IPC: pipe anonime

Laboratorio Sistemi Operativi

Antonino Staiano
Email: antonino.staiano@uniparthenope.it

La sincronizzazione in Unix

- La gestione della sincronizzazione in Unix riguarda due aspetti
  - scambio di dati
  - segnalazione di eventi
- Lo scambio di dati avviene attraverso
  - *pipe*: operazioni di I/O su code FIFO, sincronizzate dal S.O.
  - *messaggi*: invio e ricezione di messaggi tipizzati su coda FIFO
  - *memoria condivisa*: allocata e associata al processo attraverso system call
- La segnalazione di azioni avviene attraverso
  - *segnali (signal):* interrupt software
  - *semafori*: una generalizzazione dei semafori classici

IPC: Interprocess Communication

- Meccanismi che permettono a processi distinti di comunicare e scambiare informazioni
- I processi che comunicano possono risiedere
  - sulla stessa macchina (segnali, pipe, fifo, socket)
  - su macchine diverse (socket)
- La comunicazione può essere finalizzata a:
  - cooperazione: i processi scambiano dati per l'ottenimento di un fine comune
  - sincronizzazione: lo scambio di informazioni permette a processi indipendenti, ma correlati, di schedulare correttamente la propria attività (es. di non accedere contemporaneamente ad una risorsa condivisa)

Pipe anonime

- Il termine pipe è usato per indicare situazioni in cui si connette un flusso di dati da un processo ad un altro
- Gli utenti Linux hanno familiarità con il concetto di pipe:
  - Meccanismo utilizzato dalla shell per connettere l'output di un comando all'input di un altro (pipeline)

```
$ cmd1 | cmd2
```

    - I due processi connessi da una pipe sono eseguiti concorrentemente
    - Memorizza automaticamente l'output dello scrittore (cmd1) in un buffer
    - Se il buffer è pieno, lo scrittore si sospende fino a che alcuni dati non vengono letti
    - Se il buffer è vuoto, il lettore (cmd2) si sospende fino a che diventano disponibili dei dati in output

Pipe anonime e con nome (FIFO)

- Pipe Anonime
  - Presenti in tutte le versioni di UNIX
  - utilizzate, ad es., dalle shell per le pipeline
- Pipe con nome o FIFO
  - Presenti in UNIX System V (anche in BSD per compatibilità)

Pipe anonime

- Una pipe anonima è un canale di comunicazione (creato con pipe()), mantenuto a livello kernel, che unisce due processi
  - unidirezionale
  - permette la comunicazione solo tra processi con un antenato comune
- Una pipe presenta due lati di accesso (in/out), ciascuno associato ad un descrittore di file
  - Il lato di lettura è acceduto invocando read()
  - Il lato di scrittura è acceduto invocando write()
  - Memorizza il suo input in un buffer (massima dimensione (PIPE_BUF) , ma è tipicamente intorno ai 4K)
- Quando un processo ha finito di usare un lato di una pipe chiude il descrittore con close()

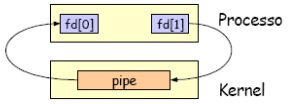
Pipe anonime

- Il flusso di dati avviene in una sola direzione (half-duplex)
- 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
- Ogni volta che si digita una sequenza di comandi in una pipeline da eseguire nella shell, la shell stessa crea un processo separato per ciascun comando e collega lo standard output di uno allo standard input dell'altro usando una pipe

Pipe anonime: pipe()

```
#include <unistd.h>
int pipe (int fd[2])
```

- crea una pipe anonima e restituisce due descrittori di file
  - lato di lettura fd[0] (aperto in lettura)
  - lato di scrittura fd[1] (aperto in scrittura)



- Fallisce restituendo -1 se il kernel non ha più spazio per una nuova pipe; altrimenti restituisce 0

