

# Sistemi Operativi II

Corso di Laurea in Ingegneria Informatica

Facolta' di Ingegneria, Universita' "La Sapienza"

Docente: Francesco Quaglia

## **Pipes e named pipes:**

1. Nozioni preliminari
2. Pipes e named pipes (FIFO) in sistemi UNIX
3. Pipes e named pipes sistemi Windows

# Concetti di base sulle PIPE

- permettono a più processi di comunicare come se stessero accedendo a dei file sequenziali
- il termine “pipe” significa tubo in Inglese, e la comunicazione avviene in modo monodirezionale
- una volta lette, le informazioni spariscono dalla PIPE e non possono più ripresentarsi, a meno che non vengano riscritte all'altra estremità del tubo
- a livello di sistema operativo, i PIPE non sono altro che buffer di dimensione più o meno grande (solitamente 4096 byte), quindi è possibile che un processo venga bloccato se tenta di scrivere su un PIPE pieno
- i processi che usano un PIPE devono essere “relazionati”
- le named PIPE (FIFO) permettono la comunicazione anche tra processi non relazionati
- in sistemi UNIX l'uso della PIPE avviene attraverso la nozione di descrittore, in sistemi Windows attraverso la nozione di handle

# PIPE in Sistemi UNIX

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

---

**Descrizione** invoca la creazione di una PIPE

---

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

---

**Restituzione** -1 in caso di fallimento

fd[0] è un canale aperto in lettura che consente ad un processo di leggere dati da una PIPE

fd[1] è un canale aperto in scrittura che consente ad un processo di immettere dati sulla PIPE

fd[0] e fd[1] possono essere usati come normali descrittori di file tramite le chiamate read() e write()

# Avvertenze

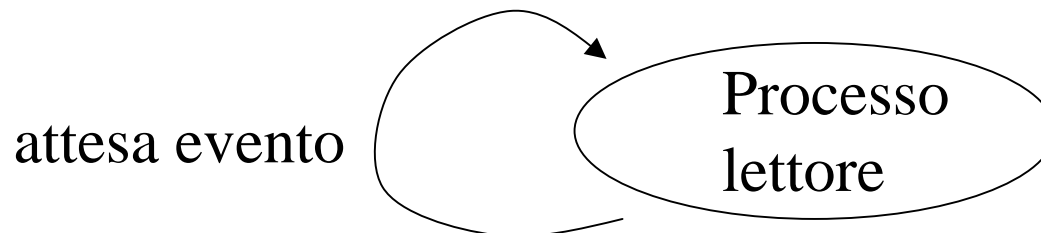
- le PIPE non sono dispositivi fisici, ma logici, pertanto viene spontaneo chiedersi come un processo sia in grado di vedere la fine di un “file” su una PIPE
- per convenzione, ciò avviene quando tutti i processi scrittori che condividevano il descrittore `fd[1]` lo hanno chiuso
- tecnicamente, in questo caso la chiamata `read()` effettuata da un lettore restituisce zero come notifica dell'evento che tutti gli scrittori hanno terminato il loro lavoro
- allo stesso modo, un processo scrittore che tenti di scrivere sul descrittore `fd[1]` quando tutte le copie del descrittore `fd[0]` siano state chiuse (non ci sono lettori sulla PIPE), riceve il “segnale SIGPIPE”, altrimenti detto Broken-pipe

# PIPE e deadlock

Per fare in modo che tutto funzioni correttamente e non si verifichino situazioni di deadlock è necessario che tutti i processi chiudano i descrittori di PIPE che non gli servono, usando una normale `close()`

Si noti che ogni processo lettore che erediti la coppia  $(fd[0],fd[1])$  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 è impegnato a leggere, e si potrebbe avere un deadlock



# Un esempio: trasferimento stringhe tramite PIPE

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

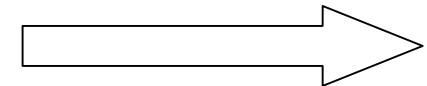
int main(int argc, char *argv[]) {
    char messaggio[30];
    int pid, status, fd[2];

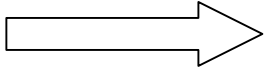
    ret = pipe(fd); /* crea una PIPE */
    if ( ret == -1 ) Errore_("Errore nella chiamata pipe");

    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]);

