3º Trabalho[1]

Sistema de Arquivos

Quinta-feira, 25 de março de 2021.

Desenvolva um programa chamado filesystem para simular o funcionamento de um Sistema de Arquivos (*File System*). Todas as operações desse programa devem ser realizas em memória, nenhum arquivo deve ser realmente criado no sistema de arquivos do sistema operacional. Os arquivos serão criados em um disco virtual ou lógico que é mantido somente na memória principal do computador, nada é gravado no disco do computador.

Ao ser executado o programa deve mostrar o *prompt* abaixo.

```
fs>
```

Esse *prompt* será usado pelo usuário para executar os comandos a seguir. Após a execução de cada comando o *prompt* deve ser reexibido, pois essa é a forma de interação do usuário com o sistema de arquivos.

Esse sistema de arquivos deve oferecer os seguintes serviços:

1. Criar disco

Cria um disco virtual no sistema de arquivos.

Sintaxe: `fs> cd <disco>`

Exemplos: Cria os discos virtuais C e D.

```
fs> cd C
fs> cd D
```

2. Criar arquivo

Cria um arquivo no disco virtual.

Sintaxe: `fs> ca <disco> <arquivo>`

Exemplos: Cria os arquivos palavras e frases, respectivamente, nos discos virtuais C e D.

```
fs> ca C palavras
fs> ca D frases
```

[1] Atualizado em 29/03/2021.

3. Abrir arquivo

Abre um arquivo armazenado no disco virtual. Essa operação é essencial, porque o conteúdo do arquivo só poderá ser exibido, escrito, lido ou fechado se o arquivo estiver aberto.

Sintaxe: `fs> aa <disco> <arquivo>`

Exemplos: Abre os arquivos palavras e frases, respectivamente, nos discos virtuais C e D.

```
fs> aa C palavras
fs> aa D frases
```

4. Escrever arquivo

Escreve um número (inteiro ou real) ou uma *string* no arquivo armazenado no disco virtual. A escrita sempre deve ser feita no fim do arquivo.

Sintaxe: `fs> ea <disco> <arquivo> <número | string>`

Exemplos: Escreve os números inteiros 25, 10. 5, 150 e 2000 no arquivo números armazenado no disco D. Depois escreve as palavras sol, praia e mar no arquivo palavras no disco C.

```
fs> ea D numeros 25 10.5 150 2000
fs> ea C palavras sol praia mar
```

5. Ler arquivo

Lê todo o conteúdo de um arquivo armazenado no disco virtual e exibe na tela.

Sintaxe: `fs> la <disco> <arquivo>`

Exemplos: Lê e exibe o conteúdo dos arquivos palavras e frases, respectivamente, nos discos virtuais C e D.

```
fs> la C palavras
fs> la D frases
```

6. Fechar arquivo

Fecha um arquivo armazenado no disco virtual. Após essa operação o conteúdo do arquivo não poderá ser exibido, escrito ou lido até que ele seja aberto novamente.

Sintaxe: `fs> fa <disco> <arquivo>`

Exemplos: Fecha os arquivos palavras e frases, respectivamente, nos discos virtuais C e D.

```
fs> fa C palavras
fs> fa D frases
```

7. Excluir arquivo

Apaga (*delete)* um arquivo armazenado no disco virtual. Essa operação só pode ser realizada após o usuário confirmar respondendo S ou N para, respectivamente, confirmar (sim) ou cancelar (não) a exclusão do arquivo.

Sintaxe: `fs> da <disco> <arquivo>`

Exemplos: Apaga os arquivos palavras e frases, respectivamente, dos discos virtuais C e D.

```
fs> da C palavras
fs> da D frases
```

8. Listar disco

Exibe informações sobre o disco e cada arquivo armazenado nele.

Sintaxe: `fs> ld <disco>`

Exemplo: Exibe as informações do disco virtual C.

```
fs> ld C
```

Conteúdo do disco C

Nome	Tipo	Hora	Tamanho	Blocos
Palavras	Numérico	18:15:45	4.096 bytes	1.024
Números	Texto	18:20:13	64 bytes	16

Número de arquivos: 2

Capacidade do disco: 1.048.576 bytes
Espaço ocupado: 4.160 bytes
Espaço livre: 1.044.416 bytes

Total de blocos lógicos: 262.144
Blocos lógicos ocupados: 1.040
Blocos lógicos livres: 261.104

O leiaute usado pelo programa filesystem para exibir os dados do disco deve ser o mesmo do exemplo acima.

