

Programmazione di sistema in Linux: System Call per le IPC (InterProcessCommunication)

E. Mumolo

Tipi di IPC

- PIPE
- FIFO
- Shared Memory
 - `shm_open()` + `mmap()`, `shm_unlink()`, `munmap()`
- Semafori
 - `sem_open()`, `sem_close()`, `sem_unlink()`, `sem_post()`, `sem_wait()`,
- Segnali
- Scambio di Messaggi
 - `mq_open()`, `mq_close()`, `mq_send()`, `unlink()`, `mq_receive()`, `sem_init()`
- Socket

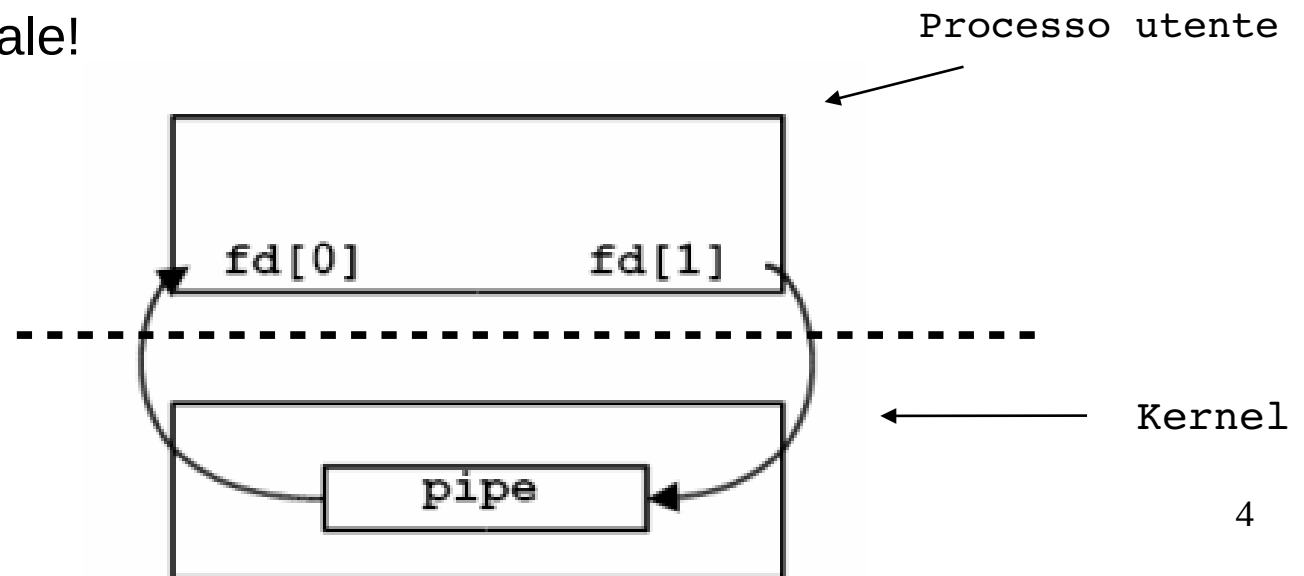
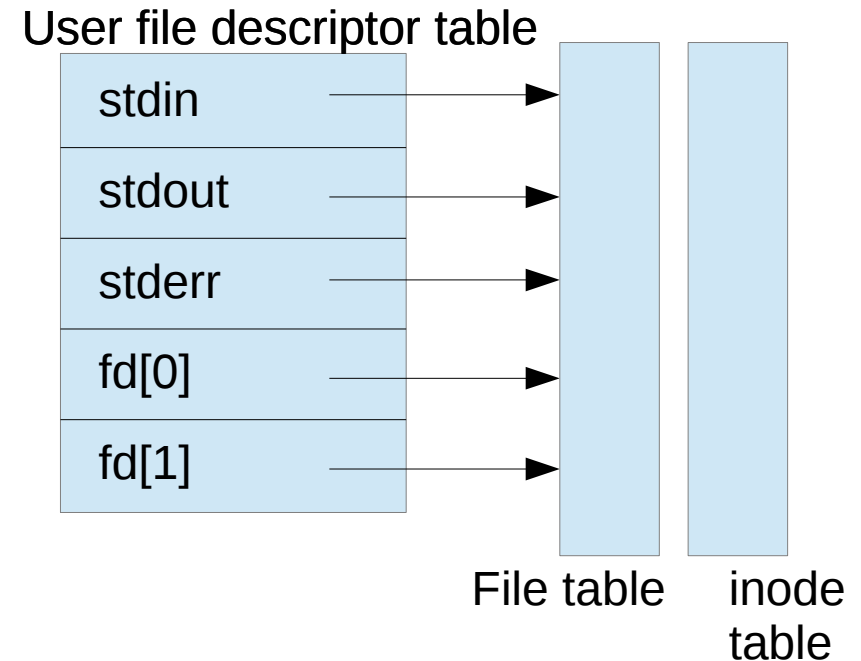
Pipe

- **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) introdotta 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 fd[2]);
```

- Ritorna due descrittori di file attraverso l'argomento **fd**
 - **fd[0]** è aperto in lettura
 - **fd[1]** è aperto in scrittura
 - L'output di **fd[0]** è l'input di **fd[1]**
- La pipe è monodirezionale!
- Implementa il modello PRODUTTORE CONSUMATORE



Stringhe passate tramite Pipe tra due processi

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define MAXLINE 1024
```

```
int main(int argc, char *argv[])
{
```

```
    int n, pfd[2];
```

```
    pid_t pid;
```

```
    char line[MAXLINE], msg[80];
```

```
    strcpy(msg, argv[1]); /* copia la stringa data in linea */
```

```
    if (pipe(pfd) < 0) {perror("pipe"); exit(0);}
```

```
    if ( (pid = fork()) < 0) {perror("pipe"); exit(0);}
```

```
    else if (pid > 0) { /* client */
```

```
        close(pfd[0]);
```

```
        printf("sono il padre. Sto scrivendo il messaggio nella pipe\n");
```

```
        sleep(2); /* aspetto a scrivere */
```

```
        write(pfd[1], msg, sizeof(msg));
```

```
        wait();
```

```
    } else { /* server */
```

```
        close(pfd[1]);
```

```
        printf("sono il processo figlio. Aspetto il messaggio\n");
```

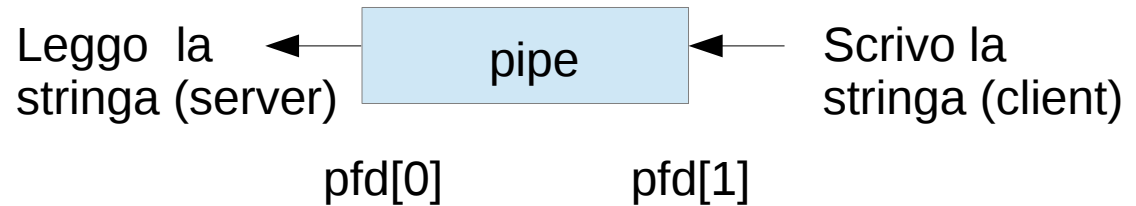
```
        n = read(pfd[0], line, MAXLINE);
```

```
        printf("ricevuto: %s\n", line);
```

```
    }
```

```
    exit(0);
```

```
}
```



