Pipe

Le *pipe* sono la forma più "antica" di comunicazione tra processi UNIX. Una pipe, che letteralmente significa *tubo*, costituisce un vero e proprio canale di comunicazione tra due processi: si possono inviare dati da un *lato* della pipe e riceverli dal *lato opposto*. Tecnicamente, la pipe è una porta di comunicazione (nominazione indiretta) con send asincrona e receive sincrona.

Esistono due forme di pipe in UNIX: **senza nome** e **con nome**. Le prime sono utilizzabili solo da processi con antenati comuni, in quanto sono risorse che vengono ereditate dai genitori. Le seconde, invece, hanno un nome nel filesystem e costituiscono quindi delle porte che tutti i processi possono utilizzare.

Pipe senza nome

Le pipe senza nome sono utilizzate per combinare comandi Unix direttamente dalla shell tramite il simbolo "|" (pipe). Prima di proseguire con questa esercitazione vi consiglio di provare i semplici esempi illustrati in aula la lezione scorsa.

Per creare una pipe si utilizza la systemcall `pipe(int filedes[2])` che restituisce in `filedes` due descrittori:

- `filedes[0]` per la lettura;
- `filedes[1]` per la scrittura;

Notiamo quindi che le pipe sono **half-duplex** (monodirezionali): esistono due distinti descrittori per leggere e scrivere. Per il resto, una pipe si utilizza come un normale file come mostra il seguente esempio:

```
#include<stdio.h>
#include <string.h>
main() {
        int fd[2];

        pipe(fd); /* crea la pipe */
        if (fork() == 0) {
                char *phrase = "prova a inviare questo!";

                close(fd[0]);                          /* chiude in lettura */
                write(fd[1],phrase,strlen(phrase)+1); /* invia anche 0x00 */
                close (fd[1]);                         /* chiude in scrittura */
        } else {
                char message[100];
                memset(message,0,100);
                int bytesread;

                close(fd[1]);                          /* chiude in scrittura */
                bytesread = read(fd[0],message,100);
                printf("ho letto dalla pipe %d bytes: '%s' \n",bytesread,message);
                close(fd[0]);                          /* chiude in lettura */
        }
}
```

Descrizione: Viene creata una pipe e, subito dopo, un nuovo processo. Le pipe vengono ereditate dai figli e quindi entrambi i processi, dopo la fork, condividono la pipe. Il processo figlio invia una stringa (incluso il terminatore 0×00) e il processo padre la legge. Notare che il processo che scrive chiude subito la pipe in lettura mentre quello che legge la chiude in scrittura. Questo è molto importante: ogni processo tiene aperte solo le risorse che intende utilizzare.

Se eseguiamo il programma otteniamo il seguente output:

```
$ ./prova_pipe
ho letto dalla pipe 24 bytes: 'prova a inviare questo!'
$
```

Vediamo che il processo padre riceve correttamente il messaggio. Come nella lettura da file, la read restituisce il numero di byte letti.

Invio e ricezione su una pipe "chiusa"

Ci sono alcune situazioni, tipiche delle pipe, che analizziamo in dettaglio:

- *Cosa accade se si fa una read da una pipe che è vuota ed è stata chiusa in scrittura (non ci sono più dati nel buffer e non ci sono più scrittori)?*
  La read ritorna 0, corrispondente a un end-of-file.

- *Cosa accade se si fa una write su una pipe che è stata chiusa in lettura (non ci sono più lettori)?*
  Viene generato il segnale `SIGPIPE` che di default termina il processo. Se si ignora o si gestisce il segnale la write ritorna un errore e `errno` è settata a `EPIPE`

NOTA: Qui si capisce l'importanza di **chiudere subito** una risorsa che non verrà utilizzata. Se il processo che legge solamente non chiude in scrittura la pipe, tale processo non riceverà l'end-of-file nel caso l'altro processo chiuda la pipe in scrittura. Infatti, esiste ancora un processo scrittore attivo e il sistema tiene aperta la pipe. Lo stesso accade con `SIGPIPE` se il processo che scrive solamente non chiude la pipe il lettura.

