

”

E-fólio A | Folha de resolução para E-fólio

UNIDADE CURRICULAR: Sistemas Operativos

CÓDIGO: 21111

DOCENTE: Paulo Shirley

A preencher pelo estudante

NOME: Luís Carlos Crispim Pereira

N.º DE ESTUDANTE: 2300163

CURSO: LEI – Licenciatura em Engenharia Informática

DATA DE ENTREGA: 07/04/24

TRABALHO / RESOLUÇÃO:

Introdução:

Este relatório descreve a implementação do programa 'fdup.c', seguindo as orientações descritas no EFolio A, sendo que aqui irei reportar as minhas escolhas em termos de código e também demonstrar através de testes o resultado conforme esperado. O programa 'fdup.c', foi concebido em linguagem C, como uma ferramenta de procura de duplicatas num determinado diretório num sistema Unix (neste caso Ubuntu em VM num Mac). Ele realiza uma série de operações, incluindo a procura de arquivos com extensão específica, a classificação desses arquivos com base em determinados critérios e a identificação de duplicatas dentro desses arquivos. O programa faz uso extensivo de chamadas de sistema, como fork, execvp e wait, para criar e gerir processos.

Opções:

Bibliotecas: As cinco bibliotecas incluídas neste programa têm funções específicas que são usadas para fornecer acesso a diversas funcionalidades necessárias para manipulação de arquivos, processos e entrada/saída de dados, que são essenciais para a implementação do programa:

1. **stdio.h:** Usada para entradas e saídas padrão. Ela fornece funções como printf, fprintf, scanf, fscanf, getchar, putchar, entre outras.
2. **stdlib.h:** Fornece funções como malloc, calloc, realloc e free para alocação e liberação de memória dinâmica. Também inclui funções como exit e abort para sair ou abortar um programa. Embora esta biblioteca não seja utilizada no programa, optei por manter pois é uma biblioteca "padrão".
3. **unistd.h:** Faculta o acesso a um grande número de funções do sistema operativo. Ela inclui funções para manipulação de arquivos (open, close, read, write), manipulação de diretórios (opendir, readdir, closedir), controle de processos (fork, exec, wait, exit) e manipulação de pipelines (pipe, dup, dup2).

4. `sys/wait.h`: Proporciona as declarações para a função `wait`, usada para aguardar o término de um processo filho.
5. `fcntl.h`: Fornece funções e constantes para a manipulação de descritores de arquivo. Ela inclui constantes para definir e controlar propriedades de arquivos abertos, como modo de acesso (`O_RDONLY`, `O_WRONLY`, `O_RDWR`), flags de criação (`O_CREAT`, `O_TRUNC`, `O_APPEND`) e operações de bloqueio (`O_NONBLOCK`, `O_EXCL`).

Abordagem:

Optei por uma programação modular, para facilitar a sua leitura e interpretação, e abstração funcional, sendo que cada secção é apenas responsável por uma parte do programa.

O programa começa pela verificação de argumentos é diferente de 3 (`argc != 3`) e caso o seja, o utilizador não iniciou o programa corretamente e o programa encerra demonstrando a mensagem de como o deveria usar.

No segundo passo o programa verifica se existe ou não o diretório especificado (`argv[1]`) pelo utilizador e caso não exista encerra demonstrando uma mensagem a informar a inexistência.

No passo seguinte é criado um array de ponteiros `cmds[]` para todos os comandos a executar (`find`, `sort`, `uniq`, `awk`) e um array de ponteiros bidimensional `args[][]` para os argumentos dos comandos. Esta opção permite organizar os comandos e seus argumentos de forma eficiente para passá-los no fim para a função ‘`exec`’ pretendida. Foi também criado um array de ponteiros `proc[]` (para os processos B C D E, pois, os mesmos não estão por ordem A B C D). E por fim outro array de ponteiros `adress[]` com os paths (`find`, `sort`, `uniq`, `awk`).

Neste passo são criados os ficheiros temporários, com os nomes exigidos (`tmp1.txt`, `tmp2.txt`, `tmp3.txt`) optei por criar os ficheiros de forma “forçada”, pois assim caso os mesmos não sejam criados por algum motivo o programa encerra neste ponto demonstrando a mensagem de erro (Erro ao criar ficheiros temporários).

Nesta fase do código criei um ciclo for que é responsável pela criação de 4 processos filhos sequenciais com iteração i, tem aqui também uma pequena verificação que caso o processo filho não seja criado o ID retornado será -1 e caso pid seja -1 irá terminar o programa, reportando o erro numa mensagem. Cada vez que o processo filho inicia é impressa a mensagem solicitada no enunciado. "Processo X: PID=xxx PPID=xxx". A parte seguinte do código (ainda dentro do ciclo for) é responsável pela gestão do stdin e stdout (onde uso o array dos ficheiros temporários anteriormente criados) e onde são utilizadas as 4 funções da família exec() (execl, execlp, execv, execvp) onde para tal são utilizados todos os arrays anteriormente criados (args[], cmds[], adress[]). Entre cada iteração do i existe um wait para o “pai” esperar pelo fim do processo “filho”.

