Chiamate di sistema per la Inter Process Communication (IPC) in POSIX

E.Mumolo, DEEI

mumolo@units.it

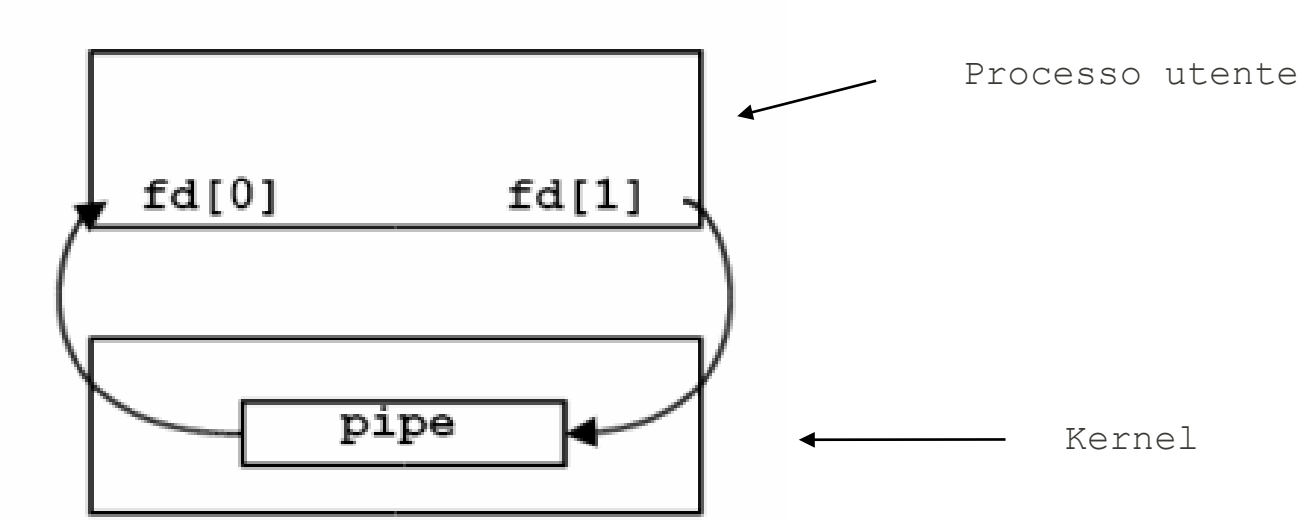
Pipe

- **Cos'è un pipe?**
  - E' un canale di comunicazione che unisce due processi
- **Caratteristiche:**
  - La più vecchia e la più usata forma di interprocess communication (IPC) utilizzata in Unix
- Limitazioni
  - Sono half-duplex (comunicazione in un solo senso)
  - **Utilizzabili solo tra processi con un "antenato" in comune**
- Come superare queste limitazioni?
  - Gli *stream pipe* sono full-duplex
  - *FIFO* (*named pipe*) possono essere utilizzati tra più processi
  - *named stream pipe* = stream pipe + FIFO

Pipe

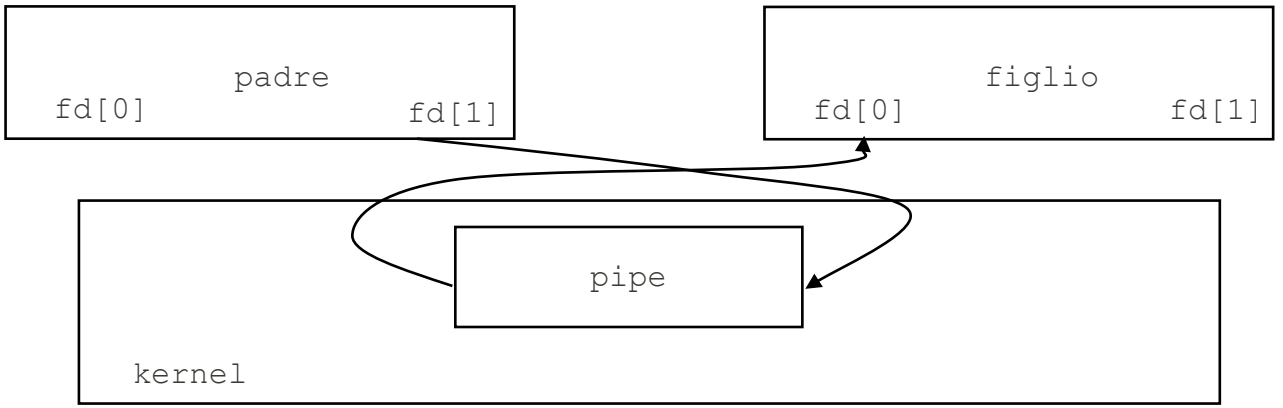
```
#include <unistd.h>
int pipe(int fildes[2]);
```

