LabSO 2025

Laboratorio Sistemi Operativi - A.A. 2024-2025

Michele Grisafi - michele.grisafi@unitn.it

Architettura

Kernel Unix

Il kernel è l'elemento di base di un sistema Unix-like, ovvero il nucleo del sistema operativo. Il kernel è incaricato della gestione delle risorse essenziali: CPU, memoria, periferiche, ecc...

Ad ogni boot il sistema verifica lo stato delle periferiche, monta la prima partizione (root file system) in read-only e carica il kernel in memoria. Il kernel lancia il primo programma (*systemd*, sostituto di *init*) che, a seconda della configurazione voluta (target), inizializza il sistema di conseguenza.

Il resto delle operazioni, tra cui l'interazione con l'utente, vengono gestite con i programmi eseguiti dal kernel.

Kernel e memoria virtuale

I programmi utilizzati dall'utente che vogliono accedere alle periferiche chiedono al kernel di farlo per loro.

L'interazione tra programmi ed il resto del sistema viene mascherata da alcune caratteristiche intrinseche ai processori, come la gestione hardware della memoria virtuale (attraverso la MMU).

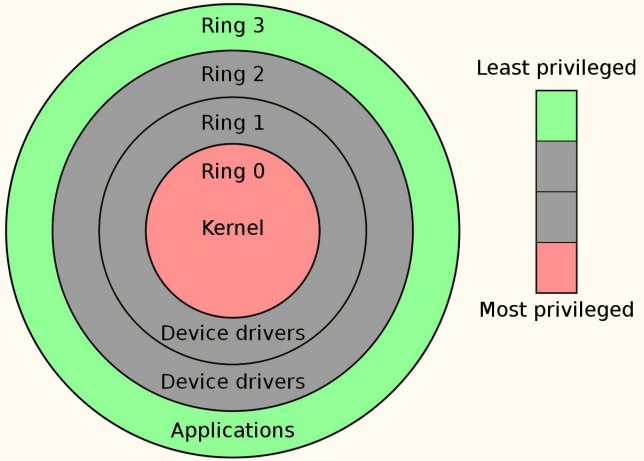
Ogni programma vede se stesso come **unico possessore della CPU** e non gli è dunque possibile disturbare l'azione degli altri programmi → stabilità dei sistemi Unix-like!

Privilegi

Nei sistemi Unix-like ci sono due livelli di privilegi:

- **User space:** ambiente in cui vengono eseguiti i programmi.
- **Kernel space:** ambiente in cui viene eseguito il kernel.

x86

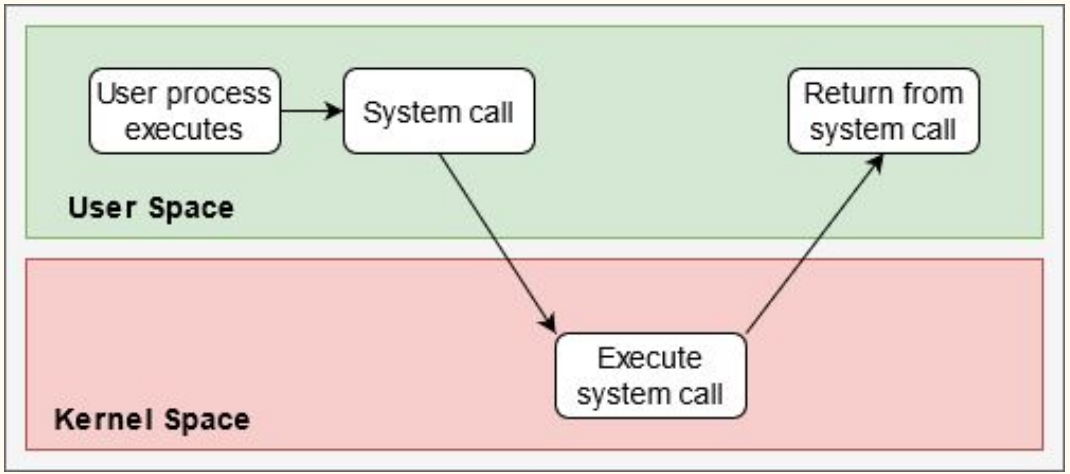


ARM

System calls

System calls

Le interfacce con cui i programmi accedono all'hardware si chiamano **system calls.** Letteralmente "chiamate al sistema" che il kernel esegue nel kernel space, restituendo i risultati al programma chiamante nello user space.



