Sistemas Operativos

Licenciatura em Engenharia de Sistemas Informáticos (pós-laboral)

Escola Superior de Tecnologia IPCA

2023/24

Professor Fernando Gomes

Fernando Salgueiro - 39

Hélder Costa – 29576

Hugo Lopes - 30516

Cláudio Fernandes - 30517

Nuno Cruz - 30518

Maio de 2024

Índice

Introdução

O presente trabalho prático tem como objetivo a utilização prática dos conceitos fundamentais de gestão de processos e de ficheiros em sistemas operativos.

Ao longo deste projeto, iremos explorar e implementar um conjunto de comandos para manipulação de ficheiros, e proceder com o desenvolvimento de um interpretador de linha de comandos.

Esta parte do projeto será implementada em linguagem C, e será executada em ambiente Linux.

Iremos também abordar questões relacionadas com a gestão de sistemas de ficheiros, em que engloba a criação de discos e respetivas partições, criação e montagem de sistemas de ficheiros, entre outros procedimentos.

Este trabalho será realizado com um grupo 5 elementos, conforme orientações do Professor.

A parte 1 e 2 foi implementada pelos alunos:

- Fernando Salgueiro - 39
- Hélder Costa – 29576
- Hugo Lopes - 30516
- Cláudio Fernandes – 30517

A parte 3 foi implementada pelo aluno:

- Nuno Cruz – 30518

Parte 1: Manipulação de ficheiros

a) Implementação do comando “mostra”

Este comando deve apresentar no ecrã todo o conteúdo do ficheiro indicado como parâmetro. Caso o ficheiro não exista, o comando deve avisar o utilizador que o ficheiro não existe.

- Validação dos argumentos:

```
if (argc != 2)
{
    write(2, "Erro: Digite os argumentos: ", 28);
    write(2, argv[0], strlen(argv[0]));
    write(2, " <nome_ficheiro>\n", 18);
    return 1;
}
```

Figura 1 - mostra - Validação de argumentos

O código que se encontra na figura 1, verifica se o número de argumentos passados na linha de comandos é exatamente dois (o nome do programa e um argumento adicional relativo ao nome do ficheiro pretendido).

Caso o número de argumentos seja diferente de dois, o programa exibe uma mensagem de erro utilizando a função *write* para imprimir uma mensagem de erro no *stderr*, com o objetivo de instruir o utilizador acerca do uso correto para a execução do programa.

No final retorna 1, para indicar que ocorreu um erro devido ao número incorreto de argumentos.

- Abertura e leitura do ficheiro:

```
// Abre o ficheiro para leitura
fd = open(argv[1], O_RDONLY);
if (fd == -1) {
    write(2, "Erro na abertura do ficheiro\n", 30);
    return 1;
}

// Lê e mostra a informação do ficheiro
while ((tam = read(fd, buffer, BUFFER_SIZE)) > 0)
{
    // Escreve os dados lidos no stdout
    if (write(1, buffer, tam) != tam)
    {
        write(2, "Erro na escrita no stdout\n", 27);
        close(fd);
        return 1;
    }
}
```

Figura 2 - mostra: Abertura e leitura do ficheiro

O código que se encontra na figura 2, efetua a abertura e a leitura do ficheiro indicado (argumento) pelo utilizador.

Para abrir o ficheiro, utiliza a função *open* em modo de leitura (O_RDONLY).

Caso ocorra algum problema na abertura do ficheiro (if (fd == -1), exibe uma mensagem de erro no stderr e retorna 1 para indicar o referido erro.

Para a leitura e escrita do conteúdo do ficheiro, o programa utiliza um ciclo '*while*' para ler o conteúdo do ficheiro em blocos de tamanho *BUFFER_SIZE* e guardar no *buffer*, até que a função '*read*' retorne '0' (final do ficheiro):

- while ((tam = read(fd, buffer, BUFFER_SIZE)) > 0)

Ainda dentro do ciclo, a função '*write*' é utilizada para escrever os dados lidos no stdout.

Escrita dos dados lidos no stdout (file descriptor 1): if (write(1, buffer, tam) != tam)

Verifica se a quantidade escrita é igual à quantidade lida (tam). Se a escrita falhar, retorna 1.

- Tratamento de erros e fecho do ficheiro:

```
// Valida se ocorreu algum erro durante a leitura do ficheiro
if (tam == -1) {
    write(2, "Erro na leitura do ficheiro\n", 29);
    close(fd);
    return 1;
}

// Valida se ocorreu algum erro no fecho do ficheiro
if (close(fd) == -1)
{
    write(2, "Erro no fecho do ficheiro\n", 27);
    return 1;
}

return 0;  // Sucesso
```

Figura 3 - mostra: Tratamento de erros e fecho do ficheiro

Após ler e apresentar o conteúdo do ficheiro, são efetuadas duas validações para garantir que as operações foram executadas com sucesso:

- if (tam == -1): Verifica se ocorreu um erro durante a leitura (read retorna -1 em caso de erro).
- if (close(fd) == -1): Se existir uma falha ao fechar o ficheiro (close retorna -1 em caso de erro), retorna 1.

b) Implementação do comando “copia”

Este comando deve criar um novo ficheiro, cujo nome é “ficheiro.copia”, cujo conteúdo é uma cópia de (todo) o conteúdo do ficheiro passado como parâmetro no comando, com o nome ficheiro. Caso o ficheiro não exista, deve ser apresentado um aviso ao utilizador.

- Validação dos argumentos:

```
if (argc != 2)
{
    write(2, "Erro: Digite os argumentos: ", 29);
    write(2, argv[0], strlen(argv[0]));
    write(2, " <nome_ficheiro>\n", 18);
    return 1;
}
```

Figura 4 - copia - Validação de argumentos

O código que se encontra na figura 4, verifica se o número de argumentos passados na linha de comandos é exatamente dois (o nome do programa e um argumento adicional relativo ao nome do ficheiro pretendido).

Caso o número de argumentos seja diferente de dois, o programa exibe uma mensagem de erro utilizando a função ‘*write’* para imprimir uma mensagem de erro no *stderr*, com o objetivo de instruir o utilizador acerca do uso correto para a execução do programa.

No final retorna ‘1’, para indicar que ocorreu um erro devido ao número incorreto de argumentos.

- Leitura do ficheiro de entrada e criação do ficheiro de saída:

```
// Abre o ficheiro de entrada para leitura
fdInput = open(argv[1], O_RDONLY);
if (fdInput == -1)
{
    write(2, "Ficheiro não encontrado\n", 26);
    return 1;
}

// Abre ou cria o ficheiro de saída para escrita
fdOutput = open("ficheiro.copia", O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
if (fdOutput == -1)
{
    write(2, "Erro na abertura ou na criação do ficheiro de saída\n", 56);
    close(fdInput);
    return 1;
}
```

Figura 5 - copia: Leitura do ficheiro de entrada e criação do ficheiro de saída

O ficheiro de entrada é aberto em modo de leitura (O_RDONLY), através da função open. Caso a função ‘*open’* retorne ‘-1’, quer dizer que o ficheiro não foi encontrado. Exibe uma mensagem de erro no stderr e retorna 1 para indicar o referido erro.

O ficheiro de saída é aberto em modo de escrita (*O_WRONLY*). No entanto, caso o ficheiro não exista, o mesmo será criado com o nome "ficheiro.copia" (*O_CREAT*).

As permissões do ficheiro são definidas como leitura e escrita para o proprietário (*S_IRUSR* | *S_IWUSR*), leitura para o grupo (*S_IRGRP*), e leitura para outros (*S_IROTH*).

Para validar se a abertura ou criação do ficheiro de saída falhou: *if (fdOutput == -1).*

Neste caso, exibe uma mensagem de erro no *stderr*. fecha o ficheiro de entrada e retorna '1' para indicar a falha.

```
// Copia o conteúdo do ficheiro de entrada para o ficheiro de saída
while ((tam = read(fdInput, buffer, BUFFER_SIZE)) > 0)
{
    if (write(fdOutput, buffer, tam) != tam)
    {
        write(2, "Erro na escrita do ficheiro de saída\n", 39);
        close(fdInput);
        close(fdOutput);
        return 1;
    }
}
```

Figura 6 - copia: Copia dados do ficheiro de entrada para o ficheiro de saída

O código que se encontra na figura 6, é o responsável pela cópia do conteúdo do ficheiro de entrada para o ficheiro de saída. Efetua a leitura dos dados do ficheiro de entrada em blocos e escreve os mesmos no ficheiro de saída, tratando erros de leitura e escrita:

- while ((tam = read(fdInput, buffer, BUFFER_SIZE)) > 0): Lê blocos de dados do ficheiro de entrada para o buffer.
- if (write(fdOutput, buffer, tam) != tam): Escreve os dados lidos no ficheiro de saída. Verifica se a quantidade escrita é igual à quantidade lida. Se a escrita falhar, retorna 1.

- Tratamento de erros e fecho dos ficheiros:

```
// Valida se ocorreu algum erro durante a leitura do ficheiro de entrada
if (tam == -1)
{
    write(2, "Erro na leitura do ficheiro de entrada\n", 40);
    close(fdInput);
    close(fdOutput);
    return 1;
}

// Fecha os ficheiros de entrada e de saída
close(fdInput);
close(fdOutput);

write(1, "Ficheiro criado com sucesso\n", 28);

return 0;
```

Figura 7 - copia: Tratamento de erros e fecho dos ficheiros

Após copiar o conteúdo do ficheiro de entrada para o ficheiro de saída, são efetuadas validações para garantir que as operações foram executadas com sucesso (figura 7).

Caso exista um erro durante a leitura do ficheiro de entrada, o programa exibe uma mensagem de erro no stderr e retorna '1' para indicar o erro.

Posteriormente os ficheiros de entrada e de saída são fechados, através da função '*close*'.

No final, se todo o programa foi executado sem qualquer erro, é exibida uma mensagem a indicar que o ficheiro foi criado com sucesso e é retornado '0' (operação executada com sucesso).

c) Implementação do comando 'acrescenta'

Este comando deve acrescentar (todo) o conteúdo da "origem" no final do "destino". Caso algum dos ficheiros não exista, deve ser apresentado um aviso ao utilizador.

