3.0 LIBRERIE STANDARD I/O

Parliamo di libreria standard perché rientra nello standard C ed è stata implementata su molti sistemi operativi oltre che su UNIX. La modalità con cui si accede ai file è attraverso quelli che vengono chiamati **stream** (flusso di dati) e non si utilizza il File Descriptor come visto precedentemente. C'è quindi un passaggio logico da FD a Stream quando si passa da I/O di base a standard I/O.
Uno stream è un puntatore a una struttura di tipo FILE, questa struttura al suo interno contiene molte informazioni: File descriptor usato per l'I/O, Puntatore al buffer per lo stream, Dimensione del buffer, Contatore di caratteri, Etc….
Ogni processo che viene creato ha sempre 3 stream predefiniti che sono già aperti quando il nostro processo viene eseguito. Questi 3 sono:

- **stdin** che punta allo *standard input*.
- **stdout** che punta allo *standard output*.
- **stderr** che punta allo *standard error*.

Stiamo ritrovando la stessa situazione descritta per i File Descriptor, solo che in questo caso parlando di Stream. In effetti, questi 3 Stream si riferiscono esattamente ai File Descriptor descritti nei precedenti appunti.
Quello che si aggiunge quando parliamo di questo argomento, è il concetto di **Buffering**. Tutte le funzioni di standard I/O utilizzano questo concetto di immagazzinare temporaneamente le informazioni da scrivere e poi al momento giusto chiamare *read* e *write*. Le librerie standard automaticamente allocano il buffer chiamando *malloc*. Ci sono 3 tipi di buffering: **fully buffered**, **line buffered**, **unbuffered**.

3.0.1 FULLY BUFFERED

In questa modalità, le operazioni di I/O avvengono nella realtà soltanto quando il Buffer è pieno. Il termine **flush** descrive la scrittura di un buffer standard di I/O, più in particolare esso significa "writing out" il contenuto di un buffer. In generale tutti gli stream sono di questo tipo, a meno che non si riferiscono ad un terminale.

3.0.2 LINE BUFFERED

In questa modalità, le operazioni di I/O avvengono quando si incontra il carattere di *newline* sull'input o output, oppure se si riempisse il buffer. Questa modalità è usata tipicamente su stream che si riferiscono ad un terminale (standard input o output).

3.0.3 UNBUFFERED

In questo caso, la libreria standard di I/O non bufferizza i caratteri, di fatto significa che le operazioni avvengono immediatamente. Tipicamente, lo stream **standard error** utilizza questa modalità, in quanto vogliamo che un messaggio di errore venga visto immediatamente e non aspettare che sia abilitata la scrittura in un certo momento. È possibile modificare la modalità di buffering associata ad un dispositivo.
Con questa porzione di codice, innanzitutto si dichiara una stringa con un certo contenuto di 15 caratteri. Con il ciclo while, ad ogni iterazione c'è una chiamata putchar, che va a stampare o almeno ci aspettiamo che lo faccia, un carattere. Quello che accade effettivamente è che putchar viene eseguita, mentre l'operazione di I/O non viene effettuata. Questo perché associaciato ad uno standard output, c'è un buffer. Ogni chiamata di putchar scrive un carattere nel buffer. Nei primi 15 secondi di questo ciclo, verrà scritto un carattere per volta il testo nel buffer. Appena esce dal ciclo, inserirà nel buffer il carattere "\n", l'effetto sarà un flush di tutto il buffer. Dunque, dopo 15 secondi viene stampato in un colpo solo "Uno alla volta?".

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

int main(void)
{
 char *stringa="Uno alla volta?";
 while (*stringa) {
    putchar(*stringa++);
    sleep(1);  /* fermiamo il processo per un secondo */
  }
 putchar('\n');
 sleep(4);
 return(0);
}
```

Un modo per poter realmente stampare un carattere alla volta è forzare la scrittura modificando il buffering. Se si modifica il buffer si può modificare l'output. In questo caso ci interessa un funzionamento come quello di unbuffered, coerentemente alla sua definizione.

3.1 setbuf – setvbuf

```c
#include <stdio.h>
void setbuf (FILE *fp, char *buf);
int setvbuf (FILE *fp, char *buf, int mode, size_t size);
```

Restituiscono 0 se OK, un valore diverso da 0 in caso di errore.
La **setbuf**, prende come primo parametro un puntatore ad un FILE aperto. La **setvbuf** è di livello un po' più basso in quanto consente di effettuare delle operazioni più precise, il primo parametro sarà sempre un puntatore ad uno stream.
Se alla **setbuf** gli si passa un *buf* NULL lo stream passato come primo parametro diventa unbuffered. Viceversa, se come secondo parametro passiamo un buffer diverso da NULL, avviene che sarà associato al nostro stream un buffer di lunghezza ben definita e il tipo del buffering sarà fully buffered o line buffered a seconda di cosa sia il nostro buffer (automaticamente l'stdio effettua questa associazione considerando il nostro FILE di che tipo è).

Function	*mode*	*buf*	Buffer & length	Type of buffering
setbuf		nonnull	user *buf* of length BUFSIZ	fully buffered or line buffered
		NULL	(no buffer)	unbuffered
setvbuf	_IOFBF	nonnull	user *buf* of length *size*	fully buffered
		NULL	system buffer of appropriate length	
	_IOLBF	nonnull	user *buf* of length *size*	line buffered
		NULL	system buffer of appropriate length	
	_IONBF	(ignored)	(no buffer)	unbuffered

Con la **setvbuf** abbiamo la possibilità di cambiare il buffering, ma anche di cambiare la dimensione del buffer. Con il campo *mode* si specifica quale buffer di vuole associare al file. Se scegliamo una modalità **_IONBF**, il *buf* che passiamo viene ignorato. Se invece scegliamo una modalità **_IOFBF** e passiamo un *buf* NULL, vuol dire che la dimensione sarà quella di default scelta dallo standard I/O. Valore diverso da NULL indica che scegliamo la size.
Queste due funzioni devono essere chiamate dopo che lo stream è stato aperto, e prima di ogni altra operazione sullo stream.
Tornando all'esempio precedente:
Un **setbuf(stdout, NULL)**, ci permette di ottenere quello che volevamo.

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

int main(void)
{
 char *stringa="Uno alla volta?";  /*questa volta SI*/
 setbuf(stdout, NULL);  /*stdout unbuffered */
 while (*stringa) {
     putchar(*stringa++);
     sleep(1);   /*fermiamo il processo per un secondo*/
  }
      return(0);
}
```

