



ENGENHARIA DE SISTEMAS INFORMÁTICOS

UC: SISTEMAS OPERATIVOS 2024-2025

RELATÓRIO DE PROJETO

Nome	Nº de Aluno
Gonçalo Santos	27985
Rodrigo Cruz	27971

25 de maio de 2025

Conteúdo

1	Introdução	4
1.1	Contextualização do Projeto	4
2	Distribuição de Trabalho	5
3	Implementação de Comandos	6
3.1	Abordagem Geral	6
3.2	Comando "mostra ficheiro" (Alínea a)	7
3.3	Comando "copia ficheiro" (Alínea b)	9
3.4	Comando "acrescenta ficheiro" (Alínea c)	12
3.5	Comando "conta linhas" (Alínea d)	16
3.6	Comando "apaga ficheiro" (Alínea e)	18
3.7	Comando "informa ficheiro" (Alínea f)	20
3.8	Comando "lista [diretoria]" (Alínea g)	23
4	Interpretador de Linha de Comandos	26
4.1	Arquitetura do Interpretador	26
4.2	Estrutura do Código	26
4.3	Gestão de Processos	27
4.4	Tratamento de Erros	27
4.5	Testes do Interpretador	27
5	Gestão de Sistemas Ficheiros	29
5.1	Alínea a) – Adicionar Disco Virtual e Criar Partição	29
5.1.1	Definição de tamanho	31
5.1.2	Criação da Partição com fdisk	33
5.2	Alínea b) – Criação de Volume Físico e Volumes Lógicos	34
5.3	Alínea c) – Criação de Sistemas de Ficheiros	36
5.4	Alínea d) – Montagem nos diretórios	37
5.5	Alínea e) – Criação de Ficheiro com Permissões Específicas	38

6	Análise de Sistemas de Ficheiros	39
6.1	Alínea a) Montagem da Imagem	39
6.2	Alínea b) Conteúdo do ficheiro	39
7	Conclusão do Trabalho	41
8	Dificuldades Encontradas	42

Lista de Figuras

1	Teste comando mostra	8
2	Teste comando copia	11
3	Teste comando acrescenta	15
4	Teste comando conta	17
5	Teste comando apaga ficheiro	19
6	Teste comando informa	22
7	Teste comando lista	25
8	Teste ./interpretador	28
9	Teste termina	28
10	Teste Makefile	28
11	Definições da VM	29
12	Opção Virtual Disk Image	30
13	Opção Virtual Disk Image	31
14	Informações do Disco	32
15	Configurar a partição	33
16	Grupo para os Volumes	34
17	Criação de Volumes	34
18	Informação dos Volumes Lógicos	35
19	Sistema de ficheiro ext3	36
20	Sistema de ficheiro ext4	36
21	Ficheiro fstab	37
22	Montagem de Sistemas de Ficheiros	37
23	Criação de Ficheiro e permissões para o mesmo	38
24	Montagem da imagem fs.img	39
25	Listagem de diretórios de fs.img	39
26	Conteúdo do ficheiro ipca.txt	40

1 Introdução

1.1 Contextualização do Projeto

Este projeto consiste na implementação de um interpretador de linha de comandos desenvolvido na linguagem C. O principal objetivo é aprofundar o conhecimento sobre manipulação de ficheiros, diretórios e chamadas ao sistema em ambiente Unix/Linux. Para tal, foram desenvolvidos comandos personalizados que simulam funcionalidades comuns de sistemas operativos, permitindo a leitura, cópia, modificação e análise de ficheiros e diretórios. Além disso, o interpretador suporta também a execução de comandos nativos do sistema, oferecendo uma experiência semelhante a um terminal real. Este trabalho visa consolidar competências na programação em C e no funcionamento de sistemas operativos, com especial enfoque na interação com o sistema de ficheiros.

2 Distribuição de Trabalho

A tabela seguinte apresenta a distribuição das tarefas realizadas ao longo do projeto, indicando para cada alínea o(s) responsável(eis) e uma breve descrição do trabalho desenvolvido.

Alínea	Responsável(eis)	Descrição
1.a	Gonçalo	Implementação do comando "mostra"
1.b	Rodrigo	Implementação do comando "copia"
1.c	Gonçalo	Implementação do comando "acrescenta"
1.d	Rodrigo	Implementação do comando "conta"
1.e	Gonçalo	Implementação do comando "apaga"
1.f	Rodrigo	Implementação do comando "informa"
1.g	Gonçalo	Implementação do comando "lista"
2	Gonçalo e Rodrigo	Implementação do interpretador
2.a	Gonçalo	Criação do Makefile
3.a	Rodrigo	Análise do sistema de ficheiros - identificação de blocos
3.b	Rodrigo	Análise do sistema de ficheiros - conteúdo do ipca.txt

Tabela 1: Distribuição de Responsabilidades e Descrições

3 Implementação de Comandos

3.1 Abordagem Geral

A implementação dos comandos de manipulação de ficheiros foi realizada exclusivamente com recurso a chamadas ao sistema (*system calls*), conforme estipulado no enunciado do projeto. Esta abordagem permitiu um contacto direto com as funcionalidades de baixo nível disponibilizadas pelo sistema operativo, reforçando os conhecimentos sobre o funcionamento interno da gestão de ficheiros em ambiente Unix/Linux.

As principais chamadas ao sistema utilizadas foram:

- `open()` – Para abrir ficheiros
- `read()` – Para ler dados dos ficheiros
- `write()` – Para escrever dados nos ficheiros
- `close()` – Para fechar descritores de ficheiros.
- `stat()` – Para obter informações sobre ficheiros
- `unlink()` – Para remover ficheiros
- `opendir()`, `readdir()`, `closedir()` – Para manipular diretórios

Estas chamadas permitiram a implementação de comandos personalizados com controlo total sobre a leitura, escrita e manipulação de ficheiros e diretórios, sem recorrer a funções de mais alto nível da biblioteca padrão `stdio.h`.

3.2 Comando "mostra ficheiro" (Alínea a)

Funcionalidade: Apresenta no ecrã todo o conteúdo do ficheiro especificado.

Implementação:

```
1 int mostra(const char *filename) {
2     int fd, n;
3     char buffer[4096];
4
5     fd = open(filename, O_RDONLY);
6     if (fd == -1) {
7         fprintf(stderr, "Erro: O ficheiro '%s' nao existe...\n",
8             filename);
9         return 1;
10    }
11
12    while ((n = read(fd, buffer, sizeof(buffer))) > 0) {
13        write(STDOUT_FILENO, buffer, n);
14    }
15
16    close(fd);
17    return 0;
18 }
```

Código 1: Função mostra()

Tratamento de Erros: Caso o ficheiro especificado não exista, é apresentada uma mensagem de erro no `stderr`, informando o utilizador da situação.

Testes Realizados:

- Teste com ficheiro existente: Funciona corretamente
- Teste com ficheiro inexistente: Apresenta mensagem de erro
- Teste com ficheiro vazio: Não apresenta nada (comportamento esperado)

Screenshot dos Testes:


```
% mostra ./out/texto.txt
Olá,
este é um ficheiro de teste para o interpretador.
Aqui vamos verificar se a leitura e escrita de ficheiros está a funcionar corretamente.

2024/2025

Ficheiro mostrado com sucesso.
Terminou comando mostra com código 0
% █
```

Figura 1: Teste comando mostra

3.3 Comando "copia ficheiro"(Alínea b)

Funcionalidade: Cria uma cópia exata do ficheiro especificado, adicionando o sufixo .copia ao nome original.

