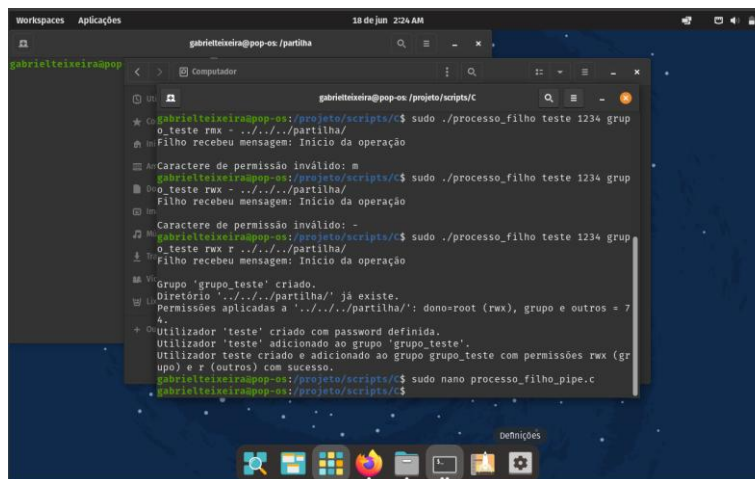


Tecnologias e Programação de Sistemas de Informação

Gestão Automatizada de Utilizadores e Tarefas em Ambiente Unix/Linux com *Bash* e C



```
gabrielteixeira@pop-os: /partilha
gabrielteixeira@pop-os: /projeto/scripts/C$ sudo ./processo_filho teste 1234 grup
o_teste rwx - ../../partilha/
Filho recebeu mensagem: Inicio da operação
Caractere de permissão inválido: m
gabrielteixeira@pop-os: /projeto/scripts/C$ sudo ./processo_filho teste 1234 grup
o_teste rwx - ../../partilha/
Filho recebeu mensagem: Inicio da operação
Caractere de permissão inválido: -
gabrielteixeira@pop-os: /projeto/scripts/C$ sudo ./processo_filho teste 1234 grup
o_teste rwx r ../../partilha/
Filho recebeu mensagem: Inicio da operação
Grupo 'grupo_teste' criado.
Diretório '../../partilha/' já existe.
Permissões aplicadas a '../../partilha/': dono=root (rwx), grupo e outros = 7
4.
Utilizador 'teste' criado com password definida.
Utilizador 'teste' adicionado ao grupo 'grupo_teste'.
Utilizador teste criado e adicionado ao grupo grupo_teste com permissões rwx (gr
upo) e r (outros) com sucesso.
gabrielteixeira@pop-os: /projeto/scripts/C$ sudo nano processo_filho.pipe.c
gabrielteixeira@pop-os: /projeto/scripts/C$
```

Gabriel Teixeira e Tiago Jorge

Cantanhede,

2024 / 2025

ISEC – Politécnico de Coimbra

Tecnologias e Programação de Sistemas de Informação

Relatório

Sistemas Operativos

Gestão Automatizada de Utilizadores e Tarefas em Ambiente Linux com Bash e C

Gabriel Teixeira e Tiago Jorge

Cantanhede,

Ano Letivo 2024/2025

*Fracasso é uma possibilidade. Se as coisas não estão a fracassar,
é porque não está a inovar o suficiente.*

Elon Musk

Índice Geral

Introdução	1
Instalação do Sistema operativo Pop!_OS	2
Atualizar	14
Scripts em Bash	15
Utilizadores com permissões predefinidas	15
Criar grupos	15
Criar utilizadores	18
Backup	20
Monitorização	21
Programas em C	23
Algoritmos de escalonamento	23
FCFS (First Come First Served)	23
Round Robin	26
Sinais	27
Processo filho	30
Conclusão	35
Bibliografia	36

Índice de Figuras

Figura 1: Pop!_OS - Selecionar o idioma	2
Figura 2: Pop!_OS - Selecionar a versão do Português	2
Figura 3: Pop!_OS - Escolha do teclado.....	3
Figura 4: Pop!_OS - Escolha do teclado v2.....	3
Figura 5: Pop!_OS - Escolha do tipo de instalação	4
Figura 6: Pop!_OS - Aviso de falta de carregador	4
Figura 7: Pop!_OS - Escolha do Disco/partição.....	5
Figura 8: Pop!_OS - Criação de Utilizador: Nome e “user name”	5
Figura 9: Pop!_OS - Criação de Utilizador: criação de “Password”.....	6
Figura 10: Pop!_OS - Encriptação de Disco	6
Figura 11: Pop!_OS - Instalação	7
Figura 12: Pop!_OS - Fim da Instalação.....	7
Figura 13: Pop!_OS - Escolha do utilizador	8
Figura 14: Pop!_OS - Iniciar Sessão	8
Figura 15: Pop!_OS - Configuração 1: Barra de tarefas	9
Figura 16: Pop!_OS - Configuração 2: Barra Superior.....	9
Figura 17: Pop!_OS - Configuração 3: Abrir aplicações	10
Figura 18: Pop!_OS - Configuração 4: Gestos de navegação	10
Figura 19: Pop!_OS - Configuração 5: Aparência.....	11
Figura 20: Pop!_OS - Configuração 6: Definições de privacidade	11
Figura 21: Pop!_OS - Configuração 7: Fuso Horário	12
Figura 22: Pop!_OS - Configuração 8: Iniciar sessão a contas.....	12
Figura 23: Pop!_OS - Fim da configuração	13
Figura 24: Pop!_OS - Ambiente de trabalho	13
Figura 25: Pop!_OS - Configuração da Hora	14
Figura 26: Atualizações.....	14
Figura 27: Erro executar script	17
Figura 28: Criação de grupo.....	18
Figura 29: Criação de utilizador.....	19
Figura 30: Utilizador Projeto	19
Figura 31: Pasta partilhada	20
Figura 32: Monitorização	22
Figura 33: Algoritmo de escalonamento – FCFS	25
Figura 34: Algoritmo de escalonamento - Round Robin	27

Figura 35: Sinais	29
-------------------------	----

Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de Sistemas Operativos (SO), com o intuito de aprofundar conhecimentos teóricos e práticos sobre a gestão e funcionamento de sistemas baseados em Unix/Linux. Através da criação de scripts em *Bash* e programas em C, explorámos diversas funcionalidades essenciais para a administração e monitorização do sistema, tais como a gestão automatizada de utilizadores e grupos, definição de permissões, criação de backups e monitorização de recursos.

A componente em *Bash* permitiu-nos desenvolver ferramentas para automatizar tarefas rotineiras, como a criação de utilizadores com permissões predefinidas, a configuração de grupos e a realização de backups automáticos de diretórios. Já a componente em C proporcionou uma experiência mais próxima do núcleo do sistema operativo, nomeadamente na implementação de algoritmos de escalonamento de processos (como *FCFS* e *Round Robin*), na criação e comunicação entre processos filhos via *pipes*, bem como no tratamento de sinais do sistema, com registo em *logs*.

Este projeto teve como objetivo não só a implementação funcional destas ferramentas, mas também a compreensão aprofundada dos conceitos sobre o funcionamento do sistema operativo.

Instalação do Sistema operativo Pop!_OS

Primeiro é necessário escolher o idioma e a sua versão (Português de Portugal).

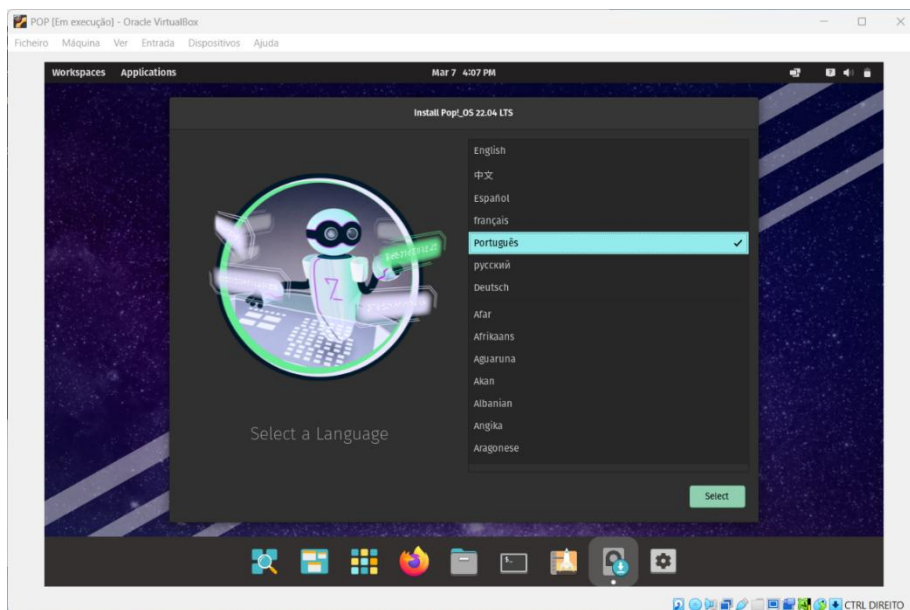


Figura 1: Pop!_OS - Selecionar o idioma

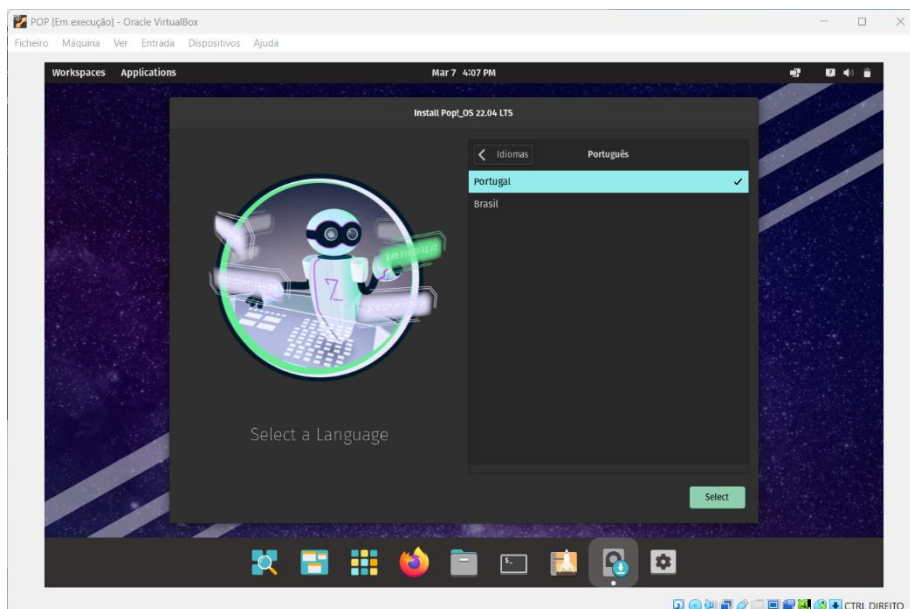


Figura 2: Pop!_OS - Selecionar a versão do Português

Escolha e verificação do teclado.