        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);
    }
}
```

# Named PIPE (FIFO) 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 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 esattamente come per i file mediante la chiamata di sistema `unlink()`

# Avvertenze

- normalmente, l'apertura di una FIFO è bloccante, nel senso che il 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 è possibile aggiungere il flag `O_NONBLOCK` al valore del parametro `mode` passato alla system call `open()` su di una FIFO
- ogni FIFO deve avere sia un lettore che uno scrittore: se un processo tenta di scrivere su una FIFO che non ha un lettore esso riceve una notifica di errore da parte del sistema operativo

# Un esempio: client/server tramite FIFO

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

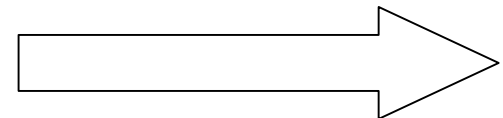
```
typedef struct {
    long type;
    char fifo_response[20];
} request;
```

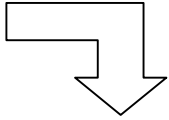
```
int main(int argc, char *argv[]){
    char *response = "fatto";
    int pid, fd, fdc, ret;
    request r;

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

## Processo server

continua





```
fd = open("serv",O_RDWR);
```

```
while(1) {  
    ret = read(fd, &r, sizeof(request));  
    if (ret != 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 */  
            fdc = open(r.fifo_response,O_WRONLY);  
            ret = write(fdc, response, 20);  
            ret = close(fdc);  
            exit(0);  
        } /* end if */  
    } /* end if */  
}  
}
```

```
#include <stdio.h>
#include <fcntl.h>
typedef struct {
    long type;
    char fifo_response[20];
} request;
```

## Processo client

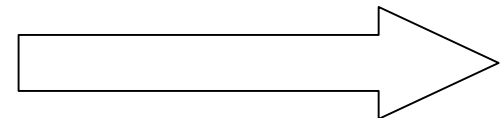
```
int main(int argc, char *argv[]) {
    int pid, fd, fdc, ret; request r;    char response[20];

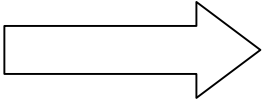
    printf("Selezionare un carattere alfabetico minuscolo: ");
    scanf("%s",r.fifo_response);

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

    ret = mkfifo(r.fifo_response, O_CREAT|0666);
    if ( ret == -1 ) {
        printf("\n servente sovraccarico - riprovare \n");
        exit(1);
    }
}
```

continua





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

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

fdc = open(r.fifo_response,O_RDWR);
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

- `hReadPipe`: puntatore a una variabile in cui viene scritto l'handle all'estremità di lettura dalla pipe
- `hWritePipe`: puntatore a una variabile in cui viene scritto l'handle all'estremità di scrittura sulla pipe
- `lpPipeAttributes`: puntatore a una struttura `SECURITY_ATTRIBUTES` che specifica se gli handle ritornati sono ereditabili da processi figli
- `nSize`: dimensione suggerita della PIPE (0 attiva il default)

## Restituzione

- 0 in caso di fallimento

# Avvertenze

- `hReadPipe` è un canale aperto in lettura che consente ad un processo di leggere dati da una PIPE
- `hWritePipe` è un canale aperto in scrittura che consente ad un processo di immettere dati sulla PIPE
- `hReadPipe` e `hWritePipe` possono essere usati come se fossero normali handle di file tramite le chiamate `ReadFile()` e `WriteFile()`
- anche per Windows, la fine di un file su una PIPE è visibile, per convenzione, quando tutti i processi scrittori che condividevano l'handle `hWritePipe` lo hanno chiuso
- tecnicamente, in questo caso la chiamata `ReadFile()` effettuata da un lettore restituisce zero come notifica dell'evento che tutti gli scrittori hanno terminato il loro lavoro.

# Un esempio: trasferimento stringhe tramite PIPE

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

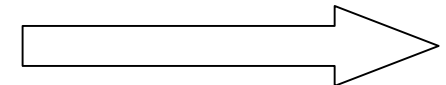
int main(int argc, char *argv[]) {
    char messaggio[30];
    SECURITY_ATTRIBUTES security; BOOL rit;
    HANDLE readHandle, writeHandle, temp_readHandle;
    DWORD result;  BOOL newprocess;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    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");
```

**Processo padre**

continua



```
/* genero il processo scrittore */
    memset(&si, 0, sizeof(si));
    memset(&pi, 0, sizeof(pi));
    si.cb = sizeof(si);

