

PIPE

Introduzione

Si è visto come i **processi** possano utilizzare i **segnali** per scambiarsi delle semplici informazioni.

Se due **processi** sono nella relazione **genitore/figlio**, condividono i **descrittori** dei file aperti, che possono essere usati **per comunicare**.

Esistono in Unix altri **strumenti** per la **comunicazione interprocesso** (IPC), che permettono lo scambio di informazioni tra processi su di uno **stesso calcolatore**.

Le pipe

Una **pipe** è creata mediante la chiamata alla funzione **pipe**.

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

Due descrittori di file sono ritornati mediante l'argomento ***filedes***:

***filedes*[0]** è aperto in lettura,

***filedes*[1]** è aperto in scrittura.

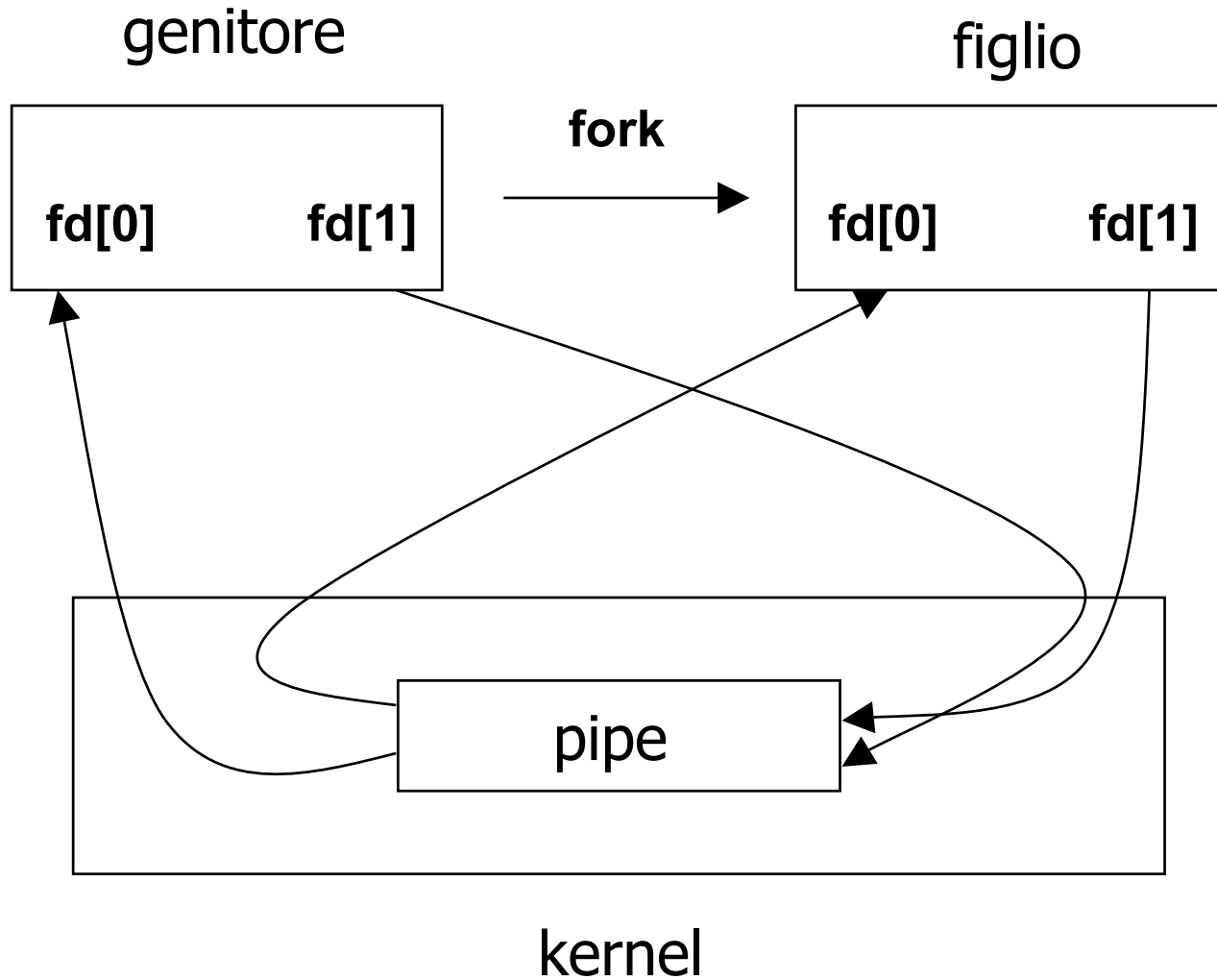
L'output di ***filedes*[1]** è l'input per ***filedes*[0]**.

