.*

   R.: Foi criada uma variável global estática que controla qual foi o último identificador criado. Então, toda vez que um arquivo é aberto, é inserida em uma lista encadeada de arquivos abertos uma instancia da estrutura FileHandle, que contém o valor do identificador e demais informações necessárias para a manipulação do arquivo aberto. Como o identificador é um inteiro, há uma ampla faixa de valores de handles que podem ser criados durante uma sessão. Então, para reutilizar um valor de um identificador, seria necessário encerrar a aplicação e iniciá-la novamente. Visto que um usuário muito dificilmente irá abrir em uma sessão mais de $_2 32$ arquivos, essa decisão de implementação pareceu razoável.

6. *Como você implementou a gerência do contador de posição (current pointer) usado pela função t2fs seek? R.:* Na estrutura HandleFile definida, existe um campo chamado filePointer que aponta para a próxima posição

no arquivo onde um byte será escrito/lido. Então, toda vez que for realizada uma escrita/leitura, a instancia HandleFile correspondente do arquivo aberto é acessada na lista encadeada de arquivos abertos, então o campo filePointer é atualizado conforme os dados são escritos/lidos.

7. *A escrita em um arquivo (realizada pela função t2fs write) requer uma sequência de leituras e escritas de blocos de dados e de blocos de controle. Qual é a sequência usada por essa função? Se essa sequência for interrompida (por falta de energia, por exemplo) entre duas operações de escrita de bloco, qual será o efeito na consistência dos dados no disco? É possível projetar uma sequência de escritas no disco que minimize a eventual perda de dados?*

   R.: Inicialmente, é verificado na estrutura HandleFile o valor do contador de posição (filePointer) afim de descobrir qual o bloco do disco deve ser carregado. Descoberto o bloco, ele é carregado e todas as escritas são realizadas em memória. Então, para o caso em que uma sequência de bytes não excede o tamanho do bloco, no termino da escrita em memória, o bloco modificado é salvo no disco. Caso a sequência de bytes exceda um bloco de dados, o bloco cheio é salvo em disco e é calculado qual deve ser o próximo bloco a ser escrito. Então, o próximo bloco é carregado e a escrita continua nele. Por fim, ao acabar a sequencia de bytes, esse bloco é salvo no disco. Além de armazenar no disco os blocos de dados, o registro do arquivo, na área de root, também é acessado para atualizar o valor do tamanho em bytes/blocos e os ponteiros para os blocos de dados. Se durante o processo de escrita, que é realizado em memória, ocorrer uma falta de energia, poderia ocorrer um problema de consistência caso os blocos de dados fossem salvos no disco, mas não o registro na área de root. Com isso, as informações de tamanho do arquivo e os ponteiros para os blocos de dados estariam inconsistentes com a nova situação do arquivo.

   Obs: Devido a alguns problemas que não resolvemos, blocos de dados que podem ser endereçados somente com ponteiros de indireção estão inacessíveis. Então, na nossa implementação, não são carregados blocos de índices. Logo, não faz parte da sequencia de acessos ao disco carregar e atualizar blocos de índice.

8. *Algumas estruturas gravadas no disco são mais facilmente manipuláveis se estiverem na memória principal (como se fosse uma cachê). Por outro lado, isso aumenta a possibilidade de perda de dados, pois as informações existentes nessa cachê e que não foram escritas no disco, podem ser perdidas, caso ocorra alguma interrupção de operação do sistema. Quais informações do disco você está mantendo (e gerenciando) na memória principal e porque você as escolheu? Qual a política que você usou para decidir quando escrevê-las no disco?*

   R.: Como descrito anteriormente, em uma escrita, é acessada na lista de arquivos abertos a instancia da estrutura HandleFile do arquivo que sofrerá a escrita. Essa estrutura, além do filePointer, contém uma cópia do descritor (registro do arquivo). Então, sempre que é realizada uma escrita em um arquivo, os campos fileSizeInBlocks e fileSizeInBytes e os ponteiros podem sofrer alterações, sendo necessário, então, atualizar o registro no root. O registro no root só é atualizado no momento do fim da escrita, o

que pode ser um problema no caso de falta de energia (vide item anterior). Essa estrutura é mantida em memória, pois, como o arquivo está aberto, as informações do registro são importantes para poder calcular quais blocos carregar, em qual posição do arquivo escrever, se o arquivo já atingiu o tamanho máximo. Então, toda vez que uma dessas informações for necessária, não será necessário fazer um acesso ao disco, o que poderia impactar um desempenho ruim para a aplicação.

9. *Todas as funções implementadas funcionam corretamente? Relate, para cada uma das funções desenvolvidas, como elas foram testadas?*

   R.: Utilizando o disco virtual fornecido, todas as funções implementadas funcionaram.

   Para testar algumas funções, foi utilizado um software HT Editor para analisar os bytes armazenados no disco fornecido.

   - *t2fs_create*: Por ser uma das primeiras funções criadas, para testá-la, foi necessário utilizar, inicialmente, o HT Editor poder visualizar os bytes dentro do arquivo. Então, em um momento inicial, era verificado se a função armazenava corretamente no disco o nome do arquivo e as informações de tamanho e ponteiros. Ao longo do desenvolvimento, com a criação das demais funções, o funcionamento foi comprovado, visto que eram criados de forma correta os arquivos, o que permitia que as outras funções manipulassem esses arquivos.
   - *t2fs_delete:* Para essa função, foi necessário novamente o HT Editor. Para verificar o funcionamento, apenas analisávamos se o bit 7 do primeiro caractere do nome era zerado e se os bits na área de bitmap correspondentes ao blocos liberados eram setados para zero.
   - *t2fs_open:* Por se tratar de uma função que cria uma instancia da estrutura FileHandle na lista de arquivos abertos, era apenas verificado se estavam corretos os dados contidos no nó da lista, como por exemplo, o identificador, contador de posição e os dados do registro do arquivo.
   - *t2fs_close:* Era apenas verificado se o nó na lista de arquivos abertos era removido
   - *t2fs_read:* Inicialmente, quando a função t2fs_write ainda não havia sido implementada, ela foi testada lendo os dados do arquivo que estava armazenado no disco fornecido. Para verificar se a leitura estava correta, foi utilizado ainda o HT Editor para analisar os bytes do disco. Ao longo do desenvolvimento, já com a função t2fs_write implementada, foram feitos testes conjuntos com essas duas funções.
   - *t2fs_write:* Para verificar se os dados eram gravados corretamente e no bloco correto, era feito uma análise através do HT Editor para verificar se eram escritos no bloco correto os dados fornecido como parâmetro de entrada. Além disso, foi utilizada a função t2fs_read para ler o conteúdo escrito e, assim, realizar ao mesmo tempo o teste dessas duas funções.
   - *t2fs_seek:* Essa função foi testada setando um offset arbitrário afim de verificar se, quando usada a função t2fs_read, seria retornada

a sequência correta de bytes a partir da posição setada através da função t2fs_seek.

- *t2fs_first:* Essa função foi testada em conjunto com a função t2fs_next. Chamando, inicialmente, a função t2fs_first, ao utilizar a função t2fs next, deveria ser retornado o registro do primeiro arquivo válido em disco, que no caso era aquele previamente armazenado, o "arq1".
- *t2fs_next:* Para o teste dessa função, foram criados vários arquivos e em seguida alguns deles foram deletados. Então, a função t2fs_next só poderia retornar o registro dos arquivos válidos (não deletados).