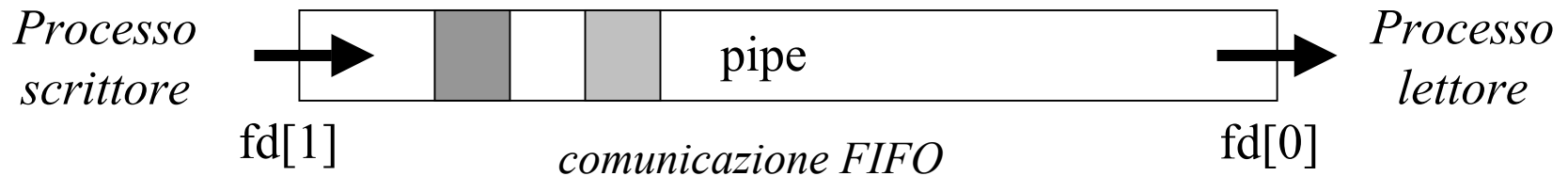


Pipe

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

Le pipe sono canali di comunicazione unidirezionali che costituiscono un primo strumento di comunicazione (con diverse limitazioni), basato sullo scambio di messaggi, tra processi UNIX

La creazione di una *pipe* mediante la primitiva omonima restituisce in `fd` due descrittori: `fd[0]` per la lettura e `fd[1]` per la scrittura

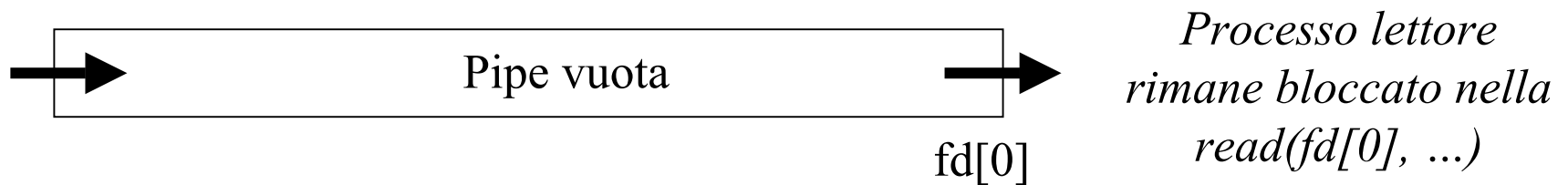


Lettura e scrittura sulla *pipe* sono ottenute mediante le normali primitive `read` e `write`

Pipe

L'accesso alle pipe è regolato da un meccanismo di sincronizzazione:

- la lettura da una pipe da parte di un processo è bloccante se la pipe è vuota (in attesa che arrivino i dati)



- la scrittura su una pipe da parte di un processo è bloccante se la pipe è piena



Pipe

La sincronizzazione è legata alla implementazione delle pipe:

ad ogni pipe viene associato un buffer circolare nel kernel (e quindi inaccessibile direttamente ai processi) di dimensione prefissata (ad es. 4 Kb)

- la lettura da una pipe vuota è bloccante per proteggere dall' underflow del buffer
- la scrittura su una pipe piena è bloccante per proteggere dall' overflow del buffer

Pipe

Le pipe sono anche dette pipe anonime (*unnamed pipe*) perché non sono associate ad alcun nome nel File System:

⇒ solo i processi che possiedono i descrittori possono comunicare attraverso una pipe

La tabella dei file aperti di un processo (contenente i descrittori della pipe) viene duplicata per i processi figli:

⇒ la comunicazione attraverso una pipe anonima è quindi possibile solo per processi in relazione di parentela che condividono un progenitore

Pipe

Esempio di comunicazione su pipe

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define N_MESSAGGI 10

int main()
{
    int pid, j,k, piped[2];

    /* Apre la pipe creando due file descriptor,
       uno per la lettura e l'altro per la scrittura
       (vengono memorizzati nei due elementi dell'array piped[]) */
    if (pipe(piped) < 0)
        exit(1);

    if ((pid = fork()) < 0)
        exit(2);
    else if (pid == 0) /* Il figlio eredita una copia di piped[] */
    {
        /* Il figlio e' il lettore dalla pipe: piped[1] non gli serve */
        close(piped[1]);