La lettura della pipe aspetta che ci sia qualcosa dentro

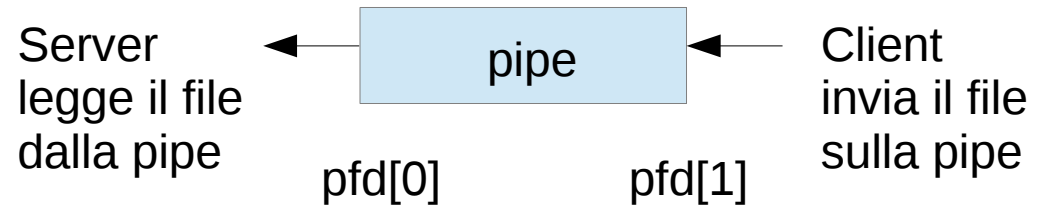
Invio di un file con approccio client-server

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define maxsize 1000

char buffer [maxsize];

int main(){
    int pfd[2], pid, status;

    pipe(pfd);
    pid=fork();
    if(pid>0)
    {
        close(pfd[1]);
        server(pfd[0]);
        wait(&status);
        exit(0);
    }
    else
    {
        close(pfd[0]);
        sleep(1); /* il client aspetta mentre il server legge la pipe */
        client(pfd[1]);
        exit(0);
    }
}
```

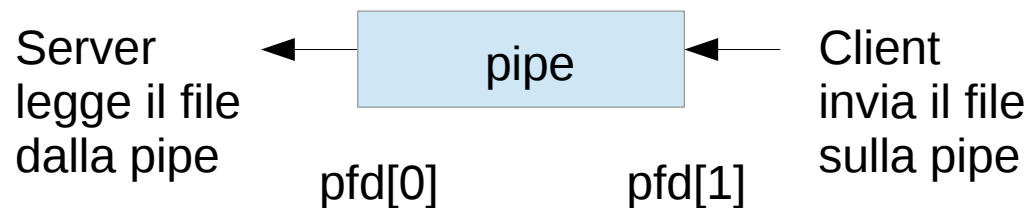


Invio di un file con approccio client-server

```
void server(int readfd){
    while (read(readfd,buffer,sizeof(buffer)) > 0) /*legge le righe dalla pipe */
        printf("%s",buffer);                      /*stampa*/
    printf("\n");
}

void client(int writefd){
    FILE * fp;
    char line[1000];

    fp=fopen("/home/mumolo/testsmall.txt","r");
    while(fgets(line,sizeof(line),fp)!=NULL) /* legge per righe il file */
        write(writefd,line,sizeof(line));    /* scrive le righe sulla pipe */
}
```



Pipe bidirezionali

- L'unico modo per avere pipe bidirezionali è di averne due, una per ogni direzione

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define maxsize 1000
char buffer [maxsize];
```

```
int main()
{
    int pfd1[2],pfd2[2], pid, status;
```

```
    pipe(pfd1); pipe(pfd2);
```

```
    pid=fork();
```

```
    if(pid>0)
```

```
    {
```

```
        close(pfd1[0]); close(pfd2[1]);
```

```
        sleep(1); /* il client aspetta mentre il server legge la pipe */
```

```
        server(pfd2[0],pfd1[1]);
```

```
        wait(&status);
```

```
        exit(0);
```

```
    }
```

```
    else
```

```
    {
```

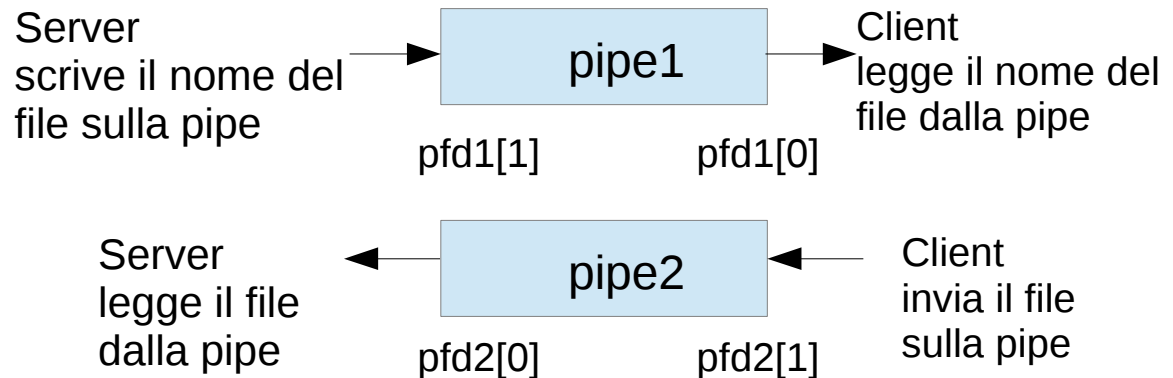
```
        close(pfd1[1]); close(pfd2[0]);
```

```
        client(pfd1[0],pfd2[1]);
```

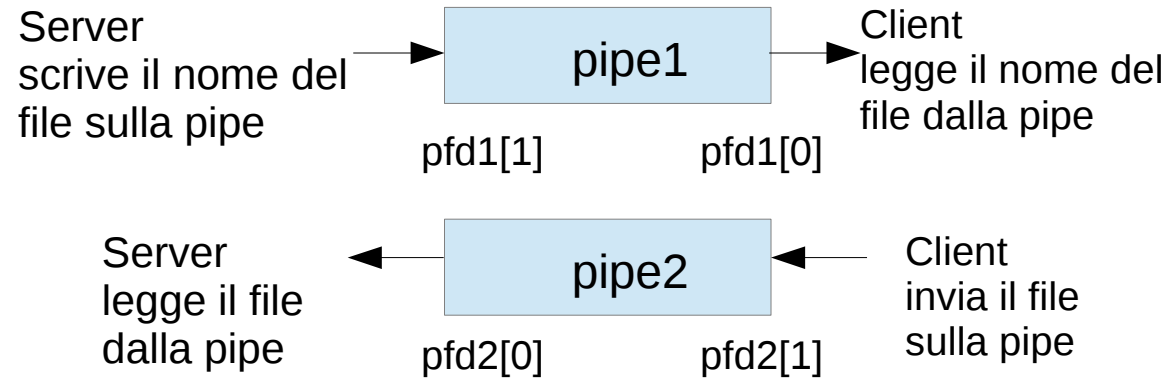
```
        exit(0);
```

```
    }
```

```
}
```



Pipe bidirezionali



```
void server(int readfd,int writefd)
{
    printf ("dai il pathname: ");

    fgets (buffer, sizeof (buffer), stdin);

    write(writefd,buffer, sizeof (buffer));
    while (read(readfd,buffer,sizeof(buffer)) > 0)    printf("%s",buffer);
    printf("\n");
}

void client(int readfd,int writefd)
{
    FILE * fp;
    char line[1000];

    read(readfd,buffer,sizeof(buffer));
    if (strchr (buffer, '\n')) *strchr(buffer, '\n') = '\0';
    fp=fopen(buffer,"r");

    while(fgets(line,sizeof(line),fp)!=NULL)
        write(writefd,line,sizeof(line));
}
```

