Trabalho Prático 02 - Sistemas Operacionais
Sistema de arquivos
Universidade Federal de São João Del Rei

João Vítor Gonçalves, Millas Násser, Welton Santos

Dezembro de 2017

1 Introdução

O trabalho apresenta uma simulação de um sistema de arquivo simples que é baseado em uma tabela de entradas. Foram utilizadas estruturas especiais que simulam a memória principal e um arquivo simulando o disco, além de tipos de dados que implementam diretórios e arquivos, cada entrada da tabela possui 16 bits e armazena o endereço dos blocos de arquivos.

O objetivo principal é realizar a simulação e verificar a granularidade do disco ao final de uma série de comandos com o intuito de verificar a fragmentação gerada.

2 FAT-16

O sistema de arquivos FAT-16 consiste em uma tabela que armazena quais são os endereços de um arquivo. Uma posição na FAT armazena qual é o próximo bloco a ser acessado para realizar a leitura de um arquivo.

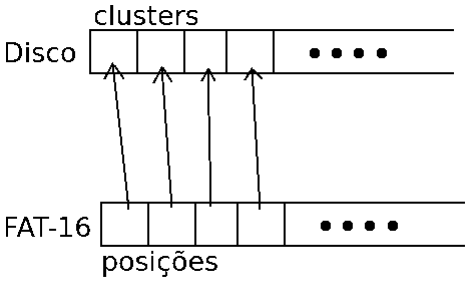

Figura 1: Mapeamento de endereços da FAT-16 para *clusters do disco*

A ideia principal é mapear todas as entradas do disco em uma tabela que o armazena. Para isso, cada entrada do sistema de arquivos mapeia um *cluster* do disco (1024 bytes) e o disco possui no total 4096 entradas, 10 destas são utilizadas para o mapeamento de uma região conhecida do disco para que ele possa dar boot no sistema operacional. Esta região possui 3 áreas especiais:

- *Boot block*
- *FAT-16*
- *Diretório Root*

Na região onde o *boot block* possui a função de indicar o início dos *clusters* conhecidos para o *boot*, utiliza-se 1 *cluster*. Os próximos 9 *clusters* são para a persistir a FAT no disco, e por fim 1 *cluster* para o diretório root.

Visto que existem 4086 *clusters* disponíveis para escrita livre, a Figura 1 representa sucintamente o mapeamento da FAT para as posições livres dentro

do disco onde cada posição da FAT há um *cluster* correspondente, levando à um mapeamento direto.

Há dois tipos de arquivos que podem ser escritos na FAT-16: diretórios e arquivos de dados. O primeiro consome sempre um *cluster* do disco, entretanto, arquivos podem consumir mais que um *cluster*. Para o sistema de arquivos resolver estes arquivos, utiliza-se o mapeamento direto e as posições da FAT como ponteiros para as próximas posições do arquivo, tal como representado na Figura 2.

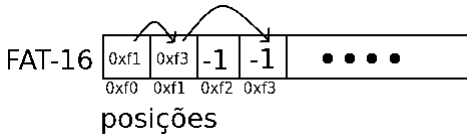

Figura 2: Mapeamento de um arquivo pela FAT-16 com as respectivas posições utilizadas

De acordo com a imagem, um arquivo qualquer inicia no cluster *0xf0*. Ao acessar a posição correspondente na FAT, o conteúdo é o endereço da próxima posição que deve ser acessada para continuar a ler o arquivo, que é o endereço *0xf1*. Esse processo é repetido até encontrar o valor -1, o qual representa o final de um arquivo. Em cada leitura de uma posição da FAT-16, é lido também o respectivo *cluster* no disco com o conteúdo do arquivo. Arquivos que na primeira posição da FAT-16 já possuem -1 indicam que seu tamanho é de apenas 1 *cluster*, tal como em 0xf2.

3 Simulador FAT-16

A proposta deste simulador é providenciar recursos suficientes para analisar o desempenho do sistema de arquivos FAT-16, em relação a pesquisa e alocação de memória secundária. A memória secundária será simulada por completa através de um arquivo de texto fornecendo acesso aleatório.

4 Sistema de Arquivos

O Sistema de arquivos é uma camada de software entre o hardware e o sistema operacional, responsável por criar uma abstração de hierarquia de arquivos para facilitar a manipulação pelo sistema operacional e também pelos usuários. O sistema de arquivos define como uma partição organiza seus arquivos, quais recursos estão disponíveis, limite no tamanho de arquivos e diretórios, atributos de um arquivo, sistema de proteção contra falhas, entre outros parâmetros não relevantes ao contexto. Existem vários sistemas de arquivos para diversas situações e neste trabalho será abordado o sistema FAT-16, presente no sistema MS-DOS e Windows 95.