3.2 fflush

```
#include <stdio.h>
int fflush (FILE *fp);
```

Questa funzione forza il flushing di uno stream. Se chiamo questa funzione passandogli uno stream, l'effetto sarà di scrivere il contenuto del buffer sul file puntato da fp. Restituisce 0 se OK, EOF in caso di errore.
Se passassimo NULL a questa funzione, si effettua il flush di tutti gli stream aperti.

Se mettiamo la fflush dopo il while, verrebbe stampato in un solo colpo la stringa su standard output.

```
#include <stdio.h>

int main(void)
{
  char *stringa="Uno alla volta?";
  while (*stringa) {
    putchar(*stringa++);
    sleep(1);   /*fermiamo il processo per un secondo*/
   }
   fflush(stdout);  /*invece di putchar('\n') */
   sleep(4);
   return(0);
}
```

Vediamo quelle che sono le differenza tra una system call e le funzioni di libreria I/O:

- Una system call di I/O viene invocata ed immediatamente eseguita.
- L'esecuzione di una funzione di libreria di I/O passa attraverso il buffer.

3.3 APERTURA E CHIUSURA DI UNO STREAM

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char *type);
FILE *freopen(const char *pathname, const char *type, FILE *fp);
FILE *fdopen(int fd, const char *type);
int fclose(FILE *fp);
```

La funzione **fopen**, preso un *pathname*, assocerà ad esso un puntatore a file, con il *type* specifichiamo in che modalità vogliamo aprire il file.
La funzione **freopen** apre il file *pathname* sullo stream *fp*, chiudendo questo se era già aperto.
La funzione **fdopen** prende un file descriptor (che è stato ottenuto per esempio con una open) e gli associa uno standard I/O stream.

Con queste funzioni non indichiamo mai quali sono i permessi di accesso di un file, banalmente vengono usati i permessi standard per i file che apriamo con queste funzioni. Si ricordi che nella open invece si decidono i permessi per i file.

type	Description
r or rb	open for reading
w or wb	truncate to 0 length or create for writing
a or ab	append; open for writing at end of file, or create for writing
r+ or r+b or rb+	open for reading and writing
w+ or w+b or wb+	truncate to 0 length or create for reading and writing
a+ or a+b or ab+	open or create for reading and writing at end of file

La funzione **fclose** chiude uno stream aperto. Restituisce 0 se OK, EOF in caso di errore. Ogni dato output presente nel buffer è "flushed" prima che il file sia chiuso. Quando un processo termina:

- Tutti gli stream di I/O con dati bufferizzati non scritti sono "flushed".
- Tutti gli stream di I/O aperti sono chiusi.

3.4 LETTURA E SCRITTURA DI UNO STREAM

Una volta che uno stream è stato aperto possiamo scegliere tra:

- I/O non formattato:
  - Un carattere alla volta: **getc, getchar, putc, putchar …**
  - Una linea alla volta: **fgets, fputs …**
  - Diretto (I/O binario, record oriented, structure oriented): **fread …**
- I/O formattato: **scanf, printf, fscanf, fprintf**

3.5 EOF

Le funzioni precedenti restituiscono EOF sia su errore che quando incontrano la fine del file. Possiamo dire che la EOF è un segnale che ci dice che c'è stato un errore di qualche tipo. In molte implementazioni sono mantenuti due flag per ogni stream: **flag di errore** e **flag di end-of-file**.
Sono flag binari, quando una funzione restituisce EOF perché si trova alla fine del file, il flag di end-of-file diventerà 1, al contrario se una funzione restituisce un errore allora il flag di errore diventerà 1. Per testare il flag settato da queste due funzioni si ricorre alle seguenti funzioni:

```
#include <stdio.h>
int ferror(FILE *fp);
int feof(FILE *fp);
```

Si testa il flag contenuto nel flag di errore o end-of-file.

3.6 POSIZIONAMENTO IN UNO STREAM

```
#include <stdio.h>
long ftell(FILE *fp);
long fseek(FILE *fp, long offset, int whence); /* simile a lseek */
void rewind(FILE *fp);
int fgetpos(FILE *fp, fpos_t *pos);
int fsetpos(FILE *fp, const fpos_t *pos)
int fileno(FILE *fp);
```

La funzione **ftell** restituisce l'indicatore della posizione corrente (misurato in byte), -1 in caso di errore.
La funzione **fseek** è identica alla lseek, cambia solo il primo parametro in quanto non si passa un File Descriptor ma un puntatore a file.
La funzione **rewind** riporta ci fa riposizionare all'inizio del file.
La funzione **fgetpos** pone nell'oggetto puntato da *pos* l'indicatore della posizione del file fp.
La funzione **fsetpos** prende dall'oggetto puntato da *pos* l'indicatore della posizione del file fp.
La funzione **fileno** restituisce il file descriptor associato allo stream, utile se vogliamo chiamare per esempio la **dup**.