Trabalho Prático
Sistemas Operacionais
Universidade Federal São João Del Rei

Paulo Tobias, Rafael Campos, Ygor Magalhães

19 de Dezembro de 2017

1 Introdução

O gerenciamento de arquivo, também conhecido como sistema de arquivo, é o componente responsável pela gerência e manutenção dos arquivos, estabelecendo como os mesmos são organizados e protegidos, além de definir quais operações podem ser realizadas sobre eles. De acordo com [Tanenbaum 2003], através do sistema de arquivos os usuários terão uma interface para armazenamento e recuperação de seus dados de forma a abstrair os detalhes da implementação e organização destes arquivos. Por serem atividades comumente utilizadas pelos usuários, o sistema de arquivos tornou-se a parte mais visível dos SOs

2 Problema Proposto

Este trabalho consiste na implementação de um simulador de um sistema de arquivos simples baseado em tabela de alocação de 16 bits (FAT) e um shell usado para realizar operações sobre este sistema de arquivos. O sistema de arquivos virtual é armazenado em uma partição virtual e suas estruturas de dados mantidas em um único arquivo.
A partição virtual teria um tamanho total determinado por:
• 512 bytes por setor;
• cluster de 1024 bytes (2 setores por cluster);
• 4096 clusters;
O primeiro cluster é definido como boot block, e conterá informações referentes ao volume (partição). Por motivos de simplificação, o boot block terá o tamanho de 1 cluster (1024 bytes) e A FAT terá um tamanho determinado por 4096 clusters de dados * 2 bytes por entrada (16 bits) = 8192 bytes (8 clusters). O diretório root estará localizado logo após a FAT e terá um tamanho de 1 cluster (assim como todos os outros diretórios). Como representado na figura abaixo:

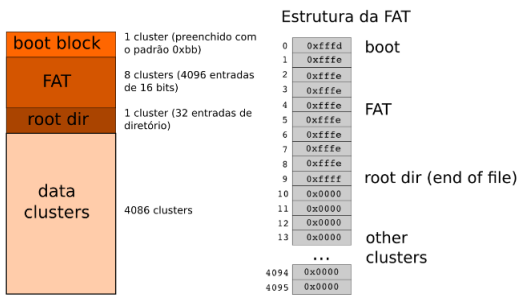

Figura 1: Quantidade de Blocos Livres

O Sistema de arquivo possui limitações para que foram determinadas par simplificar a simulação.
A primeira limitação refere-se ao tamanho da FAT, onde é possível armazenar apenas 4096 entradas para blocos, o que limita o tamanho da partição virtual em 4MB. Se mais entradas fossem necessárias (para um disco maior), seriam necessários blocos adicionais. A segunda limitação refere-se ao número de entradas de diretório em cada nível da árvore. Cada entrada ocupa 32 bytes, o que limita o número de entradas de diretório em 32, tanto no diretório raiz quanto em subdiretórios.

3 Conceitos Trazidos

FAT: é a sigla para File Allocation Table (traduzindo: Tabela deAlocação de Arquivos). A primeira versão do FAT surgiu em 1977, para trabalhar com o sistema operacional MS-DOS, mas foi padrão até o Windows 95. Trata-se de um sistema de arquivos que funciona com base em uma espécie de tabela que indica onde estão os dados de cada arquivo. Esse esquema é necessário porque o espaço destinado ao armazenamento é dividido em blocos, e cada arquivo gravado pode ocupar vários destes, mas não necessariamente de maneira sequencial: os blocos podem estar em várias posições diferentes. Assim, a tabela acaba atuando como um "guia"para localizá-los.
Diretório: Um diretório é uma subdivisão lógica de um sistema de arquivos, que permite o agrupamento de arquivos que se relacionam de alguma forma. Diretórios são frequentemente chamados de pastas em função de uma analogia presente nos sistemas Windows que mais recentemente foi adotada por diversos outros sistemas. A divisão proporcionada por um diretório é lógica, no sentido que não existe necessariamente uma divisão física das informações relativas a um diretório.