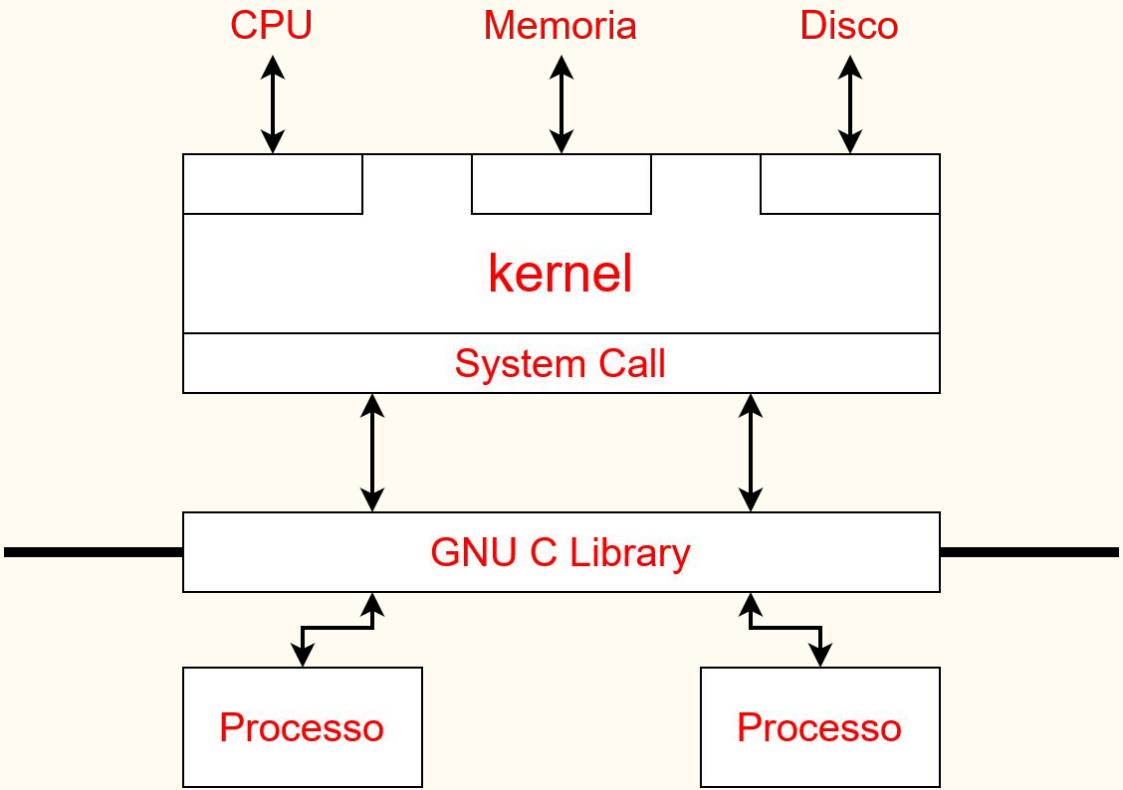
Le chiamate restituiscono "-1" in caso di errore e settano la variabile globale `errno`. Errori validi sono numeri positivi e seguono lo standard POSIX, il quale definisce degli alias.

System calls: librerie di sistema

Utilizzando il comando di shell `ldd` su di un eseguibile si possono visualizzare le librerie condivise caricate e, fra queste, vi sono tipicamente anche *ld-linux.so*, e *libc.so*.