- Validação dos argumentos:

```
if (argc != 3)
{
    write(2, "Erro: Digite os argumentos: ", 29);
    write(2, argv[0], strlen(argv[0]));
    write(2, " nome_ficheiro_origem nome_ficheiro_destino\n", 45);
    return 1;
}
```

Figura 8 - acrescenta: Validação de argumentos

O código que se encontra na figura 8, verifica se o número de argumentos passados na linha de comandos é exatamente três (o nome do programa, nome do ficheiro de origem e o nome do ficheiro de destino).

Caso o número de argumentos seja diferente de três, o programa exibe uma mensagem de erro utilizando a função '*write'* para imprimir uma mensagem de erro no *stderr*, com o objetivo de instruir o utilizador acerca do uso correto para a execução do programa.

No final retorna '1', para indicar que ocorreu um erro devido ao número incorreto de argumentos.

- Abertura dos ficheiros de entrada e saída:

```
// Abertura do ficheiro de entrada para leitura
fdInput = open(argv[1], O_RDONLY);
if (fdInput == -1)
{
    write(2, "Erro na abertura do ficheiro de entrada\n", 41);
    return 1;
}

// Abertura do ficheiro de saída para escrita (opção O_APPEND)
fdOutput = open(argv[2], O_WRONLY | O_APPEND);
if (fdOutput == -1)
{
    write(2, "Erro na abertura do ficheiro de saída\n", 40);
    close(fdInput);
    return 1;
}
```

Figura 9 - acrescenta: Abertura dos ficheiros de entrada e saída

O ficheiro de entrada é aberto em modo de leitura (*O_RDONLY*) através da função '*open'*. Caso a função retorne '-1', o ficheiro não foi aberto e consequentemente é exibida uma mensagem de erro no *stderr*. Retorna '1' para indicar a falha.

O ficheiro de saída é aberto em modo de escrita (*O_WRONLY*) com a opção '*O_APPEND'*, por forma a que todos os dados escritos no ficheiro sejam adicionados no final do mesmo. Caso a função retorne '-1', ocorreu um erro na abertura do ficheiro de saída e sendo assim é exibida uma mensagem de erro no stderr. Fecha o ficheiro de entrada e retorna 1 para indicar a falha.

- Leitura do ficheiro de entrada e escrita no ficheiro de saída:

```
while ((tamLeitura = read(fdInput, buffer, BUFFER_SIZE)) > 0)
{
    tamEscrita = write(fdOutput, buffer, tamLeitura);
    if (tamEscrita != tamLeitura)
    {
        write(2, "Erro na escrita do ficheiro de saída\n", 39);
        close(fdInput);
        close(fdOutput);
        return 1;
    }
}
```

Figura 10 - acrescenta: Leitura do ficheiro de entrada e escrita no ficheiro de saída

O código descrito na figura 10 é responsável por efetuar a leitura do conteúdo do ficheiro de entrada e escrevê-lo no ficheiro de saída. Executa operações de leitura e escrita em blocos de dados, tratando erros de escrita:

  - while ((tamLeitura = read(fdInput, buffer, BUFFER_SIZE)) > 0): Lê blocos de dados do ficheiro de origem para o buffer.
  - if (write(fdOutput, buffer, tamLeitura) != tamLeitura): Escreve os dados lidos no ficheiro de destino. Verifica se a quantidade escrita é igual à quantidade lida. Se a escrita falhar, retorna 1.

- Tratamento de erros e fecho dos ficheiros:

```
// Valida se ocorreu algum erro durante a leitura do ficheiro de entrada
if (tamLeitura == -1)
{
    write(2, "Erro na leitura do ficheiro de entrada\n", 40);
    close(fdInput);
    close(fdOutput);
    return 1;
}

// Fecho dos ficheiros de entrada e saída
if (close(fdInput) == -1)
{
    write(2, "Erro no fecho do ficheiro de entrada\n", 38);
    return 1;
}

if (close(fdOutput) == -1)
{
    write(2, "Erro no fecho do ficheiro de saída\n", 37);
    return 1;
}

// Caso não existam erros, indica que os dados foram inseridos com sucesso
write(1, "Dados inseridos com sucesso\n", 29);

return 0;  // Sucesso
```

Figura 11 - acrescenta: Tratamento de erros e fecho dos ficheiros

Analisa se a última operação de leitura (*tamLeitura*) retornou '-1', que indica um erro na leitura do ficheiro de entrada. Em caso afirmativo, exibe uma mensagem de erro no *stderr* e fecha os ficheiros de entrada e de saída. Retorna '1' para indicar a falha.

Para fechar os ficheiros de entrada e de saída, é utilizada a função '*close*'.

No caso de ocorrer algum erro durante o fecho de qualquer um dos ficheiros supracitados, é exibida uma mensagem de erro e retorna '1' para indicar o erro.

No final, se todo o programa foi executado sem qualquer erro, é exibida uma mensagem a indicar que o ficheiro foi criado com sucesso e é retornado '0' (operação executada com sucesso).

d) Implementação do comando 'conta'

Este comando deve contar o número de linhas existentes num ficheiro. Se o ficheiro não existir, deverá ser indicado ao utilizador uma mensagem de erro;

- Validação dos argumentos:

```
if (argc != 2) {
    write(2, "Erro: Digite os argumentos: ", 29);
    write(2, argv[0], strlen(argv[0]));
    write(2, " <nome_ficheiro>\n", 18);
    return 1;
}
```

Figura 12 - conta: Validação de argumentos

O código que se encontra na figura 12, verifica se o número de argumentos passados na linha de comandos é exatamente dois (o nome do programa e um argumento adicional relativo ao nome do ficheiro pretendido).

Caso o número de argumentos seja diferente de dois, o programa exibe uma mensagem de erro utilizando a função '*write*' para imprimir uma mensagem de erro no *stderr*, com o objetivo de instruir o utilizador acerca do uso correto para a execução do programa.

No final retorna '1', para indicar que ocorreu um erro devido ao número incorreto de argumentos.

- Abertura e leitura do ficheiro:

```
fd = open(argv[1], O_RDONLY);
if (fd == -1)
{
    write(2, "Erro na abertura do ficheiro\n", 30);
    return 1;
}

// Ciclo para a leitura do ficheiro caractere a caractere e para contar o número de linhas
while (read(fd, &ch, 1) == 1)
{
    if (ch == '\n')
    {
        numLinhas++;
    }
}
```

Figura 13 - conta: Abertura e leitura do ficheiro

O código da figura 13 efetua a abertura de um ficheiro para leitura e conta o número de linhas que se encontram no mesmo. A leitura é efetuada de forma sequencial, caractere por caractere.

O ficheiro indicado no primeiro argumento da linha de comandos (*argv[1]*) é aberto em modo de leitura (*O_RDONLY*) utilizando a função '*open*'. Caso ocorra algum problema, retorna '-1', e é exibida uma mensagem de erro no *stderr*. Retorna 1 para indicar o erro.

Para a leitura do ficheiro e consequentemente contar as linhas do mesmo, é utilizado um ciclo *while* para ler caractere por caractere. A função '*read*' lê um caractere de cada vez (1 byte) e guarda na variável '*ch*'.

O contador de linhas (numLinhas) é incrementado, sempre que encontra um caractere de nova linha ('\n'). O ciclo continua até que a função '*read*' retorne 0 (fim do ficheiro) ou -1 (erro).

- Conversão do número de linhas e imprime no terminal:

```
char linhaString[20];
int len = 0;
temp = numLinhas;
do
{
    linhaString[len++] = temp % 10 + '0';
    temp /= 10;
} while (temp);

// Escreve a string relativa ao número de linhas, invertida no terminal
for (int i = len - 1; i >= 0; i--)
{
    write(1, &linhaString[i], 1);
}
write(1, "\n", 1);

return 0;
```

Figura 14 - conta: Converter e imprimir número de linhas

O código descrito na figura 14, converte o número de linhas de um ficheiro (guardado na variável 'numLinhas') para uma string e imprime no terminal. Inicialmente, guarda cada dígito num array de caracteres em sentido contrário (de trás para frente). Depois imprime os caracteres na ordem correta. No fim, é inserida uma nova linha no terminal para formatar a saída de forma legível.

Inicialização de Variáveis:

- linhaString[20]: Array de caracteres para guardar em string do número de linhas.
- len: Contador para verificar o comprimento da string.
- temp: Variável temporária para guardar o valor de 'numLinhas' durante a conversão.

De forma a converter o número de linhas para *string*, utilizamos um ciclo *do-while*. Em cada iteração, o dígito mais à direita de *temp* é alcançado (usando o operador % 10), convertido para caractere (adicionando '0') e guardado na *linhaString*. Sendo assim, *temp* é dividido por 10 para eliminar o dígito mais à direita. O ciclo continua até que *temp* seja 0. Após esta conversão, a *string* fica guardada em sentido contrário (de trás para frente) na *linhaString*.

É utilizado um ciclo *for* para imprimir os caracteres da *string* pela ordem inversa, de modo a que o número seja apresentado de forma correta. Através da função *write,* são escritos cada um dos caracteres no terminal. Após isto, é inserida uma nova linha ('\n') para separar a saída da linha seguinte no terminal.

e) Implementação do comando 'apaga'

Este comando deve apagar o ficheiro com o nome indicado. No caso de o ficheiro indicado não existir, e apenas, deve ser apresentado um aviso ao utilizador;

- Validação dos argumentos:

```
if (argc != 2)
{
    write(2, "Erro: Digite os argumentos: ", 29);
    write(2, argv[0], strlen(argv[0]));
    write(2, " <nome_ficheiro>\n", 18);
    return 1;
}
```

Figura 15 - apaga: Validação de argumentos

O código que se encontra na figura 15, verifica se o número de argumentos passados na linha de comandos é exatamente dois (o nome do programa e um argumento adicional relativo ao nome do ficheiro pretendido).

Caso o número de argumentos seja diferente de dois, o programa exibe uma mensagem de erro utilizando a função '*write*' para imprimir uma mensagem de erro no *stderr*, com o objetivo de instruir o utilizador acerca do uso correto para a execução do programa.

No final retorna '1', para indicar que ocorreu um erro devido ao número incorreto de argumentos.

- Eliminação do ficheiro:

```
if (unlink(argv[1]) == -1)
{
    write(2, "Erro na eliminação do ficheiro\n", 34);
    return 1;
}

write(1, "Eliminação do ficheiro efetuada com sucesso\n", 47);

return 0;
```

Figura 16 - apaga: Eliminação do ficheiro

O código descrito na figura 16, elimina um ficheiro com o nome que é passado como argumento na linha de comandos. É utilizada a chamada ao sistema *unlink* para eliminar o ficheiro e indica ao utilizador se foi executado com sucesso.

