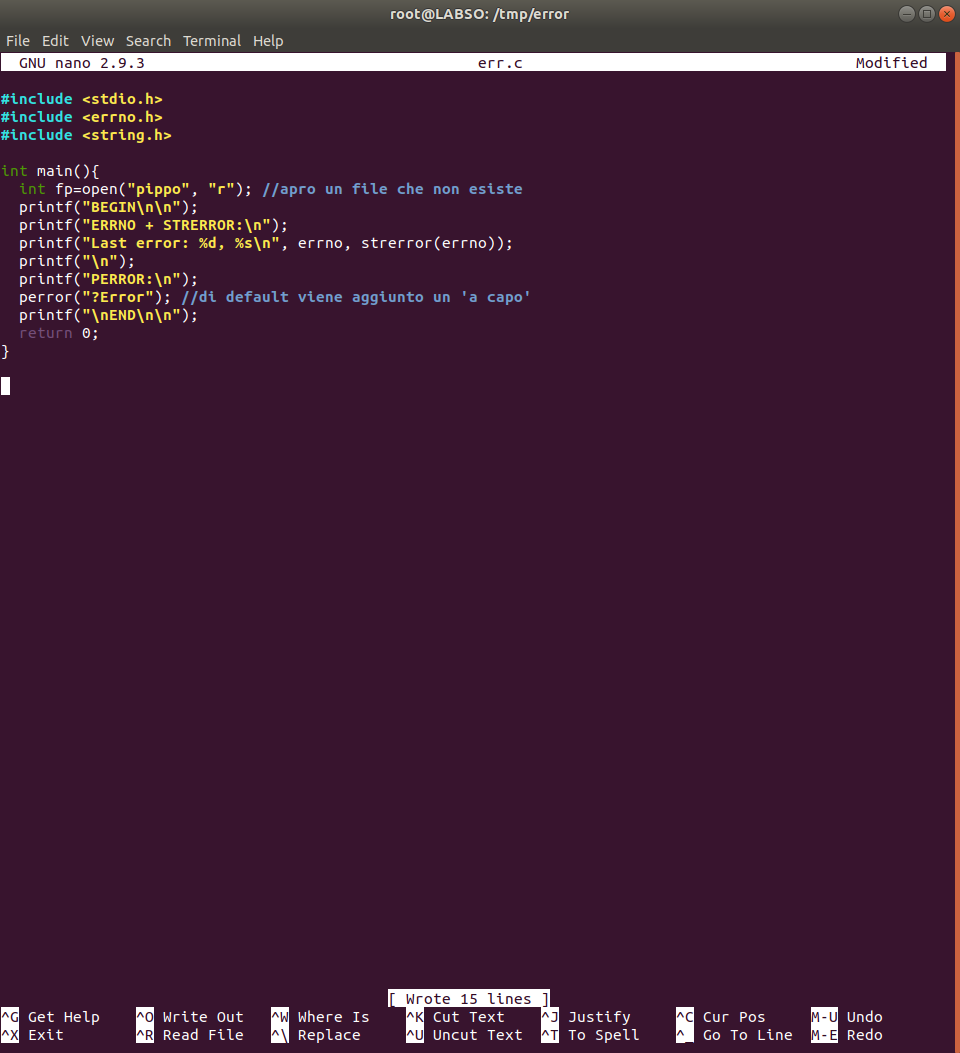
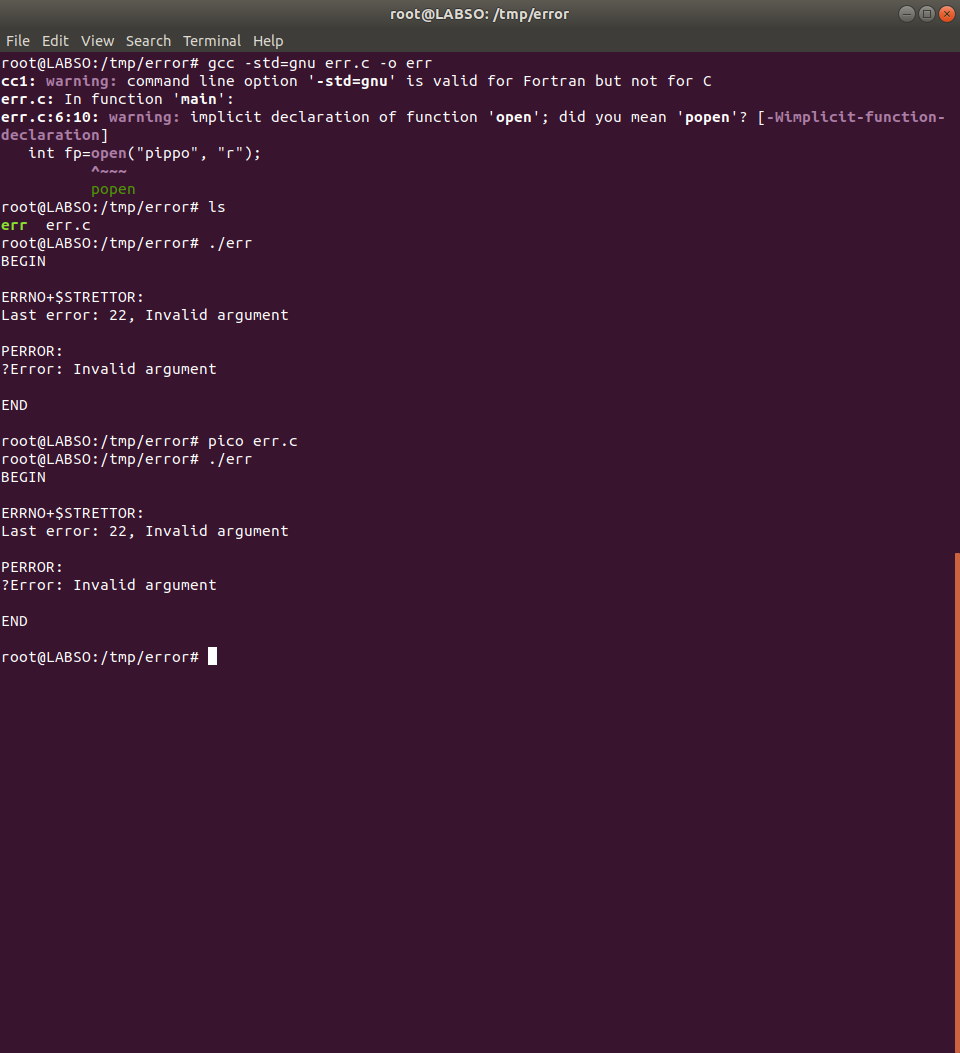
