**Autore:** Luciano VIVIANI

**Classe:** QUARTA INFORMATICA (4IB)

**Anno scolastico:** 2004/2005

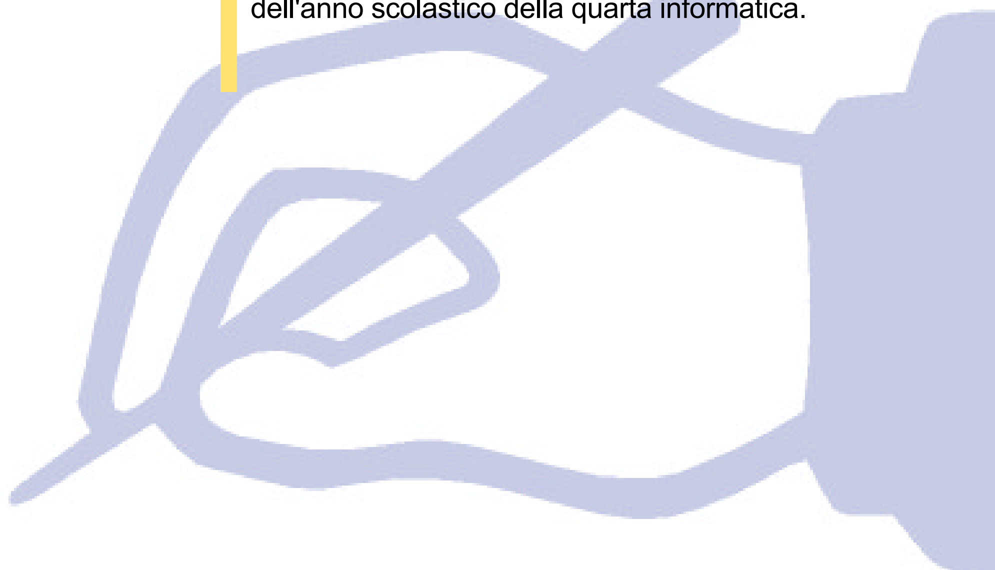
**Scuola:** Itis Euganeo

# ESERCIZI SULLA PROGRAMMAZIONE DI SISTEMA GNU/LINUX

Questa dispensa non vuole essere un manuale di programmazione di sistema, si limita a illustrare alcuni temi che si ritengono importanti per la comprensione di un sistema operativo e a proporre alcuni esercizi.

Come sistema operativo si è scelto Unix, nella sua versione Linux.

Si richiese una certa conoscenza della programmazione in linguaggio C (La dispensa si basa nella prima parte sulla documentazione della libreria C GNU), fornito dall'insegnamento Informatica nel terzo e quarto anno, e viene preceduta dalla spiegazione teorica dei concetti fondamentali relativi ai sistemi operativi, quindi si colloca nel secondo quadrimestre dell'anno scolastico della quarta informatica.



# Esercizi sulla programmazione di sistema GNU/Linux

Dispensa per il laboratorio di sistemi, classe quarta indirizzo Abacus
LucianoViviani 2005 - luciano_viviani@libero.it

## Introduzione

Ritengo che per comprendere il funzionamento di un sistema operativo, come di molte altre cose, la realizzazione pratica in laboratorio di programmi che ne evidenzino le caratteristiche fondamentali sia di primaria importanza. Come sistema operativo ho scelto Unix, che nella sua versione Linux è attualmente facilmente installabile sui PC di un qualsiasi laboratorio, anche in versioni che convivano senza particolari problemi con Windows, per la sua caratteristica di sistema aperto e ampiamente documentato e per la facilità con cui si possono realizzare programmi che funzionino con più processi contemporanei (con le relative problematiche di competizione per alcune risorse condivise).

La dispensa si basa nella prima parte sulla documentazione della libreria C GNU, disponibile in Internet e di cui consiglio la consultazione per chi volesse approfondire la tematica, che ho in parte tradotto e riassunto. Nel testo sono presenti alcuni listati di programma, inseriti nel testo come **Esempio:** ciò significa che sono tratti dalla documentazione GNU e non verificati nell'esecuzione. Nella seconda parte sono inseriti degli esercizi verificati in laboratorio con proposte per eventuali varianti. Per la scrittura dei programmi basta un semplice editor di testo, per la loro compilazione ho impiegato il compilatore ***gcc*** sempre presente nelle distribuzioni Linux che ho provato.

Questa dispensa non vuole essere un manuale di programmazione di sistema, si limita a illustrare alcuni temi che ritengo importanti per la comprensione di un sistema operativo e a proporre alcuni esercizi.

Ovviamente si richiese una certa conoscenza della programmazione in linguaggio C, fornito dall'insegnamento Informatica nel terzo e quarto anno, e viene preceduta dalla spiegazione teorica dei concetti fondamentali relativi ai sistemi operativi, quindi si colloca nel secondo quadrimestre dell'anno scolastico.

# 1 Documentazione (libreria GNU)

## 1.1 Processi

I processi sono le unità primitive per allocare le risorse del sistema. Ciascun processo ha un proprio spazio di indirizzi e solitamente un'unica sequenza (thread) di controllo. Un *processo* esegue un *programma*, si possono avere più processi che eseguono lo stesso programma ma ciscuno esegue la sua copia del programma nel suo spazio di indirizzi e lo esegue in modo indipendente dalle altre copie.

L'organizzazione dei processi è gerarchica, ciascun processo ha un genitore da cui eredita gran parte delle caratteristiche.

### 1.1.1 Concetto di creazione di un processo

Ciascun processo è contrassegnato da un numero di identificazione detto ID, quando un processo viene creato gli viene assegnato un ID univoco. La vita di un processo termina quando la sua fine viene notificata al processo genitore, in questo istante tutte le risorse del processo, incluso il suo ID, sono rese libere per altri processi. I processi sono creati dalla chiamata di sistema `fork()` che crea un processo figlio come una copia del genitore, tranne l'ID.

Dopo questa specie di *clonazione* entrambi i processi continuano la loro esecuzione dall'istruzione successiva alla `fork()` anche se il valore del PID ritornato dalla `fork()`, sempre 0 per il processo figlio, può essere usato per diversificare il flusso di esecuzione dei due processi con un'istruzione `if`.

Avere svariati processi che eseguono lo stesso programma è utile solo occasionalmente (per esempio se si hanno più utenti con più terminali), in genere il figlio può eseguire un altro programma mediante una delle funzioni exec. Il contesto di esecuzione di un processo viene detto *immagine* del processo, con le funzioni exec l'immagine del processo figlio viene sostituita dalla nuova.

