

Pipe con nome

[Pagina intenzionalmente vuota]

Pipe tradizionali

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

Crea una *pipe* e restituisce due descrittori di file:

- `fildes[0]` aperto in lettura e
- `fildes[1]` aperto in scrittura

Semantica:

- accesso FIFO via `read(fildes[0], ...)` ai dati scritti con `write(fildes[1], ...)`
- data unit arbitrario per ciascuna operazione
- data boundaries non preservate, i.e. `read` vs. `write` non atomiche
- atomicità (non interleaving) di `write` concorrenti
- `read` bloccante solo se pipe vuota
- `write`
 - bloccante con `>PIPE_BUF` (in `limits.h`) bytes in pipe;
 - causa invio `SIGPIPE` al processo se la pipe non ha lettori

Limitazioni:

- supporta comunicazione solo tra parenti
- lifetime pipe = lifetime processi interessati

Esempio pipe: ls | sort

```
/* pipeord1.c */

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

pid_t pid;
int pipefd[2];

int main()
{
    pipe(pipefd);

    if ((pid = fork()) == (pid_t)0) { // processo figlio
        close(1); // close stdout (assumed open)
        dup(pipefd[1]); // dups arg to min free descriptor
        close(pipefd[0]);
        execlp("ls", "ls", "-lt", (char *)0);
    } else if (pid > (pid_t)0) { // processo padre
        close(0); // close stdin (assumed open)
        dup(pipefd[0]);
        close(pipefd[1]);
        execlp("sort", "sort", "-r", (char *)0);
    }
    return(0);
}
```

Esempio pipe: read bloccante

```
/* pipeord2.c */

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

#define BUFLLEN 8192
#define NW

pid_t pid;
int pipefd[2];

int main()
{
    char outbuf[] = "ciao*fine";
    char inbuf[BUFLLEN];
    int nr;

    pipe(pipefd);

    if ((pid = fork()) > (pid_t)0) { // father/reader
        printf("waiting to read from pipe\n");
        nr = read(pipefd[0], inbuf, BUFLLEN);
        inbuf[nr] = '\0';
        printf("read %d bytes: %s\n", nr, inbuf);
        nr = read(pipefd[0], inbuf, BUFLLEN);
        inbuf[nr] = '\0';
        printf("read %d bytes: %s\n", nr, inbuf);
        close(pipefd[0]);
    } else if (pid == (pid_t)0) { // child-writer
        sleep(3);
        write(pipefd[1], outbuf, 4);
        sleep(4);
    }
    // write(pipefd[1], outbuf+5, 4);
    close(pipefd[1]);
}
return(0);
}
```

Pipe con nome: nozione e API

Concetto di named pipe

Differenze rispetto a pipe tradizionali:

- una *named pipe* è un file speciale nel FS (visibile con `ls`)
N.B.: l'informazione è passata via strutture dati del kernel, non FS; il file speciale serve solo per *referirsi* alla pipe
- quindi la named pipe è *permanente*
- la named pipe può collegare processi non discendenti di quello che l'ha creata;
per fare riferimento alla pipe di nome *p* basta aprire *p*

Proprietà specifiche di sincronizzazione:

- una named pipe si può aprire o no in modo `O_NONBLOCK`
- aprire in `O_RDONLY | O_NONBLOCK` non blocca e ha sempre successo
- aprire in `O_RDONLY` senza `O_NONBLOCK` blocca se nessuno ha aperto in scrittura
- aprire in `O_WRONLY | O_NONBLOCK` non blocca e:
 - ha successo se la pipe è già aperta in lettura
 - fallisce (errore `ENXIO`) altrimenti
- aprire in `O_WRONLY` senza `O_NONBLOCK` blocca se nessuno ha aperto in lettura
- in Linux, aprire in `O_RDWR` non blocca mai, indipendentemente da `O_NONBLOCK` (effetto indefinito in POSIX)
- come per le pipe, se si perviene a scrivere su una named pipe che nessuno ha aperto in lettura, si causa invio di `SIGPIPE`

Named pipe: creazione, apertura, chiusura

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- `dev` è ignorato (tranne che per i file-device)
- `mode` è l'or di `S_IFIFO` e dei permessi
N.B.: come per i file, il permesso sarà `mode & ~umask` (ogni bit 1 di `umask` toglie un permesso)
p. es. `umask 022=000 010 010`, `mode 0666=rw-rw-rw-` implicano file creati con permesso: `0666 & ~022 = 0644 = rw-r--r--`

`mkfifo(3)` è per tutti; `mknod` solo per root, se `mode` non include `S_IFIFO`:

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

qui `S_IFIFO` in `mode` è inutile.