Le pipe (cont.)

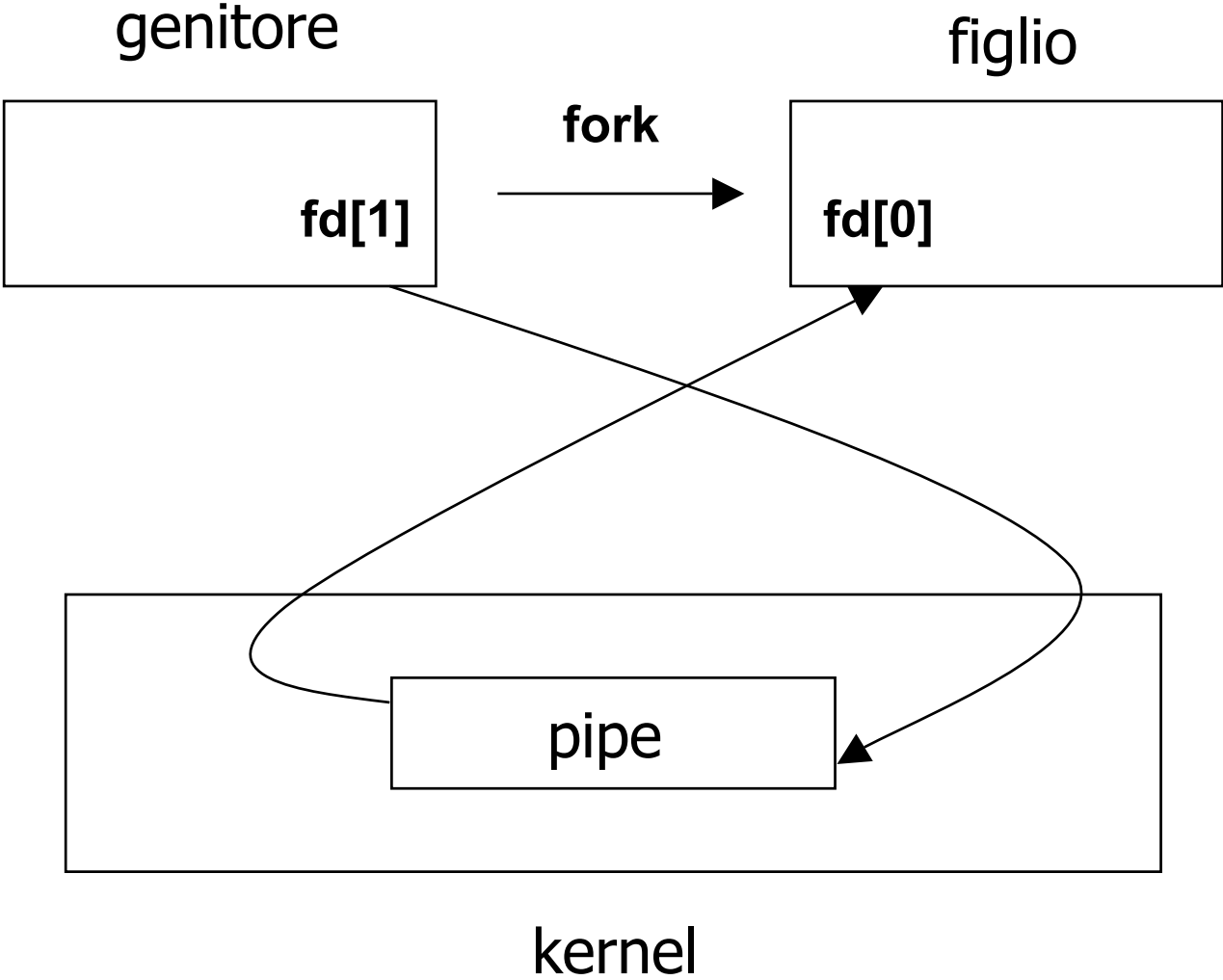
Di solito un processo che chiama **pipe** dopo chiama **fork**, creando un **canale** di **comunicazione** dal **genitore** verso il **figlio** o viceversa.