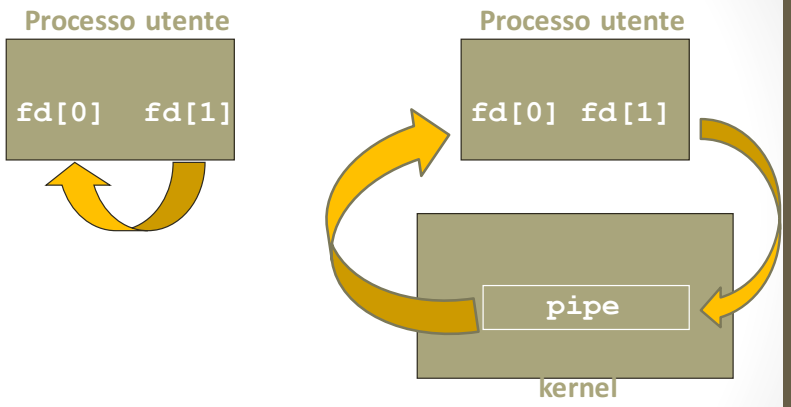
Pipe anonime: lettura

- Se un processo legge da una pipe:
  - se il lato scrittura è stato chiuso, read() restituisce 0 che indica la fine dell'input (*dopo che sono stati letti tutti i dati*)
  - se la pipe è vuota e il lato di scrittura è ancora aperto, si sospende fino a che diventa disponibile qualche input
  - se il processo tenta di leggere più byte di quelli presenti nel buffer associato, i byte disponibili vengono letti e read() restituisce il numero dei byte effettivamente letti

Pipe anonime: scrittura

- Se un processo scrive su di una pipe:
  - se il lato di lettura è stato chiuso, write() fallisce (ritorna -1) ed allo scrittore è inviato un segnale SIGPIPE, la cui azione di default è di far terminare il ricevente (errno impostato a EPIPE)
  - se scrive meno byte di quelli che una pipe può contenere, write() viene eseguita in modo atomico (non possono avvenire intrecci dei dati scritti da processi diversi sulla stessa pipe)
  - se scrive più byte di quelli che una pipe può contenere (PIPE_BUF), non c'è garanzia di atomicità
- lseek() non ha senso se applicata ad un pipe
- Dato che l'accesso ad una pipe anonima avviene tramite descrittore di file, solo il processo creatore ed i suoi discendenti possono accedere alla pipe

Pipe anonime: un singolo processo



- Sinistra: i due estremi di una pipe connessi in un processo singolo
- Destra: flusso dei dati nella pipe attraverso il kernel

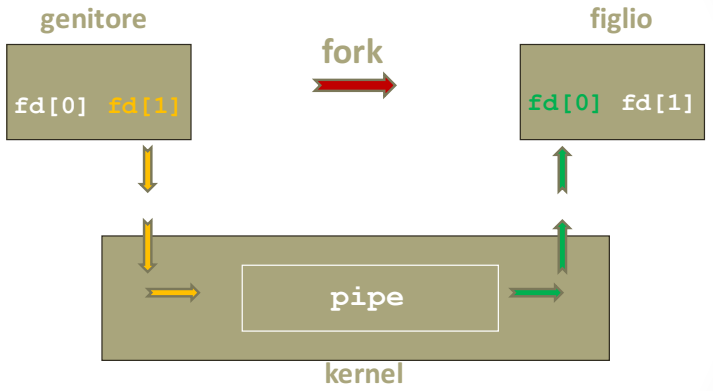
Pipe anonime

- La chiamata di sistema fstat() restituisce il tipo di file FIFO per il descrittore di file delle estremità di una pipe
- Possiamo verificare il tipo pipe con la macro S_ISFIFO
- Utilizzare una pipe in un processo singolo non ha senso
  - Normalmente, il processo che crea una pipe successivamente invoca una fork(), creando un canale di IPC dal genitore al figlio o viceversa

Pipe anonime: pipe()

- La tipica sequenza di eventi è
  - il processo crea una pipe anonima (pipe())
  - il processo crea un figlio (fork())
  - lo scrittore chiude il suo lato di lettura della pipe ed il lettore chiude il suo lato scrittura (close())
  - i processi comunicano usando write() e read()
  - ogni processo chiude (close()) il suo descrittore quando ha finito
- Una comunicazione bidirezionale si può realizzare utilizzando due pipe