- Ritorna due descrittori di file attraverso l'argomento **filedes**
  - **filedes[0]** è aperto in lettura
  - **filedes[1]** è aperto in scrittura
  - L'output di **filedes[1]** è l'input di **filedes[0]**



Pipe

- **Come utilizzare i pipe?**
  - Cosa succede dopo la **fork** dipende dalla direzione dei dati?
  - I canali non utilizzati vanno chiusi
- **Esempio: padre | figlio**
  - Il padre chiude l'estremo di input (**close(fd[0]);**)
  - Il figlio chiude l'estremo di output (**close(fd[1]);**)



Pipe

- **Come utilizzare le pipe?**
  - Una volta creati, sono utilizzati con le normali chiamate **read**/**write** sugli estremi
- **La chiamata read**
  - se l'estremo di ouput è aperto
    - restituisce i dati disponibili, ritornando il numero di byte
    - successive chiamate si bloccano fino a quando nuovi dati non saranno disponibili
  - se l'estremo di output è stato chiuso
    - restituisce i dati disponibili, ritornando il numero di byte
    - successive chiamate ritornano 0
- **La chiamata write**
  - se l'estremo di input è aperto
    - i dati scritti rimangono fino a quando non saranno letti dall'altro processo
  - se l'estremo di input è stato chiuso
    - viene generato un segnale **SIGPIPE**
    - ignorato/catturato: write ritorna **–1** e **errno=EPIPE**
    - azione di default: terminazione

Esempio: padre|figlio

- **Passi fondamentali:**

  chiamo pipe(pfd)

  chiamo fork → padre scrive su pdf[1] –

  → figlio legge su pdf[0]

- **Codice:**

```
#include  <unistd.h>
#define MAXLINE 1024
int main(void)
{
  int   n, fd[2];
  pid_t pid;
  char  line[MAXLINE];

  if (pipe(fd) < 0)  err_sys("pipe error");

  if ( (pid = fork()) < 0)  err_sys("fork error");

  else if (pid > 0) {     /* padre */
    close(fd[0]);
    write(fd[1], "hello world\n", 12);
  } else {                          /* figlio */
    close(fd[1]);
    n = read(fd[0], line, MAXLINE);
    write(STDOUT_FILENO, line, n);
  }

  exit(0);
}
```

Esempio: who|wc

- **Passi fondamentali:**

  `pipe(pfd)`

  `fork → chiude file 1 → dup(pfd[1]) → execlp("who")`

  `fork → chiude file 0 → dup(pfd[0]) → execlp("wc")`

- **Codice:**

```
#include <stdio.h>
main()
{       int pfd[2];
        if(pipe(pfd) == -1)  syserr("pipe");
        switch(fork()) {
           case -1: syserr("fork");
           case 0:
                if (close(1) == -1)  syserr("close");
                if (dup(pfd[1]) != 1) fatal("dup");
                if (close(pfd[0]) == -1 || close(pfd[1]) == -1) syserr("close2");
                execlp("who", "who", NULL);
                syserr("execl");
        }
        switch(fork()) {
           case -1: syserr("fork");
           case 0:
              if(close(0) == -1) syserr("close3");
              if (dup(pfd[0]) != 0) fatal("dup2");
              if (close(pfd[0]) == -1 || close(pfd[1]) == -1) syserr("close4");
              execlp("wc", "wc", NULL);
              syserr("execl2");
        }
        if (close(pfd[0]) == -1 || close(pfd[1]) == -1) syserr("close5");
        while(wait(NULL) != -1) ;
}
```