Se si vuole creare un canale dal padre verso il figlio, allora il genitore chiude il canale di lettura della pipe (***filedes*[0]**) e il figlio quello di scrittura (***filedes*[1]**).

Le pipe (cont.)



Le pipe (cont.)



Le pipe (cont.)

Le **pipe** rappresentano la forma più vecchia di **IPC** e sono supportate da tutte le implementazioni.

Le pipe hanno due limitazioni:

- Esse sono *half-duplex*, cioè i dati fluiscono in un'unica direzione.
- Tale **comunicazione** può avvenire solo **tra processi** che abbiano in comune uno **stesso antenato**.

Di solito un processo crea una pipe, poi crea un figlio, e la pipe è utilizzata tra padre e figlio.

Le pipe (cont.)

Quando una delle estremità della pipe è chiusa, valgono le seguenti regole:

1. se si legge (**read**) da una pipe il cui canale di scrittura è stato chiuso, dopo che sono stati letti tutti i dati **read** ritorna 0 per indicare la fine del file;
2. se si scrive (**write**) su di una pipe il cui canale di lettura è stato chiuso, il segnale **SIGPIPE** viene generato. Se si ignora il segnale, o lo si intercetta, **write** ritorna con un errore e **errno** è **EPIPE**.

La grandezza del buffer di pipe nel kernel è **PIPE_BUF**. Se più processi scrivono su di una stessa pipe, le **write** di **PIPE_BUF** byte o vengono interamente eseguite o vanno in errore.

```
# include "ourhdr.h"
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) {      /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Ecco come si crea una pipe da genitore a figlio e si inviano dati.

```
# include <sys/wait.h>
# include "ourhdr.h"

#define DEF_PAGER "/bin/more" /* default pager */

int main(int argc, char *argv[ ])
{
    int    n, fd[2];
    pid_t  pid;
    char   line[MAXLINE], *pager, *argv0;
    FILE   *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ( (fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if (pipe(fd) < 0)    err_sys("pipe error");
    if ( (pid = fork()) < 0)    err_sys("fork error");
    else if (pid > 0) {      /* parent */
```

```
if ( (fp = fopen(argv[1], "r")) == NULL)
    err_sys("can't open %s", argv[1]);
```

```
if (pipe(fd) < 0)    err_sys("pipe error");
if ( (pid = fork()) < 0)    err_sys("fork error");
else if (pid > 0) {
```

```
    close(fd[0]);    /* close read end */
```

```
    /* parent copies argv[1] to pipe */
```

```
    while (fgets(line, MAXLINE, fp) != NULL) {
```

```
        n = strlen(line);
```

```
        if (write(fd[1], line, n) != n)
```

```
            err_sys("write error to pipe");
```

```
    }
```

```
    if (ferror(fp))    err_sys("fgets error");
```

```
    close(fd[1]);    /* close write end of pipe for reader */
```

```
    if (waitpid(pid, NULL, 0) < 0)    err_sys("waitpid error");
```

```
    exit(0);
```

```
} else {
```

```
    M. Guarracino - PIPE
    /* child */
```

```
if (waitpid(pid, NULL, 0) < 0)    err_sys("waitpid error");
exit(0);
```

```
} else {                          /* child */
    close(fd[1]); /* close write end */
    if (fd[0] != STDIN_FILENO) {
        if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        close(fd[0]); /* don't need this after dup2 */
    }

