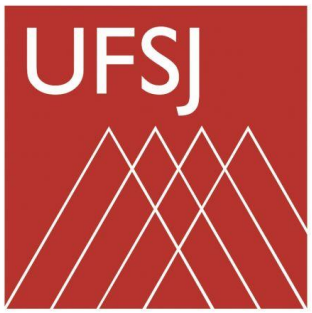

Universidade Federal de São João del-Rei

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI

Terceiro Trabalho Prático da Disciplina de Sistemas Operacionais

Documentação referente ao terceiro trabalho prático da disciplina Sistemas Operacionais 2019/2, desenvolvido pelos alunos Júlio Hebert Silva e Tomaz Miranda de Oliveira do curso de Ciência da Computação da UFSJ.

São João Del Rei
Dezembro/2019

1 Introdução

O sistema de arquivos é o principal componente responsável pela manutenção e gerência dos arquivos, intitulando regras de como serão protegidos e organizados, além de definir operações sobre eles. Segundo [Tanenbaum, 2003], os usuários através do sistema de arquivos poderão ter uma interface de armazenamento e recuperação de seus dados, sendo transparente a eles os detalhes de implementação. E, por ser algo muito demandado pelos usuários, o sistema de arquivos tornou-se parte fundamental dos sistemas operacionais.

2 Problema proposto

A entrada do programa acontece por meio de um arquivo que é passado por parâmetro ao executar o programa, assim como outras informações importantes tais como, qual algoritmo utilizar, tamanho da página e tamanho da memória. Caso arquivo passado exista, será lida linha por linha até o fim, e cada linha deve conter um endereço físico em hexadecimal e um comando especifico, como ‘R’ e ‘W’. O algoritmo, caso a linha lida não possua comando ele irá utilizar o mesmo do último comando passado

O trabalho consiste na implementação de um simulador de um simples sistemas de arquivos baseado em FAT16 e um shell utilizado para realizar as operações em cima do sistema de arquivos. O sistema é armazenado em uma partição virtual e são mantidas em um único arquivo suas estruturas de dados. A partição teria um tamanho total definido por:

- 512 bytes por setor;
- 1024 bytes por cluster(dois setores por cluster);
- 4096 clusters.

O primeiro cluster é definido como *boot block* e as informações sobre o volume ficam nele. Para simplificar a implementação, o *boot block* será do tamanho de um cluster e o tamanho *FAT* será de 4096 clusters de dados * 2bytes por entrada = 8192 bytes. O sistema de arquivos será definido como na Figura 1.

O sistema de arquivos possui limitações para que foram determinadas para 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 por diretório em cada nível da arvore. Cada entrada ocupa 32 bytes, o que limita o número de entradas de diretório em 32, tanto no diretório raiz quanto em subdiretórios.