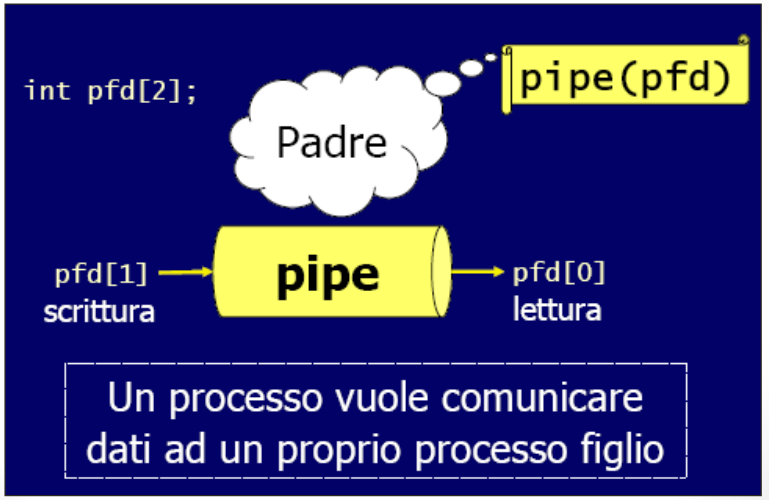
Pipe half-duplex dopo una fork



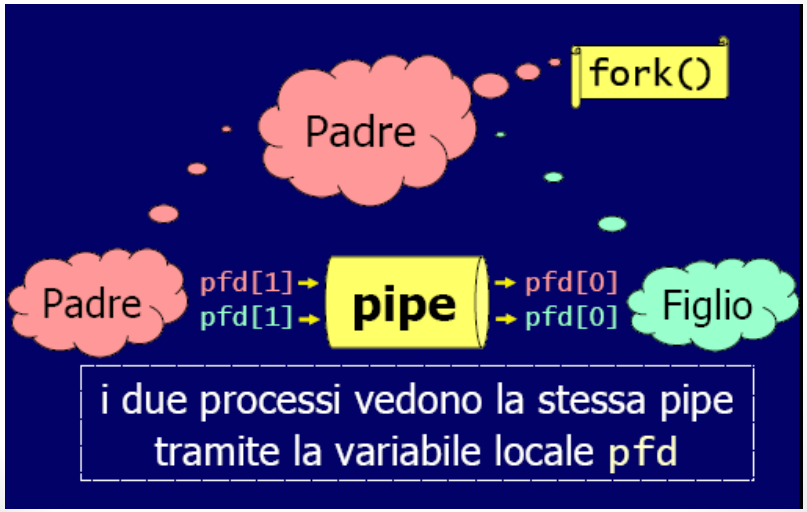
Pipe anonime e fork

- Cosa accade dopo la fork() dipende dalla direzione del flusso di dati che vogliamo
  - Per una pipe dal genitore al figlio, il genitore chiude l'estremità in lettura della pipe (fd[0]), ed il figlio chiude l'estremità in scrittura (fd[1])
  - Per una pipe dal figlio al genitore, il genitore chiude fd[1] ed il figlio chiude fd[0]

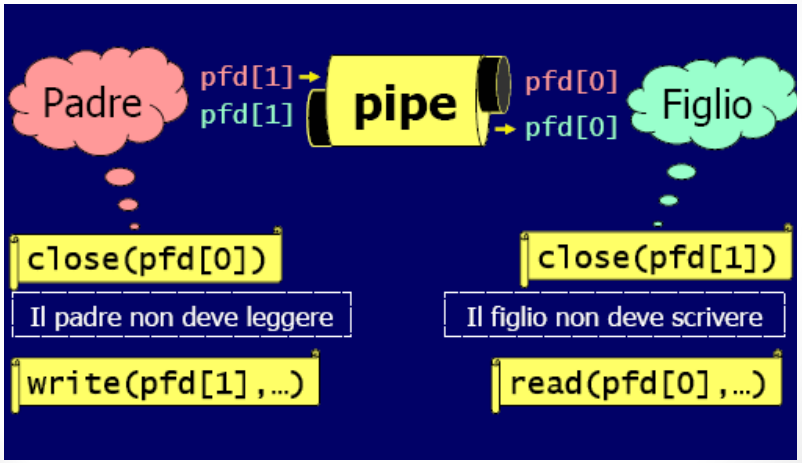
Pipe anonima: comunicazione padre-figlio