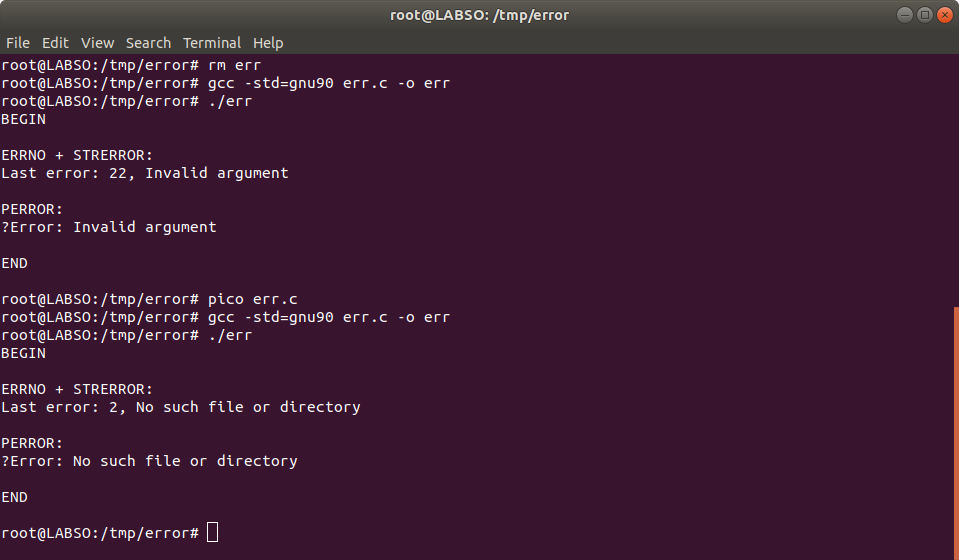
**Laboratorio Sistemi Operativi: Piping**

