

Pipe

Davide Lamanna
lamanna@dis.uniroma1.it

Contenuti

- Nozioni preliminari sulle PIPE
- PIPE e Named PIPE (o FIFO) in sistemi UNIX
- PIPE e Named PIPE in sistemi WINDOWS

Concetti di base sulle PIPE (1)

- ❑ Permettono a più processi di comunicare come se stessero accedendo a file sequenziali
- ❑ La comunicazione avviene in modo monodirezionale
- ❑ Una volta lette le informazioni non sono più disponibili, a meno di ri-scritture
- ❑ A livello di sistema operativo, le PIPE non sono altro che buffer di dimensione variabile (tipicamente 4096 Byte)
 - È possibile quindi che un processo venga bloccato se tenta di scrivere su una PIPE piena

Concetti di base sulle PIPE (2)

- ❑ I processi che usano PIPE devono essere “relazionati”
 - Es. Processo padre e figlio dopo una fork

- ❑ Le FIFO (o Named PIPE) permettono la comunicazione anche tra processi non relazionati

- ❑ In sistemi UNIX l'uso delle PIPE avviene attraverso la nozione di descrittore di file

- ❑ In sistemi WINDOWS l'uso delle PIPE avviene attraverso la nozione handle

PIPE in sistemi UNIX

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

Descrizione: Invoca la creazione di una PIPE

e:

Argomenti: fd: puntatore ad un buffer di due interi (in fd[0] viene restituito il descrittore di lettura della PIPE, in fd[1] viene restituito il descrittore di scrittura della PIPE)

Restituzione: -1 in caso di fallimento

- ❑ fd[0] è un canale in **lettura** che consente ad un processo di leggere dati dalla PIPE
- ❑ fd[1] è un canale in **scrittura** che consente ad un processo di scrivere dati sulla PIPE
- ❑ fd[0] e fd[1] possono essere usati come normali descrittori di file tramite le chiamate **read()** e **write()**

Avvertenze!!!

- Un processo lettore vede la “fine” della PIPE quando tutti i processi scrittori hanno chiuso il descrittore `fd[1]`
 - La chiamata **`read()`** effettuata da un lettore restituisce 0 come notifica dell’evento che tutti gli scrittori hanno terminato il loro lavoro

- Un processo scrittore che tenti di scrivere sul descrittore `fd[1]` quando tutte le copie del descrittore `fd[0]` sono state chiuse (cioè non ci sono lettori sulla PIPE) riceve il segnale `SIGPIPE`

PIPE e deadlock

- Per evitare deadlock è necessario che tutti i processi chiudano i descrittori di PIPE che non gli servono usando la **close()**
 - Ogni processo lettore deve chiudere la propria copia di `fd[1]` prima di mettersi a leggere da `fd[0]` dichiarando così di non essere uno scrittore
 - Se così non facesse, l'evento "tutti gli scrittori hanno terminato" non potrebbe mai avvenire; se il lettore tenta di leggere rimane bloccato.
 - Analogamente ogni processo scrittore deve chiudere la propria copia di `fd[0]`

Esempio: trasferimento stringhe

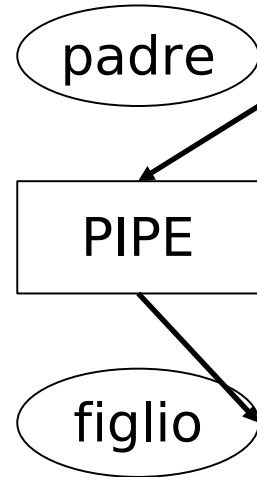
```
#include <stdio.h>
#define Errore_(x) { puts(x); exit(1); }

int main(int argc, char *argv[]) {
    char messaggio[30];
    int pid, status, fd[2], ret;

    ret = pipe(fd); /* crea una PIPE */
    if ( ret == -1 ) Errore_("Errore nella chiamata pipe");

    pid = fork(); /* crea un processo figlio */
    if ( pid == -1 ) Errore_("Errore nella fork");

    if ( pid == 0 ) { /* processo figlio: lettore */
        close(fd[1]); /* il lettore chiude fd[1] */
        while( read(fd[0], messaggio, 30) > 0 )
            printf("letto messaggio: %s", messaggio);
        close(fd[0]);
    }
}
```