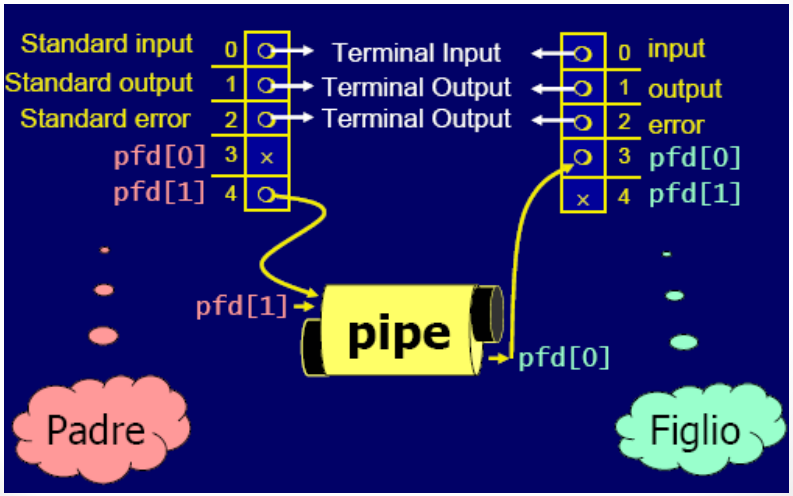
Pipe anonima: comunicazione padre-figlio (2)



Pipe anonima: padre scrive – figlio legge (3)



Pipe e tabella descrittori



Esempio 1: padre scrive - figlio legge

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define SIZE 1024
int main(int argc, char*argv[]) {
  int pfd[2];
  int nread;
  int pid;
  char buf[SIZE];
  if (pipe(pfd)== -1) {
      perror("pipe() fallita)");
      exit(-1);
  }
  if((pid=fork()) < 0 ) {
      perror("fork() fallita");
      exit(-2);
  }
…
```

Esempio 1 (cont.)

```
...
if(pid==0) { /* figlio */
   close(pfd[1]);
   while ( (nread=read(pfd[0], buf, SIZE)) != 0)
  {
      printf("il figlio legge: %s\n", buf);
  }
  close(pfd[0]);
} else { /* padre */
   close(pfd[0]);
   strcpy(buf,"Sono tuo padre!");
   write(pfd[1], buf, strlen(buf)+1);
   close(pfd[1]);
}
exit(0);
}
```


Esempio 2: Leggere messaggi dal figlio

```
/* talk.c */
#include <stdio.h>
#include <string.h>
#define READ 0
#define WRITE 1
char *frase = "Messaggio...";
int main() {
int fd[2], bytesRead;
char message[100];

pipe(fd);
if(fork() == 0) {
close(fd[READ]);
write(fd[WRITE],frase,strlen(frase)+1);
close(fd[WRITE]);
}
```

```
else {
close(fd[WRITE]);
bytesRead=read(fd[READ],message,
100);
printf("Letti %d byte: %s\n",
    bytesRead, message);
close(fd[READ]);
}
}


$ a.out
Letti 12 byte: Messaggio...
$
```


Comunicazione tramite pipe

- Quando un processo “scrittore” invia più messaggi di lunghezza variabile tramite una pipe, occorre fissare un protocollo di comunicazione che permetta al processo “lettore” di individuare la fine di ogni singolo messaggio
- Alcune possibilità sono:
  - inviare la lunghezza del messaggio (dato di dimensione fissa e nota) prima del messaggio stesso
  - terminare un messaggio con un carattere speciale come '\0' o un newline
- Più in generale, il protocollo stabilisce la sequenza di messaggi attesa delle due parti

Protocol.c (Semplice protocollo)

```
/* Semplice protocollo di comunicazione tramite pipe anonima. Dovendo
   inviare messaggi di lunghezza variabile ogni messaggio è preceduto
   da un intero che fornisce la sua lunghezza in caratteri */

