Indice

Appunti completati l'11/01/25 da Elena, sono una guida ottima per affrontare l'esame pratico di sistemi operativi.
Consigli: che usiate l'intelligenza artificiale o meno è importante capire cosa si sta scrivendo, i problemi spesso sono ricorrenti quindi se sapete come svolgere i problemi passati non avrete molti problemi.

C

Gli import e librerie utili per C

Consiglio: importare le librerie principali usate per esercizi e poi levare quelle non utili

```c
#include <stdio.h>
#include <stdlib.h>
#include <err.h>

#include <string.h>

#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>

#include <fcntl.h>
```

```c
#include <limits.h>
#include <linux/limits.h>
```

```c
#include <stdio.h>
```

Serve per usare gli standard Input/output

```c
#include <stdlib.h>
```

Standard Library, permette la gestione di funzioni comuni

```c
#include <limits.h>
```

Definisce i limiti dei tipi di dato interi, come `INT_MAX`, `INT_MIN` o `CHAR_BIT`

```c
#include <sys/types.h>
```

Serve per lavorare con informazioni sui file, quindi dimensioni di un file, permessi, e ***tipo di file***.

```c
#include <sys/stat.h>
```

Lo usiamo per ***informazioni e permessi del file***, possiamo ottenere determinate info su un file, creare directory e gestire permessi.

```c
#include <sys/inotify.h>
```

Questa libreria è una interfaccia Linux per ***monitorare eventi su filesystem in tempo reale,*** ci permette di sapere quando un file viene creato, modificato, cancellato, aperto o rinominato.

Le ***funzioni/strutture*** che contiene sono:

- `inotify_init()` serve per inizializzare inotify
- `inotify_add_watch()`
- `inotify_rm_watch()`
- `struct inotify_event()` è una struttura che contiene un inotify event

E tra alcuni degli ***eventi comuni*** ci sono:

- `IN_CREATE` mi notificano quando un file è stato creato
- `IN_DELETE` mi notificano quando un file viene eliminato
- `IN_MODIFY` mi notificano quando un file viene modificato
- `IN_MOVED_FROM`
- `IN_MOVED_TO`
  link alla documentazione

```c
#include <sys/wait.h>
```

Gestisce la ***sincronizzazione tra processi padre e figli***, per esempio per attendere la terminazione di un processo figlio o recuperarne lo stato di uscita.

- `wait()`
- `waitpid()`

```c
#include <string.h>
```

Ci permette di gestire le stringhe e la memoria

```c
#include <dirent.h>
```

Importante per lavorare con le ***directory***, lo usiamo per leggere il contenuto di una cartella, come file e sottocartelle

---

```c
#include <time.h>
```

Questa è la libreria standard C per la gestione del tempo, possiamo usarla per ottenere data e ora, posso anche misurarci tempo di esecuzione e convertire formati di tempo

---

```c
#include <utime.h>
```

La libreria serve per modificare i ***timestamp*** di un file, quindi ultimo accesso e ultima modifica.

```c
#include <fcntl.h>
```

Libreria POSIX per il controllo dei file descriptor, possiamo aprire i file con ***open()***, e specificare flag di apertura (readonly, writeonly, lettura e scrittura ecc...).

```c
#include <unistd.h>
```

Libreria POSIX per il controllo dei file descriptor, possiamo aprire i file con ***open()***, e specificare flag di apertura (readonly, writeonly, lettura e scrittura ecc...).

```c
#include <unistd.h>
```

Libreria POSIX per l'interazione con il sistema operativo, mi permette di fare operazioni di basso livello su file e gestire processi e file descriptor.

```c
#include <errno.h>
```

Gestisce il codice di errore globale impostato dalle funzioni di sistema, quando una syscall fallisce ritorna -1 e imposta `errno`

```c
#include <err.h>
```

Libreria per la gestione degli errori, posso usare `err(EXIT_FAILURE, "es")`, oppure con `EXIT_SUCCESS` nel codice.

Costruire un path nuovo

Abbiamo due metodi principali per costruire un nuovo path, uno usando `strcpy` e `strcat` (sconsigliato) e l'altro usando `snprintf`
Facciamo un esempio, vogliamo costruire il path `rootDir/entry->d_name`, prima di tutti dobbiamo creare un buffer `char newDir[PATH_MAX]` e la nostra `rootDir` pronta.

```c
char newDir[PATH_MAX];
char *rootDir = "/home/user";
struct dirent *entry;   // entry->d_name = "docs"
```

Usando `strcpy` e `strcat` il risultato viene, però in modo più lungo e non viene controllato che `newDir` sia abbastanza grande, causando possibili problemi di buffer overflow.

```c
strcpy(newDir, rootDir); //copio rootDir in newDir
strcat(newDir, "/"); //aggiungo "/"
strcat(newDir, entry->d_name); //aggiungo il nome del file
```

L'opzione migliore è usare `snprinf` (che stampa *dati formattati* nel buffer), che scrive tutto in un colpo solo con un massimo di `PATH_MAX` byte e termina sempre in `\0`.

```c
snprintf(newDir, PATH_MAX, "%s/%s", rootDir, entry->d_name);

//stritura originale
int snprintf(char *buffer, size_t n, const char *format-string, argument-list);
```

Funzioni ausiliari per liste

```c
typedef struct NodeList {
    void* data; //dato generico, se vogliamo farlo per esempio per inode scriverei ino_t inode; ->ino_t
tipo definito dal sistema
    struct NodeList* next; //nodo successivo
}NodeList;

typedef NodeList* NodeList_ptr; //testa della lista
typedef int (*CompareFunc)(void*, void*);


//funzione che mi returna 0 se non è nella lista e 1 se è nella lista
int isElementInList(NodeList* head, void* element, CompareFunc cmp) {
    if(head == NULL) {
        return 0;
    }else {
        NodeList* tmp = head;
        while (tmp!=NULL) {
            if (cmp(tmp->data, element)==0) {
                return 1; //trovato

            }
            tmp=tmp->next;
        }
```

```
        return 0; // non trovato
    }
}

void insertElement(NodeList_ptr* head, void* element) {
    NodeList_ptr newNode = malloc(sizeof(NodeList));
    if(newNode==NULL){
        return;
    }
    newNode->data = element;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }
    NodeList_ptr tmp = *head;
    while (tmp->next != NULL) {
        tmp=tmp->next;
    }
    tmp->next = newNode;
    return;
}
```

La ricerca ricorsiva all'interno di un file system

```
void visitDir(const char *rootDir) {
    DIR *dir = opendir(rootDir);
    if (!dir) err(EXIT_FAILURE, "opendir");

    struct dirent *entry;
    char path[PATH_MAX];

    while ((entry = readdir(dir)) != NULL) {
        if (!strcmp(entry->d_name, ".") || !strcmp(entry->d_name, ".."))
            continue; //evito loop
        //creo il nuovo path
        snprintf(path, PATH_MAX, "%s/%s", rootDir, entry->d_name);

        struct stat st;
        if (lstat(path, &st) < 0) //controllo di aver aperto correttamente il path
            err(EXIT_FAILURE, "stat");

        if (S_ISDIR(st.st_mode)) {
            visitDir(path); //chiamata ricorsiva per leggere dentro
        } else if (S_ISREG(st.st_mode)) {
            //mi trovo davanti ad un file regolare
        }else{
            //ecc se voglio altro
        }
    }
    closedir(dir);
}
```

Controllare l'i-node

Se voglio assicurarmi che il file che sto controllando non sia uno stesso file che ho già incontrato (magari con nome diverso), posso usare l'***i-node***
Lo controllo con `entry->d_ino`

Diversi tipi di entry->d_type che possono essere utili

- `DT_DIR` directory

- `DT_REG` file regolare
- `DT_LINK` link simbolico, o riferimento ad un file
- `DT_FIFO` named pipe
- `DT_SOCK` socket
- `DT_CHR` device a caratteri
- `DT_BLK` device a blocchi
- `DT_UNKNOWN` non so il tipo, NON è un errore

Alberi pathfile

Negli esercizi può venire richiesto di **esplorare, stampare o unire dei pathfile** (sotto forma di albero).
Per questo tipo di esercizio è necessario utilizzare il metodo di ricerca ricorsiva all'interno di un file system.

dall'esame 22/07/2022

> *Il programma da realizzare si chiama tree mostra un sottoalbero del file system.*
> *es data una directory A che contiene una sottodirectory B e un file C; B contiene i file E F e G:*
> *tree A dovrebbe produrre*
> *B*

```
*E*
*F*
*G*
```

> *C*

```c
typedef int fd_t; //alias per int
int indent; //tiene traccia del livello di profondità nella directory

void printTree(char *);

int main(int argc, char ** argv){
    if(argc !=2 ){ //controllo argomenti, ne voglio uno
        err(EXIT_FAILURE, "argv");
    }
    char * file=argv[1]; //mi salvo il file iniziale della directory
    indent=0;
    printTree(file); //chiamo la funzione ricorsiva
    exit(EXIT_SUCCESS);
}

void printTree(char * path){ //essenzialmente mi visita ricorsivamente la directory, e salva quanto siamo
"dentro"
    //apro la directory
    fd_t dirFd = openat(0, realpath(path, NULL), 0);
    //realpath(path, NULL) converte path in un percorso assoluto, e alloca la memoria automatiamente
    //openat(0, ..., 0) apre la directory e restituisce un file descriptor

    if(dirFd == -1){
        err(EXIT_FAILURE, "openat");
    }
    DIR * dir = fdopendir(dirFd); //questa conversione serve per usare readdir

    struct dirent * ent;
    while((ent = readdir(dir))!=NULL){ //legge una entry alla volta
        if(strncmp(ent->d_name, ".", 2)==0 || strncmp(ent->d_name, "..", 3)==0) continue; //escludo
directory corrente e directory padre per evitare loop infiniti

        printf("%.*s%s\n", indent, "\t\t\t\t\t\t\t\t\t\t", ent->d_name); //questa riga mi stampa solo
indent caratteri, quindi mi stampa tante tabulazioni quanto indent

        if(ent->d_type==DT_DIR){ //controllo che sia una directory
            size_t dirEntNameLen = strlen(ent->d_name, 256);
```

```
            char * tmp = malloc(sizeof(char)*(strlen(path)+dirEntNameLen+1));
            tmp=strcpy(tmp, path);
            tmp=strcat(tmp, "/");
            tmp=strcat(tmp, ent->d_name); //CREO NUOVO PERCORSO
            indent++; //aumento indent prima di entrare qui
            printTree(tmp); //entro nella cartella
            indent--; //diminuisco l'indent
            free(tmp);
        }
    }
    closedir(dir);
    close(dirFd);
}
```