        /* Continua ... */
    }
}
```

Pipe

```
/* ... continua */

for (j = 1; j <= N_MESSAGGI; j++)
    {
        read(piped[0], &k, sizeof (int));
        printf("Figlio: ho letto dalla pipe il numero %d\n", k);
    }
    exit(0);
}
else {
    /* Processo padre */
    /* Il padre e' scrittore sulla pipe: piped[0] non gli serve */
    close(piped[0]);
    for (j = 1; j <= N_MESSAGGI; j++)
        {
            write(piped[1], &j, sizeof (int));
        }

    wait(NULL);
    exit(0);
}
}
```

Pipe

Implementazione del *piping* di comandi

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>

int join(char* com1[], char *com2[])
{
    int status, pid;
    int piped[2];

    switch(fork())
    {
        case -1: /* errore */ return 1;
        case 0: /* figlio */ break; /* esce dal case */
        default: /* padre: attende il figlio */ wait(&status); return
status;
    }

    /* il figlio crea la pipe e un proprio figlio (nipote del primo
processo) */
    if (pipe(piped) < 0)
        return 2;
    if ((pid = fork()) < 0)
        return 3;
    else if (pid == 0)
        { /* Nipote: continua ... */
```

Pipe

```
/* ... continua il nipote */
/* Lo stdin corrente non interessa a com2 (lettore): chiude il
descrittore 0 */
    close(0);
    /* Si richiede un nuovo descrittore per la lettura dalla pipe:
       il primo descrittore libero e' lo 0 (stdin) che viene
       associato alla lettura dalla pipe */
    dup(piped[0]);

    /* Gli altri descrittori delle pipe non servono piu':
       il lettore usa stdin per leggere dalla pipe */
    close(piped[0]);
    close(piped[1]);

    /* Esecuzione del comando 2 (che ha la pipe come stdin) */
    execvp(com2[0], com2);
    perror("exec com2"); return 4;
}
else { /* Figlio */
    /* Lo stdout corrente non interessa a com1 (scrittore): chiude
il descrittore 1 */
    close(1);
    /* Si richiede un nuovo descrittore per la scrittura sulla pipe:
       il primo descrittore libero e' 1 (stdout) che viene
       associato alla scrittura sulla pipe */
    dup(piped[1]); /* Figlio continua ... */
}
```

Pipe

```
/* ... continua figlio */
    /* Gli altri descrittori delle pipe non servono piu':
       lo scrittore usa stdout per scrivere sulla pipe */
    close(piped[0]);
    close(piped[1]);

    /* Esecuzione del comando 1 (che ha la pipe come stdout) */
    execvp(com1[0], com1);
    perror("exec com1"); return 5;
}
}

int main(int argc, char **argv)
{
    int integri, i, j;
    char *temp1[10], *temp2[10];

    /* si devono fornire nella linea di comando due comandi distinti,
       utilizzando \| oppure '|' o "|" per indicare il piping : occorre
       evitare che il simbolo "|" venga direttamente interpretato dallo shell
       come una pipe) */

    /* main continua ... */
```

Pipe

```
/* ... continua */

if (argc > 2) {
    for (i = 1; (i < argc) && ((strcmp(argv[i], "|"))!=0); i++)
        temp1[i-1] = argv[i];
    temp1[i-1] = (char *)0;
    i++;
    for (j = 1; i < argc; i++, j++)
        temp2[j-1]=argv[i];
    temp2[j-1] = (char *)0;
}
else {
    printf("errore");
    exit(-2);
}
integi = join(temp1, temp2);
exit(integi);
}
```

FIFO

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

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

Sono anche dette pipe con nome (quelle classiche sono anonime)

Viene creata una entità nel filesystem (`pathname`) accessibile anche da processi non in relazione di parentela

Vanno utilizzate le normali SysCall che si usano per file e pipe (`open`, `read/write`, `close`, ...)