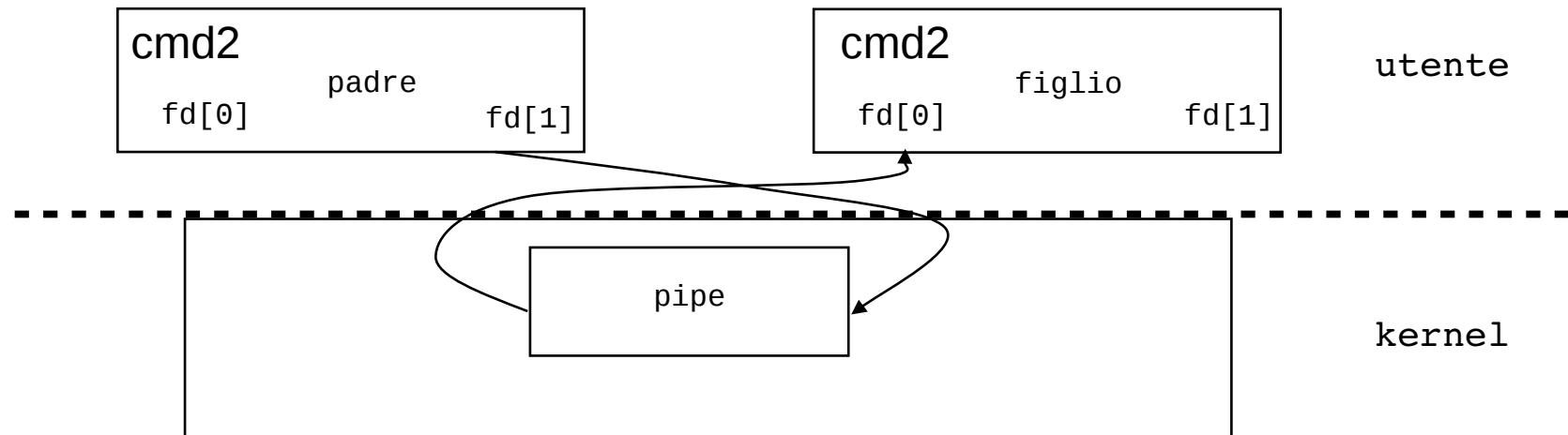
Comunicazione tra processi: *cmd1 | cmd2*

■ Come utilizzare le pipe?

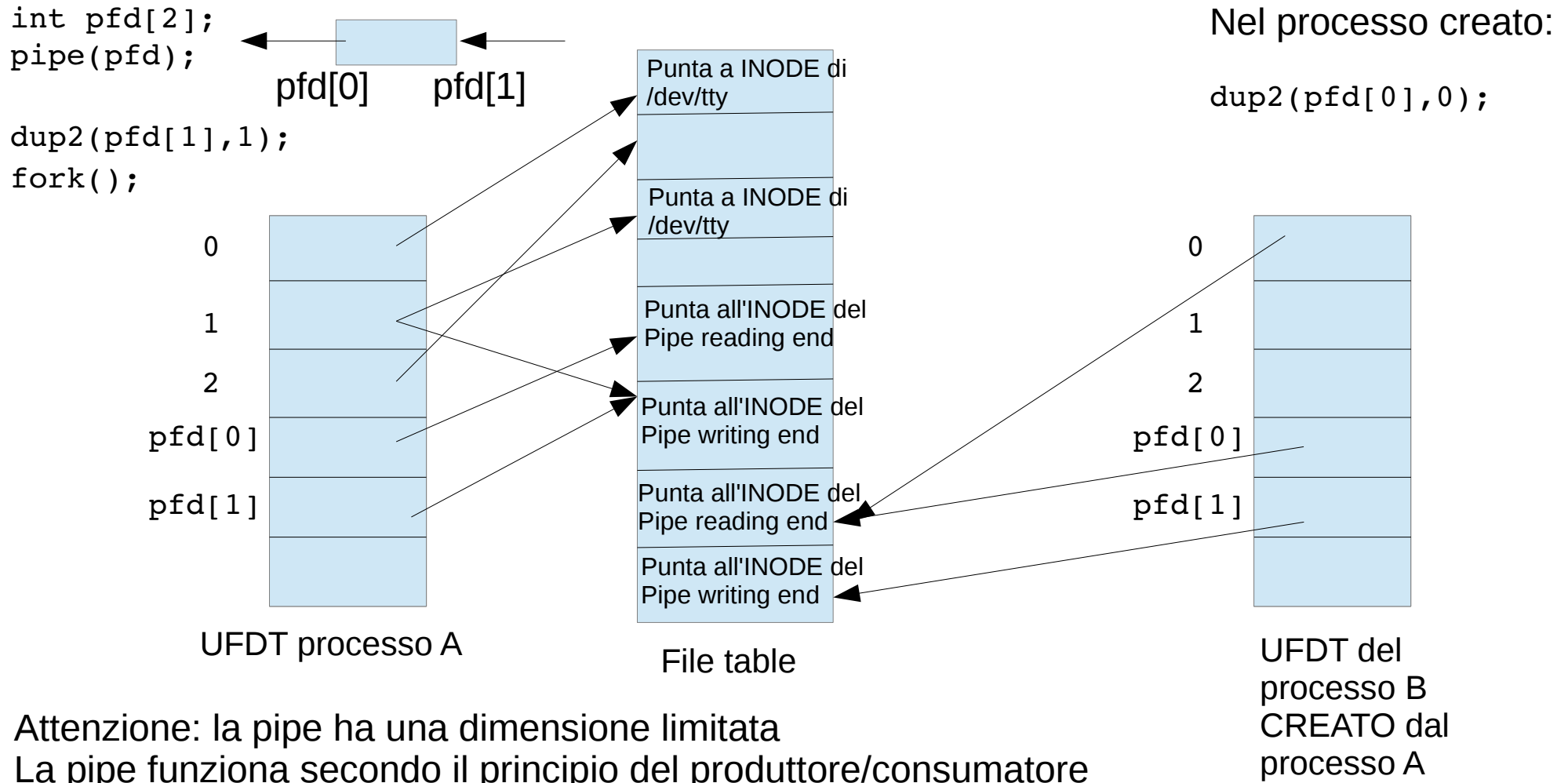
- ❑ Gli eventuali figli, ereditano la UserFileDescriptorTable
- ❑ I canali non utilizzati vanno chiusi
- ❑ Redirezione standard input e output

■ Esempio: padre | figlio

- ❑ Il padre
 - Duplica **fd[1]** sulla 1
 - Chiude **fd[0]**
- ❑ Il figlio
 - Duplica **fd[0]** sulla 0
 - Chiude **fd[1]**