5 FAT (File Allocation Table)

O FAT consiste em um sistema de arquivos dentre vários outros existentes *(vfat, ext, ntfs, ufs, entre outros...)*, atualmente mais utilizado em mídias de menor capacidade como pendrives e memória para dispositivos com sistemas embarcados. Isso devido a algumas limitações como a baixa segurança contra perda de dados, limitação no tamanho de arquivos (limite de 2GB para FAT-16 com bloco de 32KB) e principalmente o alto consumo de memória necessário para gerenciar mídias com capacidade de armazenamento elevadas (disco rígidos atuais). A gerência de arquivos é feita através de uma tabela situada em uma região separada na partição contendo referência para todos os blocos de dados.

5.1 Fragmentação (Alocação Contígua x FAT)

Considerado fundamental, o acesso á aleatório a memória é um recurso que permite a um usuário alterar o estado de um arquivo persistido. Diferente da alocação contígua, um arquivo pode ter seu conteúdo expandido, removido ou até mesmo excluído da mídia de armazenamento sem que haja um grande desperdício de espaço, ocorrido no método de alocação contígua onde o surgimento de lacunas (trechos de memória não utilizadas) é comum. A existência de lacunas dificulta a reutilização da memória logo que é provinda geralmente pela exclusão de arquivos com tamanhos distintos, raramente encontra-se outro arquivo com o tamanho exato ao do removido para ocupar o espaço liberado. Este fator causa o problema da **segmentação externa**. Apesar da melhor gestão de memória apresentada, a leitura em sistemas de alocação contígua é mais eficiente devido a disposição linear dos dados na mídia. Esta forma de alocação evita gastos excessivos como o movimento do braço de um disco rígido para diferentes regiões em curtos intervalos de tempo, necessário para lidar com a **segmentação interna** do arquivo. Portanto, pode-se dizer que a grande vantagem da FAT em economia de memória é descontada na velocidade de pesquisa em relação à alocação contígua.

5.2 Acesso Aleatório

Outra vantagem é a facilidade na pesquisa. Para acessar regiões aleatórias de um arquivo o mesmo não precisa ser enviado por completo para memória, permitindo que trechos menores do arquivo sejam carregados conforme a necessidade, dispensando o gasto computacional para encontrar trechos específicos, o que é comum em sistemas como o presente em CD's-ROM (Read Only Memory), onde sempre é necessário uma pesquisa linear pelo CD para encontrar o dado desejado. Para dispositivos com memória reduzida, esta qualidade é um diferencial crucial.

6 Implementação

O simulador trabalha semelhante a um shell de comandos existentes em sistemas UNIX e derivados, onde através de comandos e parâmetros é possível: ler, criar, apagar, alterar arquivos e diretórios. Todos os arquivos são mapeados na FAT e nas entradas de diretório sendo persistidos em disco (arquivo de texto), podendo ser carregados ao iniciar o programa através do comando load (ativado por padrão), simulando desta forma o boot do sistema. Conforme dito anteriormente, é possível criar arquivos, encerrar o programa e recarregá-los, logo, também é possível que o limite de blocos disponíveis se esgote ao adicionar vários arquivos (principalmente arquivos com elevados volumes de dados), o que ocasiona no total consumo do disco, sendo necessário o uso do comando unlink ou init para liberar espaço no mesmo.

6.1 Simulação do disco

A simulação do disco é realizada em arquivo de texto, onde são salvos todos os dados (similar à política presente no sistema MS-DOS da Microsoft). O arquivo possui 4MB de memória abstraidos em 8192 clusters de 512 bytes ou 4096 blocos de 1KB, com 10 blocos reservados que são distribuidos da seguinte forma: 1 bloco para armazenar as configurações de boot, 8 para armazenar a FAT e o último para o diretório raiz.

6.2 Arquivos e Diretórios

Por questões de praticidade, foram adotadas as seguintes restrições:

1. Arquivos podem conter tamanho dinâmico e ocupar mais de 1 bloco (1024 bytes).
2. Diretórios podem ocupar apenas 1 bloco.
3. Diretórios podem conter até 32 ponteiros (entradas de 32 bytes) para outros arquivos e subdiretórios.
4. Arquivos são representados com byte de atributo 0 e diretórios 1.