* **Error**:
  + **Librerie**: errno.h e string.h
  + **errno**:
    - **Variabile d’errore globale** che **contiene l’ultimo errore** che si è verificato (inteso come **valore numerico**).
    - Eseguendo una syscall, questa **può generare un errore** o restituire un valore d’uscita; questo valore di ritorno viene salvato nella variabile **errno**.
    - Ogni syscall eseguita, salva il suo valore di ritorno nella variabile **errno**, **sovrascrivendo quello precedentemente salvato**.
  + **strerror(int errno)**:
    - Funzione che prende come **argomento un codice d’errore** e **ritorna una stringa** (vettore di char) **descrittiva che riguarda l’errore** che si è generato.
    - Viene usata per **stampare a video un messaggio d’errore testuale** **associato all’errore** numero passato come parametro.
    - Partendo da un codice numerico, riusciamo a stampare un messaggio testuale d’errore.
    - La **lista di messaggi** è disponibile all’interno del **vettore globale sys\_errlist** (si accede via indice al messaggio contenuto).
  + **perror(char \*msg)**:
    - Funzione che **accetta come argomento una stringa** e **stampa a video**:

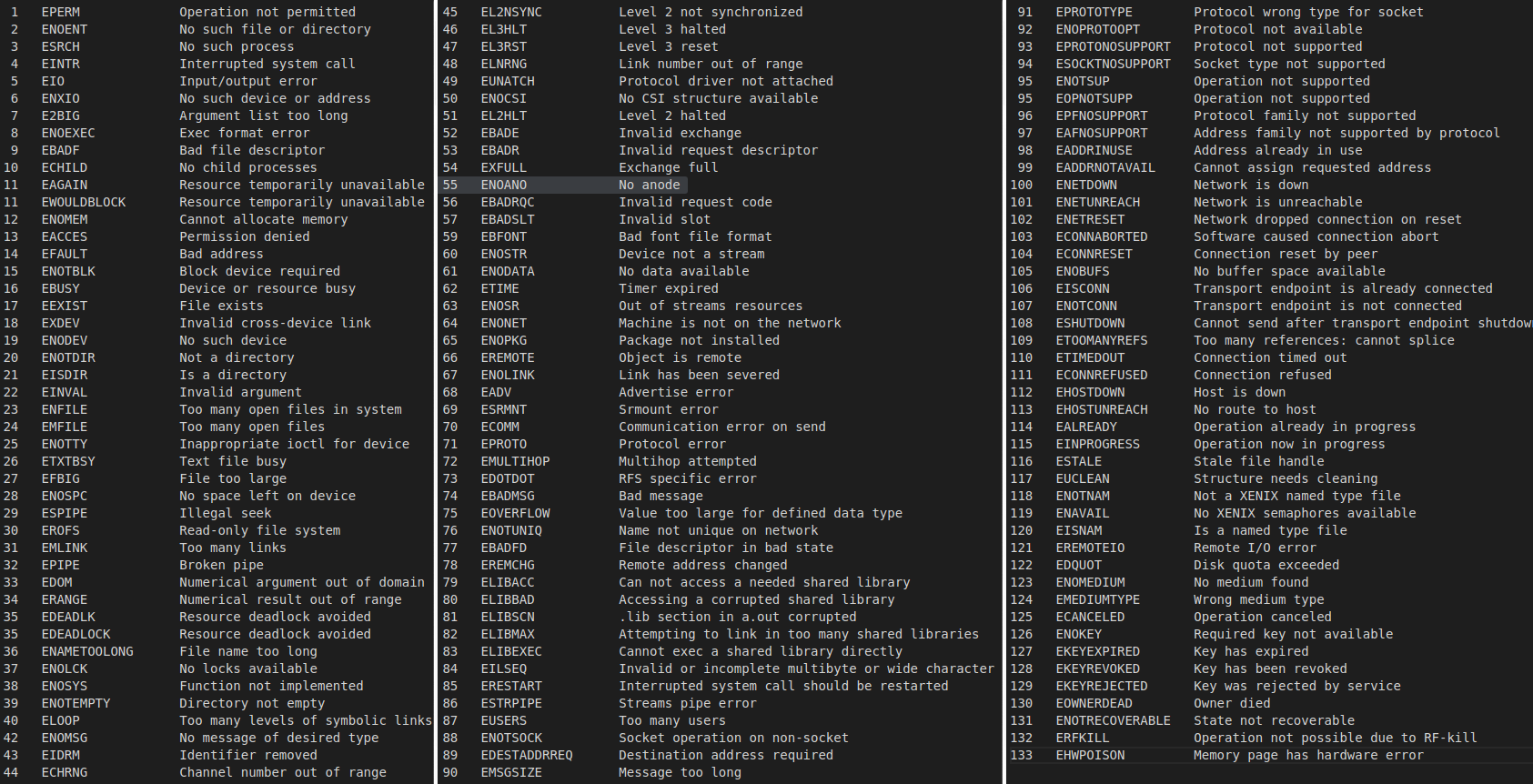
**msg + “:” + <messaggio d’errore testuale>**

* + - **Errore stampato è equivalente a quello che si ottiene con strerror()**.