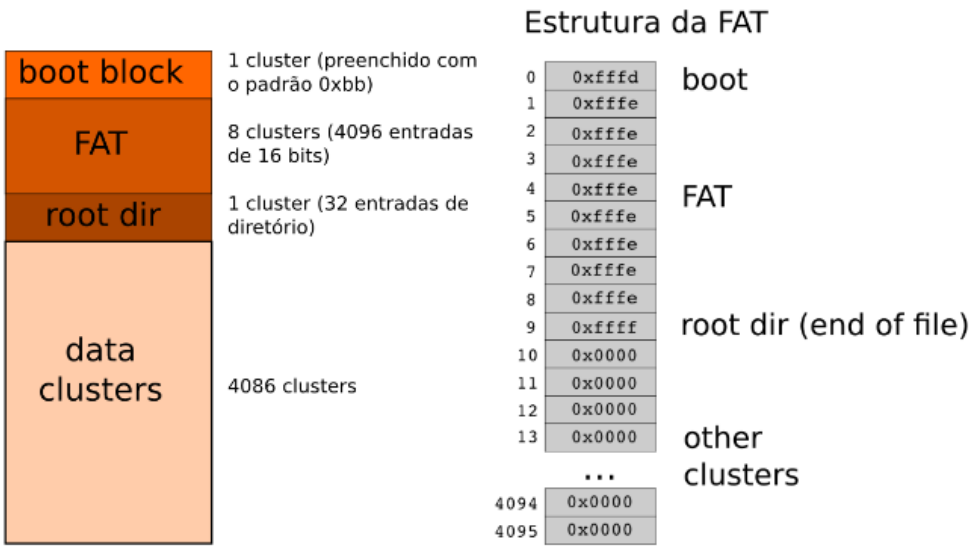

Figura 1 Quantidade de blocos livres

3 Implementação

Implementado na linguagem C, uma das grandes dificuldades encontradas foi a manipulação de Strings, para isso utilizamos funções da biblioteca *string.h,* tais com **strcmp(), strcpy(), strcat().** Além disso, tivemos que implementar uma função extras, **lixo()**, pois a cada loop da interação principal, estava sendo mantido na *string* lixo da execução anterior, gerando conflitos ao fazermos comparações de *strings.*

Ao iniciar, o algoritmo verifica se o arquivo f*at.part* existe, caso positivo é chamado a função *load(),* caso negativo chama-se *init().* Feito isso, o programa entra em estado de *loop* infinito, no qual ele recebe comando passados pelo usuário pelo terminal, numa espécie de *Shell* e chama a respectiva função para aquele parâmetro.

3.1 Estrutura de Dados

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

3.2 Lista de Rotinas

3.2.1 Fat

init: Função para iniciar uma nova partição *fat* vazia, caso exista alguma será pagada.
load: Função para carregar a *fat* e o diretório raiz para a memória.

fechar_salvar: Utilizada ao encerrar a execução do programa para atualizar o arquivo *fat.part* com os arquivos/diretórios que foram atualizados durante a execução.
integridade: Ao fim de cada chamada de comando que altera algum arquivo/diretório, essa função atualiza o respectivo cluster no arquivo *fat.part*.
existe_fat: Ao iniciar o algoritmo, essa função verifica se o arquivo *fat.part:* existe, caso exista será carregada na memória, caso não, será criado uma nova.
cluster_da_memoria: Retorna o cluster que foi solicitado do arquivo *fat.part* para a memória.

3.2.2 Shell

pos_vazia_fat: Retorna a primeira posição que estiver vazia na tabela.
pos_vaiza_pai: Procura qual das 32 referências do diretório passado está vazio.
arruma_destinos: Caso o caminho passado tenha subpastas, essa função separa o primeiro caminho do restante.
excluir: Exclui um arquivo ou diretório. Caso seja um diretório, este só será apagado caso esteja vazio.(. e .. não são contabilizados).
read: Escreve no terminal o conteúdo do arquivo.
append: Anexa a *string* passada no final do arquivo.
write: Sobrescrever a string no arquivo.
mkdir_create: Cria um novo diretório ou arquivo comum.
ls: Lista o conteúdo de um diretório.

4 Análise de resultados

Conseguimos implementar grande maioria das funcionalidades para o sistema de arquivos funcionar normalmente. Apesar de que, caso o arquivo ultrapasse os seus cem caracteres previstos, esses restantes serão perdidos. Todavia, não é permitido criar diretórios/arquivos com o mesmo nome dentro do mesmo “pai”, além se não ser possível escrever nem ler um diretório.

Ainda é possível criar pastar no diretório raiz e fazer a listagem (ls) desde, com o comando “ls /” e, o sistema ainda reconhece uma passagem de comando sem a primeira barra no caminho.

Pudemos notar, que quanto mais a *fat* cresce, mais demorado o sistema ficava para operar e que, ficar criando e apagando diretórios/arquivos ao acaso, gera uma fragmentação indesejada nos *clusters*.

5 Considerações Finais

Neste trabalho, foi possível perceber que, por mais simples que uma tabela *fat* possa aparentar, temos que ela envolve muitos detalhes que se tornaram-se dificuldades no desenvolvimento deste trabalho. Podemos destacar, manipulação de *string* que representa o caminho a ser criado, listado ou removido, que demandou bastante tempo para ser implementado. Mesmo assim isso representou uma oportunidade de aprendizado.

Além disso, o tamanho da tabela *fat* cresce muito com o tamanho da partição, e persisti-la na memória principal o tempo todo é inviável (computados atuais possuem centenas de gigas de memória disponível). Por tanto, nos dias atuais, partições usando o sistema de arquivos *fat* são usadas em sua maioria, quando a quantidade de memória total não é muito grande.

6 Referências

- TANENBAUM, Andrew S. Sistemas Operacionais Modernos. 3 ed. Pearson, 2015.