#define READ 0 /* Estremità in lettura della pipe */
#define WRITE 1 /* Estremità in scrittura della pipe */
char *msg[3] = { "Primo", "Secondo", "Terzo" };
int main (void) {
  int fd [2], i, length, bytesRead;
  char buffer [100]; /* Buffer del messaggio */
  pipe (fd); /* Crea una pipe anonima */
  if (fork () == 0) { /* Figlio, scrittore */
        close(fd[READ]); /* Chiude l'estremità inutilizzata */
        for (i = 0; i < 3; i++) {
                length=strlen (msg[i])+1 ; /* include \0 */
                write (fd[WRITE], &length, sizeof(int));
                write (fd[WRITE], msg[i], length); }
        close (fd[WRITE]); /* Chiude l'estremità usata */
  }
```

Protocol.c (cont.)

```
else { /* Genitore, lettore */
  close (fd[WRITE]); /* Chiude l'estremità non usata */
  while (read(fd[READ], &length, sizeof(int))) {
       bytesRead = read (fd[READ], buffer, length);
       if (bytesRead != length) {
              printf("Errore!\n");
              exit(1);
       }
       printf("Padre:Letti %d byte:%s\n",bytesRead,buffer);
  }
  close (fd[READ]); /* Chiude l'estremità usata */
}
exit(0);
}
```


Pipe ed exec()

- I due processi comunicanti devono conoscere i descrittori della pipeline
- Finora si è sfruttato lo sdoppiamento delle variabili locali ed il fatto che un processo figlio eredita la tabella dei descrittori del padre
- Una exec() non altera la tavola dei descrittori anche se si perde lo spazio di memoria ereditato dal padre prima della exec()

Pipe ed exec() (cont.)

- Supponiamo di voler implementare:

  `$ ls | wc -w`

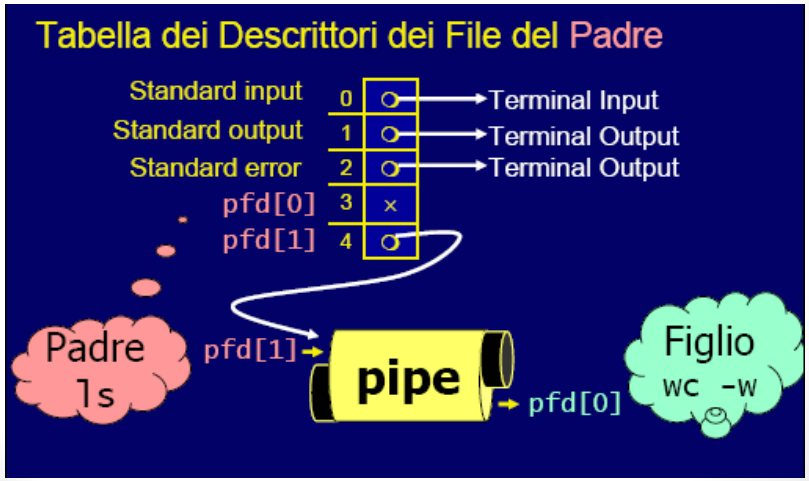
  - il padre esegue fork()
  - il padre esegue exec() per 'ls'
  - il figlio esegue exec() per 'wc -w'
- La exec() non cambia la tavola dei descrittori ma si perdono le variabili locali: come fare per accedere alla stessa pipe se gli unici riferimenti comuni erano in variabili locali?

Esempio 3

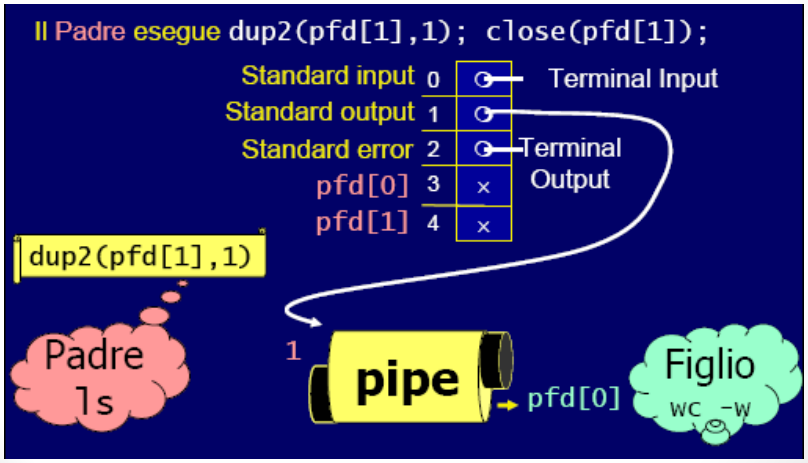
```
#include <stdio.h>
#define SIZE 1024
int main(int argc,char** argv){
int pfd[2], pid;
if (pipe(pfd) == -1) {
  perror("pipe() failed)");
  exit(-1);
}
if((pid=fork()) < 0 ) {
  perror("fork() failed");
  exit(-2);
}
…
```