ESERCIZIO: modificare il programma precedente in modo da osservare i due eventi discussi qui sopra (assenza di scrittori, assenza di lettori)

Esempio: La pipe della shell

La seguente linea di comando prende l'output di `prog1` e lo manda in input a `prog2`.

```
$ prog1 | prog2
```

Per capire come la shell implementa questo comportamento proviamo a simularlo in C:

```c
#include <stdio.h>
#include <unistd.h>
main(int argc, char * argv[])
{
        int fd[2], bytesread;

        pipe(fd);
        if (fork() == 0)        {
                close(fd[0]);                                   /* chiude in lettura */
                dup2 (fd[1],1);                                 /* fa si che 1 (stdout) sia una copia di fd[1] */
                                                                /* da qui in poi l'output va sulla pipe */
                close(fd[1]);                                   /* chiude il descrittore fd[1] */
                execlp(argv[1],argv[1],NULL);                   /* esegue il comando */
                perror("errore esecuzione primo comando");
        } else {
                close(fd[1]);                                   /* chiude in scrittura */
                dup2 (fd[0],0);                                 /* fa si che 0 (stdin) sia una copia di fd[0] */
                                                                /* da qui in poi l'input proviene dalla pipe */
                close(fd[0]);                                   /* chiude il descrittore fd[0] */
                execlp(argv[2],argv[2],NULL);                   /* esegue il comando */
                perror("errore esecuzione secondo comando");
        }
}
```

Descrizione: Viene creata una pipe e viene eseguita una fork. I due processi ereditano la pipe e chiudono i descrittori che non useranno. Successivamente "copiano" i descrittori di scrittura e lettura della pipe, rispettivamente, sullo standard output e input (vedi la nota sotto). Poi chiudono definitivamente i descrittori della pipe e fanno una exec.

NOTA: `dup2(fd[1],1)` fa si che il descrittore dello stdout (1) da questo momento in poi punti a ciò che è puntato da `fd[1]` (diventano intercambiabili). Tutto ciò che verrà mandato sullo standard output andrà a finire sulla pipe. Se 1 puntasse ad un normale file il file verrebbe chiuso, se non riferito da altri descrittori. Notare che dopo la dup2 possiamo chiudere il descrittore della pipe, in quanto non verrà più utilizzato.

Facciamo un test per vedere che il programma si comporta come ci attendiamo (il comando 'wc' word-count, conta il numero di newline, parole e byte date in input).

```
$ who
focardi  pts/0        2012-11-07 14:13 (sally.dsi.unive.it)
focardi  pts/1        2012-11-13 23:47 (sally.dsi.unive.it)
$ who | wc
      2      10     120
$ ./simula_pipe who wc
      2      10     120
$
```

Il programma simula quindi l'esecuzione con la pipe della shell. In realtà la shell esegue due fork, una per ogni processo eseguito e attende la terminazione. Per semplificare abbiamo simulato la pipe con una sola fork.

Atomicità

Letture e scritture sulle pipe sono atomiche se inferiori alla dimensione `PIPE_BUF` (`limits.h`), usualmente 4096 bytes. Sopra tale dimensione l'atomicità non è garantita. È importante ricordarsi questo aspetto perché sopra tale dimensione, se più processi scrivono contemporaneamente, non è detto che i byte in scrittura non si 'mischino' tra loro.

Pipe con nome

Le pipe senza nome non possono essere utilizzate da processi che non hanno un antenato in comune in quanto l'unico modo per leggere e scrivere è tramite i descrittori di lettura e scrittura. Per ovviare a questa limitazione esistono le 'pipe con nome'. Tali pipe possono essere create con il comando `mkfifo`.

```
$ mkfifo myPipe     <==== (oppure mknod myPipe p)
$ ls -al
totale 36
drwxrwxr-x    2 focardi  focardi      4096 mag 23 00:57 .
drwxrwxr-x    7 focardi  focardi      4096 mag 23 00:16 ..
...
prw-rw-r--    1 focardi  focardi         0 mag 23 00:57 myPipe
...
$
```

Da questo momento in poi la pipe esiste nel filesystem e qualsiasi processo con i diritti di accesso al file può utilizzarla.

Esempio: Un lettore e tanti scrittori

Consideriamo un processo *lettore* (destinatario) che accetta, su una pipe con nome, messaggi provenienti da più *scrittori* (mittenti). Gli scrittori mandano 3 messaggi e poi terminano. Quando tutti gli scrittori chiudono la pipe il lettore ottiene 0 come valore di ritorno dalla `read` ed esce. Lettori e scrittori sono processi distinti lanciati indipendentemente (non necessariamente parenti).