- **ld-linux.so**: quando un programma è caricato in memoria, il sistema operativo passa il controllo a *ld-linux.so* anzichè al normale punto di ingresso dell'applicazione. ld-linux trova e carica le librerie richieste, prepara il programma e poi gli passa il controllo.
- **libc.so**: la libreria GNU C solitamente nota come glibc che contiene le funzioni basilari più comuni.

System calls: librerie di sistema



Syscall e i vari wrappers

```c
#include <sys/syscall.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(void){
  pid_t pid, pid2;
  pid = getpid();
  pid2 = (pid_t)syscall(SYS_getpid);
  printf("Process ID: %d %d\n", pid,pid2);
  __asm__("movq $60, %rax"); //Set up the syscall ID (SYS_exit) (x86-64)
  __asm__("movq $10, %rdi"); //Set up the syscall parameter (ret code 10)
  __asm__("syscall"); //Trigger the syscall
  printf("This line will not be printed\n");
}
```

C - files

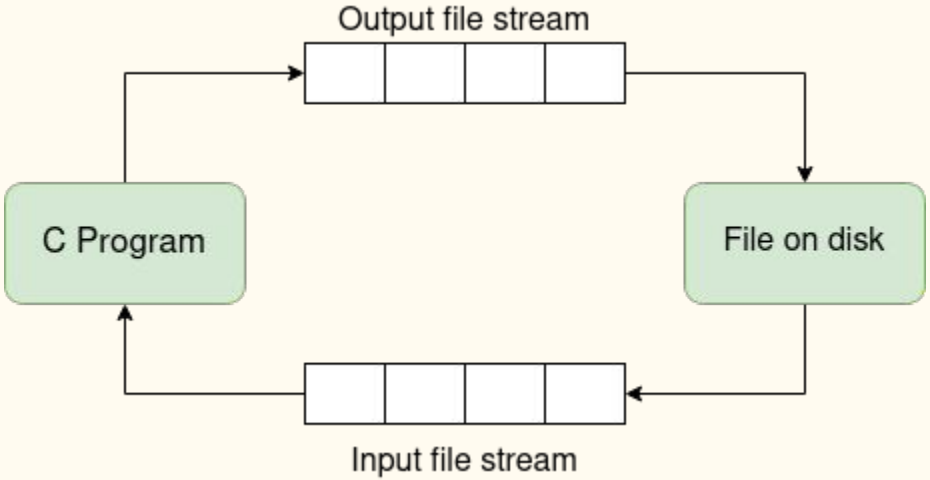
Interazione con i file

In UNIX ci sono due modi per interagire con i file: **streams** e **file descriptors.**

- **Streams**: forniscono strumenti come la formattazione dei dati, bufferizzazione, ecc...
- **File descriptors**: interfaccia di basso livello costituita dalle system call messe a disposizione dal kernel.

Interazione con i file - Streams

Utilizzando gli streams, un file è descritto da un puntatore a una struttura di tipo FILE (definita in stdio.h). I dati possono essere letti e scritti in vari modi (un carattere alla volta, una linea alla volta, ecc.) ed essere interpretati di conseguenza.



Aprire e chiudere un file

```
FILE *fopen(const char* filename, const char* mode);
```

Restituisce un FILE pointer (o NULL se errore → controllare esito!) per gestire il `filename` nella modalità specificata da `mode`.

- `r`: read
- `w`: write or overwrite (create)
- `r+`: read and write
- `w+`: read and write. Create or overwrite
- `a`: write at end (create)
- `a+`: read and write at end (create)

```
int fclose(FILE *stream);
```

Leggere un file

`int fgetc(FILE *stream)`
Restituisce un carattere dallo stream.