### 1.1.2 Identificazione dei processi

Il tipo di dati `pid_t` rappresenta l'ID di un processo, la funzione getpid fornisce l'ID del processo corrente, mentre getppid fornisce l'ID del genitore. Il programma deve includere gli headers `unistd.h` e `sys/types.h` per usare queste funzioni.

Tipo di dati: **pid_t**

Nella libreria GNU questo tipo di dato è un `int`.

Funzione: `pid_t getpid (void)` la funzione ritorna il valore del PID del processo in esecuuzione

Funzione `pid_t getppid (void)` ritorna il valore del PID del genitore

### 1.1.3 Creazione di un processo

la funzione `fork` è la primitiva per creare un processo, dichiarata nel file header `unistd.h`.

Funzione `pid_t fork (void)` crea un nuovo processo

Se l'operazione ha successo, dopo esistono due processi identici, il genitore e il figlio ma il valore ritornato dalla funzione è diverso nei due casi: nel figlio vale sempre 0, mentre nel padre il suo valore è il PID del figlio.

Se l'operazione fallisce, fork torna al padre il valore -1;

### 1.1.4 Esecuzione di un file

Per fare in modo che il figlio esegua un nuovo file come immagine del processo si usa una delle funzioni seguenti, dichiarate nel file `unistd.h`.

Funzione: `int execv (const char *filename, char *const argv[])`

La funzione esegue il file (eseguibile) il cui nome è `filename` producendo una nuova immagine del processo corrente.

L'argomento `argv` è un array di stringhe che vengono passate alla funzione main del programma da eseguire. L'utimo elemento deve essere un puntatore a `null` e, per convenzione, il primo argomento è il nome del file che contiene il programma da eseguire.

Funzione: `int execl (const char *filename, const char *arg0, ...)`

La funzione è simile alla precedente, ma le srtringhe vengono inserite una ad una esplicitamente nella chiamata della funzione, l'ultima deve puntare a `null`.

### 1.1.5 Fine di un processo

Un processo normalmente temina quando il suo programma chiama la funzione exit.

Funzione: `void exit(int status)`

La funzione non ritorna alcun valore, il processo padre può leggere la variabile status del figlio.

Un processo padre può attendere la fine di un processo figlio con la funzione wait dichiarata nel file `sys/wait.h`.

Funzione: `pid_t wait (int *status-ptr)`

Questa funzione, eseguita dal processo padre, attende la fine di un processo figlio, la chiamata

```
wait (&status)
```

attende la terminazione di un figlio e ne restituisce lo stato.

## 1.2 Gestione dell'Input/Output

Prima di poter leggere o scrivere il contenuto di un file è necessario stabilire una *connessione*, ovvero un *canale di comunicazione* con quel file, questa operazione si chiama *aprire* il file. La connessione a un file aperto è rappresentata come un flusso (stream) o come un descrittore del file (un numero). che deve essere passata come argomento alle funzioni che eseguono la lettura o la scrittura, per indicare su quale file operare. Certe funzioni si aspettano di ricevere un flusso, altre un descrittore.

Quando si termina la lettura o la scrittura, la connessione può essere rilasciata con una operazione detta *chiusura* del file.

### 1.2.1 Descrittori di flussi e file

I descrittori sono oggetti di tipo `int` , numeri interi, mentre i flussi sono oggetti di tipo `FILE *` , puntatori a file.

I descrittori forniscono una interfaccia primitiva, di basso livello alle operazioni di input output. Il sistema operativo Unix inoltre non prevede differenze di trattamento tra file su disco, terminali, stampanti, pipe o FIFO (connessioni tra processi sulla stessa macchina) o socket (connessione tra processi su macchine diverse), quindi il descrittore o lo stream possono essere associati a ognuno di questi oggetti.

Gli stream forniscono un'interfaccia a un livello più alto, che impiega al suo interno le primitive che usano descrittori.

Il vantaggio principale nell'uso degli stream consiste nel disporre di un insieme di funzioni molto più ricche e potenti delle corrispondenti che operano su descrittori, per esempio funzioni di input e output formattato (printf e scanf) e funzioni orientate alle righe e ai caratteri. Dato che anche gli stream sono realizzati impiegando i descrittori si possono compiere operazioni su un file aperto usando entrambi i metodi, in genere è preferibile usare lo stream a meno che non si vogliano compiere operazioni particolari di basso livello, occorre ricordare inoltre che le operazioni su stream sono più facilmente trasferibile su altri sistemi operativi.

### 1.2.1 Posizione nel file

Per ogni file aperto è inoltre mantenuto un numero che indica la posizione, in byte dall'inizio del file, dove il prossimo byte sarà letto o scritto, questo numero sarà incrementato ad ogni scrittura o lettura, si parla quindi di *accesso sequenziale*, anche se esistono funzioni che cambiano questa posizione. Nel caso in cui un file sia aperto per una operazione di *append* la lettura o scrittura avviene alla fine del file.

Un file può anche essere aperto più volte, dallo stesso processo o da processi diversi, ciascuna delle aperture avrà un indicatore di posizione diverso.

### 1.2.3 Funzioni di Input/Output su stream

Tipo di dato: **FILE** dichiarato nel file header `stdio.h`

> È il tipo di dato usato per rappresentare oggetti stream e contiene tutte le informazioni sullo stato della connessione con il file associato, ad esempio la posizione nel file e la situazione del buffer. Ha inoltre indicatori di errore e fine file che possono essere letti con le funzioni appropriate.

Gli oggetti FILE sono gestiti dalle librerie di I/O, i programmi lavorano solo con puntatori a questi oggetti, non con gli oggetti stessi.

### 1.2.4 Stream standard

Quando la funzione `main` di un programma viene chiamata ha già tre stream aperti e disponibili, definiti nel file header `stdio.h`:

Variabile: `FILE * stdin` stream usato normalmente come input per il programma (tastiera)

Variabile: `FILE * stdout` stream usato normalmente come output del programma (video)

Variabile: `FILE * stderr` stream usato normalmente per i messaggi di errore (solitamente video)

Nella libreria GNU queste variabili possono essere modificate, coma tutte le altre, per esempio:

```
fclose (stdout);
stdout = fopen ("standard-output-file", "w");
```

L'esecuzione delle due funzioni chiude lo stream normalmente assegnato all'output standard e lo invia a un altro file.

### 1.2.5 Apertura e chiusura di stream

La funzione `fopen` crea un nuovo stream e stabilisce una connessione tra esso e un file (se non esiste potrebbe essere creato), è dichiarata nel file header `stdio.h` .

