**Universidade Federal de São João Del Rei**

**João Vitor Alves Morais**
**Luan Liduário Silva**
**Marcella Netto de Oliveira**

# Sistemas Operacionais

**Trabalho (2)**

**São João Del Rei – Minas Gerais**
**14 de Abril de 2021**

**Universidade Federal de São João Del Rei**

**João Vitor Alves Morais**
**Luan Liduário Silva**
**Marcella Netto de Oliveira**

Trabalho apresentado ao curso de Ciências da Computação – Universidade Federal de São João Del Rei – UFSJ, como parte dos requisitos necessários da disciplina de Sistemas Operacionais ministrada pelo professor Rafael Sachetto.

**São João Del Rei – Minas Gerais**
**14 de Abril de 2021**

# 1. Introdução

O mundo está em constante evolução assim como as diversas funções, programas e atividades realizadas pelo Sistema Operacional nos computadores. Hoje, os HDs de baixo armazenamento não são tão úteis quanto antigamente, passou a ser comum, ao comprar ou expandir um simples computador, exigir discos de 1 TB ou mais. Dessa forma, esse crescimento da capacidade de armazenamento dos discos rígidos contribuiu para a variedade de Sistemas de Arquivos.

Um sistema de arquivos é um conjunto de estruturas lógicas, ou seja, feitas diretamente via software, o que permite ao sistema operacional ter acesso e controlar os dados gravados no disco.

Cada sistema operacional lida com um sistema de arquivos diferente e cada sistema de arquivos possui as suas peculiaridades, como limitações, qualidade, velocidade, gerenciamento de espaço, entre outras características. É o sistema de arquivos que define como os bytes que compõem um arquivo serão armazenados no disco e de que forma o sistema operacional terá acesso aos dados.

Neste trabalho, através de um programa, iremos apresentar uma simulação de um sistema de arquivos simples baseado em tabela de alocação de 16 bits (FAT) e um Shell utilizado para realizar operações sobre este sistema de arquivos.

# 2. Problema Proposto

Com o objetivo de colocar em prática alguns dos conceitos vistos em aula, o trabalho proposto consiste em criar um código que simula um sistema de arquivos simples baseado em tabela de alocação de 16 bits (FAT) e um Shell utilizado para realizar operações sobre este sistema de arquivos. Além disso, o sistema de arquivos virtual deverá ser adicionado em uma partição virtual e suas estruturas de dados mantidas em um único arquivo nomeado fat.part.

# 3. Implementação

O algoritmo utilizado para a realização do trabalho proposto foi desenvolvido em linguagem C.

A implementação se baseia no modelo de partição virtual:
- 512 bytes por setor.
- Cluster de 1024 bytes (2 setores por Cluster).
- 4096 clusters.

O sistema de arquivos é preenchido dos seguintes setores:

- Boot Block (1 cluster = 1024 bytes preenchido com 0xBB)
- FAT (4096 entradas de 16 bits)
- Diretório root (1 cluster = 1024 bytes e 32 entradas de diretório)
- Demais entradas (4086 clusters)

A execução do código baseia-se no no arquivo *fat.part* que armazena os dados representando o disco.

### 3.1 Bibliotecas e Arquivos de Inclusão

Para a implementação e pleno funcionamento do código foram utilizadas as seguintes bibliotecas e arquivos de inclusão:

#include <stdio.h>
I/O entrada e saída

#include <stdlib.h>
Ordenação e pesquisa

#include <stdint.h>
Definição de tipos de dados inteiros

#include <string.h>
Strings

### 3.2 Shell

O Shell, por definição, é uma interface de usuário para acessar os serviços de um sistema operacional. Em função do trabalho, o shell irá receber os comandos e controlar a execução das funções através de 13 diferentes comandos, são eles:

- **init:** Inicializa o sistema de arquivos em branco, o que se assemelha a uma formatação de um disco. Dessa forma, ele escreve no arquivo de texto e, em seguida carrega a tabela FAT e o diretório raiz para a memória.
- **load:** Carrega um sistema de arquivos do disco, ou seja, transporta a tabela FAT e o diretório raiz para a memória. É importante ressaltar que a tabela se mantém desatualizada no disco caso a operação descrita não seja realizada.
- **ls:** Lista o diretório. Para listar o diretório raiz é necessário utilizar o argumento [ls /]. Além disso, todos os diretórios devem ser iniciados com [/caminho/diretorio].
- **mkdir:** Cria um novo diretório. [/caminho/diretorio].
- **create**: Cria um novo arquivo de texto dentro do diretório. [/caminho/arquivo].
- **unlink:** Deleta um arquivo ou diretório, apagando sua referência na tabela FAT, mas deixando os dados intactos no cluster pois quando se perde a referência para o cluster, é o mesmo que caso ele não existisse. [/caminho/arquivo] ou [/caminho/diretorio].
- **write:** Escreve (sobrescreve) uma String em um arquivo de texto, cada cluster de dados pode guardar até 1024 caracteres, os demais caracteres são armazenados em outros clusters. [string /caminho/arquivo].
- **append:** Concatena uma String em um arquivo de texto, cada cluster de dados pode guardar até 1024 caracteres, os demais caracteres são armazenados em outros clusters. [string /caminho/arquivo].
- **read:** Lê um arquivo de texto e imprime seu conteúdo na tela. [/caminho/arquivo].
- **exit:** Encerra o programa e libera a memória.

## 4. Estruturas de dados

```
typedef struct
{
        uint8_t filename[18];
        uint8_t attributes;
        uint8_t reserved[7];
        uint16_t first_block;
        uint32_t size;
} dir_entry_t;
```

Representa a entrada de diretório, 32 bytes cada.

```
uint16_t fat[4096];
```

8 clusters da tabela FAT, 4096 entradas de 16 bits = 8192 bytes

```
typedef union
{
        dir_entry_t dir[CLUSTER_SIZE / sizeof(dir_entry_t)];  //1024 bytes / 32 bytes = 32 posições
        uint8_t data[CLUSTER_SIZE];
} data_cluster;
```

Diretórios (incluindo ROOT), 32 entradas de diretório com 32 bytes cada = 1024 bytes ou bloco de dados de 1024 bytes.

### 4.1 Variáveis Globais e constantes

**#define CLUSTER_SIZE 1024** – Define o número de clusters

**#define STRINGS_SIZE 50** – Define o tamanho limite das Strings

## 5. Funções

***int main()***

A função main simula o shell e gerencia suas entradas. Nessa função, são chamadas as funções específicas de acordo com o comando digitado pelo usuário.

***int init()***

A função init cria ou sobrescreve o arquivo "fat.part" para simulação do disco, conforme especificado no trabalho. Além disso, preenche a tabela FAT com os valores iniciais.

***int load()***

A função load abre o arquivo, já criado, "fat.part" para leitura e preenche a tabela FAT com os dados existes no arquivo.

***data_cluster lerCluster(int index)***

A função lerCluster recebe um inteiro que corresponde ao índice do cluster que será lido. A função abre o arquivo "fat.part" para leitura, procura na FAT o cluster correspondente ao índice recebido e retorna esse cluster.

***int procurarDIr(char *diretorio, char *dirAtual, int procura)***

A função tem como objetivo encontrar um diretório correspondente á string recebida, e retornar o índice no cluster do mesmo.

É importante ressaltar que ela separa o caminho recebido na variável *diretorio* e, através dessa separação ela busca nas entradas do cluster o diretório atual que foi separado. Após encontrar, a mesma realiza a verificação para determinar se é um diretório ou um arquivo, por fim ela atualiza o index e realiza a leitura do cluster com o novo index.

Além disso, a função realiza três tipos de procuras distintos:
1 – Procura pelo diretório pai do último diretório ou arquivo do caminho.
2 – Procura até o último diretório.
3 – Procura um arquivo na última posição do caminho e retorna o index do cluster referente ao diretório onde o mesmo está.

***void ls(char *diretorio)***

A função ls recebe uma string como parâmetro. Essa string define o caminho do diretório que deverá ser listado. A função procurarDIr é chamada para encontrar o index do cluster do diretório que será listado. Após isso, a função irá realizar a leitura do cluster do diretório que possui o índice encontrado.

O passo seguinte é percorrer as 32 posições de entrada desse cluster e caso estiver preenchido e for um diretório é impresso o nome do mesmo. De forma análoga o mesmo procedimento é realizado verificando se é um arquivo.