Fifo

- **Pipe "normali"**
  - possono essere utilizzate solo da processi che hanno un "antenato" in comune
  - motivo: unico modo per ereditare descrittori di file
- **Named pipe**
  - permette a processi non collegati di comunicare
  - utilizza il file system per "dare un nome" al pipe
  - chiamate **stat, lstat**
    - Utilizzando queste chiamate su pathname che corrisponde ad un fifo, la macro **S_ISFIFO** restituirà **true**)
  - la procedura per creare un fifo è simile alla procedura per creare file

Fifo

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

- crea un FIFO dal **pathname** specificato
- la specifica dell'argomento **mode** è identica a quella di **open (O_RDONLY, O_WRONLY, O_RDWR, etc)**

Come si usa un FIFO?

- una volta creato un FIFO, le normali chiamate **open**, **read**, **write**, **close**, possono essere utilizzate per leggere il FIFO
- il FIFO può essere rimosso utilizzando **unlink**
- le regole per i diritti di accesso si applicano come se fosse un file normale

Fifo

- **Chiamata open:**
  - File aperto senza flag O_NONBLOCK
    - Se il file è aperto in lettura, la chiamata si blocca fino a quando un altro processo non apre il FIFO in scrittura
    - Se il file è aperto in scrittura, la chiamata si blocca fino a quando un altro processo non apre il FIFO in lettura
  - File aperto con flag O_NONBLOCK
    - Se il file è aperto in lettura, la chiamata ritorna immediatamente
    - Se il file è aperto in scrittura, e nessun altro processo è stato aperto in lettura, la chiamata ritorna un messaggio di errore

Fifo

- **Chiamata write**
  - se nessun processo ha aperto il file in lettura viene generato un segnale **SIGPIPE**
    - ignorato/catturato: write ritorna **–1** e **errno=EPIPE**
    - azione di default: terminazione

- **Atomicità**
  - Quando si scrive su un pipe, la costante **PIPE_BUF** specifica la dimensione del buffer del pipe
  - Chiamate **write** di dimensione inferiore a **PIPE_BUF** vengono eseguite in modo atomico
  - Chiamate **write** di dimensione superiore a **PIPE_BUF** possono essere eseguite in modo non atomico
    - La presenza di scrittori multipli può causare interleaving tra chiamate **write** distinte

Uso delle FIFO: comunicazione client-server

- **Processo base:**
- **Cliente:** chiama una fork;
  apre la fifo in scrittura;
  chiude l'unità di scrittura (1);
  duplica il descrittore della fifo sull'unità 1;
  Esegue il programma cliente

- Server: chiama una fork;
  apre la fifo in lettura;
  Chiude l'unità di lettura (0);
  duplica il descrittore della fifo sulla unità 0;
  Esegue il programma server che aspetta sulla fifo

Uso delle FIFO: comunicazione client-server

```
#include <stdio.h>
#include <fcntl.h>

/* semplice esempio di who|sort realizzato con FIFO
 Attenzione: la FIFO si accede tramite il nome, quindi questo meccanismo
   si puo' usare anche tra processi diversi
*/

main()
{
    int fd;
    mknod("ff", 010777,0);   /* creo la FIFO */
    switch(fork()){
    case -1: syserr("fork_who");
    case 0 : fd=open("ff", O_WRONLY);
          close(1); dup(fd); close(fd);
          execlp("who", "who", NULL); /* esegue  "who" che scrive */
    }                                       /*    sulla FIFO */
    switch(fork()){
    case -1: syserr("fork_wc");
    case 0: fd=open("ff", O_RDONLY);
        close(0); dup(fd); close(fd);
        execlp("sort", "sort", NULL); /* esegue sort che legge dalla FIFO */
    }
    wait(NULL);
        unlink("ff");
}
```