`open(2)`, con le named pipe, deve seguire la creazione:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
```

flag più usati: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_NONBLOCK`

`open` restituisce un descrittore di file o un errore (-1).

```
#include <unistd.h>
int close(int fd);
```

Named pipe: lettura e scrittura

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

- legge fino a `count` byte
- il valore restituito è il n. di byte letti o `-1` in caso di errore
- leggere dopo apertura con flag `O_NONBLOCK`, in mancanza di dati, causa `errno=EAGAIN`
- leggere da una pipe bloccante,
 - blocca se la pipe:
 - * non è stata ancora aperta in scrittura
 - * o è stata aperta, ma non contiene (più) dati
 - legge 0 bytes **se la pipe era aperta in scrittura e viene chiusa**
NB: questo **non** va interpretato come messaggio “vuoto” inviato da `write(p, buf, 0)` (questa `write` non lascia traccia sulla pipe `p`), **ma come equivalente a leggere eof da un file.**

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

- scrive fino a `count` byte
- il valore restituito è il n. di byte scritti o `-1` in caso di errore
- scrivere su pipe non aperta in lettura causa invio di `SIGPIPE`; se questo segnale è ignorato o gestito, `write` torna con errore `errno=EPIPE`
- `write` torna con errore `errno=EAGAIN`, se la pipe non bloccante non ha spazio per scrivere i dati immediatamente

Named pipe e shell

```
mknod [-m=mode]... NAME p
```

(b o c anziché p crea device, ma solo per root).

```
mkfifo [-m=mode]... NAME
```

permesso a ogni utente.

Le named pipe possono essere oggetto di ridirezione:

```
gp: ~/os2 $ mkfifo tmp
gp: ~/os2 $ /bin/ls -l tmp
prw-----  1 gp  gp           0 Jan 11 11:13 tmp
gp: ~/os2 $ /bin/ls -lt *aux >tmp &
[1] 950
gp: ~/os2 $ sort <tmp
cdk2.aux
lab.aux
pipes.aux
[1]+  Done                    /bin/ls -lt *aux >tmp
gp: ~/os2 $ rm tmp
```

Pipe con nome: esempi

Semplici client-server con named pipe

Server 1 (messaggi di lunghezza fissa)

Il client invia messaggi di lunghezza fissa ad un server, che:

1. crea pipe con nome (argomento)
2. la apre in lettura
3. in un ciclo, legge un messaggio dalla pipe e lo scrive
4. esce se il messaggio inizia con End.

```
/* server1.c */

/* accetta e scrive messaggi, finché non riceve End. */
/* NB: funziona solo con messaggi di lunghezza costante e *
 * finche' un processo tiene aperta la pipe in scrittura */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#define MSGLEN 8

#define PERMS 0644

int main(int argc, char * argv[])
{
    int npip, res;
    char buffer[MSGLEN];

    if( (res = mknod(argv[1], S_IFIFO|PERMS, 0)) == -1 )
        {perror("creating pipe"); exit(-1); }
    if( (npip = open(argv[1],O_RDONLY)) == -1 )
        {perror("opening pipe"); exit(-2); }
    while (1) { // ciclo di attesa server
        if( (res = read(npip, buffer, MSGLEN)) == -1 ) {
            perror("\nreading pipe");
            close(npip); exit(-3);
        }
        write(1,buffer,MSGLEN); // scrive su stdout
        if(strncmp(buffer,"End",3) == 0) // condizione
            break; // di uscita
    }
    close(npip); unlink(argv[1]);
    exit(0);
}
```

Client 1

Schema client:

1. apre pipe (argomento) in lettura
2. invia 5 messaggi di lunghezza fissa ("aaaaaaa", "bbbbbbb", ...)
3. invia messaggio "End...".