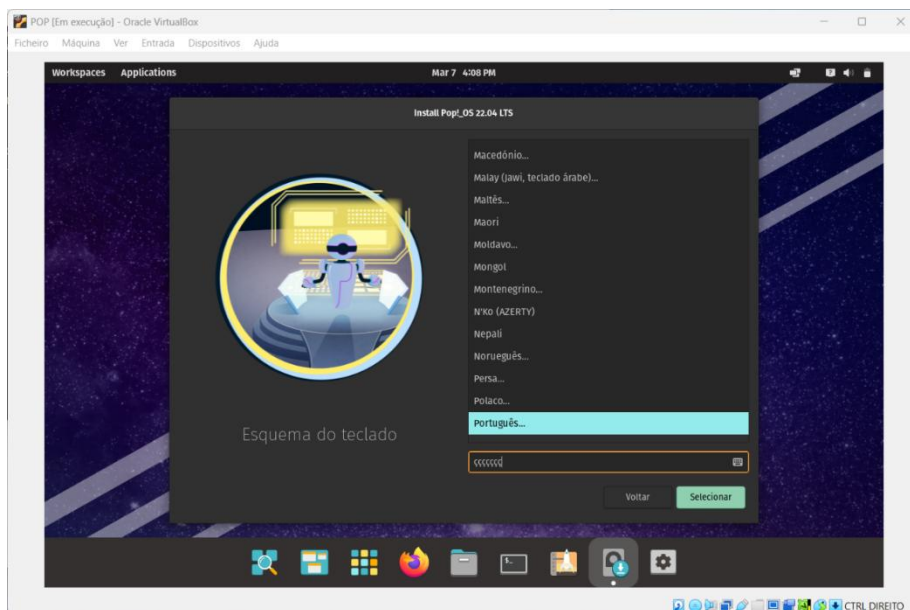


Figura 3: Pop!_OS - Escolha do teclado

Garantir que todo o teclado funciona corretamente.

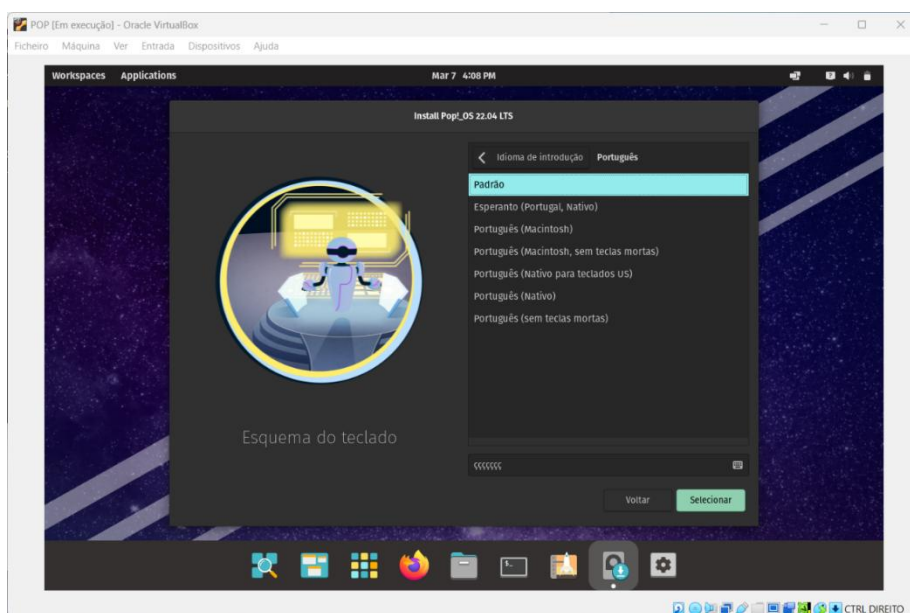


Figura 4: Pop!_OS - Escolha do teclado v2

Escolha do Tipo de instalação.

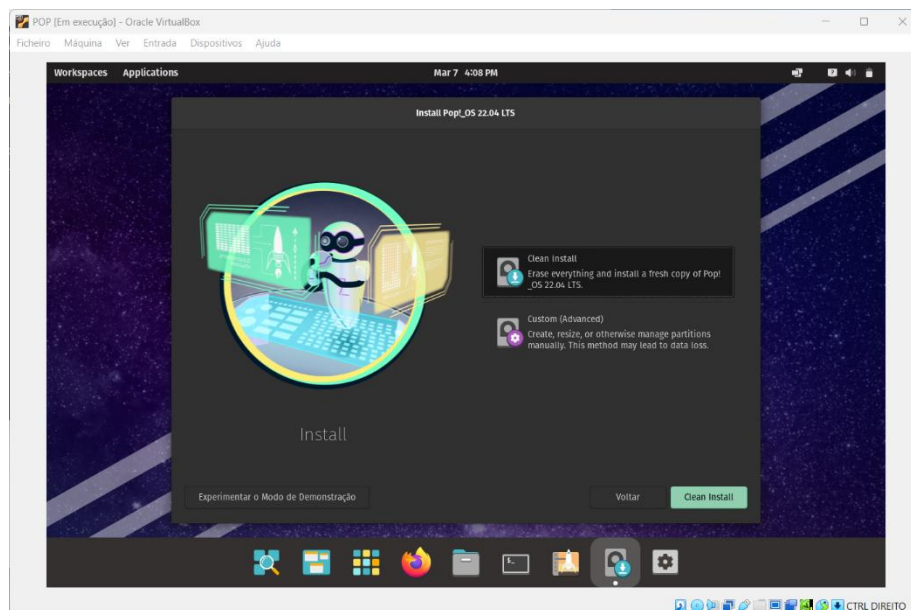


Figura 5: Pop!_OS - Escolha do tipo de instalação

Aviso para ligar carregador.

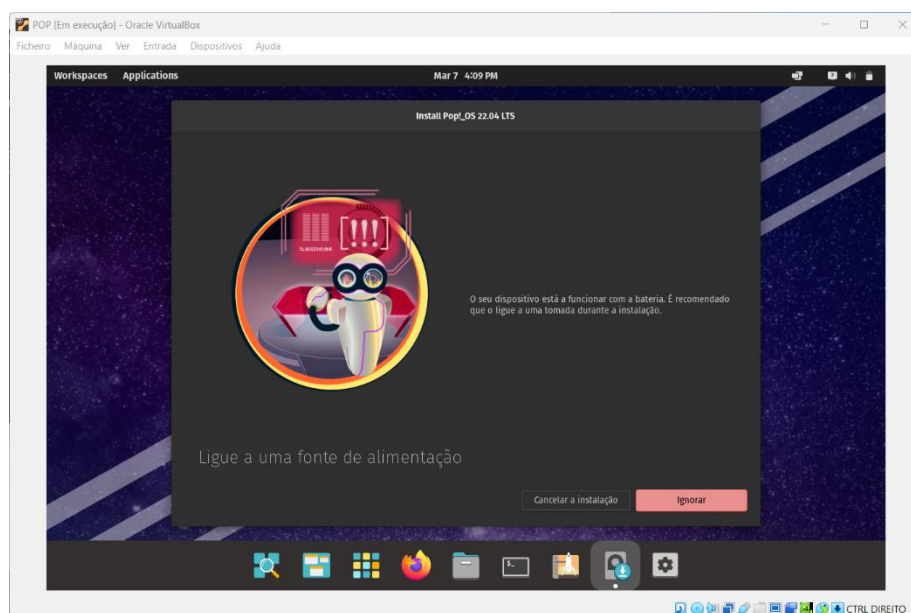


Figura 6: Pop!_OS - Aviso de falta de carregador

Selecionar o Disco ou partição para a instalação.

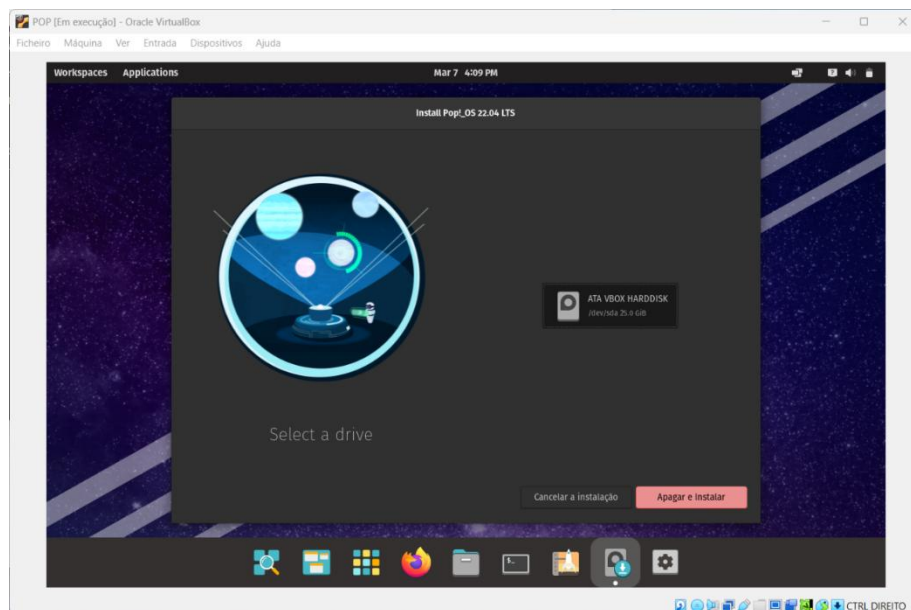


Figura 7: Pop!_OS - Escolha do Disco/partição

Criação de conta de utilizador (Nome completo e Username).

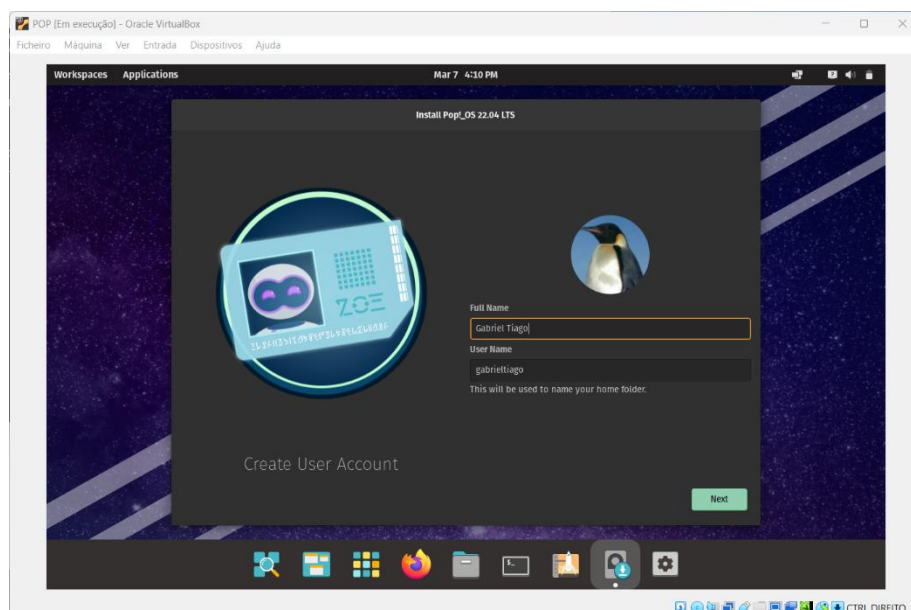


Figura 8: Pop!_OS - Criação de Utilizador: Nome e "user name"

Criação de uma palavra-passe para a conta de utilizador.