Implementação:

```
1 int copia(const char *filename) {
2     int fd_src, fd_dest, n;
3     char buffer[4096];
4     char dest_filename[1024];
5
6     // Verificar se o ficheiro de origem existe
7     if (access(filename, F_OK) != 0) {
8         fprintf(stderr, "Erro: O ficheiro '%s' nao existe.\n",
9 filename);
10        return 1;
11    }
12
13    // Construir nome do ficheiro de destino
14    snprintf(dest_filename, sizeof(dest_filename), "%s.copia",
15 filename);
16
17    // Abrir o ficheiro de origem
18    fd_src = open(filename, O_RDONLY);
19    if (fd_src == -1) {
20        fprintf(stderr, "Erro: O ficheiro '%s' nao existe ou nao
21 pode ser aberto.\n", filename);
22        return 1;
23    }
24
25    // Criar ficheiro de destino
26    fd_dest = open(dest_filename, O_WRONLY | O_CREAT | O_TRUNC,
27 0644);
28    if (fd_dest == -1) {
```

```
25     fprintf(stderr, "Erro: Nao foi possivel criar o ficheiro '%s'.\n", dest_filename);
26     close(fd_src);
27     return 1;
28 }
29
30 // Copiar conteudo
31 while ((n = read(fd_src, buffer, sizeof(buffer))) > 0) {
32     write(fd_dest, buffer, n);
33 }
34
35 // Fechar ficheiros
36 close(fd_src);
37 close(fd_dest);
38
39 printf("\n\nFicheiro copiado com sucesso para '%s'.\n", dest_filename);
40 return 0;
41 }
```

Código 2: Implementacao do comando "copia"

Tratamento de Erros:

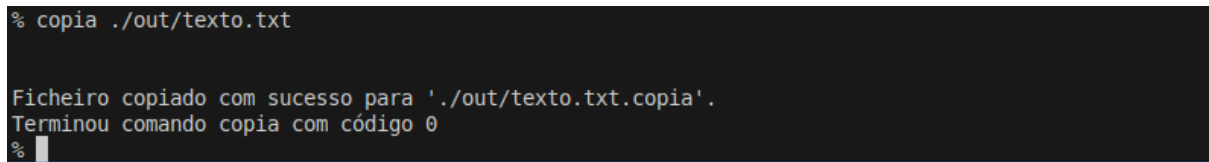
- Verifica se o ficheiro de origem existe com `access()`.
- Apresenta erro caso o ficheiro de origem não exista ou não possa ser aberto.
- Apresenta erro caso o ficheiro de destino não possa ser criado.
- Garante o fecho correto dos descritores de ficheiros mesmo em caso de erro.

Testes Realizados:

- **Ficheiro existente com conteúdo:** Criação correta de uma cópia com o mesmo conteúdo e com o nome `<original>.copia`.

- **Ficheiro inexistente:** Apresenta mensagem de erro: "Erro: O ficheiro '*nome*' não existe."
- **Ficheiro vazio:** Cria corretamente uma cópia vazia com o nome `<original>.copia`. O conteúdo está corretamente vazio, o que confirma que o ciclo de leitura termina de forma apropriada.
- **Ficheiro sem permissões de leitura:** Apresenta erro ao tentar abrir o ficheiro original. A mensagem é adequada: "Erro: O ficheiro '*nome*' não existe ou não pode ser aberto."

Screenshot dos Testes:



```
% copia ./out/texto.txt
Ficheiro copiado com sucesso para './out/texto.txt.copia'.
Terminou comando copia com código 0
% 
```

Figura 2: Teste comando copia

3.4 Comando "acrescenta ficheiro"(Alínea c)

Funcionalidade: Acrescenta o conteúdo do ficheiro de origem ao final do ficheiro de destino, sem alterar o conteúdo original do destino. Esta operação é útil para consolidar dados de vários ficheiros num único.

Implementação:

```
1 int acrescenta(const char \*origem, const char \*destino) {
2     int fd_src, fd_dest, n;
3     char buffer[4096];
4     struct stat stat_src, stat_dest;
5
6     if (access(origem, F_OK) != 0) {
7         fprintf(stderr, "Erro: O ficheiro de origem '%s' nao existe.\n",
8             origem);
9         return 1;
10    }
11
12    if (access(destino, F_OK) != 0) {
13        fprintf(stderr, "Erro: O ficheiro de destino '%s' nao existe.\n",
14            , destino);
15        return 1;
16    }
17
18    fd_src = open(origem, O_RDONLY);
19    if (fd_src == -1) {
20        fprintf(stderr, "Erro: Nao foi possivel abrir o ficheiro de
21            origem '%s'.\n", origem);
22        return 1;
23    }
24
25    fd_dest = open(destino, O_WRONLY | O_APPEND);
26    if (fd_dest == -1) {
27        fprintf(stderr, "Erro: Nao foi possivel abrir o ficheiro de
```

```
destino '%s'.\n", destino);
25     close(fd_src);
26     return 1;
27 }
28
29 if (fstat(fd_src, &stat_src) == -1 || fstat(fd_dest, &stat_dest) ==
    -1) {
30     fprintf(stderr, "Erro: Falha ao obter informacoes dos ficheiros
        .\n");
31     close(fd_src);
32     close(fd_dest);
33     return 1;
34 }
35
36 if (stat_src.st_ino == stat_dest.st_ino && stat_src.st_dev ==
    stat_dest.st_dev) {
37     fprintf(stderr, "Erro: Os ficheiros de origem e destino sao o
        mesmo. Operacao cancelada.\n");
38     close(fd_src);
39     close(fd_dest);
40     return 1;
41 }
42
43 while ((n = read(fd_src, buffer, sizeof(buffer))) > 0) {
44     if (write(fd_dest, buffer, n) != n) {
45         fprintf(stderr, "Erro: Falha ao escrever no ficheiro de
            destino.\n");
46         close(fd_src);
47         close(fd_dest);
48         return 1;
49     }
50 }
```

```
51
52 if (n < 0) {
53     fprintf(stderr, "Erro: Falha ao ler o ficheiro de origem.\n");
54 }
55
56 close(fd_src);
57 close(fd_dest);
58
59 printf("\n\nConteúdo de '%s' acrescentado com sucesso a '%s'.\n",
        origem, destino);
60 return 0;
61
62 }
```

Código 3: Implementacao do comando "acrescenta"

Tratamento de Erros:

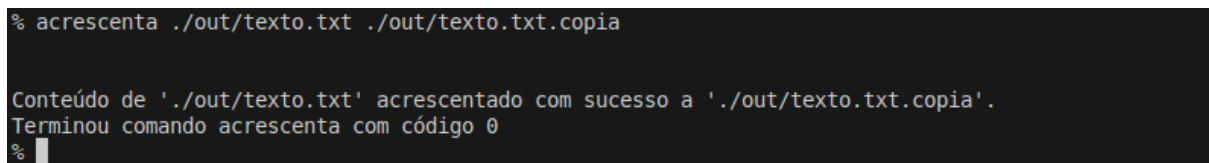
- Verifica se o ficheiro de origem e o ficheiro de destino existem com `access()`.
- Garante que os ficheiros de origem e destino não são o mesmo, comparando o `inode` e o `device`.
- Apresenta erro caso algum dos ficheiros não possa ser aberto.
- Verifica e trata erros na leitura e escrita dos ficheiros.
- Garante o fecho correto dos descritores de ficheiro mesmo em caso de erro.

Testes Realizados:

- **Origem com conteúdo, destino com conteúdo:** O conteúdo do ficheiro de origem foi corretamente acrescentado ao final do ficheiro de destino, sem apagar ou alterar o conteúdo já existente.
““
- **Ficheiro de origem inexistente:** Apresenta mensagem de erro: "Erro: O ficheiro de origem '*nome*' não existe."

- **Ficheiro de destino inexistente:** Apresenta mensagem de erro: "Erro: O ficheiro de destino '*nome*' não existe."
- **Ficheiro de origem igual ao ficheiro de destino:** Deteta corretamente esta situação e apresenta mensagem: "Erro: Os ficheiros de origem e destino são o mesmo. Operação cancelada."
- **Ficheiro de origem vazio:** O conteúdo do ficheiro de destino permanece inalterado, confirmando que o ciclo de leitura e escrita lida corretamente com ficheiros vazios.
- **Ficheiro de destino sem permissões de escrita:** Apresenta erro ao tentar abrir o ficheiro de destino com modo de escrita. ""

Screenshot dos Testes:



```
% acrescenta ./out/texto.txt ./out/texto.txt.copia

Conteúdo de './out/texto.txt' acrescentado com sucesso a './out/texto.txt.copia'.
Terminou comando acrescenta com código 0
%
```

Figura 3: Teste comando acrescenta

3.5 Comando "conta linhas" (Alínea d)

Funcionalidade: Conta e apresenta no ecrã o número total de linhas existentes no ficheiro especificado.

