Comunicazione fra processi: le pipe

I processi possono comunicare attraverso dei canali chiamati **pipe** (condotti). Una pipe corrisponde ad un file accessibile ad un estremo in lettura e all'estremo opposto in scrittura. I dati immessi nel canale con successive scritture risultano accodati nell'attesa che un altro processo le legga; la coda è gestita con criterio FIFO.

Le pipe forniscono anche una sincronizzazione tra processi. Quando un processo tenta di scrivere, può essere sospeso se la pipe è piena. Analogamente, un processo che tenti di leggere può essere sospeso se la pipe è vuota.

Per creare una pipe un processo invoca la funzione **pipe**:

```
pipe(fd)
```

in cui `fd` è un vettore di due elementi interi: all'elemento `fd[0]` viene assegnato il descrittore del file aperto in lettura corrispondente al lato di uscita della pipe, mentre a `fd[1]` viene assegnato il descrittore del file aperto in scrittura corrispondente al lato di ingresso.

La pipe è subordinata alla organizzazione gerarchica dei processi: l'unico modo che hanno due processi per comunicare è di avere un antenato comune che abbia predisposto una pipe a questo fine

Esempio:

```
int files[2];
...
pipe(files);
if(fork() == 0)                /* crea il produttore */
     exec("producer",...);
if(fork == 0)             /* crea il consumatore */
     exec("cosumer",...);
i = wait(&status);        /* attende la terminazione
i = wait(&status);           di entrambi */
```

Pipe (1)

Le regole seguenti si applicano ai processi che leggono da una pipe:

1. se un processo tenta di leggere da una pipe in cui è chiuso il lato di scrittura, allora `read()` ritorna il valore 0. Questo viene interpretato come "end-of-input"

2. se un processo tenta di leggere da una pipe vuota il cui lato di scrittura è ancora aperto, allora il processo viene posto in stato di "sleep" fino a quando un dato di ingresso diventa disponibile

3. se un processo tenta di leggere più byte da una pipe di quanti ne siano effettivamente presenti, l'intero contenuto della pipe viene letto e la funzione `read()` restituisce il numero di byte letti

Le regole seguenti si applicano ai processi che scrivono su una pipe:

1. se un processo cerca di scrivere su una pipe il cui lato di lettura è chiuso, la `write()` fallisce e il processo riceve un segnale SIGPIPE. L'azione di default associata a questo segnale è quella di interrompere il processo

2. se un processo cerca di scrivere su una pipe piena il cui lato di lettura è ancora aperto, il processo viene posto in stato di sleep finché non si rendano byte disponibili

3. se un processo vuole scrivere un numero di byte che è minore della disponibilità della pipe, la funzione di scrittura write() è **atomica**, cioè il processo scrittore porterà a termine l'operazione di scrittura senza che altri processi si inseriscano.
   Nel caso in cui il numero di byte da scrivere è maggiore, questa garanzia non è data

La primitiva `lseek` non può essere impiegata sulle pipe.

Pipe (2)

Le pipe sono usate in genere come meccanismo di comunicazione tra un processo padre ed uno figlio.

La sequenza di eventi associata a questo tipo di interazione è la seguente:

1. il processo padre crea la pipe usando la primitiva pipe()
2. il processo padre crea il figlio con una fork
3. lo scrittore chiude il lato di lettura della pipe, e il lettore chiude il lato di scrittura
4. i processi comunicano usando read() e write()
5. al termine della comunicazione i processi chiudono il lato della pipe di loro "competenza"

Esempio:
Scrivere un programma che usi le pipe per leggere messaggi dal proprio figlio.

```
$ cat talk.c
#include <stdio.h>
#define   READ      0
#define   WRITE     1

char *frase = "Questo e' il messaggio..."

main()  {

int fd[2], bytesRead;
char message[100];   /* il buffer del processo padre */

pipe(fd)             /* crea la pipe */
if( fork() == 0) {   /* figlio: scrittore */
     close(fd[READ]);     /* chiudi pipe non usata */
     write(fd[WRITE],frase,strlen(frase)+1); /*c'è NULL*/
     close(fd[WRITE]);
     }
else {               /* padre: lettore */
     close(fd[WRITE]);    /* chiudi pipe non usata */
     bytesRead = read(fd[READ], message, 100);
     printf("Letti %d byte: %s\n", bytesRead, message);
     close(fd[READ]);
     }

} /* end of main */

$ talk.exe                      ... esegui il programma
Letti 25 byte: Questo e' il messaggio...
$ _
```

Dimensioni di una pipe

E' importante sottolineare che le dimensioni di una pipe sono finite, e cioè soltanto un numero definito di byte può essere scritto o letto. Il limite è in genere 5120 byte.

Questo limite può avere implicazioni sull'uso di `read()` e `write()`. Se una `write` è eseguita su una pipe e c'è sufficiente spazio libero, allora i dati vengono scritti e la `write` ritorna immediatamente.

Se, invece, non c'è abbastanza spazio, allora il processo viene (normalmente) sospeso finché un altro processo non legge dalla pipe, liberando così spazio.