A função *unlink* é utilizada para eliminar o ficheiro indicado por argv[1].

Caso a função *unlink* retorne '-1', significa que existiu um erro ao tentar efetuar a eliminação do ficheiro. Se a eliminação do ficheiro foi executada com sucesso, é exibida uma mensagem de sucesso no terminal, através da função write.

f) Implementação do comando 'informa'

Este comando apresenta apenas a informação do sistema de ficheiros em relação ao ficheiro indicado, tipo de ficheiro (normal, diretoria, link, etc.), i-node, utilizador dono em formato textual e datas de criação, leitura e modificação em formato textual.

- Validação dos argumentos:

```
if (argc < 2)
{
    write(2, "Erro: Digite os argumentos: ", 29);
    write(2, argv[0], strlen(argv[0]));
    write(2, " nome_ficheiro\n", 16);
    return 1;
}
```

Figura 17 - informa: Validação de argumentos

O código que se encontra na figura 17, verifica se o número de argumentos passados na linha de comandos é exatamente dois (o nome do programa e um argumento adicional relativo ao nome do ficheiro pretendido).

Caso o número de argumentos seja diferente de dois, o programa exibe uma mensagem de erro utilizando a função '*write*' para imprimir uma mensagem de erro no *stderr*, com o objetivo de instruir o utilizador acerca do uso correto para a execução do programa.

No final retorna '1', para indicar que ocorreu um erro devido ao número incorreto de argumentos.

- Informação e tipo do ficheiro:

```
char *filename = argv[1];
struct stat file_info;

// Informações sobre o ficheiro
if (stat(filename, &file_info) == -1)
{
    const char *error = "Erro na leitura das informações do ficheiro\n";
    write(2, error, 47);
    return 1;
}

// Determina e imprime o tipo do ficheiro
if (S_ISREG(file_info.st_mode))
    write(1, "Tipo de ficheiro: Ficheiro regular\n", 36);
else if (S_ISDIR(file_info.st_mode))
    write(1, "Tipo de ficheiro: Diretoria\n", 29);
else if (S_ISLNK(file_info.st_mode))
    write(1, "Tipo de ficheiro: Link\n", 24);
else if (S_ISCHR(file_info.st_mode))
    write(1, "Tipo de ficheiro: Ficheiro especial de caracteres\n", 51);
else if (S_ISBLK(file_info.st_mode))
    write(1, "Tipo de ficheiro: Ficheiro especial de blocos\n", 47);
else
    write(1, "Tipo de ficheiro: Outro\n", 25);
```

Figura 18 - informa: Informação e tipo de ficheiro

O código descrito na figura 18, determina e imprime as informações relativas de um ficheiro com o nome passado como argumento na linha de comandos. É utilizada a chamada ao sistema *stat* para identificar estas informações e determina o tipo de ficheiro.

Declaração de Variáveis:

- *filename*: Apontador para a *string* que tem o nome do ficheiro, adquirida pelo argumento da linha de comandos *(argv[1])*.
- *file_info*: Estrutura *stat* que será preenchida com as informações relativas ao ficheiro.

Para obter as informações do ficheiro indicado em '*filename*', é utilizada a função *stat*. Caso retorne -1, significa que ocorreu um erro ao tentar obter as informações. Sendo assim, é escrita uma mensagem de erro no stderr e o programa retorna '1'.

Para identificar o tipo de ficheiro, são utilizadas macros específicas (*S_ISREG, S_ISDIR, S_ISLNK, S_ISCHR, S_ISBLK*) e aplicadas ao campo *st_mode* da estrutura *stat*. Posteriormente escreve uma mensagem no *stdout* com o tipo de ficheiro.

Nesse sentido, é possível reconhecer se o ficheiro é um ficheiro regular, um diretório, um link, ou um ficheiro especial de caracteres ou blocos.

- Inode do ficheiro:

```
write(1, "Inode do ficheiro: ", 20);
char inode_str[20];
int inode_length = 0;
int temp_inode = file_info.st_ino;
if (temp_inode == 0)
{
    inode_str[0] = '0';
    inode_length = 1;
} else
{
    while (temp_inode != 0)
    {
        inode_str[inode_length++] = (temp_inode % 10) + '0';
        temp_inode /= 10;
    }
}
for (int i = inode_length - 1; i >= 0; i--)
{
    write(1, &inode_str[i], 1);
}
write(1, "\n", 1);
```

Figura 19 - informa: Inode do ficheiro

Declaração de variáveis:

- *inode_str: Array* de caracteres para guardar em *string* do *inode*.
- *inode_length*: Inteiro para guardar o comprimento da *string*.
- *temp_inode*: Variável temporária para guardar o valor do *inode*.

Valida se o *inode* é zero e em caso afirmativo, guarda '0' no *inode_str* e define *inode_length* como '1'. A seguir, por forma a converter o número do *inode* para *string* divide *temp_inode* repetidamente por 10 para obter cada dígito e guardar em *inode_str*.

Posteriormente os caracteres da *string* do *inode* são escritos em ordem contrária (devido ao facto de terem sido guardados em ordem contrária durante a conversão).

- Proprietário do ficheiro:

```c
struct passwd *pw = getpwuid(file_info.st_uid);
if (pw != NULL)
{
    const char *owner = "Proprietário do ficheiro: ";
    write(1, owner, 28);
    write(1, pw->pw_name, strlen(pw->pw_name));
    write(1, "\n", 1);
} else {
    char uid_str[20];
    int uid_length = 0;
    int temp_uid = file_info.st_uid;
    if (temp_uid == 0)
    {
        uid_str[0] = '0';
        uid_length = 1;
    } else
    {
        while (temp_uid != 0)
        {
            uid_str[uid_length++] = (temp_uid % 10) + '0';
            temp_uid /= 10;
        }
    }
    for (int i = uid_length - 1; i >= 0; i--)
    {
        write(1, &uid_str[i], 1);
    }
    write(1, "\n", 1);
}
```

Figura 20 - informa: Proprietário do ficheiro

Para identificar o utilizador a partir do UID, utilizamos a função *getpwuid*. Caso seja executada com sucesso, retorna uma estrutura *passwd* onde contém as informações do utilizador.

Na verificação e impressão do nome do Proprietário, caso *pw* não seja nulo, o nome do proprietário é alcançado e impresso. Nesse sentido, escreve a mensagem "Proprietário do ficheiro: " e em seguida o nome do Proprietário.

Para o tratamento de erros e impressão do UID, caso *getpwuid* falhe (retorna *NULL*), o UID é convertido para string e impresso. Se o UID é zero, a string é "0".

Caso contrário, o UID é convertido caractere a caractere, que irá guardar cada dígito na *string uid_str*.

Após isto, a *string* do UID é impressa em ordem contrária, porque os dígitos foram guardados em ordem contrária durante a conversão.

- Data de criação, última leitura e modificação do ficheiro:

```
// Imprime a data de criação do ficheiro
const char *creation = "Data da criação do ficheiro: ";
write(1, creation, 32);
write(1, ctime(&file_info.st_ctime), 24);

// Imprime a data da última leitura do ficheiro
const char *access = "\nData da última leitura do ficheiro: ";
write(1, access, 39);
write(1, ctime(&file_info.st_atime), 24);

// Imprime a data da última modificação do ficheiro
const char *modification = "\nData da última modificação do ficheiro: ";
write(1, modification, 45);
write(1, ctime(&file_info.st_mtime), 24);
write(1, "\n", 1);
```

Figura 21 - informa: Data de criação, última leitura e última modificação do ficheiro

O código descrito na figura 21, imprime as datas de criação, última leitura e última modificação do ficheiro, utilizando as funções *ctime* para formatar os valores de tempo que se encontram na estrutura *stat*. Com isto, fornece ao utilizador uma visão completa sobre a história do ficheiro.

g) Implementação do comando 'lista'

Este comando deve apresentar uma lista de todas as pastas e ficheiros existentes na diretoria indicada ou na diretoria atual se não especificada. Adicionalmente, deve distinguir ficheiros simples de diretorias através de uma indicação textual.

- Validação dos argumentos:

```
const char *path;   // Caminho do diretório que será listado

// Verifica se o número de argumentos é correto
if (argc == 2)
{
    path = argv[1];  // Se fornecido um caminho como argumento, usa esse caminho
} else if (argc == 1)
{
    path = ".";      // Se nenhum caminho for fornecido, usa o diretório atual
} else {
    // Se o número de argumentos for inválido, devolve uma mensagem para passar os argumentos corretos
    write(2, "Erro: Digite os argumentos: ", 29);
    write(2, argv[0], strlen(argv[0]));
    write(2, "\n", 1);
    return 1;
}
```

Figura 22 - lista: Validação de argumentos

Declaração da variável path para guardar o caminho do diretório que será listado.

O código descrito na figura 22 verifica o número de argumentos passados na linha de comando. Se existirem dois argumentos, o segundo é atribuído como o caminho do diretório a ser listado. Se existir apenas um argumento, significa que nenhum caminho foi fornecido e é utilizado diretório atual.

Caso o número de argumentos seja diferente de 1 ou 2, é exibida uma mensagem de erro, para indicar ao utilizador que tem de fornecer os argumentos corretos.

- Validação da diretoria:

```
DIR *dir = opendir(path);
if (!dir)
{
    // Se ocorrer um erro ao abrir o diretório, imprime uma mensagem de erro
    write(2, "Erro ao abrir diretoria\n", 25);
    return 1;
}
```

Figura 23 - lista: Validação da diretoria

É utilizada a função *opendir* para abrir o diretório indicado no caminho (*path*). Caso o diretório não seja aberto com sucesso, é exibida uma mensagem de erro.