4 Implementação

Abaixo temos um pseudo código para ilustrar de uma maneira geral o comportamento do programa. Depois, mostraremos sua estrutura de dados e sua lista de rotinas descrevendo suas funções.

Algorithm 1 Comportamento

```
1:  procedure FUNÇÃO PRINCIPAL
2:      /* Procura pelo arquivo fat.part para usar a função load e em caso de
3:      falha, chama a função init. */
4:      inicializar_sistema()
5:  loop_infinito:
6:      /* Lê um comando da entrada padrão. */
7:      comando ← ler_comando()
8:
9:      /* Divide o comando em até 3 argumentos. */
10:     argumentos[] ← parse_command(comando)
11:     /* Chama a função correta de acordo com o primeiro argumento. */
12:     process_command(argumentos)
13:
14:     goto loop_infinito.
```

4.1 Estrutura de Dados

dir_entry_t: Contém as informações sobre um diretório/arquivo.
data_cluster: Contém os dados de um arquivo/diretório. Para arquivos, o dado é uma string. E para diretórios, são até 32 *dir_entry_t* (incluindo . e ..)

4.2 Lista de Rotinas

4.2.1 Fat

init: Função para excluir todos os dados e inicializar uma nova partição vazia.
load: Função para carregar a FAT e o diretório raiz para a memória.
exit_and_save: Utilizada para finalizar o programa e atualiza o arquivo com as modificações feitas durante a execução do programa.
fat_get_free_cluster: Retorna o primeiro cluster livre na FAT. Usa pesquisa linear para encontrar o cluster.
fat_free_cluster: Libera todos os cluster na FAT apontados pelo parâmetro *first_block*.
read_data_cluster: Função para ler o cluster, apontado pela varável *block*, do arquivo para um *data_cluster* e retornar o endereço deste cluster. Importante ressaltar que o dado será armazenado no vetor global *clusters* e o offset do vetor é dado por *block*. Portanto, o vetor é usado como uma espécie de espelho para

o arquivo.
write_data_cluster: Escreve o cluster apontado pela variável block no disco. O *data_cluster* não precisa ser passado por parâmetro pelo fato do vetor global clusters ser um espelho do arquivo. Portanto apenas o endereço do cluster é necessário.
set_entry: Função de conveniência para facilitar alterar os dados de um *dir_entry_t*.
search_file: Função que recebe um caminho absoluto ou relativo juntamente de um atributo e verifica se o arquivo/diretório existe. Em caso positivo, as informações sobre o arquivo/diretório é retornada. NULL é retornado em caso de falha.
fat_log: Função para medir a fragmentação externa da FAT. ***get entry block***: Mede a fragmentação externa da FAT.

4.2.2 Shell

cd: Muda o diretório corrente.
stat: Mostra informações sobre um arquivo ou diretório.
ls: Lista o conteúdo de um diretório.
mkdir: Cria um novo diretório. Caso os diretórios parentes ao longo do caminho não existam, eles poderão ser criados automaticamente com a flag -r ou -p.
create_file: Cria um novo arquivo comum. Caso os diretórios parentes no caminho não existam, eles poderão ser criados automaticamente com a flag -r ou -p.
unlink_file: Exclui um arquivo ou diretório. Caso seja um diretório, este só será apagado caso esteja vazio (. e .. não são contabilizados).
write_file: Sobrescreve o conteúdo de um arquivo pela string passada. Aloca ou libera clusters, dependendo do novo tamanho da string em relação ao que será sobrescrito.
read_file: Escreve na saída padrão o conteúdo de um arquivo.
append: Anexa a string passada ao final do arquivo. Mais clusters vão sendo alocados caso necessário.
shell_parse_command: Função para receber a entrada do usuário e dividí-la em até 3 argumentos. O primeiro sempre será a função desejada e os eventuais segundo e terceiro parâmetros variam de acordo com a função.
shell_process_command: Separa o comando em até 3 strings usando a função *shell_parse_command* e chama a função correta.