..... continua

```
/* processo padre: scrittore */
else {
    close(fd[0]); /* lo scrittore chiude fd[0] */
    puts("digitare testo da trasferire (quit per
terminare):");

    do {
        fgets(messaggio,30,stdin);
        write(fd[1], messaggio, 30);
        printf("scritto messaggio: %s", messaggio);
    } while( strcmp(messaggio,"quit\n") != 0 );
    close(fd[1]);

    wait(&status);
}
}
```

FIFO (Named PIPE) in sistemi UNIX

```
int mkfifo(char* name, int mode)
```

Descrizione: Invoca la creazione di una FIFO

Argomenti:

1. name: puntatore ad una stringa che identifica il nome della FIFO da creare
2. mode: intero che specifica la modalità di creazione e permessi di accesso alla FIFO

Restituzione: -1 in caso di fallimento, altrimenti un descrittore per l'accesso alla FIFO

- ❑ La rimozione di una FIFO dal file system avviene mediante la chiamata di sistema **unlink()**
- ❑ Per usare la FIFO si dovrà esplicitamente usare la system call **open()** per aprirla (in modalità lettura o scrittura)

Avvertenze!!!

- L'apertura di una FIFO è bloccante
 - Un processo che tenta di aprirla in lettura (scrittura) viene bloccato fino a quando un altro processo non la apre in scrittura (lettura)
- Se si vuole inibire questo comportamento si deve passare alla **open ()** il flag **O_NONBLOCK**
- Ogni FIFO deve avere sia un processo lettore che un processo scrittore
 - Se un processo tenta di scrivere su una FIFO che non ha un lettore esso riceve il segnale "SIGPIPE" da parte del SO

Esempio: client/server tramite FIFO

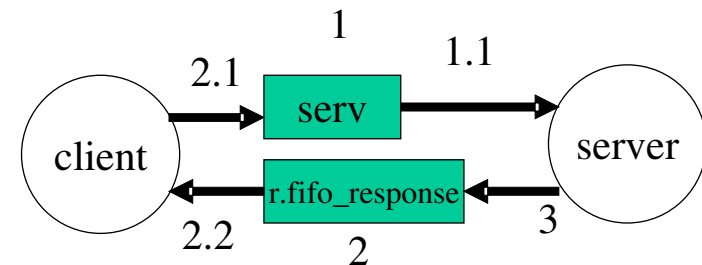
```
#include <stdio.h>
#include <fcntl.h>
```

```
typedef struct {
    long type;
    char fifo_response[20];
} request;
```

```
void disconnetti_pipe() {
    printf("Server fermato. Rimozione FIFO dal sistema....");
    unlink("serv");
    printf("fatto.\n");
    exit(0);
}
```

```
int main(int argc, char *argv[]){
    char *response = "fatto";
    int pid, fd, fdc, ret;
    request r;

    signal(SIGTERM, disconnetti_pipe);
    signal(SIGHUP, disconnetti_pipe);
    signal(SIGINT, disconnetti_pipe);
```



..... continua (server)

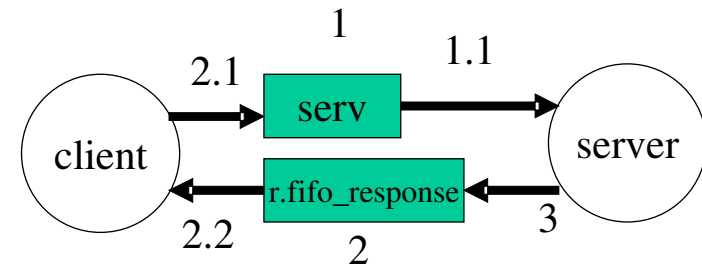
```

(1) ret = mkfifo("serv", 0_CREAT|0666);
    if ( ret == -1 ) {
        printf("Errore nella chiamata mkfifo\n");
        exit(1);
    }

    fd = open("serv",0_RDWR);
    while(1) {
(1.1) if ((ret = read(fd, &r, sizeof(request))) != 0) {
        pid = fork();
        if (pid == 0) {
            printf("Richiesto un servizio (fifo di restituzione = %s)\n",
                r.fifo_response);

            /* switch sul tipo di servizio */
            sleep(10); /*emulazione di ritardo per il servizio*/
(3) fdc = open(r.fifo_response,0_WRONLY);
            ret = write(fdc, response, 20);
            ret = close(fdc);
            exit(0);
        } /* end if */
    } /* end if */
}
}