Possiamo anche ricevere esercizi di richiesta di ***merge di un albero***
dall'esame 20/07/2023

> *Scrivere un programma cprl che si comporti come il comando "cp -rl". cprl ha due parametri: cprl a b deve copiare l'intera struttura delle directory dell'albero che ha come radice a in un secondo albero con radice b. I file non devono essere copiati ma collegati con link fisici. (l'operazione deve essere fatta dal codice C, senza lanciare altri programmi/comandi)*

```
void copy_tree(int scr_fd, int dst_fd);

int main(int argc, char **argv){ //con main apro le due directory,quella sorgente e quella di
destinazione (che viene creata se non esiste)
    if(argc != 3){
        err(EXIT_FAILURE, "argv");
    }

    //apriamo la sorgente
    int src_fd = open(argv[1], O_RDONLY | O_DIRECTORY);
    if(scr_fd == -1) err(EXIT_FAILURE, "open src");

    //se non esiste creiamo e apriamo la destinazione
    mkdir(argv[2], 0755);
    int dst_ds = open(argv[2], O_RDONLY | O_DIRECTORY);
    if(dst_fd == -1) err(EXIT_FAILURE, "open dst");
    //ora abbiamo i due file descriptor e possiamo applicar la funzione copy_tree

    copy_tree(src_fd, dst_fd);

    close(src_fd);
    close(dst_fd);
    return 0;

    void copy_tree(int src_fd, int dst_fd){
        DIR *dir = fdopendir(src_fd);
        if(dir == -1) err(EXIT_FAILURE, "fdopendir");

        struct dirent *ent;

        while((ent = readdir(dir))!= NULL){ //guardo passo per passo dentro la directory
            //saltiamo directory stessa e padre per evitare i loop
            if(!strcmp(ent->d_name, "." || !strcmp(ent->d_name, ".."))) continue;

            struct stat st

            if(ent->d_type == DT_DIR)){
                // crea directory corrispondente
                if (mkdirat(dst_fd, ent->d_name, 0755) == -1 && errno != EEXIST)
                    err(EXIT_FAILURE, "mkdirat");

                // apri le due directory per la ricorsione
                int new_src = openat(src_fd, ent->d_name, O_RDONLY | O_DIRECTORY);
                int new_dst = openat(dst_fd, ent->d_name, O_RDONLY | O_DIRECTORY);
```

```
                if (new_src == -1 || new_dst == -1)
                    err(EXIT_FAILURE, "openat");

                copy_tree(new_src, new_dst); //se è una directory lo faccio dentro la cartella in modo
ricorsivo

                close(new_src);
                close(new_dst);
            } else if (ent->d_type==DT_REG)) {
                // crea hard link
                if (linkat(src_fd, ent->d_name, dst_fd, ent->d_name, 0) == -1)
                    err(EXIT_FAILURE, "linkat"); //se è un file faccio semplicemente un hard link
            }
        }

    }
}
```

dall'esame 20/07/2023 - merge di un albero con hard link

> *Scrivere un programma cprl che si comporti come il comando "cp -rl". cprl ha due parametri:*
> *cprl a b*
> *deve copiare l'intera struttura delle directory dell'albero che ha come radice a in un secondo albero con radice b. I file non devono essere copiati ma collegati con link fisici.*
> *(l'operazione deve essere fatta dal codice C, senza lanciare altri programmi/comandi)*

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <dirent.h>
#include <err.h>
#include <limits.h>

typedef int fd_t;

void copyTree(char *, char *);

int main(int argc, char ** argv){ //assumiamo che dest esista già
    if(argc != 3) err(EXIT_FAILURE, "argv");

    char * src = argv[1];
    char * dest = argv[2]; //apriamo sia la source che la destination
    copyTree(src, dest);

}

void copyTree(char *src, char *dest){
    DIR* dirs=opendir(src);
    DIR* dird = opendir(dest); //apriamo le directory

    //controllo se sono nulle
    if(dirs == NULL || dird == NULL) err(EXIT_FAILURE, "directory");

    struct dirent* entry;

    while((entry = readdir(dirs))!= NULL){
        if (entry->d_type & DT_DIR) {

            if (strcmp(entry->d_name, ".") == 0 ||
                strcmp(entry->d_name, "..") == 0) //evitiamo di avere loop
                continue;
```

```
            char src_subdir[PATH_MAX];
            char dest_subdir[PATH_MAX];

            snprintf(src_subdir, PATH_MAX, "%s/%s", src, entry->d_name);
            snprintf(dest_subdir, PATH_MAX, "%s/%s", dest, entry->d_name);

            mkdir(dest_subdir, 0755); //creiamo la directory nella destinazione, faccio così perché non
posso creare un hard link di una cartella
            copyTree(src_subdir, dest_subdir); //entriamo ricorsivamente


        }else{
            char srcPath[PATH_MAX];
            char dstPath[PATH_MAX];

            snprintf(srcPath, PATH_MAX, "%s/%s", src, entry->d_name);
            snprintf(dstPath, PATH_MAX, "%s/%s", dest, entry->d_name);

            link(srcPath, dstPath); //hard link
        }
    }
    closedir(dirs);
    closedir(dird);

}
```

Aprire la directory corrente

Può essere richiesto negli esercizi di aprire direttamente la directory attuale, magari direttamente questo oppure può essere chiesto in caso non vengano dati parametri di input (`argc==1`).

```
char currentDir[PATH_MAX]; //pathmax è la lunghezza max di un percorso nel filesystem, dalla libreria
limits.h
DIR* dir; //dir è un tipo che rappresenta una directory aperta
getcwd(currentDir, sizeof(currentDir)); //ottiene la directory corrente (libreria unistd.h)
dir=opendir(currentDir); //apre la directory corrente (libreria dirent.h), restituisce DIR* se ha
successo e NULL se ho un errore
struct dirent *entry; //puntatore ad una struttura che rappresenta un elemento della directory
if(dir==NULL){
    printf("Unable to read directory\n");
    exit(EXIT_FAILURE);
}
while((entry==readdir(dir))!=NULL){
    //scorro tutta la directory
}

closedir(dir); //mi chiude la directory
```

Modificare gli attributi di un file

Mi può venir chiesto di modificare per esempio l'ultima modifica di un file, o la data di creazione. Posso farlo manipolando determinati attributi di un file
ESEMPIO - invecchio di 10 giorni un file

```
struct stat fileStat; //struttura che contiene tutte le informazioni su un file, contiene diversi campi
importanti
```

Campi di stat (libreria sys/stat.h)

- `st_size` dimensioni del file
- `st_mode` tipo + permessi
- `st_atime` ultimo accesso
- `st_mtime` ultima modifica

- `st_ctime` ultima modifica metadata
  Prima di essere usata deve essere popolata tramite la funzione stat()

```
if(stat(argv[i], & fileStat)==0){ //oppure semplicemente stat(filePath,&fileStat)
    struct utimbuf newTime; //struttura usata per impostare i tempi di un file, actime (access time) e
modtime (modification time)
    //10 giorni in secondi=864000s
    fileStat.st_atime = fileStat.st_atime - 864000;
    fileStat.st_mtime = fileStat.st_mtime - 864000; //modifico i timestamp
    newTime.actime = fileStat.st_atime;
    newTime.modtime = fileStat.st_mtime; //faccio la copia dei tempi, copio i nuovi valori calcolati e
preparo la struttura per utime
    utime(argv[i], &newTime); //oppure utime(filePath, &newTime)
    //applico i nuovi tempi, modifico il tempo di accesso e il tempo di modifica
}
```

Creare, modificare e cancellare directory e file

Aprire un file

- `int open(const char *pathname, int flags)` ***apre il file specificato*** dal pathname, che può essere o assoluto o relativo secondo la modalità di accesso specificata, e restituisce il file descriptor a cui si riferirà (o -1 in caso di errore). Un file può essere aperto più volte contemporaneamente e avere più file descriptor associati.
- I valori di ***flags***:
  - `O_RDONLY` readonly -> 0
  - `O_WRONLY` writeonly -> 1
  - `O_RDWR` read and write -> 2
  - `O_CREAT` ***creazione di un nuovo file***, con questo è necessario specificare un terzo argomento `mode_t mode`
    - possiamo anche scrivere `int creat(const char pathname, mode_t mode)`, crea un ***file regolare*** con pathname e lo apre in scrittura, se esiste già lo SVUOTA, tuttavia `owner` (coincide con user id del processo) e `mode` (permessi iniziali) restano invariati.
  - `O_APPEND` scrittura alla fine del file.
  - `O_SYNC` write sincrono, ogni write ritorna solo quando i dati sono effettivamente scritti su dispositivo hardware.

Chiudere un file

- `int close(int filedes)` chiude il file associato al file descriptor (return di open), ritorna l'esito dell'operazione-> 0 o 1, quando un processo termina, tutti i suoi file rimasti aperti vengono chiusi automaticamente dal kernel.
  ESEMPIO