```
…
if(pid==0) { /* figlio */
  close(pfd[1]);
  dup2(pfd[0],0);
  close(pfd[0]);
  execlp("wc","wc","-w",NULL);
  perror("wc fallita");
  exit(-3);
} else { /* padre */
  close(pfd[0]);
  dup2(pfd[1],1);
  close(pfd[1]);
  execlp("ls","ls",NULL);
  perror("ls fallita");
  exit(-4);
}
exit(0);
}
```

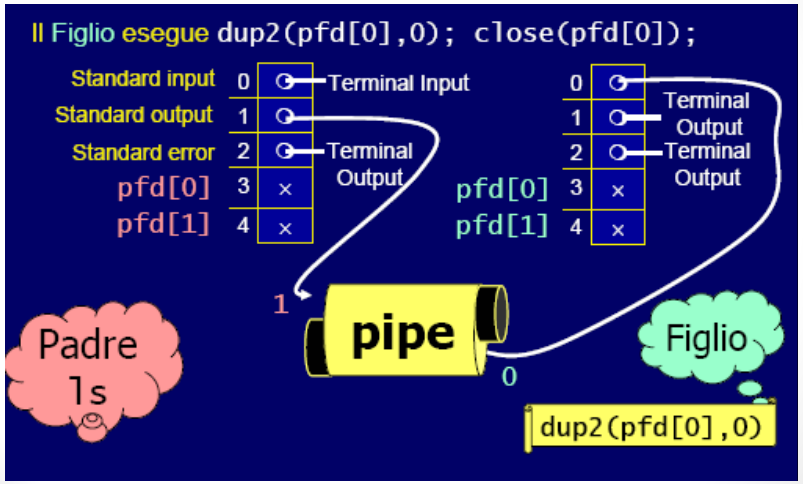
dup2 e pipeling



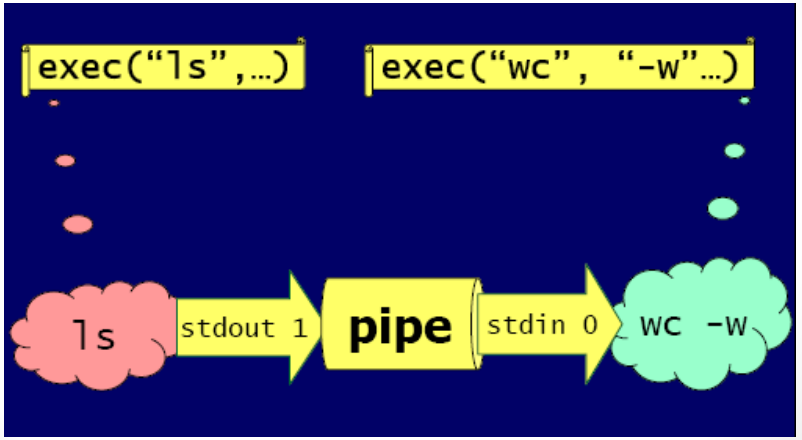
dup2 e pipeling (cont.)



dup2 e pipeling (cont.)



dup2 e pipeling (cont.)



Esempio 4

- Il programma seguente usa dup2() per inviare l'output da una pipe al comando sort
- Dopo aver creato la pipe, il programma invoca fork(). Il processo genitore stampa alcune stringhe nella pipe
- Il processo figlio connette il descrittore di file in lettura della pipe al suo standard input usando dup2(). Poi esegue il programma sort

Esempio 4 (cont.)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main ()
{
  int fds[2];
  pid_t pid;
/* Crea una pipe. I descrittori per le estremità sono in fds.*/
  pipe (fds);
/* Crea un processo figlio. */
  pid = fork ();
  if (pid == (pid_t) 0) {
/* Processo figlio. Chiude descrittore lato scrittura */
       close (fds[1]);
/* Connette il lato lettura della pipe allo standard input. */
       dup2 (fds[0], STDIN_FILENO);
/* Sostituisce il processo figlio con il programma  "sort". */
       execlp ("sort", "sort", 0);
  }
```