```



..... continua (client)

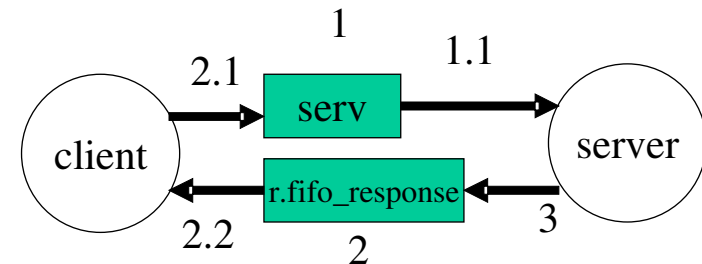
```
#include <stdio.h>
#include <fcntl.h>

typedef struct {
    long type;
    char fifo_response[20];
} request;

void disconnetti_pipe() {
    printf("Server fermato. Rimozione FIFO dal sistema....");
    fflush(stdout);
    unlink(r.fifo_response);
    printf("fatto.\n");
    exit(0);
}

int main(int argc, char *argv[]) {
    int pid, fd, fdc, ret; request r;
    char response[20];

    signal(SIGTERM, disconnetti_pipe);
    signal(SIGHUP, disconnetti_pipe);
    signal(SIGINT, disconnetti_pipe);
```



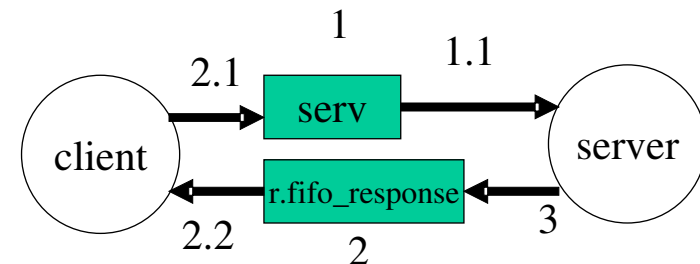
..... continua (client)

```
printf("Selezionare un carattere alfabetico minuscolo: ");
scanf("%s", r.fifo_response);
if (r.fifo_response[0] > 'z' || r.fifo_response[0] < 'a' ) {
    printf("carattere selezionato non valido, ricominciare
           operazione\n");
    exit(1);
}
```

```
(2) r.fifo_response[1] = '\0';
    ret = mkfifo(r.fifo_response, 0_CREAT|0666);
```

```
if ( ret == -1 ) {
    printf("\n server sovraccarico - riprovare \n");
    exit(1);
}
```

```
fd = open("serv", O_WRONLY);
if ( fd == -1 ) {
    printf("\n servizio non disponibile \n");
    ret = unlink(r.fifo_response);
    exit(1);
}
```



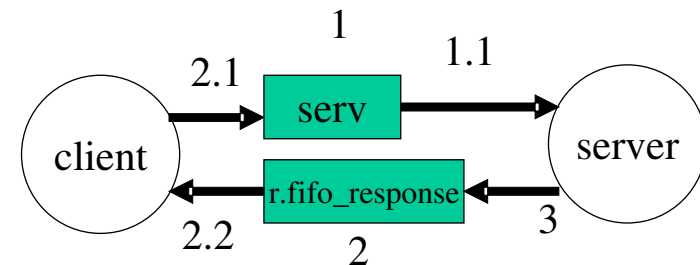
..... continua (client)

```
(2.1)ret = write(fd, &r, sizeof(request));  
ret = close(fd);
```

```
fdc = open(r.fifo_response, O_RDWR);  
(2.2)ret = read(fdc, response, 20);  
printf("risposta = %s\n", response);
```

```
ret = close(fdc);  
ret = unlink(r.fifo_response);
```

```
}
```