- Leitura e lista os ficheiros da diretoria:

```c
struct dirent *entry;  // Estrutura para guardar informações sobre os itens do diretório
while ((entry = readdir(dir)))
{
    // Ignora as entradas '.' e '..'
    if (entry->d_name[0] == '.' && (entry->d_name[1] == '\0' || entry->d_name[1] == '.'))
        continue;

    char full_path[PATH_MAX_LEN];    // Caminho completo para o item do diretório
    size_t path_len = strlen(path);  // Comprimento do caminho do diretório
    size_t name_len = strlen(entry->d_name);  // Comprimento do nome do item do diretório
    if (path_len + name_len + 2 > PATH_MAX_LEN)
    {
        // Se o caminho for muito extenso, imprime uma mensagem de erro
        write(2, "Caminho muito extenso\n", 20);
        closedir(dir);
        return 1;
    }
    strcpy(full_path, path);               // Copia o caminho do diretório para o caminho completo
    full_path[path_len] = '/';             // Insere uma barra no final do caminho
    strcpy(full_path + path_len + 1, entry->d_name);  // Insere o nome do item do diretório no final do caminho
    full_path[path_len + name_len + 1] = '\0';       // Insere um terminador de string no final do caminho completo

    struct stat file_stat;  // Estrutura para guardar informações sobre o item do diretório
    if (lstat(full_path, &file_stat) == -1) return 1;  // Determina as informações do item do diretório

    // Determina o tipo do item (diretório ou ficheiro) e exibe-o juntamente com o nome
    const char *type = S_ISDIR(file_stat.st_mode) ? "[diretoria]" : "[ficheiro]";
    write(1, type, strlen(type));
    write(1, " ", 1);
    write(1, entry->d_name, strlen(entry->d_name));
    write(1, S_ISDIR(file_stat.st_mode) ? "/" : "", S_ISDIR(file_stat.st_mode) ? 1 : 0);
    write(1, "\n", 1);
}

closedir(dir);
```

Figura 24 - lista: Informações relativas à diretoria

É utilizada a estrutura *dirent* para ler cada item do diretório e ignora as entradas '.' e '..' no diretório.

Após isto, constrói o caminho completo para cada um dos itens do diretório e valida se o comprimento do caminho não excede o limite.

Para determinar as informações sobre o item do diretório, utiliza *lstat*.

Assim que obtém o tipo do item (diretório ou ficheiro), exibe esta informação com o nome do item.

Após a listagem da informação, utiliza a função *closedir* para fechar o diretório.

Parte 2: Interpretador de linha de comandos

O programa implementa um interpretador de comandos que lê comandos do utilizador e executa várias operações.

- Leitura e análise de comandos:

```c
char comando[MAX_LENGTH];
char *args[MAX_LENGTH / 2];
int status;
char prompt[] = "% ";

while (1)
{
    // Exibe a linha de comandos do interpretador
    write(1, prompt, strlen(prompt));

    // Lê o comando do utilizador
    if (fgets(comando, MAX_LENGTH, stdin) == NULL)
    {
        write(2, "Erro na leitura do comando\n", 28);
        continue;
    }

    // Remove o caractere de nova linha
    comando[strcspn(comando, "\n")] = 0;

    // Verifica se o comando está vazio
    if (comando[0] == '\0')
    {
        continue; // Ignora e volta ao prompt
    }

    // Verifica se o comando é "termina"
    if (strcmp(comando, "termina") == 0)
    {
        break;
    }
```

Figura 25 - Interpretador: Leitura e análise dos comandos

O interpretador faz a leitura dos comandos do utilizador através da função *fgets* e elimina o caractere de nova linha no final da *string* de comando.

Efetua a validação de comandos vazios, sendo que os ignora por forma a que o utilizador escreva um comando específico.

- Leitura e análise de comandos:

```
if (strcmp(comando, "help") == 0)
{
    write(1, "Comandos disponíveis:\n", 24);
    write(1, "- mostra\n", 10);
    write(1, "- copia\n", 9);
    write(1, "- acrescenta\n", 14);
    write(1, "- conta\n", 9);
    write(1, "- apaga\n", 9);
    write(1, "- informa\n", 11);
    write(1, "- lista\n", 9);
    write(1, "- termina\n", 11);
    continue;
}
```

Figura 26 - Interpretador: Comando 'help'

Através do código da figura 26, foi implementado um comando interno '*help*' por forma a ajudar e a indicar ao utilizador a lista de todos os comandos disponíveis para executar.

- Validação de comandos:

```
// Divide o comando em argumentos
int i = 0;
args[i] = strtok(comando, " ");
while (args[i] != NULL)
{
    args[++i] = strtok(NULL, " ");
}

// Verifica se o comando é reconhecido
if (strcmp(args[0], "mostra") != 0 &&
    strcmp(args[0], "copia") != 0 &&
    strcmp(args[0], "acrescenta") != 0 &&
    strcmp(args[0], "conta") != 0 &&
    strcmp(args[0], "apaga") != 0 &&
    strcmp(args[0], "informa") != 0 &&
    strcmp(args[0], "lista") != 0)
{
    write(2, "Comando não reconhecido\n", 26);
    write(1, "\nDigite 'help' para verificar comandos disponíveis\n", 53);
    continue;
}
```

Figura 27 - Interpretador: Validação de comandos

O código descrito na figura 27, divide a linha de comando inserida pelo utilizador em argumentos utilizando *strtok*. A seguir, valida se o comando inserido existe ao comparar com uma lista definida de comandos válidos (mostra, copia, acrescenta, conta, apaga, informa, lista). Se o comando não existir na lista, exibe uma mensagem de erro a informar que o comando não é reconhecido e sugere ao utilizador para escrever o comando *help* que lista os comandos disponíveis.

- Validação de comandos:

```
    // Cria um novo processo
    pid_t pid = fork();
    if (pid == -1)
    {
        write(2, "Erro na criação de um novo processo\n", 39);
    }
    else if (pid == 0)
    {
        // Processo filho: executa o comando
        char path[MAX_LENGTH];
        sprintf(path, "./%s", args[0]); // Assume que o comando é um ficheiro na mesma diretoria
        if (execv(path, args) == -1)
        {
            write(2, "Erro na execução do comando\n", 31);
            exit(EXIT_FAILURE);
        }
    }
    else
    {
        // Processo pai: espera que o processo filho termine
        waitpid(pid, &status, 0);
        if (WIFEXITED(status))
        {
            char output[MAX_LENGTH];
            sprintf(output, "Terminou o comando %s com código %d\n", args[0], WEXITSTATUS(status));
            write(1, output, strlen(output));
        }
    }
}
```

Figura 28 - Interpretador: Criação e execução do processo

O código descrito na figura 28, cria um novo processo para executar comandos inseridos pelo utilizador.

A função *fork* é utilizada para criar um processo filho e caso a criação do processo falhe, exibe uma mensagem de erro.

No processo filho, o comando escolhido é executado através do *execv*, que consiste na substituição do processo filho pelo novo programa.

O caminho do comando é assumido que se encontra no diretório atual.

Caso a execução do comando falhe, exibe uma mensagem de erro e faz com que o processo filho termine com um código de erro.

No processo pai, *waitpid* é utilizado para esperar que o processo filho termine. Após a conclusão do comando, o código de termo do processo filho é verificado e exibe uma mensagem a informar se o comando foi concluído com sucesso e o respetivo código de retorno.

Este mecanismo garante a execução dos comandos de forma isolada em processos filhos, permitindo ao processo pai continuar a receber e processar novos comandos após a conclusão do comando atual.

Parte 3 A: Gestão de Sistemas de Ficheiros

a) Adicionar disco e criar partição

Num servidor virtual, adicione um disco novo com o tamanho de 10GB (espaço alocado dinamicamente) e crie uma partição.

O hypervisor utilizado é o “VMware Workstation”.

- Para adicionar novo disco, basta clicar com botão direito do rato sobre um servidor virtual existente e escolher a opção “Settings”, no meu caso o servidor tem o nome “VM-IPCA”:

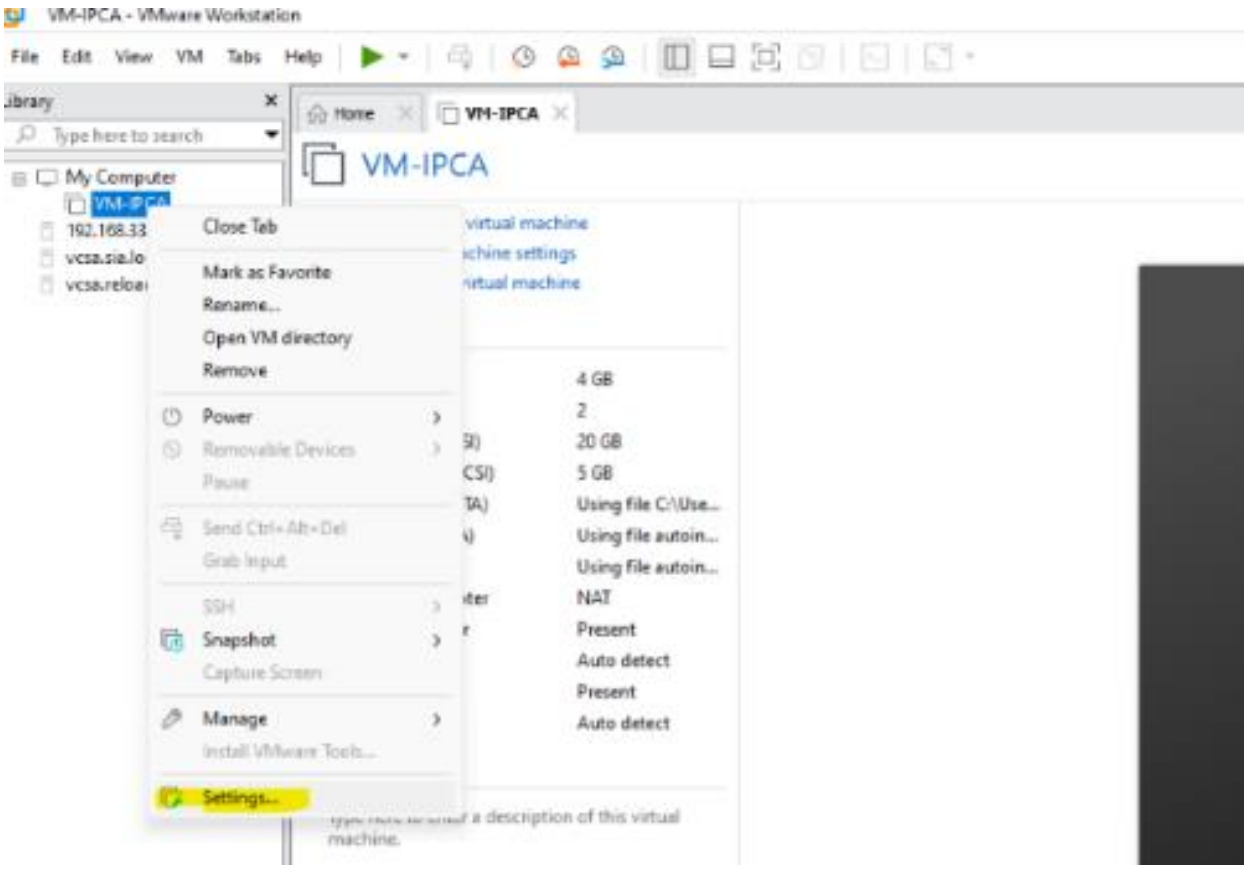

Figura 29 - Adicionar novo disco.