9. Formatar disco

Formata logicamente o disco virtual apagando todo o conteúdo de sua tabela de alocação de arquivos, consequentemente apagando todos os arquivos do disco. Essa operação só pode ser realizada após o usuário confirmar respondendo S ou N para, respectivamente, confirmar (sim) ou cancelar (não) a formatação do disco.

Sintaxe: `fs> fd <disco>`

Exemplo: Formata o disco virtual C.

```
fs> fd C
```

Desenvolva o programa usando as definições abaixo. Estes protótipos de funções representam as chamadas de sistema a serem executadas pelo Sistema de Arquivos do filesystem quando o usuário solicitar a execução dos comandos apresentados acima.

```
#ifndef FILE_SYSTEM_H
#define FILE_SYSTEM_H

#define FALSE 0
#define TRUE 1
#define ERRO_ABRIR_ARQUIVO 2
#define ERRO_CRIAR_ARQUIVO 3
#define ERRO_FECHAR_ARQUIVO 4
#define ERRO_ESCREVER_ARQUIVO 5
#define ERRO_EXCLUIR_ARQUIVO 6
#define ERRO_LER_ARQUIVO 7
#define FIM_DE_ARQUIVO 8

// Número máximo de caracteres no nome do arquivo.
#define TAMANHO_NOME_ARQUIVO 13

// Comprimento da hora de criação do arquivo no formato hh:mm:ss.
#define TAMANHO_HORA 9

// Tamanho do bloco lógico em bytes.
#define TAMANHO_BLOCO_LOGICO 4

// Número máximo de arquivos do disco virtual.
#define NUMERO_DE_ARQUIVOS_DO_DISCO 255

// Representação descritiva dos tipos de dados que um arquivo pode armazenar.
#define TIPO_NUMERICO "Numérico"
#define TIPO_TEXTO "Texto"

/* Representa os tipos de dados que um arquivo pode armazenar.
   Os tipos são numérico (int ou float) e string.
*/
typedef enum {NUMERICO, STRING} TIPO_DE_ARQUIVO;

// Representa os atributos de um arquivo.
typedef struct
{
    // Descritor de arquivo usado para identificar um arquivo na tabela de alocação de arquivos do disco.
    unsigned short fd;

    // Nome e hora de criação do arquivo no formato hh:mm:ss.
    char nome[TAMANHO_NOME_ARQUIVO], horaCriacao[TAMANHO_HORA];

    // Tamanho do arquivo em bytes.
    unsigned int tamanho;
```

```
    // Número de blocos lógicos do arquivo.
    unsigned short blocosLogicos;

    // Armazena o conteúdo do arquivo. O tipo de dado do conteúdo é definido pelo campo tipo.
    void* conteudo;

    // Representa o tipo de dado que o arquivo pode armazenar.
    TIPO_DE_ARQUIVO tipo;

} Arquivo;

// Representa os atributos de um disco virtual.
typedef struct
{
        // Letra de identificação da unidade de disco.
        char unidade;

        // Capacidade de armazenamento do disco.
        unsigned int capacidade;

        // Espaço ocupado do disco.
        unsigned int espacoOcupado;

        // Espaço livre do disco.
        unsigned int espacoLivre;

        // Número total de blocos lógicos.
        unsigned short blocosLogicosTotais;

        // Número de blocos lógicos ocupados do disco.
        unsigned short blocosLogicosOcupados;

        // Número de blocos lógicos livres do disco.
        unsigned short blocosLogicosLivres;

        /*A tabela de alocação de arquivos é usada para armazenar todos os arquivos do disco.
          A posição de um arquivo nessa tabela é igual ao valor inteiro do seu descritor de arquivo.
        */
        Arquivo *tabelaArquivos[NUMERO_DE_ARQUIVOS_DO_DISCO];

} Disco;

/*
   Cria um disco virtual.
   A capacidade especifica a capacidade máxima de armazenamento em bytes do disco, ou seja, o seu
   tamanho "físico". O parâmetro unidade especifica um caractere usado para identificação do disco, por
   exemplo C, D, E, etc. Em caso de sucesso, retorna um ponteiro para estrutura do disco criada, caso
   contrário, NULL.
*/
Disco* criaDisco(char unidade, unsigned int capacidade);
```

```
/*
   Formata logicamente o disco virtual apagando todo o conteúdo de sua tabela de alocação de arquivos,
   consequentemente apagando todo os arquivos do disco.
   O parâmetro unidade especifica um caractere usado para identificação do disco, por exemplo C, D, E, etc.
   Reinicializa os dados da estrutura do disco em caso de sucesso e retorna TRUE, caso contrário, FALSE.
   Todas as memórias alocadas dinamicamente para armazenar a estrutura e o conteúdo de cada arquivo
   devem ser devolvidas ao sistema operacional.
*/
int formatar(char unidade, Disco* disco);

/*
   Cria um arquivo para leitura e escrita no disco identificado pela unidade.