```
int fd;
...
fd = open(pathname, ...);
if(fd==-1){
    //gestione errore
}
//file aperto correttamente
read(fd, ...);
write(fd, ...);
close(fd);
//file chiuso
```

Alcune delle funzioni principali

- `unlink(const char* path)` ***rimuove un file*** dal sistema, restituisce 0 in caso di successo e -1 in caso di errore
- `mkdir(const char* path, mode_t mode)` crea una ***nuova cartella***
- `rmdir(const char* path)` ***elimina una directory*** solo se vuota
  Creazione e modifica di un file
- `fopen(path, "w"` o `open(path, O_CREAT)` per la creazione

- `fprint()`, `fwrite()` o `write()` per scrivere dati

Navigare all'interno di un file

Per spostarci all'interno di un file possiamo usare `seek`.

- `off_t lseek(int filedes, off_t offset, int whence);`
  Sposta la posizione del puntatore nel file `filedes` di `offset` byte a partire dalla posizione `whence`, che può assumere i valori:
- `SEEK_SET` inizio del file
- `SEEK_CUR` posizione corrente
- `SEEK_END` fine del file
  Restituisce il current file offset, o -1 se ha errore.

Lettura e scrittura di byte su un file

- `ssize_t read(int filedes, void *buf, size_t nbyte);`
  Legge dal file indicato con `filedes` una sequenza di `nbyte` a partire dal puntatore e li salva nel buffer che inizia da `buf`. Questa funzione aggiorna il valore del current file offset, e restituisce il numero di file effettivamente letti (<= `nbyte`) o -1 se da errore.
- `ssize_t write(int filedes, const void *buf, size_t nbyte);`
  Questa funzione scrive nel file indicato, parallela a read. Il modo per essere sicuri che i dati siano stati scritti effettivamente scritti è invocare `fsync` dopo le varie chiamate.

Eseguire un file da un programma

Dall'esercizio 16/09/2021

> *Ricevere in input una cartella vuota una directory chiamata exec. Scrivere un programma execname che se viene aggiunto un file nella directory exec interpreti il nome del file come comando con parametri, lo esegua e cancelli il file.*

In questo esercizio si richiede di avere accesso ad una directory, se viene aggiunto un file questo deve essere preso come parametro, venire eseguito e cancellato.
Per fare in modo che il programma viene aggiornato quando viene creato un nuovo file nella directory usiamo ***inotify***.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <err.h>
#include <dirent.h>
#include <sys/inotify.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <limits.h>

#define BUF_LEN (1024 * (sizeof(struct inotify_event) + 16))
#define MAXARGS 64

int main(int argc, char *argv[]) {
    //controlli iniziali
    if (argc != 2) errx(EXIT_FAILURE, "usage: inotirun <dir>");

    //inizializzazione di inotify
    int fd = inotify_init(); //creo istanza inotify e file descriptor per gli eventi
    if (fd < 0) err(EXIT_FAILURE, "inotify_init");

    //dico a inotify di avvisare quando viene creato qualcosa nella directory
    int wd = inotify_add_watch(fd, argv[1], IN_CLOSE_WRITE);
    if (wd < 0) err(EXIT_FAILURE, "inotify_add_watch");

    char buf[BUF_LEN];
```

```c
//loop principale
    while (1) {
        //restituisce tutti gli eventi disponibili
        ssize_t len = read(fd, buf, sizeof(buf));
        if (len <= 0) continue; //nessun evento

        //ciclo sugli eventi letti
        for (char *p = buf; p < buf + len; ) {
            struct inotify_event *ev = (struct inotify_event *)p;

            //filtro solo file e non directory
            if (!(ev->mask & IN_ISDIR) && ev->len > 0) {
                //costruisco il percorso completo
                char filepath[PATH_MAX];
                snprintf(filepath, sizeof(filepath),
                         "%s/%s", argv[1], ev->name);

                // apro e leggo il file
                FILE *f = fopen(filepath, "r");
                if (!f) {
                    p += sizeof(*ev) + ev->len;
                    continue;
                }
                //faccio parsing del contenuto
                char *argv_exec[MAXARGS];
                char line[PATH_MAX];
                int argc_exec = 0;

                //leggo goni riga del file
                while (fgets(line, sizeof(line), f) &&
                       argc_exec < MAXARGS - 1) {
                    line[strcspn(line, "\n")] = 0; //rimuovo \n
                    argv_exec[argc_exec++] = strdup(line);
                }
                argv_exec[argc_exec] = NULL;
                fclose(f);

                //creazione del processo figlio e l'esecuzione
                pid_t pid = fork();
                if (pid == 0) {
                    execv(argv_exec[0], argv_exec);
                    err(EXIT_FAILURE, "execv");
                }
                waitpid(pid, NULL, 0);
                //cancello il file e libero la memoria
                unlink(filepath);

                for (int i = 0; i < argc_exec; i++)
                    free(argv_exec[i]);
            }
            //passo al'evento sucessivo
            p += sizeof(*ev) + ev->len;
        }
    }
}
```

Dire se una directory è vuota

```c
//mi dice se la directory e' vuota
int dir_empty(const char *path) {
    DIR *d = opendir(path);
    struct dirent *e;
    int n = 0;
```

```
    if (!d) return 0;

    while ((e = readdir(d)) != NULL) {
        if (++n > 2) break;
    }
    closedir(d);
    return (n <= 2);
}
```

Link simbolici (posso aggiungere esame 2022)

Un **link simbolico** è un file speciale che contiene il pathname assoluto di un altro file, è un ***puntatore indiretto ad un file*** (a differenza degli hard link che sono puntatori diretti agli inode).
Introdotto per superare le limitazioni degli hard link, che sono possibili solo fra file dello stesso filesystem e che le hard link a directory sono possibili solo al superuser.
Funzioni

- `int symlink(char* oldpathm char* newpath);`
  Crea una nuova directory `newpath` con un link simbolico che punta al file specificato da `oldpath`, e non devono necessariamente risiedere nello stesso file system
- `int readlink(char* path, char* buf, int size);`
  Ci permette di ottenere il contenuto del link simbolico, copia in `buf` il valore del link simbolico.

È possibile che nel compito compaiano esercizi di gestione di link simbolici e assoluti!

Dall'esame 13/02/2025

> *Scrivere un programma `ckfile` che dati il pathname di un file f e di una directory d stampi, a seconda di una opzione:*
> *• l'elenco dei link simbolici che puntano a f presenti nel sottoalbero del file system generato dalla directory d, (opzione -s), `ckfile -s /tmp/file /tmp`*
> *• l'elenco dei link fisici di f presenti nel sottoalbero del file system generato dalla directory d (opzione -l), `ckfile -l /tmp/file /tmp`*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <dirent.h>
#include <err.h>
#include <limits.h>
#include <linux/limits.h>
#include <sys/types.h>

char* flag;
int inode;
char path[PATH_MAX];

void visitDir(char *rootDir){
    DIR *dir = opendir(rootDir);
    if(!dir) return;

    struct dirent *entry;
    char newPath[PATH_MAX];

    while((entry = readdir(dir)) != NULL){
        if(!strcmp(entry->d_name, ".") || !strcmp(entry->d_name, "..")) continue;

        snprintf(newPath, PATH_MAX, "%s/%s", rootDir, entry->d_name);
        if(strcmp(newPath, path) == 0) continue;   // esclude originale

        struct stat st;
        if(lstat(newPath, &st) < 0) continue;
```

```
        if(S_ISDIR(st.st_mode)){
            visitDir(newPath);
        }
        else if(strcmp(flag, "-s") == 0){
            if(S_ISLNK(st.st_mode)){
                struct stat tgt;
                if(stat(newPath, &tgt) == 0 && tgt.st_ino == inode)
                    printf("%s\n", newPath);
            }
        }
        else{ // -l
            if(st.st_ino == inode)
                printf("%s\n", newPath);
        }
    }
    closedir(dir);
}
```

Dall'esercizio 1 - 17/02/2022

> *Scrivere il programma nonest_symlink che ha come parametro il pathname di una directory.*
> *Il programma deve cercare tutti i link simbolici presenti nella directory e cancelli ogni link simbolico nidificato (link simbolico che punta a link simbolico).*

```
char * joinPath(char *, char*); //unisce un percorso di directory e un nome di file

int isNested(char *); //controlla se un link simbolico punta ad un altro link

int main(int argc, char ** argv){
    //error
    if(argc != 2){ //voglio solo 1 argomento olre al nome del programma, quindi lo controllo
        err(EXIT_FAILURE, "argv");
    }
    char * dirPath = argv[1];

    //itero sulla directory
    DIR * dir = opendir(dirPath);
    if(dir==NULL){
        err(EXIT_FAILURE, "opendir"); //controllo che l'apetrura del programma non fallisca
     }

    struct dirent * iter;
    while((iter = readdir(dir))!=NULL){ //legge una voce alla volta nella directory
        //get the file stat
        if(iter->d_type!= DT_LINK) continue; // così analizzo solo i link simbolici
        char * actualFile=joinPath(dirPath, iter->d_name); //ottengo il percorso completo
        if(isNested(actualFile)){
            remove(actualFIle);
        }
        free(actualFile);
    }
    closedir(dir);
}

char * joinPath(char * dir, char * name){
    char * res = malloc(sizeof(char)*(strlen(dir)+strlen(name)+2)); //alloca memoria sufficiente per il
percorso completo
    res = strcpy(res, dir);
    res = strcat(res, "/");
    res = strcat(res, name);
    return res; //dopo aver creato il percorso lo restituisco
}

int isNested(char * path){
    char link2[PATH_MAX];
```

```
    if(readlink(path, link2, PATH_MAX)== -1){ //leggo contenuto del link simbolico
        err(EXIT_FAILURE, "readlink");
    }
    struct stat fileStat;
    if(lstat(link2, &fileStat)==-1){ //ottengo info sul file puntato
        err(EXIT_FAILURE, "lstat");
    }
    return S_ISLINK(fileStat.st_mode) //1 se è un link, 0 altrimenti