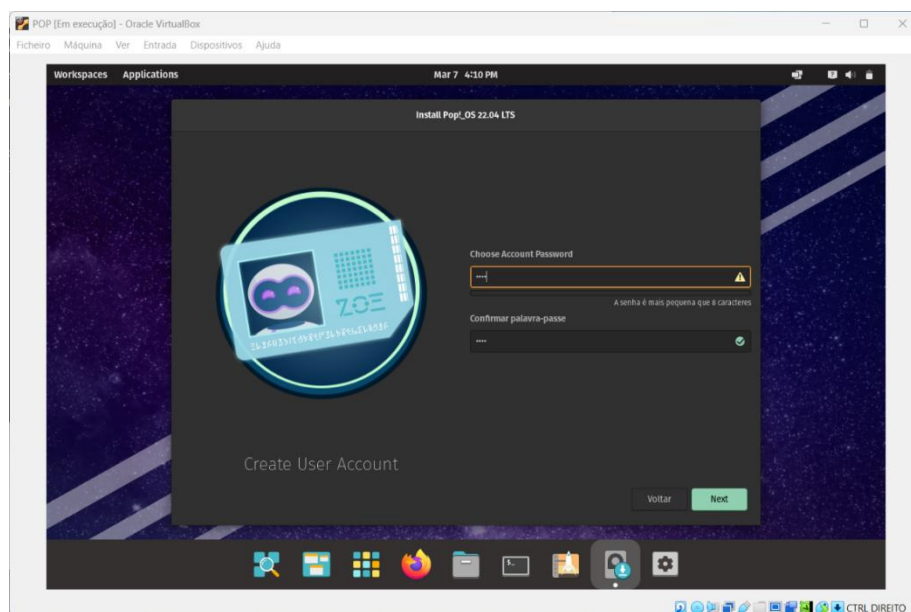


Figura 9: Pop!_OS - Criação de Utilizador: criação de "Password"

Aceitar (ou não) a encriptação total do disco.

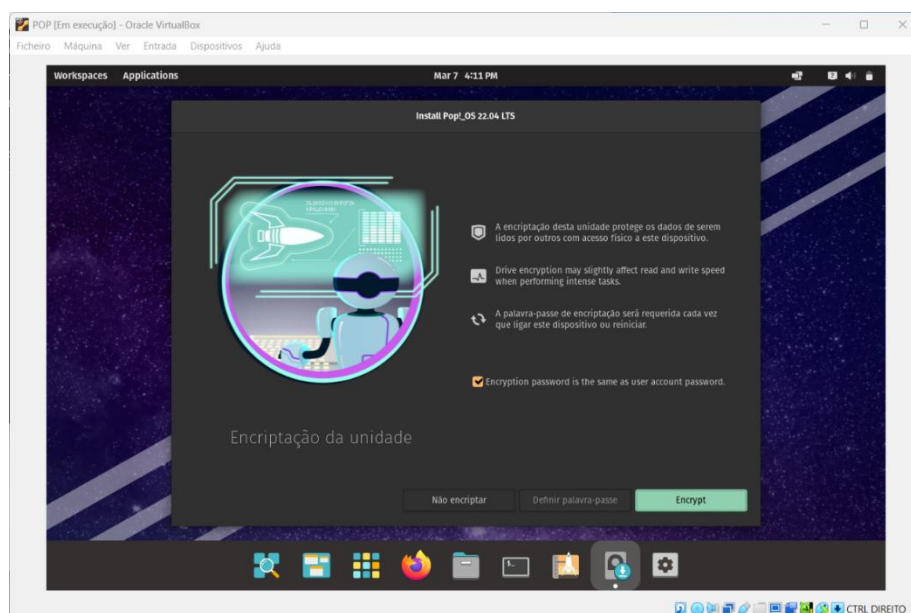


Figura 10: Pop!_OS - Encriptação de Disco

Esperar que a instalação seja concluída.

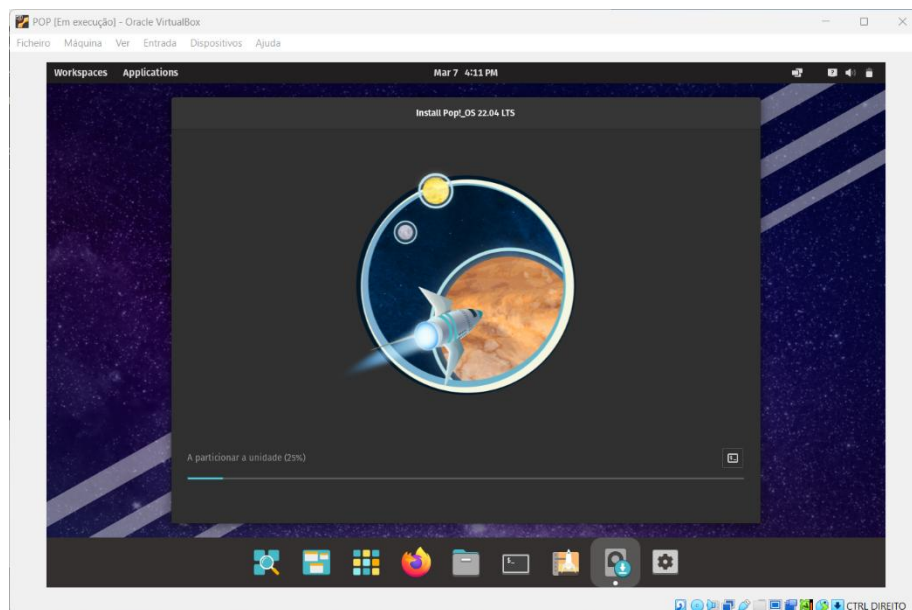


Figura 11: Pop!_OS - Instalação

Fim da instalação e iniciar a configuração.

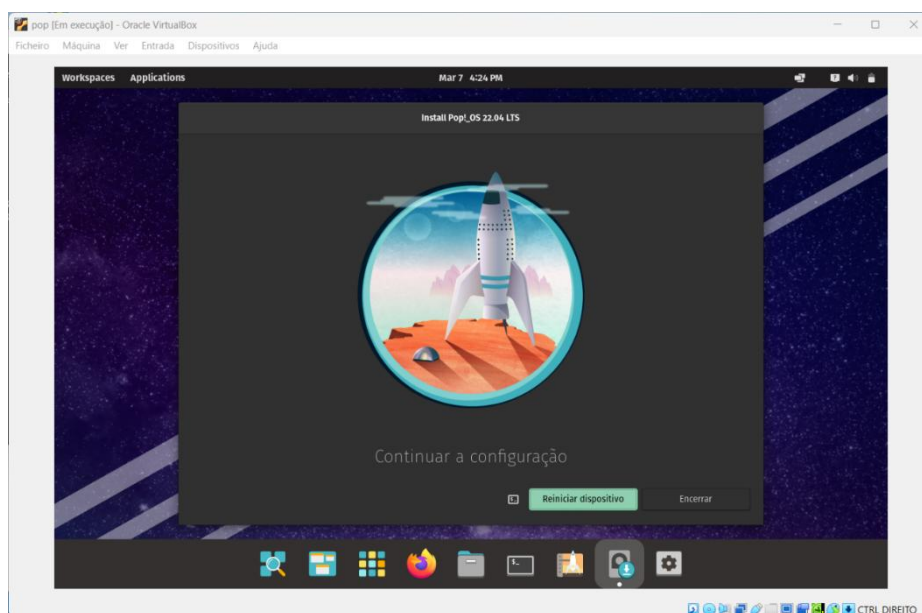


Figura 12: Pop!_OS - Fim da Instalação

Após reiniciar é necessário iniciar sessão na conta criada na instalação.

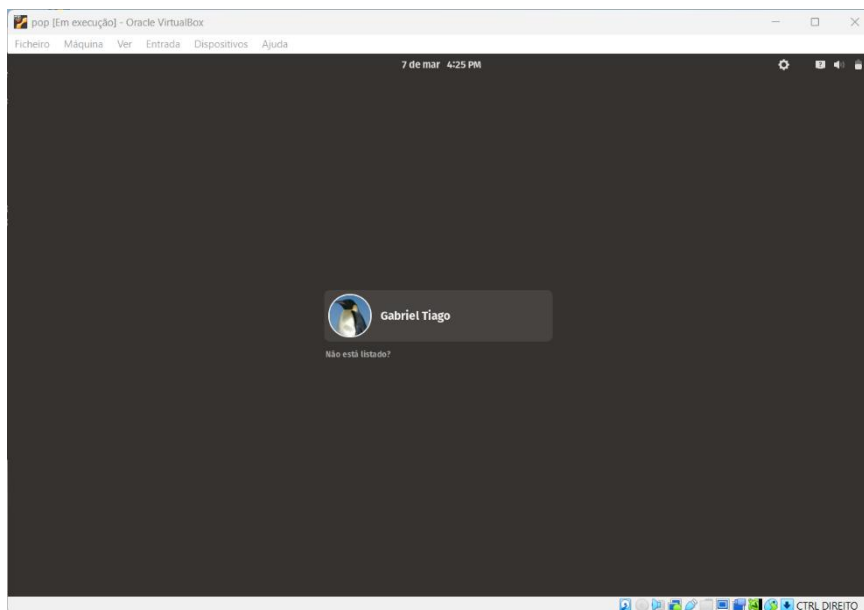


Figura 13: Pop!_OS - Escolha do utilizador

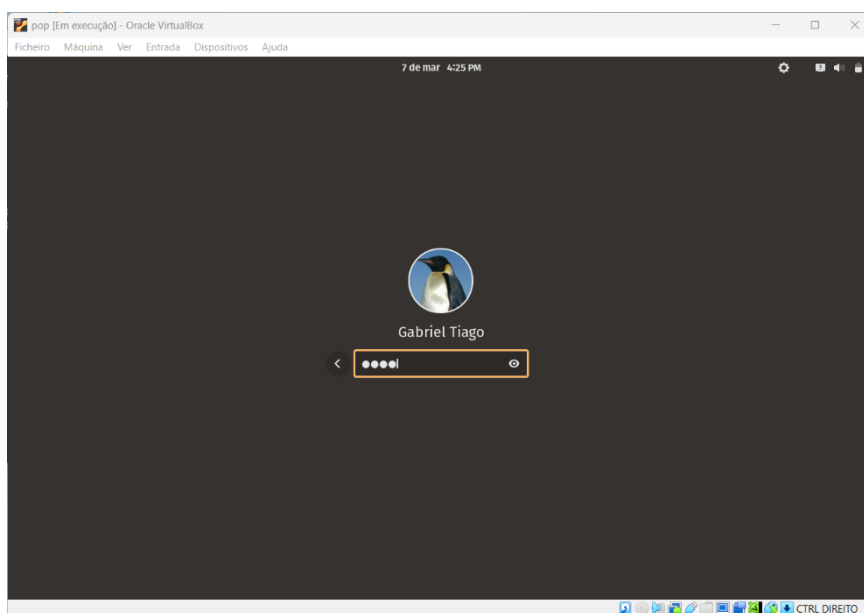


Figura 14: Pop!_OS - Iniciar Sessão

Escolher a visualização da barra de tarefas.