- No menu “Virtual Machine Settings” escolher a opção “Add”:

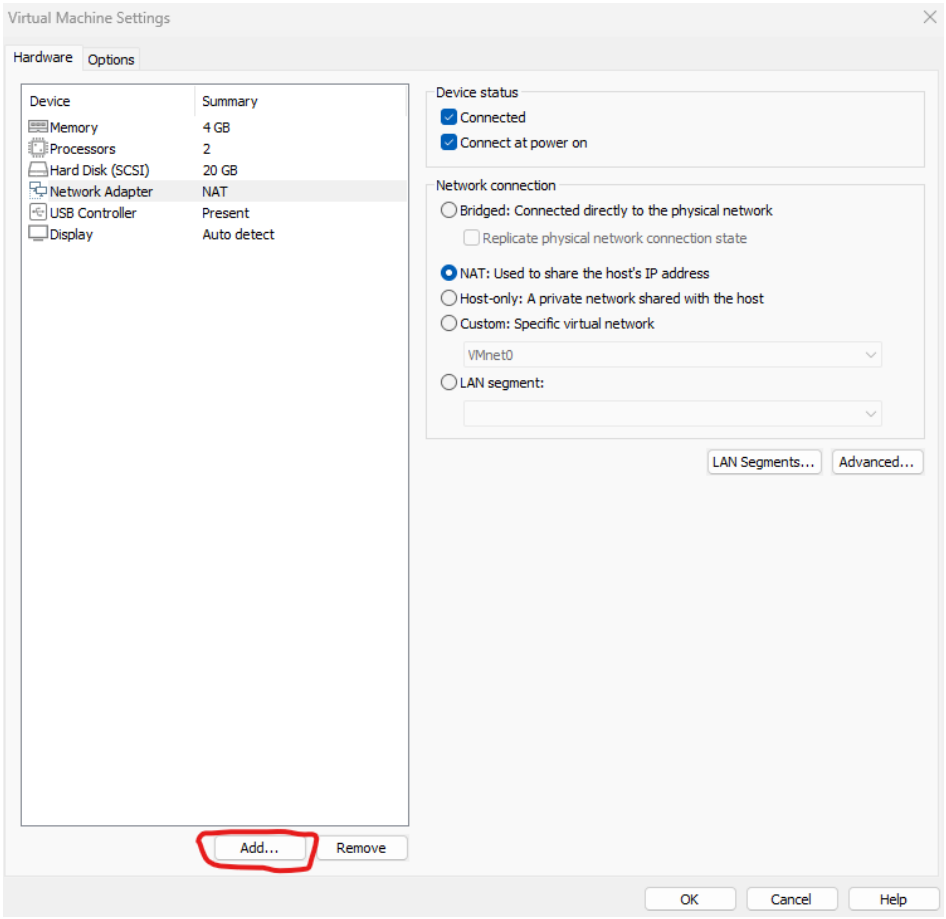

Figura 30 – Menu Virtual Machine Settings.

- Escolher hardware do tipo “Hard Disk”:

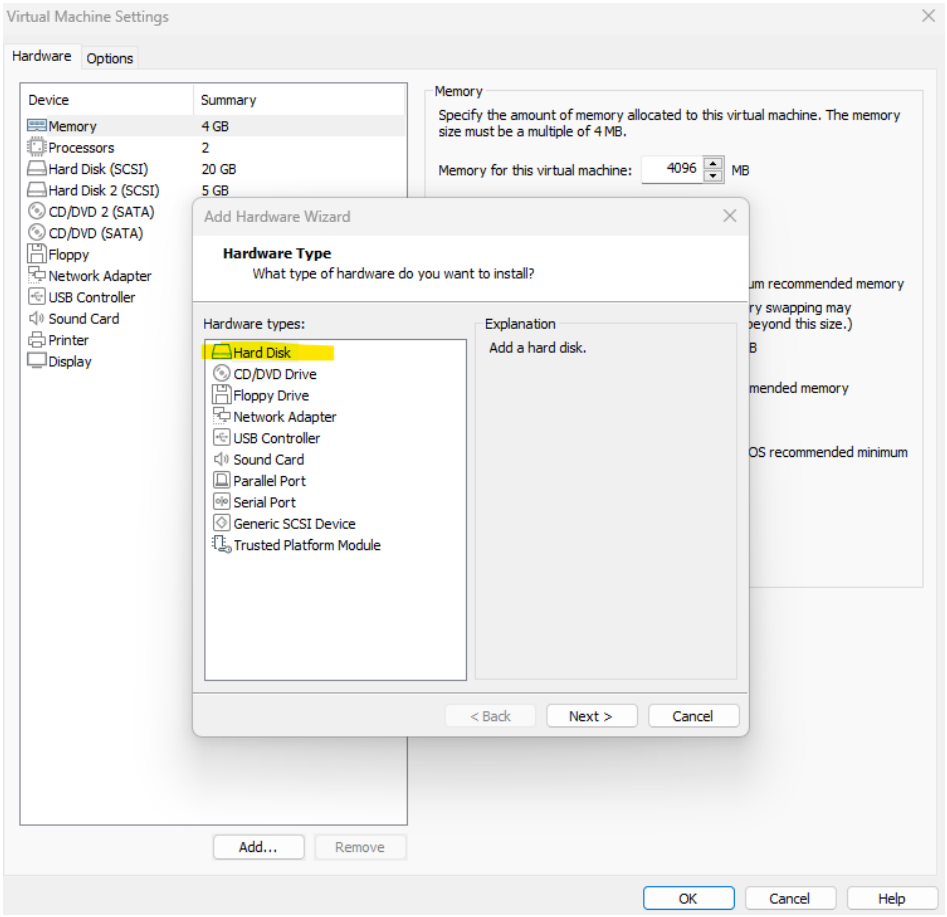

Figura 31 - Add Hardware Wizard (Hardware Type).

- No menu seguinte, surge opção de escolher o “Virtual disk type”, neste caso podemos optar pelo recomendado pelo hypervisor:

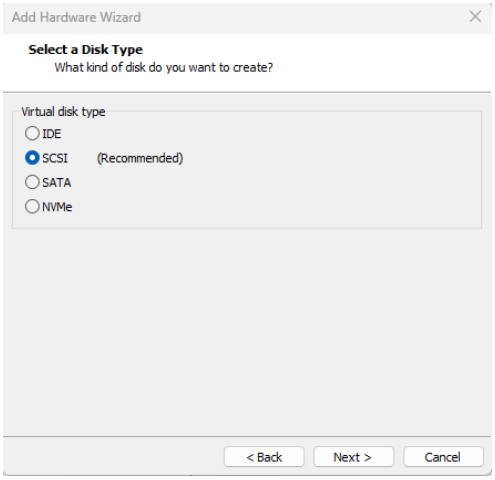

Figura 32 - Add Hardware Wizard (Select a Disk Type).

- No menu seguinte, escolhemos a opção “Create a new virtual disk”:

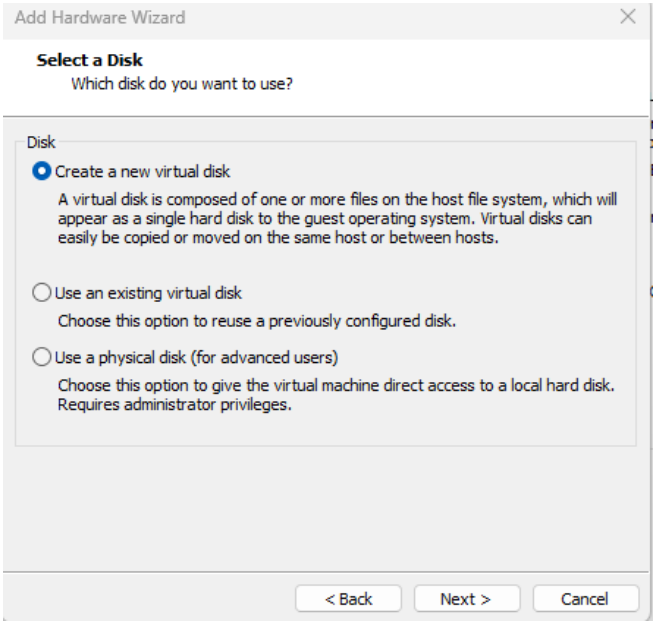

Figura 33 - Add Hardware Wizard (Select a Disk).

- No próximo menu, especificamos o tamanho indicado pelo enunciado, 10GB, e deixamos por marcar a opção “Allocate all disk space now” para que desta forma o espaço ocupado no hypervisor seja o mínimo possível e à medida que seja necessário irá ocupar mais, nunca ultrapassando o valor máximo do tamanho definido. Se a opção fosse marcada, o tamanho máximo do disco ficaria logo todo ocupado.

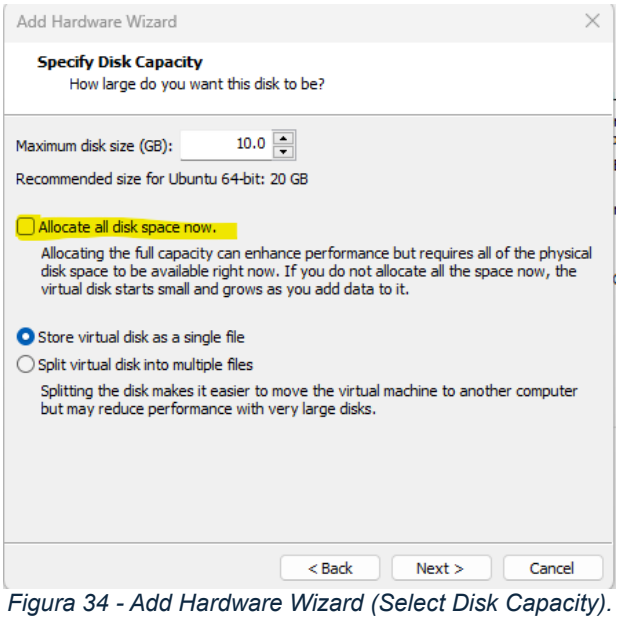

Figura 34 - Add Hardware Wizard (Select Disk Capacity).