}
```

Link fisici

I link fisici (hard link), sono delle associazioni dirette tra un nome di un file e un i-node specifico del file system, condividendo gli stessi dati e metadati del file originale.
E' una vera e propria ***copia speculare di un file***, però ha delle differenze rispetto al soft link:

- può essere creato solo per file e non per directory
- deve essere creato all'interno dello stesso file system
- se si cancella, sposta o rinomina il file originale l'hard link continua ad esistere
- stesso numero inode del file originale

Cos'è un soft link? (o ***link simbolico***)
E' un file, contenente all'interno un collegamento ad un altro file o directory e il link simbolico e file originale hanno un inode differente, quindi per il filesystem sono ***elementi diversi***, anche se legati da questo link. Non contiene alcun dato, e può avere anche un nome diverso da quello originale, possiamo anche avere più link simbolici che puntano allo stesso file originale.
Un modo di pensarla è come una "scorciatoia".

Gestione tempo

I valori temporali DI UN FILE sono tenuti nella struttura stat.

Dall'esame 14/06/2023

> *Facendo uso dei timerfd (vedi `timerfd_create`) scrivere un programma che stampi una stringa a intervalli regolari. (il parametro ha tre campi separati da virgola: il numero di iterazioni, l'intervallo fra iterazione e la stringa da salvare:*
> `tfdtest 4,1.1,ciao`
> *deve stampare ciao quattro volte, rispettivamente dopo 1.1 secondi, 2.2 secondi, 3.3 secondi 4.4 secondi e terminare.*
> *L'esecuzione dovrebbe essere simile alla seguente:*
> `$ tfdtest 4,1.1,ciao`
> `1.100267 ciao`
> `2.200423 ciao`
> `3.300143 ciao`
> `4.400053 ciao`

```
#define MAX_STR 1000

//conversions, macro veloci per fare calcoli sui tempi
#define s2ns(s) s*1000000000
#define ns2s(ns) ns/1000000000
#define s2ms(s) s*1000
#define ms2s(ms) ms/1000
#define ns2ms(ns) ns*1000000
#define ms2ns(ms) ms/1000000

typedef int fd_t;
struct timespec getTimer();

int main(int argc, char ** argv){
    //errore arguments passati (deve essere 1) formato N,INTERVALLO,STRINGA
    if(argc!=2) err(EXIT_FAILURE, "argv");

    //parsing the string
    int times;
```

```
    float tmpInterval;
    char * str =malloc(sizeof(char)*MAX_STR);
    sscanf(argv[1], "%d,%f,%s", &times, &tmpInterval, str);
    //times è il numero di volte che l'intervallo scatta, impInterval intervallo in secondi e str è la
stringa da stampare

    struct timespec interval; //questa struttura rappresenta un intervallo di tempo, è definita dalla
libreria time.h, abbiamo tv_sec che sono i secondi, e tv_nsec che sono i nanosecondi
    interval.tv_sec = tmpInterval;
    tmpInterval -= interval.tv_sec;
    interval.tv_nsec=s2ns(tmpInterval); //costruisco l'intervallo

    //creo il timer
    fd_t timerFd = timerfd_create(CLOCK_REALTIME, 0); //basato su orologio reale, e restituisce un file
descriptor
    if(timerFd == -1) err(EXIT_FAILURE, "timerfd_create");

    //setto il timer
    struct itimerspec timerSetting;
    timerSetting.it_value = timerSetting.it_interval = interval;
    //it_value è la prima scadenza
    //it_interval è l'intervallo periodico
    //se sono uguali abbiamo un timer periodico
    timerfd_settime(timerFd, 0, 0&timerSetting, NULL); //faccio partire subito il timer




    //set timer reader
    getTimer();
    for(int i=0; i<times; i++){
        //aspetto il timer
        char * buf = malloc(sizeof(char)*MAX_STR);
        if(read(timerFd, buf, sizeof(char)*MAX_STR) == -1){ //blocca finché non scade
            err(EXIT_FAILURE, "read");
        }
        struct timespec actualTime = getTimer();
        double timeElaps = actualTime.tv_sec+ns2s((double)actualTime.tv_nsce);
        printf("%.6f %s\n", timeElaps, str) //stampa come richiesto
    }
    close(timerFd);
}

struct timespec getTimer(){
    static int isFirstCall = 1;
    static struct timespec startTime;  //conservo il valore tra una chiamata e l'altra
    struct timespec currTime = {0, 0};
    if(isFirstCall){ //salva il tempo iniziale
        isFirstCall = 0;
        if(clock_gettime(CLOCK_MONOTONIC, &startTime)== -1) err(EXIT_fAILURE, "clock_gettime");
        if(clock_gettime(CLOCK_MONOTONIC, &currtime)== -1) err(EXIT_FAILURE, "clock_gettime");
    }
    if(clock_gettime(CLOCK_MONOTONIC, &currtime)== -1) err(EXIT_FAILURE, "clock_gettime");
    currTime.tv_sec -= startTime.tv_sec;
    currTime.tv_nsec -= startTime.tv_nsec;
    if(currTime.tv_nsec<0){ //corregge se i nanosecondi diventano negativi
        currTime.tv_sec--;
        currTime.tv_nsec += s2ns(1);
    }
    return currTime;
}
```

dall'esame 11/09/2024

> *Scrivere un programma timeout che esegua un programma e lo termini se supera una durata massima prefissata. timeout ha almeno due argomenti: il primo è la durata massima in millisecondi, i parametri dal secondo in poi sono il programma*

> *da lanciare coi rispettivi argomenti.*
> *Es:*
> *timeout 5000 sleep 2*
> *temina in due secondi (sleep termina in tempo).*
> *timeout 3000 sleep 5*
> *passati tre secondi il programma sleep viene terminato.*
> *Timeout deve essere scritto usando le system call poll, pidfd_open, timerfd**.

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdlib.h>
#include <poll.h>
#include <signal.h>
#include <sys/timerfd.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <stdint.h>

//wrapper per pidfd_open, mi resituisce un file descriptor
//mi permette di attendere un processo con poll senza segnali
static int pidfd_open(pid_t pid)
{
    return syscall(SYS_pidfd_open, pid, 0);
}

int main(int argc, char *argv[])
{
    //creazione del processo figlio
    pid_t pid = fork();
    if (pid == 0)
        execvp(argv[2], &argv[2]); //programma + argomenti

    //creazione del pidfd nel padre, rappresenta il processo figlio
    //quando il figlio termina pidfd diventa leggibile
    int pidfd = pidfd_open(pid);

    //creazione del timer come file descriptor
    int tfd = timerfd_create(CLOCK_MONOTONIC, 0);
    struct itimerspec t = { //impostiamo il timeout
    //lo abbiamo in millisecondi e lo convertiamo in secondi e nanosecondi
        .it_value.tv_sec = atoi(argv[1]) / 1000,
        .it_value.tv_nsec = (atoi(argv[1]) % 1000) * 1000000
    };
    //faccio partire il timer
    timerfd_settime(tfd, 0, &t, NULL);

    //preparo la poll, gli stiamo dicendo sveglia il programma
    //quando finisce il processo oppure quando scade il timer
    struct pollfd fds[2] = {
        { pidfd, POLLIN, 0 },
        { tfd,   POLLIN, 0 }
    };

    //attesa bloccante e aspetta uno degli eventi detti prima
    poll(fds, 2, -1);

    //CASO SCADE IL TIMEOUT
    if (fds[1].revents & POLLIN) {
        uint64_t exp;
        read(tfd, &exp, sizeof(exp));   // 🔑 OBBLIGATORIO
        kill(pid, SIGKILL);
        waitpid(pid, NULL, 0);
        return 1;                       // timeout
    }
    //CASO TERMINA IL PROCESSO IN TEMPO
```

```
    waitpid(pid, NULL, 0);
    return 0;                           // terminato in tempo
}
```

Gestione processi

Un processo è un programma in esecuzione, identificato da un PID (process ID), le sue informazioni sono contenute in `/proc/<pid>/` .
Alcuni esempi utili dentro `/proc`

- `/proc/<pid>/cmdline` comando di esecuzione
- `/proc/<pid>/status`
- `/proc/<pid>/fd` file descriptor
- `/proc` elenco di tutti i processi