```
/* client1.c */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

# define MSGLEN 8
char buffer[MSGLEN];

int main(int argc, char * argv[])
{
    int npip, res, n_msg;

    npip = open(argv[1],O_WRONLY);
    if (npip == -1)
        {perror("opening pipe"); exit(-1); }
    for (n_msg = 0; n_msg < 5; n_msg++) { // invia 5 messaggi
        char *p, c;
        c = 'a' + n_msg;
        for (p = buffer; p < buffer+MSGLEN; p++)
            *p = c;
        p[-1] = '\n';
        if ( (res = write(npip, buffer, MSGLEN)) == -1 ) {
            close(npip);
            perror("writing to pipe"); exit(-2);
        }
    }
    strcpy(buffer, "End...\n"); // messaggio fi nale
    if( (res = write(npip, buffer, MSGLEN)) == -1 ) {
        close(npip);
        perror("writing to pipe"); exit(-3);
    }
    close(npip);
    exit(0);
}
```

Server 2: messaggi di lunghezza variabile

- client invia messaggi di lunghezza variabile (2o arg)
- server accetta e scrive messaggi, finché `read` non legge 'End...' oppure 0 bytes (accade se la pipe non è aperta in write).

```
/* server2.c */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#define MSGLEN 8

int main(int argc, char * argv[])
{
    int npip, res;
    char buffer[MSGLEN], *p;

    unlink(argv[1]);
    if( (res = mknod(argv[1],S_IFIFO|0622,0)) == -1 ) {
        perror("creating pipe"); // diagnostico mnemonico
        exit(-1);
    }
    if( (npip = open(argv[1],O_RDONLY)) == -1 )
        {perror("opening pipe"); exit(-2); }
    while(1) { // ciclo di attesa server
        // inizializza buffer (non in server1)
        for (p = buffer; p < buffer+MSGLEN; p++) *p = '\0';
        // legge messaggio
        if( (res = read(npip, buffer, MSGLEN)) == -1 ) {
            close(npip);
            perror("reading from pipe"); exit(-3);
        }
        printf("ricevuto: <%s> (%d byte)\n", // server1 ha write(1,...
            buffer, res);
        if (res == 0 // pipe non aperta in write (non in server1)
            || strcmp(buffer, "End", 3) == 0)
            break; // uscita
    }
    close(npip); unlink(argv[1]);
    exit(0);
}
```

Client 2

Schema client:

1. apre pipe (argomento 1) in lettura
2. invia un messaggio (argomento 2)

```
/* client2.c */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

int main(int argc, char * argv[])
{
    int npip, res;

    if((npip = open(argv[1],O_WRONLY)) == -1)
        {perror("opening pipe"); exit(-2); }
    if( (res = write(npip, argv[2], strlen(argv[2]))) == -1 ) {
        close(npip);
        perror("writing to pipe"); exit(-3);
    }
    printf("Data sent:  <%s>\n", argv[2]);
    close(npip);
    exit(0);
}
```

Ma il server non legge messaggi consecutivi, come `server1`:

```
gp: ~/os2.gp2 $ ./server2 ciccio &
[11] 1836
gp: ~/os2.gp2 $ ./client2 ciccio ciao
Scritto:  <ciao>
ricevuto: <ciao> (4 byte)
ricevuto: <> (0 byte)
```

- nel loop del server, la 2a `read` non trova messaggi, ma non si blocca e dà 0 byte \Rightarrow il server esce!
- questo *perché la pipe non è più aperta in scrittura!* (il client ha chiuso `npip` e non ci sono in giro descrittori aperti in scrittura)

Mantenere la pipe aperta in lettura

Per far funzionare `client2` gli si affianca `clnkeepw`:

1. `clnkeepw` viene lasciato in attesa di un messaggio da *stdin*; nel frattempo, non scrive sulla pipe, né la chiude;
2. così la 2a `read` di `client2` trova un descrittore in scrittura per la pipe e si blocca fino alla ricezione di un messaggio "vero"

N.B.: anziché `clnkeepw` va bene ogni accorgimento che assicuri (2)

```
/* clnkeepw.c */
/* tiene pipe aperta in wr, evita che server2 termini su rd */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