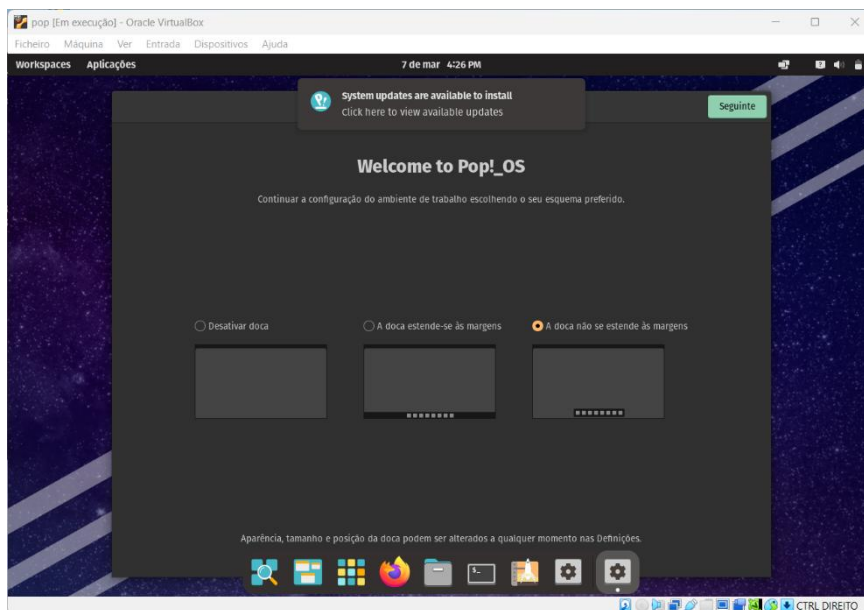


Figura 15: Pop!_OS - Configuração 1: Barra de tarefas

Configuração da barra superior.

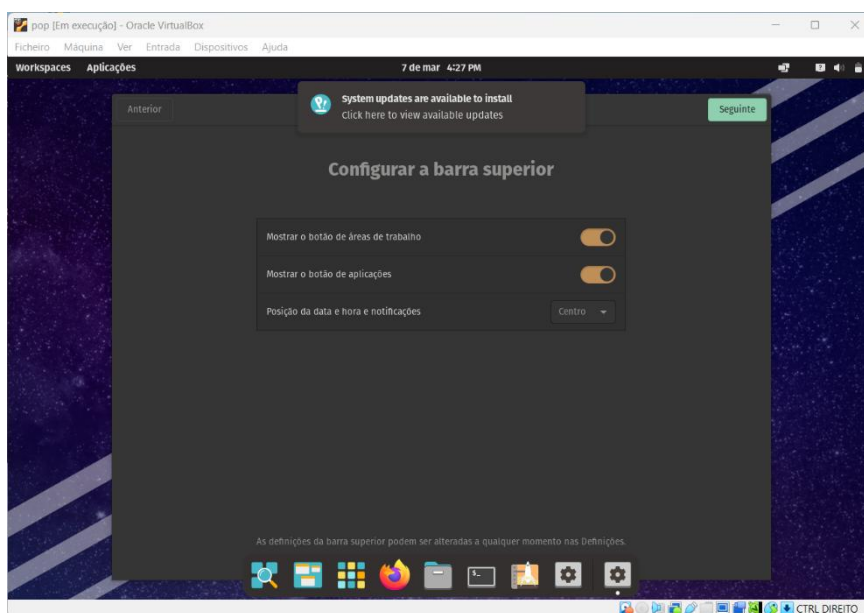


Figura 16: Pop!_OS - Configuração 2: Barra Superior

Apresentação da funcionalidade de abertura rápida de aplicações.

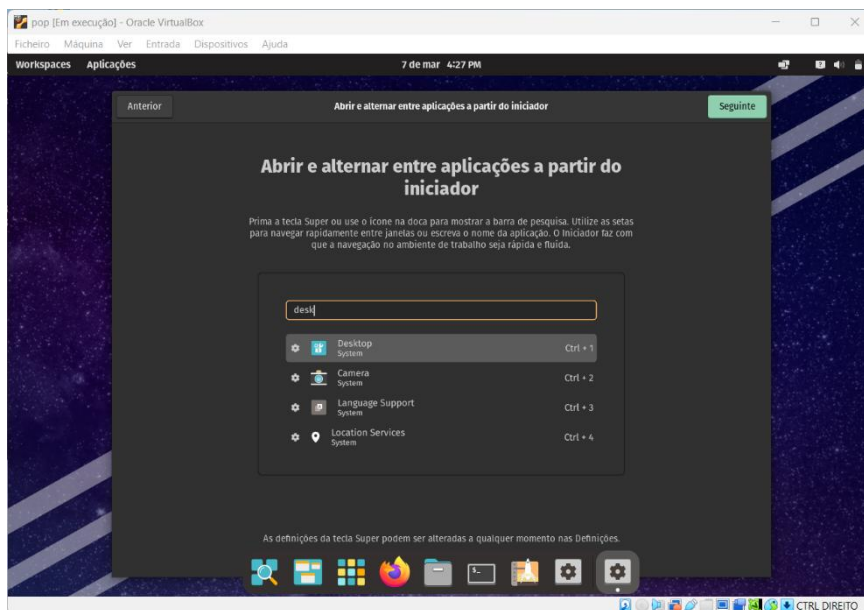


Figura 17: Pop!_OS - Configuração 3: Abrir aplicações

Apresentação da funcionalidade de gestos.

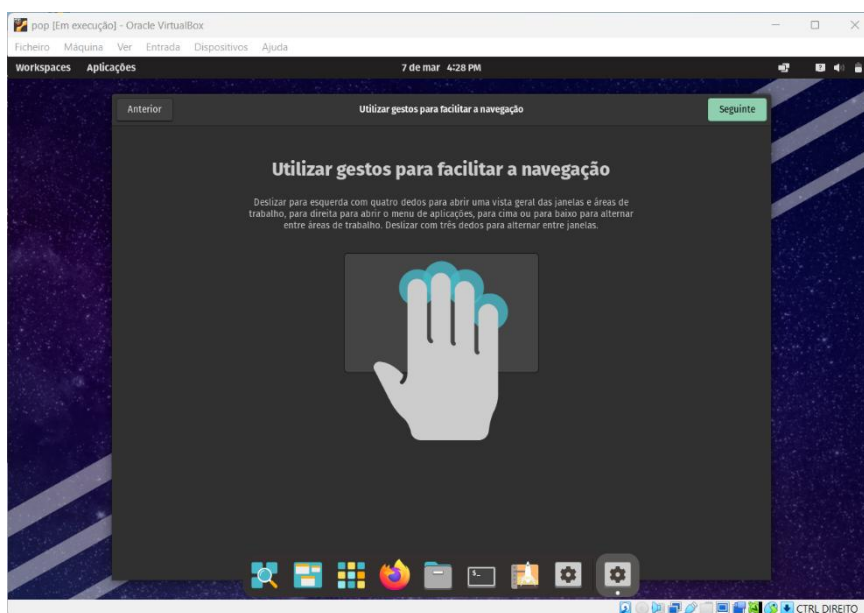


Figura 18: Pop!_OS - Configuração 4: Gestos de navegação

Aparência clara ou escura.

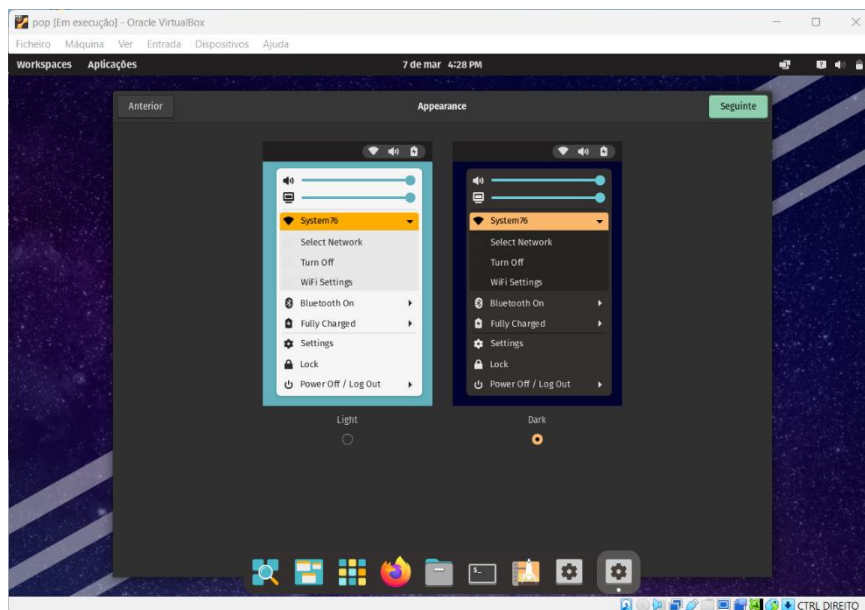


Figura 19: Pop!_OS - Configuração 5: Aparência

Definições de privacidade (acesso a localização).

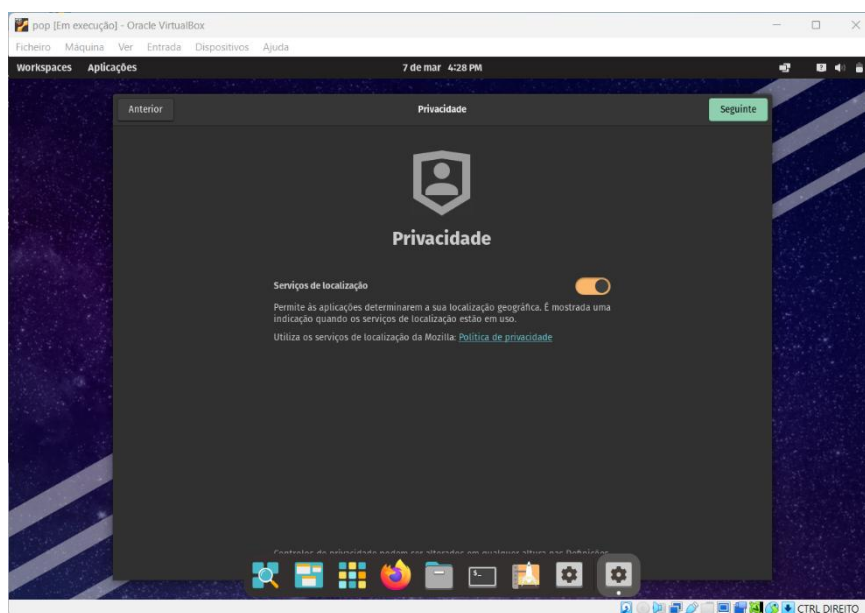


Figura 20: Pop!_OS - Configuração 6: Definições de privacidade

Escolha do Fuso Horário português.