Funzione: `FILE * fopen (const char *filename, const char *opentype)`

> Apre un file e restituisce un puntatore al file ad esso associato.

L'argomento `opentype` controlla come il file viene aperto e specifica gli attributi dello stream:

- `r` apre un file esistente in sola lettura
- `w` apre un file per sola scrittura, se esiste viene azzerato altrimenti viene creato
- `a` apre un file per scrivervi nuovi byte alla fine, se non esiste viene creato
- `r+` lettura e scrittura, il file viene mantenuto e la posizione è all'inizio
- `w+` lettura e scrittura, se il file esiste viene cancellato, altrimenti creato
- `a+` lettura e append, se il file esiste la lettura avviene dall'inizio del file, mentre la scrittura aggiunge alla fine

Se la funzione fallisce, restituisce un puntatore a `null`

Quando si chiude uno stream con fclose, la connessione con il corrispondente file viene persa e non si possono eseguire altre operazioni, la funzione è dichiarata in stdio.h

Funzione: `int fclose (FILE *stream)`

La funzione ritorna 0 se chiusa correttamente e EOF se è stato rilevato un errore, per esempio di disco pieno

Funzione: `int fcloseall (void)`

Come la precedente ma chiude tutti gli stream aperti da un processo compresi quelli standard. Va usata solo in casi eccezionali.

Funzione: `FILE * freopen (const char *filename, const char *opentype, FILE *stream)`

Questa funzione è una combinazione di `fclose` e `fopen`. Prima chiude lo `stream` (terzo parametro), ignorando eventuali errori, poi apre il file il cui nome compare come primo parametro, nel modo specificato da `opentype` e lo associa allo stesso oggetto stream. se l'operazione fallisce, viene restituito un puntatore a `null`, altrimenti un puntatore allo stream. Può essere usata per ridirigere `stdin`.

### 1.2.5 Funzioni per l'output di caratteri e linee

Le funzioni che seguono sono dichiarate nel file header `stdio.h`

Funzione: `int fputc (int c, FILE *stream)` converte il carattere c nel tipo unsigned char e lo scrive sullo stream, ritorna EOF in caso di errore, altrimenti ritorna il carattere scritto

Funzione: `int putc (int c, FILE *stream)` come la precedente, tranne che spesso è implementata come macro.

Funzione: `int putchar (int c)` equivalente a `putc` con `stdout` come stream

Funzione: `int fputs (const char *s, FILE *stream)` scrive la stringa, senza terminatore `null` e senza `newline`

Funzione: `int puts (const char *s)` scrive la stringa su stdout con newline alla fine

### 1.2.6 Funzioni per l'input di caratteri

Le funzioni che seguono sono dichiarate nel file header `stdio.h` le funzioni tornano un `int` che può essere il carattere in input o `EOF` (di solito -1) in caso di errore. Prima di convertire il valore in `char` è opportuno verificare che non sia `EOF` per non scambiarlo col carattere, lecito -1 (FF in esadecimale)

Funzione: `int fgetc (FILE *stream)` legge un `char` dallo stream e lo ritorna dopo averlo convertito in `int`. In caso di errore di letture o fine del file ritorna `EOF`

Funzione: `int getc (FILE *stream)` come `getc` ma spesso è realizzata come macro e quindi più efficiente

Funzione: `int getchar (void)` equivalente a `getc` con `stdin` come argomento

### 1.2.7 Funzioni per l'input di linee

Talvolta è comodo leggere dallo stream di ingresso una intera linea di testo (fino al carattere newline), per questo la libreria GNU fornisce le seguenti funzioni, dichiarate in `stdio.h`

Funzione: `ssize_t getline (char **lineptr, size_t *n, FILE *stream)` la funzione legge una intera linea dallo stream immagazzinando tutta il testo incluso newline e lo 0 di terminazione, l'indirizzodel buffer viene fornito in `*lineptr`.

Funzione: `char * fgets (char *s, int count, FILE *stream)` la funzione legge i caratteri dello stream di input fino a newline, per un massimo di `count-1` e aggiunge uno 0

per marcare la fine della stringa. In caso di errore o di fine dello stream viene restituito un puntatore a `null`.

## 1.3 Primitive di Input Output

La funzioni `read` e `write` sono le primitive usate dal sistema per realizzare l'input e l'output su file, pipe e FIFO. Queste funzioni sono dichiarate nel file header `unistd.h`.

Tipo di dato: `ssize_t`

Questo tipo di dato è usato per rappresentare la dimensione dei blocchi che possono essere letti o scritti in una sola operazione.

Funzione: `ssize_t read (int filedes, void *buffer, size_t size)`

La funzione `read` legge fino a *size* dal file il cui descrittore è *filedes*, e immagazina il risultato in *buffer*. I byte non rappresentanp necessariamente caratteri.

Il valore tornato è il numero effettivo di dati letti, che può essere minore di size, per esempio perché nel file o nel FIFO ci sono meno caratteri disponibili. Quando la fine file è raggiunta il valore ritornato è zero, in caso di errore il valore è -1.

Normalmente, se non vi sono dati in ingresso immediatamente disponibili, la funzione attende un input (per esempio un carattere inserito da tastiera), è possibile però settare per il file il flag `O_NONBLOCK`, in questo caso la funzione torna subito, riportando un errore. La funzione `read` è la primitiva usata da tutte le funzioni che leggono da un flusso (*stream*).

Funzione: `ssize_t write (int filedes, const void *buffer, size_t size)`

La funzione `write` scrive fino a *size* byte prelevandoli da *buffer* nel file con descrittore *filedes*

Il valore ritornato è il numero di dati effettivamente scritto sul file, che può in ogni caso essere minore di quanto si voleva scrivere, per assicurarsi di scrivere tutti i dati conviene controllare da programma quesat eventualità ed eventualmente ripetere la scrittura per i bute restanti.

Nel caso di scrittura su disco `write` non assicura la memorizzazione permanente dei dati, il sistema operativo infatti può decidere di mettere insieme più scritture ed eseguirle assieme.

In caso di errore il valore ritornato è -1. Generalmente write attende il completamento dell'operazione con successo, il flag `O_NONBLOCK` impone la scrittura immediata, con eventuale errore.

La funzione `write` è la primitiva usata da tutte le funzioni che scrivono un flusso (*stream*).

