C - files

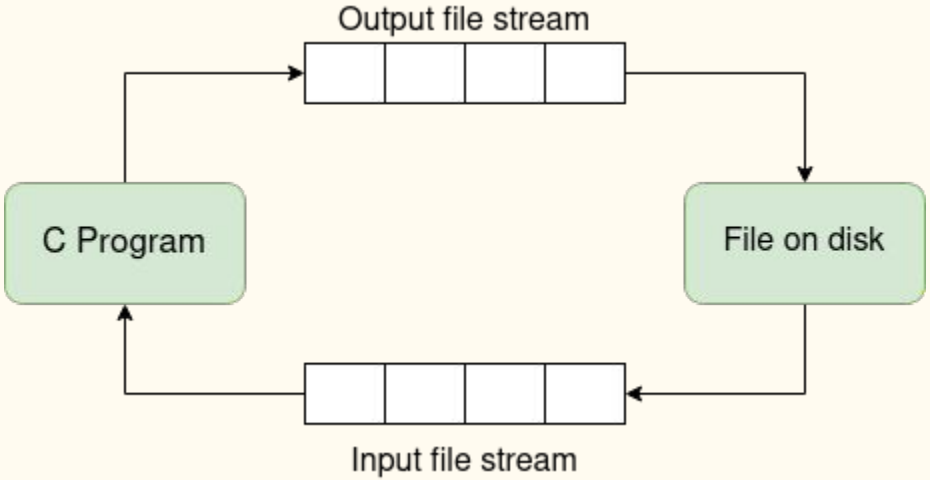
Interazione con i file

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.



Interazione con i file - Streams

```c
#include <stdio.h>

FILE *ptr;  //Declare stream file
ptr = fopen("filename.txt","r+"); //Open

int id;
char str1[10], str2[10];
while (!feof(ptr)){ //Check end of file
  //Read int, word and word
  if (fscanf(ptr,"%d %s %s", &id, str1, str2)>0) {
    printf("%d %s %s\n",id,str1,str2); }
  }

printf("End of file\n");

fclose(ptr); //Close file
```

filename.txt:

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

Modes:

- r: read
- w: write or overwrite (create)
- r+: read and write (update existing)
- w+: read and write. (truncate if exists or create)
- a: write at end (update)
- a+: read and write at end (create)

Interazione con i file - Stream

```c
#include <stdio.h>
#define N 10

FILE *ptr;
ptr = fopen("fileToWrite.txt","w+");
fprintf(ptr,"Content to write"); // Write content to file
rewind(ptr); // Reset pointer to begin of file
char chAr[N], inC;
fgets(chAr,N,ptr); // store the next N-1 chars from ptr in chAr
printf("(N=%d) '%d' '%s'\n", N, chAr[N-1], chAr);
do {
     inC = fgetc(ptr); // return next available char or EOF
     if (inC != EOF) printf("%c",inC);
} while(inC != EOF); printf("\n");
fclose(ptr);
```

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 un buffer alla volta di cui spetta al programmatore stabilire la dimensione.

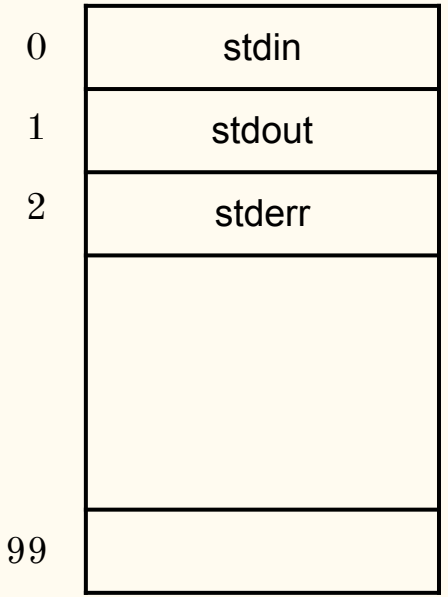
Un insieme di system call permette di effettuare le operazioni di input e output mantenendo un controllo maggiore su quanto sta accadendo a prezzo di un'interfaccia meno amichevole.