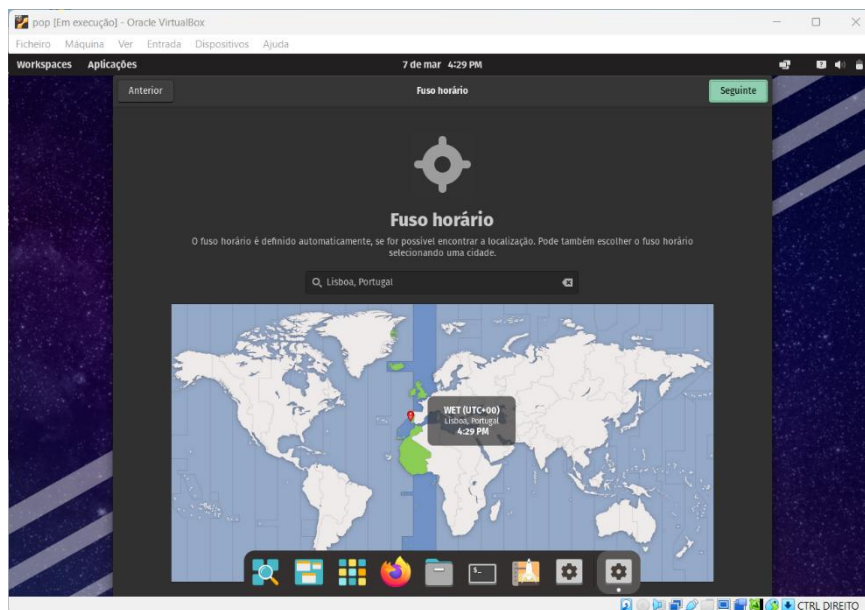


Figura 21: Pop!_OS - Configuração 7: Fuso Horário

Ligação de contas importantes.

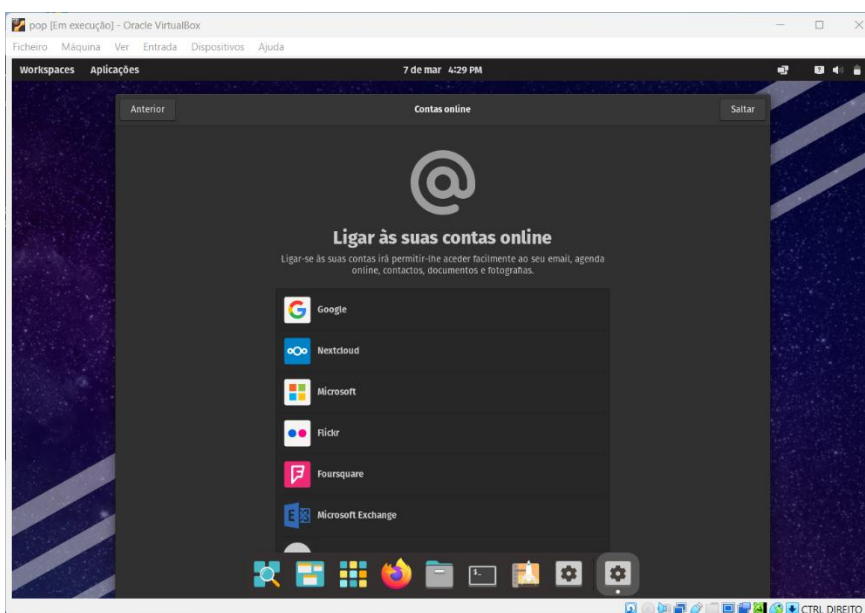


Figura 22: Pop!_OS - Configuração 8: Iniciar sessão a contas

Terminar a configuração.

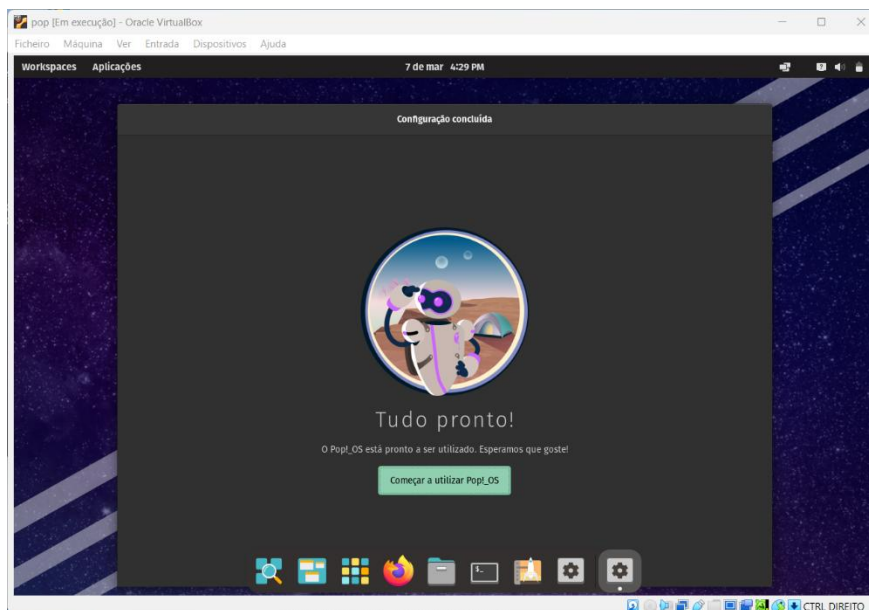


Figura 23: Pop!_OS - Fim da configuração

Configuração padrão terminada, falta apenas alterar a hora para formato “24h”, para isso é necessário abrir as definições selecionadas a vermelho na barra de tarefas (na imagem).

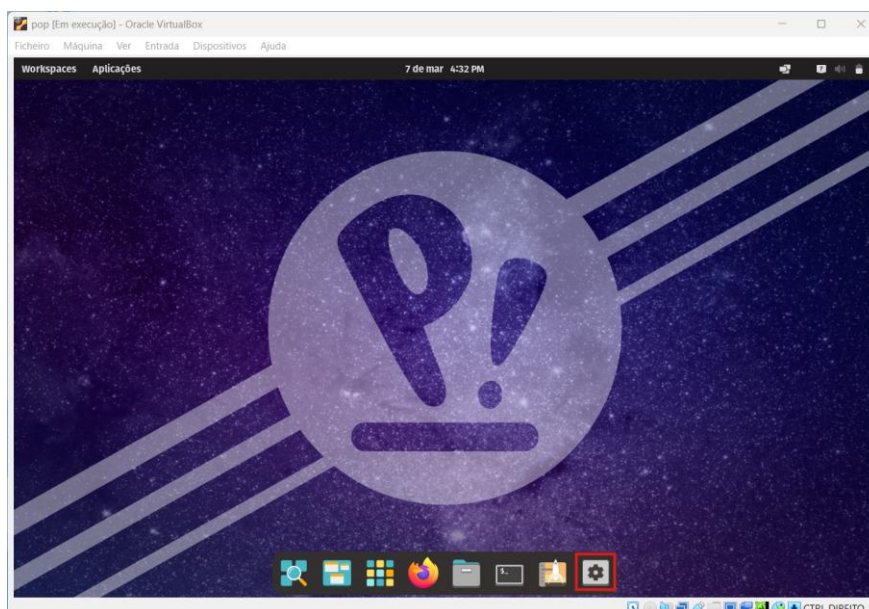


Figura 24: Pop!_OS - Ambiente de trabalho

Abrir as definições de “Data e Hora” e alterar o “Formato de hora” para “24 horas”.

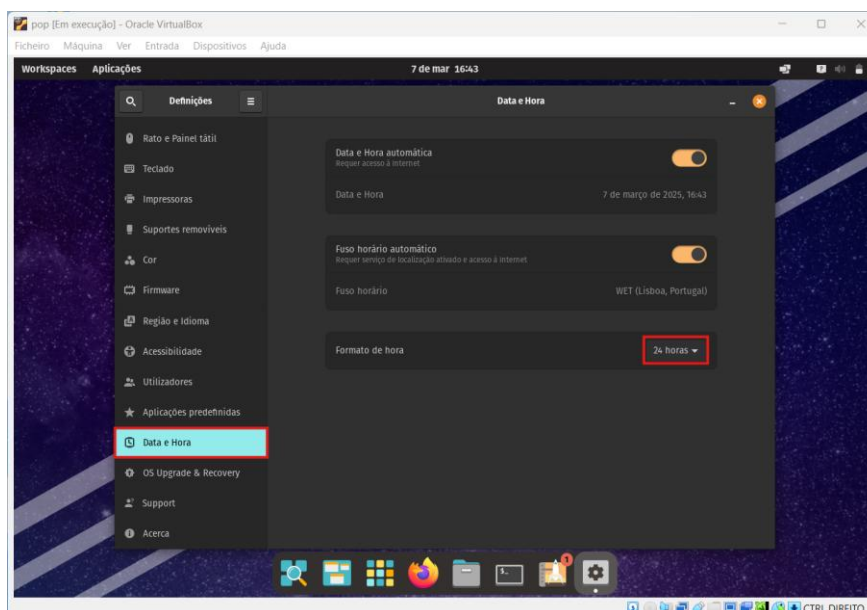


Figura 25: Pop!_OS - Configuração da Hora

Atualizar

Dado que o sistema Pop!_OS já se encontrava instalado, começámos por assegurar que o ambiente estava devidamente atualizado, recorrendo para isso aos seguintes comandos executados no terminal:

```
sudo apt update  
sudo apt upgrade
```

```
gabrielteixeira@pop-os:~$ sudo apt update  
[sudo] senha para gabrielteixeira:  
Atg:2 http://apt.pop-os.org/proprietary jammy InRelease  
Atg:3 http://apt.pop-os.org/release jammy InRelease  
Atg:4 http://packages.microsoft.com/repos/code stable InRelease  
Atg:5 http://apt.pop-os.org/ubuntu jammy InRelease  
Atg:6 http://apt.pop-os.org/ubuntu jammy-security InRelease  
Atg:7 http://apt.pop-os.org/ubuntu jammy-updates InRelease  
Atg:8 http://apt.pop-os.org/ubuntu jammy-backports InRelease  
A ler as listas de pacotes... Pronto  
A construir árvore de dependências... Pronto  
A ler a informação de estado... Pronto  
217 pacotes podem ser atualizados. Corra 'apt list --upgradable' para vê-los.  
gabrielteixeira@pop-os:~$ sudo apt upgrade  
A ler as listas de pacotes... Pronto  
A construir árvore de dependências... Pronto  
A ler a informação de estado... Pronto  
A calcular a atualização... Pronto  
Serão instalados os seguintes NOVOS pacotes:  
linux-headers-6.12.10-76061203 linux-headers-6.12.10-76061203-generic linux-image-6.12.10-76061203-generic  
linux-modules-6.12.10-76061203-generic mesa-libgallium systemd-hwe-hwdb  
Serão atualizados os seguintes pacotes:
```

Figura 26: Atualizações

- `sudo apt update`: atualiza a lista de pacotes disponíveis e as respetivas versões.
- `sudo apt upgrade`: instala as versões mais recentes dos pacotes existentes.

Scripts em Bash

O docente propôs a resolução de três exercícios em *bash*:

- Script para automatizar a criação de utilizadores com permissões predefinidas.
- Script para criar backups automáticos de diretórios de utilizadores.
- Script de monitorização (disco, CPU, RAM, etc.) com registo em ficheiro.

Para a criação dos ficheiros necessários, recorreremos a comandos como *mkdir*, *touch* e *cd*.