Esempio 4 (cont.)

```
else {/* Processo genitore */
/* Chiude il descrittore lato lettura */
       close (fds[0]);
       write(fds[1], "This is a test.\n", 16);
       write(fds[1], "Hello, world.\n", 14);
       write(fds[1], "My dog has fleas.\n",18 );
       write(fds[1], "This program is great.\n",23);
       write(fds[1], "One fish, two fish.\n",20);

       close (fds[1]);
/* Aspetta che il processo figlio finisca */
       waitpid (pid, NULL, 0);
  }
return 0;
}
```


Esempio 5

```
#include <stdio.h>
#include <unistd.h>
/* Esempio di creazione pipe per lanciare ls | sort */
int main()
{ int pid;
  int fd[2];
  pipe(fd);
  if ((pid = fork()) == 0)
{ /* figlio */
/* chiusura lettura da pipe */
close(fd[0]);
/* redirezione stdout a pipe */
dup2(fd[1], 1);
execlp("ls", "ls", NULL);
} else if (pid > 0)
{ /* padre */
/* chiusura scrittura su pipe */
close(fd[1]); /* redirezione stdin a pipe */
dup2(fd[0],0);
execlp("sort", "sort", NULL);}
}
```

Esempio: redirezione con pipe(connect.c)

- Il programma **connect.c** esegue due programmi e connette l'output di uno all'input dell'altro
- I nomi dei programmi sono passati come argomenti e si assume che nessuno dei due programmi sia invocato con argomenti

```
connect cmd1 cmd2
```


Esempio: redirezione con pipe (connect.c)

```
#include <stdio.h>
#define READ 0
#define WRITE 1
int main (int argc, char *argv []) {
  int fd [2];
  pipe (fd); /* Crea una pipe senza nome */
  if (fork () != 0) { /* Padre, scrittore */
    close (fd[READ]); /* Chiude l'estremità non usata */
    dup2 (fd[WRITE], 1); /* Duplica l'estremità usata allo stdout */
    close (fd[WRITE]); /* Chiude l'estremità originale usata */
    execlp (argv[1], argv[1], NULL); /* Esegue il programma scrittore */
    perror ("connect"); /* Non dovrebbe essere eseguita */
  }
  else { /* Figlio, lettore */
    close (fd[WRITE]); /* Chiude l'estremità non usata */
    dup2 (fd[READ], 0); /* Duplica l'estremità usata allo stdin */
    close (fd[READ]); /* Chiude l'estremità originale usata */
    execlp (argv[2], argv[2], NULL); /* Esegue il programma lettore */
    perror ("connect"); /* Non dovrebbe essere eseguita */
  }
}
```


Implementazione delle pipeline

- In generale, il processo padre
  - Crea tante pipe quanti sono i processi figli meno uno
  - Crea tanti processi figli quanti sono i comandi da eseguire
  - chiude tutte le estremità delle pipe
  - Attende la terminazione di ciascun processo figlio
- Ciascun processo figlio
  - Chiude le estremità delle pipe che non usa
  - Imposta le estremità delle pipe che usa sugli opportuni canali standard
  - Invoca una funzione della famiglia exec() per eseguire il proprio comando

Esercizi

- Esercizio: scrivere un programma che esegua il comando di shell "ls- R | grep <pat>" dove <pat> è un pattern inserito dall'utente
- Esercizio: scrivere un programma che esegua il comando di shell "ls| sort| grep <pat>" con tre processi distinti