File Descriptors

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 (circa 100 elementi), 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)

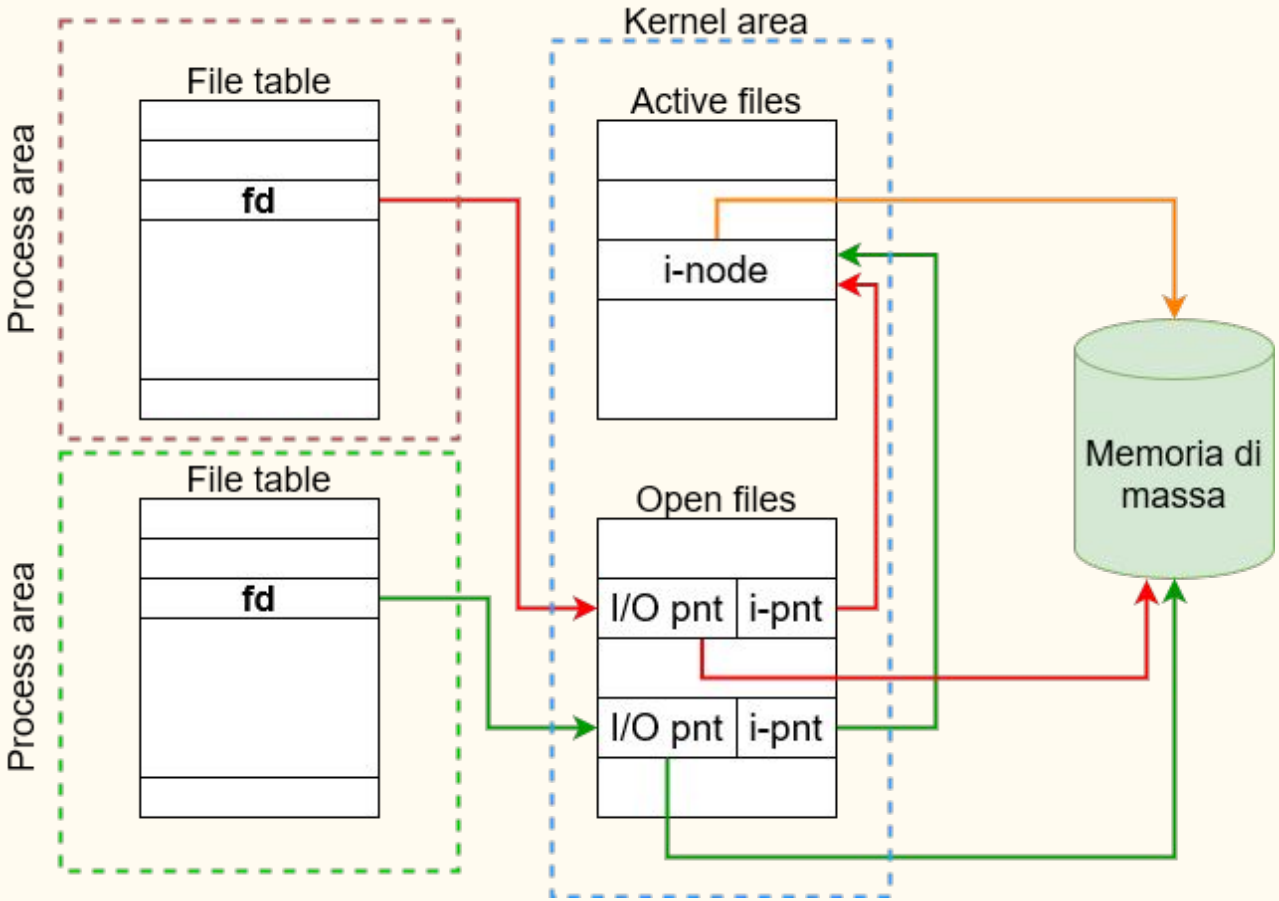


File Descriptors

Il kernel gestisce l'accesso ai files attraverso due strutture dati: **la tabella dei files attivi e la tabella dei files aperti**. La prima contiene una copia dell'inode 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

Interazione files - File Descriptors

L'Input/Output Unix è basato essenzialmente su cinque funzioni: **open**, **read**, **write**, **lseek** e **close.**

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

//Open new file in Read only
int openedFile = open("filename.txt", O_RDONLY);
char content[10]; int canRead, bytesRead;
do{
     bytesRead = read(openedFile,content,9); // read 9 bytes from openedFile to content
     content[bytesRead]=0; // set closing byte as to be seen as a "string"
     printf("%s",content);
} while(bytesRead > 0);
close(openedFile);
```

Interazione files - File Descriptors

L'Input/Output Unix è basato essenzialmente su cinque funzioni: **open**, **read**, **write**, **lseek** e **close.**

```
#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);
char toWrite[] = "Professor\n";

