La Comunicazione tra Processi in Unix

`pipe`

La pipe è un canale di comunicazione tra processi:

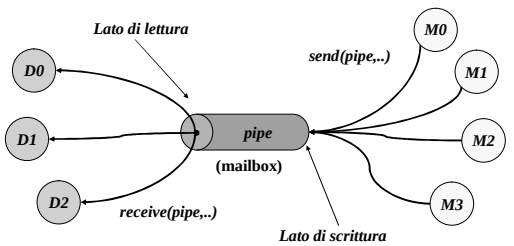
- ***unidirezionale***: accessibile ad un estremo in lettura ed all'altro in scrittura
- **molti-a-molti:**
  - più processi possono spedire messaggi attraverso la stessa pipe
  - più processi possono ricevere messaggi attraverso la stessa pipe
- **capacità limitata:**
  - la **pipe** è in grado di gestire l'**accodamento** di un numero limitato di messaggi, gestiti in modo FIFO: il limite è stabilito dalla **dimensione** della pipe (es.4096 bytes).

Interazione tra processi Unix

- I processi Unix non possono condividere memoria (**modello ad ambiente locale**)
- L'**interazione tra processi** può avvenire:
  - **mediante la condivisione di file:**
    - complessità : realizzazione della sincronizzazione tra i processi.
  - **attraverso specifici strumenti di *Inter Process Communication*:**
    - per la comunicazione tra processi **sulla stessa macchina**:
      - ✓`pipe` **(tra processi della stessa gerarchia)**
      - ✓`fifo` **(qualunque insieme di processi)**
    - per la comunicazione tra processi in nodi diversi della stessa **rete**:
      - ✓`socket`

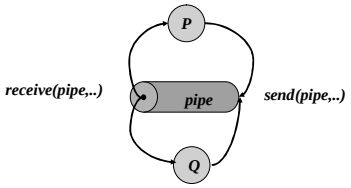
Comunicazione attraverso `pipe`

- Mediante la pipe, la comunicazione tra processi è **indiretta** (senza naming esplicito): ***mailbox***



Pipe: unidirezionalità/bidirezionalità

- **Uno stesso processo può:**
  - **sia depositare messaggi nella pipe (*send*), mediante il lato di scrittura**
  - **che prelevare messaggi dalla pipe (*receive*), mediante il lato di lettura**



- **la pipe può anche consentire una comunicazione *"bidirezionale"* tra i processi P e Q (ma va rigidamente disciplinata !)**

System call `pipe`

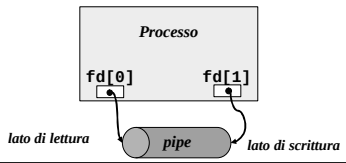
- **Per creare una pipe:**

```
int pipe(int fd[2]);
```

  - **`fd`** è il puntatore a un vettore di 2 **file descriptor**, che verranno inizializzati dalla system call in caso di successo:
    - **`fd[0]` rappresenta il lato di lettura della pipe**
    - **`fd[1]` è il lato di scrittura della pipe**
- **la system call pipe restituisce:**
  - **un valore negativo, in caso di fallimento**
  - **0, se ha successo**

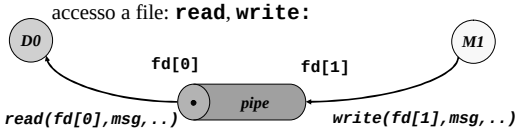
Creazione di una pipe

- **Se `pipe(fd)` ha successo:**
  - **vengono allocati due nuovi elementi nella tabella dei file aperti del processo e i rispettivi file descriptor vengono assegnati a `fd[0]` e `fd[1]`:**
    - **`fd[0]`: lato di lettura (*receive*) della pipe**
    - **`fd[1]`: lato di scrittura (*send*) della pipe**



Omogeneità con i file

- Ogni lato di accesso alla pipe è visto dal processo in modo **omogeneo** al file (file descriptor):
  - si può accedere alla pipe mediante le system call di accesso a file: **`read`**, **`write:`**



