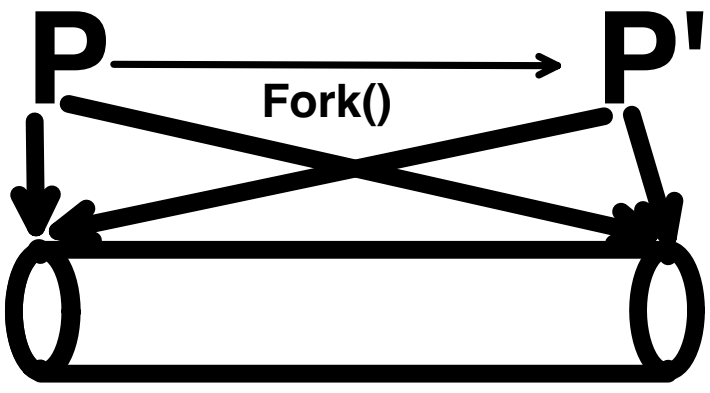
Esempio (manca)

mercoledì 3 maggio 2023 12:32

Example - Easy

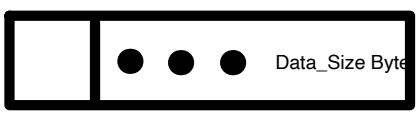
Quello che noi abbiamo all'interno di questo programma è la realizzazione di un'applicazione in cui noi abbiamo sicuramente una fork(). Utilizziamo le PIPE unix esattamente per costruire questo ambito:
Abbiamo un processo P, CHE ESEGUE UNA FORK, e viene generata P'. Ma prima di generare P', il processo P ha anche generato una PIPE. Il processo P può arrivare sui due estremi della PIPE, ma la stessa cosa è vera per P': anche P' può arrivare a scrivere e leggere, perché quando P' viene generato tramite la fork eredita i canali di I/O tramite la duplicazione della tabella dei file descriptor di P, per arrivare sulla PIPE.



Abbiamo due processi che utilizzano questa PIPE, ad uno dei due facciamo inserire byte e all'altro facciamo estrarre i byte inseriti, quindi abbiamo un flusso che va da un processo ad un altro.
E quindi possiamo in qualche modo implementare - utilizzando questa PIPE - questo scambio di informazioni tra i due processi.
Nel codice la prima cosa che viene fatta all'interno del main è proprio la creazione di una PIPE.

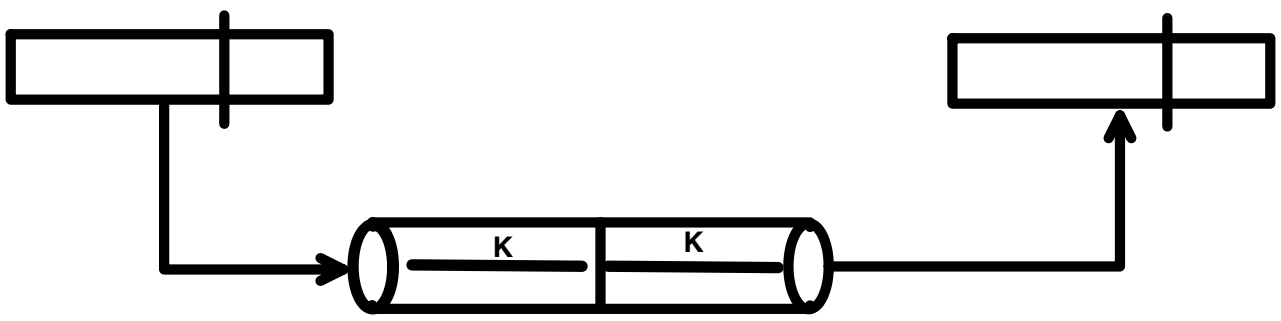
```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

#define Errore_(x) { puts(x); exit(EXIT_FAILURE); }

#define DATA_SIZE 1024

int main(int argc, char *argv[]) {

        char messaggio[DATA_SIZE];
        int  pid, status, fd[2];
    int ret;

        ret = pipe(fd); /* crea una PIPE */
        if ( ret == -1 ) Errore_("Errore nella chiamata pipe");

        pid = fork();   /* crea un processo figlio */
        if ( pid == -1 ) Errore_("Errore nella fork");


        if ( pid == 0 ) {    /* processo figlio: lettore */
               close(fd[1]); /* il lettore chiude fd[1] */
               while( (ret = read(fd[0], messaggio, DATA_SIZE)) > 0 ){
                    printf("processo %d - letto messaggio: ", getpid());
                fflush(stdout);
                write(1,messaggio,ret);
              }
               close(fd[0]);
        }

    /* processo padre: scrittore */
        else {

            close(fd[0]);

            printf("processo %d - digitare testo da trasferire (quit per terminare):\n",getpid());

            do {
                fgets(messaggio,DATA_SIZE,stdin);
                write(fd[1], messaggio, strlen(messaggio));
                printf("processo %d - scritto messaggio: %s", getpid(), messaggio);
        fflush(stdout);
            } while( strcmp(messaggio,"quit\n") != 0 );

            close(fd[1]);

            wait(&status);
        }

    }
```