Una FIFO deve essere aperta con `open` dopo che è stata creata con `mkfifo`

FIFO

Effetto del flag `O_NONBLOCK` :

- in sua assenza una `open` in sola lettura blocca il processo fino a quando un altro processo apre la FIFO in scrittura (lo stesso per la sola scrittura);
- se `O_NONBLOCK` viene specificato, l'apertura in sola lettura ritorna immediatamente ma quella in sola scrittura ritorna con errore che vale `ENXIO` se non ci sono processi lettori

E' normale avere piu' scrittori su una FIFO

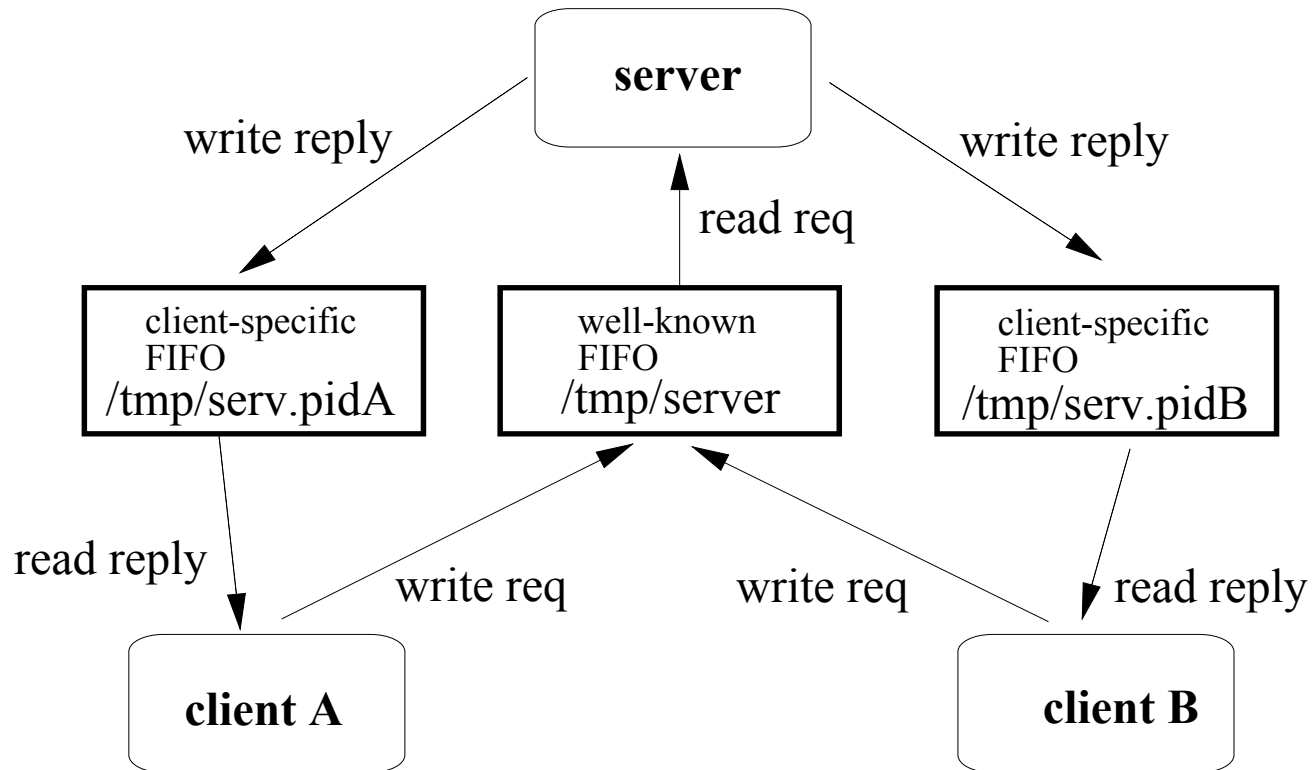
Le `write` sono atomiche se riguardano meno di `PIPE_BUF` bytes; in Linux la definizione di `PIPE_BUF` si trova in `/usr/include/linux/limits.h`)