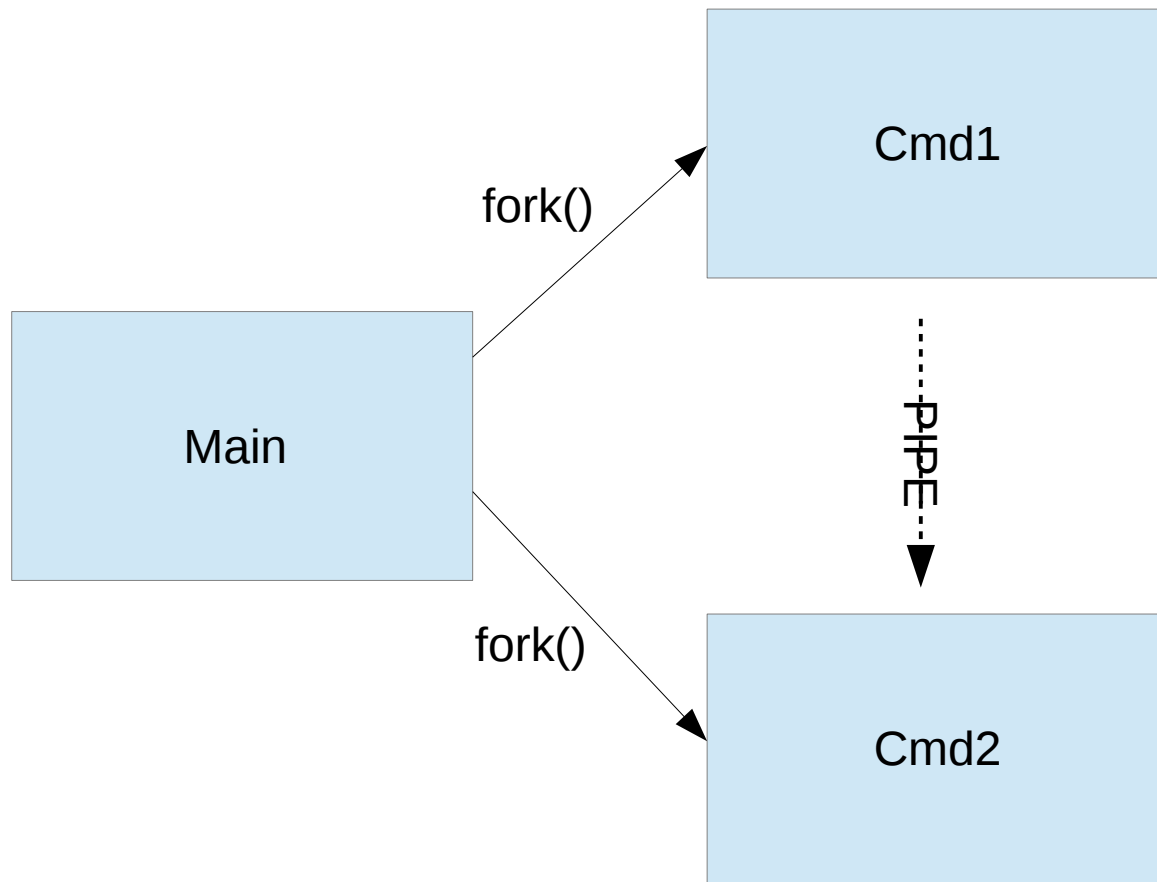


Comunicazione tra processi: : creazione e redirezione delle pipe



Il processo A scrive sullo standard output (1) in realtà scrive sul writing end della pipe!
Il processo B legge dallo standard input (0). In realtà legge dal reading end della pipe!

Prima architettura di cmd1 | cmd2

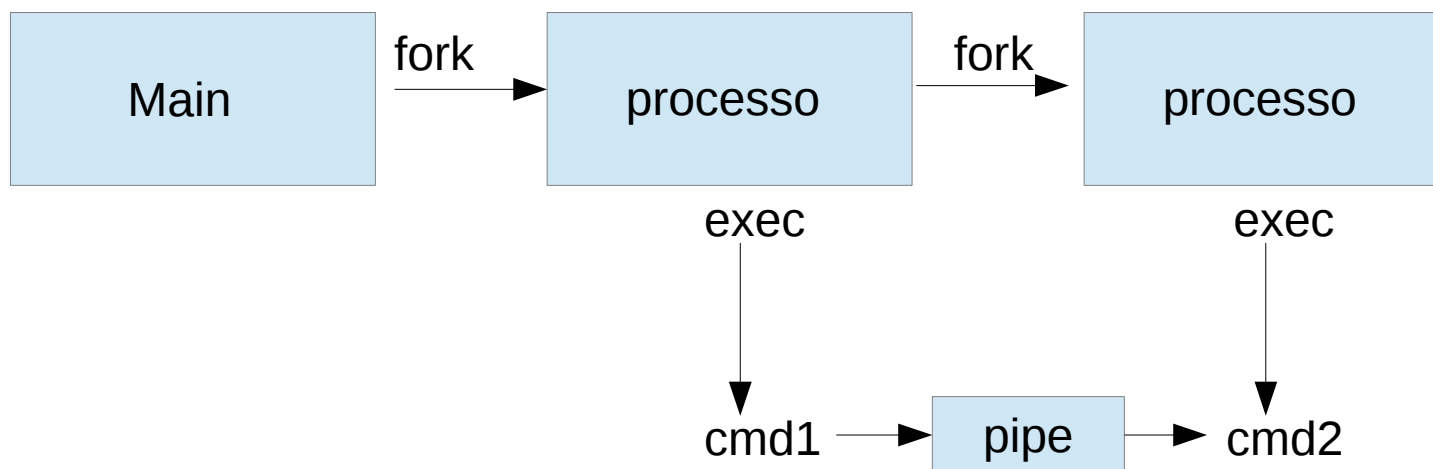


Prima architettura di cmd1 | cmd2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
main()
{
    int pfd[2];
    pipe(pfd);
    switch(fork()) {
        case -1: err_sys("fork");
        case 0: if (close(1) == -1) err_sys("close");
                dup(pfd[1]);
                close(pfd[0]); close(pfd[1]);
                execlp("who", "who", NULL);
                err_sys("execl1");
    }

    switch(fork()) {
        Case -1: err_sys("fork");
        case 0: if(close(0) == -1) err_sys("close3");
                dup(pfd[0]);
                close(pfd[0]); close(pfd[1]);
                execlp("wc", "wc", NULL);
                err_sys("execl2");
    }
    close(pfd[0]); close(pfd[1]);
    wait(NULL) ;
}
```

Architettura alternativa cmd1 | cmd2

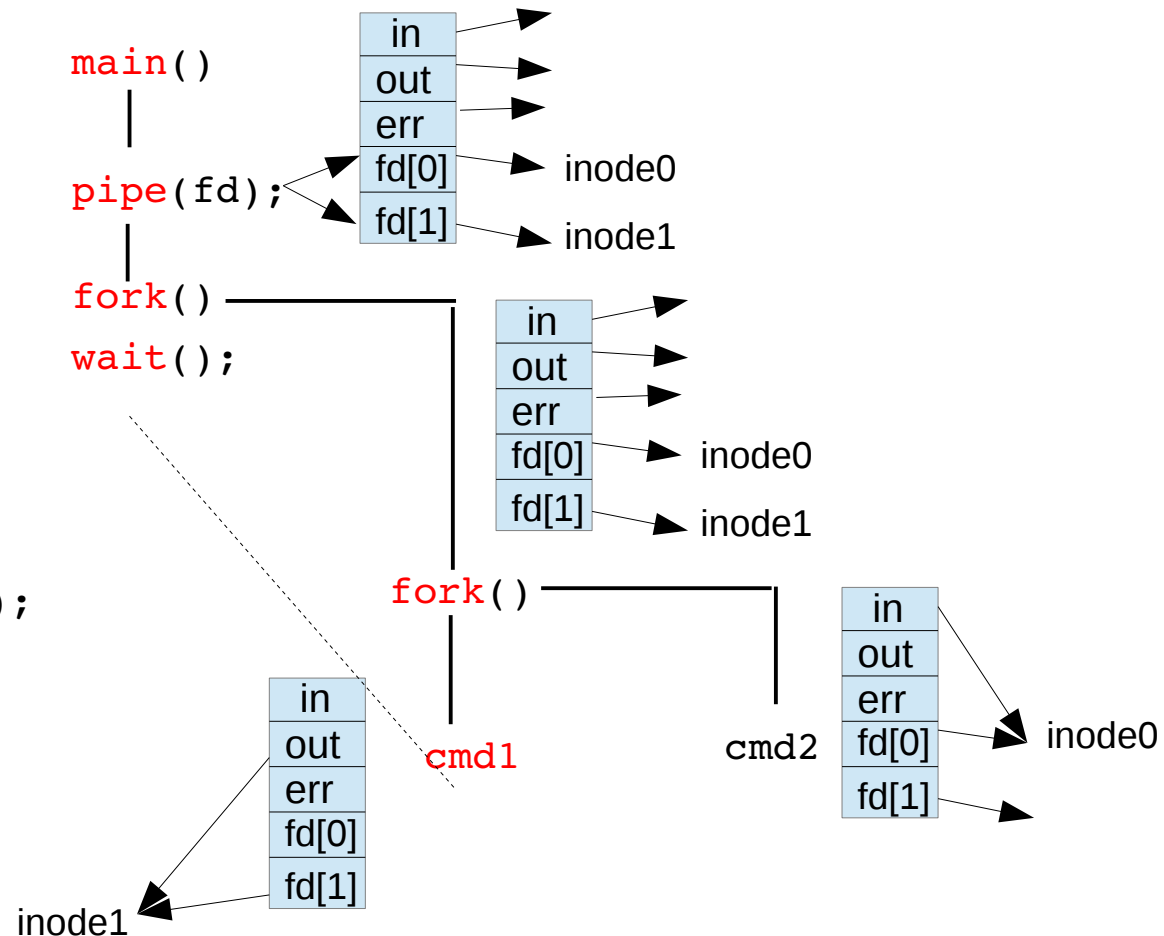


Architettura alternativa