Implementação:

```
1 int conta(const char *filename) {
2     int fd, n;
3     char buffer[4096];
4     int line_count = 0;
5
6     // Abrir o ficheiro
7     fd = open(filename, O_RDONLY);
8     if (fd == -1) {
9         fprintf(stderr, "Erro: O ficheiro '%s' nao existe ou nao
10        pode ser aberto.\n", filename);
11        return 1;
12    }
13
14    // Ler e contar linhas
15    while ((n = read(fd, buffer, sizeof(buffer))) > 0) {
16        for (int i = 0; i < n; i++) {
17            if (buffer[i] == '\n') {
18                line_count++;
19            }
20        }
21    }
22
23    if (n < 0) {
24        fprintf(stderr, "Erro: Falha ao ler o ficheiro '%s'.\n",
25        filename);
26        close(fd);
27        return 1;
28    }
29 }
```

```
27
28 // Fechar ficheiro
29 close(fd);
30
31 printf("\n\n0 ficheiro '%s' tem %d linhas.\n", filename,
line_count);
32 return 0;
33 }
```

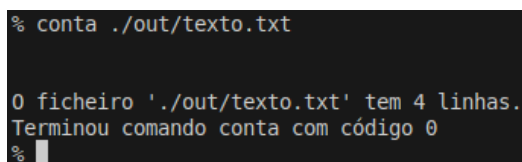
Código 4: Função conta()

Tratamento de Erros: Caso o ficheiro não exista ou não possa ser aberto, é exibida uma mensagem de erro apropriada no `stderr`. Também é tratado o erro de leitura durante o processo de contagem.

Testes Realizados:

- **Ficheiro existente com várias linhas:** Conta corretamente o número de linhas.
- **Ficheiro inexistente:** Apresenta mensagem de erro informando que o ficheiro não existe ou não pode ser aberto.
- **Ficheiro vazio:** Conta zero linhas, conforme esperado.
- **Ficheiro com linhas sem ' ' finais:** Conta apenas as linhas terminadas por ' ', comportando-se de acordo com a definição de linhas.

Screenshot dos Testes:



```
% conta ./out/texto.txt
0 ficheiro './out/texto.txt' tem 4 linhas.
Terminou comando conta com código 0
% 
```

Figura 4: Teste comando conta

3.6 Comando "apaga ficheiro" (Alínea e)

Funcionalidade: Apaga (remove) o ficheiro especificado do sistema de ficheiros.

Implementação:

```
1 int apaga(const char *filename) {
2     struct stat st;
3
4     // Verificar se o ficheiro existe
5     if (stat(filename, &st) != 0) {
6         fprintf(stderr, "Erro: O ficheiro '%s' nao existe.\n",
7             filename);
8         return 1;
9     }
10
11    // Tentar apagar o ficheiro
12    if (unlink(filename) != 0) {
13        fprintf(stderr, "Erro: Nao foi possivel remover o ficheiro
14            '%s'.\n", filename);
15        return 1;
16    }
17
18    printf("\n\nFicheiro '%s' removido com sucesso.\n", filename);
19    return 0;
20 }
```

Código 5: Função apaga()

Tratamento de Erros: Se o ficheiro não existir, é apresentada uma mensagem de erro apropriada no stderr. Caso o ficheiro não possa ser removido (por exemplo, devido a permissões), também é mostrado um erro.

Testes Realizados:

- **Ficheiro existente:** Ficheiro é removido corretamente e mensagem de sucesso é exibida.
- **Ficheiro inexistente:** Apresenta mensagem de erro indicando que o ficheiro não existe.

- **Ficheiro sem permissões de remoção:** Apresenta mensagem de erro adequada informando que não foi possível remover o ficheiro.

Screenshot dos Testes:

```
% apaga ./out/texto.txt.copia  
  
Ficheiro './out/texto.txt.copia' removido com sucesso.  
Terminou comando apaga com código 0  
% █
```

Figura 5: Teste comando apaga ficheiro

3.7 Comando "informa ficheiro" (Alínea f)

Funcionalidade: Apresenta informações detalhadas sobre o ficheiro especificado, incluindo tipo, i-node, utilizador dono, e datas de criação, acesso e modificação.

Implementação:

```
1 int informa(const char *filename) {
2     struct stat file_stat;
3     struct passwd *pw;
4     char time_str[100];
5
6     // Verificar se o ficheiro existe
7     if (access(filename, F_OK) == -1) {
8         fprintf(stderr, "Erro: O ficheiro '%s' nao existe.\n",
9 filename);
10        return 1;
11    }
12
13    if (stat(filename, &file_stat) == -1) {
14        fprintf(stderr, "Erro: Nao foi possivel obter informacoes do
15 ficheiro '%s'.\n", filename);
16        return 1;
17    }
18
19    // Tipo de ficheiro
20    printf("Tipo de ficheiro: ");
21    if (S_ISREG(file_stat.st_mode))
22        printf("Ficheiro regular\n");
23    else if (S_ISDIR(file_stat.st_mode))
24        printf("Diretoria\n");
25    else if (S_ISLNK(file_stat.st_mode))
26        printf("Link simb lico\n");
27    else if (S_ISFIFO(file_stat.st_mode))
28        printf("FIFO/pipe\n");
```

```
27     else if (S_ISSOCK(file_stat.st_mode))
28         printf("Socket\n");
29     else if (S_ISCHR(file_stat.st_mode))
30         printf("Dispositivo de caracteres\n");
31     else if (S_ISBLK(file_stat.st_mode))
32         printf("Dispositivo de blocos\n");
33     else
34         printf("Tipo desconhecido\n");
35
36     // Numero i-node
37     printf("i-node: %lu\n", (unsigned long)file_stat.st_ino);
38
39     // Nome do dono
40     pw = getpwuid(file_stat.st_uid);
41     printf("Utilizador dono: %s\n", pw ? pw->pw_name : "Desconhecido");
42
43     // Data de criacao
44     strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S",
45             localtime(&file_stat.st_ctime));
46     printf("Data de criacao: %s\n", time_str);
47
48     // Data de ultimo acesso
49     strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S",
50             localtime(&file_stat.st_atime));
51     printf("Data do ultimo acesso: %s\n", time_str);
52
53     // Data de ultima modificacao
54     strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S",
55             localtime(&file_stat.st_mtime));
56     printf("Data da ultima modificacao: %s\n", time_str);
```

```
55     printf("\n\nInformacoes do ficheiro '%s' mostradas com sucesso.\n\n", filename);  
56  
57     return 0;  
58 }
```

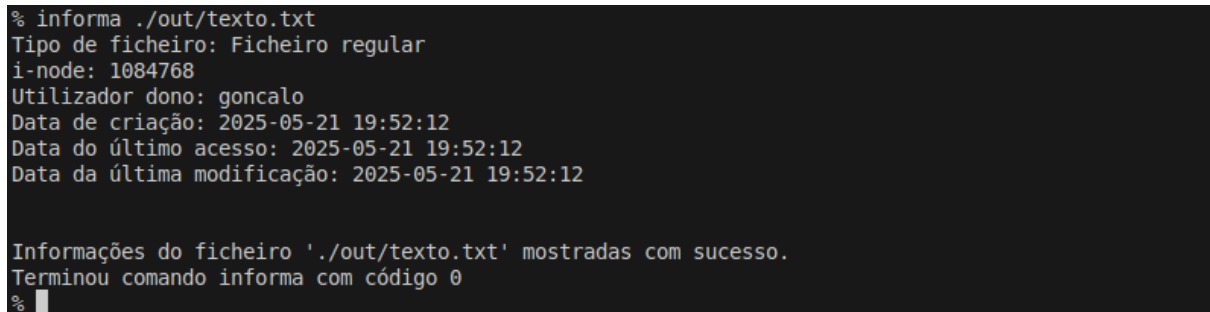
Código 6: Função informa()

Tratamento de Erros: Caso o ficheiro não exista ou ocorra falha ao obter as informações com `stat()`, são exibidas mensagens de erro no `stderr`.