  - **`read:`** realizza la ***receive***
  - **`write:`** realizza la ***send***

Sincronizzazione dei processi comunicanti

- Il canale (la ***pipe***) ha capacità limitata: come nel caso di produttore/consumatore è necessario sincronizzare i processi:
  - **se la** *pipe* **è vuota: un processo che legge** si blocca
  - **se la** *pipe* **è piena: un processo che scrive** si blocca

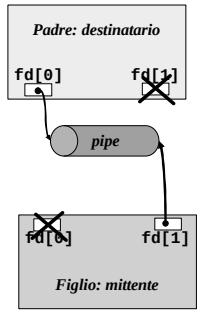
☞ **Sincronizzazione automatica:** **`read`** **e** **`write`** **da/verso pipe possono essere sospensive !**

Quali processi possono comunicare mediante pipe?

- **Per mittente e destinatario il riferimento al canale di comunicazione è un file descriptor:**
  - ☞ Soltanto i processi appartenenti a una stessa **gerarchia** (cioè, che hanno un ***antenato*** in comune) possono scambiarsi messaggi mediante pipe; ad esempio, possibilità di comunicazione:
    - **tra processi fratelli (**che **ereditano** la pipe dal processo padre**)**
    - **tra un processo *padre* e un processo *figlio*;**
    - **tra *nonno* e *nipote***
    - **etc.**

Esempio: comunicazione tra padre e figlio

```
main()
{int pid;
 char msg[]="ciao babbo";
 int fd[2];
 pipe(fd);
 pid=fork();
 if (pid==0)
{/* figlio */
  close(fd[0]);
  write(fd[1], msg, 10);
  ...}
else /* padre  */
{ close(fd[1]);
  read(fd[0], msg, 10);
  ...
}}
```



Ogni processo chiude il lato della pipe che non usa.

Chiusura di pipe

- Ogni processo può chiudere un estremo della pipe con una **`close`**.
- Un estremo della pipe viene ***effettivamente*** **chiuso** (cioè, la comunicazione non e` piu` possibile) quando tutti i processi che ne avevano visibilità hanno compiuto una **`close`**
- **Se un processo P:**
  - **tenta una lettura da una pipe il cui *lato di scrittura* è effettivamente chiuso: `read` ritorna 0**
  - **tenta una scrittura da una pipe il cui *lato di lettura* è effettivamente chiuso: `write` ritorna -1, ed il segnale SIGPIPE viene inviato a P (*broken* pipe).**

Esempio

```
/* Sintassi: progr N
padre(destinatario) e figlio(mittente) si scambiano una
   sequenza di messaggi di dimensione (DIM) costante; la
   lunghezza della sequenza non  e` nota a priori; il
   destinatario decide di interrompere la sequenza di
   scambi di messaggi dopo N secondi */

#include <stdio.h>
#include <signal.h>
#define DIM 10

int fd[2];
void fine(int signo);
void timeout(int signo);
```


Esempio

```
main(int argc, char **argv)
{int pid, N;
 char messaggio[DIM]="ciao ciao ";
  if (argc!=2)
  {   printf("Errore di sintassi\n");
      exit(1);}
  N=atoi(argv[1]);
  pipe(fd);
  pid=fork();
  if (pid==0) /* figlio */
  {   signal(SIGPIPE, fine);
      close(fd[0]);
      for(;;)
            write(fd[1], messaggio, DIM);
  }
```


Esempio

```
else if (pid>0) /* padre */
 {              signal(SIGALRM, timeout);
                close(fd[1]);
                alarm(N);
                for(;;)
                {       read(fd[0], messaggio, DIM);
                        write(1, messaggio, DIM);
                }
  }
}/* fine main */
```


Esempio

```
/* definizione degli handler dei segnali */
void timeout(int signo)
{ int stato;
  close(fd[0]);
   wait(&stato);
   if ((char)stato!=0)
      printf("Term. inv. figlio (segnale %d)\n",
                  (char)stato);
   else printf("Term. Vol. Figlio (stato %d)\n",
                  stato>>8);
    exit(0);
}

void fine(int signo)
{ close(fd[1]);
  exit(0);
}
```

System call dup

- Per duplicare un elemento della tabella dei file aperti di processo:

  `int dup(int fd)`

  - **fd è il file descriptor del file da duplicare**

  L'effetto di una **dup** è copiare l'elemento `fd` della tabella dei file aperti nella prima posizione libera (quella con l'indice minimo tra quelle disponibili).
- Restituisce il nuovo file descriptor (del file aperto copiato), oppure -1 (in caso di errore).