Alcune funzioni chiave per i processi in c

- `fork()` duplica il processo
- `exec*()` sostituisce il processo corrente con un altro
- `getpid()` ottiene il pid del processo corrente
- `getppid()` ottiene il pid del padre

dall'esame 31/05/2023

> *Scrivere un programma pidcmd che stampi i pid dei processi attivi lanciati con una specifica riga di comando. (Devono coincidere tutti gli argomenti)*
> *es: Il comando "`pidcmd less /etc/hostname`" deve stampare il numero di processo dei processi attivi che sono stati lanciati con "`less /etc/hostname`"*
> *(hint: cercare nelle directory dei processi in `/proc` i "file" chiamati `cmdline`)*

```
#include <err.h>
#include <fcntl.h>
#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>

#define PROCDIR "/proc/"
#define CMDLINE "/cmdline"

typedef int fd_t;


char * getCmd(char **, int *);

int main(int argc, char ** argv) {
    //errors
    if(argc<=1) err(EXIT_FAILURE, "argc"); //se non ho argomenti esco con un errore

    //get the command
    int len=argc-1;
    char * command=getCmd(argv+1, &len);
    for(int i=0; i<len; i++) printf("%c", command[i //costruisco la stringa contenente il comando
generato

    printf("\nSearching for: %s\n", command);

    //opening the directory proc
    DIR * dirPtr;
    if((dirPtr=opendir(PROCDIR))==NULL) err(EXIT_FAILURE, "opendir");
```

```
    //read all directory entry
    struct dirent * dirEnt;
    while((dirEnt=readdir(dirPtr))!=NULL) { //scorro le voci nella directory
        if(dirEnt->d_type!=DT_DIR) continue; //voglio solo directory
        //if it's a directory read it
        //get the path
        size_t dirEntNameLen=strnlen(dirEnt->d_name, 256); //calcolo lunghezza del nome della voce
        char * tmp=malloc(sizeof(char)*(strlen(PROCDIR)+dirEntNameLen+strlen(CMDLINE)+1));
        tmp=strcat(tmp, PROCDIR);
        tmp=strcat(tmp, dirEnt->d_name);
        tmp=strcat(tmp, CMDLINE); //costruisco il percorso /proc/<pid>/cmdline

        //open the file
        fd_t fd;
        if((fd=open(tmp, O_RDONLY))==-1) continue;
        //if the file exists, read it
        //read
        char * buf=malloc((len+1)*sizeof(char));
        int byteread=read(fd, buf, len+1);
        if(memcmp(command, buf, len)==0) printf("%s\n", dirEnt->d_name);
        //confronto il comando cercato con quello del processo e se coincidono stampa il pid
        close(fd);
    }
    closedir(dirPtr);
}

char * getCmd(char ** argv, int * len) {
    //get the lenght
    int c=0;
    for(int i=0; i<*len; i++) c+=strlen(argv[i]);//+1; calcolo num totale dei caratteri degli argomenti
    //get the cmd
    char * cmd=malloc(sizeof(char)*c);
    char * tmp=cmd;
    for(int i=0; i<*len; i++) { //copia ogni argomento consecutivamente
        int steps=strnlen(argv[i], c)+1;
        tmp=memcpy(tmp, argv[i], steps);
        tmp+=steps;
    }
    return cmd;
}
```

dall'esame 30/05/2024

> *Scrivere un programma cloneproc dato il pid di un processo passato come unico parametro, è in grado di eseguirne una copia. (deve rieseguire lo stesso file con lo stresso argv.*
> *consiglio: cercare in `/proc/pid/exe` e `/proc/pid/cmdline` le informazioni necessarie (dove pid è il numero di processo.)*
> *scrivere inoltre un semplice programma che ne dimostri il funzionamento.*

Questo esercizio è quindi diviso in due file

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <err.h>
#include <limits.h>

int main(int argc, char **argv) {
    if (argc != 2)
        errx(EXIT_FAILURE, "argc");

    //buffer per i path, non possiamo usare stringhe letterali perche' i path vanno costruiti
dinamicamente
    char exe_path[PATH_MAX];
```

```
    char proc_exe[PATH_MAX];
    char proc_cmdline[PATH_MAX];

    // /proc/<pid>/exe
    snprintf(proc_exe, sizeof(proc_exe), "/proc/%s/exe", argv[1]);
    //costruiamo il path a /exe, importante questo file esiste finche' il processo sta andando avanti, e'
un symlink

    ssize_t len = readlink(proc_exe, exe_path, sizeof(exe_path) - 1);
    if (len == -1)
        err(EXIT_FAILURE, "readlink");

    exe_path[len] = '\0'; //ora abbiamo il file da eseguire, aggiungiamo \0 manualmente

    // /proc/<pid>/cmdline
    snprintf(proc_cmdline, sizeof(proc_cmdline), "/proc/%s/cmdline", argv[1]);

    int fd = open(proc_cmdline, O_RDONLY);
    if (fd == -1)
        err(EXIT_FAILURE, "open cmdline");

    char buffer[4096];
    ssize_t size = read(fd, buffer, sizeof(buffer));
    close(fd); //cosi' buffer contiene tutti gli argv originali

    if (size <= 0)
        err(EXIT_FAILURE, "read cmdline");

    // Ricostruzione argv, quello dell'input del comando
    //dobbiamo farlo perche' lo abbiamo nel formato "a\0b\0c" ecc (dove i parametri sono a b c)

    char *newargv[256];
    int i = 0;

    char *p = buffer;
    while (p < buffer + size) {
        newargv[i++] = p;
        p += strlen(p) + 1; //+1 ci salta il \0
        //ogni p viene salvato come argv[i]
    }
    newargv[i] = NULL;

    execv(exe_path, newargv); //rilancio del processo
    err(EXIT_FAILURE, "execv"); // se l'operazione prima fallisce
}
```

E il file di test, che quello che fa è avviare un processo, che rimane in vita abbastanza per essere clonato e testato

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    printf("PID: %d\n", getpid());
    for (int i = 0; i < argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]); //stampa arg del processo e pid per testare
    sleep(120); //manitene il processo in vita 120 secondi per darci il tempo di clonare e testare
    return 0;
}
```

Passare dati da un programma all'altro

- scrivere: `write(STDOUT_FILENO, argv[i]`

- ricevere: `read(STDIN_FILENO, buffer, sizeof(buffer))`
  dall'esame 10/01/2024

> Scrivere un programma ***argsend*** che converta i parametri del programma (da argv[1] in poi) in una unica sequenza di caratteri: vengono concatenati i parametri (compreso il terminatore della stringa).*
> *Esempio di funzionamento:*
> `$ ./argsend ls -l /tmp | od -c`
> `0000000 l s \0 - l \0 / t m p \0`
> `0000013`
> Scrivere un secondo programma ***argrecv*** che preso in input l'output di argsend esegua il comando con gli argomenti passati a argsend. Per esempio:*
> `$ ./argsend ls -l /tmp | ./argrecv`
> `total 8988`
> `-rw-r--r-- 1 renzo renzo 150532 Jan 9 16:57 ....`

Nell'esercizio si sono fatti due file diversi, il primo `argsend.c`

```c
int main(int argc, char **argv)
{
    if (argc < 2)
        errx(EXIT_FAILURE, "uso: argsend comando [arg...]");

    for (int i = 1; i < argc; i++) { //da 1 perche' argc[0] e' il nome del programma
        size_t len = strlen(argv[i]) + 1;   // include '\0'
        if (write(STDOUT_FILENO, argv[i], len) != len) //PUNTO FONDAMENTALE
            err(EXIT_FAILURE, "write");
    } //stdout mi permette di "collegarmi" a''esercizio

    return 0;
}
```

Secondo file `argrecv.c`

```c
#define MAXARGS 128

int main(void)
{
    char *argv[MAXARGS];
    int argc = 0;
    //costruiamo manualmente un argv

    char buffer[4096];
    ssize_t n = read(STDIN_FILENO, buffer, sizeof(buffer));
    //leggiamo tuti i byte della pipe, che finiscono in buffer, n ci indica quanti byte abbiamo letto

    if (n <= 0)
        err(EXIT_FAILURE, "read");

    size_t i = 0;
    while (i < n) { //scansioniamo il buffer byte per byte
        if (argc >= MAXARGS - 1)
            errx(EXIT_FAILURE, "troppi argomenti");

        argv[argc++] = &buffer[i];
        i += strlen(&buffer[i]) + 1;
    }

    argv[argc] = NULL;

    execvp(argv[0], argv);
    err(EXIT_FAILURE, "execvp");
}
```

dall'esame 24/06/2025

> *Scrivere due programmi C:* `semsend` *e* `semrecv`*.*
> *Il programma* `semrecv` *stampa il proprio* `pid` *e si pone in attesa. Il programma* `semsend` *ha due parametri: il* `pid` *del processo* `semrecv` *e una stringa. La stringa passata come ultimo parametro a* `semsend` *deve essere trasferita a* `semrecv` *e da quest'ultimo stampata.*
> *Ogni carattere della stringa (incluso il terminatore) deve essere spedito bit a bit usando i segnali* `SIGUSR1` *e* `SIGUSR2`*.*

```c
//semsend.c
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

// SIGUR1 e' bit 0, invece SIGUR2 e'bit 1
//semrecv ricostruisce i caratteri e li stampa e termina quando riceve '\0'

int main(int a,char**v){
    pid_t p=atoi(v[1]); // lettura dei parametri, converte il pid passato da linea di comando
    for(char *s=v[2];;s++) //cicla sui caratteri
        for(int b=0;b<8;b++){ //ciclo sui bit, inviamo un bit per bit
            //invio del segnale
            //estraggo il bit b, se e' 1 mando SIGUSR2, altrimenti SIGUSR1
            kill(p,(*s>>b)&1?SIGUSR2:SIGUSR1);
            usleep(1000); //pausa per evitare che i segnali arrivino troppo velocemente
            if(!*s&&b==7) return 0; //termino quando ho mandato tutti i bit
        }
}
```

```c
//semrecv.c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int bit, c;
//bit posizione del bit (0-6), c e' il carattere in costruzione
//globali perche' signal handler deve ricordarle tra un segnale e l'altro

void h(int s){
    //gestore dei segnali
    c |= (s==SIGUSR2) << bit;
    //se arriva SIGURS1 allora vale zero, altrimenti vale 1
    //il bit viene messo nella posizione bit di c

    //ogni segnale e' 1 bit, quando si arriva a 9 il carattere e' completo
    if(++bit==8){
        //controllo se il carattere e' '\0'
        if(!c) _exit(0);
        //stampo carattere e resetto per il sucessivo
        write(1,&c,1);
        bit=c=0;
    }
}

int main(){
    //stampa il pid
    printf("%d\n",getpid());
    signal(SIGUSR1,h); //prepara il gestore per entrambi i segnali
    signal(SIGUSR2,h);
    while(1) pause(); //li attende all'infinito
}
```

Riconoscere un file eseguibile

Definire un file executable

```
 struct stat st;
 if (stat(newDir, &st) < 0) continue;
 if(st.st_mode & S_IXUSR){ /* ci troviamo in una situazione con un eseguibile*/}
```