## 1.4 Operazioni di controllo sui file

La funzione `fcntl` compie varie operazione di controllo sui file modificandone, per esempio i flag, ed è dichiarata nel file header `fcntl.h`. Ne esaminiamo solo alcune caratteristiche, rinviando alla documentazione GNU per una descrizione più completa.

Funzione: `int fcntl (int filedes, int command, ...)` la funzione esegue l'operazione specificata da `command` sul file il cui descrittore è `fildes`. Alcuni comandi richiedono argomenti addizionali, ne vediamo solo due.

`F_GETFL` recupera i flag associati a un file aperto

`F_SETFL` setta i flag associati a un file aperto

Dato che i flag sono rappresentati da singoli bit, per modificarne solo alcuni bisogna esegure la lettura dei vecchi flag, operare con AND o OR sui singoli bit e poi riscriverli.negli esercizi abbiamo usato il flag `O_NONBLOCK` per eseguire un read senza aspettare il carattere da tastiera.

## 1.5 Pipe e FIFO

Pipe (letteralmente tubo) e FIFO sono meccanismi per la comunicazione tra processi, dati scritti in un pipe possono essere letti da un altro processo nell'ordine in cui sono scritti (First In First Out, FIFO); la differenza tra loro è che il pipe non ha nome mentre il FIFO ha un nome come i file ordinari.

Un pipe o un FIFO può essere visto come un tubo con due estremità (una per la scrittura che immette dati nel tubo e l'altra per la lettura che toglie i dati dal tubo), si ahnno condizioni di errore sia se si legge da un pipe in cui nessuno scrive, sia se si scrive in un tubo da cui nessono legge, nulla esclude che a lscrivere e leggere sia lo stesso processo.

A differenza dei file, nei pipe e nei FIFO si scrive sempre appendendo dati alla fine del buffer e si legge sempre dall'inizio.

### 1.5.1 Creazione di un pipe

La primitiva per creare un pipe è la funzione pipe; l'uso tipico consiste nella creazione di un pipe immediatamente prima di una chiamata alla funzione fork() per creare un processo figlio e la pipe viene successivamente usata per la comunicazione tra genitore e figlio o tra processi fratelli. La funzione `pipe` è dichiarata nel file header `unistd.h`.

Funzione: `int pipe (int filedes[2])` La funzione `pipe` crea un pipe e mette due descrittori di file (numeri che identificano un file) rispettivamente in `fildes[0]` per la lettura e in `fildes[1]` per la scrittura. (per facilità di memeorizzazione si può ricordare che il descrittore 0 viene usato per il file `stdin` e 1 per `stdout`). In caso di successo la funzione ritorna il valore 0, in caso di fallimento -1. Si ricorda che un file che viene aperto ha un solo descrittore.

Esempio di creazione e uso di un pipe

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
/* Legge dei caratteri da un pipe e li invia a stdout */
void read_from_pipe (int file)
{
  FILE *stream;
  int c;
  stream = fdopen (file, "r");
  while ((c = fgetc (stream)) != EOF)
    putchar (c);
  fclose (stream);
}

/* Scrive del testo in un pipe */
void write_to_pipe (int file)
{
  FILE *stream;
  stream = fdopen (file, "w");
  fprintf (stream, "ciao a tutti\n");
  fclose (stream);
}

/* Il programma mostra l'impiego dei pipe */
int main (void)
{
  pid_t pid;
  int mypipe[2];

  /*Crea il pipe */
```

```
  if (pipe (mypipe))
    {
      fprintf (stderr, "Pipe fallito \n");
      return EXIT_FAILURE;
    }

  /* Crea il processo figlio */
  pid = fork ();
  if (pid == (pid_t) 0)
    {
      /* Codice del figlio
         chiude il pipe dal lato di scrittura e legge dell'altro */
      close (mypipe[1]);
      read_from_pipe (mypipe[0]);
      return EXIT_SUCCESS;
    }
  else if (pid < (pid_t) 0)
    {
      /* fallimento della fork */
      fprintf (stderr, "Fork fllita.\n");
      return EXIT_FAILURE;
    }
  else
    {
      /* Processo padre
         Chiude il lato di lettura e scrive */
      close (mypipe[0]);
      write_to_pipe (mypipe[1]);
      return EXIT_SUCCESS;
    }
}
```

### 1.5.3 File speciali FIFO

Un file speciale FIFO è simile a un pipe ma viene creato in un modo differente e gli viene attribuito un nome con la funzione `mkfifo` dichiarata ne file header `sys/stat.h`.

Funzione: `int mkfifo (const char *filename, mode_t mode)`

la funzione crea un FIFO con il nome filename, l'argomento mode è usato per stabilire i permessi per il file (lettura, scrittura) come per un file normale. In caso di successo ritorna il valore 0, in caso di fallimento -1.

## 1.6 Directory

Dato che il file system Unix è strutturato in modo gerarchico in directory, è importante saper gestire questa struttura, ricordiamo che una directory è un file che contiene informazioni sugli altri file, in essa contenuti, compresa la directory stessa (`.`) e la directory genitore (`..`).

### 1.6.1 Directory di lavoro

A ciascun processo è associata una directory chiamata directory di lavoro (*working directory*) che viene usata per risolvere nomi di file relativi (nomi che non cominciano con /) e per ogni utente è inizialmente posta alla sua directory home.

In questo paragrafo esaminiamo alcune primitive associate alla gestione delle directory i cui prototipi sono dichiarati nell'header `` `unistd.h' ``.

Funzione: `char * getcwd (char *buffer, size_t size)` La funzione restituisce un nome di file assoluto che rappresenta la directory corrente, lo memorizza nell'array di caratteri buffer di dimensione size. Il valore di ritorno è il puntatore a buffer in caso di successo o un puntatore a null in caso di insuccesso. Sono definiti i seguenti numeri di errore:

`EINVAL` :`size` è 0 e buffer non è un puntatore a null

`ERANGE` : `size` è minore della lunghezza del nome della directory

`EACCES` : non c'è permesso di accesso in lettura

Funzione: `int chdir (const char *filename)` Questa funzione viene utilizzata per cambiare la directory corrente di lavoro, `filename` punta alla stringa che denota la nuova directory. In caso di successo ritorna 0, in caso di errore -1. Oltre agli errori normali relativi alla creazione di file `errno` fornisce:

`ENOTDIR` : indica che il nome del file non rappresenta correttamente una directory

### 1.6.2 Funzioni per leggere il contenuto di una directory

Queste funzioni permettono di leggere i descrittori dei file contenuti in una directory. Tutti i simboli sono dichiarati nel file header `` `dirent.h' ``.

Tipo di dato: `struct dirent`

È la struttura che contiene le informazioni relative a ciascun file della directory, contiene i segunti campi:

`char d_name[]` la stringa (terminata con null) che contiene il nome del file.

`ino_t d_fileno` il numero che identifica il file

`unsigned char d_namlen` lunghezza del nome del file, null escluso

`unsigned char d_type` tipo del file, può assumere i seguenti valori

`DT_UNKNOWN` sconosciuto

`DT_REG` file regolare

`DT_DIR` directory

`DT_FIFO` fifo

`DT_SOCK` socket

`DT_CHR` dispositivo con accesso a carattere

`DT_BLK` dispositivo con accesso a blocchi

Un file con più nomi (link) ha più rappresentazioni nella directory, con lo stesso numero `d_fileno`

Tipo di dato: DIR

Rappresenta uno stream di tipo directory e crea una struttura di tipo dirent associata ad esso, a questi oggetti ci si riferisce mediante i puntatori restituiti dalle funzioni seguenti:

Funzione: `DIR * opendir (const char *dirname)` La funzione apre le directory di nome dirname e ritorna un puntatore che permette di leggerne il contenuto. In caso di errore ritorna un puntatore NULL , sono inoltre definiti seguenti codici di errore:

`EACCES` Permesso di lettura non concesso

`EMFILE` Il processo ha troppi file aperti.

Funzione: `struct dirent * readdir (DIR *dirstream)` La funzione legge il descrittore del file successivo, normalmente ritorna un puntatore alla struttura che contiene informazioni relative a quel file

Esempio di lettura del contenuto di una directory