    /* get arguments for execl() */
    if ( (pager = getenv("PAGER")) == NULL)    pager = DEF_PAGER;
    if ( (argv0 = strrchr(pager, '/')) != NULL) argv0++;
    else    argv0 = pager; /* no slash in pager */

    if (execl(pager, argv0, (char *) 0) < 0)
        err_sys("execl error for %s", pager);
    }
}
```

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

static void charatime(char *);

int main(void)
{
    pid_t pid;

    TELL_WAIT();

    if ( (pid = fork()) < 0)    err_sys("fork error");
    else if (pid == 0) {
        charatime("output from child\n");
    } else {
        charatime("output from parent\n");
    }
    exit(0);
}
```

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

```
static void charatime(char *);
```

```
int main(void)
```

```
{
```

```
    pid_t  pid;
```

```
    TELL_WAIT();
```

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

```
    else if (pid == 0) { /* child */
```

```
        WAIT_PARENT(); /* parent goes first */
```

```
        charatime("output from child\n");
```

```
    } else { /*parent */
```

```
        charatime("output from parent\n");
```

```
        TELL_CHILD(pid);
```

```
    }
```

```
    exit(0);
```

```
}
```

```
static void charatime(char *str)
{
    char *ptr;
    int    c;

    setbuf(stdout, NULL);    /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
}
```

```
#include "ourhdr.h"
static int pfd1[2], pfd2[2];
void TELL_WAIT()
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("pipe error");
}
```

```
void TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1) err_sys("write error");
}
```

```
void WAIT_PARENT(void)
{
    char c;

    if (read(pfd1[0], &c, 1) != 1) err_sys("read error");
    if (c != 'p') err_quit("WAIT_PARENT: incorrect data");
}
```

```
void WAIT_PARENT(void)
```

```
{  
    char c;  
  
    if (read(pfd1[0], &c, 1) != 1) err_sys("read error");  
    if (c != 'p') err_quit("WAIT_PARENT: incorrect data");  
}
```

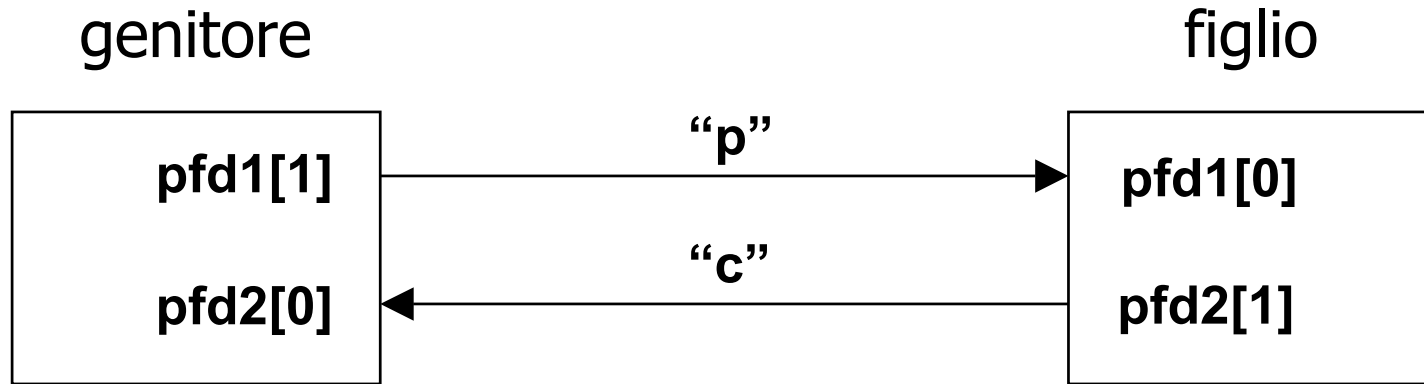
```
void TELL_CHILD(pid_t pid)
```

```
{  
    if (write(pfd1[1], "p", 1) != 1) err_sys("write error");  
}
```

```
void WAIT_CHILD(void)
```

```
{  
    char c;  
  
    if (read(pfd2[0], &c, 1) != 1) err_sys("read error");  
    if (c != 'c') err_quit("WAIT_CHILD: incorrect data");  
}
```