Testes Realizados:

- **Ficheiro existente:** Informações corretamente apresentadas.
- **Ficheiro inexistente:** É apresentada mensagem de erro apropriada.
- **Ficheiro com permissões restritas:** Se o acesso for negado, a função informa do erro.

Screenshot dos Testes:



```
% informa ./out/texto.txt  
Tipo de ficheiro: Ficheiro regular  
i-node: 1084768  
Utilizador dono: goncalo  
Data de criação: 2025-05-21 19:52:12  
Data do último acesso: 2025-05-21 19:52:12  
Data da última modificação: 2025-05-21 19:52:12  
  
Informações do ficheiro './out/texto.txt' mostradas com sucesso.  
Terminou comando informa com código 0  
%
```

Figura 6: Teste comando informa

3.8 Comando "lista [diretoria]" (Alínea g)

Funcionalidade: Lista o conteúdo de uma diretoria, identificando cada entrada como [Ficheiro], [Diretoria] ou [Outro]. Caso nenhum argumento seja fornecido, lista o conteúdo da diretoria atual.

Implementação:

```
1 int lista(const char *path) {
2     DIR *dir;
3     struct dirent *entry;
4     struct stat file_stat;
5     char full_path[1024];
6
7     // Se path e NULL, usar diretoria atual
8     if (path == NULL) {
9         path = ".";
10    }
11
12    // Abrir diretoria
13    dir = opendir(path);
14    if (dir == NULL) {
15        fprintf(stderr, "Erro: Nao foi possivel abrir a diretoria '%s'.\n", path);
16        return 1;
17    }
18
19    printf("Conteudo da diretoria '%s':\n", path);
20
21    // Ler entradas da diretoria
22    while ((entry = readdir(dir)) != NULL) {
23        // Ignorar "." e ".."
24        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
25            continue;
```



```
26     }
27
28     // Construir caminho completo para obter tipo
29     snprintf(full_path, sizeof(full_path), "%s/%s", path, entry
30     ->d_name);
31
32     if (stat(full_path, &file_stat) == -1) {
33         fprintf(stderr, "Erro: Nao foi possivel aceder a '%s'.\n",
34         full_path);
35         continue;
36     }
37
38     // Mostrar nome com tipo textual
39     if (S_ISDIR(file_stat.st_mode)) {
40         printf("[Diretoria] %s\n", entry->d_name);
41     } else if (S_ISREG(file_stat.st_mode)) {
42         printf("[Ficheiro] %s\n", entry->d_name);
43     } else {
44         printf("[Outro] %s\n", entry->d_name);
45     }
46 }
47
48 // Fechar diretoria
49 closedir(dir);
50 printf("\n\nConteudo da diretoria listado com sucesso.\n");
51 return 0;
```

Código 7: Função lista()

Tratamento de Erros: Caso a diretoria não exista ou não possa ser aberta, é apresentada uma mensagem de erro. Se uma entrada não puder ser acedida com `stat()`, o erro é comunicado mas o programa continua com as restantes entradas.

Testes Realizados:

- **Diretoria existente com vários ficheiros/subdiretorias:** Todos os elementos listados com o tipo correspondente.
- **Diretoria inexistente:** Mensagem de erro apropriada.
- **Chamada sem argumentos (NULL):** Diretoria atual listada com sucesso.

Screenshot dos Testes:

```
% lista .  
Conteúdo da diretoria '.':  
[Ficheiro] comandos_ficheiros.h  
[Ficheiro] interpretador.o  
[Ficheiro] interpretador  
[Ficheiro] Makefile  
[Ficheiro] interpretador.c  
[Ficheiro] comandos_ficheiros.o  
[Diretoria] out  
[Ficheiro] comandos_ficheiros.c  
  
Conteúdo da diretoria listado com sucesso.  
Terminou comando lista com código 0
```

Figura 7: Teste comando lista

4 Interpretador de Linha de Comandos

4.1 Arquitetura do Interpretador

O interpretador foi desenvolvido com base nos seguintes princípios fundamentais:

- **Loop Principal:** Lê continuamente os comandos do utilizador até que este introduza o comando termina.
- **Parsing:** A linha de comandos é dividida nos seus argumentos para posterior processamento.
- **Execução:** Tenta executar primeiro como comando personalizado. Se não for reconhecido, tenta como comando do sistema.
- **Gestão de Processos:** Utiliza as funções `fork()` e `execvp()` para executar comandos do sistema.
- **Relatório de Status:** Após cada execução, é sempre mostrado o código de terminação do comando.

4.2 Estrutura do Código

A estrutura principal do interpretador é baseada numa função `main()` com um ciclo infinito, como ilustrado abaixo:

```
1 int main() {  
2     char command[MAX_COMMAND_LENGTH];  
3     char *args[MAX_ARGS];  
4     pid_t pid;  
5     int status;  
6  
7     while (1) {  
8         printf("%s ");  
9         // L comando  
10        // Analisa comando
```

```
11     // Executa comando
12     // Reporta resultado
13 }
14 }
```

Código 8: Estrutura base do interpretador

4.3 Gestão de Processos

- **Comandos Personalizados:**
 - Executados diretamente no processo principal.
 - O retorno do comando é imediato, com o respetivo código de erro.

4.4 Tratamento de Erros

O interpretador lida com os seguintes tipos de erro:

- **Comando Inexistente:** Apresenta mensagem de erro apropriada.
- **Falha na criação de processo:** Informa o utilizador da falha na chamada ao `fork()`.
- **Argumentos em falta:** Verifica a validade dos argumentos e apresenta mensagens de ajuda quando necessário.

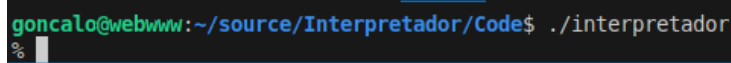
4.5 Testes do Interpretador

Nesta secção são apresentados alguns testes importantes relacionados com o funcionamento global do interpretador.

Teste 1: Execução do Interpretador com `./interpretador`

Este teste verifica se o binário do interpretador foi corretamente compilado e se arranca como esperado.

```
1 $ ./interpretador
2 %
```



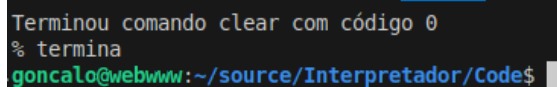
```
goncalo@webwww:~/source/Interpretador/Code$ ./interpretador
```

Figura 8: Teste ./interpretador

Teste 2: Comando termina

Este teste verifica se o interpretador encerra corretamente quando é introduzido o comando termina.

```
1% termina
```



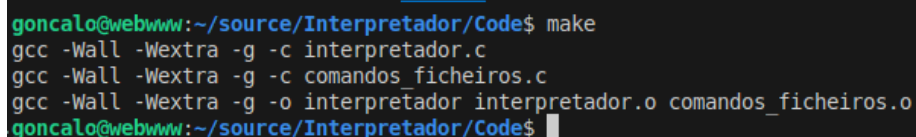
```
Terminou comando clear com código 0  
% termina  
goncalo@webwww:~/source/Interpretador/Code$
```

Figura 9: Teste termina

Teste 3: Utilização do Makefile

Este teste assegura que o interpretador pode ser corretamente compilado com o comando make, demonstrando a correta configuração do Makefile.

```
1$ make
```



```
goncalo@webwww:~/source/Interpretador/Code$ make  
gcc -Wall -Wextra -g -c interpretador.c  
gcc -Wall -Wextra -g -c comandos_ficheiros.c  
gcc -Wall -Wextra -g -o interpretador interpretador.o comandos_ficheiros.o  
goncalo@webwww:~/source/Interpretador/Code$
```

Figura 10: Teste Makefile

5 Gestão de Sistemas Ficheiros

5.1 Alínea a) – Adicionar Disco Virtual e Criar Partição

Para adicionar um novo disco de 10GB ao servidor virtual com Ubuntu, recorreremos ao menu de configurações da máquina virtual.