```
//Il programma legge i nomi dei file contenuti nella directory corrente
#include <stddef.h>
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int main (void)
{
  DIR *dp;
  struct dirent *ep;

  dp = opendir ("./");
  if (dp != NULL)
    {
      while (ep = readdir (dp))
        puts (ep->d_name);
      (void) closedir (dp);
    }
  else
    puts ("Non posso aprire la directory.");
  return 0;
}
```

## 1.7 Segnali

I segnali sono interruzioni software inviate ai processi. Il sistema operativo usa i segnali per comunicare situazioni eccezionali ad un programma in esecuzione, per esempio un riferimento ad un indirizzo di memoria non valido o la disconnessione di una linea telefonica. I segnali possono essere gestiti dal processo mediante una funzione appositamente definita (handler). Anche i processi possono scambiarsi segnali, questo permette la sincronizzazione di processi separati. Possiamo suddividere i segnali in tre categorie:

1. segnali di errore (divisione per 0, accesso ad un indirizzo di memoria non valido);
2. segnali causati da una evento esterno (disponibilità di un canale di I/O);
3. segnali causati da una esplicita richiesta, mediante la funzionie `kill`.

In questo paragrafo ci occuperemo in particolare di questi ultimi, gli altri sono in genere gestiti dal sistema operativo.

Quando il segnale è inviato il processo che lo riceve esegue l'azione specificata per esso, questa può essere fissa (per esempio la terminazione del processo), ma in genere il programma può scegliere tra le seguenti opzioni:

1. ignorare il segnale;
2. gestirlo con una funzione specifica;
3. accettare l'azione di default per quel segnale.

La funzione `signal` determina quale azione verrà eseguita dal processo. Il numero corrispondente ad ogni segnale è definito nel file `signal.h` che deve essere incluso nel programma

### 1.7.1 Descrizione di alcuni segnali

I segnali sono definiti nel file `signal.h` mediante costanti simboliche, ne elenchiamo alcune.

int SIGKILL : causa la terminazione immediata del processo che lo riceve, non può essere ignorato, gestito o bloccato.

int SIGTERM : è il segnale generico per chiedere a un processo di terminare, a differenza di SIGKILL può essere ignorato o gestito dal programma che lo riceve.

int SIGALRM : viene usato per segnalare a un processo che un timer collegato all'orologio di sistema ha terminato il tempo ad esso assegnato.

int SIGIO : è inviato quando un file, collegato a un dispositivo di I/O, è pronto ad eseguire l'operazione richiesta.

int SIGSTOP : ferma un processo, non può essere gestito o ignorato.

int SIGCONT : riavvia un processo fermato.

int SIGPIPE : segnala che un pipe o un FIFO è interrotto, ciò accade se un processo apre un pipe in lettura prima che un altro inizi a scrivere.

int SIGUSR1, int SIGUSR2 : sono due segnali gestiti dall'utente, in genere per essi viene scritta una funzione di gestione che permette la sincronizzazione tra due processi

### 1.7.2 Gestione dei segnali

La funzione `signal` stabilisce il comportamento tenuto dal processo che riceve il segnale, il prototipo è definito nel file signal.h

Tipo di dato: `sighandler_t`

> Questo è il tipo delle funzioni che gestiscono i signal. Tali funzioni hanno un argomento intero che specifica il numero del segnale e non ritornano valori, pertanto devono essere dichiarate nel seguente modo:
>
> `void handler (int signum) { ... }`

Funzione: `sighandler_t signal(int signum, sighandler_t action)`

> La funzione `signal` stabilisce l'azione per il segnale identidificato dal numero `signum` (specificato con un nome simbolico, per esempio `SIGUSR1`, l'azione può essere una delle seguenti:

| | |
|---|---|
| `SIG_DFL` | azione di default per quel particolare segnale; |
| `SIG_IGN` | il segnale sarà ignorato; |
| *`handler`* | fornisce l'indirizzo (il nome) della funzione che gestirà il segnale |

La funzione `kill` può esser usata per inviare un segnale a un processo. Non causerà necessariamente, nonostante il nome, la terminazione del processo ma può essere usata per moltri altri scopi, per esempio:

- Un processo genera un processo figlio con un particolare compito, magari eseguito con un loop infinito e lo termina quando non ne ha più bisogno;
- due processi hanno bisogno di sincronizzarsi per lavorare insieme,
- un processo vuole informare un altro processo che terminarà.

La funzione kill è dichiarata in `signal.h`

Funzione: `int kill (pid_t pid, int signum)`

Invia il segnale il cui numero è `signal` al processo identificato da `pid.` Il valore ritornato è 0 se il segnale può essere inoltrato con successo, altrimenti ritorna –1. Normalmente la funzione kill viene usata solamente tra processo parenti.

#### Esempio di uso di un segnale

In questo esempio il processo padre genera un processo figlio e aspetta che questo abbia completato una procedura di inizializzazione. Il figlio lo informerà di essere pronto inviandogli il segnale `SIGUSR1`.

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

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

/* Quando arriva SIGUSR1 questa variabile verrà settata dalla funzione*/
volatile sig_atomic_t usr_interrupt = 0;

void synch_signal (int sig)
{
  usr_interrupt = 1;
}

/*handler del segnale SIGUSR1*/
void pronto(int signum)
{
usr_interrupt = 1;
}

/* Funzione eseguita dal processo figlio */
void child_function (void)
{
  /* Inizializzazione*/
  printf ("Sono pronto!!!  Il mio pid e' %d.\n", (int) getpid ());

  /* Segnala al padre che è pronto */
  kill (getppid (), SIGUSR1);

  /* Saluta e termina*/
  printf ("Ciao, ciao\n");
  exit (0);
}

int     main (void)
{

  pid_t child_id;

  /* stabilisce l'handler */
  signal(SIGUSR1, pronto);

  /* Crea il procseeo figlio */
  child_id = fork ();
  if (child_id == 0) child_function (); //Funzione eseguita dal figlio

  /* Aspetta il segnale dal figlio */
  while (!usr_interrupt);

  /* E continua */
  printf("Ho ricevuto il segnale dal figlio, ciao a tutti");
  return 0;
}
```

