

# Sistemi di Calcolo - Secondo modulo (SC2)

## Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica  
A.A. 2016-2017

**Prof. Roberto Baldoni**

# Pipes

# Contenuti

---

## **Pipes e named pipes:**

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

# Concetti di base sulle PIPE (1/2)

- 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, le 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 una PIPE piena

## Concetti di base sulle PIPE (2/2)

- i processi che usano una PIPE devono essere “relazionati”
- le named PIPE (FIFO) permettono la comunicazione anche tra processi non relazionati
- in sistemi UNIX l’uso delle PIPE avviene attraverso la nozione di descrittore

# PIPE nei Sistemi UNIX

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

---

**Descrizione**    invoca la creazione di una PIPE

---

**Argomenti**    fd: puntatore ad un buffer di due interi

- fd[0] : descrittore di lettura dalla PIPE
- fd[1]: descrittore di scrittura sulla PIPE

---

**Restituzione**    -1 in caso di fallimento, 0 altrimenti

---

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

---

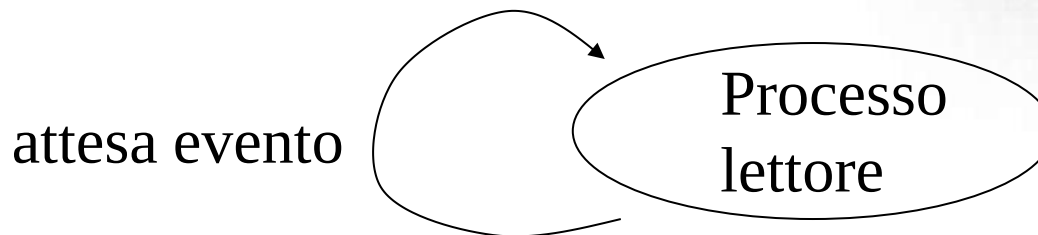
- le PIPE non sono dispositivi fisici, ma logici, pertanto viene spontaneo chiedersi come un processo sia in grado di vedere la fine di un “file” su una PIPE
- per convenzione, ciò avviene quando tutti i processi scrittori che condividevano il descrittore `fd[1]` lo hanno chiuso
- in questo caso la chiamata `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

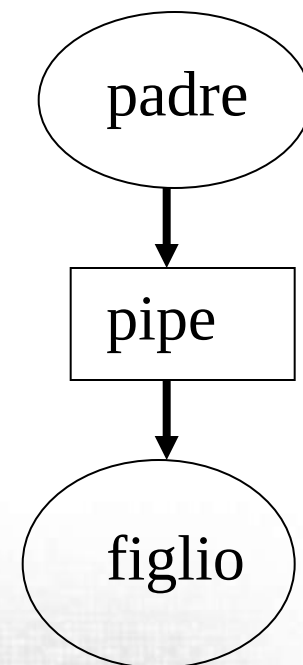
```
#include <stdio.h>
#define Errore_(x) { puts(x); exit(1); }

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: nome della FIFO da creare
- 2) mode: specifica i permessi di accesso alla FIFO

---

**Restituzione** -1 in caso di fallimento, 0 altrimenti

La rimozione di una FIFO dal file system avviene mediante la chiamata di sistema `unlink()`

# Avvertenze

---

- normalmente, l'apertura di una FIFO è bloccante, nel senso che il processo che tenti 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 il “segnale `SIGPIPE`” da parte del sistema operativo

# Un esempio: client/server tramite FIFO

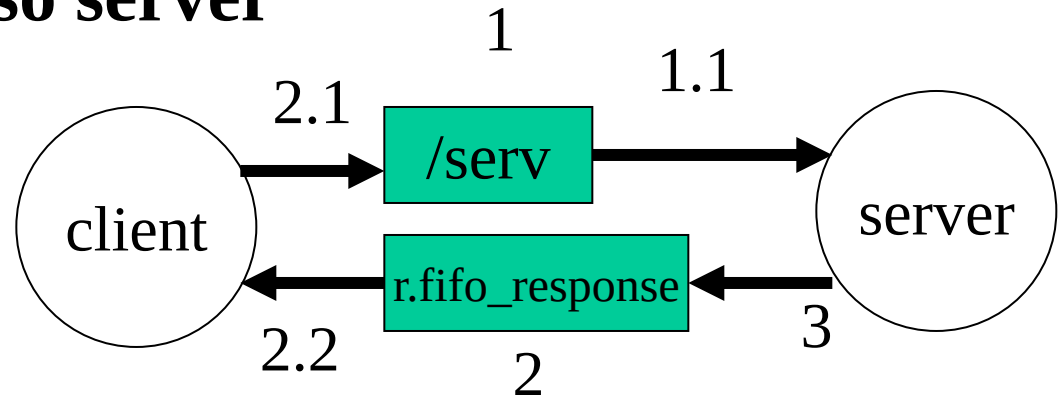
## Processo server

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



1

# .....continua (Processo Server)

```
fd = open("/serv", O_RDONLY);
```

```
while(1) {
```

```
    ret = read(fd, &r, sizeof(request));
```

1.1

```
    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);
```

3

```
            write(fdc, response, 20);
```

```
            close(fdc);
```

```
            exit(0);
```

```
        } /* end if (pid == 0) */
```

```
    } /* end if (ret != 0) */
```

```
}
```

```
}
```

# Processo client

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

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, 0666);
    if ( ret == -1 ) {
        printf("\n servente sovraccarico - riprovare \n");
        exit(1);
    }
}
```

2

## .....continua (Processo Client)

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

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

2.1

2.2

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

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

```
}
```