`char *fgets(char *str, int n, FILE *stream)`
Restituisce una stringa da `stream` e la salva in `str`. Si ferma quando `n-1` caratteri sono stati letti, una nuova linea (`\n`) è letta o la fine del file viene raggiunta. Inserisce anche il carattere di terminazione e, eventualmente, '`\n`'

`int fscanf(FILE *stream, const char *format, ...)`
Legge da `stream` dei dati, salvandoli nelle variabili fornite (simile a `printf`) secondo la stringa `format`. Restituisce il numero di variabili lette

`int feof(FILE *stream)`
Restituisce 1 se lo `stream` ha raggiunto la fine del file, 0 altrimenti.

File Streams

filename.txt:

```
1 Nome1 Cognome1
2 Nome2 Cognome2
3 Nome3 Cognome3
```

```c
#include <stdio.h>      //fscan1.c

FILE *ptr;  //Declare stream file
ptr = fopen("filename.txt","r+"); //Open
if(ptr == NULL) return 4;
int id;
char str1[10], str2[10];
while (!feof(ptr)){ //Check end of file
    //Read int, word and word
    int numOfConversions = fscanf(ptr,"%d %s %s", &id, str1, str2);
    printf("%d %s %s\n",id,str1,str2);
}

printf("End of file\n");
fclose(ptr); //Close file
```

File Streams

filename2.txt:

```
1 Nome1 Cognome1 Extra1
2 Nome2 Cognome2 Extra2
3 Nome3 Cognome3 Extra3
```

```c
#include <stdio.h>      //fscan2.c

FILE *ptr;  //Declare stream file
ptr = fopen("filename.txt","r+"); //Open
if(ptr == NULL) return 4;
int id;
char str1[10], str2[10];
while (!feof(ptr)){ //Check end of file
    //Read int, word and word
    int numOfConversions = fscanf(ptr,"%d %s %s", &id, str1, str2);
    printf("%d %s %s\n",id,str1,str2);
    if(numOfConversions < 3) return 5; //Error!
}

printf("End of file\n");
fclose(ptr); //Close file
```

File Streams - per carattere

filename.txt:

```
1 Nome1 Cognome1 Extra1
2 Nome2 Cognome2 Extra2
3 Nome3 Cognome3 Extra3
```

```
FILE *ptr;  //Declare stream file
ptr = fopen("filename.txt","r+"); //Open
char str[10][90];
char newChar = fgetc(ptr);
int row = 0, col = 0;
while(newChar != EOF){
      str[row][col++] = newChar;
      if(newChar == '\n'){
            str[row++][col] = '\0';
            col = 0;
      }
      newChar = fgetc(ptr); //Read char by char
};
str[row][col] = '\0';
for(int i = 0; i <= row; i++){
      printf("%s",str[i]);
}
printf("End of file\n");
fclose(ptr); //Close file
```

File Streams - per riga

filename.txt:

```
1 Nome1 Cognome1 Extra1
2 Nome2 Cognome2 Extra2
3 Nome3 Cognome3 Extra3
```

```
FILE *ptr;  //Declare stream file
ptr = fopen("filename.txt","r+"); //Open
char str[10][90];
int row = 0;
while(feof(ptr) != 1){
    fgets(str[row++],90,ptr);
};
for(int i = 0; i <= row; i++){
    printf("%s",str[i]);
}
printf("End of file\n");
fclose(ptr); //Close file
```

Scrivere su un file

`int fputc(int char, FILE *stream)`
Scrive un singolo carattere `char` su `stream`.

`int fputs(const char *str, FILE *stream)`
Scrive una stringa `str` su `stream` senza includere il carattere null.

`int fprintf(FILE *stream, const char *format, ...)`
Scrive il contenuto di alcune variabili su `stream`, seguendo la stringa `format`, senza includere il carattere null.

E molte altre.... Quando scriviamo su un file non serve il terminatore!