***void mkdir(char *diretorio)***

A função mkdir recebe uma string como parâmetro. Essa string define o caminho do diretório e o nome do novo diretório. A função procurarDIr é chamada para encontrar o index do cluster do diretório pai. Após isso, a função irá realizar a leitura do cluster do diretório que possui o índice encontrado. Em seguida, é verificada a existência de espaço vazio e um diretório com o mesmo nome. Caso as condições sejam verificadas é criado um novo diretório neste cluster. Por fim, a tabela FAT é preenchida e atualizada no arquivo.

***void create(char *dir)***

A função create recebe uma string como parâmetro. Essa string define o caminho do diretório e o nome do novo arquivo. A função procurarDIr é chamada para encontrar o index do cluster do diretório onde o arquivo será salvo. Após isso, a função irá realizar a leitura do cluster do diretório que possui o índice encontrado. Em seguida, é verificada a existência de espaço vazio e um arquivo com o mesmo nome. Caso as condições sejam verificadas é criado um novo arquivo neste cluster. Por fim, a tabela FAT é preenchida e atualizada no arquivo.

***void unlink(char *diretorio)***

A função unlink recebe uma string como parâmetro. Essa string define o caminho do diretório e o nome do diretório ou arquivo que será excluído. A função procurarDIr é chamada para encontrar o index do cluster do diretório onde o arquivo se encontra ou o diretório pai. Após isso, a função irá realizar a leitura do cluster do diretório que possui o índice encontrado. Em seguida, é verificada a existência do arquivo ou do diretório que será excluído.

Caso seja um diretório, é verificado se o mesmo está vazio, caso esteja, um diretório vazio é salvo em seu lugar. Caso seja um arquivo, as posições na tabela FAT que referem aos clusters que guardam os dados do mesmo são limpas.

***void write(char *paramentros)***

A função write recebe uma string como parâmetro. Essa string define a string que será salva e o caminho do arquivo. Em seguida, a função *separaString* é utilizada para separar a string e o caminho. O próximo passo consiste em chamar a função procurarDIr para encontrar o index do cluster do diretório do arquivo. Após isso, a função irá chamar *quebraStringClusters* que retorna os clusters necessários para salvar a string, os mesmos são salvos quando se encontra o espaço vazio na FAT. Por fim, a tabela FAT é atualizada.

***void append(char *parametros)***

A função append recebe uma string como parâmetro. Essa string define a string que será concatenada e o caminho do arquivo. Em seguida, a função *separaString* é utilizada para separar a string e o caminho. O próximo passo consiste em chamar a função procurarDIr para encontrar o index do cluster do diretório do arquivo. Após isso, o cluster com os dados é lido e é verificado se a nova string cabe no mesmo, caso contrário, é salvo o suficiente para preencher o tamanho do cluster, o restante será separado nos clusters necessários utilizando a função *quebraStringCluster*, os mesmos são salvos quando se encontra o espaço vazio na FAT. Por fim, a tabela FAT é atualizada.

***void read(char *diretorio)***

A função read recebe uma string como parâmetro. Essa string define o caminho do arquivo que será lido. A função procurarDIr é chamada para encontrar o index do cluster do arquivo. Após isso, a função irá realizar a leitura do cluster do arquivo que possui o índice encontrado. Em seguida, a string do cluster é impressa e é verificada a existência de outros clusters com dados. Caso a condição seja verificada os arquivos também são impressos.

***void separaString(char *string1, char *string2, char *string3, char *separador)***

A função separaString recebe 3 strings: a primeira é a string que será dividida, a segunda é a string que receberá a primeira parte da divisão e a terceira é a string que receberá o restante da divisão. Além disso, a função também recebe um caractere que determinará como a primeira string será dividida. Basicamente, a função divide a string1 em duas partes: uma antes da primeira ocorrência do caractere separado e outra parte depois dessa ocorrência.

***data_cluster *quebrarStringClusters(char *string, int *numClusters)***

A função quebrarStringClusters recebe uma string, calcula quantos clusters serão necessários para armazená-la, cria um array de clusters com o tamanho necessário, preenche esse array com a string e retorna o array.