6.3 Shell

O shell de comandos é um simples processador de texto que mantém o programa em loop esperando por uma entrada aravés de uma leitura do buffer padrão. Ao ser inserido o comando, este é enviado para uma função de processamento de string. Depois, é identificada a função correspondente do sistema de arquivos, e então o controle do programa é transferido para mesma. Abaixo segue o pseudocódigo referente a modo de funcionamento do shell.

```
Shell:
    var comando
    enquanto comando != exit:
```

leitura_do_comando(comando);
identifica_função(comando);
chama_função(comando)

7 Análise de Resultados

O desempenho de um sistema de arquivos pode ser avaliado sobre certos critérios como segurança, velocidade de acesso, organização, recursos oferecidos, entre outros. Neste trabalho, foi escolhido a velocidade de acesso como referencial. Portanto, para medir o desempenho do FAT, observou-se o comportamento de 2 parâmetros: **fragmentação por arquivo** e **média de arquivos fragmentados**. A fragmentação por arquivo é obtida a partir da seguinte expressão: $\frac{AD}{MB}$, onde AD é o numero de acessos não adjacentes e MB o número de arquivos com tamanho em bloco maior que 1 e a média de fragmentação por arquivos, $\frac{MB}{MB+SB}$, onde SB é o número de arquivos com tamanho igual ou inferior a um único bloco.

7.1 Descrição do ambiente de teste

Para obter dados para análise foi utilizado a função teste_generator, que cria chamadas recursivas e constroi uma árvore de diretórios com arquivos e durante o processo de construção aplica operações de remoção e append. Portanto, os testes foram realizados variando a quantidade de diretórios, arquivos e tamanho dos arquivos com sucessivas aplicações de remoção em arvores de diretórios não densas, intermediárias e densas ocupando todo o disco. Abaixo segue a tabela com so resultados obtidos e as respectivas configurações dos testes.

Ramif | profun	Quantidade arquivos	Segmentação (%)	Segmentação por arquivo (%)
3 3	1	0,00	0,00
3 6	13	0,00	0,00
3 9	73	0,01	0,07
6 3	27	0,07	0,25
6 6	86	0,13	0,39
6 9	134	0,07	0,22
9 3	267	0,17	0,56
9 6	1382	0,16	0,48
9 9	2802	0,15	0,45
média	531,67	0,08	0,27

Figura 3: Tabela de Resultados

Como visto na tabela acima, pode-se observar que: A segmentação da FAT aumentou proporcionalmente com o aumento do número de arquivos, porém a quantidade de operações (unlink e append) influenciou com maior significância na fragmentação mostrando que não só a quantidade de arquivos é um fator de influência, uma vez que para uma árvore 9 x 3 a média de segmentações atingiu

valores superiores para as árvores 9 x 6 e 9 x 9. Para uma melhor visualização, segue abaixo um gráfico com a seguinte relação.

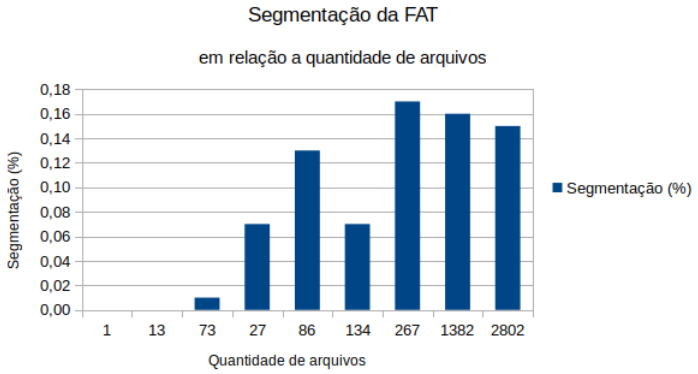

Figura 4: Caption

Outra relação importante é a taxa de segmentação interna do arquivo, que segue a mesma relação que a segmentação geral da FAT, mas diferentemente o maior valor atingido foi uma fragmentação de 56%, que também é resultados de várias operações aplicadas no sistema de arquivos. Abaixo segue o gráfico com que representa a média de fragmentação interna dos arquivos maiores que 1024 bytes (multi clusters).