Ma come li possiamo distinguere es tra un file elf e uno script?

```
unsigned char magic[4];
if (strcmp(entry->d_name, find) != 0) continue;
int fd = open(newDir, O_RDONLY);
if(fd < 0) continue;
if(read(fd, magic,4)!= 4) continue;
close(fd);
if(magic[0] == 0x7f &&
magic[1] == 'E' &&
magic[2] == 'L' &&
magic[3] == 'F'){ //FILE ELF
    printf("%s: ELF executable\n",newDir);
}else
    if(magic[0] == '#' && magic[1] == '!'){ //FILE SCRIPT
        printf("%s: script\n",newDir);
    }
}
```

dall'esame 14/02/2024

> *Scrivere un programma `search_name` che deve cercare nel sottoalbero della directory corrente tutti i file eseguibili con un nome file specifico passato come primo e unico parametro indicando per ogni file il tipo di eseguibile (script o eseguibile binario).*
> *Ad esempio il comando:*
> `./search_name testprog`
> *deve cercare i file eseguibili chiamati testprog nell'albero della directory corrente. Poniamo siano* `./testprog`, `./dir1/testprog`, `./dir/dir3/testprog`, `search_name` *deve stampare:*
> `./testprog: script`
> `./dir1/testprog: ELF executable`
> `./dir/dir3/testprog: ELF executable`

```
char *find;

void visitDir(char * currentDir){
    DIR* dir;
    dir=opendir(currentDir);
    struct dirent *entry;
    if(dir == NULL) err(EXIT_FAILURE, "directory");

    while((entry = readdir(dir))!= NULL){
        char newDir[PATH_MAX];
        strcpy(newDir, currentDir);
        strcat(newDir, "/");
        strcat(newDir, entry->d_name);
        if(entry->d_type & DT_DIR){
            if(!(strcmp(entry->d_name,"..")==0) && !(strcmp(entry->d_name,".")==0)){
                visitDir(newDir);
            }

        }else{
            struct stat st;
            if (stat(newDir, &st) < 0) continue;
            if(st.st_mode & S_IXUSR){
                unsigned char magic[4];
                if (strcmp(entry->d_name, find) != 0)
                    continue;
```

```
                int fd = open(newDir, O_RDONLY);
                if(fd < 0) continue;
                if(read(fd, magic,4)!= 4) continue;
                close(fd);
                if(magic[0] == 0x7f &&
                magic[1] == 'E' &&
                magic[2] == 'L' &&
                magic[3] == 'F'){
                    printf("%s: ELF executable\n",newDir);
                }else
                    if(magic[0] == '#' && magic[1] == '!'){
                        printf("%s: script\n",newDir);
                    }


            }
        }
    }
}

void main(int argc, char **argv){
    if(argc!= 2) err(EXIT_FAILURE, "argc");
    find = argv[1];
    char currentDir[PATH_MAX];
    getcwd(currentDir, sizeof(currentDir));
    visitDir(currentDir);

}
```

Accedere alla dimensione (byte) di un file

dall'esame 21/01/2025

> *Scrivere il programma samecont che presi come parametri i pathname di un file f e di una directory d stampi l'elenco dei file che hanno la stessa ampiezza (numero di byte) di f ma non sono link fisici di f presenti nel sottoalbero del file system generato dalla directory d*

```
char ogFile[PATH_MAX];
off_t sizeOg;
ino_t inodeOg;

void visitDir(const char *rootDir) {
    DIR *dir = opendir(rootDir);
    if (!dir) err(EXIT_FAILURE, "opendir");

    struct dirent *entry;
    char path[PATH_MAX];

    while ((entry = readdir(dir)) != NULL) {
        if (!strcmp(entry->d_name, ".") || !strcmp(entry->d_name, ".."))
            continue;

        snprintf(path, PATH_MAX, "%s/%s", rootDir, entry->d_name);

        struct stat st;
        if (lstat(path, &st) < 0)
            err(EXIT_FAILURE, "stat");

        if (S_ISDIR(st.st_mode)) {
            visitDir(path);
        } else if (S_ISREG(st.st_mode)) {
            // controllo la dimensione e che non sia un hard link al file
            if (st.st_size == sizeOg && st.st_ino != inodeOg) {
                printf("%s\n", path);
            }
```

```
        }
    }
    closedir(dir);
}

int main(int argc, char **argv) {
    if (argc != 3) err(EXIT_FAILURE, "uso: samecont f d");
    //prendo il file da cercare
    strcpy(ogFile, argv[1]);

    struct stat st; //apro lo stat
    if (stat(ogFile, &st) < 0) err(EXIT_FAILURE, "stat");

    //del file originale prendo la dimensione e l'inode
    sizeOg = st.st_size;
    inodeOg = st.st_ino;

    //visito la directory
    visitDir(argv[2]);
    return 0;
}
```

Python

Librerie python

`import os, sys`
Queste due librerie sono due ***moduli standard fondamentali per la programmazione di sistema***.
`os` lavora con il sistema operativo, mi permette di ***gestire il filesystem*** e con l'ambiente.
Alcune delle operazioni che posso fare sono:

- `os.getcwd()` ottiene la directory corrente
- `os.chdir("cartella")` cambia la directory
- `os.listdir(".")` elenca i file della directory attuale
- `os.mkdir("nuova")` mi crea una nuova directory
- `os.remove("file.txt")` mi fa rimuovere un file
- `os.path.join("dir", "file.txt")` mi permette di unire i percorsi
- `os.system("ls")` mi permette di eseguire programmi esterni

`sys` riguarda ***l'interprete python*** e l'input output, alcune delle operazioni che posso fare sono:

- `sys.argv` accede agli argomenti da riga di comando
- `sys.exit(1)` esce dal programma
- `sys.path` mi permette di uscire dal programma

`import getpass`
Libreria che serve per chiedere all'utente una password senza visualizzarla a schermo, le funzioni principali sono:

- `getpass.getpass()` chiede una password e nasconde l'input
- `getpass.getuser()` prova a ottenere il nome dell'utente corrente

`import pwd`
Libreria che ci permette le informazioni degli utenti dal file di sistema, alcune delle funzioni sono:

- `pwd.getpwuid(uid)` recupera l'utente dato un UID
- `pwd.getpwnam(name)` recupera l'utente dato un nome
- `pwd.getpwall()` restituisce tutti gli utenti

`import subprocess`
Serve per eseguire comandi o programmi esterni dallo script di python.
MI permette di lanciare programmi, catturare l'output, passare input ecc.

- `result = subprocess.run(["ls", "-l"], capture_output=True, text=True)`

`import binascii`
Lavora con dati binario e conversioni tra binario e testo, in particolare quando si maneggiano file binari, checksum e codifiche hex o base64

Ricerca ricorsiva nelle cartelle

Schema di ricerca ricorsiva nelle cartelle in python

```
import os

def fileByDir(directory):
    allFiles = {}

    for ent in os.listdir(directory):
        path = os.path.join(directory, ent)

        if os.path.isdir(path): #CARTELLA
            tmp = fileByDir(path) #CHIAMATA RICORSIVA
            for k in tmp:
                allFiles.setDefault(k, [])
                allFiles[k] += tmp[k]
        else: #FILE
            allFiles.setdefault(ent,[])
            allFiles[ent].append(directory)
    return allFiles
```

Posso anche farlo tramite la funzione `os.walk()`

```
for dirPath, dirNames, files in os.walk(sys.argv[1]): ...
```

Scende automaticamente tutte le sottocartelle, per ogni directory trovata mi restituisce

- `dirPath` percorso della directory corrente
- `dirNames` sottocartelle contenute
- `files` file presenti in quella directory

Dall'esame 16/09/2021 - esempio di uso di ricerca ricorsiva in una cartella

> Scrivere un programma/script che faccia la lista ricorsiva dei file in un sottoalbero riportando in ordine alfabetico per nome di file in quale/quali sottodirectory compare.
> e.g.
> `rlsr mydir`
> `ciao: . ./a`
> `mare: ./a ./b`
> `sole: .`
> Significato: un file con nome ciao esiste nella directory mydir ma anche all'interno della sottodirectory a (esistono cioè i file `mydir/ciao` e `mydir/a/ciao`).

```
import os, sys

def fileByDir(directory):
    allFiles = dict() #dizionario vuoto
    for ent in os.listdir(directory): #scorre il contenuto della directory
        filePath = os.path.join(directory, ent) #ci costruisce il percorso completo
```

```
        if os.path.isdir(filePath): #se incontro una cartella, ci entro dentro
            tmp = fileByDir(filePath) #chiamata ricorsiva
            for key in tmp.keys(): #unisce i dati della ricorsione
                if allFiles.get(key) == None: #sta costruendo una mappa globale di dove sono i file
                    allFiles[key]=[]
                allFiles[key] += tmp[key]
            continue
        if allFIles.get(ent)==None: #se incontro un filem aggiunge la directory corrente all'elenco
            allFIles[ent]=[]
        allFiles[ent].append(directory)
    return allFiles

def main(directory):
    files = fileByDir(directory) #prende il dizionario creato
    if directory.endswith("/"):
        directory = directory.removesuffix("/") #percorso di input normalizzato, fatto per evitare doppi
"/"
    for key in files.keys(): #stampa i risultati
        print(f"{key}: ", end="")
        for elem in files[key]:
            if elem.startswith(directory):
                elem="."+elem.removeprefix(directory)
            print(f"{elem} ", end="")
        print()
if __name__ == "__main__":
    main(sys.argv[1])
```

Processi in python

Un processo è un programma in esecuzione, e ogni processo ha un **PID** (identificatore univoco), un **utente proprietario**, della **memoria** assegnata, uno **stato**, i **file aperti e risorse** e un **processo padre** + eventuali **figli**, su Linux molte informazioni le troviamo in `/proc/<PID>`.