FIFO

Le FIFO sono adatte per applicazioni client-server in locale:

- il server apre una FIFO con un nome noto (ad es. /tmp/server)
- i client scrivono alla FIFO le loro richieste

Per le risposte il server utilizza una FIFO per ogni cliente



Esercizi

Uno schema generale di soluzione per gli esercizi UNIX

5 fasi principali con un preciso ordinamento:

1 Controllo degli argomenti di invocazione

2 Selezione della politica di gestione dei segnali

- per il processo padre (main) e/o per i figli che la ereditano

3 Creazione pipe ed eventuale inizializzazione del contenuto

- le pipe saranno così accessibili ai figli e ai discendenti

4 Creazione dei processi figli

5 Esecuzione dei figli confinata in specifiche funzioni

- nel caso alcuni processi abbiano lo stesso comportamento si ottiene una maggiore compattezza del codice:
⇒ le funzioni vanno scritte in modo parametrico rispetto ai propri argomenti e al PID del processo corrente

Esercizi

Esercizio n. 1

```
/******
```

Un processo padre crea N (N numero pari) processi figli.

Ciascun processo figlio P_i è identificato da una variabile intera i ($i=0,1,2,3\dots,N-1$). Due casi:

1. Se $argv[1]$ è uguale ad 'a' ogni processo figlio P_i con i pari manda un segnale (SIGUSR1) al processo $i+1$;

2. Se $argv[1]$ è uguale a 'b' ogni processo figlio P_i con $i < N/2$ manda un segnale (SIGUSR1) al processo $i + N/2$.

```
*****/
```

```
#include <stdio.h>
#include <ctype.h>
#include <signal.h>
```

```
#define N2 5
#define N N2*2
```

```
int pg[2];
int tabpid[N];
char arg1;
```

```
/* Continua ... */
```

Esercizi

Esercizio n. 1 (cont.)

```
void handler(int signo)
{
    printf("Sono il processo %d e ho ricevuto il segnale %d\n",
        getpid(), signo);
}

/* Funzione eseguita da ciascun figlio: ne definisce il
comportamento a regime */
int body_proc(int id)
{
    printf("Sono il processo %d con id=%d\n", getpid(), id);

    if (arg1=='a')
    { /* % è l'operatore modulo, il resto della divisione intera */
        if (id % 2) pause(); /* id dispari */
        else
            { /* id pari */
                read(pg[0], tabpid, sizeof tabpid);
                write(pg[1], tabpid, sizeof tabpid);
                kill(tabpid[id+1], SIGUSR1);
            }
    }
    else
    { /* Continua ... */
```

Esercizi

Esercizio n. 1 (cont.)

```
if (id >= N/2) pause();
    else
        {
            read(pg[0],tabpid,sizeof tabpid);
            write(pg[1],tabpid,sizeof tabpid);
            kill(tabpid[id+N/2],SIGUSR1);
        }
    }
return(0);
}

main (int argc, char* argv[])
{
    int i, status;

    /* 1) Controllo argomenti */

    if(argc != 2)
        {
            fprintf(stderr,"Uso:  %s a\n(oppure)\n%s b \n",argv[0],argv[0]);
            exit(-1);
        }
    /* Continua ... */
```

Esercizi

Esercizio n. 1 (cont.)

```
arg1= argv[1][0]; /* primo carattere del secondo argomento */

/* 2) Gestione dei segnali che i figli erediteranno */
signal(SIGUSR1,handler);

/* 3) Creazione della pipe di comunicazione tra padre e figli */
if (pipe(pg)<0)
{
    perror("creazione pipe"); exit(-1);
}

/* 4) Creazione dei processi figli */
for (i=0;i<N;i++)
{
    if ((tabpid[i]=fork())<0)
    {
        perror("fork");
        exit(-1);
    }
    else
        if (tabpid[i]==0)
            { /* 5) Esecuzione dei figli confinata all'interno di una
              funzione specifica */
                status= body_proc(i);
                /* Continua ... */
            }
```