Screenshots do Processo:

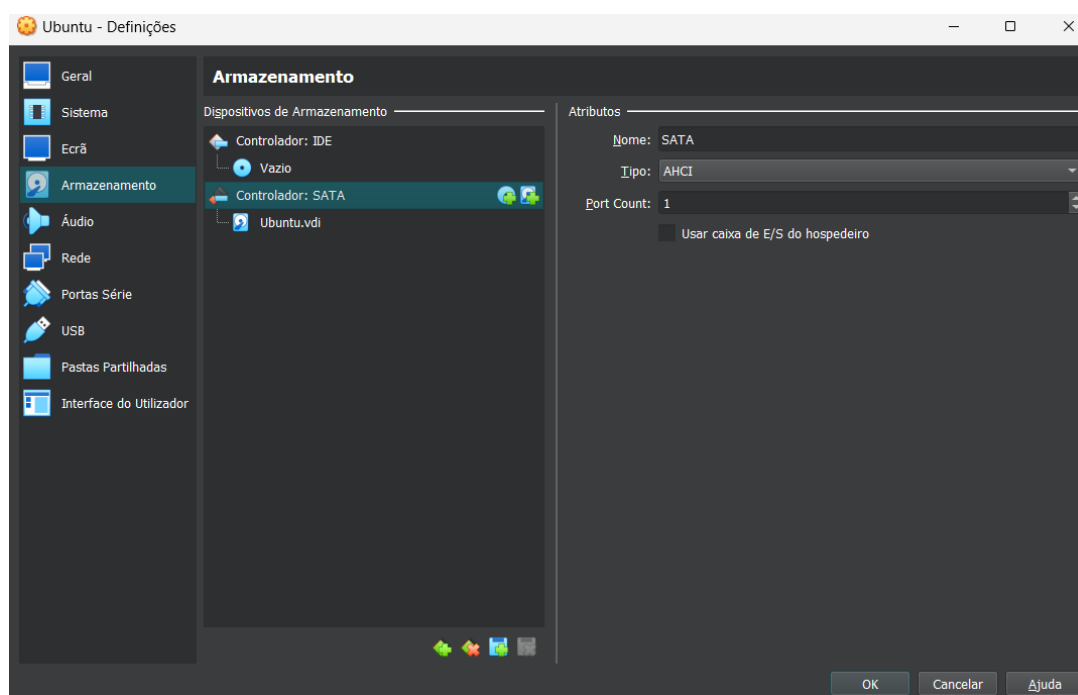


Figura 11: Definições da VM

Este processo foi feito fora da máquina virtual, diretamente nas definições da VM. Acedemos ao menu de configurações da máquina, seleccionámos o **Controlador SATA** e clicámos em **Adicionar Disco**. Depois, escolhemos **Criar novo disco** e definimos o tamanho para **10GB**.

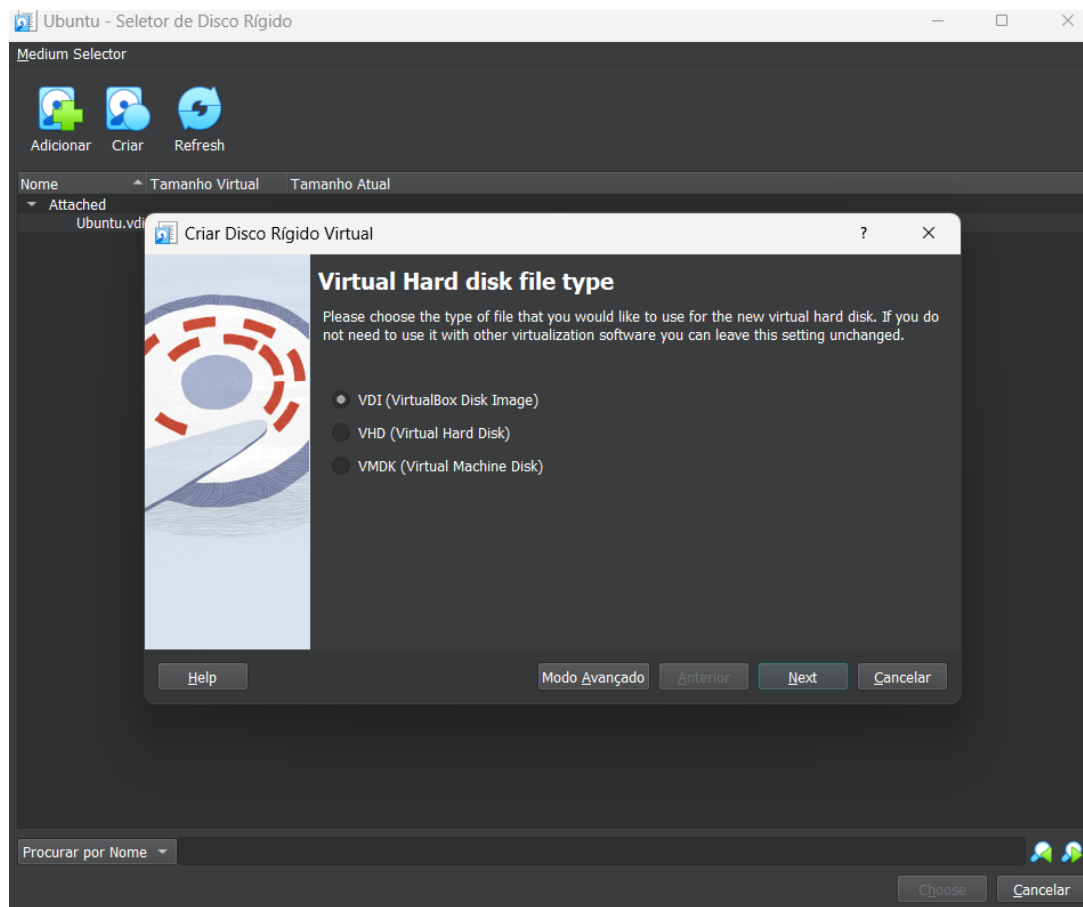


Figura 12: Opção Virtual Disk Image

Durante o processo de criação do disco, foi selecionado o tipo **VDI (VirtualBox Disk Image)**. Este formato é o padrão do VirtualBox e serve para armazenar o conteúdo do disco virtual. É eficiente e compatível com todas as funcionalidades da plataforma. Criamos o disco alocado dinamicamente.