- De seguinte tem de ser especificado o nome e a localização do novo disco e damos como concluída a criação.

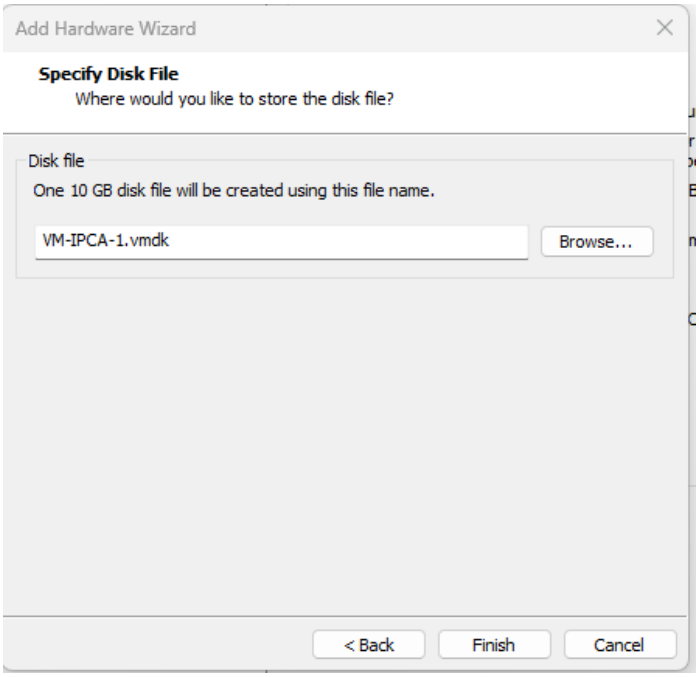

Figura 35 - Add Hardware Wizard (Select Disk File).

- Assim, já é possível ver o novo disco no hardware da VM (virtual machine):

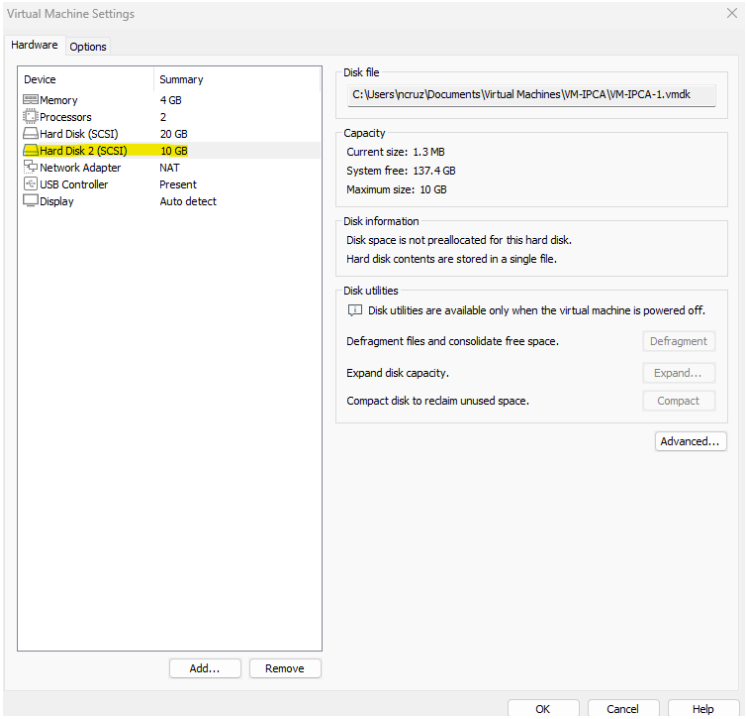

Figura 36 - Discos Virtual Machine

- Confirmamos na pasta de destino, que o disco criado (para alocar dinamicamente 10GB), apenas está a consumir 1,31MB:

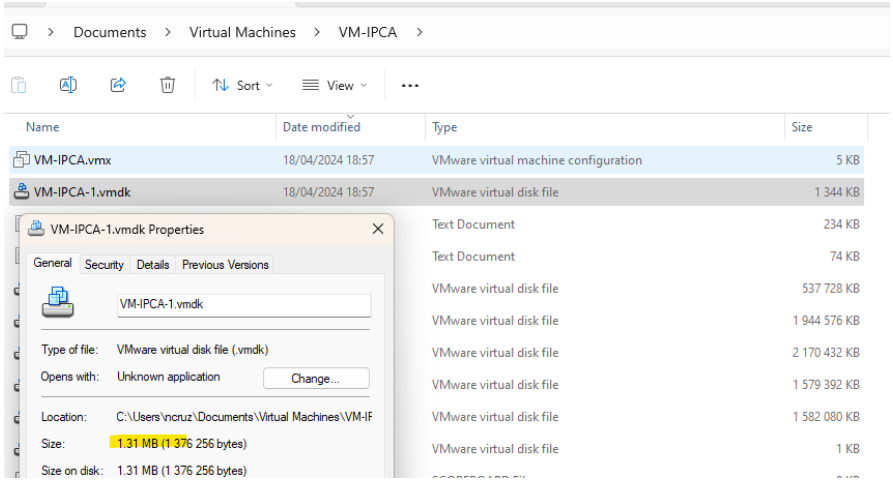

Figura 37 - Disco criado.

- No sistema operativo “Ubuntu”, no terminal, usando o comando “sudo fdisk -l” podemos verificar os 2 discos existentes:

```
Disk /dev/sda: 20 GiB, 21474836480 bytes, 41943040 sectors
Disk model: VMware Virtual S
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 79F612B9-C929-4A90-BA4F-A394894DA14D

Device       Start      End  Sectors  Size Type
/dev/sda1     2048     4095     2048    1M BIOS boot
/dev/sda2     4096  1054719  1050624  513M EFI System
/dev/sda3  1054720 41940991 40886272 19,5G Linux filesystem


Disk /dev/sdb: 10 GiB, 10737418240 bytes, 20971520 sectors
Disk model: VMware Virtual S
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

Figura 38 - Comando "fdisk -l".

- Para criar a partição que necessitamos, usamos o comando “fdisk“ no novo disco adicionado:

```
ncruz@VM-IPCA:~$ sudo fdisk /dev/sdb
[sudo] password for ncruz:

Welcome to fdisk (util-linux 2.37.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0xba496f1d.
```

Figura 39 - Comando "fdisk".

- De seguida usamos os seguintes comandos para criar a partição:
  - "n" (n de new, para criar a partição)
  - "p" (p de primary, para criar uma partição primária)
  - "1" (1 o número da partição a criar)
  - ENTER (para assumir o início da partição automaticamente)
  - ENTER (para assumir o fim da partição automaticamente)
  - "w" (w de write, para guardar as alterações na tabela de partições)

```
Command (m for help): n
Partition type
   p   primary (0 primary, 0 extended, 4 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-20971519, default 2048):
Last sector, +/-sectors or +/-size{K,M,G,T,P} (2048-20971519, default 20971519):

Created a new partition 1 of type 'Linux' and of size 10 GiB.

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
```

Figura 40 - Comandos para criar partição.

- Usando o comando "sudo fidsik -l" podemos ver a nova partição criada:

```
Disk /dev/sdb: 10 GiB, 10737418240 bytes, 20971520 sectors
Disk model: VMware Virtual S
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xba496f1d

Device     Boot Start      End  Sectors Size Id Type
/dev/sdb1        2048 20971519 20969472  10G 83 Linux
```

Figura 41 - Comando para ver a nova partição

b) Criar volume e adicionar dois volumes lógicos (5GB cada)

No disco virtual criado na alínea a), deve criar um volume, que ocupe o espaço todo, e dentro desse volume, deve adicionar dois volumes lógicos, cada um com o tamanho de 5GB.

- Antes da criação de volumes lógicos, temos de criar um physical volume e um volume group, para tal, usamos os seguintes comandos:
  - Physical Volume na partição /dev/sdb1: "sudo pvcreate /dev/sdb1"
  - Volume Group usando a partição anterior: "sudo vgcreate vgparte3a /dev/sdb1"
  - Podemos verificar a existência dos volumes físicos: "sudo pvs"
  - Verificamos também a existência do grupo de volumes: "sudo vgs"

```
ncruz@VM-IPCA:~$ sudo pvcreate /dev/sdb1
[sudo] password for ncruz:
  Physical volume "/dev/sdb1" successfully created.
ncruz@VM-IPCA:~$ sudo vgcreate vgparte3a /dev/sdb1
  Volume group "vgparte3a" successfully created
ncruz@VM-IPCA:~$ sudo pvs
  PV         VG        Fmt  Attr PSize   PFree
  /dev/sdb1  vgparte3a lvm2 a--  <10,00g <10,00g
ncruz@VM-IPCA:~$ sudo vgs
  VG        #PV #LV #SN Attr   VSize   VFree
  vgparte3a   1   0   0 wz--n- <10,00g <10,00g
ncruz@VM-IPCA:~$
```

Figura 42 - Criação de um physical volume.

- No volume group criado anteriormente, "vgparte3a", vamos criar dois logical volume, para tal usamos os seguintes comandos:
- Criar o primeiro logical volume: sudo lvcreate -L 5G -n lvparte3a1 vgparte3a
- Criar o segundo logical volume: sudo lvcreate -L 5G -n lvparte3a2 vgparte3a
- Verificar a criação do novo Logical Volume: “sudo lvs”