# 2 Esercizi

In questa parte della dispensa vengono riportati e commentati dei brevi programmi realizzati in C che hanno lo scopo di mostrare come il sistema operativo Unix supporti la creazione di più processi concorrenti e comunicanti tra di loro e come ne semplifichi la realizzazione.

I programmi sono volutamente brevi e semplici, ma i concetti sottostanti non sono banali, prima di realizzarli è opportuno avere chiari concetti come programma, processo, immagine di un processo in un sistema operativo multiprogrammato funzionante in time sharing. I programmi però possono interagire in modo proficuo con le lezioni teoriche.

### 2.1 Verifica dei comandi di visualizzazione su terminale

L'esecuzione dei programmi che seguono usa come `stdout` un terminale che emula un terminale VT100/ANSI, questo può essere un terminale virtuale di una shell grafica (Gnome o KDE) oppure può essere il terminale `telnet` di una rete intranet, spesso presente nei laboratori scolastici. Se si vuole che l'output sullo schermo del terminale non sia il banale scroll delle righe mandate dal programma sul dispositivo si possono usare delle stringhe di comando ANSI che permettono di scrivere nella posizione desiderata, questo breve programma ne verifica il funzionamento.

```
//prova codici di controllo VT100/ANSI
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>
```

***Includo gli header delle librerie di sistema anche se non sarebbe necessario in questo programma, si può proporre agli studenti, come esercizio, di mantenere solo quelle indispensabili***

```
int main(){
      int i;
      printf("\033[0H\033[2J");
            //porta il cursore nella posizione home e cancella lo schermo
```

***Le sequenze di caratteri di controllo riconosciute da un terminale ANSI sono costituite o da un solo carattere (codici< 20 esadecimale) o dal carattere Escape (1b esadecimale o 033 in ottale) seguito da una stringa di controllo. Nella stringa abbiamo usato due comandi, il primo*** `\033[0H` ***porta il cursore sulla prima riga, il secondo*** `\033[2J` ***cancella lo schermo fino alla fine.***

```
      printf( "PROVA HOME\n");
      for(i=0; i<5; i++){
        sleep(1);
        printf("\033[4H");//porta il cursore sulla Riga 4
        printf("PROVA RIGA 4: %d \n",(i+1));
        sleep(1);
        printf("\033[8H");//porta il cursore sulla Riga  8
        printf("PROVA RIGA 8: %d \n",(i+1));
        sleep(1);
        printf("\033[10;10H");//porta il cursore sulla Riga 10;Colonna 10
        printf("PROVA RIGA 10 COLONNA 10: %d \n",(i+1));
      }
   printf("\n\n");
   exit(0);
}
```

## 2.2 Creazione e terminazione di processi figli

### 2.2.1 Creazione di processi figli

L'esercizio seguente si propone i seguenti obiettivi:

- mostrare il meccanismo di creazione e terminazione di processi;
- mostrare come i processi creati evolvono indipendentemente l'uno dall'altro.
- chiarire il meccanismo della chiamata di sistema `fork()`

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>
//Numero di secondi di durata di ogni ciclo del processo
```

```
#define SEC_1 1
#define SEC_2 3
#define SEC_3 2

int main(){
int i,stato,n;
  pid_t pid[3];
  pid_t term[3];
  // padre
  printf("\033[0H\033[2J");//porta il cursore nella posizione home e cancella lo
schermo
  printf( "INIZIO di PROCESSI\n");
  //creo un processo figlio
  pid[0] = fork();
  if(pid[0]==0) { //figlio 1
      for(i=0; i<10; i++){
      sleep(SEC_1);
            printf("\033[4H"); //porta il cursore sulla riga 4
      printf("PROCESSO1: ciclo %d / %d s\n",(i+1),(SEC_1*(i+1)));
            }
    exit(1);
    } // figlio 1 - fine
  //genero un altro processo figlio
  pid[1] = fork();
  if(pid[1]==0) { // figlio 2
      for(i=0; i<10; i++){
            sleep(SEC_2);
            printf("\033[6H");//porta il cursore sulla riga 6
            printf("PROCESSO2: ciclo %d / %d s\n",(i+1),(SEC_2*(i+1)));
      }
    exit(2);
  } // figlio 2 - fine

  pid[2] = fork();
  if(pid[2]==0) { // figlio 3
      for(i=0; i<10; i++){
      sleep(SEC_3);
            printf("\033[8H");//porta il cursore sulla riga 8
            printf("PROCESSO3: ciclo %d / %d s\n",(i+1),(SEC_3*(i+1)));
      }
    exit(3);
  } // figlio 3 - fine
  //IL PADRE SCRIVE I PID dei figli
   if(pid[0]!=0){
            printf("\033[10H");//porta il cursore sulla riga 10
            printf( "PID PROCESSO 1 =%d\n",pid[0]);
    }
   if(pid[1]!=0){
            printf("\033[11H");//porta il cursore sulla riga 11
            printf( "PID PROCESSO 2 =%d\n",pid[1]);
    }
   if(pid[2]!=0){
            printf("\033[12H");//porta il cursore sulla riga 12
            printf( "PID PROCESSO 3 =%d\n",pid[2]);
    }
for(n=0;n<3;n++){ // attende la terminazione dei processi figli
      term[n] = wait(&stato);
      if((stato>>8)==1){
            printf("\033[13H");//porta il cursore sulla riga 13
      printf("Padre: terminato figlio %d PID %d\n",stato>>8,pid[0]);
            }
      if((stato>>8)==2){
            printf("\033[14H");//porta il cursore sulla riga 14
```