Fase final do programa uma nova iteração de ciclo for para encerrar todos os ficheiros temporários criados.

Testes:

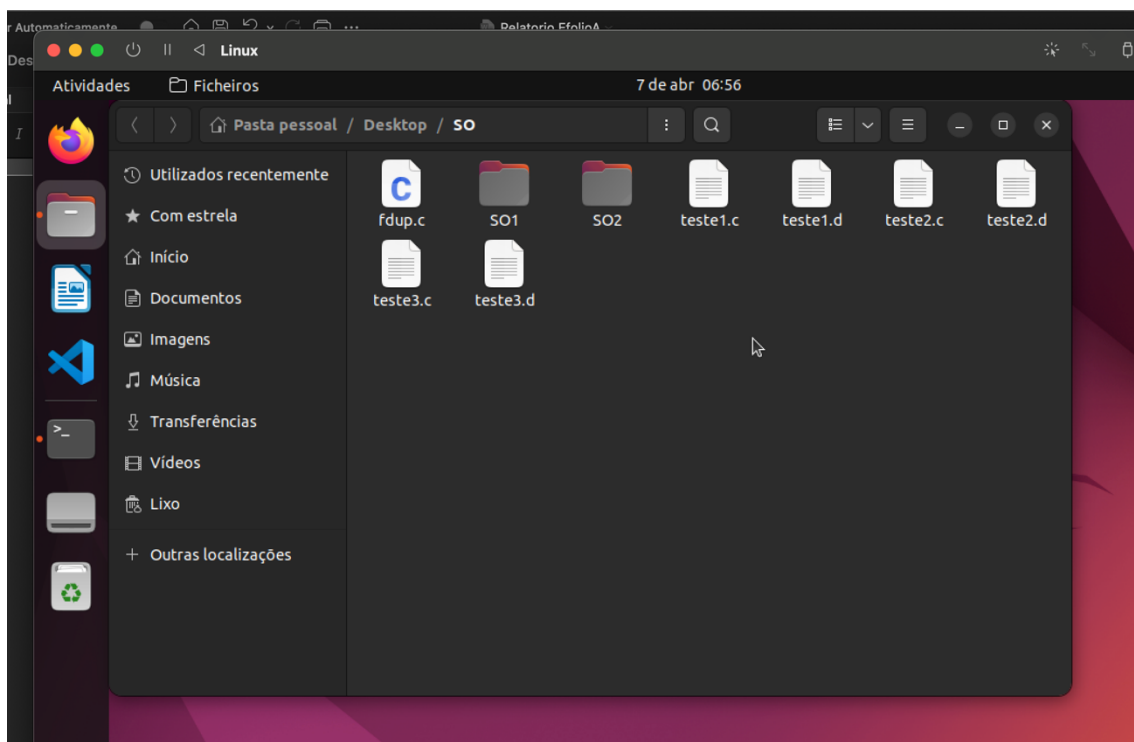


Figura 1 – Diretoria Raiz (Pasta SO no Desktop onde testei o programa)

Erros possíveis de acontecer que consegui replicar:

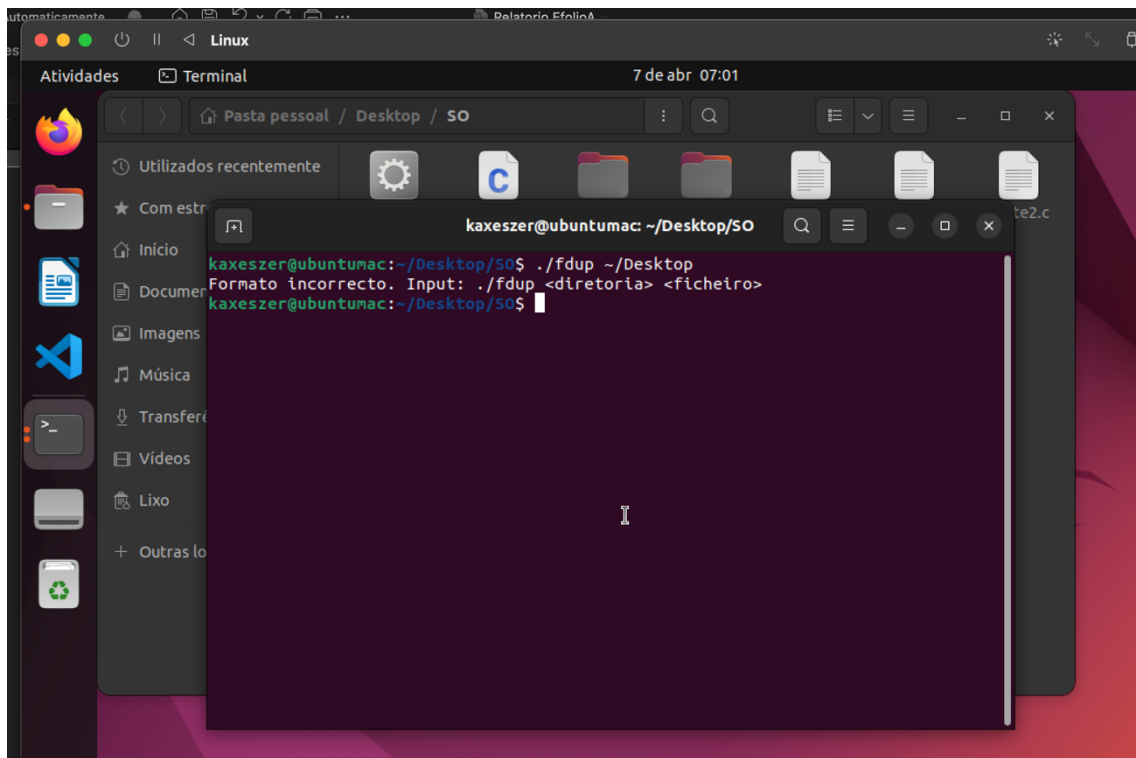


Figura 2 – Utilizador não colocou o número de argumentos correto.

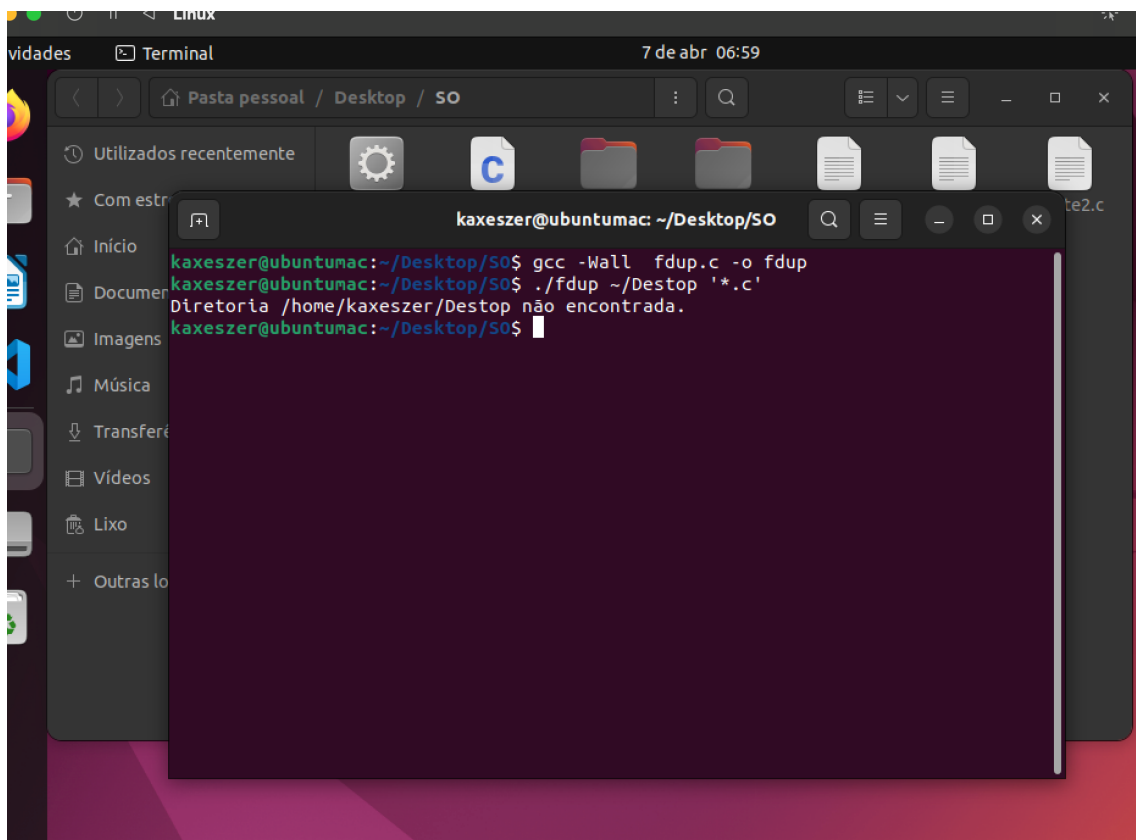


Figura 3 – Diretoria Inexistente – Utilizador não colocou path correto.