5 Analise de Resultados

Nesta seção, apresentaremos os resultados dos testes realizados em forma de 3 gráficos, nos quais serão analisados, respectivamente, a maior quantidade de blocos consecutivos, a quantidade total de blocos livres e fragmentação externa. Foram realizados 4 tipos de testes diferentes, que estão divididos nas quatro colunas de cada figura. Todos os arquivos criados em todos os testes ocupavam somente um cluster.

Para o primeiro teste, foram criados 117 arquivos consecutivos e, após isso, 48 arquivos foram apagados ao acaso. O segundo teste é semelhante o teste 1, porém mais arquivos foram criados e excluídos. Após isso, 338 arquivos foram apagados aleatoriamente. O terceiro teste teve como objetivo testar o pior caso do programa. Primeiro, o disco foi totalmente preenchido. Após isso, 2043 arquivos foram excluídos aleatoriamente, de modo a minimizar a maior seção de blocos livres consecutivos. O quarto teste foi semelhante ao terceiro. Porém, o comando para excluir um arquivo era gerado após dois comandos *create*. Este teste simula a criação de arquivos temporários, que tem tempo de vida curto e são apagados pouco tempo depois de sua criação.

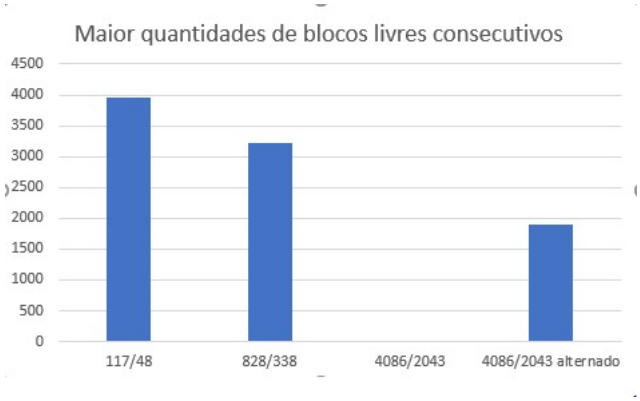

Figura 2: Blocos Livres Consecutivos

Na Figura 2, apresentamos a maior quantidade de blocos livres consecutivos. Desta forma, podemos observar que essa quantidade, para poucos arquivos, é aproximadamente dada por $4086 - quantidade\_arquivos$. Porém, quando a partição virtual está com poucos blocos livres, o modo de como os arquivos foram criados e excluídos influencia no tamanho do maior cluster livre. Se todos os arquivos são gerados para depois serem excluídos, então a quantidade de blocos livres consecutivos é drasticamente reduzida, pois os blocos livres acabam ficando espalhados pela partição. Se os arquivos são gerados e excluídos quase imediatamente após sua criação (teste 4, quarta coluna), então os novos arquivos vão sendo inseridos nos locais que estavam ocupados pelos arquivos excluídos e, com isso, boa parte da partição nem chega a ser utilizada. Sendo assim, a quantidade de blocos livres consecutivos foi a maior possível.

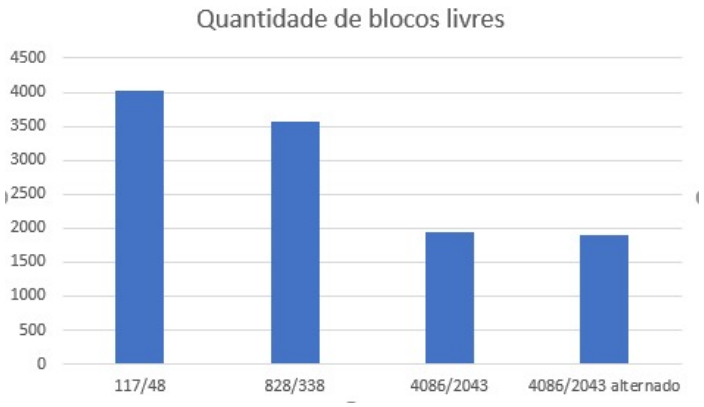

Figura 3: Memória Livre

Na Figura 3, apresentamos a quantidade total blocos livres, que demonstra a quantidade de blocos que não foram utilizados ou que foram usados, mas depois foram liberados. Como esperado, o valor foi consistente com a quantidade de arquivos criados e excluídos.

