# Capitolo 1

# FIFO (First-In, First-Out) o Named Pipes

## 1.1 Introduzione alle FIFO

Le **FIFO** (acronimo di **First-In, First-Out**) sono un meccanismo di comunicazione inter-processo (**IPC**) in ambiente UNIX-like. Sono note anche come **Named Pipes** (Pipe con Nome).

A differenza delle **Pipe anonime** (viste in precedenza, create con `pipe()` e utilizzabili solo tra processi con una relazione di parentela, es. padre/figlio), le FIFO:

- **Hanno un nome** nel filesystem, appaiono come un file speciale (non contengono dati, ma rappresentano il canale di comunicazione).
- Possono essere utilizzate per la comunicazione tra **processi non correlati** (ad esempio, due programmi eseguiti da shell diverse).
- Persistono nel filesystem fino a quando non vengono esplicitamente rimosse, anche dopo che i processi che le utilizzavano sono terminati (a differenza delle pipe anonime che svaniscono).

### 1.1.1 Creazione di una FIFO: `mkfifo()`

La creazione di una FIFO avviene tramite la system call `mkfifo()`.

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

Listing 1.1: Sintassi di mkfifo()

- `pathname`: è il percorso e il nome che la FIFO assumerà nel filesystem (es. `"/tmp/myfifo"`).
- `mode`: specifica i permessi di accesso, proprio come nella creazione di un file (es. `0666` per lettura/scrittura a tutti).

La funzione restituisce 0 in caso di successo, e $-1$ in caso di errore (es. se la FIFO esiste già o non si hanno i permessi).

### 1.1.2 Apertura e Utilizzo di una FIFO: `open()`, `read()`, `write()`

Una volta creata, una FIFO viene aperta e utilizzata come un normale file con le system call `open()`, `read()` e `write()`.

```
#include <fcntl.h>
#include <unistd.h>
int fd_w = open("nome_fifo", O_WRONLY); // Apertura per
    Scrittura
int fd_r = open("nome_fifo", O_RDONLY);  // Apertura per
    Lettura

// Scrittura
write(fd_w, buffer, sizeof(buffer));

// Lettura
read(fd_r, buffer, sizeof(buffer));
```

Listing 1.2: Apertura e I/O su FIFO

**Comportamento Bloccante (*Blocking* Behavior)**:

- L'`open()` in **sola lettura** (`O_RDONLY`) si blocca finché la FIFO non viene aperta in scrittura da almeno un altro processo.
- L'`open()` in **sola scrittura** (`O_WRONLY`) si blocca finché la FIFO non viene aperta in lettura da almeno un altro processo.

Questo assicura che ci sia sempre un mittente e un destinatario pronti prima che la comunicazione inizi. Questo comportamento può essere modificato aggiungendo il flag `O_NONBLOCK`.

### 1.1.3 Chiusura ed Eliminazione: `close()` e `unlink()`

- La chiusura della FIFO avviene con la normale `close()`.
- Poiché la FIFO persiste come file nel filesystem, deve essere rimossa esplicitamente una volta terminato l'utilizzo, tipicamente dal processo che l'ha creata (il server), usando la system call `unlink()`.

```
1 #include <unistd.h>
2 unlink("nome_fifo"); // Rimuove il "file" FIFO dal
    filesystem
```

Listing 1.3: Eliminazione di una FIFO

# Capitolo 2

# Segnali (Signals)

## 2.1 Introduzione ai Segnali

I **Segnali** sono un meccanismo di comunicazione asincrona tra i processi e il sistema operativo. Essenzialmente, sono delle brevi notifiche che vengono inviate a un processo per informarlo che è successo un evento.

Esempi di eventi che generano segnali:

- L'utente preme `CTRL+C` (`SIGINT`).
- Un processo tenta una divisione per zero (`SIGFPE`).
- Un figlio termina (`SIGCHLD`).
- Un altro processo invia un segnale specifico (`kill`).

### 2.1.1 Le Azioni di Default

Quando un processo riceve un segnale, il sistema operativo esegue una delle seguenti azioni di **default**:

1. **Terminazione**: il processo viene interrotto (es. `SIGINT`, `SIGTERM`).
2. **Terminazione con Core Dump**: il processo termina e salva un'immagine della memoria (*core file*) per il debugging (es. `SIGSEGV`).
3. **Ignoramento**: il segnale viene ignorato (es. `SIGCHLD`).
4. **Stop**: il processo viene sospeso (es. `SIGSTOP`, `SIGTSTP`).
5. **Continua**: il processo sospeso riprende l'esecuzione (es. `SIGCONT`).

### 2.1.2 Gestione dei Segnali: `sigaction()`

La funzione `signal()` è obsoleta e meno robusta; il metodo moderno e raccomandato per la gestione dei segnali è `sigaction()`.