Fd è un puntatore ad un'area di memoria dove ci sono almeno due interi. Quindi il sistema ha la possibilità di rendere i canali noti per arrivare sulla suddetta PIPE. Chiamiamo la fork(), viene generato un processo e se sono nel processo figlio, opero da lettore. Quindi attenzione: la prima cosa che devo fare è chiudere il canale di scrittura su quella PIPE, perché altrimenti potrei rischiare di chiamare delle letture per leggere dati che potrei inserire solo io sulla PIPE, COSA CHE NON DEVE SUCCEDERE per evitare i deadlock.
Quindi io chiudo il canale di scrittura, ho comunque il canale di lettura valido, e quindi in un while chiamo delle letture da questo canale, gli passo f[0], chiamo una read e cerco di farmi dare i dati all'interno di un area di memoria lunga 1024 byte, ossia messaggio, e infine chiedo al più quanti dati posso ospitare all'interno di questo array.
Finchè questa read mi ritorna un valore maggiore di zero, eseguo le operazioni associate all'interno del while: questa read ha chiesto di estrarre data_size:



Messaggio

La read ci va a restituire sull'array al più data_size byte, ma la stringa può essere rappresentata come un sottoinsieme di quei byte, può essere più piccola, io ho cercato comunque di estrarne DATA_SIZE. Questo ci introduce un problema da considerare quando lavoriamo sulle PIPE, e più in generale sugli stream di I/O, che riguarda il fatto che quando noi lavoriamo chiamando delle system call, specifichiamo il numero di byte che vogliamo estrarre o scrivere su questi oggetti di I/O, ma all'interno di questo numero di byte poi è possibile che ci sia un'informazione di interesse utilizzando una quantità minore rispetto ai byte che noi stiamo cercando di estrarre.
Questa comunicazione che stiamo cercando di effettuare utilizzando lo streaming, va avanti utilizzando una sorta di "PADDING". Colui che eventualmente ha scritto all'interno della PIPE le informazioni che io adesso sto estraendo per andarle a caricare all'interno del mio array messaggio, ha scritto una stringa più piccola. I byte che non sono stati digitati specificatamente sono riestratti comunque successivamente. Questo padding ci permette di dire che lo stream che noi abbiamo è fatto da tante porzioni fatte ciascuna dallo stesso numero di byte K, in particolare il numero dei byte che rappresenta la taglia dei buffer che stiamo utilizzando.



Dobbiamo stare molto attenti quando cerchiamo di scrivere una stringa all'interno dello stream. Se scriviamo solo la stringa, dall'altro lato non si sa quanti byte devono essere estratti per prelevare esattamente questa stringa, chiaramente chi legge le informazioni che sono scritte qui in streaming non sa che informazioni sono estratte o precedentemente scritte. Quindi l'idea è che se vengono scritte queste informazioni "cadenzandole" con un certo numero di byte ad ogni scrittura, estraendo questo numero di byte riesco a poter estrarre esattamente la stringa che era stata scritta precedentemente.

Ovviamente sul lato del parent, quindi sulla parte dello scrittore, chiudiamo fd[0], e poi andiamo a chiamare questa print per mandare in output il messaggio che ci dice chi siamo, e poi abbiamo un do-while prendiamo con fgets() delle informazioni da standard input e le scriviamo nel buffer messaggio, al più prendiamo una quantità di byte pari a Data_Size (riga 44), e poi andiamo a scrivere esattamente questa quantità di byte su quella pipe f[1] prelevando le informazioni che avevamo ricevuto all'interno dell'area messaggio ovviamente, e poi andiamo in print ad indicare che abbiamo eseguito questa operazione.

Questo do-while va avanti finche non verifichiamo che l'ultima stringa che abbiamo letto, ossia l'ultima sequenza di byte che abbiamo letto utilizzando fgets() che abbiamo scritto all'interno di questo array "messaggio", sia pari esattamente a "quit". Alla fine chiudiamo anche fd[1] e infine quando non dobbiamo più scrivere più niente, attendiamo l'altro processo con una wait di status.

Andiamo a compilare questo programma.

```
/baseline-pipe-example> ./a.out
processo 32234 - digitare testo da trasferire (quit per terminare):
```

A questo punto abbiamo due processi attivi - il child a ricevuto a valle della fork() i due canali che possono essere utilizzati per lavorare sulla PIPE ed è in attesa che arrivino dati da questa PIPE. Ovviamente i dati possono arrivare, possiamo scrivere quello che vogliamo.

```
/baseline-pipe-example> ./a.out
processo 32234 - digitare testo da trasferire (quit per terminare):
;lmfe;kfm ew;lfew;kmew ew;lmg;ewkm
processo 32234 - scritto messaggio: ;lmfe;kfm ew;lfew;kmew ew;lmgprocesso 32234 - scritto messaggio: ;ewkm
process 32235 - letto messaggio: ;lmfe;kfm ew;lfew;kmew ew;lmgprocess 32235 - letto messaggio: ;ewkm
```

Abbiamo che il processo 32234 che era appunto lo scrittore, ci va a dire che è stata scritta la linea che abbiamo fornito in input, e l'altro processo ci sta indicando le stringhe che sono state lette.

```
/baseline-pipe-example> ./a.out
processo 32254 - digitare testo da trasferire (quit per terminare):
ciao
processo 32254 - scritto messaggio: ciao
process 32255 - letto messaggio: ciao
```

Queste PIPE vengono utilizzate in maniera "massiva" nel momento in cui utilizziamo le shell.

Quando utilizziamo una PIPE su shell per specificare che l'output di un comando è un input di un altro comando, in realtà l'applicazione iniziale SCRIVE sulla PIPE delle informazioni e l'applicazione che riceve legge i dati dalla PIPE.
Quindi lo standard output del primo processo è ridirezionato sulla PIPE.
E lo standard input del secondo comando che riceve i dati deve andare a prelevare i dati da questa PIPE.
Chi scrive va a prendere il canale 1 associato allo std::out, lo va a chiudere, e scrive il canale che ci porta in input sulla PIPE.
Chi riceve chiude il canale da std::in (canale 0) e scrive il canale che ci porta in output sulla PIPE, ossia il canale di lettura (a dx) della pipe.