***int getNumDiretorios(char *caminho)***

A função getNumDiretorios recebe uma string que define o caminho de um diretório e a percorre contanto quantos diretórios fazem parte do caminho.

***void salvarCluster(int index, data_cluster cluster)***

A função salvarCluster recebe um número de índice e um cluster. A função abre o arquivo “fat.part” para leitura, aponta para o endereço do cluster correspondente ao índice e salva o cluster recebido nesse endereço.

***void atualizarFat();***

A função atualizarFat abre o arquivo fat.part para realizar a leitura e escrita e então o ponteiro é movido para a FAT do arquivo e, é escrito a tabela FAT que está no programa.

## 5. Execução

Para compilar o algoritmo é necessário abrir a pasta dos códigos com o terminal e digitar o comando “make”.
Em seguida digite:

Caso queira utilizar o teclado como entrada de comandos: ./main

## 6. Testes e resultados

```
luan@DESKTOP-EPRNT4S:/mnt/d/pc/UFSJ/EER2/SO/s02/tp2/Fat-16$ ./main
$init
$ls /
DIRETORIOS:

ARQUIVOS:

$mkdir /teste
$mkdir /teste2
$create /arq.txt
$ls /
DIRETORIOS:
teste teste2
ARQUIVOS:
arq.txt
$mkdir /teste/subteste
$mkdir /teste/subteste2
$mkdir /teste2/sub
$ls /teste
DIRETORIOS:
subteste subteste2
ARQUIVOS:

$ls /teste2
DIRETORIOS:
sub
ARQUIVOS:

$exit
```

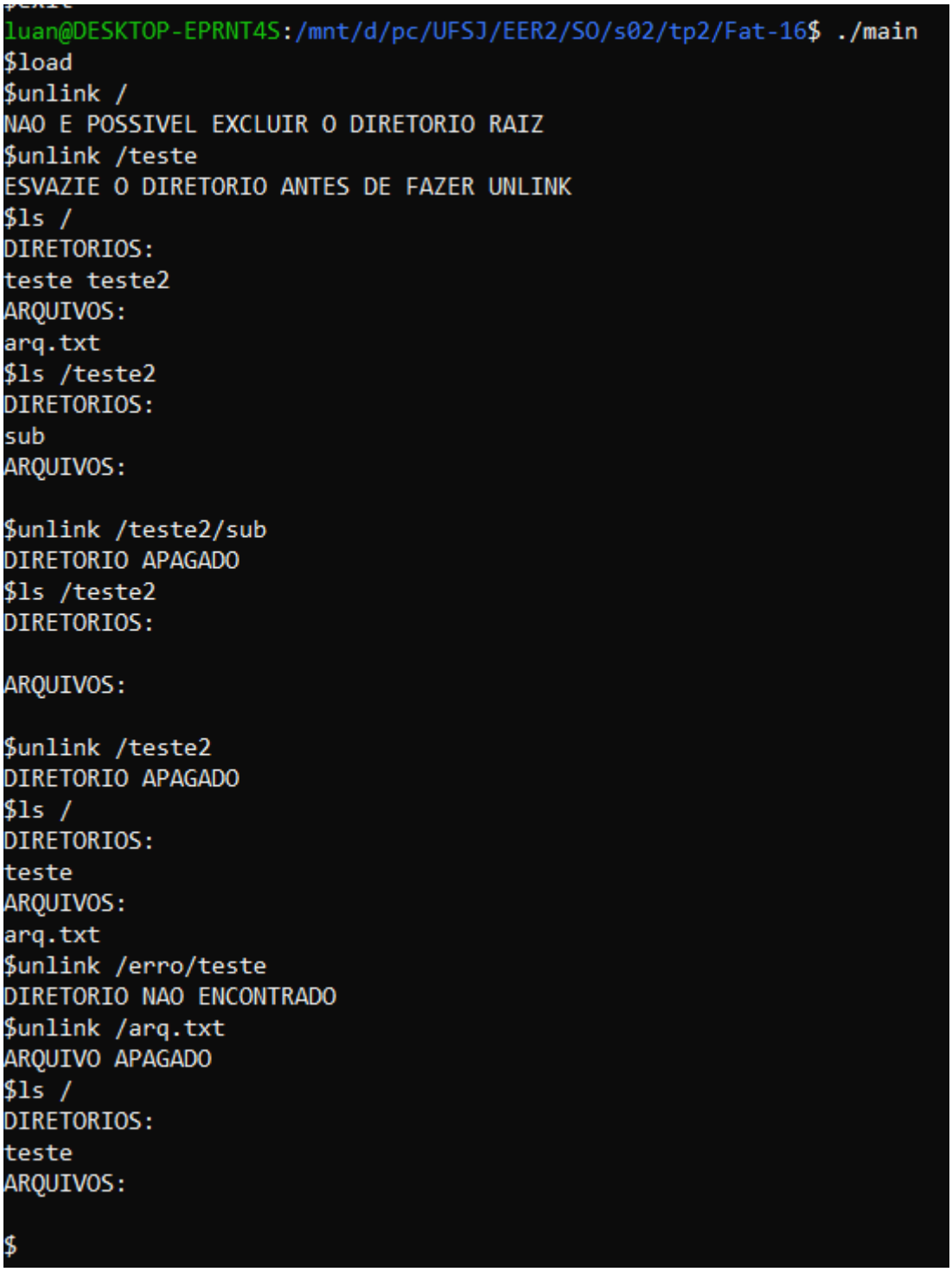
Pode ser observado a utilização do init, mkdir, create e ls. Após a utilização do mkdir e do create, quando executamos o ls/ os diretórios e o arquivo criado foram listados corretamente.

```
luan@DESKTOP-EPRNT4S:/mnt/d/pc/UFSJ/EER2/SO/s02/tp2/Fat-16$ ./main
$load
$ls /
DIRETORIOS:
teste teste2
ARQUIVOS:
arq.txt
$write teste de escrita /arq.txt
$read /arq.txt
teste de escrita
$append  adicionar string /arq.txt
$read /arq.txt
teste de escrita  adicionar string
$exit
```