```
#include <stdlib.h>
#include <stdio.h>
char **cmd1 = { "/bin/ps", "-e", 0 };
char **cmd2 = { "/usr/bin/wc", "-l", 0 };
int fd[2];
void runpipe()
```

```
{
    int pid;
    switch (pid = fork()) {
        case 0: dup2(fd[0], 0);
                close(fd[1]);
                execvp(cmd2[0], cmd2);
                perror(cmd2[0]);
        default: dup2(fd[1], 1);
                 close(fd[0]);
                 execvp(cmd1[0], cmd1);
                 perror(cmd1[0]);
        case -1: perror("fork"); exit(1);
    }
}
```

```
int main()
{
    int pid, status;
    pipe(fd);
    switch (pid = fork()) {
        case 0: runpipe();exit(0);
        default: wait();break;
        case -1: perror("fork"); exit(1);
    }
    exit(0);
}
```



Struttura del secondo modello di shell

```
char buf[MAXLINE]; char *cmd1; char *cmd2; char *arg1[MAX]; char *arg2[MAX];

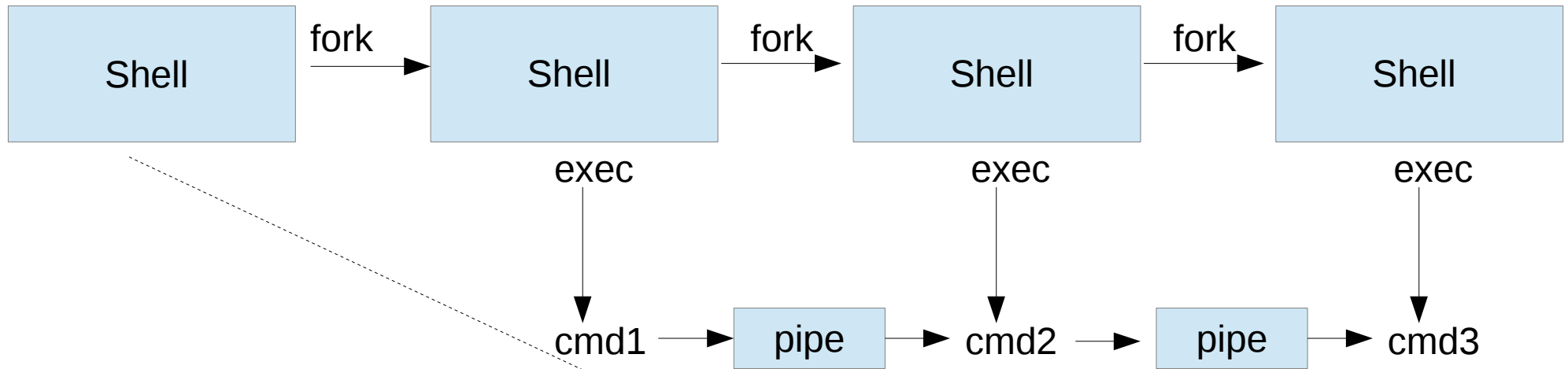
void main() {
    printf("%% "); /* prompt */

    while(fgets(buf, MAXLINE, stdin) != NULL) { /* ciclo di letture */

        Parsing(); /*esamina la stringa di ingresso cmd1 <opz> | cmd2 <opz>
                   e la divide in:   cmd1 cmd2 arg1 arg2 */

        pid = fork(); /* prima fork */
        if(pid == 0) { /* processo generato */
            pipe(pipeFd); /* creazione della pipe */
            if(fork() == 0) {
                close(pipeFd[0]); /* chiusura della pipe in lettura */
                dup2(pipeFd[1], STDOUT_FILENO); /* redirectione */
                execvp(cmd1, arg1); /* esecuzione del 1o comando */
            } else {
                close(pipeFd[1]); /* chiusura della pipe in scrittura */
                dup2(pipeFd[0], STDIN_FILENO); /* redirectionw */
                execvp(cmd2, arg2); /* esecuzione del 2o comando */
            }
        } else {
            waitpid(pid, &status, 0); /* attesa della terminazione */
            printf("%% "); /* prompt */
        }
    }
}
```

Architettura alternativa cmd1 | cmd2 | cmd3



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

FIFO: comunicazione client-server

```
// SERVER
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE    "MIA_FIFO"

int main(void)
{
    FILE *fp;
    char readbuf[80];

    /*crea la FIFO */
    umask(0);
    mknod(FIFO_FILE, S_IFIFO|0666, 0);

    while(1)
    {
        fp = fopen(FIFO_FILE, "r");
        fgets(readbuf, 80, fp);
        printf("messaggio ricevuto: %s\n", readbuf);
        fclose(fp);
    }

    return(0);
}
```

Client-server con FIFO

```
// CLIENT
#include <stdio.h>
#include <stdlib.h>

#define FIFO_FILE    "MIA_FIFO"

int main(int argc, char *argv[])
{
    FILE *fp;

    if ( argc != 2 ) {
        printf("USO: client [string]\n");
        exit(1);
    }