Todos os scripts iniciam com a declaração do intérprete *Bash*, conforme indicado abaixo:

```
#!/bin/bash
```

Utilizadores com permissões predefinidas

Para a concretização deste exercício, foram desenvolvidos dois scripts distintos: um destinado à criação de grupos com permissões específicas e outro responsável pela criação dos utilizadores e pela sua associação aos respetivos grupos.

Criar grupos

A primeira parte do script verifica se foram fornecidos, pelo menos, dois argumentos. Caso contrário, é exibida uma mensagem explicativa sobre a forma correta de utilizar o comando, seguida de instruções para definir as permissões do grupo e dos outros utilizadores (sendo que o proprietário recebe automaticamente as permissões máximas).

```
if [ "$#" -lt 3 ]; then
    echo "Uso: $0 <nome_grupo> <permissoes_grupo_outros> <diretorio1> [diretorio2]
    ..."
    echo "Permissões grupo e outros em formato octal, ex: 70 (rwx para grupo, --- para
    outros)"
    exit 1
fi
```

O nome do grupo e as permissões são extraídas para variáveis e removidas dos argumentos.

```
grupo="$1"  
permissoes_go="$2"  
shift 2
```

O excerto seguinte verifica se o grupo não existe, descartando a saída do comando (para que não seja exibida no ecrã) através da utilização de "> /dev/null". Caso o grupo não exista, este é criado; caso contrário, é apresentada uma mensagem a indicar que o grupo já existe.

```
# Criar grupo se não existir  
if ! getent group "$grupo" > /dev/null; then  
    groupadd "$grupo"  
    echo "Grupo '$grupo' criado."  
else  
    echo "Grupo '$grupo' já existe."  
fi
```

É verificado se a estrutura das permissões está correta através do operador "=", que compara se uma variável corresponde a uma expressão regular (*regex*). Neste caso, a *regex* utilizada é "[0-7]{2}\$", onde:

- "^": indica o início da cadeia;
- "[0-7]": representa um dígito entre 0 e 7;
- "{2}": significa que o dígito deve repetir-se exatamente duas vezes;
- "\$" assinala o fim da cadeia.

Os duplos parênteses retos "[[...]]" são obrigatórios para que o operador "=" seja interpretado corretamente. Caso os valores fornecidos sejam inválidos, o script termina e apresenta uma mensagem de apoio ao utilizador.

```
if [[ ! "$permissoes_go" =~ ^[0-7]{2}$ ]]; then  
    echo "Permissões inválidas: devem ser 2 dígitos entre 0 e 7, ex: 70"  
    exit 1  
fi
```


Como referido anteriormente, o dono recebe automaticamente as permissões máximas (7), às quais são adicionadas as permissões atribuídas aos restantes utilizadores.

```
permissoes="7$permissoes_go"
```

Com recurso a um ciclo for os argumentos são percorridos de forma a passar por todos (estes argumentos são diretórios).

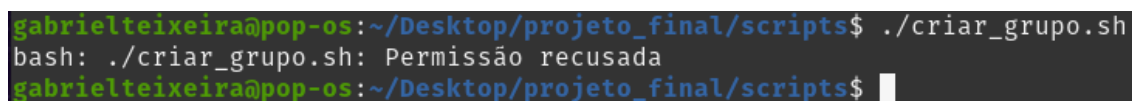
Para cada caminho/diretório é verificado se já está criado, através da opção “-d”, se existir imprime uma mensagem a passar essa informação ao utilizador, se o não tiver criado, cria com recurso à opção “-p” que garante que todos os diretórios do caminho são criados.

O comando “chown” é utilizado para determinar que o dono do diretório passe a ser o *root* e define o grupo a ele associado. Já o comando “chmod” aplica as permissões ao diretório.

```
for dir in "$@"; do
    if [ -d "$dir" ]; then
        echo "Diretório '$dir' já existe."
    else
        mkdir -p "$dir"
        echo "Diretório '$dir' criado."
    fi

    chown root:"$grupo" "$dir"
    chmod "$permissoes" "$dir"
    echo "Permissões aplicadas a '$dir': dono=root (rwx), grupo e outros = $permissoes_go."
done
```

Ao tentar executar o script, surgiu o erro “Permissão recusada”. Recordando que o professor tinha abordado este tipo de erros nas aulas, consultámos os materiais disponibilizados e verificámos que era necessário conceder permissões de execução ao ficheiro. Esta operação é realizada com o comando “chmod”, utilizando a opção “+x” seguida do nome do ficheiro.



```
gabrielteixeira@pop-os:~/Desktop/projeto_final/scripts$ ./criar_grupo.sh
bash: ./criar_grupo.sh: Permissão recusada
gabrielteixeira@pop-os:~/Desktop/projeto_final/scripts$
```

Figura 27: Erro executar script

Após definir as permissões, o script já pode ser executado, como demonstrado na imagem:

```
gabrielteixeira@pop-os:/projeto/scripts/bash$ sudo ./criar_grupo.sh grupo_projeto 70 ../../../../pasta_partilhada/
Grupo 'grupo_projeto' criado.
Diretório '../../../../pasta_partilhada/' já existe.
Permissões aplicadas a '../../../../pasta_partilhada/': dono=root (rwx), grupo e outros = 70.
gabrielteixeira@pop-os:/projeto/scripts/bash$
```

Figura 28: Criação de grupo

Criar utilizadores

O código responsável pela criação de utilizadores é semelhante ao utilizado para a criação de grupos, também começa pela verificação do número de argumentos fornecidos.

```
if [ "$#" -lt 3 ]; then
    echo "Uso: $0 <utilizador> <password> <grupo1> [grupo2] [grupo3] ..."
    exit 1
fi
```

O nome do utilizador e a palavra-passe são guardados e, de seguida, removidos dos argumentos. Para isso, utiliza-se o comando “*shift 2*”, que elimina apenas os dois primeiros argumentos

```
utilizador="$1"
password="$2"
shift 2
```

É verificado se o utilizador já existe; caso não exista, este é criado e é-lhe atribuída a palavra-passe de forma segura.

```
if id "$utilizador" &>/dev/null; then
    echo "O utilizador '$utilizador' já existe."
else
    # Criar utilizador com home
    useradd -m "$utilizador"
    echo "$utilizador:$password" | chpasswd
    echo "Utilizador '$utilizador' criado com password definida."
fi
```


Com a mesma lógica aplicada na criação de grupos, os utilizadores são associados aos respetivos grupos. Caso um grupo não exista, este é criado automaticamente. A associação é efetuada através do comando “*usermod*”.

```
for grupo in "$@"; do
    # Criar grupo se não existir
    if ! getent group "$grupo" > /dev/null; then
        groupadd "$grupo"
        echo "Grupo '$grupo' criado."
    fi

    # Adicionar utilizador ao grupo
    usermod -aG "$grupo" "$utilizador"
    echo "Utilizador '$utilizador' adicionado ao grupo '$grupo'."
done
```

Após atribuir permissões de execução, o script pode ser executado, conforme ilustrado na imagem. Para o exemplo, utilizei o mesmo grupo criado anteriormente.

```
gabrielteixeira@pop-os:/projeto/scripts/bash$ sudo ./criar_utilizadores.sh utilizador_projeto 1234 grupo_projeto
Utilizador 'utilizador_projeto' criado com password definida.
Utilizador 'utilizador_projeto' adicionado ao grupo 'grupo_projeto'.
gabrielteixeira@pop-os:/projeto/scripts/bash$
```

Figura 29: Criação de utilizador

O utilizador foi criado com sucesso, como podemos observar na imagem.



Figura 30: Utilizador Projeto

Autentiquei-me com o novo utilizador e confirmei que conseguia aceder à pasta, bem como verificar as permissões atribuídas. Testei também se as permissões estavam configuradas corretamente.

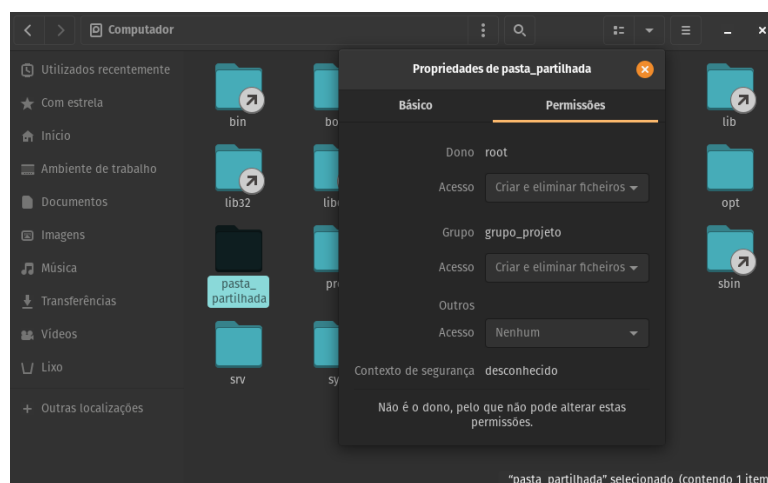


Figura 31: Pasta partilhada

Backup

Após a verificação dos argumentos, os diretórios do caminho, são criados com o comando “`mkdir`” e “-p”.

```
mkdir -p "$destino_backup"
```

Depois é definida uma variável com a data atual (timestamp) que vai ser utilizada para o nome do ficheiro.

```
# Data para nomear arquivos de backup  
data=$(date +%Y-%m-%d_%H-%M-%S')
```

Para cada argumento (diretório), é obtido o nome do último diretório com ajuda da função “`basename`”, cria o nome do ficheiro com o nome do diretório original. O comando “`tar`” serve para criar ficheiros “.tar”, que podem ser comprimidos com “`gzip`” através da opção “-z”, originando ficheiros com extensão “.tar.gz”. A opção “-c” indica que se pretende criar um novo arquivo, enquanto “-f” define o nome e o local do ficheiro de saída. A opção -C “\$(dirname "\$dir")” muda o diretório de trabalho para o diretório pai

de “\$dir”, o que evita que o caminho completo seja incluído no interior do ficheiro “.tar.gz”. Por fim, ao indicar apenas “\$nome_dir” como o conteúdo a incluir.

```
for dir in "$@"; do
    if [ -d "$dir" ]; then
        nome_dir=$(basename "$dir")
        arquivo_backup="${destino_backup}/${nome_dir}_backup_${data}.tar.gz"
        tar -czf "$arquivo_backup" -C "$(dirname "$dir")" "$nome_dir"
        echo "Backup do diretório '$dir' criado em '$arquivo_backup'."
    else
        echo "Diretório '$dir' não existe. Ignorando."
    fi
done
```

Monitorização

O código começa por verificar a existência do argumento, de forma semelhante aos exemplos anteriores. Em seguida, é criada uma variável que armazena o argumento (diretório de destino) e o diretório é criado caso ainda não exista.

```
dir_logs="$1"
mkdir -p "$dir_logs"
```

É criado um ficheiro de monitorização com a data e hora atuais, onde os dados são inseridos através de redirecionamento (>).

```
log_file="${dir_logs}/monitorizacao_$(date +%Y-%m-%d_%H-%M-%S).log"
{
    echo "=== Monitorização do Sistema em $(date) ==="
    echo ""
    echo "--- Uso do Disco ---"
    df -h
    echo ""
    echo "--- Uso da CPU ---"
    # Média de carga do sistema
    uptime
    echo ""
    echo "--- Uso da RAM ---"
    free -h
    echo ""
    echo "--- Processos principais (top 5 por uso CPU) ---"
```

```
ps -eo pid,comm,%cpu,%mem --sort=-%cpu | head -n 6  
echo ""  
} > "$log_file"
```

Ao executar o script, o ficheiro de registo da monitorização foi criado na pasta seleccionada.