Esempio 3 (cont.)



Funzioni `popen` e `pclose`

Poiché aprire e usare una pipe con un altro processo è un'operazione comune, la libreria standard I/O implementa le funzioni `popen` e `pclose`.

`popen` crea una pipe, crea il figlio e esegue il comando, `pclose` chiude le estremità non utilizzate della pipe, e attende che il comando sia terminato.

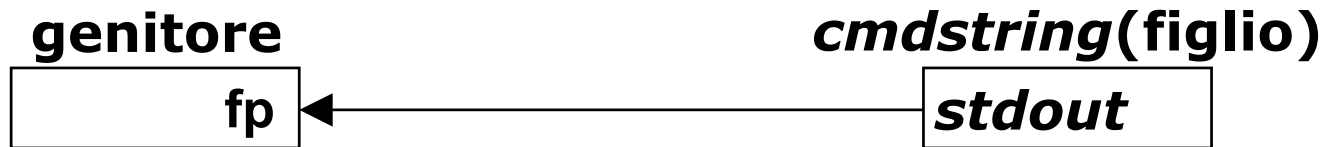
```
# include <stdio.h>
```

```
FILE *popen(const char *command, const char *type);
```

```
int pclose(FILE *stream);
```

Funzioni `popen` e `pclose` (cont.)

La funzione `popen` esegue un `fork` ed una `exec` per eseguire `command` e ritorna un puntatore a file. Se `type` è `"w"`, il puntatore a file è connesso allo standard input di `command`, se `"r"` allo standard output.



La funzione `pclose` chiude lo stream di I/O e attende che il comando sia terminato, ritornando lo stato di terminazione della shell.

`command` è eseguito come `sh -c command`, il che significa che la shell espande i caratteri speciali. Ad esempio si può scrivere

```
fp=popen("ls *.c", "r"); M. Guarracino - PIPE
```

```
# include <sys/wait.h>
# include "ourhdr.h"
#define PAGER "${PAGER:-more}"
int main(int argc, char *argv[ ])
{
    char line[MAXLINE];
    FILE *fpin, *fpout;

    if (argc != 2) err_quit("usage: a.out <pathname>");
    if ( (fpin = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if ( (fpout = popen(PAGER, "w")) == NULL) err_sys("error");

    /* copy argv[1] to pager */
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF) err_sys("fputs error to pipe");
    }
    if (ferror(fpin)) err_sys("fgets error");
    if (pclose(fpout) == -1) err_sys("pclose error");
    exit(0);
}
```

```
# include <ctype.h>
# include "ourhdr.h"
int main(void) /* myuclc */
{
    int c;
    while ( (c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

```
#include <sys/wait.h>
#include "ourhdr.h"
```

```
int main(void)
{
```

```
    char line[MAXLINE];
    FILE *fpin;
```

```
    if ( (fpin = popen("myucl", "r")) == NULL) err_sys("error");
```

```
    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read pipe */
            break;
        if (fputs(line, stdout) == EOF) err_sys("fputs error ");
```