Esempio: mediante la dup è possibile ridirigere stdin e stdout su pipe

```
main()
{ int pid, fd[2]; char msg[3]="bye";
  pipe(fd);
  pid=fork();
  if (!pid) /* processo figlio */
  {   close(fd[0];
      close(1);
      dup(fd[1]); /* ridirigo stdout sulla pipe */
      close(fd[1]);
      write(1,msg, sizeof(msg)); /*scrivo su pipe*/
      close(1);
}else /*processo padre
{ close(fd[1]);
  read(fd[0], msg, 3);
  close(fd[0]);}}
```

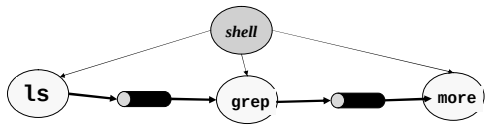

Dup & piping

- Mediate la dup si puo` realizzare il piping di comandi; ad esempio:

  `$ ls –lR |grep Jun |more`

➔ Vengono creati 3 processi (uno per ogni comando), in modo che:

- Lo std. output di **`ls`** sia ridiretto nello std. input di **`grep`**
- Lo std. output di **`grep`** sia ridiretto nello std. input di **`more`**



Esempio: piping di 2 comandi senza argomenti

```
/* sintassi: programma com1 com2  significa:
   com1|com2   */

main(int argc, char **argv)
{ int pid1, pid2, fd[2],i, status;
  pipe(fd);
  pid1=fork();
  if (!pid1) /* primo processo figlio: com2 */
  {   close(fd[1]);
      close(0);
      dup(fd[0]); /* ridirigo stdin sulla pipe */
      close(fd[0]);
      execlp(argv[2], argv[2],(char *)0);
      exit(-1);
}
```

```
else /*processo padre
{  pid2=fork();
   if (!pid2) /* secondo figlio: com1 */
   {   close(fd[0]);
       close(1);
       dup(fd[1]);
       close(fd[1]);
       execlp(argv[1], argv[1], (char *)0);
       exit(-1);
   }
  for (i=0; i<2;i++)
   {   wait(&status);
       if((char)status!=0)
             printf("figlio terminato per segnale%d\n",
             (char)status);
   }
exit(0);
}
```


Pipe

- La pipe ha **due svantaggi:**
  - consente la comunicazione solo tra processi in relazione di parentela
  - non è persistente: viene distrutta quando terminano tutti i processi che la usano.

Per realizzare la comunicazione tra una coppia di processi non appartenenti alla stessa gerarchia?

☞*FIFO*

fifo

- **È una pipe con nome nel file system:**
  - **canale unidirezionale del tipo *first-in-first-out***
  - **è rappresentata da un file nel file system: persistenza, visibilità potenzialmente globale**
  - **ha un proprietario, un insieme di diritti ed una lunghezza**
  - **è creata dalla system call `mkfifo`**
  - **è aperta e acceduta con le stesse system call dei file**

Creazione di una fifo: `mkfifo`

- Per creare una fifo:

```
int mkfifo(char* pathname, int mode);
```

  - **`pathname`** è il nome della fifo
  - **`mode`** esprime i permessi

Restituisce:

  - 0, in caso di successo
  - un valore negativo, in caso contrario

Apertura/Chiusura di fifo

- Una volta creata, una fifo può essere aperta (come tutti i file), mediante una **`open;`** ad esempio, un processo destinatario di messaggi**:**

```
int fd;
fd=open(“myfifo”, O_RDONLY);
```

- Per chiudere una fifo, si usa la close**:**

```
close(fd);
```

- Per eliminare una fifo, si usa la unlink**:**

```
unlink(“myfifo”);
```


Accesso a fifo

- Una volta aperta la fifo può essere acceduta (come tutti i file), mediante **`read/write;`** ad esempio, un processo destinatario di messaggi**:**

```
int fd;
char msg[10];
fd=open(“myfifo”, O_RDONLY);
read(fd, msg,10);
```