5.1.1 Definição de tamanho

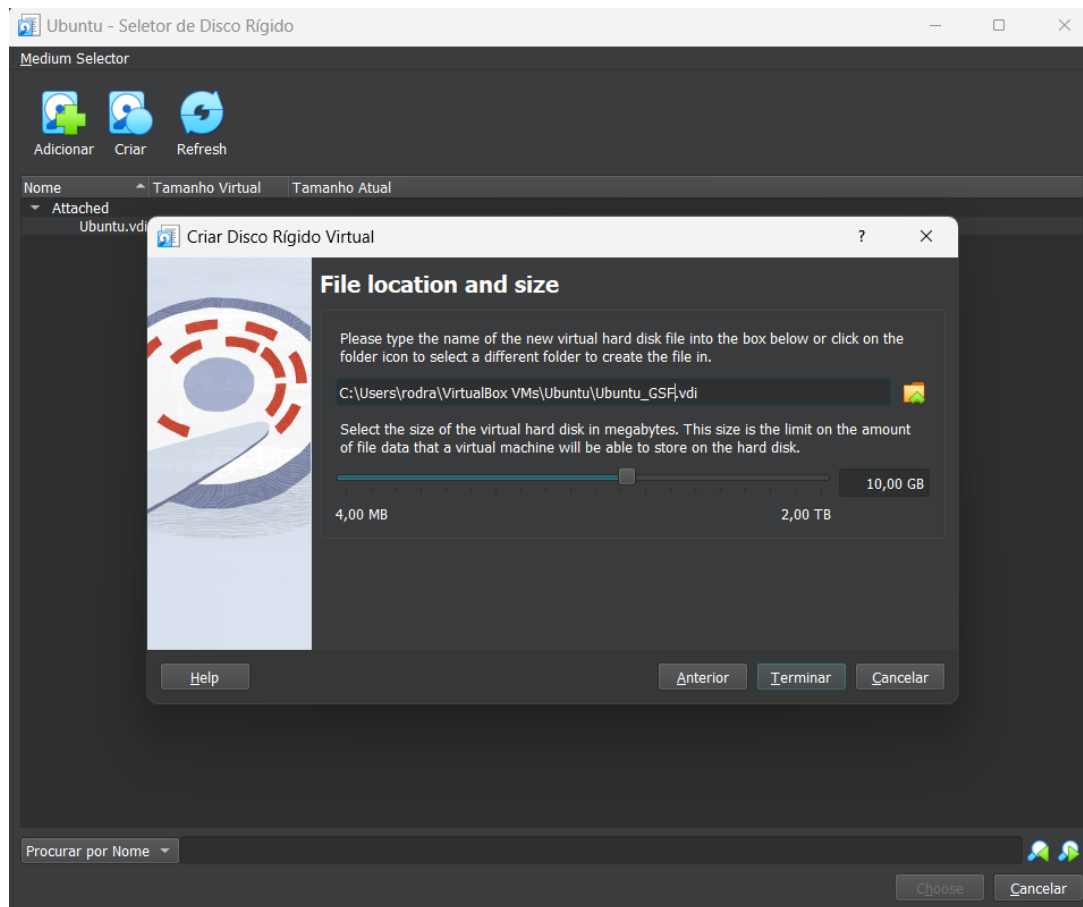


Figura 13: Opção Virtual Disk Image

Como pedido, alterámos o nome do disco para `Ubuntu_GSF` e definimos o tamanho do disco para **10GB**.

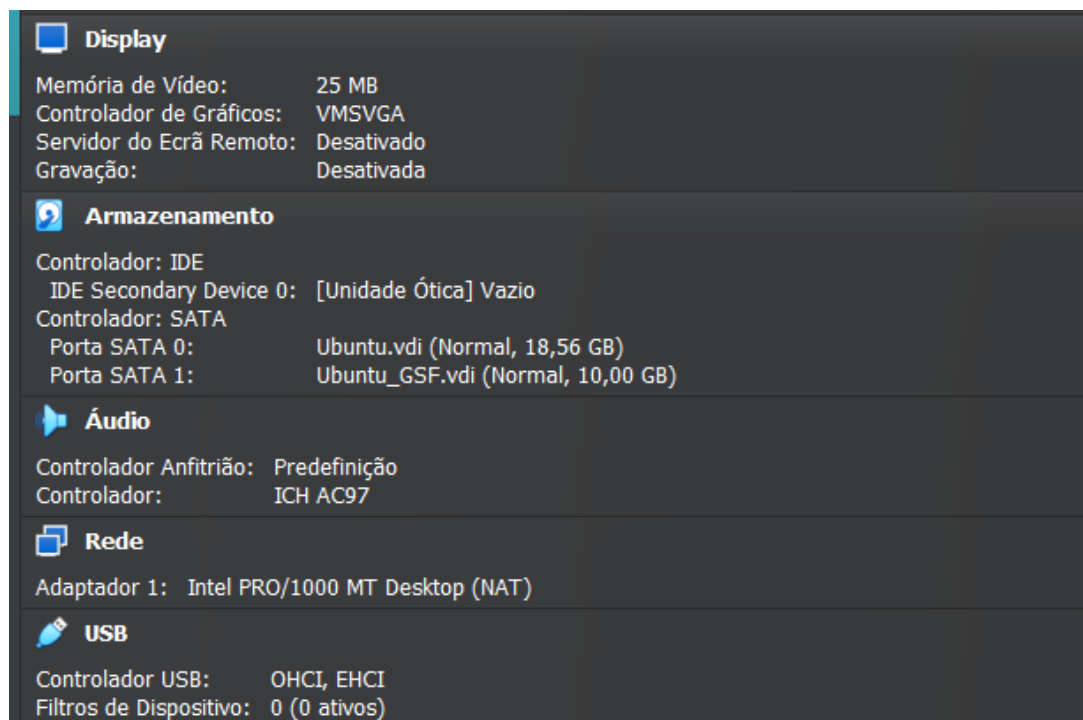
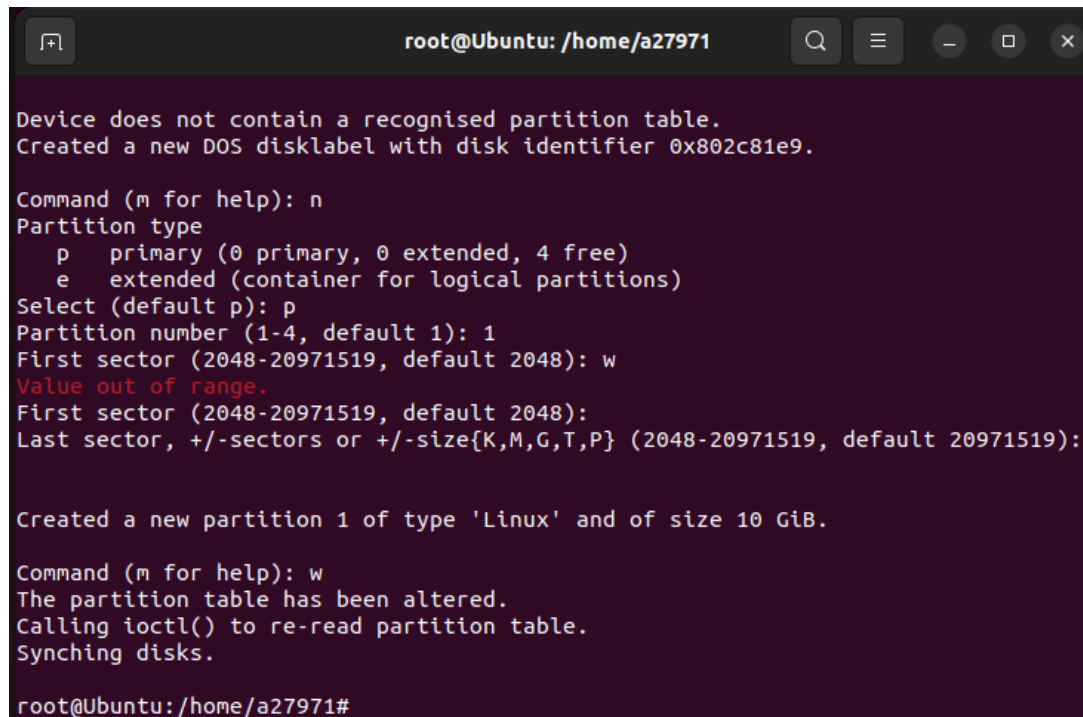


Figura 14: Informações do Disco

A partição Ubuntu_GSF foi criada com sucesso com **10GB**.

5.1.2 Criação da Partição com fdisk



```
root@Ubuntu: /home/a27971
Device does not contain a recognised partition table.
Created a new DOS disklabel with disk identifier 0x802c81e9.

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): w
Value out of range.
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.

root@Ubuntu: /home/a27971#
```

Figura 15: Configurar a partição

Após a criação do novo disco, foi utilizado o comando **fdisk /dev/sdb** para iniciar o processo de particionamento. Este comando abre a ferramenta interativa **fdisk**, permitindo a gestão de partições em dispositivos de armazenamento.

Dentro da ferramenta, os seguintes passos foram executados:

- **n** – Criar uma nova partição
- **p** – Selecionar o tipo de partição como primária
- **1** – Número da partição (primeira partição do disco)
- Pressionar **Enter** para aceitar o setor inicial por defeito
- Pressionar **Enter** novamente para aceitar o setor final e usar todo o espaço disponível
- **w** – Gravar as alterações e sair

Depois de concluírmos o processo, a nova partição ficou criada, pronta a ser formatada e utilizada pelo sistema operativo.