PIPE in sistemi WINDOWS

```
BOOL CreatePipe(PHANDLE hReadPipe,  
                PHANDLE hWritePipe,  
                LPSECURITY_ATTRIBUTES lpPipeAttributes,  
                DWORD nSize)
```

Descrizione: Invoca la creazione di una PIPE

Argomenti:

1. hReadPipe: puntatore a una variabile in cui viene scritto l'handle all'estremità di lettura della PIPE
2. hWritePipe: puntatore a una variabile in cui viene scritto l'handle all'estremità di scrittura della PIPE
3. lpPipeAttributes: puntatore a una struttura SECURITY_ATTRIBUTES che specifica se gli handle ritornati sono ereditabili da processi figli
4. nSize: dimensione suggerita della PIPE (0 setta il valore di default)

Restituzione: 0 (false) in caso di fallimento

Avvertenze!!!

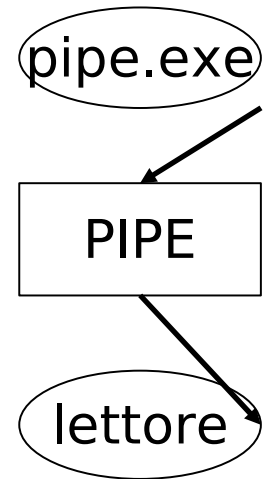
- ❑ `hReadPipe` è un canale aperto in lettura che consente ad un processo di leggere dati dalla PIPE
- ❑ `hWritePipe` è un canale aperto in scrittura che consente ad un processo di scrivere dati sulla PIPE
- ❑ `hReadPipe` e `hWritePipe` si usano come se fossero handle di file
 - Cioè attraverso le chiamate **`ReadFile()`** e **`WriteFile()`**
- ❑ Anche per WINDOWS, la fine di un file sulla PIPE è visibile quando tutti i processi scrittori che condividono l'handle `hWritePipe` lo hanno chiuso
 - In questo caso la chiamata **`ReadFile()`** restituisce 0

Esempio: trasferimento stringhe

```
#include <stdio.h>
#include <windows.h>
#define Errore_(x) { puts(x); ExitProcess(1); }

int main(int argc, char *argv[]) {
    char messaggio[30];
    SECURITY_ATTRIBUTES security;  BOOL rit;
    HANDLE readHandle, writeHandle, temp_readHandle;
    DWORD result;  BOOL newprocess;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    if (argc==1) { /* Creazione pipe */
        security.nLength = sizeof(security);
        security.lpSecurityDescriptor = NULL;
        security.bInheritHandle = TRUE;

        rit = CreatePipe(&readHandle,
                        &writeHandle,
                        &security, 0);
        if (!rit) Errore_("Errore nella CreatePipe");
    }
}
```



..... continua

```
/* genero il processo scrittore */
```

```
memset(&si, 0, sizeof(si));
```

```
memset(&pi, 0, sizeof(pi));
```

```
si.cb = sizeof(si);
```

```
temp_readHandle = GetStdHandle(STD_INPUT_HANDLE);
```

```
SetStdHandle(STD_INPUT_HANDLE, readHandle);
```

```
newprocess = CreateProcess(".", "\\pipe.exe",  
                           ".", "\\pipe.exe lettore",
```

```
NULL, NULL,
```

```
TRUE,
```

```
NORMAL_PRIORITY_CLASS,
```

```
NULL, NULL,
```

```
&si, &pi);
```

```
if (newprocess == 0) {
```

```
    printf("Errore nella generazione dello scrittore!\n");
```

```
    ExitProcess(-1);
```

```
}
```

```
SetStdHandle(STD_INPUT_HANDLE, temp_readHandle);
```

```
CloseHandle(readHandle);
```

```
printf("digitare testo da trasferire (quit per terminare):\n");
```