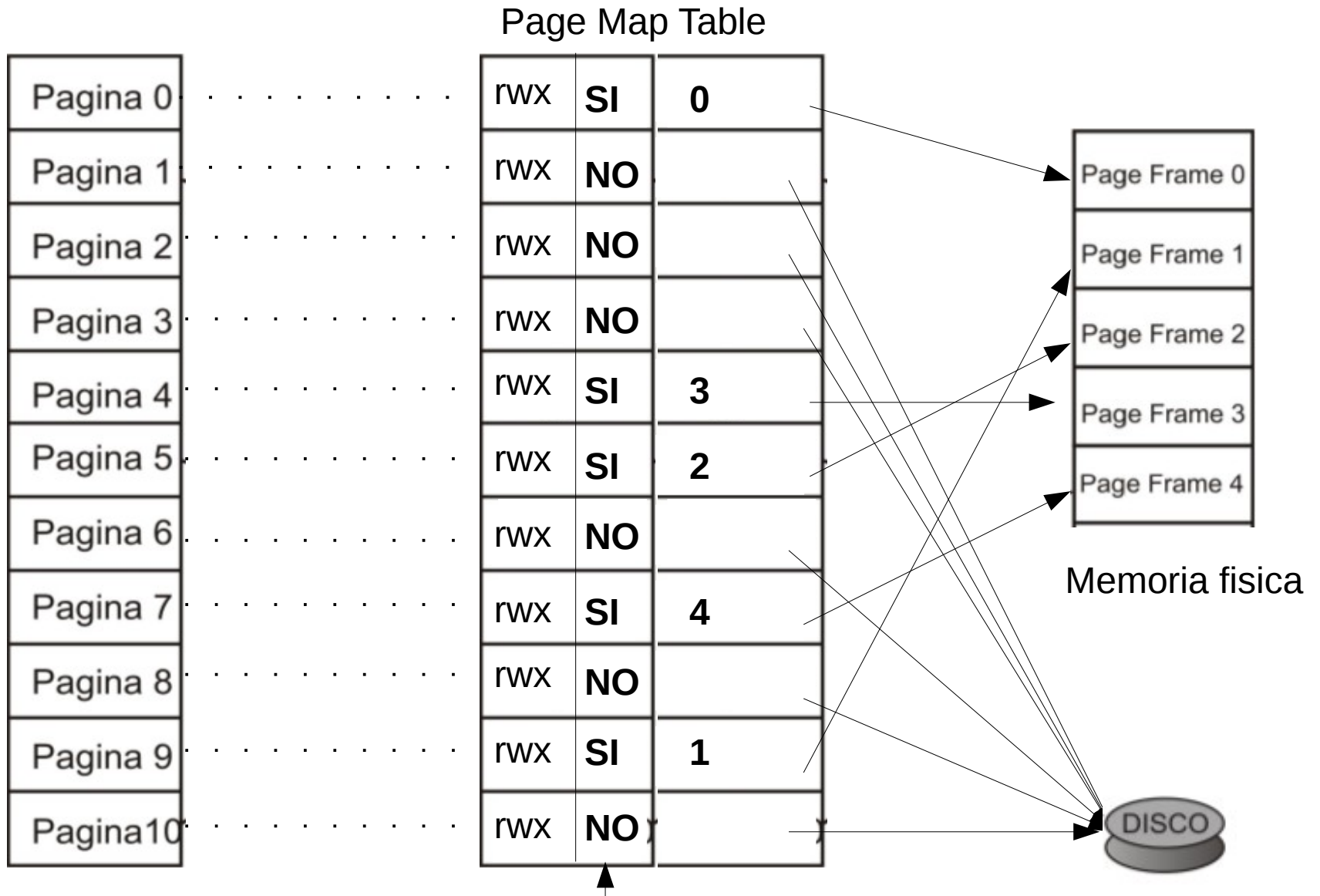
    if((fp = fopen(FIFO_FILE, "w")) == NULL) {
        perror("fopen");
        exit(1);
    }

    fputs(argv[1], fp);