5.2 Alínea b) – Criação de Volume Físico e Volumes Lógicos

```
root@Ubuntu:/home/a27971# pvcreate /dev/sdb1
Physical volume "/dev/sdb1" successfully created.
root@Ubuntu:/home/a27971# vgcreate vgsosd /dev/sdb1
No device found for /dev/sdb1.
root@Ubuntu:/home/a27971# vgcreate vgsosd /dev/sdb1
Volume group "vgsosd" successfully created
root@Ubuntu:/home/a27971# pvs
PV          VG      Fmt Attr PSize  PFree
/dev/sdb1   vgsosd lvm2 a--  <10.00g <10.00g
root@Ubuntu:/home/a27971#
```

Figura 16: Grupo para os Volumes

Após criar a partição, utilizámos o comando **"pvcreate /dev/sdb1"** para inicializar a partição como um volume físico. Este comando prepara a partição para ser utilizada no sistema de gestão de volumes lógicos (LVM).

De seguida, criámos um Grupo de Volumes com o comando **"vgcreate vg_dados /dev/sdb1"**. Este grupo permite agregar vários volumes físicos para criar uma área de armazenamento maior e mais flexível.

```
root@Ubuntu:/home/a27971# lvcreate -L 5G -n lvsosd1 vgsosd
Logical volume "lvsosd1" created.
root@Ubuntu:/home/a27971# lvcreate -L 5G -n lvsosd2 vgsosd
Volume group "vgsosd" has insufficient free space (1279 extents): 1280 required.
root@Ubuntu:/home/a27971# lvcreate -L 1279 -n lvsosd2 vgsosd
Rounding up size to full physical extent 1.25 GiB
Logical volume "lvsosd2" created.
root@Ubuntu:/home/a27971#
```

Figura 17: Criação de Volumes

Com o grupo de volumes criado, foi possível definir um volume lógico (Logical Volume) através do comando **"lvcreate -L 5G -n lvsosd1 vg_dados"**. Inicialmente, pensámos que seria possível criar duas partições com 5G cada, mas ao tentarmos criar a segunda partição, verificámos que apenas estavam disponíveis **1279 extents**, enquanto seriam necessários **1280** para os 5G completos.

Por esse motivo, utilizámos o comando **"lvcreate -l 1279 -n lvsosd2 vg_dados"**, recorrendo à opção **"-l"**, que define o tamanho em extents. Esta segunda partição acabou por ter um tamanho ligeiramente inferior, aproximadamente **4,995G**.

```
root@Ubuntu:/home/a27971# sudo lvcreate -L 5G -n lvsosd2 vgsosd
Volume group "vgsosd" has insufficient free space (1279 extents): 1280 required.
root@Ubuntu:/home/a27971# sudo lvcreate -l 1279 -n lvsosd2 vgsosd
Logical volume "lvsosd2" created.
root@Ubuntu:/home/a27971# lvsdisplay
--- Logical volume ---
LV Path                /dev/vgsosd/lvsosd1
LV Name                 lvsosd1
VG Name                 vgsosd
LV UUID                 HHjALM-VPfi-s68f-Injp-c7Hl-1S3T-KR93LH
LV Write Access         read/write
LV Creation host, time  Ubuntu, 2025-05-22 00:31:40 +0100
LV Status                available
# open                  0
LV Size                 5.00 GiB
Current LE              1280
Segments                1
Allocation               inherit
Read ahead sectors      auto
- currently set to      256
Block device            252:0

--- Logical volume ---
LV Path                /dev/vgsosd/lvsosd2
LV Name                 lvsosd2
VG Name                 vgsosd
LV UUID                 PnuKyh-N5YV-sTku-7QLe-qreM-AzLf-QKvxLx
LV Write Access         read/write
LV Creation host, time  Ubuntu, 2025-05-22 00:39:30 +0100
LV Status                available
# open                  0
LV Size                 <5.00 GiB
Current LE              1279
Segments                1
Allocation               inherit
Read ahead sectors      auto
- currently set to      256
Block device            252:1

root@Ubuntu:/home/a27971# s
```

Figura 18: Informação dos Volumes Lógicos

Podemos ver na imagem acima que ambas as partições têm aproximadamente **5G** cada.

5.3 Alínea c) – Criação de Sistemas de Ficheiros

```
root@Ubuntu:/home/a27971# mkfs.ext3 /dev/vgsosd/lvsosd2
mke2fs 1.46.5 (30-Dec-2021)
Creating filesystem with 1309696 4k blocks and 327680 inodes
Filesystem UUID: c7b2fab3-1950-4788-95b8-9b53dc358149
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

root@Ubuntu:/home/a27971#
```

Figura 19: Sistema de ficheiro ext3

```
root@Ubuntu:/home/a27971# mkfs.ext4 /dev/vgsosd/lvsosd1
mke2fs 1.46.5 (30-Dec-2021)
Creating filesystem with 1310720 4k blocks and 327680 inodes
Filesystem UUID: ae831ecc-d586-4723-aab4-01c2d95da902
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912, 819200, 884736

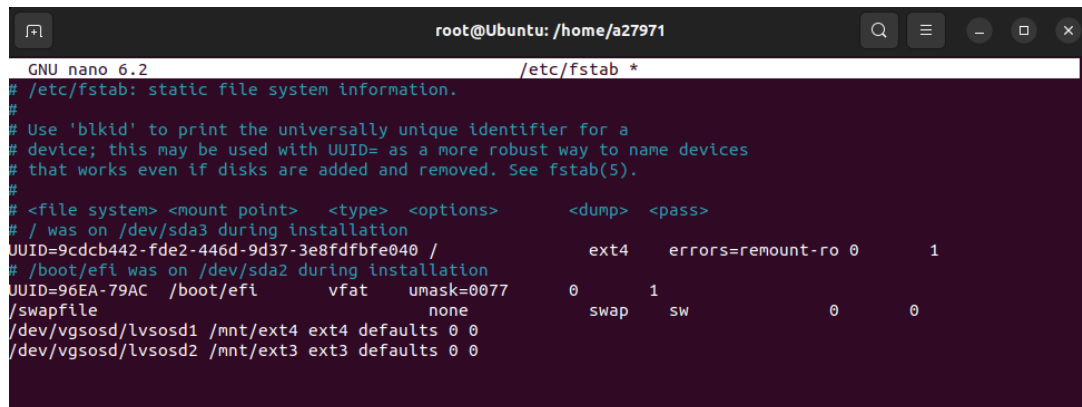
Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

root@Ubuntu:/home/a27971#
```

Figura 20: Sistema de ficheiro ext4

Ao utilizar o código **"mkfs.ext3 /dev/vgsosd/lvsosd2"** criamos um sistema de ficheiros no primeiro volume lógico e **"mkfs.ext4 /dev/vgsosd/lvsosd1"** fazemos o mesmo mas para o segundo.

5.4 Alínea d) – Montagem nos diretórios



```

root@Ubuntu: /home/a27971
GNU nano 6.2 /etc/fstab *
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point> <type> <options> <dump> <pass>
# / was on /dev/sda3 during installation
UUID=9cdecb442-fde2-446d-9d37-3e8fdbfe040 / ext4 errors=remount-ro 0 1
# /boot/efi was on /dev/sda2 during installation
UUID=96EA-79AC /boot/efi vfat umask=0077 0 1
/swapfile none swap sw 0 0
/dev/vgsosd/lvsosd1 /mnt/ext4 ext4 defaults 0 0
/dev/vgsosd/lvsosd2 /mnt/ext3 ext3 defaults 0 0

```

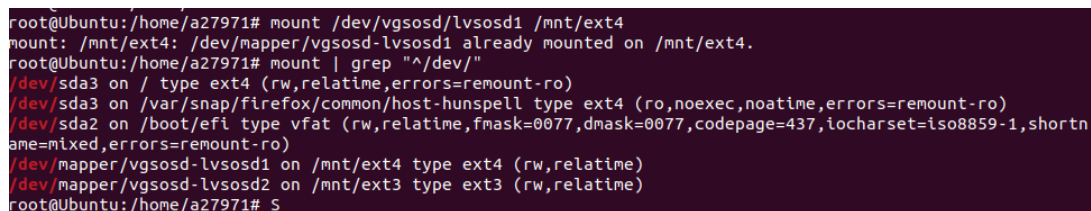
Figura 21: Ficheiro fstab

Depois de criados os diretórios com o comando **"mkdir /mnt/ext3"** e **"mkdir /mnt/ext4"**. Como pedido tornamos a montagem mais resistente a reboots e para isso utilizamos o comando **"nano/etc/fstab"** que serve para abrir o ficheiro em modo de editor de texto.