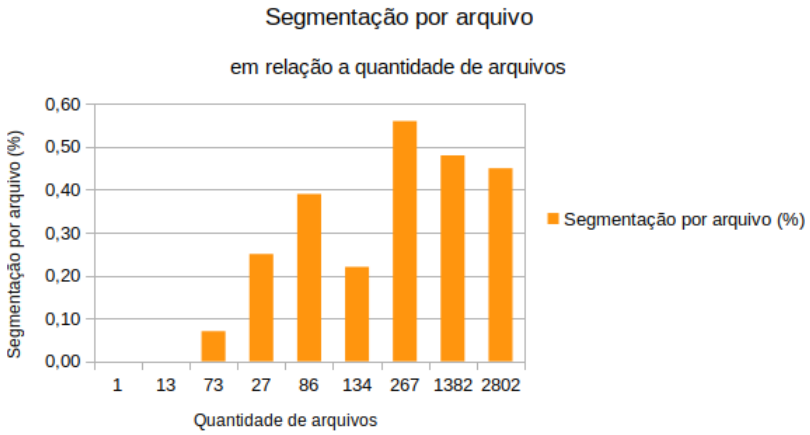

Figura 5: Média de segmentação interna do arquivo

8 Conclusão

Assim como o esperado observou que a fragmentação da FAT não é totalmente interligada a quantidade de arquivos, mas sim também a quantidade de alterações realizadas depois da persistência de dados no disco. Desta forma conclui-se que para partições em que o sistema FAT esta presente e a operação de leitura predomina é esperado que ocorra uma menor fragmentação no sistema de arquivos.

9 Apêndice

9.1 fat.h

9.1.1 Funções de Leitura/Escrita

findCluster: Recebe um caminho e o bloco atual deste caminho. Retorna o bloco do próximo diretório ou arquivo e também o restante da string. Exemplo: /home/user home = Bloco 5.

free_entry: Verifica se existe espaço para inserir mais uma entrada no diretório.

9.1.2 Funções de Manipulação de Bloco

free_blocks: Procura blocos livres em uma ordem de complexidade de O(n).

9.1.3 Funções de Suporte

new_dir: Cria um diretório padrão.

9.1.4 Funções do Sistema de Arquivo

init: Formata o disco (fat.part).

load: Carrega o diretório ROOT e a FAT para memória.

ls: Lista as entradas de diretório de um diretório.
Exemplo: ls /home
-o: mostra diretórios ocultos.
Exemplo: ls -o /home

mkdir: Recebe o caminho e o nome do diretório a ser criado. Se nenhum caminho for passado antes do diretório, o diretório base é tomado como referência.

create_file: Recebe o caminho e o nome do arquivo a ser criado. Se nenhum caminho for passado antes do diretório, o diretório base é tomado como referência.

__**write**: Recebe o caminho e o nome do arquivo a ser criado. Se nenhum caminho for passado antes do diretório, o diretório base é tomado como referência. Se o arquivo nao existir ele é criado. Se o arquivo existir ele é sobrescrito.

append: Recebe o caminho e o nome do arquivo a ser criado. Se nenhum caminho for passado antes do diretório, o diretório base é tomado como referência. Se o arquivo nao existir ele é criado. Se o arquivo existir o texto é adicionado ao seu final.

__**read**: Recebe o caminho e o nome do arquivo e armazena o texto em um buffer, printando ao final.

__**unlink**: Recebe o caminho de um arquivo ou diretório, e remove-o se possível, marcando os blocos do arquivo com livre. Diretório são removidos somente se estiverem vazios.

9.1.5 Constantes

CLUSTER_DATA: Marca o cluster inicial do espaço de dados.

FAT_ENTRY: Mapeia posição de blocos em bytes da fat no disco através do id de um bloco.

ROOT_ENTRY: Mapeia entrada de diretórios da raiz no disco.

9.2 shell.h

format: Formata o texto bruto capturado pelo scanf e converte para uma matriz de comandos.

9.3 teste.h

builder_tree: Cria uma arvore de diretórios com arquivos e também remove arquivos durante o processo de recursão. Possui o objetivo de gerar a segmentação de blocos na FAT de forma automática.

teste_generator: Chama a função builder_tree e inicia as recursões a partir do diretório raiz.

fragmentacao: Calcula os nível de fragmentação da fat, baseado no número de arquivos maiores que 1024 e no número de acessos não adjacentes.

Bibliografia

- TNEMBAUM, Andrew S. *Sistemas Operacionais Modernos*. 3 ed. São Paulo: Pearson, 2010. Acesso em Novembro de 2017.