File Streams

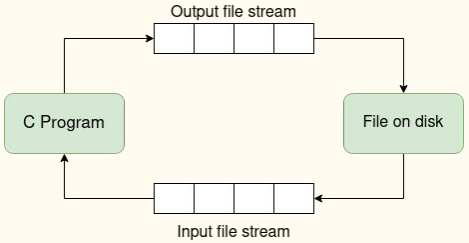
```
#include <stdio.h>

FILE *ptr;
char * str = “his is an interesting text to be written”;
ptr = fopen("fileToWrite.txt","w+");

fputc(’T’,ptr); //Write only one char
fputs(str,ptr); //Write one string without terminator
rewind(ptr); // Reset pointer to begin of file
fprintf(ptr,"Writing %d characters",21); //Write string without terminator


fclose(ptr);
```

Flush e rewind



Seguendo l'immagine, il contenuto di un file viene letto e scritto con degli streams (dei buffer) di dati. Come tali, è comprensibile come queste operazioni non siano immediate: i dati vengono scritti sul buffer e solo successivamente scritti sul file. Il **flush** è l'operazione che trascrive il file dallo stream al file. Questa operazione avviene quando:

- Il programma termina con un `return` dal main o con `exit()`.
- `fprintf()` inserisce una nuova riga ('\n').
- `int fflush(FILE *stream)` viene invocato.
- `void rewind(FILE *stream)` viene invocato.
- `fclose()` viene invocato

`rewind` consente inoltre di ripristinare la posizione della testina all'inizio del file.

File Descriptors

Un file è descritto da un semplice **intero** (file descriptor) che punta alla rispettiva entry nella file table del sistema operativo. I dati possono essere letti e scritti soltanto tramite un buffer alla volta di cui spetta al programmatore stabilire la dimensione.

Un insieme di system call permette di effettuare le operazioni di input e output mantenendo un controllo maggiore su quanto sta accadendo a prezzo di un'interfaccia meno amichevole.

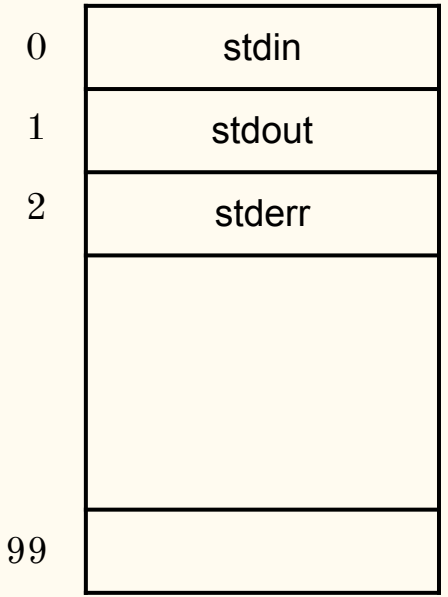
PS: i file streams utilizzano (sotto) i file descriptors

File Descriptors - file table

Per accedere al contenuto di un file bisogna creare un canale di comunicazione con il kernel, aprendo il file con la system call `open` la quale localizza l'i-node del file e aggiorna la *file table* del processo.

A ogni processo è associata una tabella dei file aperti di dimensione limitata, dove ogni elemento della tabella rappresenta un file aperto dal processo ed è individuato da un indice intero (il "file descriptor")

I file descriptor 0, 1 e 2 individuano normalmente standard input, output ed error (aperti automaticamente).