```
ncruz@VM-IPCA:~$ sudo lvcreate -L 5G -n lvparte3a1 vgparte3a
  Logical volume "lvparte3a1" created.
ncruz@VM-IPCA:~$ sudo lvcreate -L 5G -n lvparte3a2 vgparte3a
  Volume group "vgparte3a" has insufficient free space (1279 extents): 1280 required.
ncruz@VM-IPCA:~$ lvs
  WARNING: Running as a non-root user. Functionality may be unavailable.
  /run/lock/lvm/P_global:aux: open failed: Permission denied
ncruz@VM-IPCA:~$ sudo lvs
  LV         VG        Attr       LSize Pool Origin Data%  Meta%  Move Log Cpy%Sync Convert
  lvparte3a1 vgparte3a -wi-a----- 5,00g
ncruz@VM-IPCA:~$
```

Figura 43 - Criação de dois logical volume.

- Conforme podemos verificar no print acima, não foi possível criar o segundo logical volume com 5GB. Analisando o volume group através do comando “vgdisplay” podemos verificar o seguinte:

```
ncruz@VM-IPCA:~$ sudo vgdisplay
  --- Volume group ---
  VG Name               vgparte3a
  System ID
  Format                lvm2
  Metadata Areas        1
  Metadata Sequence No  2
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                1
  Open LV               0
  Max PV                0
  Cur PV                1
  Act PV                1
  VG Size               <10,00 GiB
  PE Size               4,00 MiB
  Total PE              2559
  Alloc PE / Size       1280 / 5,00 GiB
  Free  PE / Size       1279 / <5,00 GiB
  VG UUID               kLtBF8-VFml-KQHO-3XS6-zlLA-k4VE-ZDK241
```

Figura 44 - Comando “vgdisplay”.

Para encontramos o tamanho máximo que podemos criar um novo Logical Volume associado a este Volume Group, precisamos considerar o espaço livre disponível no Volume Group.

Podemos ver no campo "Free PE / Size" a indicação que há 1279 Physical Extents (PE) livres, correspondendo a menos de 5,00 GiB.

Os Physical Extents (PEs) são partes básicas de armazenamento num Volume Group (VG) quando se utiliza o LVM (Logical Volume Manager).

Cada PE é uma parcela continua do espaço disponível no VG. Quando criamos Logical Volumes (LVs) dentro do VG, alocamos espaço em termos de PEs, não em bytes ou blocos físicos.

Os PEs são uma camada de abstração entre os dispositivos físicos (como discos rígidos) e os LVs. Isto facilita a gestão do armazenamento, permitindo adicionar, remover ou ajustar LVs sem preocupações sobre os detalhes dos discos físicos.

Assim, o tamanho máximo que podemos criar para um novo logical volume depende do tamanho desses PEs livres e do tamanho de cada PE, que é especificado pelo campo "PE Size".

Para calcular o tamanho máximo do novo logical, multiplicamos o número de PEs livres pelo tamanho de cada PE:

- 1279 PEs livres * 4,00 MiB/PE = 5116 MiB

Então, o tamanho máximo que podemos criar para um novo logical volume neste volume group é de aproximadamente 5116 MiB, ou seja, cerca de 4,99 GiB.

- Vamos então criar o segundo logical volume com 4.99GiB, com o comando:
- “sudo lvcreate -L 4.99G -n lvparte3a2 vgparte3a”

```
ncruz@VM-IPCA:~$ sudo lvcreate -L 4.99G -n lvparte3a2 vgparte3a
  Rounding up size to full physical extent 4,99 GiB
  Logical volume "lvparte3a2" created.
ncruz@VM-IPCA:~$ sudo vgdisplay
  --- Volume group ---
  VG Name               vgparte3a
  System ID
  Format                lvm2
  Metadata Areas        1
  Metadata Sequence No  3
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                2
  Open LV               0
  Max PV                0
  Cur PV                1
  Act PV                1
  VG Size               <10,00 GiB
  PE Size               4,00 MiB
  Total PE              2559
  Alloc PE / Size       2558 / 9,99 GiB
  Free  PE / Size       1 / 4,00 MiB
  VG UUID               kLtBF8-VFml-KQHO-3XS6-zlLA-k4VE-ZDK241

ncruz@VM-IPCA:~$ sudo lvs
  LV         VG        Attr       LSize Pool Origin Data%  Meta%  Move Log Cpy%Sync Convert
  lvparte3a1 vgparte3a -wi-a----- 5,00g
  lvparte3a2 vgparte3a -wi-a----- 4,99g
```

Figura 45 - Criação de segundo logical volume.

c) Criar sistema de ficheiros ext4 e ext3

Nos volumes lógicos criados no passo b), crie um sistema de ficheiros ext4 em um deles e ext3 no outro.

Para criar um sistema de ficheiros ext4 e outro ext3, usamos os seguintes comandos:

- sudo mkfs.ext4 /dev/vgparte3a/lvparte3a1
- sudo mkfs.ext3 /dev/vgparte3a/lvparte3a2

```
ncruz@VM-IPCA:/$ sudo mkfs.ext4 /dev/vgparte3a/lvparte3a1
[sudo] password for ncruz:
mke2fs 1.46.5 (30-Dec-2021)
Creating filesystem with 1310720 4k blocks and 327680 inodes
Filesystem UUID: 8e53657d-7b33-4e82-a601-162b1cf930c2
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

ncruz@VM-IPCA:/$ sudo mkfs.ext3 /dev/vgparte3a/lvparte3a2
mke2fs 1.46.5 (30-Dec-2021)
Creating filesystem with 1308672 4k blocks and 327680 inodes
Filesystem UUID: e3b567b4-f4bd-41c4-bb72-2b102e5366c0
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done
```

Figura 46 - Criação de um sistema de ficheiros.

d) Montar cada sistema de ficheiro criado em diretorias

Monte cada um dos sistemas de ficheiros criados em c) nas diretorias /mnt/ext4 e /mnt/ext3, respetivamente, ficando persistente a reboots.

- As diretorias indicadas no enunciado não existem, por isso em primeiro lugar vamos usar o comando “mkdir” para criar as mesmas:

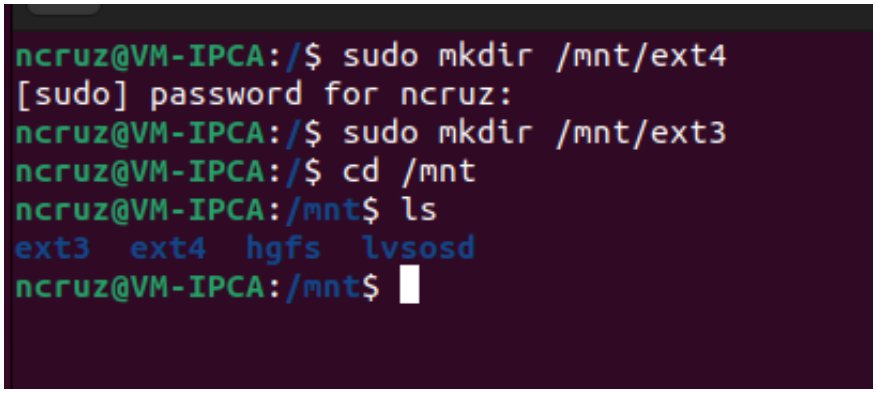

Figura 47 - Comando "mkdir"

- Para os diretórios serem persistentes, temos de adicionar os mesmos no ficheiro de configuração responsável pelos sistemas de ficheiros serem montados automaticamente durante o boot do sistema, o fstab:

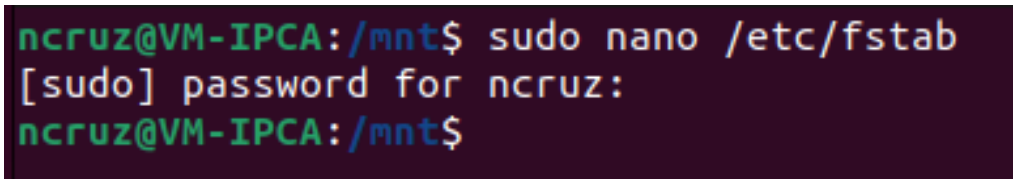

Figura 48 - Edição fstab

Figura 49 - View fstab

- De seguida é possível ver, usando o comando “df -h”, que os sistemas de ficheiros não estão montados ainda. Para montar de imediato usamos o comando “mount -a”. De seguida repetimos o comando “df -h”:

```
ncruz@VM-IPCA:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           388M  2,1M  386M   1% /run
/dev/sda3        20G   14G  5,1G  73% /
tmpfs           1,9G     0  1,9G   0% /dev/shm
tmpfs           5,0M     0  5,0M   0% /run/lock
/dev/sda2       512M  6,1M  506M   2% /boot/efi
tmpfs           388M  112K  387M   1% /run/user/1000
ncruz@VM-IPCA:~$ sudo mount -a
ncruz@VM-IPCA:~$ df -h
Filesystem                         Size  Used Avail Use% Mounted on
tmpfs                              388M  2,1M  386M   1% /run
/dev/sda3                           20G   14G  5,1G  73% /
tmpfs                              1,9G     0  1,9G   0% /dev/shm
tmpfs                              5,0M     0  5,0M   0% /run/lock
/dev/sda2                          512M  6,1M  506M   2% /boot/efi
tmpfs                              388M  112K  387M   1% /run/user/1000
/dev/mapper/vgparte3a-lvparte3a1   4,9G   24K  4,6G   1% /mnt/ext4
/dev/mapper/vgparte3a-lvparte3a2   4,9G   92K  4,6G   1% /mnt/ext3
```

Figura 50 - Comando "df -h" e "mount -a".

- Vamos de seguida reiniciar o sistema para confirmar que os sistemas de ficheiros são montados no arranque:

```
ncruz@VM-IPCA:~$ uptime
 18:13:47 up 4 min,  1 user,  load average: 0,29, 1,10, 0,59
ncruz@VM-IPCA:~$ sudo reboot
```

Figura 51 - Comando "uptime" e "reboot".