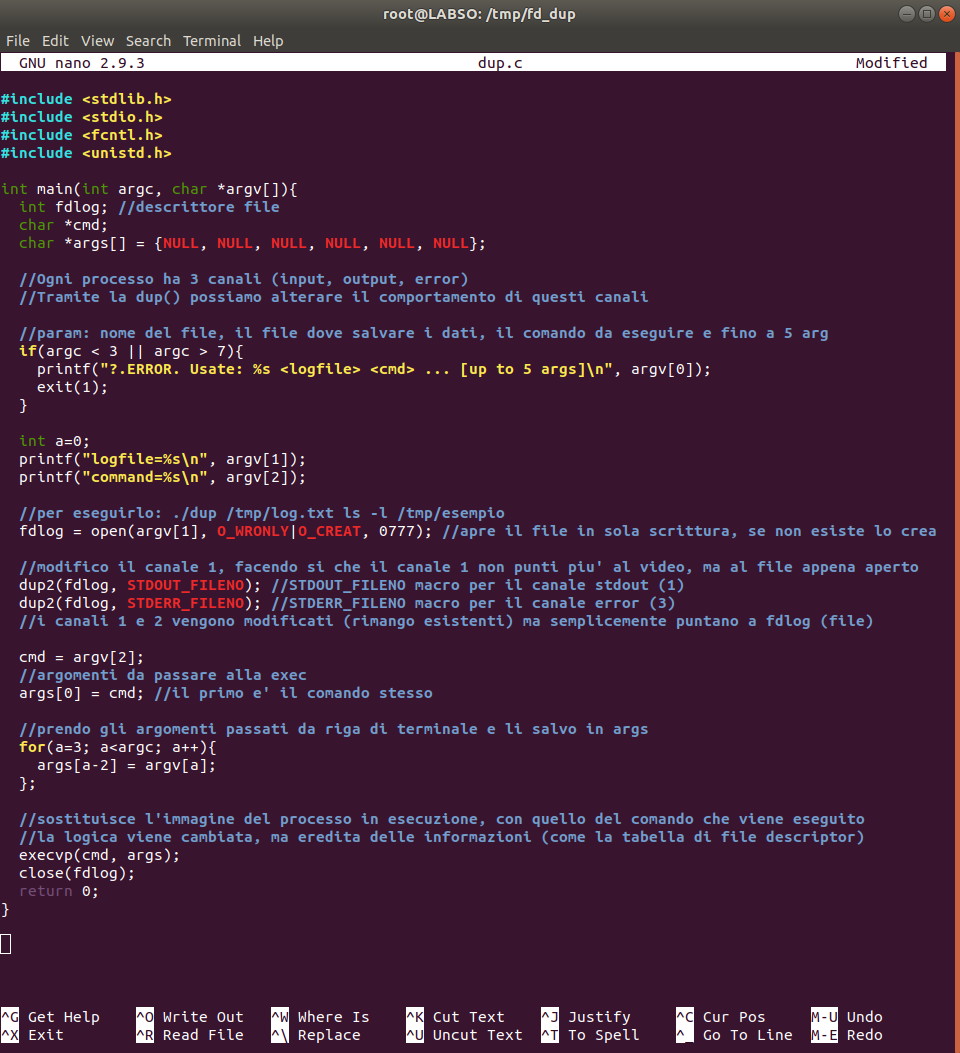
 

* + Valori di error:



* **File Descriptor**:
  + **Operazioni base**:
    - **open**/**close**/**read**/**write**
  + **Operazioni avanzate**:
    - **lseek**:
      * Permette di **posizionare l’indice di lettura/scrittura** del file in un punto specificato.
    - **int dup(int oldfd)**:
      * Accetta come argomento un **unico parametro intero** che è un **riferimento ad un file descriptor**.
      * Una volta eseguita, **si ottiene un nuovo indice che punta allo stesso file** (avremo due indici nella tabella dei file descriptor che referenziano lo stesso file).
    - **Int dup2(int oldfd, int newfd)**:
      * Accetta come argomento **due interi**:
        + **oldfd** è il **riferimento al file descriptor del file che si vuole duplicare**.
        + **newfd** è il **nuovo riferimento che si vuole usare per riferirsi al** **file**.
      * Permette anche una **sovrascrittura**, newfd potrebbe essere un indice che referenzia un altro file, ma usando la funzione dup2(), questo viene sovrascritto con l’indice passato.
    - **dup()** e **dup2()** sono **funzioni che duplicano il file descritor** e sono utili quando si eseguono funzioni come exec e la fork, per far si che i nuovi processi ereditino queste modifiche alla tabella dei descrittori di file.

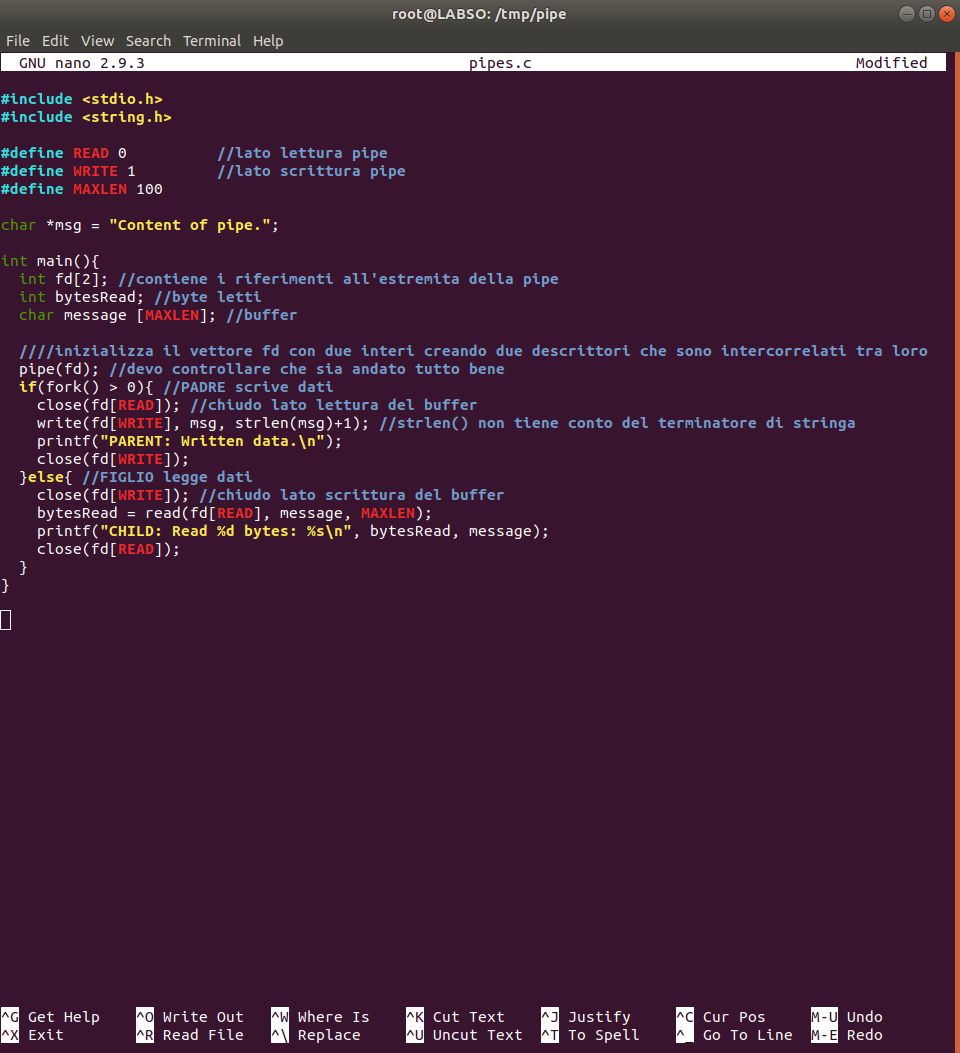


* **Piping**:
  + Esempio:
    - **ls | more**:
      * Questi due comandi vengono eseguiti su **due processi diversi in concorrenza**.
      * **Viene settato un buffer** tra i due processi:
        + **ls è un comando che genera dati** e di default questi dati vengono inviati nel buffer del canale stdout (canale 1) **ma** **tramite il comando di pipe ‘|’**, questi dati non vengono mandati nel canale standard ma **vengono inviati nel buffer che si è creato tra i due processi** (il suo canale d’uscita viene “dirottato”).
        + Questo stesso **buffer è dato in lettura al comando more** che **prende i dati generati da ls in input**, gli elabora e a sua volta genera altri dati.
        + Questi **dati** però **vengono mandati** sul buffer standard del **canale stdout** (non ci sono altre indicazioni dopo il comando more).
      * Il questo **buffer** (che è stato settato tra i processi):