..... continua

```
do {
    fgets(messaggio,30,stdin);
    rit = WriteFile(writeHandle,messaggio,30, &result, NULL);
    if (!rit) Errore_("Errore nella writefile!");
    printf("scritto messaggio: %s", messaggio);
} while( strcmp(messaggio,"quit\n") != 0 );

CloseHandle(writeHandle);
WaitForSingleObject(pi.hProcess, INFINITE);
}
else if (argc == 2 && strcmp(argv[1], "lettore") == 0)
/* processo lettore */
{
    do {
        rit = ReadFile(GetStdHandle(STD_INPUT_HANDLE),
                       messaggio, 30, &result, NULL);

        if ( rit )
            printf("letto messaggio: %s", messaggio);
        if (strcmp(messaggio, "quit\n") == 0) break;
    } while (rit);
}
}
```

Named PIPE in sistemi WINDOWS

```
BOOL CreateNamedPipe(LPCTSTR lpName, DWORD dwOpenMode, DWORD dwPipeMode,
                    DWORD nMaxInstances, DWORD nOutBufferSize,
                    DWORD nInBufferSize, DWORD nDefaultTimeOut,
                    PSECURITY_ATTRIBUTES lpSecurityAttributes)
```

Descrizione Invoca la creazione e/o l'apertura di una Named PIPE

Argomenti

1. lpName: puntatore ad una stringa che identifica il nome della Named PIPE da creare
2. dwOpenMode: specifica modalità di creazione e permessi di accesso alla Named PIPE
3. dwPipeMode: specificare la tipologia dell'handle restituito
4. nMaxInstances: numero di istanze della Named PIPE che è possibile creare
5. nOutBufferSize: dimensione massima del buffer di output (in Bytes)
6. nInBufferSize: dimensione massima del buffer di input (in Bytes)
7. nDefaultTimeOut: valore di timeout (in msec) da usare come default
8. lpSecurityAttributes: puntatore a una struttura

Restituzione: SECURITY_ATTRIBUTES che specifica se gli handle ritornati sono ereditabili da processi figli
INVALID_HANDLE_VALUE in caso di fallimento; un handle alla Named PIPE in caso di successo

Avvertenze!!!

- Il nome della Named PIPE (lpName) deve essere nel formato `\\.\\pipe\\pipename`, dove pipename può essere qualunque stringa di lunghezza inferiore a 256 caratteri e non includente i caratteri `\` e `.`
 - NOTA: per stampare il carattere `\` si deve usare il carattere di escape `\`, cioè `printf("\\");` stampa una `\`

- La chiamata **CreateNamedPipe**
 - **Crea** una nuova Named PIPE (se già non esiste)
 - **Apri** una Named PIPE esistente

- La rimozione della Named PIPE avviene nel momento in cui viene chiuso l'ultimo handle aperto sulla Named PIPE

Esempio: client/server tramite Named PIPE

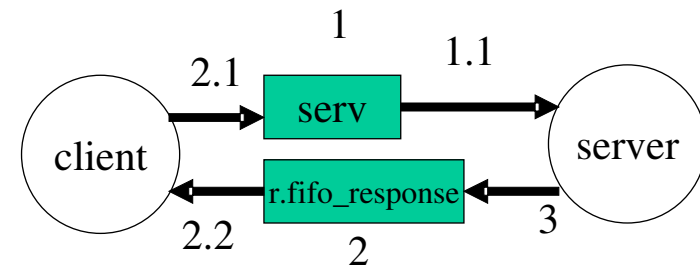
```
#include <stdio.h>
#include <windows.h>
```

```
typedef struct {
    long type;
    char fifo_response[20];
} request;
```

```
DWORD WINAPI server(LPVOID r);
```