```
ncruz@VM-IPCA:~$ uptime
 18:15:13 up 0 min,  1 user,  load average: 6,17, 1,90, 0,66
ncruz@VM-IPCA:~$ df -h
Filesystem                         Size  Used Avail Use% Mounted on
tmpfs                              388M  2,0M  386M   1% /run
/dev/sda3                           20G   13G  5,2G  72% /
tmpfs                              1,9G     0  1,9G   0% /dev/shm
tmpfs                              5,0M     0  5,0M   0% /run/lock
/dev/sda2                          512M  6,1M  506M   2% /boot/efi
/dev/mapper/vgparte3a-lvparte3a1   4,9G   24K  4,6G   1% /mnt/ext4
/dev/mapper/vgparte3a-lvparte3a2   4,9G   92K  4,6G   1% /mnt/ext3
tmpfs                              388M  104K  387M   1% /run/user/1000
ncruz@VM-IPCA:~$
```

Figura 52 - Comando "uptime" e "df -h".

e) Criar ficheiro com números de alunos e configurar as permissões

Dentro da diretoria /mnt/ext4, crie um ficheiro com o nome composto pelo grupo dos números de alunos que constituem o trabalho, e a extensão .txt (exemplo: 22222-22233-23333-24003.txt). Esse ficheiro deverá ter, apenas, permissões de escrita e leitura para o dono (que será o utilizador que está a usar o sistema sem ser root), o grupo não deve ter qualquer permissão neste ficheiro, e todos os outros devem ter permissão de leitura.

- Vamos criar o ficheiro indicado, com o comando "touch", na pasta indicava e verificar as permissões após a criação:

```
ncruz@VM-IPCA:~$ cd /mnt/ext4
ncruz@VM-IPCA:/mnt/ext4$ ls
lost+found
```

Figura 53 - Comando "touch".

- Podemos verificar que o utilizador ncruz não tem permissão para criar ficheiros na pasta indicada no enunciado.

```
ncruz@VM-IPCA:/mnt/ext4$ touch 39-29576-30516-30517-30518.txt
touch: cannot touch '39-29576-30516-30517-30518.txt': Permission denied
```

Figura 54 - Falha permissão criar ficheiros.

- Com o comando "chown" vamos mudar o owner da pasta para o utilizador atual:

```
ncruz@VM-IPCA:/mnt/ext4$ sudo chown ncruz /mnt/ext4
```

Figura 55 - Comando "chown".

```
ncruz@VM-IPCA:/mnt$ ls -l
total 16
drwxr-xr-x 3 root  root 4096 abr 20 15:44 ext3
drwxr-xr-x 3 ncruz root 4096 abr 20 18:29 ext4
drwxr-xr-x 2 root  root 4096 fev 22 18:56 hgfs
drwxr-xr-x 2 root  root 4096 mar  7 19:17 lvsosd
```

Figura 56 - Comando "ls -l"

- Repetimos o processo de criação de ficheiro com o comando touch e verificamos as permissões iniciais:

```
ncruz@VM-IPCA:/mnt/ext4$ touch 39-29576-30516-30517-30518.txt
ncruz@VM-IPCA:/mnt/ext4$ ls -l
total 16
-rw-rw-r-- 1 ncruz ncruz     0 abr 20 18:31 39-29576-30516-30517-30518.txt
drwx------ 2 root  root  16384 abr 20 15:44 lost+found
```

Figura 57 - Comando "touch".

Atualmente o ficheiro está com permissão de leitura e escrita para o owner e para o grupo todos os outros têm apenas leitura do ficheiro.

Usando o comando “chmod 604 39-29576-30516-30517-30518.txt” vamos alterar as permissões para o pretendido pelo enunciado.

Permissão	Binário	Decimal
---	000	0
--x	001	1
-w-	010	2
-wx	011	3
r--	100	4
r-x	101	5
rw-	110	6
rwx	111	7

Figura 58 - Comparativo Permissões

O “6” corresponde à permissão de leitura e escrita (rw-) do utilizador owner do ficheiro, o número seguinte, 0, corresponde à permissão do grupo, que não terá nenhuma permissão (---) e o 4 indica a permissão de leitura de todos os outros (r--).

```
ncruz@VM-IPCA:/mnt/ext4$ chmod 604 39-29576-30516-30517-30518.txt
ncruz@VM-IPCA:/mnt/ext4$ ls -l
total 16
-rw----r-- 1 ncruz ncruz     0 abr 20 18:31 39-29576-30516-30517-30518.txt
drwx------ 2 root  root  16384 abr 20 15:44 lost+found
```

Figura 59 - Permissões finais

Parte 3 B: Análise aos Sistemas de Ficheiros

Com base no ficheiro fs.img (disponível no moodle), responda às seguintes questões:

a) Identificação do bloco de início dos ficheiros

Identificar o bloco de início dos ficheiros regulares presentes do sistema de ficheiros

Para termos detalhes do sistema de ficheiros, vamos user o comando “fsstat”.

Mais propriamente o comando: “fsstat -o 0 fs.img”:

- -o: Argumento que fornece ao fsstat informações sobre um volume específico.
- 0: Argumento que específica o número do volume, complementa o argumento “-o”.

```
ncruz@VM-IPCA:~/Documents/SistemasOperativos/TP$ fsstat -o 0 fs.img
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: FAT12

OEM Name: mkfs.fat
Volume ID: 0xaa724ea
Volume Label (Boot Sector): NO NAME
Volume Label (Root Directory):
File System Type Label: FAT12

Sectors before file system: 0

File System Layout (in sectors)
Total Range: 0 - 2047
* Reserved: 0 - 0
** Boot Sector: 0
* FAT 0: 1 - 2
* FAT 1: 3 - 4
* Data Area: 5 - 2047
** Root Directory: 5 - 36
** Cluster Area: 37 - 2044
** Non-clustered: 2045 - 2047

METADATA INFORMATION
--------------------------------------------
Range: 2 - 32694
Root Directory: 2

CONTENT INFORMATION
--------------------------------------------
Sector Size: 512
Cluster Size: 2048
Total Cluster Range: 2 - 503

FAT CONTENTS (in sectors)
--------------------------------------------
65-68 (4) -> EOF
```

Figura 60 - Bloco de início

Conforme podemos ver na imagem, o bloco de início é o bloco 0.

b) Indicar conteúdo do ficheiro “ipca.txt”

Indique o conteúdo do ficheiro com o nome ipca.txt

- Primeiro, vamos listar todos os ficheiros e diretórios na imagem usando o comando “fls”:

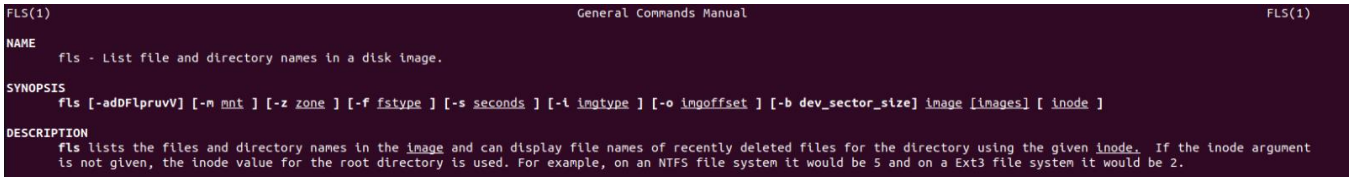

Figura 61 - Comando "fls"

- Juntamos os argumentos “-o” e “0” tal como na alínea a.

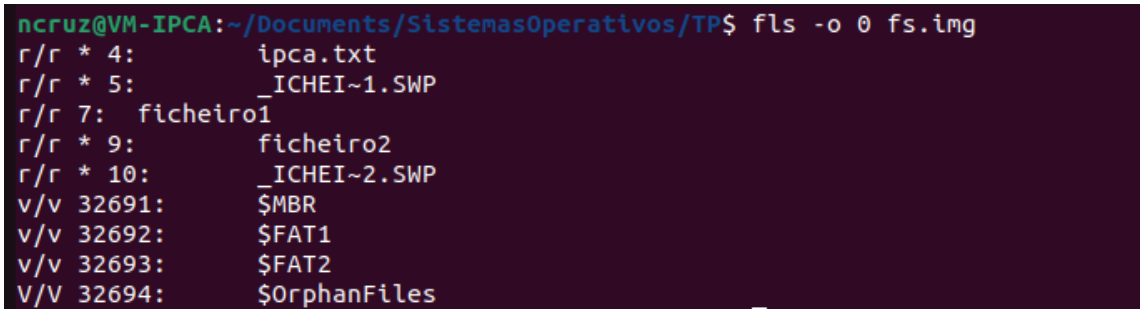

Figura 62 - Comando "fls" com "-o" "0"

- Pelo output, podemos verificar que o ficheiro ipca.txt está no bloco 4.
- Para ver o conteúdo do ficheiro, usamos o comando “icat -o 0 filesystem.fat.img 4”

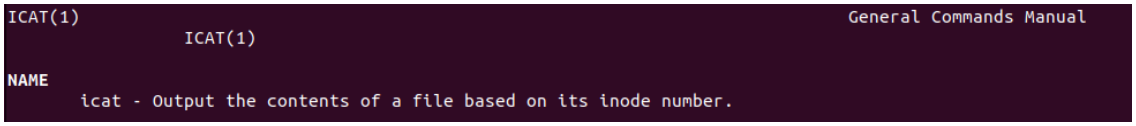

Figura 63 - Comando "icat -o".

- Os argumentos “-o” e “0” já os especificamos anteriormente. O argumento “4” refere-se ao bloco onde está o ficheiro ipca.txt.
    - Desta forma, identificamos o conteúdo do ficheiro ipca.txt que é “Este é o conteúdo do ficheiro ipca.txt”

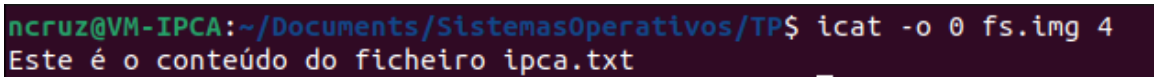

Figura 64 - Conteúdo ficheiro ipca.txt.

Conclusão

Ao longo deste projeto, foram exploradas várias questões relacionadas com a gestão de processos e ficheiros em sistemas operativos, implementação de comandos para manipulação de ficheiros e criação de um interpretador de linha de comandos.

Com este trabalho, foi possível cimentar o conhecimento adquirido na Unidade Curricular na utilização de chamadas ao sistema para interagir com o sistema operativo, tal como na implementação de funcionalidades complexas relacionadas com gestão de ficheiros e processos.

Houve uma explicação avançada na gestão de sistemas de ficheiros, incluindo a criação de discos virtuais, partições, volumes lógicos e sistemas de ficheiros.

Neste trabalho aplicamos os conhecimentos teóricos obtidos no decorrer das aulas e tivemos também possibilidade de aplicar algum do conhecimento adquirido no dia a dia da vida profissional.

Em suma, acreditamos que este relatório e o trabalho desenvolvido, sirvam como bases e recursos para aprofundar o conhecimento dos conceitos essenciais de sistemas operativos.