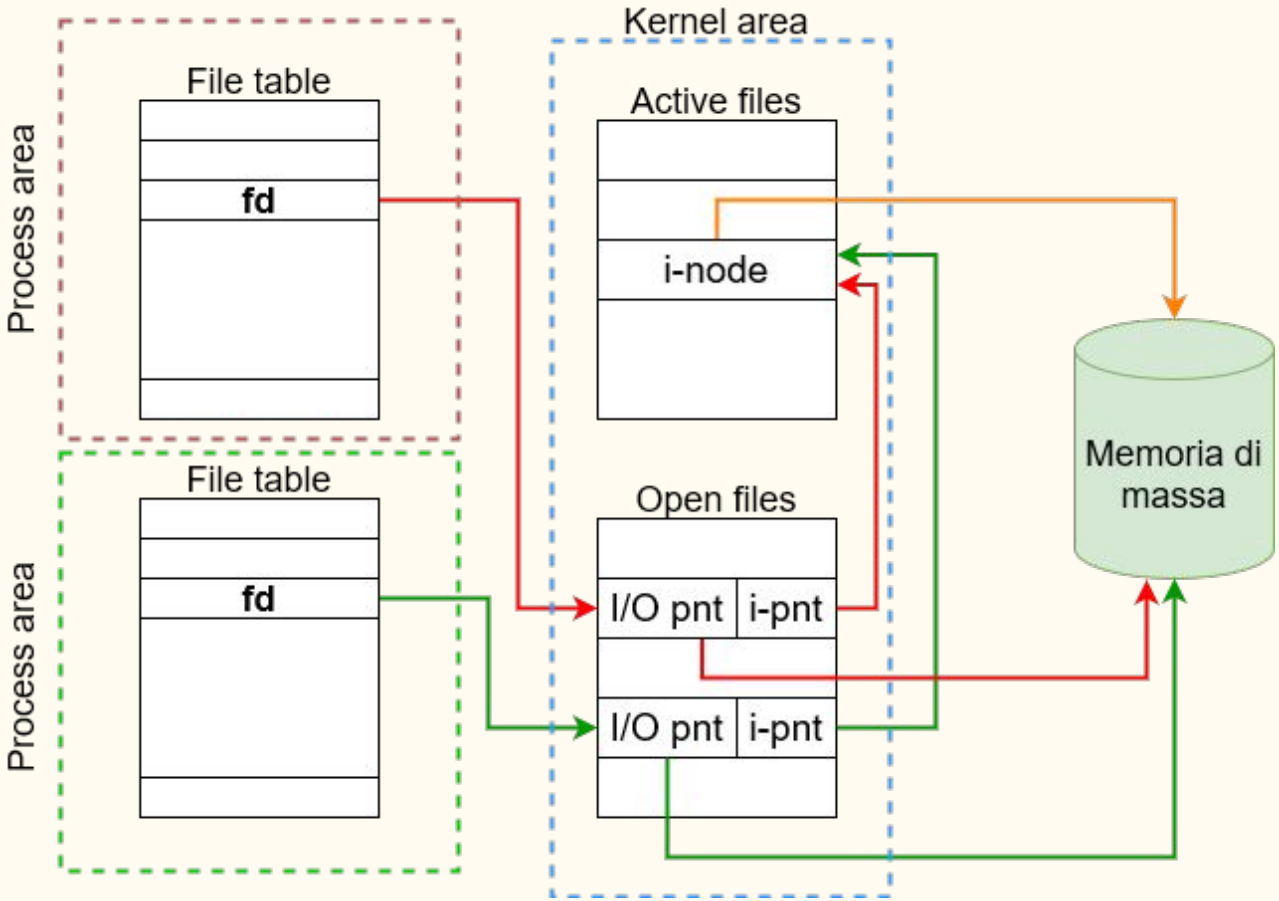


File Descriptors - active and open files

Il kernel gestisce l'accesso ai file attraverso due strutture dati: **la tabella dei file attivi e la tabella dei file aperti**. La prima contiene una copia dell'i-node di ogni file aperto (per efficienza), mentre la seconda contiene un elemento per ogni file aperto e non ancora chiuso. Questo elemento contiene:

- I/O pointer: posizione corrente nel file
- i-node pointer: Puntatore a inode corrispondente

La tabella dei file aperti può avere più elementi corrispondenti allo stesso file!