Esercizi

Esercizio n. 1 (cont.)

```
/* 5b) Terminazione dei figli: si evita che i figli
    rimanendo all'interno del ciclo for generino a loro
    volta altri processi
    exit(status);
    */
}

/* Il padre pone la tabella (che contiene tutti gli N pid dei figli)
    nella pipe */
printf("Sono il padre e scrivo sulla pipe la tabella dei pid\n");
write(pg[1],tabpid,sizeof tabpid);

exit(0);
}
```

Esercizi

Esercizio n. 2

```
/*
```

Si progetti in ambiente Unix/C l'interazione di tre processi P1, P2 e Pa mediante segnali e una pipe pa :

- P1 e P2 e inviano ciascuno segnali SIGUSR1(P1)/SIGUSR2(P2) a Pa, attendendo per un intervallo di durata casuale (massimo 5 secondi) tra un segnale e l'altro;
- il numero di segnali che ciascun processo deve inviare viene determinato dall'unico argomento di invocazione del programma main;
- al termine della spedizione degli segnali, ciascun processo P1/P2 invia mediante la pipe pa un messaggio a Pa contenente il proprio pid e il numero di segnali spediti;
- alla ricezione dei due messaggi Pa deve visualizzare il numero di segnali spediti e il numero di segnali effettivamente ricevuti.

Si utilizzino le primitive per la gestione affidabile dei segnali.

```
*/
```

Esercizi

Esercizio n. 2

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int pa[2];
int numsegnali;
int sigusr1cnt;          /* contatore segnali SIGUSR1 ricevuti */
int sigusr2cnt;          /* contatore segnali SIGUSR2 ricevuti */

/* Gestore segnale SIGUSR1 per il padre Pa */
void usr1_handler()
{
    ++sigusr1cnt;
    printf("%d riceve SIGUSR1: %d\n", getpid(), sigusr1cnt);
}
/* Gestore segnale SIGUSR2 per il padre Pa */
void usr2_handler()
{
    ++sigusr2cnt;
    printf("%d riceve SIGUSR2: %d\n", getpid(), sigusr2cnt);
}

/* Continua ... */
```

Esercizi

Esercizio n. 2 (cont.)

```
main(int argc, char *argv[])
{
    int pid;
    struct sigaction act;

    /* 1) Controllo argomenti */
    if(argc !=2)
    {
        fprintf(stderr,"Uso: %s numsegnali\n", argv[0]);
        exit(-1);
    }
    numsegnali=atoi(argv[1]);
    if(numsegnali <=0 || numsegnali>100) /* Sanity checking */
        numsegnali=10;

    /* 2) Gestione dei segnali del processo Pa */
    act.sa_handler= usr1_handler; /* Gestore del SIGUSR1 */
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask,SIGUSR2); /* Blocca SIGUSR2 nel gestore */
    act.sa_flags= SA_RESTART; /* Restart automatico di una primitiva di
                                lettura interrotta dal segnale SIGUSR1
                                (ad.es. la read dalla pipe pa) */

    sigaction(SIGUSR1,&act, NULL);
    /* Continua ... */
}
```

Esercizi

Esercizio n. 2 (cont.)

```
act.sa_handler= usr2_handler;    /* Gestore del SIGUSR2 */
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask,SIGUSR1); /* Blocca SIGUSR1 nel gestore */
act.sa_flags= SA_RESTART; /* Restart automatico di una primitiva di
                           lettura interrotta dal segnale SIGUSR2
                           (ad.es. la read dalla pipe pa)    */
sigaction(SIGUSR2, &act, NULL);

/* 3) Creazione della pipe di comunicazione tra figli e padre */
if(pipe(pa) <0)
    {perror("creazione pipe"); exit(-1);
}

/* 4) Creazione dei processi figli */
if((pid=fork())==0)
    {/* 5) Esecuzione dei figli confinata all'interno di una
        funzione specifica */
        body_figlio(SIGUSR1);
        exit(0);
    }
else
    /* Continua ... */
```

Esercizi

Esercizio n. 2 (cont.)