```
    }
    if (pclose(fpin) == -1) err_sys("pclose error");
    putchar('\n');
    exit(0);
```

Questo programma scrive un prompt sullo standard output e legge una linea dallo standard input, trasformando un qualsiasi carattere maiuscolo in minuscolo.

Le FIFO

Le **pipe** possono essere utilizzate **solo** tra **processi** con un **antenato** in comune. Con le **FIFO** è possibile far comunicare **processi qualsiasi**.

Le FIFO sono un tipo di file; uno dei possibili valori del campo **st_mode** della struttura **stat** indica che il file è una FIFO. Per provarlo si utilizza la macro **S_ISFIFO**.

Creare una FIFO è simile a creare un file e **pathname** per una FIFO esiste nel filesystem.

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

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

L'argomento **mode** per la funzione **mkfifo** è uguale all'analogo per la funzione **open**.

Le FIFO (cont.)

Una volta creata una FIFO, si apre usando **open** e si possono eseguire tutte le normali operazioni di I/O su file (**close**, **read**, **write**, **unlink**,...).

Quando si apre (**open**) una FIFO il flag **O_NONBLOCK** influenza le operazioni.

1. se **O_NONBLOCK** non è specificato, una **open** in sola lettura blocca fin quando un altro processo apre per scrivere. In modo analogo una **open** in sola scrittura blocca fin quando un altro processo apre per leggere.

2. Se **O_NONBLOCK** è specificato, una **open** in sola lettura ritorna subito, mentre una **open** in sola scrittura ritorna un errore con un valore **ENXIO** di **errno**, se non c'è un processo che ha aperto la FIFO in lettura.

Le FIFO (cont.)

Come per le pipe, se si esegue una **write** su di una FIFO che nessun processo ha aperto in lettura, viene generato **SIGPIPE**.

E' comune la situazione in cui **più processi scrivono** su di una stessa FIFO. Affinché i dati **non si interfogliano** è necessario utilizzare **operazioni atomiche** di scrittura.

Come per le pipe, la costante **PIPE_BUF** stabilisce il massimo numero di byte che possono essere scritti in maniera atomica in una FIFO.

Gli usi più comuni per le FIFO sono:

- 1.il passaggio di dati da un pipeline all'altro senza creare file,

- 2.il passaggio di dati in ambiente client-server.

Duplicazione degli stream di output

Le FIFO possono essere utilizzate per duplicare uno stream di output in una serie di comandi di shell. L'utilizzo delle FIFO permette di implementare connessioni non lineari.

Si consideri una procedura che deve processare uno stream di input due volte; utilizzando una FIFO ed il comando Unix **tee**(1) possiamo risolvere questo problema senza utilizzare file temporanei.

```
mkfifo fifo1
```

```
prog3 < fifo1 &
```

```
prog1 < infile | tee fifo1 | prog2
```

Duplicazione degli stream di output

Dopo aver creato la FIFO con **mkfifo(1)**, si lancia **prog3** in background, facendo leggere il suo input dalla FIFO.

prog1 viene quindi lanciato, e con il comando **tee** il suo output viene inviato sia allo standard output, da cui viene prelevato da **prog2** mediante una pipe, sia sulla FIFO **fifo1**, da cui viene prelevato da **prog3**.

Comunicazione Client-Server mediate FIFO

Un altro uso delle FIFO è quello di scambiare dati tra un client ed un server.

Se si ha un server contattato da numerosi client, ciascuno dei client può scrivere su di una FIFO creata dal server. Il nome della FIFO deve essere nota a tutti i client che hanno bisogno di contattare il server.

Affinché i dati di ogni client non si mischino con quelli degli altri è necessario che il numero di byte scritti da ciascuno sia inferiore a **PIPE_BUF**.

Il problema nasce quando il server vuole mandare una risposta ai client, in quanto con singola FIFO i client non potrebbero sapere quando leggere i dati a loro destinati. Una soluzione è quella di creare una FIFO per ogni client, usando ad esempio una FIFO il cui nome contiene l'identificativo di processo del client.

Se il server apre la FIFO in sola lettura, nel momento in cui il numero dei client passa da 1 a 0, il server otterrà da read un EOF.

Per evitare di incorrere in questo problema, si può aprire la FIFO in lettura e scrittura.