In python possiamo leggere informazioni sui processi accedendo a `/proc` e usando moduli come `os`, `getpass` e `pwd` (vedere librerie python per più info)
Possiamo creare nuovi processi, con `subprocess` oppure creare processi python paralleli con `multiprocessing`. Per terminare e controllare i processi possiamo usare `os.kill(pid, signal)`

Cosa contiene `/proc`?

- `/proc/<PID>/status` info generali sulla memoria, stato, UID
- `/proc/<PID>/exe` link all'eseguibile
- `/proc/<PID>/cmdline` comando usato per lanciarlo
- `/proc/<PID>/fd` file descriptor aperti

dall'esame 08/06/2022

> *Scrivere un programma python o uno script bash che esaminando i dati in /proc//status fornisca in output una tabella che indichi per ogni processo di proprietà dell'utente, il nome dell'eseguibile e l'attuale occupazione di memoria (campo vmSize).*

```
import getpass, pwd, os

PROC_PATH = "/proc" #directory che contiene le info sui processi
STATUS_FILE = "status" #file con le info di stato del processo
CMD_FILE = "exe" # link simbolico al binario eseguibile
PROP = "VmSize:" #proprietà da cercare nel file di status

def main():
    username = getpass.getUser() #inizio trovando l'utente
    for ent in os.listdir(PROC_PATH): #scansiono prop, che contiene cartelle numeriche (processi) e altre
cartelle/file di sistema
```

```
        procPath = os.path.join(PROC_PATH, ent)
        if pwd.getpwuid(os.stat(procPath).st_uid).pw_name != username:
            continue #filtro i processi dell'utente, leggo l'uid del proprietario e lo converto in nome utente, se non corrisponde all'utente attuale lo salto

            #/proc/<PID>/exe è un link simbolico al programma eseguibile
        cmdPath = os.path.join(procPath, CMD_FILE)
        if not os.path.exists(cmdPath):
            continue #nota, continue ti fa skippare da qui fino alla prossima iterazione (torna a for ent in os.. ec ec)
        cmd = os.readlink(cmdPath) #leggo dove punta quel link simbolico
        _, cmd = os.path.split(cmd) #prendo solo il nome del file

        #apro lo status del pid
        statusPath = os.path.join(procPath, STATUS_FILE)
        if not os.path.exists(statusPath):
            continue
        occupation = ""
        with open(statusPath) as statusOpened:
            lines = statusOpened.readlines()
            for line in lines: #leggiamo tutte le righe e cerchiamo quella che contiene VmSize: 123456 kB, togliamo la prima parte e lo salviamo
                id PROP in lin:
                    occupation = line.removeprefix(PROP)
                    break
        print(f"{cmd}: {occupation}") #stampiamo

if __name__ == "__main__":
    main()
```

dall'esame 30/05/2024

> *Scrivere un programma o uno script lscmd che consenta ad un utente di poter avere l'elenco di tutti i pid dei suoi processi in esecuzione raggruppati per pathname del programma eseguito..*
> *Es:*
> `$ lscmd`
> `/usr/bin/bash 2021 2044`
> `/usr/bin/xterm 2010`

```
import os, getpass, pwd

PROC_PATH="/proc"
CMD_FILE = "exe"

def main():
    files={}
    username = getpass.getuser()
    for ent in os.listdir(PROC_PATH):
        procPath = os.path.join(PROC_PATH, ent)
        if pwd.getpwuid(os.stat(procPath).st_uid).pw_name != username:
            continue # filtro i processi non dell'utente
        cmdPath = os.path.join(procPath, CMD_FILE)
        if not os.path.exists(cmdPath):
            continue
        cmd = os.readlink(cmdPath) #leggo dove punta il link simbolico
        files.setdefault(cmd, []).append(ent)
    for t in sorted(files.keys()):
        print(f"{t}:")
        for name in sorted(files[t]):
            print(name)
        print()

if __name__ == "__main__":
    main()
```

Eseguire dal programma

La libreria `subprocess` è importante per questa funzione, perché mi permette di eseguire comandi o programmi esterni.

- `subprocess.call(filePath)` eseguo il processo nel file `filePath`
  esempio di un pezzo di codice che esegue lo script 1, prende il suo output e lo usa come input dello script 2

```
import subprocess

#esegue script1 e cattura l'output
out = subprocess.check_output(["python", "script1.py"])

#passa l'output a script2 come input
subprocess.run(["python", "script2.py"], input = out)
```

dall'esame 22/07/2022

> *Il programma python o lo script bash deve eseguire uno dopo l'altro tutti gli script presenti nella directory passata come parametro (o la current directory se manca il parametro) ma non gli eseguibili binari di tipo ELF.*

```
import subprocess, binascii, sys, os

ELF_MAGIC = "7f454c4602010100000000000000000" #I file binari Linux (ELF) hanno un'intestazione
specifica, chiamata magic number
#Questa stringa rappresenta i primi byte di un file ELF in formato esadecimale

def isElf(file): #mi dice se sono davanti ad un tipo di eseguibile ELF
    magic = getMagic(file)
    return magic == ELF_MAGIC

def getMagic(file):
    tmp = ""
    with open(file, "rb") as myBin: #apro il file in modalità binaria
        header = myBin.read(24) #24 bute sufficienti per coprire header ELF
        tmp = str(binascii.hexlify(header))[2:34] #converto i byte in esadecimale, taglio da 2 per levare
il prefisso b
    return tmp

def main(directory):
    for ent in os.listdir(directory): #scorro tutti i file nella directory
        filePath = os.path.join(directory, ent)
        if os.path.isdir(filePath): #salto le directory (non vado dentro)
            continue
        if isElf(filePath): #salto i file ELF (come da richiesta)
            continue
        if ent=="execNoElf": #salto file con nome specifico
            continue
        try:
            subprocess.call(filePath) #provo a eseguire il file
        except:
            continue #se fallisce ignora l'errore e va avanti

if __name__ == "__main__":
    if len(sys.argv) < 2: #se non passo argomenti uso la directory corrente
        main(os.getcwd())
    else:
        main(sys.argv[1]) #uso la directory passata
```

Link simbolici

dall'esame 14/06/2023

> *Scrivere un programma python o uno script bash che, passata una directory come parametro, cancelli nel sottoalbero generato dalla directory passata come parametro tutti i link simbolici relativi (e non cancelli quelli assoluti)*

> *lrwxrwxrwx 1 renzo renzo 13 Jun 11 17:03 hostname1 -> /etc/hostname*
> *lrwxrwxrwx 1 renzo renzo 15 Jun 11 17:04 hostname2 -> ../etc/hostname*
> *il primo va mantenuto e il secondo cancellato*

```python
import sys, os

def getLink(directory):
    for ent in os.listdir(directory):
        filePath=os.path.join(directory, ent)
        #fileStat = os.stat(filePath)

        if os.path.islink(filePath):
            target = os.readlink(filePath)
            if not os.path.isabs(target):
                os.remove(filePath)
                print(filePath, "removed")

        if os.path.isdir(filePath):
            getLink(filePath) #ricorsione nelle cartelle

def main(directory):
    getLink(directory) #un po' inutile ma non volevo fare la ricorsione sul main

if __name__ == "__main__":
    if len(sys.argv)< 2:
        main(os.getcwd()) #prendo directory attuale
    else:
        main(sys.argv[1])
```

dall'esame 31/05/2023

> *Scrivere un programma python o uno script bash che cerchi all'interno di un sottoalbero se ci son link simbolici che indicano lo stesso file. (hint controllare se coincide il numero dell'i-node del file indicato).*

```python
import os, sys

files = dict() #dizionario globale

def getFile(directory):
    for ent in os.listdir(directory): #scorre gli elementi nella directory
        filePath = os.path.join(directory, ent) #costruisce il percorso completo
        fileStat = os.stat(filePath) #legge le informazioni del file

        if files.get(fileStat.st_ino) == None: #dalle info del file ci interessa particolarmente st_ino,
che è l'inode del file
            files[fileStat.st_ino]=[] #se non esiste nel dizionario creo una lista vuota
    files[fileStat.st_ino].append(filePath) #aggiungo il percorso del file alla lista associata all'inode

        if os.path.isdir(filePath):
            getFile(filePath) #se mi trovo davanti ad una cartella ci entro dentro


def main(directory):
    getFile(directory)
    for key in files.keys(): #mi permette di stampare solo gli inode condivisi
        l = len(files[key])
        if l>1: #se ce ne sono più di uno allora significa che ho presente un hard link, e li stampo
            print(f"{l} link for inode{key}:")
            for name in files[key]:
                print(f"\t{name}")

if __name__ == "__main__": #se ho un argument uso quello, altrimenti uso la directory locale
    if len(sys.argv) < 2:
        main(os.getcwd())
```

```
    else:
        main(sys.argv[1])
```

Spostare un file da un punto ad un altro in modo atomico

dall'esame 23/07/2024

> Scrivere un programma che crei nella directory corrente (se non esiste già) una sottodirectory di nome `...` (tre punti). Tutti i file (regolari) presenti nella directory devono essere spostati nella sottodirectory `...` (tre punti) e ogni file deve essere sostituito nella dir corrente con un link simbolico (relativo, non assoluto) alla nuova locazione. Usare la system call `rename` per fare la sostituzione in modo atomico (in nessun istante il file deve risultare inesistente).

```c
int main(){
    //apro la cartella corrente
    char currentDir[PATH_MAX];
    DIR *dir;
    if (getcwd(currentDir, sizeof(currentDir)) == NULL) err(EXIT_FAILURE, "getcwd");

    dir=opendir(currentDir);
    struct dirent *entry;
    if(dir==NULL) err(EXIT_FAILURE, "opendir");

    //creo la cartella "..."
    if (mkdir("...", 0700) == -1) err(EXIT_FAILURE, "mkdir");


    while((entry=readdir(dir))!=NULL){
        if(entry->d_type & DT_REG){
            char dest[PATH_MAX];
            char tmp[PATH_MAX];

            /* .../nomefile */
            snprintf(dest, sizeof(dest), ".../%s", entry->d_name);

            /* nomefile.tmp */
            snprintf(tmp, sizeof(tmp), "%s.tmp", entry->d_name);

            /* 1. sposta il file */
            if (rename(entry->d_name, dest) == -1)
                err(EXIT_FAILURE, "rename");

            /* 2. crea symlink relativo */
            if (symlink(dest, tmp) == -1)
                err(EXIT_FAILURE, "symlink");

            /* 3. sostituzione atomica */
            if (rename(tmp, entry->d_name) == -1)
                err(EXIT_FAILURE, "rename tmp");

        }
    }

}
```