```
      printf("Padre: terminato figlio %d PID %d\n",stato>>8,pid[1]);
            }
      if((stato>>8)==3){
            printf("\033[15H");//porta il cursore sulla riga 15
      printf("Padre: terminato figlio %d PID %d\n",stato>>8,pid[2]);
            }
      }
printf("\033[18H");//porta il cursore sulla riga 18
printf("i figli sono terminati nell'ordine:
            %d %d %d \n",term[0],term[1],term[2]);
exit(0);
}
```

### 2.2.2 Creazione di un processo figlio che esegue un programma

Questo esercizio mostra come un processo figlio possa eseguire un programma diverso con i parametri passati dal padre. Per eseguire questo esercizio devono essere creati e compilati due programmi.

Primo programma: processo padre

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>

int main(int argc, char *argv[]){
  pid_t pid;
  int stato;
  char *tab=argv[1];//il primo parametro del programma viene attribuito a tab
             //numero di tabulazioni che verranno passate al processo figlio
  printf("CREAZIONE DI UN PROCESSO FIGLIO\n");
  pid = fork();
  if(pid==0) {
    // figlio
      char *argv[] = { "procnuovo", tab , "3", "5",(char *) NULL };
                       //tab viene passato al processo figlio
      execv("procnuovo", argv);
      exit(1);
      } //Fine del processo figlio

  else { //Il padre stampa il pid del figlio e aspetta che termini
       printf("Creato il processo figlio con PID=%d\n",pid);
       pid = wait(&stato); // attende la terminazione dei figli
       printf( "Padre: è terminato il processo figlio con PD=%d\n",pid);
  }
  exit(0);
}
```

Secondo programma: codice eseguito dal figlio

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
int main(int argc, char *argv[]){
//Il programma riceve quattro parametri, il primo argv[0] è il nome del file
//il secondo è un numero di tabulazioni che serve per l'output sullo schermo
//il terzo è la durata in secondi di ciascun ciclo, il quarto il numero di cicli
//gli argomento sono stringhe, prima di usarli li trasformo in numeri interi
//controllo inoltre che i valori non siano fuori dai limiti
  if(argc>=4){
    int tab, sec, cicli;
      int t,n;
```

```
    tab = atoi(argv[1]);
    if((tab<0) || (tab>7)) tab=0;
    sec = atoi(argv[2]);
    if((sec<1) || (sec>10)) sec=1;
    cicli = atoi(argv[3]);
    if((cicli<1) || (cicli>20)) cicli=1;

    for(n=0; n<cicli; n++) {
      sleep(sec);
      for(t=0; t<tab; t++) printf("\t");
      printf("Figlio: eseguo il ciclo n. %d secondi:%d\n", n+1, (n+1)*sec);
    }

  }
  else {//Messaggio in caso di errore
    printf( "SINTASSI: procnuovo <tab> <sec> <cicli>\n");
      printf ("0<tab<7, 1<sec<20, 1<cicli<20\n");
  }
  exit(0);
}
```

### 2.2.2 Creazione di più processi figli

Questo programma dimostra come un processo padre possa creare più processi figli che vengono eseguiti contemporaneamente

Primo programma: processo padre

```
/************************************************************************
il programma crea dei figli che durano il numero di secondi passati
come argomento, se il numero è 0 il padre termina, il processo figlio
si chiama procFiglio1
************************************************************************/

#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
 char n[5] ="aaaa";     //stringa che verrà passata al figlio
 pid_t pidFiglio;
 while(n[0]!='0'){
   printf("Introdurre la durata in secondi del processo figlio, 0 termina\n");
   fgets(n,3,stdin);
      if(n[0]!='0'){
            //creo un processo figlio
            pidFiglio = fork();
            if (pidFiglio==0) {
                  char *argv[]={"procFiglio1", n,(char*) NULL};
                  execv("procFiglio1",argv);
                  }
            }
      }
return 0;
}
```

Secondo programma: codice eseguito dal figlio

```
/************************************************************************
Il processo figlio esegue un conto alla rovescia a partire dal numero
di secondi che gli vengono passati dal padre
```

```
**********************************************************************/

#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>


int main( int argc, char *argv[])
{
int n;
n=atoi(argv[1]);
if((n>0) && (n<30)) {
      for (;n>=0;n--){
            printf( "Processo figlio %d secondi %d\n", getpid(),n);
            sleep(1);
            }
      }
exit(0);
}
```

## 2.3 Processi concorrenti

```
//Prova processi concorrenti che condividono stdin e stdout
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>
#include <fcntl.h>
int main(){
int stato,n;
  pid_t pid[2];
  pid_t term[2];
  char a[10];     //buffer per read
  // padre
  printf("\033[0H\033[2J\n");
  //porta il cursore nella posizione home e cancella lo schermo
  printf( "INIZIO di PROCESSI, INSERIRE 1 O 2 PER TERMINARLI \n");
  a[0]=0;
  //creo il primo processo figlio
  pid[0] = fork();
  if(pid[0]==0) { // figlio 1, attende il carattere 1
        fcntl(STDIN_FILENO,F_SETFL,O_NONBLOCK);
      //la funzione read non attende l'input, lo prende solo se c'è
        while (a[0]!='1')
            {
            read(STDIN_FILENO,a,2);
            //stampa un carattere catturato su stdin e attende 1(leggo anche /n)
            if (a[1]==10)
                  {
                  printf("\033[3H\n");//porta il cursore sulla riga 3
                  printf("PROCESSO1:Carattere inserito=
                              %c %d, con 1 si esce\n",a[0],a[1]);
                  a[1]=0;//toglie il carattere \n
                  }
            }
      exit(1);
  } // figlio 1 - fine

  //genero il secondo processo figlio
    pid[1] = fork();
  if(pid[1]==0) { //figlio 2, attende il carattere 2
```

```
      fcntl(STDIN_FILENO,F_SETFL,O_NONBLOCK);
    while (a[0]!='2')
            {
            read(STDIN_FILENO,a,2);
            if (a[1]==10)
                  {
                  printf("\033[5H\n");//porta il cursore sulla riga 5
                  printf("PROCESSO2:Carattere inserito=
                  %c %d, con 2 si esce\n",a[0],a[1]);
                  a[1]=0;
                  }
            }
   exit(2);
  } // figlio 2 - fine