        + Se il **processo** che prova ad **inserire dati**, trova il **buffer pieno**, **interrompe la sua esecuzione e aspetta** **che si liberi spazio** prima di andare avanti.
        + Se il **processo** che prova a **leggere i dati**, trova il **buffer vuoto**, **interrompe la sua esecuzione e aspetta che vengano generati i dati**.
      * È presente una sorta di **sincronizzazione** tra i processi.
  + **Piping Anonimo**:
    - Vengono generati dei buffer che non hanno un nome, possono essere riferiti solo dai processi che stanno interagendo in quel momento (come nell’esempio precedente).
    - Collega due processi che hanno un antenato in comune (nell’esempio precedente, l’antenato in comune è la shell).
    - Al buffer sono associati **2 lati**:
      * **Lato di scrittura** (ls scrive nel buffer).
      * **Lato di lettura** (more legge i dati).
    - La **comunicazione** standard è di tipo **unidirezionale**:
      * **P1** **crea la pipe** eseguendo la chiamata di sistema (genera due descrittori).
      * **P1** **esegue una fork** che genera un **nuovo processo P2** che, **ereditando** la **tabella dei descrittori di file**, **eredita** anche i **riferimenti** di questi due descrittori particolari che sono collegati tra loro.
      * **P1 chiude uno dei due descrittori** (lato scrittura o lettura) con la close().
      * **P2 chiude l’altro descrittore**.
      * Ora **P1 e P2 condividono il** **buffer**, uno in scrittura e uno in lettura.
      * Quando hanno terminato la comunicazione, chiudono i restanti descrittori.
    - Se si volesse avere una **comunicazione bidirezionale**:
      * **P1** **crea 2 pipe** (otteniamo 4 descrittori)
      * **P1 esegue una fork** che genera un nuovo processo P2, che ereditando la tabella dei descrittori di file, eredita anche i riferimenti di questi quattro descrittori.
      * **P1 chiude** da una pipe il **lato di scrittura** e **dall’altra** pipe **quello** di **lettura**.
      * **P2 simmetricamente esegue la stessa operazione** chiudendo gli altri lati.
      * In questo modo **P1 e P2 possono entrambi leggere e scrivere** **usando** però 2 **buffer diversi**.
      * Quando hanno terminato la comunicazione, chiudono i descrittori.
    - Tramite la chiamata di sistema **pipe(…) è possibile creare il buffer** di comunicazione:

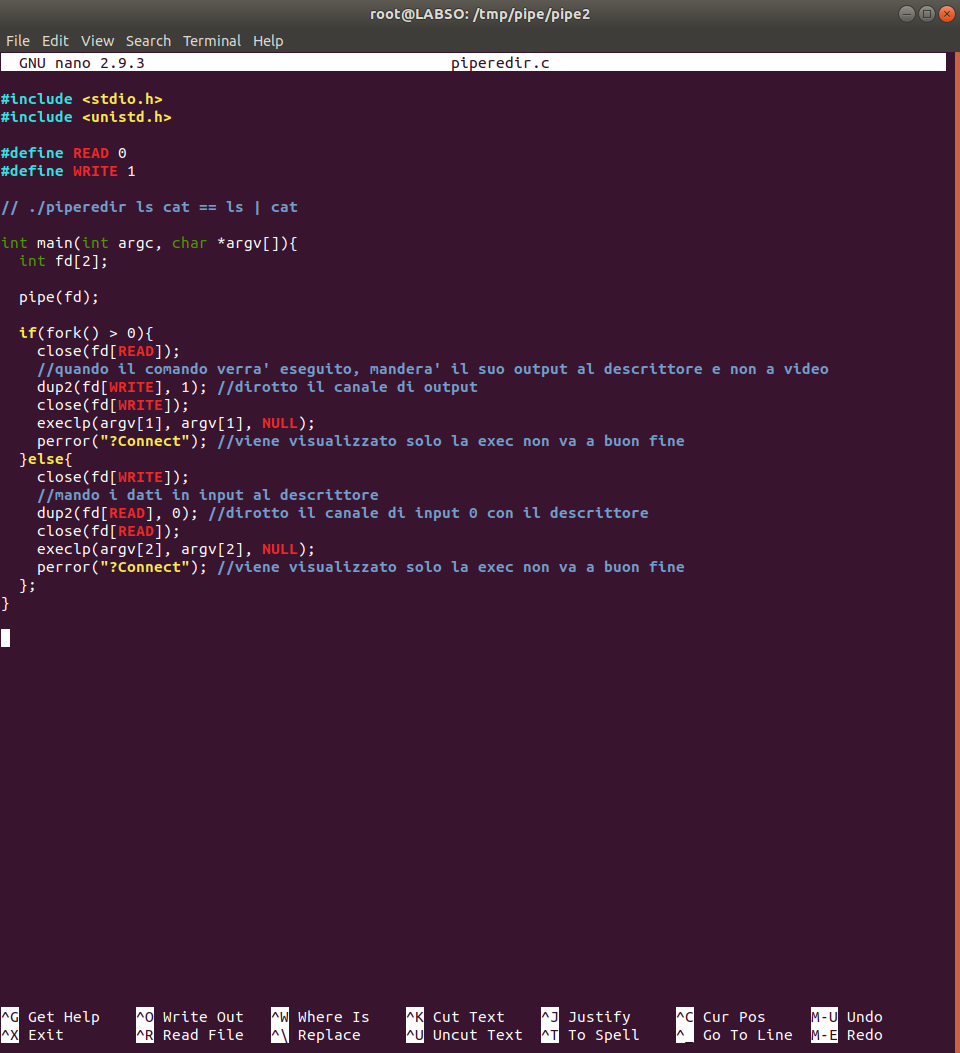
**int pipe(int fd[2]);** //funzione si aspetta un vettore di due interi

* + - * Con la chiamata **pipe(fd)**, **fd** che è un **vettore di due interi**, **viene inizializzato** con due valori (**due indici della tabella del descrittore di file**) che rappresentano:
        + **fd[0]**: lato **lettura**
        + **fd[1]**: lato **scrittura**
    - Anziché creare con la open(…) dei canali di comunicazione sul file system (creazione e uso di file per la comunicazione), **possiamo utilizzare la funzione pipe(…) per creare due descrittori di file** (generati in contemporanea dal sistema) **collegati tra loro** che **fanno riferimento alle due estremità del buffer** che viene creato.
    - È come se avessimo **creato due riferimenti ad un file** con la particolarità che questi due riferimenti, **puntano allo stesso contenuto** (buffer) **su due punti diversi** (i riferimenti rappresentano il lato di scrittura o di lettura).
    - Per la lettura:
      * **read(fd[0], data, num)**:
        + **fd[0]**: **descrittore** del **lato della lettura** dal buffer.
        + **data**: variabile dove **salvare i dati** letti.
        + **num**: **quantità** di **dati da** **leggere** dal buffer.
      * Se il **lato di scrittura è stato chiuso**, la funzione **restituisce 0**.
      * Se il **lato di scrittura è aperto** ma il **buffer è vuoto**, la **read(…)** aspetta i dati **sospendendo il processo** (o lo termina).
      * Se il **lato di scrittura è aperto** e **il buffer contiene** **dati**, **leggo i dati disponibili** (se voglio leggere una quantità di dati maggiore rispetto a quella presente del buffer non ho errori, ma possono rilevarlo in quanto la funzione **read(…) restituisce il numero di elementi letti**).
    - Per la scrittura:
      * **write(fd[1], data, num)**:
        + **fd[1]**: **descrittore** del **lato di scrittura** sul buffer.
        + **data**: variabile dove **salvare i dati** letti.
        + **num**: **quantità** di **dati da leggere** dal buffer.
      * Se il **lato di lettura è stato** chiuso, la funzione **write(…) fallisce e invia un segnale di SIGPIPE al processo stesso** (operazione che di default termina il processo, ma possiamo sovrascrivere l’handler di questo segnale).
      * Se il **lato di lettura è aperto** ma il **buffer è pieno**, la **funzione sospende il processo** **aspettando che si liberi spazio** nel buffer.
      * Se il **lato di lettura è aperto** e il **buffer non è pieno**, **posso scrivere i dati** (se la dimensione dei dati che voglio scrivere è inferiore alla dimensione del buffer o dello spazio libero rimasto, l’operazione è atomica; altrimenti può darsi che l’operazione non sia atomica, viene eseguita in più cicli).
    - È necessario **definire un protocollo di comunicazione** tra i processi per evitare problemi di sincronizzazione.