    fclose(fp);
    return(0);
}
```

Shared memory: cenni di memoria virtuale

Tendente alla dimensione del disco:
molto molto maggiore della memoria fisica

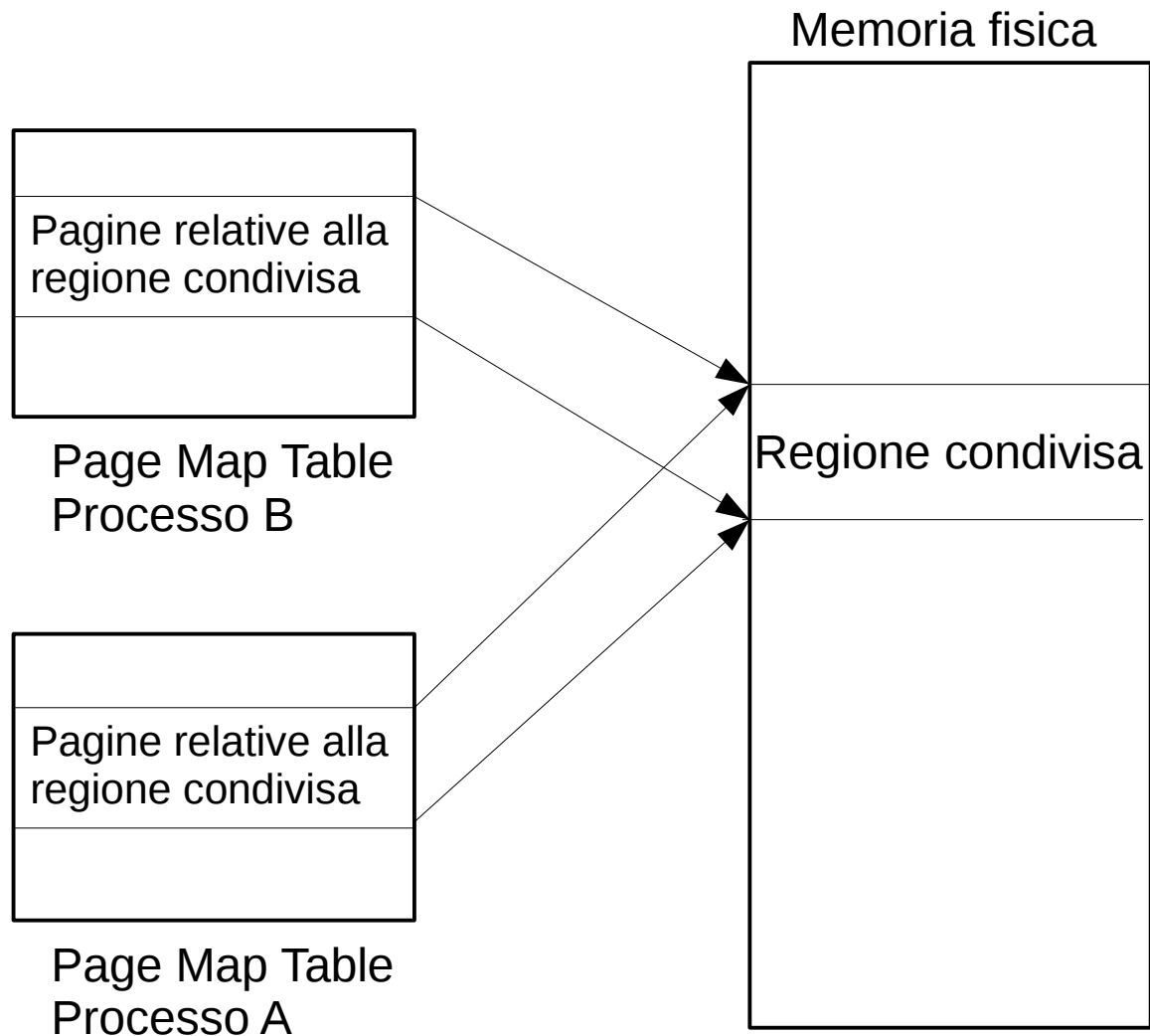


Esempio di spazio di indirizzamento
di un processo = spazio virtuale

E' caricata in
memoria fisica?

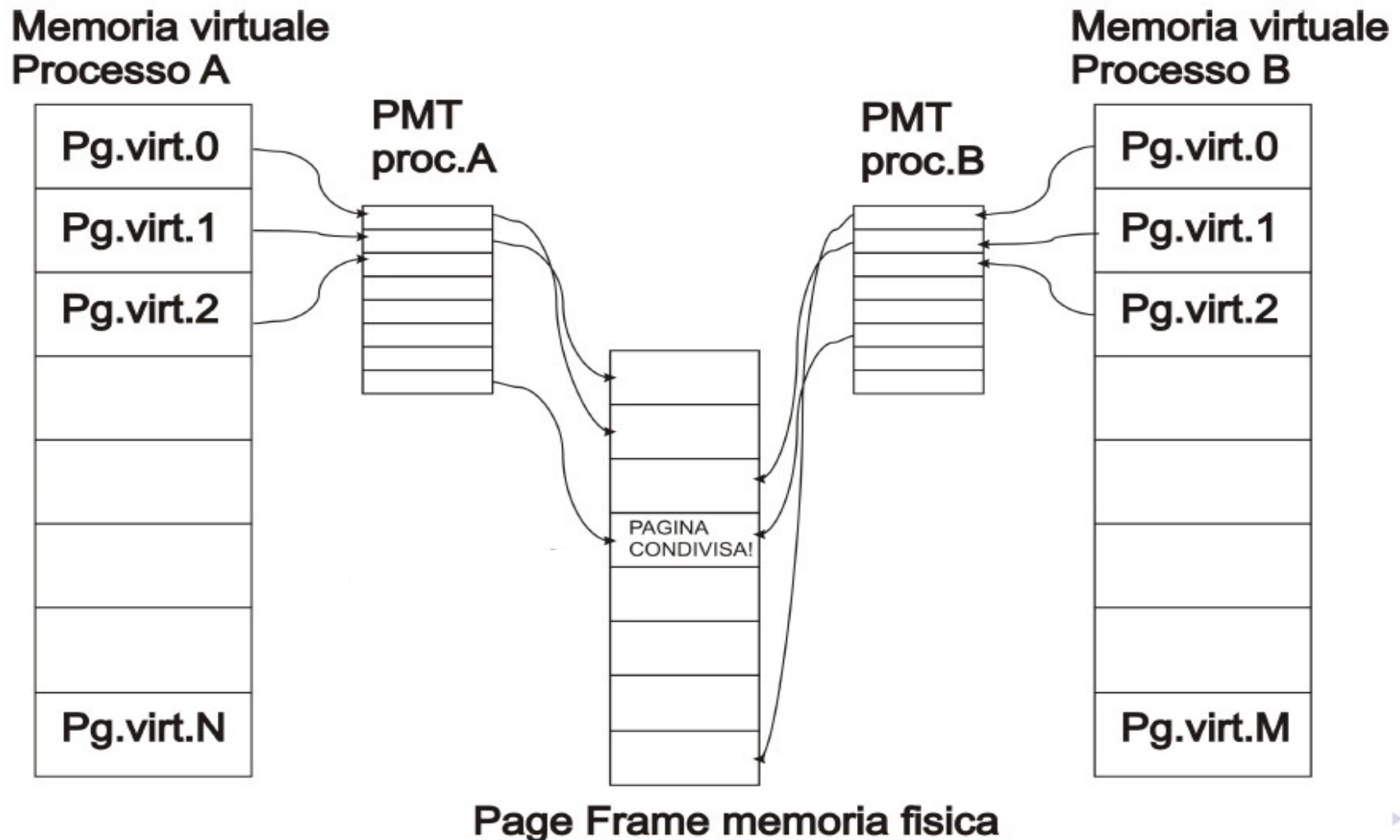
Shared memory

- E' una regione di memoria condivisa tra più processi



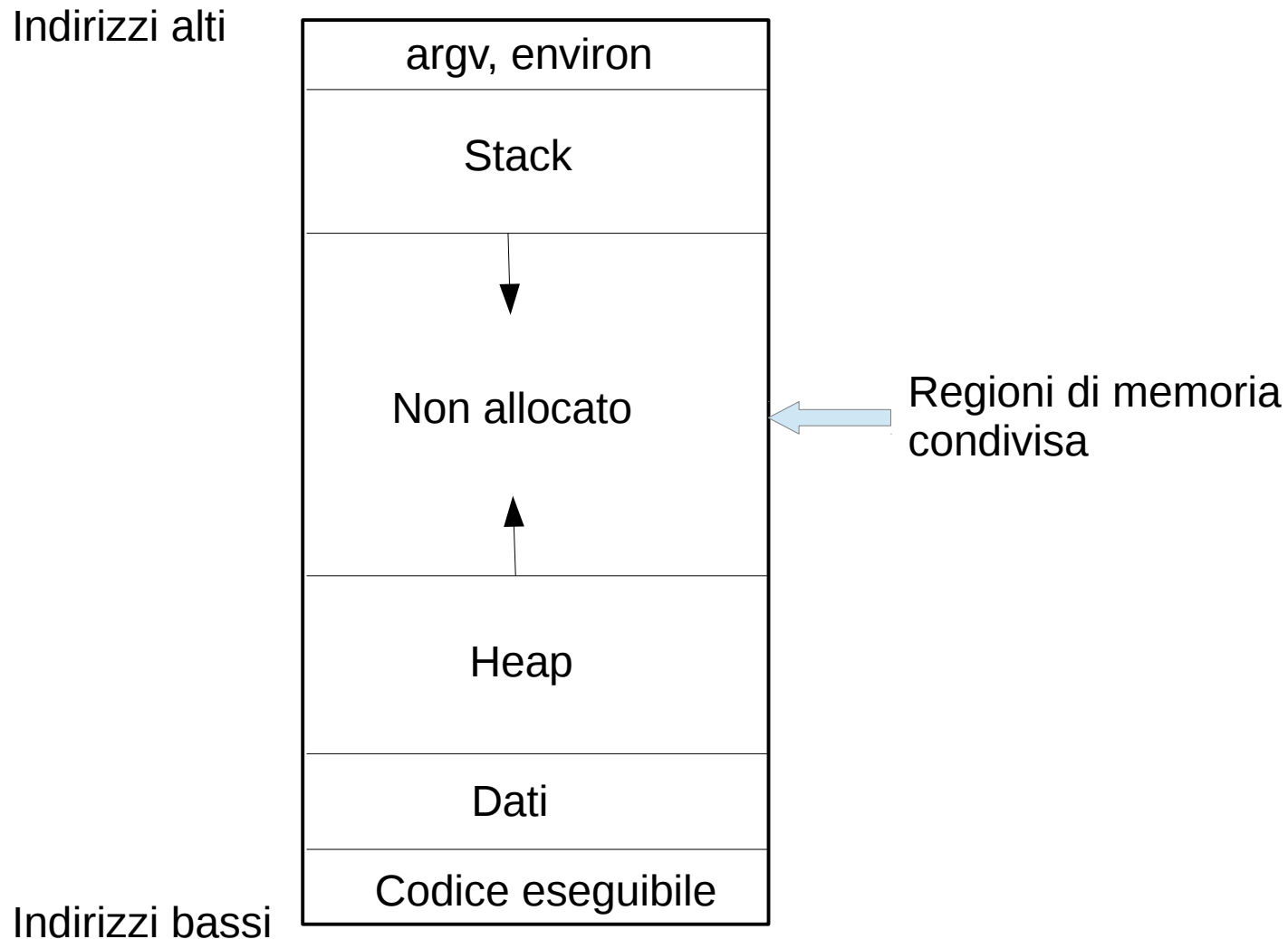
Shared memory

- Visione d'insieme



Spazio di indirizzamento di un processo in Linux

- E' uno spazio virtuale



Shared memory (Standard Posix)

- `shm_open()`: Crea una shared memory o si attacca ad una regione pre-esistente. Ritorna un descrittore numerico in UFDT.
- `shm_unlink()`: Cancella una shared memory descritta dal descrittore numerico. La regione rimane allocata finchè tutti i processi che la usano non terminano.
- `mmap()`: Mappa la shared memory nella memoria del processo. Ritorna un puntatore alla memoria
- `munmap()`: L'inverso di `mmap()`.
- `msync()`: Sincronizza la shared memory con un file.

Shared memory (Standard Posix)

- Apertura di una memoria condivisa
- Creazione di una memoria condivisa