Uso delle FIFO: comunicazione client-server

```
/* CLIFIFO.C */
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/errno.h>
main(int argc,char *argv[])
{
  char buf[100]; /* stringa che invio sulla FIFO */
  int fd; /* descrittore file della FIFO */
  int i; int pidpro; /* pid del processo */   pidpro=getpid();
  /*aggancio la FIFO, aprendola in scrittura:*/
  if((fd=open("ff", O_WRONLY))==-1) syserr("Client: open ff");
  printf("Client %d :agganciata la FIFO\n",pidpro);
  chkpar(argc,argv,pidpro); /* controllo la presenza dei parametri */
  printf("argv = %s  %s\n", argv[0], argv[1]); /*preparo il messaggio:*/
  switch(argv[1][0]) {
      case 'c':break;
      case 'm':break;
      case 'r':break;
      case 'v':break;
      case 's':break;
      case 'q':break;
      default : fprintf(stderr,"Client %d (E): comando '%s' errato\n", pidpro,argv[1]);
      /* non cancello la FIFO: ci pensa il server */
      exit(1);
      break;
  } /*switch*/
```

```
  strcpy(buf,argv[1]);
  strcat(buf," ");
  i=2;
  while (argv[i] != NULL)
  { strcat(buf,argv[i]); strcat(buf," "); i++;  }
  /*invio il comando:*/
  if(write(fd,buf,strlen(buf))==-1) syserr("write");
  close(fd);
  printf("Client %d :inviata richiesta (%s)\n",pidpro,argv[1]);
  return(0);
} /*main*/
chkpar (argc,argv,pidpro) /*FUNZIONE:controlla presenza parametri*/
  int argc;
  char *argv[];
  int pidpro;
  { if (argc < 2) {
      fprintf (stderr, "Client %d (E): primo argomento assente\n", pidpro);
      exit (1); /* non cancello la fifo: ci pensa il server */
    } else {
      char l1;
      l1 = argv[1][1];
      if ( (l1=='r' || l1=='v' || l1=='s') &&   argc < 3  ) {
         fprintf (stderr, "Client %d (E): secondo argomento assente\n", pidpro);
         exit (1);
      }
      if ( (l1 == 'm' || l1 == 'c') && argc < 4)  {
          fprintf (stderr, "Client %d (E): terzo argomento assente\n", pidpro);
          exit (1);
      }
    }
} /* chkpar */  // END OF CLIFIFO
```

```
/* SERFIFO.C */
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern int errno;
main()
{
  int fd; /* descrittore file della FIFO */
  int nread; /* numero byte letti dalla FIFO */
  int i;
  static char buf[100]; /* perche` funziona solo con lo static? */
  char **varg; /* vettore dei comandi letti dalla FIFO */
  char *bufp; /* usato nella strtok */
  struct stat sbuf; /* usata nella chiamata di sistema stat */
  /*extern char **environ;*/
  mknod("ff",S_IFIFO|0666,0);   /* creo la FIFO */
  printf("Server: creata FIFO \"ff\"\n");
  /* apro la FIFO in lettura/scrittura (non in sola lettura altrimenti il server
    si blocca: */
  if((fd=open("ff",O_RDWR))==-1)
    syserr("open ff");
  /*environ[6]=""; PS1=''*/
```

```
  /*loop infinito:*/
  while(1)
   {
    printf("\nServer: attendo richieste...\n");
    printf("Comandi:    c file1 file2 (copia)\n");
    printf("            m file1 file2 (move)\n");
    printf("            r file1 (cancella)\n");
    printf("            s file1 (statistica)\n");
    printf("            v file1 (vista di un file)\n");
    printf("            q  (cancella la fifo)\n");
      /* Ricevo il messaggio(se non c'e` nulla attendo) */
    while((nread=read(fd,buf,sizeof(buf))) == 0) ;
    if(nread==-1)
   syserr("read");