O comando ls / após o load, retornou corretamente os diretórios e arquivos criados na execução anterior. Dessa forma, o arquivo fat.part foi lido corretamente. O comando write escreveu corretamente no arquivo selecionado como pode ser observado após o comando read, o mesmo pode ser também observado com o comando append.

```
luan@DESKTOP-EPRNT4S:/mnt/d/pc/UFSJ/EER2/SO/s02/tp2/Fat-16$ ./main
$mkdir /teste
ERRO NA FAT, UTILIZE INIT OU LOAD
$load
$mkdir /erro/teste
DIRETORIO NAO ENCONTRADO
$mkdir /teste
DIRETORIO JA EXISTE
$mkdir /
NAO E POSSIVEL CRIAR O DIRETORIO RAIZ
$create /erro/texto.txt
DIRETORIO NAO ENCONTRADO
$create /arq.txt
ARQUIVO JA EXISTE
$create /
PARAMETRO INVALIDO
$exit
```

É importante ressaltar que foram tratadas diversas exceções. Como a formatação da entrada a qual deve ser respeitada em todos os comandos. As strings que serão salvas não necessitam de aspas e todo caminho deve ter uma / antes, não sendo necessário uma / no fim do caminho. As outras exceções tratadas são: se o caminho existe e, se existe ou não o arquivo ou diretório.




Nesta última imagem, é demonstrado o funcionamento do unlink onde não é permitido excluir o diretório raiz, diretórios que existem arquivos ou subdiretórios. Após a utilização do unlink é possível observar que o ls / não lista mais o teste 2.

## 8. Conclusão

Tendo em vista todas as recomendações, estudos discussões sobre o tema e implementações realizadas, foi possível finalizar o trabalho dentro do prazo descrito e com aproveitamento total de tudo aquilo que foi proposto.

Dessa forma concluímos que a realização do trabalho foi de suma importância para o aprendizado sobre sistemas de arquivos do sistema operacional, visto que trabalhamos através de uma simulação, utilizando diversos conceitos e estruturas até chegarmos no código correto e entregarmos o trabalho com maior facilidade em enxergar soluções para o que foi proposto pelo assunto e futuramente utilizar do entendimento para situações reais.

## 9. Referências

[1] Andrew S. Tanenbaum. Sistemas Operacionais Modernos - 3º Edição. Pearson 2010.

[2] Materiais postados no Campus Virtual da disciplina de Sistemas Operacionais.