#define MAXBUF 128
char buffer[MAXBUF];

int main(int argc, char * argv[])
{
    int npip, res;

    npip = open(argv[1], O_WRONLY);
    if (npip == -1) {
        perror("opening"); exit(-2);
    }
    printf("Scrivi messaggio (max %d char):\n", MAXBUF);
    gets(buffer); // gets pericolosa e sconsigliata (vedi man)!
    if ( (res = write(npip, buffer, strlen(buffer))) == -1 ) {
        perror("writing to pipe"); close(npip); exit(-3);
    }
    printf("Scritto: <%s>\n", buffer);
    close(npip);
    exit(0);
}
```

Dopo qualche successo con `client2`, `ps -al` mostra che `clnkeepw` è in attesa di input da tastiera (`read_chan`).

Mantenere la pipe aperta in lettura (+)

L'effetto di `clnkeepw` viene comunque meno: quando esso termina o quando `server2` termina e se ne avvia una nuova copia.

In questi casi `server2` termina "male", col `break` su `res==0`.

```
/* server2.c */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#define MSGLEN 8

int main(int argc, char * argv[])
{
    int npip, res;
    char buffer[MSGLEN], *p;

    unlink(argv[1]);
    if( (res = mknod(argv[1],S_IFIFO|0622,0)) == -1 ) {
        perror("creating pipe"); // diagnostico mnemonico
        exit(-1);
    }
    if( (npip = open(argv[1],O_RDONLY)) == -1 )
        {perror("opening pipe"); exit(-2); }
    while(1) { // ciclo di attesa server
        // inizializza buffer (non in server1)
        for (p = buffer; p < buffer+MSGLEN; p++) *p = '\0';
        // legge messaggio
        if( (res = read(npip, buffer, MSGLEN)) == -1 ) {
            close(npip);
            perror("reading from pipe"); exit(-3);
        }
        printf("ricevuto: <%s> (%d byte)\n", // server1 ha write(1,...
            buffer, res);
        if (res == 0 // pipe non aperta in write (non in server1)
            || strcmp(buffer, "End", 3) == 0)
            break; // uscita
    }
    close(npip); unlink(argv[1]);
    exit(0);
}
```

Client-server: osservazione su server 2

D'altra parte, in `server2` il `break` è indispensabile.

In mancanza, `server2` mostrerebbe un'apparente "raffica" di messaggi di 0 bytes (sono solo `read` senza descrittori aperti in scrittura).

```
/* server2.c */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#define MSGLEN 8

int main(int argc, char * argv[])
{
    int npip, res;
    char buffer[MSGLEN], *p;

    unlink(argv[1]);
    if( (res = mknod(argv[1],S_IFIFO|0622,0)) == -1 ) {
        perror("creating pipe"); // diagnostico mnemonico
        exit(-1);
    }
    if( (npip = open(argv[1],O_RDONLY)) == -1 )
        {perror("opening pipe"); exit(-2); }
    while(1) { // ciclo di attesa server
        // inizializza buffer (non in server1)
        for (p = buffer; p < buffer+MSGLEN; p++) *p = '\0';
        // legge messaggio
        if( (res = read(npip, buffer, MSGLEN)) == -1 ) {
            close(npip);
            perror("reading from pipe"); exit(-3);
        }
        printf("ricevuto: <%s> (%d byte)\n", // server1 ha write(1,...
            buffer, res);
        if (res == 0 // pipe non aperta in write (non in server1)
            || strcmp(buffer, "End", 3) == 0)
            break; // uscita
    }
    close(npip); unlink(argv[1]);
    exit(0);
}
```

Server 3

Un'altra soluzione è che anche il server apra la pipe in scrittura, così la 2a read del ciclo si bloccherà (fino a un 2o messaggio) ...

```
/* server3.c */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#define MSGLEN 8

int main(int argc, char * argv[])
{
    int npip, npip2, res;
    char buffer[MSGLEN], *p;

    unlink(argv[1]);
    if( (res = mknod(argv[1],S_IFIFO|0622,0)) == -1 ) {
        perror("creating pipe"); // diagnostico mnemonico
        exit(-1);
    }
    if( (npip = open(argv[1],O_RDONLY)) == -1 )
        {perror("opening pipe"); exit(-2); }
    npip2 = open(argv[1],O_WRONLY); // non è in server2

    while(1) { // ciclo di attesa server
        // inizializza buffer (non in server1)
        for (p = buffer; p < buffer+MSGLEN; p++) *p = '\0';
        // legge messaggio
        if( (res = read(npip, buffer, MSGLEN)) == -1 ) {
            close(npip);
            perror("reading from pipe"); exit(-3);
        }
        printf("ricevuto: <%s> (%d byte)\n", // server1 ha write(1,...
            buffer, res);
        if (res == 0 || // pipe non aperta in write (non in server1)
            strncmp(buffer, "End", 3) == 0)
            break; // uscita
    }
    close(npip); close(npip2); unlink(argv[1]);
    exit(0);
}
```

Dimensione massima di una pipe

Si può determinare a run-time attraverso la costante di sistema `PIPE_BUF`.