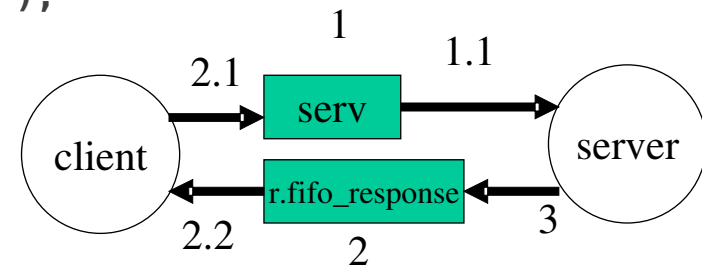
```
int main(int argc, char* argv[]) {
    HANDLE my_pipe; DWORD tid;
    DWORD ret; request r;
    HANDLE hThreadHandle;
    BOOL fConnected;
```

```
while(1) {
    (1) my_pipe = CreateNamedPipe(
        "\\\\.\\pipe\\serv",
        PIPE_ACCESS_DUPLEX, 0,
        PIPE_UNLIMITED_INSTANCES,
        0, 0, NMPWAIT_USE_DEFAULT_WAIT, NULL);
```



..... continua (server)

```
if ( my_pipe == INVALID_HANDLE_VALUE ) {  
    printf("Errore nella chiamata CreateNamedPipe\n");  
    ExitProcess(1);  
}  
  
fConnected = ConnectNamedPipe(my_pipe, NULL) ?  
    TRUE : (GetLastError() == ERROR_PIPE_CONNECTED);  
  
(1.1) if (fConnected) {  
    if (!ReadFile(my_pipe, &r, sizeof(r), &ret, NULL)) {  
        printf("Errore nella chiamata readfile\n");  
        ExitProcess(1);  
    }  
  
    hThreadHandle = CreateThread(  
        NULL, 0, server, &r, 0, &tid);  
  
    if (hThreadHandle == NULL) {  
        printf("Errore nella creazione del thread.\n");  
        ExitProcess(1);  
    }  
}  
else CloseHandle(hThreadHandle);  
}  
return(0);  
}
```



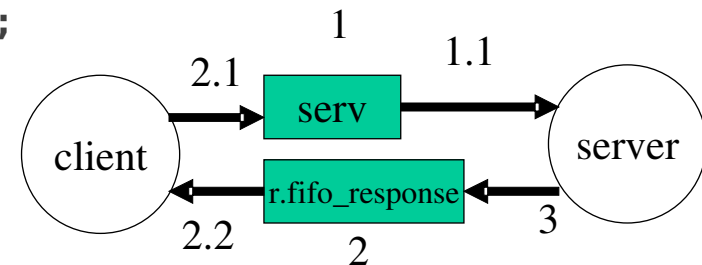
..... continua (server)

```
DWORD WINAPI server(LPVOID r) {
    HANDLE my_pipe; char fifo_response[80];
    char *response = "fatto"; DWORD result;
    printf("Richiesto un servizio (fifo di restituzione = %s)\n",
           ((request *)r)->fifo_response);
    /* switch sul tipo di messaggio: da implementare */
    /* servizio..... */
    strcpy(fifo_response, "\\.\pipe\");
    strcat(fifo_response, ((request *)r)->fifo_response);

    my_pipe = CreateFile(fifo_response, GENERIC_WRITE,
                        0, NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);
```

(3)

```
    if (!WriteFile(my_pipe, response, 20, &result, NULL)) {
        printf("Errore nella chiamata WriteFile\n");
        ExitProcess(1);
    }
    CloseHandle(my_pipe);
    ExitThread(0);
    return(0);
}
```



..... continua (client)

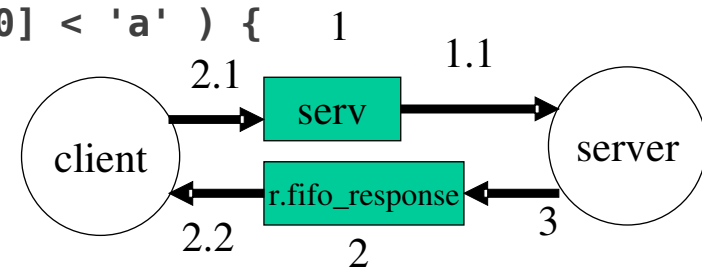
```
#include <stdio.h>
#include <windows.h>

typedef struct {
    long type;
    char fifo_response[20];
} request;

int main(int argc, char *argv[]) {
    DWORD ret;  request r;
    char response[20];  char fifo_response[80];
    HANDLE response_pipe, my_pipe;
    BOOL fDone, fConnected;

    printf("Selezionare un carattere alfabetico minuscolo: ");
    scanf("%s",r.fifo_response);

    if (r.fifo_response[0] > 'z' || r.fifo_response[0] < 'a' ) {
        printf("carattere selezionato non valido,
              ricominciare operazione\n");
        ExitProcess(1);
    }
}
```