    printf("Server: ricevo richiesta {%s}\n",buf);
    bufp=buf;
      /*printf("dopo bufp\n");*/
    for(i=0;i<3;i++) {
    if((varg[i]=strtok(bufp," "))==NULL)   break;
/*  printf("dentro lo strtok: %s\n",varg[i]);*/
    bufp=NULL;
    }
    /*  printf("prima dello switch\n"); */
```

```
switch(varg[0][0]) {
      case 'c': /*Copia di file*/
          switch(fork()){
          case -1:  syserr("fork di cp");
          case 0:   printf("Copio \"%s\" in \"%s\"\n",varg[1],varg[2]);
                    execlp("cp","cp",varg[1],varg[2],NULL);
                    syserr("exec di cp");
          default: if(wait(NULL)==-1) syserr("wait"); /*il server aspetta la fine*/
          }                                           /*dell'operazione di copia*/
          break;
      case 'm':     /*Spostamento di file*/
          switch(fork()){
          case -1:  syserr("fork di mv");
          case 0:   printf("Sposto \"%s\" in \"%s\"\n",varg[1],varg[2]);
                    execlp("mv","mv",varg[1],varg[2],NULL);
                    syserr("exec di mv");
          default: if(wait(NULL)==-1) syserr("wait"); /*il server aspetta la fine*/
           }                                         /*dell'operazione di spostamento*/
          break;
      case 'r': /*Cancellazione di file*/
          switch(fork()){
          case -1: syserr("fork di rm");
          case 0:   printf("Cancello \"%s\"\n",varg[1]);
                    execlp("rm","rm",varg[1],NULL);
                    syserr("exec di rm");
          default: if(wait(NULL)==-1) syserr("wait"); /*il server aspetta la fine*/
           }                                        /*dell'operazione di cancellazione*/
          break;
```

```
 case 's':          /*Statistica di un file*/
            if ( stat(varg[1], &sbuf) < 0 )   syserr("stat");
          printf("\nStato del file \"%s\" :\n",varg[1]);
          printf("Numero device: %ld\n",sbuf.st_dev);
          printf("Numero inode: %ld\n",sbuf.st_ino);
          printf("Tipo di file e permessi: %#o\n",(int)sbuf.st_mode);
          printf("Numero di hard links: %d\n",(int) sbuf.st_nlink);
          printf("ID utente del proprietario del file: %d\n", sbuf.st_uid);
          printf("ID gruppo del proprietario del file: %d\n", sbuf.st_gid);
          printf("Dimensione in byte del file: %ld\n", (long)sbuf.st_size);
          printf("Ora ultimo accesso: %s",ctime(&(sbuf.st_atime)));
          printf("Ora ultima modifica: %s",ctime(&(sbuf.st_mtime)));
          printf("Ora ultimo cambiamento di stato: %s", ctime(&(sbuf.st_ctime)));
          break;
      case 'v':     /*Vista di un file*/
          switch(fork()){
          case -1:  syserr("fork di cat");
          case 0:   printf("Contenuto di \"%s\"\n",varg[1]);
                    execlp("cat","cat",varg[1],NULL);
                    syserr("exec di cat");
          default: if(wait(NULL)==-1) syserr("wait"); /*il server aspetta la fine*/
           }                                          /*dell'operazione di cat*/
          break;
      case 'q':  /* cancello la FIFO */
          close(fd); unlink("ff"); printf("Server: FIFO cancellata\n");
          exit(0);
    } /*switch*/
    for(i=0;i<=30;i++) buf[i]='\0';
  } /* fine del while */
return(0);
} /*main*/
```