Resultados esperados com sucesso:

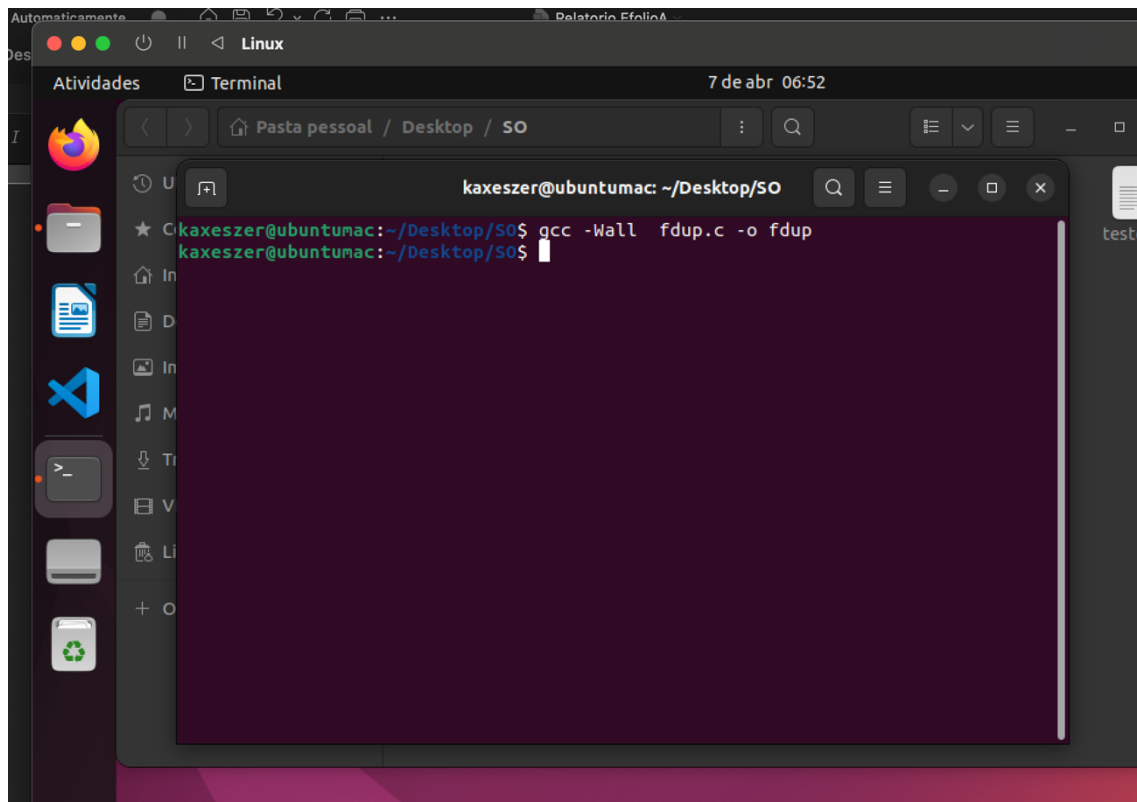


Figura 4 - Programa compila e não produz avisos (warnings) com `gcc -Wall`

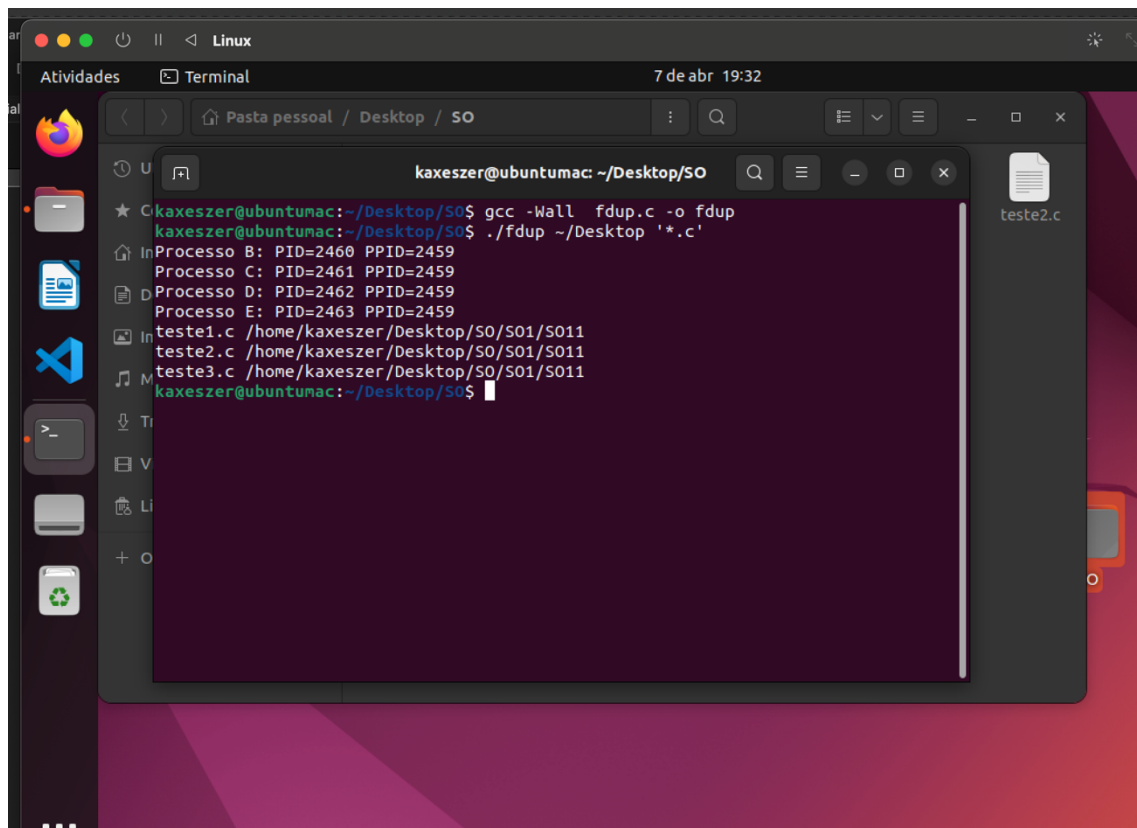


Figura 5 – Programa compila, e funciona demonstra em