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

close(openFile);
```

Open() flags - File Descriptors

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

I flags sono interi (ORed, cioè messi in “OR”) che definiscono l’apertura del file.
Più comuni per i files:

- **Deve contenere uno tra** O_RDONLY, O_WRONLY, o O_RDWR
- O_CREAT: crea il file se non esistente
- O_APPEND: apri il file in append mode (lseek automatico con ogni write)
- O_TRUNC: cancella il contenuto del file (se aperto con W)
- O_EXCL: se usata con O_CREAT, fallisce se il file esiste già

Mode: definiscono (ORed) i privilegi da dare al file creato: S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU, S_IRGRP, ..., S_IROTH

creat() e lseek()- File Descriptors

```
int creat(const char *pathname, mode_t mode);
```

Alias di `open(file,O_CREAT|O_WRONLY|O_TRUNC, mode)`

```
off_t lseek(int fd, off_t offset, int whence);
```

Muove la “testina” del file di un certo offset a partire da una certa posizione:

- SEEK_SET = da inizio file,
- SEEK_CUR = dalla posizione corrente
- SEEK_END = dalla fine del file.

restituisce la nuova posizione nel file (a partire da inizio file)

ESERCIZIO: usando solo le funzioni dei FD creare una funzione per calcolare la dimensione di un file.

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 (`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)

`printf("ciao");` e `fprintf(stdout, "ciao");` sono equivalenti!

C - files: canali standard II

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

void main() {
  printf("stdin:  stdin ->_flags = %hd, STDIN_FILENO  = %d\n",
    stdin->_flags,  STDIN_FILENO
  );
  printf("stdout: stdout->_flags = %hd, STDOUT_FILENO = %d\n",
    stdout->_flags, STDOUT_FILENO
  );
  printf("stderr: stderr->_flags = %hd, STDOUT_FILENO = %d\n",
    stderr->_flags, STDERR_FILENO
  );
}
```

C - funzioni/operatori

generali e di uso comune

printf / fprintf

```
int printf(const char *format, ...)
int fprintf(FILE *stream, const char *format, ...)
```

Inviano dati sul canale stdout (`printf`) o su quello specificato (`fprintf`) secondo il formato indicato.

Il formato è una stringa contenente contenuti stampabili (testo, a capo, ...) ed eventuali segnaposto identificabili dal formato generale:

```
%[flags][width][.precision][length]specifier
```

Ad esempio: `%d` (intero con segno), `%c` (carattere), `%s` (stringa), ...

Ad ogni segnaposto deve corrispondere un ulteriore argomento del tipo corretto.
(rivedere esempi precedenti)

exit

`void exit(int status)`

Il processo è terminato restituendo il valore `status` come codice di uscita. Si ottiene lo stesso effetto se all'interno della funzione `main` si ha `return status`.

La funzione non ha un valore di ritorno proprio perché non sono eseguite ulteriori istruzioni dopo di essa.

Il processo chiamante è informato della terminazione tramite un "segnale" apposito. I segnali sono trattati più avanti nel corso.

C - piping via bash

C - piping via bash

- In condizioni normali l'applicazione richiamata da bash ha accesso ai canali stdin, stdout e stderr comuni (tastiera/video).
- Se l'applicazione è inserita via bash in un "piping" (come in `ls | wc -l`) allora:
  - Accede all'output del comando a sinistra da stdin
  - Invia il suo output al comando di destra su stdout

```c
#define MAXBUF 10
#include <stdio.h>
#include <string.h>

int main() {
    char buf[MAXBUF];
    fgets(buf, sizeof(buf), stdin); // may truncate!
    printf("%s\n", buf);
    return 0;
}
```

```
$ gcc src.c -o pip.out
$ echo "ciao come stai" | ./pip.out
```

C - esempio piping da bash

Esempio di una semplice applicazione che legge da `stdin` e stampa su `stdout` invertendo minuscole [a-z] con maiuscole [A-Z]

```c
#include <stdio.h>

int main() {
    int c, d;
    // loop into stdin until EOF (as CTRL+D)
    while ((c = getchar()) != EOF) { // read from stdin
        d = c;
        if (c >= 'a' && c <= 'z') d -= 32;
        if (c >= 'A' && c <= 'Z') d += 32;
        putchar(d);         // write to stdout
    };
    return (0);
}
```

CONCLUSIONI

Comprendendo il funzionamento dei vari tipi di variabili, in particolare la gestione dei puntatori e dei vettori, e sfruttando poi le funzioni illustrate è possibile realizzare delle applicazioni che manipolano argomenti passati via CLI o anche interagire con processi terzi (in particolare attraverso il file-system o via bash con il piping).