Processo lettore (vedere i commenti nel codice):

```c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdlib.h>

#define PNAME "/tmp/aPipe"
main() {
        int fd;
        char str[100], leggi;

        mkfifo(PNAME,0666);     // crea la pipe con nome, se esiste gia' non fa nulla

        if ( (fd = open (PNAME,O_RDONLY)) < 0 ) { // apre la pipe in lettura
                perror("errore apertura pipe");
                exit(1);
        }

        while (read(fd,&leggi,1) ) {    // legge un carattere alla volta fino a EOF
                printf("%c",leggi);
                if (leggi == '\0') printf("\n"); // a capo dopo ogni stringa
        }

        close(fd);              // chiude il descrittore
        unlink(PNAME);          // rimuove la pipe
}
```

Processo scrittore (vedere i commenti nel codice):

```c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>

#define PNAME "/tmp/aPipe"

main() {
        int fd,i;
        char message[100], leggi;

        mkfifo(PNAME,0666);     // crea la pipe con nome, se esiste gia' non fa nulla

        // crea la stringa da scrivere sulla pipe
        sprintf(message,"Saluti dal processo %d",getpid());

        if ( (fd = open(PNAME,O_WRONLY)) < 0) { // apre la pipe in scrittura
                perror("errore apertura pipe");
                exit(1);
        }

        for (i=1; i<=3; i++){           // scrive tre volte la stringa
                write (fd,message,strlen(message)+1);
                sleep(2);
                }

        close(fd);      // chiude il descrittore
}
```

Possiamo compilare i due processi e lanciarli indipendentemente. Lanciamo, ad esempio, il lettore e tre scrittori. Notare l'uso di & per lanciare i processi in 'background' (e quindi in parallelo).

```
> ./lettore & ./scrittore & ./scrittore & ./scrittore
[1] 46998
[2] 46999
[3] 47000
Saluti dal processo 47000
Saluti dal processo 46999
Saluti dal processo 47001
Saluti dal processo 47000
Saluti dal processo 47001
Saluti dal processo 46999
Saluti dal processo 47000
Saluti dal processo 46999
Saluti dal processo 47001
[1]   Done                    ./lettore
[2]-  Exit 255                ./scrittore
[3]+  Exit 255                ./scrittore
>
```

Notare che le scritte a video vengono effettuate dal lettore dopo aver ricevuto il messaggio dalla pipe. I processi scrittori non scrivono nulla a video. Provare anche a lanciare il lettore su un terminale e i tre scrittori su un terminale differente.

ESERCIZIO: Come abbiamo già discusso la write, sotto la dimensione `PIPE_BUF`, è atomica: Più processi possono scrivere messaggi sulla stessa pipe se tali messaggi sono più corti di `PIPE_BUF`: i messaggi saranno accodati uno dopo l'altro senza che i singoli caratteri si 'mescolino' tra loro. Il lettore invece legge un carattere alla volta. Provare a lanciare più lettori per osservare interferenze.

Le pipe con nome sono bidirezionali?

L'opzione `O_RDWR` nella open permette di aprire una pipe con nome in lettura e scrittura. Come si fa però a evitare che un processo legga ciò che lui stesso ha scritto? Questo fatto rende le pipe `O_RDWR` inutilizzabili in pratica. Tale modalità esiste solo per poter aprire una pipe in scrittura quando nessuno l'ha ancora aperta in lettura. Una open in scrittura di una pipe non ancora aperta in lettura è bloccante.

Altri esercizi sulle pipe

1. Le pipe gestiscono **stream di byte**: non c'è nessuna nozione di messaggio. Non è detto, quindi, che due write vengano lette tramite due read. È possibile sperimentare questo aspetto inviando due write distinte consecutive e osservando che vengono tipicamente lette da un'unica read (fare attenzione, se si inviano stringhe, a togliere il NULL dalla prima stringa in modo da 'concatenarle'. Altrimenti, le due stringhe verranno ricevute ma la `printf` visualizzerà solo i caratteri prima del primo NULL, ovvero la prima stringa).
2. Anche se non è ovvio da visualizzare, si può provare a superare `PIPE_BUF` (4096) per vedere che le write interferiscono una con l'altra. Si suggerisce di ridirezionare l'output del lettore su file (tramite '> nomefile') in modo da esaminare l'output con un editor alla ricerca di messaggi 'mescolati'