```
/* maxpip.c */

#include <stdio.h>
#include <unistd.h>

#include <limits.h> // per PIPE_BUF vedi /usr/src/linux/include/linux/limits.h

int main(int argc, char * argv[])
{
    printf("dim max pipe per sistema = %d bytes\n",
          PIPE_BUF);
    exit(0);
}
```

Per renderci conto dell'effetto della dimensione finita delle pipe, implementiamo:

- *scrittore* su pipe (`argv[1]`) da file (`argv[2]`)
- *lettore* da pipe (`argv[1]`) su stdout

Analisi temporizzazioni critiche:

- scrittore esegue write cicliche, che si bloccano se trovano la pipe piena
- lettore rallentato con `sleep` per `argv[2]` sec, tarda a smaltire i dati
- un blocco di durata eccessiva causerà un *timeout*, come nell'esempio a pag. 21

Letture

```
/* reader.c */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <stdlib.h>

#define MAXBUF 1024

int main(int argc, char * argv[])
{
    char buffer[MAXBUF];
    int piped, retcode;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s pipe delay\n", argv[0]);
        exit(-1);
    }

    printf("At t %ld opening pipe for read\n", time(NULL));
    piped = open(argv[1], O_RDONLY);
    if(piped == -1)
        {perror("opening pipe"); exit(-2); }
    printf("opened (%ld)\n",time(NULL));

    do {
        sleep(atoi(argv[2]));
        printf("reading from pipe at time %ld\n", time(NULL));
        retcode = read(piped,buffer,MAXBUF);
        if (retcode == -1) {
            perror("reading"); close(piped); exit(-3);
        } else {
            if(retcode == 0) {
                printf("Eof on pipe t. %ld\n", time(NULL));
                break;
            }
            printf("read (at t %ld) from pipe %d bytes:\n",
                time(NULL), retcode);
            /* write(1,buffer,retcode); */ // mostra i byte letti
        }
    } while (retcode != -1);

    close(piped);
    exit(0);
}
```

Scrittore

```
/* writer.c */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <errno.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>

#define MAXBUF 1024

void allarme(int);

char nomepipe[MAXBUF];

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

    if (argc != 4) {
        fprintf(stderr, "Usage: %s srcfile pipe timeout\n",
                argv[0]);
        exit(-1);
    }

    if ((fp = fopen(argv[1], "r")) == NULL) {
        perror("opening file");
        exit(-2);
    }

    strcpy(nomepipe, argv[2]);
    printf("At t %ld opening pipe for write\n", time(NULL) );
    if ((piped = open(nomepipe, O_WRONLY)) == -1) {
        perror("opening pipe");
        exit(-3);
    }
    printf("opened (%ld)\n", time(NULL));

/* writer.c ... */
```

```

/* writer.c (cont.) */

signal(SIGALRM, allarme); // installa handler
while(fgets(buffer,MAXBUF,fp)) {
    int retcode;
    do {
        alarm(atoi(argv[3])); // dopo timeout inviami SIGALRM
        retcode =
            write(piped, buffer, strlen(buffer)+1);
    } while (retcode == -1 && // se write e' interrotta da segnale
            errno == EINTR); // riprende il ciclo alarm-write
            // cioe' riprova a scrivere

    if(retcode == -1) {
        char line[MAXBUF]; // uscita per altro errore su write
        sprintf(line, "writing to pipe at %ld",time(NULL));
        perror(line);
        close(piped);
        fclose(fp);
        exit(-4);
    }
    // riprende il while
}

close(piped);
fclose(fp);
exit(0);
}

void allarme(int signo) {

    char line[MAXBUF];

    printf("caught SIGALARM at %ld\n", time(NULL));
    sprintf(line,"ls -l %s", nomepipe);
    system(line);
}

```