```
#include <sys/mman.h>
#include <sys/stat.h>          /* mnemonici in mode */
#include <fcntl.h>             /* mnemonici in oflag */
int shm_open(const char *name, int oflag, mode_t mode);
```

- Mnemonici di oflag:

O_RDONLY, O_RDWR, O_CREAT, O_EXCL, O_TRUNC

Se entrambi settati, shm_open
non funziona se la regione esiste già

S_IRUSR | S_IWUSR
→ permesso di lettura e
scrittura

Se la regione esiste
già, la tronca a zero
byte

- Rimozione di una memoria condivisa

```
int shm_unlink(const char *name);
```

Shared memory (Standard Posix)

- Mappaggio della memoria condivisa nella memoria del processo

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

```
void *mmap(void *addr, size_t length, int prot, int flags,  
           int fd, off_t offset);
```

- Se `addr` è zero, il kernel sceglie l'indirizzo
- La lunghezza in byte della memoria condivisa è `length`
- L'argomento `prot` può essere:
 - `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`
cioè la pagina può essere letta, scritta, eseguita, non può essere acceduta
Normalmente `prot= PROT_READ|PROT_WRITE`
- `flags` determina se i cambiamenti sono visibili o meno ad altri processi
 - `MAP_ANONYMOUS`, `MAP_SHARED`, `MAP_PRIVATE`
Normalmente `flags=MAP_ANONYMOUS | MAP_SHARED`
- `fd`, `offset` sono usati per mappare la memoria su un file (vedi `msync`)

Shared memory (Standard Posix)

- Rimuove la mappatura e rende disponibile la memoria all'indirizzo addr

```
int munmap(void *addr, size_t length);
```

- La mappatura è rimossa anche quando il processo termina
- Sincronizzazione con il file indicato in mmap

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

```
int msync(void *addr, size_t length, int flags);
```

- Attenzione: queste chiamate di sistema richiedono la Realtime Extensions Library di POSIX.1b, cioè la librt. Per cui la compilazione è:

```
$gcc source.c -o source -lrt
```

Primo esempio d'uso della shared memory

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
#define SIZE sizeof(int)
int main (void)
{
    pid_t pid;
    float* shared_memory, scritto, letto ;
    int i_letto, i_scritto;

    shared_memory=mmap(0, SIZE, PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_SHARED, -1, 0);
    pid = fork();
    if (pid==0){
        sleep(1); letto = *shared_memory; printf ("\tprocesso generato. Leggo = %f\n", letto);
        sleep(1); i_letto = *(int*)shared_memory;
        printf ("\tprocesso generato. Leggo = %d\n", i_letto);
        sleep(1); letto = *shared_memory; printf ("\tprocesso generato. Leggo = %f\n", letto);
    }
    else {
        scritto=20; *shared_memory = scritto ; printf ("Processo padre. Scrivo = %f\n", scritto );
        sleep(1);
        i_scritto=47; *(int *)shared_memory = i_scritto ;
        printf ("Processo padre. Scrivo = %d\n", i_scritto );
        sleep(1);
        scritto=13; *shared_memory = scritto ; printf ("Processo padre. Scrivo = %f\n", scritto );
        sleep(1);
        wait(pid);
    }
    exit (0);
}
```

Scrittura di numeri interi e float con
delay per sincronizzazione

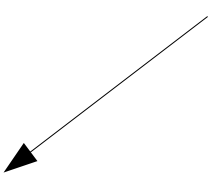
Condivisione di un array in shared memory

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#define SIZE sizeof(int)
```

```
int main (void)
```

```
{    int i, pid;
    float* shared_memory;
    float* p;
    float* q;
```

Essendo i processi imparentati non serve agganciare la regione condivisa con shm_open



```
    shared_memory=mmap(0, 5*SIZE, PROT_READ | PROT_WRITE,
                        MAP_ANONYMOUS | MAP_SHARED, -1, 0);
```

```
    pid = fork();
```

```
    if (pid==0){
```

```
        sleep(1); /* aspetta che il processo padre scriva gli elementi */
```

```
        p=shared_memory;
```

```
        printf("lettura:\n");
```

```
        for(i=0; i<5; i++)
```

```
            printf ("\tarray[i] = %f\n", *p++);
```

```
    }
```

```
    else {
```

```
        q=shared_memory;
```

```
        for(i=0; i<5; i++)
```

```
            *q++ = i*10;
```

```
        printf("scritto l'array!\n");
```

```
        wait(pid);
```

```
    }
```

```
    exit (0);
```

```
}
```

Condivisione di un array in shared memory tra processi diversi

Processo scrittore

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>      /* mnemonici in mode */
#include <fcntl.h>         /* mnemonici in oflag */
#define SIZE sizeof(int)

int main (void)
{
    int fd,i, pid,size=5*sizeof(float);
    float* shared_memory;
    float* p;
    float* q;

    fd = shm_open("mymem", O_CREAT | O_EXCL | O_RDWR, S_IRUSR | S_IWUSR);

    ftruncate(fd,size);

    shared_memory = mmap(NULL ,size,PROT_READ | PROT_WRITE, MAP_SHARED ,fd, 0);

    sleep(1);

    q=shared_memory;
    for(i=0; i<5; i++)
        *q++ = i*10;
    printf("scritto l'array!\n");
}
```

Condivisione di un array in shared memory tra processi diversi

Processo lettore

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h> /* mnemonici in mode */
#include <fcntl.h> /* mnemonici in oflag */

int main (void)
{
    int fd,i, pid,size=5*sizeof(float);
    float* shared_memory;
    float* p;
    float* q;

    fd = shm_open("mymem", O_EXCL | O_RDWR, S_IRUSR | S_IWUSR);
    shared_memory = mmap(NULL ,size,PROT_READ | PROT_WRITE, MAP_SHARED ,fd, 0);

    p=shared_memory;

    printf("lettura array:\n");
    for(i=0; i<5; i++)
        printf ("\tarray[i] = %f\n", *p++);
    exit (0);
}
```