Trovare nomi senza ascii

dall'esame 20/07/2023

> *Scrivere un programma python o uno script bash che data una directory produca un elenco dei file e delle directory che non potrebbero essere copiati in file system che supportino solo caratteri ascii nei nomi.*

```python
import sys, os
```

```python
def isNotAscii(name): #FUNZIONE PRINCIPALE PER CAPIRE SE CI TROVIAMO DAVANTI AD UN NOME ASCII
    try:
        name.encode('ascii')
        return False
    except UnicodeEncodeError:
        return True

def findNoAscii(directory):
    for ent in os.listdir(directory):
        filePath=os.path.join(directory, ent)
        if isNotAscii(filePath):
            print(filePath)
        if os.path.isdir(filePath):
            findNoAscii(filePath) #entriamo dentro le cartelle ricorsivamente

def main(directory):
    findNoAscii(directory) #per non fare ricorsione nel main (potrei aggiungere controlli)

if __name__ == "__main__":
    main(sys.argv[1])
```

Confrontare alberi nelle cartelle

dall'esame 19/01/2024

> *Scrivere un programma python o uno script bash chiamato tcmp che confronti gli alberi del file system di due directory.. A seconda del parametro deve elencare i pathname di file e di directory che*
> *• sono comuni ad entrambi i sottoalberi, se manca il parametro*
> *• esistono solo nel primo sottoalbero, se il parametro è -1*
> *• esistono solo nel secondo sottoalbero se il parametro è -2*
> *esempi:*
> `$ . /tcmp dir1 dir2`
> *stampa l'elenco dei path che esistono sia in dir1 sia in dir2*
> `$ . /tcmp -1 dir1 dir2`
> *stampa l'elenco dei path che esistono in dir1 ma non in dir2*

```python
import sys, os


def collect_paths(root):
    paths = set()
    for dirpath, dirnames, filenames in os.walk(root):
        rel = os.path.relpath(dirpath, root)
        if rel != ".":
            paths.add(rel)
        for f in filenames:
            paths.add(os.path.join(rel, f))
    return paths


def main():
    if len(sys.argv) == 3:
        mode = 0
        d1, d2 = sys.argv[1], sys.argv[2]
    elif len(sys.argv) == 4: #gestiamo in base a se abbiamo il parametro o no
        mode = int(sys.argv[1])
        d1, d2 = sys.argv[2], sys.argv[3]
    else:
        sys.exit(1)

    p1 = collect_paths(d1)
    p2 = collect_paths(d2) #raccogliamo i path sia di d1 che d2
```

```
    if mode == 0:
        result = p1 & p2
    elif mode == -1:
        result = p1 - p2
    elif mode == -2:
        result = p2 - p1
    else:
        sys.exit(1)

    for p in sorted(result):
        print(p)


if __name__ == "__main__":
    main()
```

Catalogo con "dizionario"

dall'esame 14/02/2024

> *Scrivere un programma python o uno script bash che crei un catalogo dei file presenti nella directory passata come parametro (o la directory corrente se manca il parametro).*
> *Il catalogo deve essere ordinato in categorie in base alla stringa ritornata dal comando 'file'.*
> *Es:*
> `$ catls`
> `ASCII text:`
> `testo1`
> `favourites.txt`
>
> `directory:`
> `mydir`
> `lib`
>
> `Unicode text, UTF-8 text:`
> `unitesto`

```
import sys
import os
import subprocess

def file_type(path):
    #il comando e' quello che si fa sul terminale
    #esefue filepath, per ottenere il tipo di file che abbiamo davanti
    result = subprocess.run(
        ["file", path],
        stdout=subprocess.PIPE,
        stderr=subprocess.DEVNULL,
        text=True
    )
    out = result.stdout.strip()
    return out.split(":", 1)[1].strip()
    #ritorno solo la descrizione

def visit_dir(directory):
    catalog = {} #inizializzo un dizionario
    #chiave tipo, valore i nomi

    for entry in os.listdir(directory):
        path = os.path.join(directory, entry)

        t = file_type(path)
        catalog.setdefault(t, []).

        append(entry)
        #determino la categoria e aggiungo il nome alla lista
```

```
        if os.path.isdir(path):
            sub = visit_dir(path)
            for k in sub:
                catalog.setdefault(k, []).extend(sub[k])
                #unisce i risultati delle sottodirectory mantenendo la stessa struttura

    return catalog

def main(directory):
    catalog = visit_dir(directory)

    for t in sorted(catalog.keys()): #stampa in ordine alfabetico
        print(f"{t}:")
        for name in sorted(catalog[t]):
            print(name)
        print() #stampa i nomi e una riga vuota tra le categorie

if __name__ == "__main__":
    if len(sys.argv) == 2:
        main(sys.argv[1])
    else:
        main(".")
```

"Catturare" il risultato di un comando terminale

Per farlo dobbiamo usare la libreria `subprocess`, che ci permette di eseguire e prendere il risultato di un comando

```
sha1 = subprocess.run(['sha1sum', path], capture_output=True, text=True)
fileHash=sha1.stdout
```

In questo modo stiamo eseguendo il comando `sha1sum` sul percorso `path`. Dobbiamo trovare precisamente l' `stdout`, perché altrimenti ci ritorna l'intera struttura
La struttura di `subproces` è:

```
result = subprocess.run(
    ["python", "--version"],
    stdout=subprocess.PIPE, #risultato
    stderr=subprocess.PIPE, #errore
    text=True
)
```

Dall'esame 23/07/2023

> Scrivere uno script che prende in input da linea di comando il nome di due directory ed elimina (da entrambe le directory) tutti i file che hanno la stessa hash sha1. Più precisamente se c'è un file nella prima directory e uno nella seconda che hanno la stessa hash sha1 tutti i file che hanno la stessa hash sha1 presenti nelle due directory vanno cancellati.
> (Se esistono file con la stessa hash sha1 ma solo in una delle due directory, non sono da cancellare.)

```
import sys, os, subprocess

def visitDir(directory):
    files={}
    for entry in os.listdir(directory):
        path = os.path.join(directory, entry)
        sha1 = subprocess.run(['sha1sum', path], capture_output=True, text=True)
        fileHash=sha1.stdout
        fileHash = fileHash.split(' ')[0]
        files.setdefault(fileHash, []).append(path)

        if os.path.isdir(path):
            sub = visitDir(path)
```

```
            for k in sub:
                files.setdefault(k, []).extend(sub[k])
    return files

def main(dir1, dir2):

    files1=visitDir(dir1)
    print(files1)
    files2=visitDir(dir2)

    for k in sorted(files1.keys()):
        if k in files2.keys():
            for name in sorted(files1[k]):
                os.remove(name)
            for name in sorted(files2[k]):
                os.remove(name)


if __name__=="__main__":
  if len(sys.argv)==3:
        main(sys.argv[1], sys.argv[2])
```

Ordinare delle directory per profondità

dall'esame 21/01/2025

> Lo script o il programma Python deve fornire una lista dei file all'interno di un sottoalbero del file system ordinati secondo la "profondità" a partire da quelli più profondi, per ultimi quelli della directory radice. I nodi allo stesso livello devono essere ordinati in ordine crescente del nome del file.

```
import os, sys

def lista_file(radice):
    files = []
    for root, _, fs in os.walk(radice):
        for f in fs:
            p = os.path.join(root, f)
            # calcola la profondità relativa a radice
            depth = p.count(os.sep) - radice.count(os.sep)
            files.append((depth, f, p))
    for _, _, p in sorted(files, key=lambda x: (-x[0], x[1])):
        print(p)

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Uso: python depth.py <cartella>")
        sys.exit(1)
    lista_file(sys.argv[1])
```

Scrivere su un file

dall'esame 24/06/2025

> *Scrivere un programma python o uno script bash che aggiunga alcune righe di testo all'inizio di tutti i file C, bash, python presenti nella directory corrente (riconoscibili dal suffisso .c .sh o .py)*
> *Il testo deve comparire come commento coerentemente con la sintassi del linguaggio:*
> *• per i file sorgente `C: / .... /`*
> *• per gli script bash: linee con prefisso # (dopo se esiste la prima riga #!)*
> *• per i file sorgente python: `''' .... '''`*

```
import os
```

```python
def main(dir):
    for ent in os.listdir(dir):
        filePath = os.path.join(dir, ent)

        if os.path.isdir(filePath):
            continue

        _, est = os.path.splitext(ent)

        if est not in ('.c', '.sh', '.py'):
            continue

        try:
            with open(filePath, 'r') as f:
                buff = f.read() #importante usare un buff, se facessimo .write direttamente sul file originale finiamo oer sovrascrivere tutto
        except:
            continue

        if est == '.c':
            buff = "/* prova */\n" + buff
        elif est == '.sh':
            buff = "# prova\n" + buff
        elif est == '.py':
            buff = "''' prova '''\n" + buff

        with open(filePath, 'w') as f:
            f.write(buff)

if __name__ == "__main__":
    main(os.getcwd())
```