Ao configurar as entradas no ficheiro **/etc/fstab**, utilizámos a opção **defaults 0 0**.

- **defaults**: aplica as opções padrão de montagem (leitura e escrita, permitir execução de binários, etc.).
- **0 0**: o primeiro **0** indica que o sistema de ficheiros não será incluído em backups com o comando **dump**, e o segundo **0** indica que o sistema de ficheiros não será verificado pelo **fsck** no arranque.

Com esta configuração, os volumes são montados automaticamente após cada reinício, assegurando a resistência da montagem .



```

root@Ubuntu:/home/a27971# mount /dev/vgsosd/lvsosd1 /mnt/ext4
mount: /mnt/ext4: /dev/mapper/vgsosd-lvsosd1 already mounted on /mnt/ext4.
root@Ubuntu:/home/a27971# mount | grep "^/dev/"
/dev/sda3 on / type ext4 (rw,relatime,errors=remount-ro)
/dev/sda3 on /var/snap/firefox/common/host-hunspell type ext4 (ro,noexec,noatime,errors=remount-ro)
/dev/sda2 on /boot/efi type vfat (rw,relatime,fmask=0077,dmask=0077,codepage=437,ioccharset=iso8859-1,shortname=mixed,errors=remount-ro)
/dev/mapper/vgsosd-lvsosd1 on /mnt/ext4 type ext4 (rw,relatime)
/dev/mapper/vgsosd-lvsosd2 on /mnt/ext3 type ext3 (rw,relatime)
root@Ubuntu:/home/a27971# S

```

Figura 22: Montagem de Sistemas de Ficheiros

Na imagem acima, observamos que os ficheiros foram montados, e para isso utilizámos o comando **"mount /dev/vgsosd/lvsosd /mnt/ext4"**. Podemos ver que a tentativa de montar

novamente não é possível e, para termos a certeza de que o sistema de ficheiros foi realmente montado, utilizámos o comando **mount |grep "^dev"**.

O comando **mount |grep "^dev"** serve para filtrar a saída do comando **mount**, mostrando apenas os sistemas de ficheiros que estão montados a partir de dispositivos físicos ou volumes lógicos.

5.5 Alínea e) – Criação de Ficheiro com Permissões Específicas

```
root@Ubuntu:/home/a27971# cd /mnt/ext4
root@Ubuntu:/mnt/ext4# touch 27971-27985.txt
root@Ubuntu:/mnt/ext4# chmod 604 27971-27985.txt
root@Ubuntu:/mnt/ext4#
```

Figura 23: Criação de Ficheiro e permissões para o mesmo

Na imagem acima, foi acedida a diretoria **/mnt/ext4** com o comando **"cd /mnt/ext4"**. Em seguida, foi criado um ficheiro com o nome **27971-27985.txt**, que representa os números dos alunos do grupo, utilizando o comando **"touch 27971-27985.txt"**.

Para configurar as permissões corretamente, foi usado o comando **"chmod 604 27971-27985.txt"**.

O modo **604** define as seguintes permissões:

- O dono do ficheiro tem permissões de leitura e escrita (**6**).
- O grupo associado ao ficheiro não tem qualquer permissão (**0**).
- Todos os outros utilizadores têm apenas permissão de leitura (**4**).

Desta forma, garante-se que apenas o utilizador dono do ficheiro pode modificá-lo, enquanto os restantes apenas o podem ler, e o grupo não tem qualquer acesso.

6 Análise de Sistemas de Ficheiros

6.1 Alínea a) Montagem da Imagem

```
root@Ubuntu:/home/a27971# sudo mount -o loop fs.img /mnt
root@Ubuntu:/home/a27971# ls -l /mnt
total 2
-rwxr-xr-x 1 root root 10 Feb 29 2024 ficheiro1
root@Ubuntu:/home/a27971#
```

Figura 24: Montagem da imagem fs.img

Como foi necessário montar o ficheiro **fs.img** no diretório **/mnt**, utilizámos o comando **"sudo mount -o loop fs.img /mnt"**.

O comando **"sudo mount -o loop fs.img /mnt"** serve para montar uma imagem de sistema de ficheiros **fs.img** no diretório **/mnt**. A opção **-o loop** permite tratar o ficheiro de imagem como um dispositivo virtual, possibilitando o acesso ao seu conteúdo sem ser necessário gravá-lo numa partição física.

6.2 Alínea b) Conteúdo do ficheiro

```
root@Ubuntu:/home/a27971# fls -r /home/a27971/fs.img
r/r * 4:      ipca.txt
r/r * 5:      _ICHEI~1.SWP
r/r 7:  ficheiro1
r/r * 9:      ficheiro2
r/r * 10:     _ICHEI~2.SWP
v/v 32691:    $MBR
v/v 32692:    $FAT1
v/v 32693:    $FAT2
v/v 32694:    $OrphanFiles
root@Ubuntu:/home/a27971#
```

Figura 25: Listagem de diretórios de fs.img

O comando **"fls -r /home/a27971/fs.img"** lista recursivamente todos os ficheiros e diretórios presentes na imagem do sistema de ficheiros localizada em **/home/a27971/fs.img**. A opção **-r** assegura que a listagem inclui os conteúdos dos subdiretórios, permitindo uma análise completa da estrutura do sistema de ficheiros contida na imagem.


```
root@Ubuntu:/home/a27971# find -r /home/a27971/fs.img |grep ipca.txt
r/r * 4:      ipca.txt
root@Ubuntu:/home/a27971# icat /home/a27971/fs.img 4
Este é o conteúdo do ficheiro ipca.txt
root@Ubuntu:/home/a27971#
```

Figura 26: Conteúdo do ficheiro ipca.txt

A saída do comando é filtrada pelo **grep ipca.txt**, que procura especificamente o ficheiro denominado **ipca.txt**. Desta forma, é possível identificar a localização exata deste ficheiro dentro da imagem.

Após identificar o inode correspondente ao ficheiro **ipca.txt**, foi utilizado o comando "icat /home/a27971/fs.img 4" para aceder ao seu conteúdo.

7 Conclusão do Trabalho

Este projeto permitiu-nos aplicar, de forma prática, os conhecimentos adquiridos sobre sistemas operativos, nomeadamente na manipulação de ficheiros, diretórios e processos em ambiente Unix/Linux. Através da implementação de comandos personalizados e de um interpretador em C, desenvolvemos competências em chamadas ao sistema e gestão de erros. A componente de gestão de sistemas de ficheiros reforçou a nossa compreensão sobre particionamento, volumes lógicos e montagem de sistemas de ficheiros. No geral, o projeto foi bem-sucedido e contribuiu significativamente para o nosso desenvolvimento técnico nesta área.

8 Dificuldades Encontradas

Durante a realização do trabalho, uma das principais dificuldades foi elaborar um relatório bem estruturado e com uma boa apresentação. Para isso, decidimos utilizar Latex, uma ferramenta que inicialmente não dominávamos, o que exigiu tempo para aprender um pouco da sua sintaxe e funcionalidades. Esta aprendizagem revelou-se útil para garantir um documento final mais profissional e organizado.

Além disso, na fase de análise de ficheiros, surgiram dificuldades técnicas com a máquina virtual. No nosso caso em particular, a VM em uso estava mal configurada, o que impediu a montagem correta da imagem de sistema de ficheiros. Esta situação atrasou o progresso, exigindo tempo adicional para corrigir a configuração e retomar os testes.

Apesar destes desafios, conseguimos superá-los com persistência, apoio dos materiais disponibilizados na UC e trabalho colaborativo entre os elementos do grupo.

Referências

- Sistemas de Ficheiros - Partições — https://elearning.ipca.pt/2425/pluginfile.php/833079/mod_resource/content/0/03c-Sistemas%20de%20Ficheiros%20-%20Partic%CC%A7o%CC%83es.pdf
- Funções de chamada ao Sistema — https://elearning.ipca.pt/2425/pluginfile.php/834165/mod_resource/content/0/03f-Func%CC%A7o%CC%83es%20de%20chamada%20ao%20sistema%20para%20ficheiros%20em%20Unix.pdf