Kernel area
Process area
File table
fd
Active files
i-node
Memoria di massa
Process area
File table
fd
Open files
I/O pnt
i-pnt
I/O pnt
i-pnt

Aprire e chiudere un file

```
int open(const char *pathname, int flags[, mode_t mode]);
```

`flags`: interi (ORed) che definiscono l'apertura del file. I più comuni:

- `O_RDONLY, O_WRONLY, O_RDWR`: almeno uno è obbligatorio.
- `O_CREAT`: crea il file se non esiste (con `O_EXCL` la chiamata fallisce se esiste)
- `O_APPEND`: apre il file in append-mode (auto lseek con ogni scrittura)
- `O_TRUNC`: cancella il contenuto del file (se usato con la modalità scrittura)

`mode`: interi (ORed) per i privilegi da assegnare al nuovo file: `S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU, S_IRGRP, …, S_IROTH`

La system call restituisce il primo file descriptor disponibile (3 se è il primo file)

```
int close(int fd);
```

Leggere e scrivere un file

`ssize_t` **`read`** `(int fd, void *buf, size_t count);`
Legge dal file e salva nel buffer `buf` fino a `count` bytes di dati dal file associato con il file descriptor `fd`.

`ssize_t` **`write`**`(int fd, const void *buf, size_t count);`
Scrive sul file associato al file descriptor `fd` fino a `count` bytes di dati dal buffer `buf`.

`off_t` **`lseek`**`(int fd, off_t offset, int whence);`
Riposiziona l'offset del file a seconda dell'argomento `offset` partendo da una certa posizione `whence`. `SEEK_SET:` inizio del file, `SEEK_CUR:` dalla posizione attuale, `SEEK_END:` dalla fine del file.

Scrivere su un file

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
//Open file (create it with user R and W permissions)
int openFile = open(“name.txt”, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
if(openFile == -1) return 4;
char toWrite[] = “Professor”;

write(openFile, "hello  world\n", strlen("hello world\n")); //Write to file
lseek(openFile, 6, SEEK_SET); // move I/O pointer
write(openFile, toWrite, strlen(toWrite)); //Write to file

//We use strlen since we don’t need the null terminator

close(openFile);
```

Example

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

//Open existing file in Read only
int openedFile = open("name.txt", O_RDONLY);
if(openedFile == -1) return 4;
char content[10];
int bytesRead;
do{
    bytesRead = read(openedFile,content,9); //Read 9B to content
    content[bytesRead]=0; //The file does not contain the null terminator
    printf("%s",content);
} while(bytesRead > 0);
close(openedFile);
```

C - files: canali standard I

- I canali standard (in/out/err che hanno indici 0/1/2 rispettivamente) sono rappresentati con strutture “stream” (`stdin`, `stdout`, `stderr`) e macro file descriptors (`STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`).

- La funzione `fileno` restituisce l’indice di uno “stream”, per cui si ha:
  - `fileno(stdin)=STDIN_FILENO   // = 0`
  - `fileno(stdout)=STDOUT_FILENO // = 1`
  - `fileno(stderr)=STDERR_FILENO // = 2`
- `isatty(stdin) == 1` (se l’esecuzione è interattiva) OPPURE 0 (altrimenti)

fprintf e canali standard

`fprintf` può essere usato per scrivere su file o su standard output and error. Questi sono trattati come stream, il cui file corrispondente può essere il terminale o un altro file (a seconda dell'impostazione).

```
fprintf(stderr,"...")
fprintf(stdout,"...")
```

```
printf(...) = fprintf(stdout,"..")
```

Esercizi

Crea un’applicazione che copia il contenuto di un file, leggendolo con i file streams e scrivendolo con i file descriptors. Crea poi un programma che fa il contrario. Prova a copiare carattere per carattere e linea per linea.

CONCLUSIONI

Esistono diversi modi per interagire con i file: flussi e descrittori di file. Ogni approccio ha i suoi pro e i suoi contro. Entrambi ruotano attorno a una diversa rappresentazione di un file: gli stream operano su puntatori di file (cioè buffer), mentre i descrittori di file operano su interi (cioè voci di una struttura dati).