consola a mensagem "Processo X: PID=xxx PPID=xxx", e imprime o resultado final na forma de nome de ficheiro e respetiva pathname em cada linha.

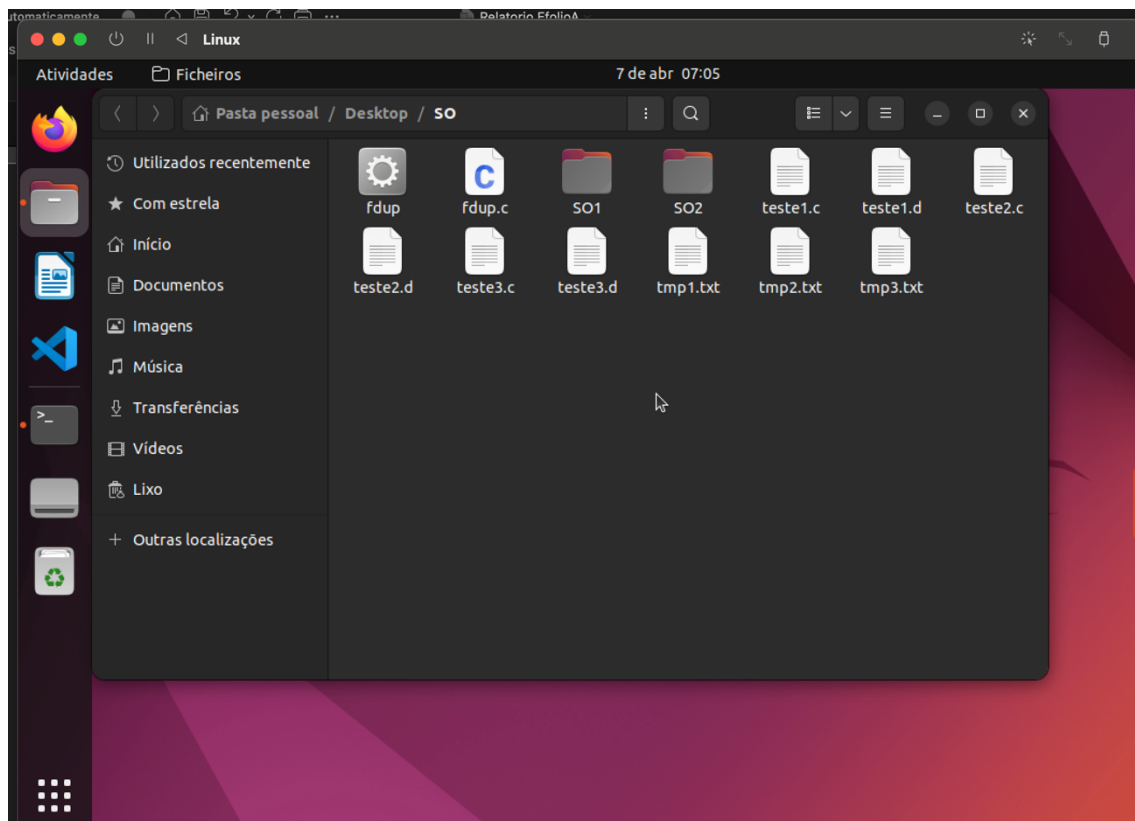


Figura 6 – Programa cria ficheiros tmp1.txt, tmp2.txt, tmp3.txt.