```
if((pid=fork())==0)
    { /* 5) Esecuzione dei figli confinata all'interno di una
        funzione specifica */
        body_figlio(SIGUSR2);
        exit(0);
    }
else
    body_padre();
}
/* Funzione eseguita da ciascun figlio: ne definisce il comportamento
a regime */

body_figlio(int signo) /* segnale che questo figlio manda al padre */
{
int i,mesg[2];

for(i=0; i<numsegnali; i++)
    {
        kill(getppid(),signo);
        printf("Processo %d invia %s\n",getpid(),
                (signo==SIGUSR1)? "SIGUSR1":"SIGUSR2");
        sleep(1+rand()%5); /* Attesa tra 1 e 5 secondi */
    }
}
/* Continua ... */
```

Esercizi

Esercizio n. 2 (cont.)

```
/* Preparazione messaggio da inviare sulla pipe pa al padre */
mesg[0]= getpid();          /* indicazione del mittente */
mesg[1]= numsegnali;         /* messaggio */

write(pa[1], mesg, sizeof(mesg));
}

body_padre()
{
int mesga[2],mesgb[2];

read(pa[0],mesga,sizeof(mesga));
read(pa[0],mesgb,sizeof(mesgb));

printf("Processo %d ha mandato %d segnali\n",mesga[0],mesga[1]);
printf("Processo %d ha mandato %d segnali\n",mesgb[0],mesgb[1]);
printf("Ricevuti %d SIGUSR1 e %d SIGUSR2\n\n", sigusr1cnt, sigusr2cnt);
}
```

Esercizi

Esercizio n. 3

```
/******
```

Si progetti in ambiente Unix/C la seguente interazione di processi:

- il sistema consiste di 3 processi: un processo padre (P1) che provvede alla creazione di 2 processi figli (P11 e P12) e di due pipe pa e pb ;
 - P1 provvede a generare la sequenza dei primi N interi, scrivendoli nella pipe pa;
 - P11 inizialmente preleva i numeri pari da pa e li riscrive in pb ;
 - P12 preleva l (con $l > N/4$) interi da pb e li scrive su un primo file;
 - dopo aver letto l messaggi, P12 invia un segnale SIGUSR1 a P11 per effetto del quale esso passa a prelevare i numeri dispari da pa e a scriverli in pb ;
 - P12 preleva i rimanenti interi da pb e li scrive su un secondo file.
- ```
*****/
```

# Esercizi

## Esercizio n. 3 (cont.)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <ctype.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ioctl.h>

#include <stdio.h>

#define FILENAME1 "prova-1.txt"
#define FILENAME2 "prova-2.txt"

#define PARI 0
#define DISPARI 1

pid_t pid11, pid12;
int pa[2],pb[2];
int n,l;

int pari_o_dispari=0; /* 0 PARI --- 1 DISPARI */
/* Continua ... */
```

# Esercizi

## Esercizio n. 3 (cont.)

```
void sigusr1_handler(int signo)
{
 if(pari_o_dispari==PARI)
 pari_o_dispari=DISPARI ;
 else
 pari_o_dispari=PARI ;

 printf("Il processo p11 ha ricevuto SIGUSR1: ora sono selezionati
i %s\n", (pari_o_dispari==PARI)? "pari":"dispari");
}

int body_p11()
{
 int i,value;

 printf("Sono il processo p11\n");
 pari_o_dispari=0;

 while(1) /* Il processo viene terminato con il SIGKILL da p12 */
 {
 read(pa[0], &value, sizeof(value));

 if(value % 2) /* dispari */
 {
 if(pari_o_dispari != PARI)
 {/* Continua ... */
```