..... continua (client)

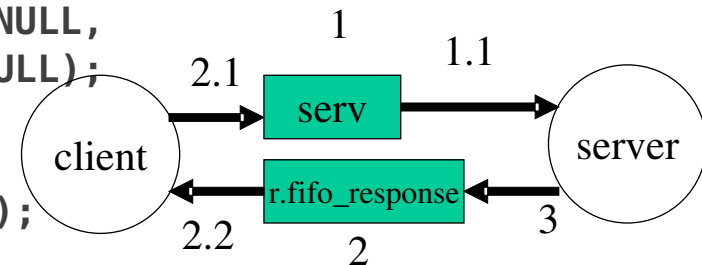
```
r.fifo_response[1] = '\0';  
strcpy(fifo_response, "\\.\pipe\\");  
strcat(fifo_response, r.fifo_response);  
fDone = FALSE;
```

```
while (1) {  
(2) response_pipe = CreateNamedPipe(  
    fifo_response, PIPE_ACCESS_INBOUND, 0,  
    PIPE_UNLIMITED_INSTANCES, 0, 0, 0, NULL);
```

```
if ( response_pipe == INVALID_HANDLE_VALUE ) {  
    printf("Errore nella chiamata CreateNamedPipe\n");  
    ExitProcess(1);  
}
```

```
if (!fDone) {  
    my_pipe = CreateFile(  
        "\\.\pipe\\serv", GENERIC_WRITE, 0, NULL,  
        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

```
if ( my_pipe == INVALID_HANDLE_VALUE ) {  
    printf("\n servizio non disponibile \n");  
    ExitProcess(1);  
}
```



..... continua (client)

```
(2.1) if (!WriteFile(my_pipe,&r,sizeof(request),&ret,NULL)) {
    printf("Errore nella chiamata WriteFile\n");
    ExitProcess(1);
}
```

```
CloseHandle(my_pipe);
fDone = TRUE;
```

```
}
fConnected = ConnectNamedPipe(response_pipe, NULL) ?
TRUE : (GetLastError() == ERROR_PIPE_CONNECTED);
```

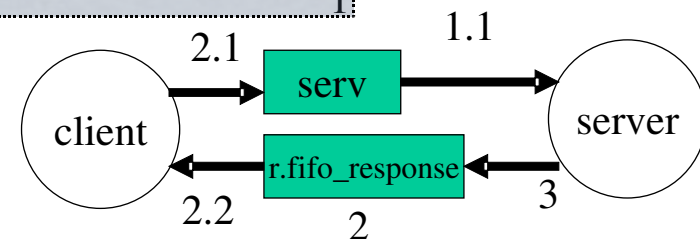
```
if (fConnected) {
    if (!ReadFile(response_pipe, response, 20, &ret, NULL)) {
        printf("Errore nella chiamata ReadFile\n");
        ExitProcess(1);
    }
    printf("risposta = %s\n", response);
    break;
}
```

(2.2)

```
else CloseHandle(response_pipe);
```

```
}
return(0);
```

```
}
```



Riferimenti

□ Linux

- Kay A. Robbins, Steven Robbins, "*UNIX SYSTEMS Programming*", Prentice Hall PTR
 - Chapter 6: "UNIX Special Files"
- Francesco Quaglia, Camil Demetrescu, "*Programmazione in Ambiente UNIX*", Roma, 1999

□ Windows

- MSDN Library: documentazione on-line sulle win API.
 - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_api_reference.asp