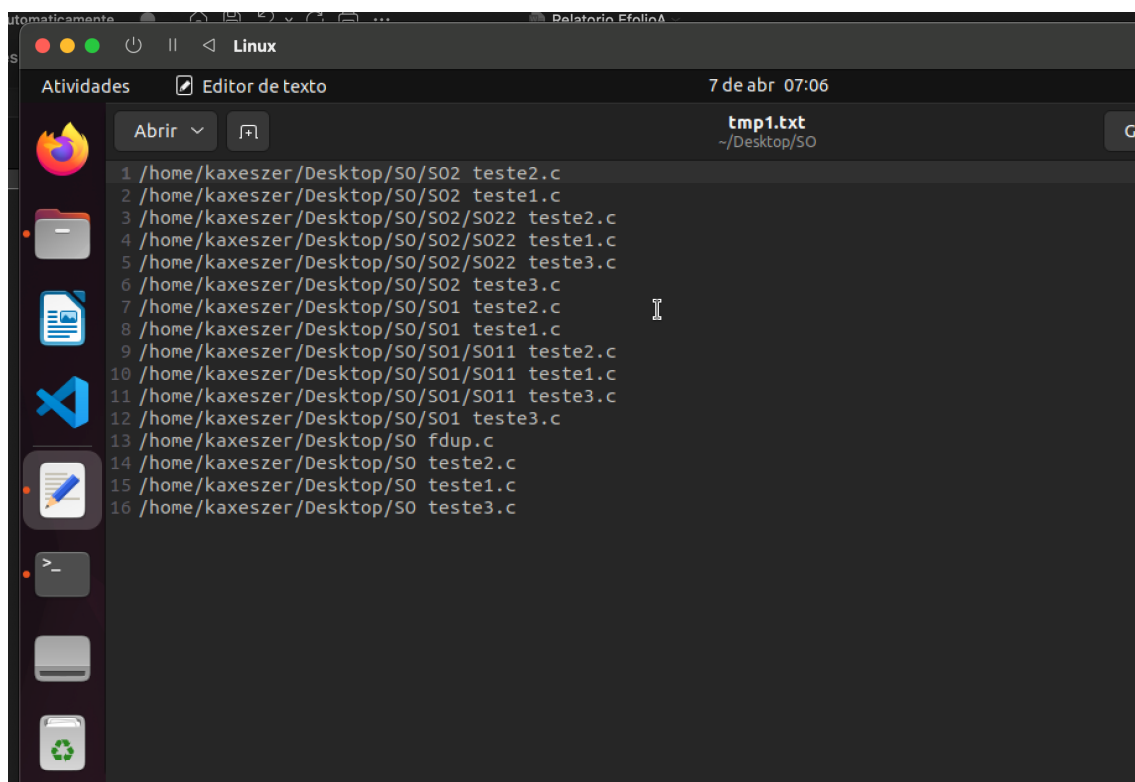
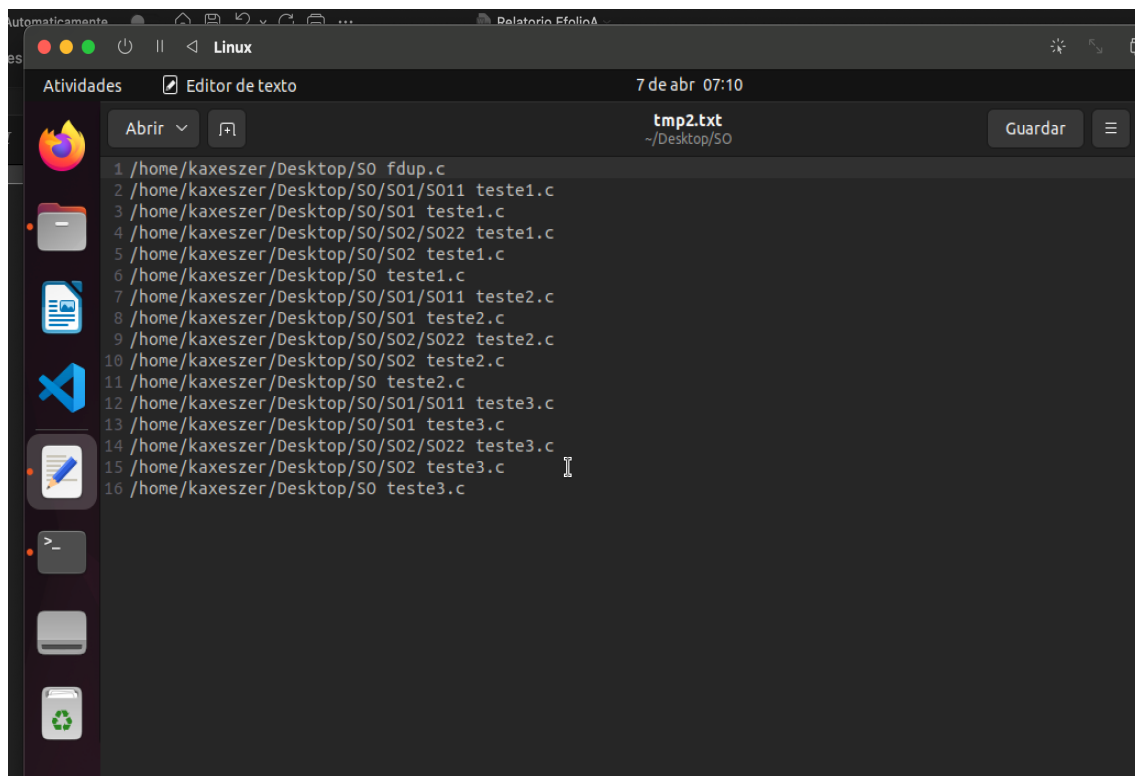
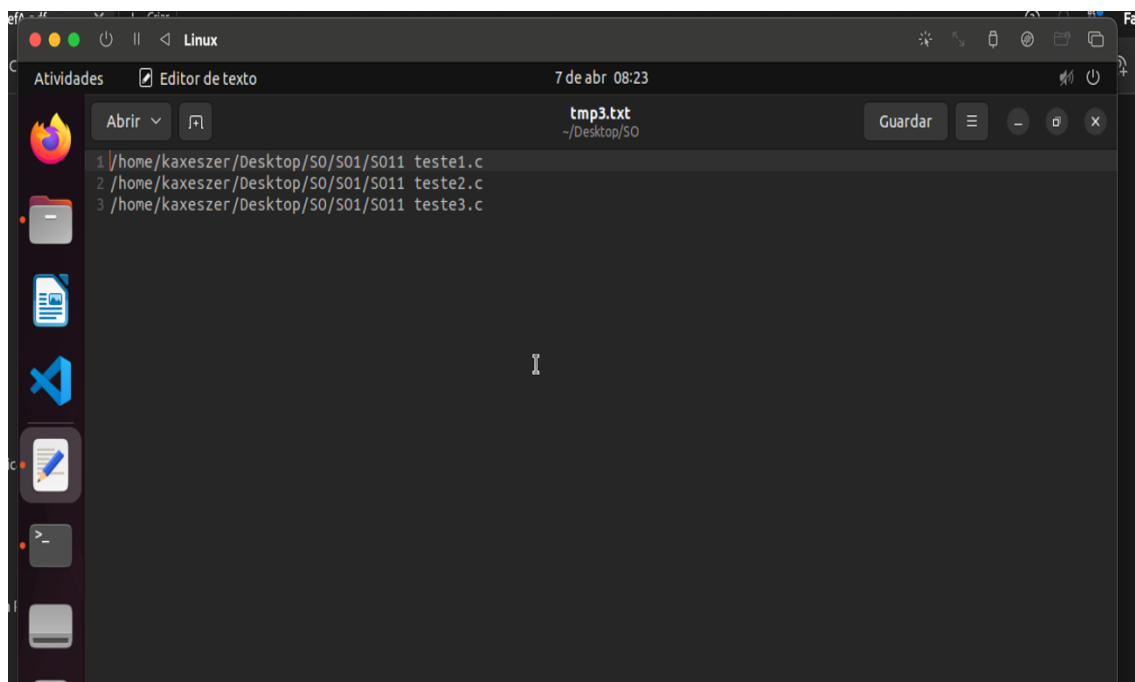