Esempio 1:



Esempio 2:

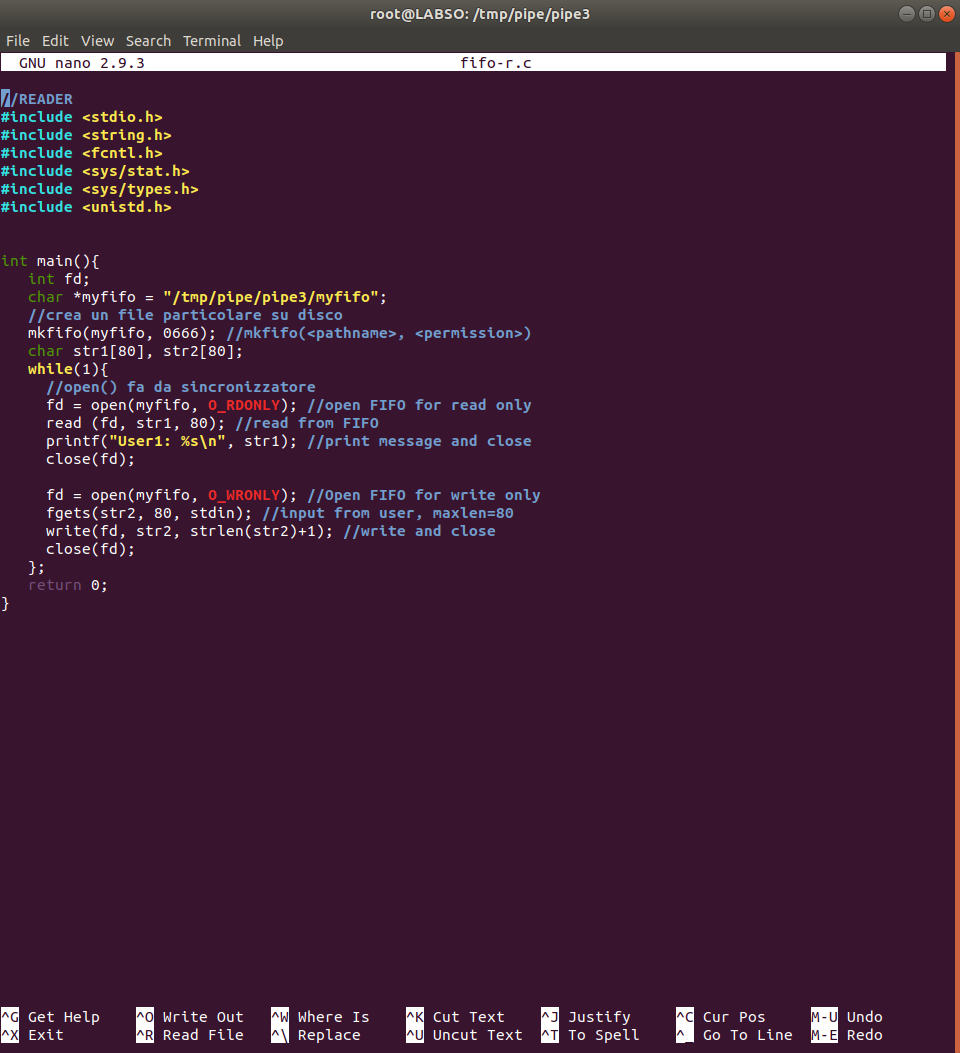


* **Piping named (fifo):**
  + Librerie: sys/types.h e sys/stat.h
  + **Creano un buffer che ha una denominazione** propria ed è gestito mediante dei **file particolari** salvati sul file system.
    - Sono **file persistenti**.
  + **Qualunque processo che conosca il riferimento di questi file speciali può accedervi** e usarli (se ha i permessi).
  + **Creazione** mediante la funzione:

**int mkfifo(const char \*pathname, mode\_t mode);** //path del file, permessi

* + Esempio:

**Processo READER**



**Processo WRITER**