     //IL PADRE SCRIVE I PID dei due figli
   if(pid[0]!=0){
            printf("\033[10H\n");//porta il cursore sulla riga 10
            printf( "PID PROCESSO 1 =%d\n",pid[0]);
    }
   if(pid[1]!=0){
            printf("\033[11H\n");//porta il cursore sulla riga 11
            printf( "PID PROCESSO 2 =%d\n",pid[1]);
    }

for(n=0;n<2;n++){ // attende la terminazione dei processi figli
      term[n] = wait(&stato);
      if((stato>>8)==1){
            printf("\033[13H\n");//porta il cursore sulla riga 13
      printf("Padre: terminato figlio %d PID %d\n",stato>>8,pid[0]);
            }
      if((stato>>8)==2){
            printf("\033[14H\n");//porta il cursore sulla riga 14
      printf("Padre: terminato figlio %d PID %d\n",stato>>8,pid[1]);
            }
      }
printf("\033[18H\n");//porta il cursore sulla riga 18
printf("i figli sono terminati nell'ordine: %d %d ",term[0],term[1]);
exit(0);
}
```

## 2.4 Processi che comunicano mediante pipe

```
//Il processo padre genera un figlio
//che legge caratteri da stdin e invia
//messaggi al padre tramite pipe
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>

int main(){
      pid_t pid;        //variabile per leggere i pid dei processi
      int i;
      int tubo[2];      //array per i descrittori del pipe
      char a[10];       //buffer per leggere stdin
      char scrivo[40]={"babbo ti invio il messaggio n.   carattere=    \n\0"};
                        //messaggio da inviare al padre
      char leggo[40];   //buffer per leggere il messaggio
      int npipe=0;      //numero di pipe
      char nstring[4];//stringa per il numero di pipe

      //Inizio del processo padre
```

```
	printf("\033[0H\033[2J");
	//porta il cursore in alto a sx e cancella lo schermo
	printf("INIZIO DEL PROCESSO PADRE\n");
	//creazione del pipe
	if(pipe(tubo)){
		printf("Pipe fallito");
		exit(-1);
		}
	//Crea il processo figlio
	pid=fork();
	if (pid==0){	//Codice del processo figlio
		close(tubo[0]);		//Chiudo il lato di lettura
		printf("\033[3H");	//Porto il cursore sulla terza riga
		printf("Processo figlio, un tasto invia un messaggio
			su pipe, f termina\n");
		do	{
			read(STDIN_FILENO,a,2); //legge 2 caratteri da stdin
			if (a[1]==10) {//solo se il secondo carattere è newline
				npipe++;
				sprintf(nstring,"%d",npipe);
				//converto il numero in stringa
				i=0;
				while(nstring[i]) {
					//copio il numero sulla stringa scrivo
					scrivo[i+27]=nstring[i];
					i++;
					}
				scrivo[37]=a[0];  //inserisco anche il carattere
				write(tubo[1],scrivo,40);
				//invio il messaggio sul pipe
				a[1]=0; //per evitare di ripetere if
				}
			}
		while(a[0]!='f'); //condizione di uscita di entrambi i processi
		exit(0);	//termina il processo figlio
	 }
	 else{ //eseguito dal processo padre
		 printf("\033[10H");
		//sposto il cursore sulla riga 10
		 printf("PID FIGLIO=%d\n",pid);
		//fork ha restituito il pid del figlio al padre
		 printf("PID PADRE=%d\n",getpid());
		//la funzione recupera il pid del processo corrente
		 printf("Il processo padre legge il pipe e termina con f\n");
		 close(tubo[1]); //chiude il lato di scrittura
		 do	{	//anche il padre esce con f
			 if(read(tubo[0],leggo,40)) printf("\033[18H %s",leggo);
				 //il padre scrive quanto ricevuto su stdout
			}
		 while(leggo[37]!='f');//condizione di uscita anche del padre
		 exit(0);
	 }
 }
```

## 2.5 Processi che comunicano tramite FIFO

### 2.5.1 Legge da un fifo

```
//Il processo legge un fifo
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <stdlib.h>
```

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

int main(){
      int i;
      int fd_fifo;      //descrittore del fifo
      char a[30];       //buffer stringhe in lettura
      printf("\033[0H\033[2J");
      printf("Processo che legge un messaggio tramite FIFO\n");
      unlink("fifo1");             // elimino un eventuale fifo preesistente

      mkfifo("fifo1",0666);        //creazione del fifo con relativi permessi

      fd_fifo=open("fifo1",O_RDONLY);
      printf("fifo n. %d\n",fd_fifo);
      for (i=0;i<4;i++)
            if(read(fd_fifo,a,30)) printf("%s",a);
      close(fd_fifo);
      unlink("fifo1");//elimino il fifo
      exit(0);
}
```

2.5.2 Scrive su un fifo

```
//Il processo scrive su un fifo che un altro può leggere
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

int main(){
      int i;
      int fd_fifo;      //descrittore del fifo
      char *stringa;    //puntatore alle stringhe
      char a[30];       //buffer stringhe in scrittura
      char scrivo[4][30]={{"Caro amico ti scrivo\n\0"},
                          {"cosi` mi distraggo un po'\n\0"},
                          {"e siccome sei molto lontano\n\0"},
                          {"piu` forte ti scrivero`\n\0"}};
                           //messaggio da inviare tramite fifo

      printf("\033[0H\033[2J");
      printf("Processo che invia un messaggio tramite FIFO\n");
      fd_fifo = open("fifo1",O_WRONLY);   //apro il fifo in lettura/scrittura

      for (i=0;i<4;i++)
            {
            sleep(2);
            stringa=&scrivo[i][0];//fornisco l'indirizzo
            sprintf(a,"%s",stringa);
            write(fd_fifo,a,30);
            }

    exit(0);
}
```

| | |
|---|---|
| **Titolo:** | Esercizi sulla programmazione di sistema GNU/Linux |
| **Autore:** | Luciano Viviani |
| **Email:** | luciano_viviani@libero.it |
| **Classe:** | QUARTA INFORMATICA (4IB) |
| **Anno scolastico:** | 2004/2005 |
| **Scuola:** | Itis Euganeo<br>Via Borgofuro 6 - 35042 Este (PD) - Italy<br>Telefono 0429.21.16 - 0429.34.72<br>Fax 0429.41.86<br>http://www.itiseuganeo.it<br>informazioni@itiseuganeo.it |
| **Note legali:** | Diffusione consentita con obbligo di citarne le fonti |