   Retorna um inteiro que representa o descritor de arquivo ou ERRO_CRIAR_ARQUIVO se o arquivo não pode
   ser criado.
 */
int criar(char unidade, const char *nomeArquivo);

/*
   Abre um arquivo para leitura e escrita localizado no disco identificado pela unidade.
   Retorna TRUE se o arquivo foi aberto e ERRO_ABRIR_ARQUIVO se ocorrer algum erro, por exemplo, o
   arquivo não existe.
*/
int abrir(char unidade, const char* nomeArquivo);

/*
   Escreve no arquivo n elementos obtidos do buffer. O valor de n é definido por tamanho.
   O arquivo é identificado pelo descritor de arquivo fd e está localizado no disco identificado pela unidade.
   Retorna o número de bytes escritos após uma escrita bem sucedida. Se ocorrer algum erro na escrita
   retorna ERRO_ESCREVER_ARQUIVO.

   NOTA: O bloco lógico é a unidade básica de alocação de espaço (memória) no disco, portanto sempre que
         uma escrita é feita no disco deve-se escrever no mínimo um bloco lógico de tamanho definido em
         TAMANHO_BLOCO_LOGICO. Sendo assim, quando o usuário solicita que seja escrito um número
         inteiro de 4 bytes (valor de TAMANHO_BLOCO_LOGICO), usa-se um bloco lógico para armazená-lo
         no arquivo, mas quando é escrito a string "Férias!", usa-se 2 blocos lógicos. Como essa string possui
         7 caracteres e cada caractere (char) ocupa 1 byte de memória, ela precisa de 7 bytes de memória.
         No entanto, como cada bloco lógico possui 4 bytes, a string exige 2 blocos lógicos (8 bytes) para ser
         armazenada no disco, logo deve-se escrever dois blocos.
*/
int escrever(char unidade, unsigned short fd, const void* buffer, unsigned tamanho);

/*
   Lê n elementos do arquivo e armazena no buffer. O valor de n é definido por tamanho.
   O arquivo é identificado pelo descritor de arquivo fd e está localizado no disco identificado pela unidade.
   Retorna o número de elementos lidos do arquivo em uma leitura bem sucedida. Retorna
   FIM_DE_ARQUIVO se o fim do arquivo for atingido após a leitura e ERRO_LER_ARQUIVO se ocorrer algum
   erro na leitura.

   NOTA: O bloco lógico é a unidade básica de alocação de espaço (memória) no disco, portanto sempre que
         uma leitura é feita no disco deve-se ler no mínimo um bloco lógico de tamanho definido em
         TAMANHO_BLOCO_LOGICO. Sendo assim, quando o usuário solicita que seja lido um número
         inteiro de 4 bytes (valor de TAMANHO_BLOCO_LOGICO), lê-se um bloco lógico para recuperá-lo do
         arquivo, mas quando é lido a string "Férias!", lê-se 2 blocos lógicos. Como essa string possui 7
```

```
        caracteres e cada caractere (char) ocupa 1 byte de memória, ela precisa de 7 bytes de memória. No
        entanto, como cada bloco lógico possui 4 bytes, a string exige 2 blocos lógicos (8 bytes) para ser
        armazenada no disco, logo deve-se ler os dois blocos.
*/
int ler(char unidade, unsigned short fd, void* buffer, unsigned tamanho);

/*
   Fecha um arquivo (descritor de arquivo fd) que está localizado no disco identificado pela unidade. Retorna
   TRUE se a operação for bem sucedida e ERRO_FECHAR_ARQUIVO se não conseguir fechar o arquivo.
*/
int fechar(char unidade, unsigned short fd);

/*
   Apaga um arquivo do disco identificado por unidade.
   Retorna TRUE se a operação for bem sucedida e ERRO_EXCLUIR_ARQUIVO se não conseguir excluir o
   arquivo do disco virtual.
*/
int excluir(char unidade, const char* nomeArquivo);

/*
    Exibe um relatório com as seguintes informações para cada arquivo do disco identificado por unidade:

    1. Nome;
    2. Tipo;
    3. Tamanho em bytes;
    4. Número de blocos lógicos;
    5. Hora de criação.

    No fim da listagem deve exibir também os seguintes dados do disco virtual:

    1. Número de arquivos;
    2. A capacidade de armazenamento;
    3. O espaço ocupado do disco;
    4. O espaço livre do disco;
    5. O número total de blocos lógicos;
    6. O número de blocos lógicos ocupados;
    7. O número de blocos lógicos livres.

    Os valores dos itens 5 a 7 devem ser exibidos em bytes.

    Retorna o endereço de memória que possui o conteúdo do relatório. Se ocorrer um erro ao obter esses
    dados retorna NULL.
*/
char* infoDisco(char unidade);

#endif
```