# Esercizi

## Esercizio n. 3 (cont.)

```
 printf("%d dispari ->\n",value);
 write(pb[1], &value, sizeof(value));
 }/* inviati a p12 */
}
else /* pari */
{
 if(pari_o_dispari == PARI)
 {
 printf("%d pari ->\n", value);
 write(pb[1], &value, sizeof(value));
 }/* inviati a p12 */
}
sleep(1);
}
}

int body_p12()
{
 int i,fd1,fd2;
 int available, timeout;
 char number[10];

 printf("Sono il processo p12\n");

 if ((fd1=open(FILENAME1, O_WRONLY | O_CREAT |O_TRUNC, 0644))<0)
 { /* Continua ... */
```

# Esercizi

## Esercizio n. 3 (cont.)

```
 perror("open");
 exit(-1);
 }
 if ((fd2=open(FILENAME2, O_WRONLY | O_CREAT |O_TRUNC, 0644))<0)
 {
 perror("open");
 exit(-1);
 }

 for(i=0;i<1;i++)
 {
 read(pb[0],&i, sizeof(i));
 printf("->%d in %s\n", i,FILENAME1);
 sprintf(number,"%d\n", i); /*Per scrivere numeri come testo*/
 write(fd1,number,strlen(number));
 sleep(1);
 }

 kill(pid11,SIGUSR1);

 timeout =0 ;
 while(timeout < 4)
 {
 ioctl(pb[0],FIONREAD, &available);
 if(available >0)
 { /* Continua ... */
```

# Esercizi

## Esercizio n. 3 (cont.)

```
 timeout=0;
 read(pb[0], &i, sizeof(i));
 printf("->%d in %s\n", i, FILENAME2);
 sprintf(number, "%d\n", i); /*Per scrivere numeri come testo*/
 write(fd2, number, strlen(number));
 }
 else
 timeout++;

 sleep(1);
}

printf("Timeout in lettura per il processo p12 che termina p11\n");
kill(pid11, SIGKILL);

return(0);
}

main (int argc, char* argv[])
{
 int i, status;
 struct sigaction act;

 /* 1) Controllo argomenti */
 if(argc != 3)
 { /* Continua ... */
```

# Esercizi

## Esercizio n. 3 (cont.)

```
 fprintf(stderr, "Uso: %s N l\n", argv[0]);
 exit(-1);
}
```

```
n = atoi(argv[1]);
l = atoi(argv[2]);
```

```
if (l <= n/4)
{
 fprintf(stderr, "l deve essere maggiore di N/4\n");
 exit(-2);
}
```

```
/* 2) Gestione dei segnali che i figli erediteranno */
```

```
act.sa_handler= sigusr1_handler;
sigemptyset(&act.sa_mask);
act.sa_flags= SA_RESTART; /* Per la read di p11 - vedi es. n. 2 */
if(sigaction(SIGUSR1, &act, NULL) <0)
{
 perror("sigaction");
 exit(-2);
}
```

```
/* 3) Creazione delle pipe di comunicazione */
```

```
if (pipe(pa)<0)
 { /* Continua ... */
```

# Esercizi

## Esercizio n. 3 (cont.)

```
 perror("pipe error");
 exit(-3);
}
if (pipe(pb)<0)
{
 perror("pipe error");
 exit(-4);
}

/* 4) Creazione dei processi figli */
if ((pid11=fork())<0)
{
 perror("fork error");
 exit(-5);
}
else
 if (pid11==0)
 { /* 5) Esecuzione del figlio in una funzione specifica */
 status= body_p11();
 exit(status);
 }

if ((pid12=fork())<0)
{
 perror("fork error");
 /* Continua ... */
}
```

# Esercizi

## Esercizio n. 3 (cont.)

```
 exit(-5);
 }
else
 if (pid12==0)
 { /* 5) Esecuzione del figlio in una funzione specifica */
 status= body_p12();
 exit(status);
 }

/* Padre */
for(i=0;i<n;i++)
 write(pa[1],&i, sizeof(i));

/* Attende la terminazione di entrambi i figli */
wait(NULL);
wait(NULL);

exit(0);
}
```