Figura 7 – tmp1.txt listou todos os ficheiros '*.c' desde a diretoria raiz ~/Desktop. (Resultado find)



```
1 /home/kaxeszer/Desktop/S0 fdup.c
2 /home/kaxeszer/Desktop/S0/S01/S011 teste1.c
3 /home/kaxeszer/Desktop/S0/S01 teste1.c
4 /home/kaxeszer/Desktop/S0/S02/S022 teste1.c
5 /home/kaxeszer/Desktop/S0/S02 teste1.c
6 /home/kaxeszer/Desktop/S0 teste1.c
7 /home/kaxeszer/Desktop/S0/S01/S011 teste2.c
8 /home/kaxeszer/Desktop/S0/S01 teste2.c
9 /home/kaxeszer/Desktop/S0/S02/S022 teste2.c
10 /home/kaxeszer/Desktop/S0/S02 teste2.c
11 /home/kaxeszer/Desktop/S0 teste2.c
12 /home/kaxeszer/Desktop/S0/S01/S011 teste3.c
13 /home/kaxeszer/Desktop/S0/S01 teste3.c
14 /home/kaxeszer/Desktop/S0/S02/S022 teste3.c
15 /home/kaxeszer/Desktop/S0/S02 teste3.c
16 /home/kaxeszer/Desktop/S0 teste3.c
```