- Critérios de avaliação

1. O trabalho será avaliado considerando:
    a. A validação dos dados fornecidos pelo usuário.

b. A lógica empregada na solução do problema.
c. O funcionamento do programa.
d. O conhecimento da linguagem de programação C.
e. O uso do princípio do menor privilégio[2]. Veja um exemplo abaixo.
f. Código fonte sem erros e sem advertências do compilador.
g. Código fonte legível, indentado, organizado e comentado.
h. Identificadores significativos para aprimorar a inteligibilidade do código fonte.

2. O programa deve ser desenvolvido integralmente usando apenas os recursos da linguagem C e do *Microsoft Visual Studio Community* 2019, versão 16.8. Programas desenvolvidos em outras linguagens, mesmo que parcialmente, receberão nota zero.

3. Para que o programa seja avaliado o código deve executar com sucesso. Programas que apresentarem erros de compilação, ligação ou *segmentation fault* receberão nota zero.

4. Trabalhos com plágio, ou seja, programas com código fonte copiados de outra pessoa (cópia integral ou parcial) receberão nota zero.

5. O desenvolvimento do trabalho é individual.

6. Escrever funções e métodos específicos, ou seja, com atribuição clara e objetiva.

   Exemplo: Pesquisa um nome em um vetor de *strings*. Retorna a posição do nome no vetor se ele for encontrado ou -1 caso contrário.

```
int pesquisarNome(const char* vetor[], const char* nome);
```

   A descrição dessa função deixa claro que não é atribuição dela ler o nome via algum dispositivo de E/S e nem exibir o resultado da consulta, somente realizar a pesquisa do nome no vetor e devolver o resultado.

   O uso do *const* no protótipo de função acima é um exemplo do princípio do menor privilégio, porque se a função não precisa alterar os argumentos usados na sua chamada, ela não pode ter parâmetros formais com esse poder ou privilégio. O uso do *const* assegura que apesar da função receber a referência dos argumentos ela não poderá modificá-los.

7. Não escrever código redundante.

8. É proibido modificar os nomes de arquivos, identificadores e os protótipos de função fornecidos neste texto ou em anexo.

9. É permitido acrescentar novas declarações e/ou definições de funções, variáveis e constantes desde que estejam de acordo com os critérios acima.

- Instruções para entrega do trabalho

1. Crie uma solução com o nome FileSystem e um projeto com o seu nome e sobrenome, por exemplo: AyrtonSenna.

[2] O **princípio do menor privilégio** declara que deve ser concedido ao código somente a quantidade de privilégio e acesso de que ele precisa para realizar sua tarefa designada, não mais que isso.

2. Limpe a solução para apagar todos os arquivos OBJ da pasta *Debug* do projeto. Confira se esses arquivos realmente foram excluídos da pasta *Debug*. De preferência exclua todo o conteúdo dessa pasta.

3. Antes de submeter os exercícios via SIGAA, compacte apenas o diretório do projeto para criar um arquivo 7z com o seu nome e sobrenome, por exemplo: AyrtonSenna.7z.

   Não inclua no arquivo 7z o diretório da solução FileSystem, somente o diretório do projeto que possui o seu nome e sobrenome.

   Assim o tamanho final do arquivo 7z não ultrapassará o limite de 10 MB do SIGAA, porque o conteúdo do diretório oculto .vs, criado dentro da pasta da solução, não será compactado, reduzindo drasticamente o tamanho final do arquivo.

   Para compactar o projeto use o *software* livre de código aberto 7-Zip, que está disponível em https://www.7-zip.org/download.html.

- Data de entrega

Quinta-feira, 1º de abril de 2021.

- Valor do trabalho

10,0 pontos.

Prof**. M**árlon **O**liveira **d**a **S**ilva
marlon.silva@ifsudestemg.edu.br