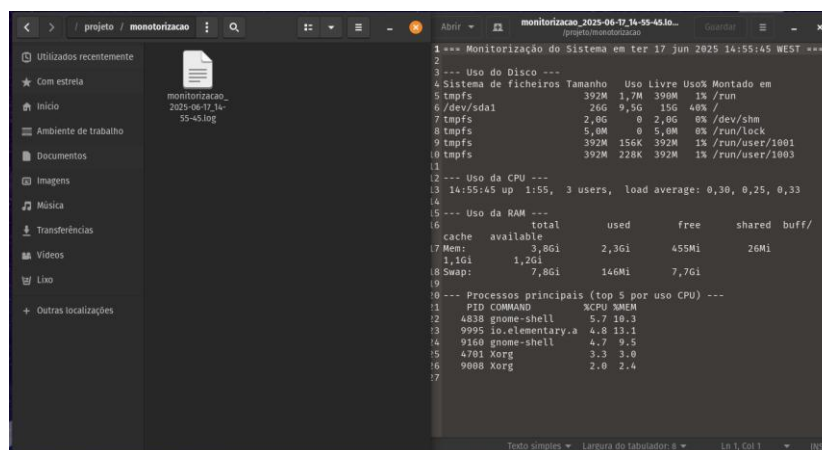


Figura 32: Monitorização

Programas em C

O docente propôs a resolução de três exercícios em linguagem C:

- Programa para criar um processo filho e comunicar com ele via pipe.
- Programa para simular 2 algoritmos de escalonamento (ex.: FCFS e Round-Robin).
- Programa que responda a sinais (SIGINT, SIGTERM, etc.) e escreva logs.

Algoritmos de escalonamento

Fizemos dois algoritmos de simulação de escalonamento:

- FCFS (First Come First Served): primeiro a chegar é o primeiro a ser executado;
- Round Robin: cada processo recebe uma fatia de tempo igual (com quantum);

FCFS (First Come First Served)

Tal como em qualquer algoritmo escrito em linguagem C, o código inicia-se pela inclusão das bibliotecas necessárias. É também definida uma constante que representa o número máximo de processos.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_PROCESSOS 100
```

É definida uma estrutura para representar os processos, que inclui o tempo de chegada e o tempo de execução.

```
typedef struct {
    int tempo_chegada;
    int tempo_execucao;
} Processo;
```

A seguinte função é utilizada para ordenar os processos por ordem de chegada, através de dois ciclos encadeados, onde se o tempo de chegada do primeiro for maior do que o do segundo trocam de posições assim sucessivamente até ficarem todos ordenados.

```
void ordenar_por_tempo_chegada(Processo p[], int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = i + 1; j < n; j++) {  
            if (p[i].tempo_chegada > p[j].tempo_chegada) {  
                Processo aux = p[i];  
                p[i] = p[j];  
                p[j] = aux;  
            }  
        }  
    }  
}
```

A função FCFS recebe como parâmetros os processos e a quantidade total de processos. É definida uma variável local para o tempo atual que inicia em zero, dentro de um ciclo for são percorridos todos os processos, começando pelo que chegou primeiro, se o tempo atual for menor que o tempo de chegada então passamos para o tempo de chegada do processo atual, a cada iteração é somado ao tempo atual o tempo de execução.

```
void FCFS(Processo p[], int n) {  
    int tempo_atual = 0;  
    for (int i = 0; i < n; i++) {  
        if (tempo_atual < p[i].tempo_chegada)  
            tempo_atual = p[i].tempo_chegada;  
        printf("Processo %d começa em %d\n", i + 1, tempo_atual);  
        tempo_atual += p[i].tempo_execucao;  
    }  
}
```

A função *main* começa por perguntar ao utilizador o número de processos, se o valor for inferior que um ou superior que o limite (100) dá erro.

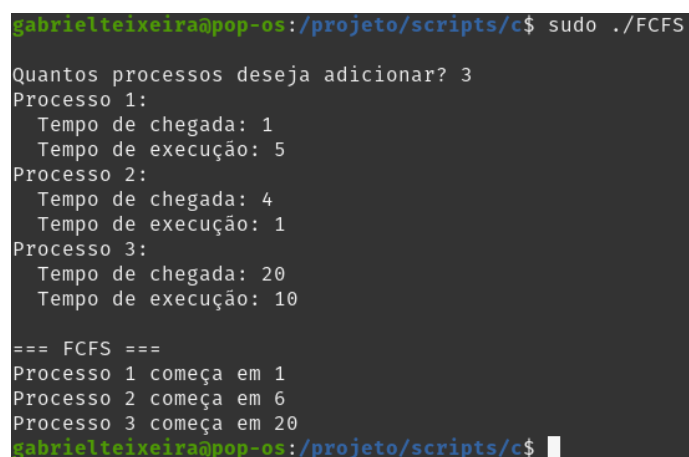
```
int main() {  
    Processo processos[MAX_PROCESSOS];  
  
    int n;  
    printf("Quantos processos deseja adicionar? ");
```

```
scanf("%d", &n);  
if (n <= 0 || n > MAX_PROCESSOS) {  
    printf("Número inválido (1 a %d).\n", MAX_PROCESSOS);  
    return 1;  
}
```

Depois são pedidos os dados dos processos, ordenados e no fim é executada a função *FCFS*.

```
for (int i = 0; i < n; i++) {  
    printf("Processo %d:\n", i + 1);  
    printf(" Tempo de chegada: ");  
    scanf("%d", &processos[i].tempo_chegada);  
    printf(" Tempo de execução: ");  
    scanf("%d", &processos[i].tempo_execucao);  
}  
  
ordenar_por_tempo_chegada(processos, n);  
  
printf("\n=== FCFS ===\n");  
FCFS(processos, n);  
  
return 0;  
}
```

Esta imagem mostra um exemplo de interação com o algoritmo.



```
gabrielteixeira@pop-os: /projeto/scripts/c$ sudo ./FCFS  
Quantos processos deseja adicionar? 3  
Processo 1:  
    Tempo de chegada: 1  
    Tempo de execução: 5  
Processo 2:  
    Tempo de chegada: 4  
    Tempo de execução: 1  
Processo 3:  
    Tempo de chegada: 20  
    Tempo de execução: 10  
  
=== FCFS ===  
Processo 1 começa em 1  
Processo 2 começa em 6  
Processo 3 começa em 20  
gabrielteixeira@pop-os: /projeto/scripts/c$
```

Figura 33: Algoritmo de escalonamento – FCFS

Round Robin

O algoritmo é semelhante ao anterior com algumas pequenas alterações, a estrutura é composta por mais um atributo, *tempo_restante*.

```
typedef struct {  
    int tempo_chegada;  
    int tempo_execucao;  
    int tempo_restante;  
} Processo;
```

A função recebe três parâmetros os processos, a quantidade e o *quantum*, as variáveis do tempo atual e processos terminados são definidas, enquanto os processos terminados forem inferiores à quantidade total de processos, vai para cada processo é verificado se o tempo restante é maior que zero e se já chegou, se ambos forem verdadeiros, vai ser verificado se o tempo restante é menor que o quanto (para impedir que o tempo restante passe a valores negativos), o menor valor é subtraído do tempo restante e somado ao tempo atual. Se o processo tiver sido terminado é somado um aos processos terminados. Caso no tempo nenhum tiver sido executado, o tempo avança uma unidade.

```
void round_robin(Processo p[], int n, int quantum) {  
    int tempo_atual = 0;  
    int processos_terminados = 0;  
  
    printf("\n=== Round Robin (Quantum = %d) ===\n", quantum);  
  
    while (processos_terminados < n) {  
        int executou = 0;  
  
        for (int i = 0; i < n; i++) {  
            if (p[i].tempo_restante > 0 && p[i].tempo_chegada <= tempo_atual) {  
                int exec = (p[i].tempo_restante < quantum) ? p[i].tempo_restante : quantum;  
                printf("Processo %d executa de %d a %d\n", i + 1, tempo_atual, tempo_atual  
+ exec);  
  
                tempo_atual += exec;  
                p[i].tempo_restante -= exec;  
                executou = 1;  
  
                if (p[i].tempo_restante == 0) {
```



```
        processos_terminados++;  
        printf("Processo %d terminou em %d\n", i + 1, tempo_atual);  
    }  
}  
}  
if (!executou) {  
    tempo_atual++;  
}  
}  
}
```

Exemplo de interação com o algoritmo:

```
gabrielteixeira@pop-os:~/projeto/scripts/c$ sudo ./Round_Robin  
Número de processos: 3  
Processo 1:  
  Tempo de chegada: 1  
  Tempo de execução: 10  
Processo 2:  
  Tempo de chegada: 3  
  Tempo de execução: 5  
Processo 3:  
  Tempo de chegada: 3  
  Tempo de execução: 2  
Quantum: 2  
  
=== Round Robin (Quantum = 2) ===  
Processo 1 executa de 1 a 3  
Processo 2 executa de 3 a 5  
Processo 3 executa de 5 a 7  
Processo 3 terminou em 7  
Processo 1 executa de 7 a 9  
Processo 2 executa de 9 a 11  
Processo 1 executa de 11 a 13  
Processo 2 executa de 13 a 14  
Processo 2 terminou em 14  
Processo 1 executa de 14 a 16  
Processo 1 executa de 16 a 18  
Processo 1 terminou em 18  
gabrielteixeira@pop-os:~/projeto/scripts/c$
```

Figura 34: Algoritmo de escalonamento - Round Robin

Sinais

A função main do algoritmo é responsável por abrir o ficheiro de *logs* e por chamar a função que regista uma entrada de log quando o programa é iniciado.

```
int main() {  
    log_file = fopen("log_sinais.txt", "a");  
    if (!log_file) {  
        perror("Erro ao abrir ficheiro de log");  
        return 1;  
    }  
    escrever_log("Programa iniciado");  
}
```

Quando este processo recebe um sinal *SIGINT* ou *SIGTERM*, em vez de terminar imediatamente, executa a função *handler_sinal()*.

```
signal(SIGINT, handler_sinal);  
signal(SIGTERM, handler_sinal);
```

O algoritmo mantém-se sempre à espera de um sinal; quando o recebe, entra em pausa, mas como o ciclo é infinito, volta a ficar em espera após a pausa.

```
while(1) {  
    pause();  
}
```

A função *handler_sinal* recebe o sinal e através de um *switch case* identifica o tipo de sinal chama a função para escrever nas *logs* e no caso de ser um *SIGTERM* fecha o ficheiro e termina o programa.