Esempio:

```
main()  {

int p[2], count;
char c = 'x';

if( pipe(p) < 0 ) {
     printf("Error: pipe failed...\n");
     exit(-1);
     }
signal( SIGALRM, alrm_action);

for( count=0; ; )  {
     alarm(20);                /* set alarm */
     write(p[1], &c, 1);       /* write down pipe */
     alarm(0);                 /* reset alarm */
     if(++count % 1024 == 0)
          printf( "%d characters in pipe\n", count);
     }
} /* end of main */


alrm_action(){
     printf("write blocked after %d characters\n", count );
     exit(0);
}
```

L'esecuzione darà:

```
1024 characters in pipe
2048 characters in pipe
:
5120 characters in pipe
write blocked after 5120 characters
```

Strutture FIFO (named pipe)

Le strutture FIFO (named pipe), introdotte in UNIX System V, sono meno restrittive delle pipe, ed offrono i seguenti vantaggi:

- hanno un nome simbolico che esiste nel file system
- possono essere usate da processi qualsiasi (non necessariamente aventi un antenato comune)
- esistono fino a quando non sono esplicitamente cancellate

Le named pipe possono essere creae in due modi:

- usando il comando `mknod`
- usando la primitiva `mknod()` (all'interno di un programma)

Esempio:

```
$ mknod myPipe p                ...crea la pipe
$ ls -l myPipe
prw-rw-r--  1   rossi     usr  0    Aug 1     21:25  myPipe
$ _
```

La letter a"p" nella prima colonna indica che si tratta di una pipe di tipo FIFO.

Per default, un processo che apre una FIFO in lettura verrà bloccato finché un altro processo non cercherà di aprire la stessa FIFO in scrittura.

Viceversa, un processo che apre una FIFO in scrittura aspetterà finché essa non verrà aperta in lettura.

Esempio:

```
$ cat < myPipe &
102
$ ls -l > myPipe
total 17
prw-rw-r--  1   rossi     usr  0    Aug 1     21:25  myPipe
-rw-rw-r--  1   rossi     usr  100  Aug 1     20:00  file1
-rw-rw-r--  1   rossi     usr  16   Aug 15    21:20  file2
-rw-rw-r--  1   rossi     usr  50   Sep 16    21:24  file3
$ _
```

Programmare con le FIFO

Per creare una FIFO da programma si usa la primitiva di sistema `mknod()`, ricordando di sommare 010000 (costante S_IFIFO in stat.h) al modo di apertura per indicare al sistema che si tratta di una pipe.

Per esempio:

```
if( mknod("fifo", 010600, 0 ) < 0 )
     printf("Error: non posso creare la pipe....\n");
```

Questo comando crea una named pipe di nome "fifo" con modalità di accesso 0600 (cioè solo il proprietario può leggere e scrivere sulla pipe).

Una volta creata, una pipe deve essere aperta usando la primitiva open (come i file ordinari), usando i flag `O_WRONLY, O_RDONLY, O_RDWR`.

Esempio: `fd = open("fifo", O_WRONLY)`

In questo caso, se nessuno ha aperto la pipe in lettura, il processo verrà posto in stato di attesa.

E' però possibile evitare questo comportamento, utilizzando aperture "non-bloccanti", che consentono al processo di riprendere il controllo anche se la pipe non è stata aperta (in modalità opposta) da nessuno. Per fare questo si usa il flag `O_NDELAY`.

Esempio: `fd = open("fifo", O_RDONLY | O_NDELAY)`

Se nessuno ha aperto la pipe in scrittura, la `open` restituirà il valore -1 , invece di bloccare il processo.

Esempio

Costruire, utilizzando le FIFO, due programmi che si scambiano messaggi.

Esempio: `$ sendmsg "message text 1" "message text 2"`

```
/*
*
*     sendmsg -- send messages via FIFO
*
*/

#include <stdio.h>

#define    MSGSIZ     63

char *fifo = "fifo"

main(int argc, char *argv[] )
{
int fd, j, nwrite;
char msgbuf[MSGSIZ+1];

if( argc < 2) {
      printf("Usage: sendmsg msg......\n");
      exit(-1);
      }

/*--- open fifo with O_NDELAY ---*/
if( (fd = open(fifo, O_WRONLY | O_NDELAY )) < 0 )
      printf("Error opening fifo....");
      exit(-1);
      }

/*--- send message ---*/
for( j=1; j < argc; j++ ) {
      if(strlen(argv[j]) > MSGSIZ)
            printf("Error: message too long: %s\n", argv[j]);
            continue;
      }

      strcpy(msgbuf, argv[j]);

      if((nwrite = write(fd,msgbuf,MSGSIZ+1)) <= 0 ) {
            printf("Error: message write failed...\n");
            exit(-1);
            }

} /* end of main */
```

```
/***
*
*     rcvmsg -- receive message via fifo
*
***/

#include <stdio.h>
#define   MSGSIZ     63


char *fifo = "fifo"

main(int argc, char *argv[] )
{
int fd;
char msgbuf[MSGSIZ+1];

/*--- open fifo for reading and writing ---*/
if( (fd = open(fifo, O_RDWR )) < 0 ) {
     printf("Error opening fifo....");
     exit(-1);
     }

/*--- receive messages ---*/
for(;;) {
     if(read(fd,msgbuf,MSGSIZ+1) < 0)
          printf("Error opening fifo....");
          exit(-1);
          }

     /* here something more useful can be done ! */

     printf("message received: %s\n", msgbuf);
     }

} /* end of main */
```

Un esempio di esecuzione:

```
$ mknod fifo p                  ... crea la pipe
$ rcvmsg &
40
$ sendmsg "message 1" " message 2"
message received: message 1
message received: message 1
$ sendmsg "message number 3"
message received: message number 3
```