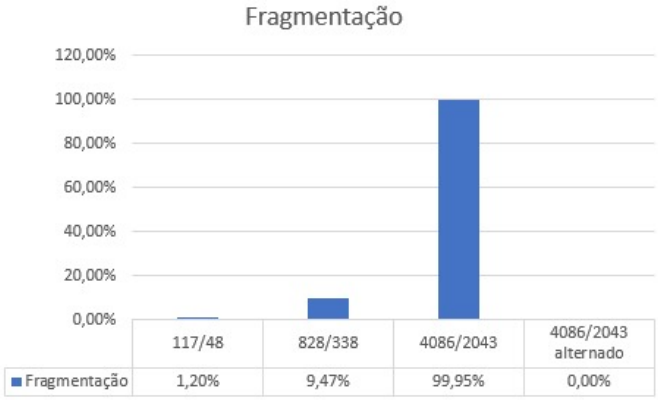

Figura 4: Fragmentação

Na Figura 4, mostramos a fragmentação externa da partição após a execução dos testes. A fragmentação é medida usando o tamanho da maior sequência de blocos livres consecutivos (dados na Figura 2) e a quantidade total da memória (Figura 3). A fórmula é dada pela Figura 5.

$$\text{FRAGMENTATION} = 1 - \frac{\text{LARGEST\_free\_block}}{\text{ALL\_free\_memory}}.$$

Figura 5: Medida da fragmentação externa

Como esperado, o pior caso ocorreu durante o teste 3. O fato da partição ser toda preenchida de antemão acabou gerando muitos blocos livres espalhados. Porém, metade da memória estava disponível, fazendo com que a fragmentação chegasse a $99,5\%$.
Porém, ter muitos arquivos não significa que a partição estará necessariamente fragmentada. No teste 4, onde os arquivos são gerados e excluídos, a fragmentação foi de 0%. Isto se deve ao fato de que novos arquivos são alocados no primeiro cluster disponível. Portanto, quando um arquivo é excluído e outro é criado em seguida, o novo arquivo ocupará o cluster do arquivo que acabou de ser excluído. Portanto, mesmo com 4086 arquivos criados, apenas metade da partição foi usada de fato, deixando metade ocupada com arquivos e a outra metade livre. Portanto, a fragmentação não existe.
Para os testes 1 e 2, que criaram e excluíram poucos arquivos, a fragmentação também foi quase nula, pois grande parte da memória disponível é a que não foi usada pelos testes e as seções fragmentadas são quase despresíveis.

6 Conclusão

A partir dos resultados apresentados aqui, notamos que a partição virtual usando o sistemas de arquivos FAT teve bom comportamento na maioria dos testes. Porém, em casos reais, não é o que acontece. O mais comum é que ocorram casos semelhantes ao teste 3, pois unidades de armazenamento tem justamente o objetivo de salvar os arquivos. Mesmo com otimizações para evitar o reuso de clusters previamente apagados, sistemas de arquivos FAT normalmente ficam muito fragmentados. Além disso, o tamanho da tabela FAT cresce muito com o tamanho da partição e persistí-la na memória principal o tempo todo é inviável (computadores atuais possuem centenas de gigas de memória disponível). Portanto, nos dias atuais, partições usando o sistema de arquivo FAT são usadas apenas quando a quantidade de memória total não é muito grande.

7 Referências

TANENBAUM, Andrew. Sistemas Operacionais Modernos. 2a ed. Pearson - Prentice Hall. 2003.

Implementação de sistema de arquivos,por Eduardo Ferreira dos Santos, disponível em : http://www.eduardosan.com/wp-content/uploads/2016/03/aula08-sistemas-arquivos.pdf

Handling memory fragmentation, Jan Lindblad disponível em: https://www.edn.com/design/systems-design/4333346/Handling-memory-fragmentation

Introdução aos Sistemas Operacionais/Sistemas de arquivos, Wikiversidade, dosponível em: https://pt.wikiversity.org/wiki/Introdu