Figura 8 – tmp2.txt apresenta os resultados de tmp1.txt ordenados. (Resultado sort)



```
1 /home/kaxeszer/Desktop/S0/S01/S011 teste1.c
2 /home/kaxeszer/Desktop/S0/S01/S011 teste2.c
3 /home/kaxeszer/Desktop/S0/S01/S011 teste3.c
```

Figura 9 – tmp3.txt apresenta os duplicados (resultado uniq)

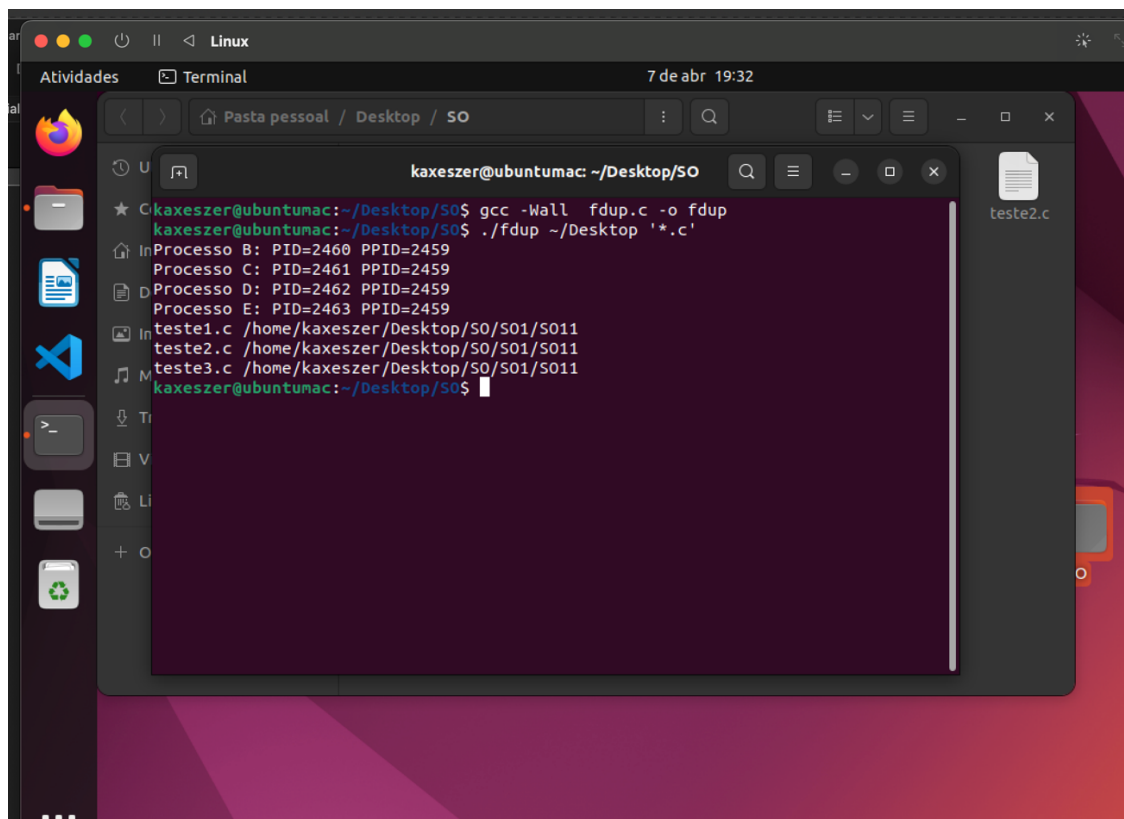


Figura 10 – Impresso em consola o resultado final do programa. (Resultado awk)

Nota final: Este programa foi desenvolvido com a interpretação que find deve apresentar os resultados em tmp1.txt, sort em tmp2.txt, uniq em tmp3.txt e awk em consola. Decidi a resolução sempre com ciclos, e criação de ponteiros pois assim o código torna-se modular e poderá ser mais facilmente usado em aplicações futuras sendo apenas necessário “editar” os ponteiros e/ou iterações.