```
void handler_sinal(int sig) {  
    switch(sig) {  
        case SIGINT:  
            escrever_log("Recebido SIGINT (Ctrl+C)");  
            break;  
        case SIGTERM:  
            escrever_log("Recebido SIGTERM (Pedido de término)");  
            fclose(log_file);  
            printf("Terminar programa.\n");  
            exit(0);  
            break;  
        default:  
            escrever_log("Recebido sinal desconhecido");  
    }  
}
```

A função `escrever_log` serve para registar mensagens num ficheiro de log, associando a cada entrada a data e hora exatas em que ocorreu. Utiliza a função `time` para obter o tempo atual e `ctime` para convertê-lo numa representação legível. Remove o carácter de nova linha (`\n`) dessa `string` para manter o formato do log numa única linha por entrada. Em seguida, escreve a mensagem formatada no ficheiro através de `fprintf`.

```
void escrever_log(const char *mensagem) {  
    time_t agora = time(NULL);  
    char *tempo_str = ctime(&agora);  
    tempo_str[strcspn(tempo_str, "\n")] = '\0';  
  
    fprintf(log_file, "[%s] %s\n", tempo_str, mensagem);  
}
```

Aqui está um exemplo onde executámos os dois sinais previstos.

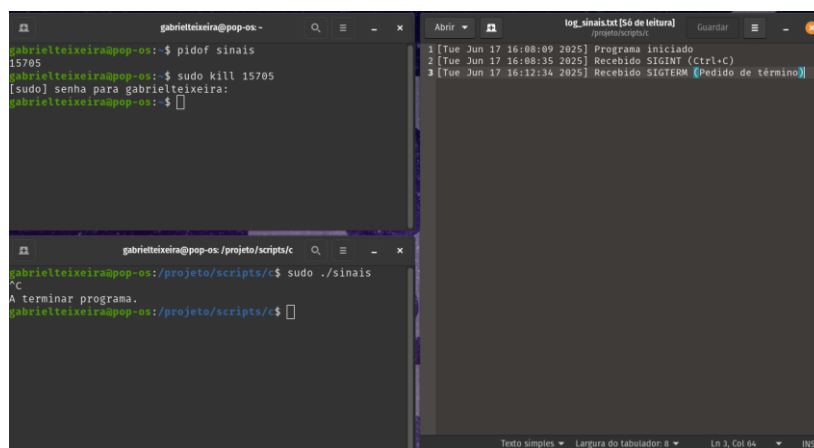


Figura 35: Sinais

Processo filho

Neste programa, optámos por complementar um exercício anterior realizado com scripts, no qual era necessário executar dois ficheiros: um para criar grupos e outro para criar utilizadores. Este programa permite introduzir um utilizador e um grupo associados a um único diretório, e, de seguida, associar o utilizador ao grupo.

O algoritmo inicia-se com a inclusão das bibliotecas necessárias, destaque para as bibliotecas `<unistd.h>` e `<sys/wait.h>`, que são fundamentais para criar e manipular processos filhos. O primeiro define funções como `fork()` e `pipe()`, o segundo permite utilizar elementos como o `wait()` e `WIFEXITED(status)`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
```

A função *main* recebe os argumentos que devem ser o *username*, a *password*, o grupo, as permissões do grupo, as permissões dos outros e o diretório.

Dentro da *main*, o código começa pela criação de um `pipe()`, com dois descritores: um para leitura `fd[0]` e outro para escrita `fd[1]`

```
int fd[2];
if (pipe(fd) == -1) {
    perror("Erro ao criar pipe");
    return 1;
}
```

Cria um processo filho com o `fork`, este comando clona os dados do pai para criar o novo processo, mas a memória é separada. Então quando o pai executa “`pid_t pid_pipe = fork();`” vai obter o *pid* do filho que vai maior do que 0, logo vai para o *else if* e executa, já o filho recebe 0, pois este não precisa de saber nenhum *pid* apenas tem que saber que é filho de outro, assim executa a primeira parte da condição.

```
pid_t pid_pipe = fork();
```

O processo filho, fecha a escrita, cria um variável para armazenar os dados, lê os dados com o descritor de leitura e escreve na tela a mensagem que recebeu. Depois fecha a leitura e termina o processo filho.

```
if (pid_pipe == 0) {  
    close(fd[1]);  
    char buffer[100];  
    read(fd[0], buffer, sizeof(buffer));  
    printf("Filho recebeu mensagem: %s\n", buffer);  
    close(fd[0]);  
    exit(0);  
}
```

O processo pai fecha a leitura, escreve uma mensagem fecha a escrita e espera que o processo filho termine.

```
else if (pid_pipe > 0) {  
    close(fd[0]);  
    char mensagem[] = "Início da operação\n";  
    write(fd[1], mensagem, strlen(mensagem) + 1);  
    close(fd[1]);  
    wait(NULL);  
}
```

Quando o *fork* retorna um valor negativo é porque é porque o sistema não conseguiu criar o processo filho

```
} else {  
    perror("fork falhou");  
    return 1;  
}
```

A parte de criação do grupo de do utilizador, começa pela verificação da quantidade de argumentos. Se for menor do que sete é porque faltam argumentos, importante notar que o *argv[0]* é o nome do programa por isso temos que somar uma unidade aos argumentos pretendidos (6 argumentos + 1 nome do programa). Se passar a referência cada referência dos argumentos é guardada em um ponteiro diferente.

```
if (argc < 7) {  
    printf("Uso:  %s  <utilizador>  <password>  <grupo>  <permissoes_grupo>  
<permissoes_outros> <diretorio>\n", argv[0]);
```

```
printf("Permissões: combinações de r, w, x (ex: rwx, rw, r, x)\n");  
return 1;  
}
```

```
char *utilizador = argv[1];  
char *password = argv[2];  
char *grupo = argv[3];  
char *perm_grupo_str = argv[4];  
char *perm_outros_str = argv[5];  
char *diretorio = argv[6];
```

As permissões são pedidas ao utilizador em formato de letras (ex: rwx), mas na criação são utilizados valores numéricos, então a função *letras_para_numero* faz a conversão.

```
int perm_grupo = letras_para_numero (perm_grupo_str);  
int perm_outros = letras_para_numero(perm_outros_str);
```

A variável é criada com três posições (2 para as permissões e um para o “\0”) a função *snprintf* escreve uma *string* formatada para dentro de uma variável.

```
char perm_final[3];  
snprintf(perm_final, sizeof(perm_final), "%d%d", perm_grupo, perm_outros);
```

São criados dois argumentos, um para o grupo e outro para o utilizador. Estes argumentos são ponteiros que recebem a referência para os argumentos, incluindo o caminho para o script e o *NULL* que identifica o fim de argumentos (obrigatório para utilizar a função *execvp*). Chama a função *executar_script*.

```
char *args_grupo[] = {"../bash/criar_grupo.sh", grupo, perm_final, diretorio, NULL};  
if (executar_script(args_grupo[0], args_grupo) != 0) {  
    printf("Erro ao criar grupo %s\n", grupo);  
    return 1;  
}  
  
char *args_utilizador[] = {"../bash/criar_utilizadores.sh", utilizador, password, grupo, NULL};  
if (executar_script(args_utilizador[0], args_utilizador) != 0) {  
    printf("Erro ao criar utilizador %s\n", utilizador);  
    return 1;  
}  
}
```

A função que converte letras para número teve duas versões, a primeira comparava a *string* enviada com todas as combinações possíveis, retornando respetivo valor.

```
int letras_para_numero(const char *letras) {
    if (strcmp(letras, "") == 0) return 0;
    else if (strcmp(letras, "x") == 0) return 1;
    else if (strcmp(letras, "w") == 0) return 2;
    else if (strcmp(letras, "wx") == 0 || strcmp(letras, "xw") == 0) return 3;
    else if (strcmp(letras, "r") == 0) return 4;
    else if (strcmp(letras, "rx") == 0 || strcmp(letras, "xr") == 0) return 5;
    else if (strcmp(letras, "rw") == 0 || strcmp(letras, "wr") == 0) return 6;
    else if (strcmp(letras, "rwx") == 0 || strcmp(letras, "rxw") == 0 || strcmp(letras, "wrx")
    == 0 ||
        strcmp(letras, "wxr") == 0 || strcmp(letras, "xrw") == 0 || strcmp(letras, "xwr") ==
    0)
        return 7;
    else {
        printf("Permissão inválida: '%s'. Use combinações de r, w, x.\n", letras);
        exit(1);
    }
}
```

Nós achámos estranho as permissões não serem sequenciais, até que percebemos que “wx” era a soma de “x” e de “w”, então surgiu esta versão que passa caracter a caracter e vai somando os valores.

```
int letras_para_numero(const char *letras) {
    int perm = 0;
    for (size_t i = 0; i < strlen(letras); i++) {
        switch (letras[i]) {
            case 'r': perm += 4; break;
            case 'w': perm += 2; break;
            case 'x': perm += 1; break;
            case '-': perm += 0; break;
            default:
                printf("Caractere de permissão inválido: %c\n", letras[i]);
                exit(1);
        }
    }
    return perm;
}
```

A função para executar os *scripts* recebe o caminho e os argumentos, cria um processo filho que através da função *execvp* que substitui o processo pelo indicado, passando os argumentos, se não funcionar escreve uma mensagem de erro no terminal (o *perror* é utilizado para termos acesso ao erro) e o processo é fechado.

```
int executar_script(char *script, char *args[]) {  
    pid_t pid = fork();  
    if (pid == 0) {  
        execvp(script, args);  
        perror("Erro ao executar script");  
        exit(1);  
    }  
}
```

Já o processo pai, espera o processo filho terminar e armazena estado de término em uma variável (o "0" garante que o processo pai vai esperar o filho concluir). Através do comando *WIFEXITED* é verificado se ocorreu algum erro, se não é indicado é extraído o estado a partir da função *WEXITSTATUS*, se deu erro retorna "-1".

```
else if (pid > 0) {  
    int status;  
    waitpid(pid, &status, 0);  
    return WIFEXITED(status) ? WEXITSTATUS(status) : -1;  
}
```


Conclusão

Ao longo deste trabalho, conseguimos desenvolver um conjunto de ferramentas práticas que automatizam tarefas essenciais na gestão de sistemas Unix/Linux, o que demonstra claramente o impacto da programação no quotidiano da administração de sistemas. A criação dos scripts em Bash e dos programas em C permitiu-nos enfrentar desafios reais, como a gestão de permissões, a comunicação entre processos e a monitorização de recursos, reforçando competências técnicas importantes.

Este projeto também evidenciou algumas dificuldades inerentes, como o rigor necessário na definição de permissões e na manipulação de processos, o que contribuiu para uma aprendizagem significativa e para o desenvolvimento do pensamento crítico relativamente à segurança e eficiência do sistema.

Em suma, este trabalho não só consolidou conceitos fundamentais da unidade curricular de Sistemas Operativos, como também proporcionou uma experiência prática valiosa, preparando-nos para futuras situações profissionais onde a automatização e o controlo do sistema são indispensáveis.

Bibliografia

W3S. Basic Bash Syntax. Acedido a 14 de junho de 2025 em:

https://www.w3schools.com/bash/bash_syntax.php

Hira, Zaira. (2023). Freecodecamp: Bash Scripting Tutorial – Linux Shell Script and Command Line for Beginners. Acedido a 15 de junho de 2025 em:

<https://www.freecodecamp.org/news/bash-scripting-tutorial-linux-shell-script-and-command-line-for-beginners/>