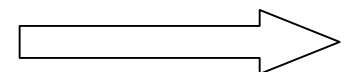
temp_readHandle = GetStdHandle(STD_INPUT_HANDLE);
SetStdHandle(STD_INPUT_HANDLE, readHandle);

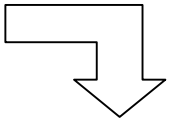
newprocess = CreateProcess(".\\pipe.exe",
                          ".\\pipe.exe lettore",
                          NULL, NULL,
                          TRUE,
                          NORMAL_PRIORITY_CLASS,
                          NULL, NULL,
                          &si, &pi);
if (newprocess == 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){

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

```
HANDLE CreateNamedPipe(LPCTSTR lpName,  
                        DWORD dwOpenMode,  
                        DWORD dwPipeMode,  
                        DWORD nMaxInstances,  
                        DWORD nOutBufferSize,  
                        DWORD nInBufferSize,  
                        DWORD nDefaultTimeout,  
                        LPSECURITY_ATTRIBUTES lpSecurityAttributes)
```

## Descrizione

- invoca la creazione e/o apertura di un Named PIPE

## Restituzione

- INVALID\_HANDLE\_VALUE in caso di fallimento; un handle alla named PIPE in caso di successo

# Argomenti

- lpName: puntatore ad una stringa che identifica il nome della named PIPE da creare
- dwOpenMode: intero che specifica modalità di creazione e permessi di accesso alla named PIPE
- dwPipeMode: intero che specifica la tipologia dell'handle restituito
- nMaxInstances: specifica il massimo numero di istanze che possono essere create
- nOutBufferSize: dimensione massima del buffer di output (in bytes)
- nInBufferSize: dimensione massima del buffer di input (in bytes)
- nDefaultTimeOut: valore di timeout (in millisecondi) da usare come default in particolari situazioni
- lpSecurityAttributes: puntatore a una struttura SECURITY\_ATTRIBUTES che specifica se l'handle ritornato è ereditabile da processi figli

# Avvertenze

- il nome della named PIPE (lpName) deve essere nel formato `\\. \pipe\pipename` dove “pipename” può essere qualunque stringa di lunghezza inferiore ai 256 caratteri e non includente i caratteri ```\` e ``:```
- se una named PIPE con questo nome già esiste, la Named PIPE viene aperta, altrimenti viene sia creata che aperta
- la rimozione di una named PIPE dal sistema avviene nel momento in cui viene chiuso l'ultimo handle aperto sulla named PIPE

# Client/server tramite named PIPE

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

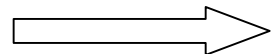
```
typedef struct {
    long type;
    char fifo_response[20];
} request;
```

Processo 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);
```

Continua



```

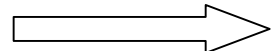
if (!WriteFile(my_pipe, response, 20, &result, NULL)) {
    printf("Errore nella chiamata WriteFile\n");
    ExitProcess(1);
}
CloseHandle(my_pipe);
ExitThread(0);
return(0);
}

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

    my_pipe = CreateNamedPipe("\\\\.\\pipe\\serv",
                              PIPE_ACCESS_DUPLEX, 0,
                              PIPE_UNLIMITED_INSTANCES,
                              0, 0, 0, NULL);

```

Continua





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

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

## Processo client

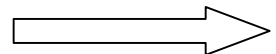
```
int main(int argc, char *argv[]) {
    DWORD ret; request r;
    char response[20]; char fifo_response[80];
    HANDLE response_pipe,my_pipe;

    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);
    }
    r.fifo_response[1] = '\\0';

    strcpy(fifo_response, "\\\\.\\pipe\\");
    strcat(fifo_response, r.fifo_response);
```

**Continua**



```

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);
}

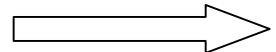
my_pipe = CreateFile("\\\\.\\pipe\\serv", GENERIC_WRITE, 0, NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                    NULL);

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

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

```

**Continua**



```
CloseHandle(my_pipe);
```

```
if (!ReadFile(response_pipe,
```

```
    response,
```

```
    20,
```

```
    &ret,
```

```
    NULL)) {
```

```
    printf("Errore nella chiamata ReadFile\n");
```

```
    ExitProcess(1);
```

```
}
```

```
printf("risposta = %s\n", response);
```

```
CloseHandle(response_pipe);
```

```
return(0);
```

```
}
```