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
    struct sigaction *oldact);
```

Listing 2.1: Sintassi di sigaction()

- `signum`: il numero del segnale da gestire (es. `SIGINT`).
- `act`: puntatore a una struttura `sigaction` che definisce l'azione da intraprendere per il segnale.
- `oldact`: (opzionale) puntatore per salvare le impostazioni precedenti.

La struttura `struct sigaction` contiene principalmente:

- `sa_handler`: l'indirizzo della funzione **handler** (gestore del segnale) da eseguire, oppure una delle costanti `SIG_DFL` (azione di default) o `SIG_IGN` (ignora il segnale).
- `sa_mask`: una maschera di segnali che specifica quali segnali devono essere bloccati (*mascherati*) mentre l'handler è in esecuzione, per prevenire interruzioni.

```
void handler_funzione(int signo) {
    printf("Ricevuto segnale %d\n", signo);
    // Logica di gestione del segnale
}

struct sigaction sa;
sa.sa_handler = handler_funzione;
sigemptyset(&sa.sa_mask); // Inizializza la maschera a
    vuota
sa.sa_flags = 0; // Nessun flag speciale

if (sigaction(SIGINT, &sa, NULL) == -1) {
    perror("sigaction");
}
```

Listing 2.2: Esempio di Configurazione Handler

### 2.1.3 Invio di Segnali: `kill()` e `raise()`

- `kill()`: Invia un segnale a un processo specifico.

```
1 int kill(pid_t pid, int sig); // pid: PID del
    destinatario, sig: segnale da inviare
```

- `raise()`: Permette a un processo di inviare un segnale a sé stesso (equivalente a `kill(getpid(), sig)`).

# Capitolo 3

# Esercizi

## 3.1 Esercizi sulle FIFO

1. **Produttore e Consumatore Semplice (Base)** Scrivere due programmi C, `produttore.c` e `consumatore.c`.
   - Il Produttore crea una FIFO chiamata `"/tmp/dati_fifo"` con permessi `0666`.
   - Il Produttore invia 10 stringhe (*es. "Messaggio X"*) al Consumatore, attendendo 1 secondo tra un invio e l'altro.
   - Il Consumatore apre la FIFO, legge le 10 stringhe e le stampa a video.
   - Il Produttore, dopo aver terminato, chiude la FIFO e usa `unlink()` per rimuoverla.
   - **Istruzioni per l'alunno**: Eseguire i due programmi in due terminali separati.
2. **Passaggio di Strutture Dati** Modificare l'esercizio precedente per passare una **struttura** (es. `struct record {int id; float valore;}`), non solo stringhe. Il Produttore deve inviare 5 record generati casualmente. Il Consumatore riceve e stampa i campi di ciascun record.
3. **Chat Unidirezionale (Server/Client con Blocco)** Scrivere `server.c` e `client.c`.
   - Il Server crea una FIFO chiamata `"/tmp/chat_server"`.

- Il Server si blocca in attesa di un messaggio.
- Il Client si connette e invia ripetutamente righe di testo digitate dall'utente.
- Il Server legge i messaggi e li stampa a video.
- Discutere il comportamento bloccante della `open()` in questo scenario e cosa succede se si esegue il Server prima del Client.

4. **Comunicazione Half-Duplex (Due FIFO)** Scrivere due programmi `processoA.c` e `processoB.c` che comunicano bidirezionalmente utilizzando **due FIFO separate**: `"/tmp/A_to_B"` e `"/tmp/B_to_A"`.
   - A invia il messaggio `"PING"` a B.
   - B riceve `"PING"`, stampa il messaggio e risponde con `"PONG"` ad A.
   - A riceve `"PONG"`, stampa il messaggio e ripete il ciclo 5 volte.
5. **Gestione degli Errori e `O_NONBLOCK`** Modificare il programma `consumatore.c` dell'esercizio 1 per aprire la FIFO con il flag `O_RDONLY | O_NONBLOCK`.
   - Cosa succede se si tenta di leggere dalla FIFO quando è vuota? (Si aspetta o restituisce errore?)
   - Implementare un ciclo di lettura non bloccante in cui il Consumatore tenta di leggere e, se `read()` fallisce con `EAGAIN` o `EWOULDBLOCK`, stampa un messaggio di attesa e `sleep()` per 1 secondo.

## 3.2 Esercizi sui Segnali

1. **Gestore di `SIGINT` (Interruzione)** Scrivere un programma C che implementa un **handler** per il segnale `SIGINT` (generato con `CTRL+C`). L'handler deve:
   - Stampare il messaggio: "Non mi puoi terminare subito! Ho bisogno di pulire."
   - Incrementare un contatore statico.
   - Se `SIGINT` viene ricevuto per la terza volta, l'handler deve stampare un messaggio di addio e chiamare `exit(0)`.
2. **Segnali Utente (`SIGUSR1` e `kill`)** Scrivere un programma che:

- Stampa il proprio PID (`getpid()`).
- Implementa un handler per `SIGUSR1` che stampa "Segnale Utente 1 ricevuto!".
- Entra in un ciclo infinito con `pause()` (che attende un segnale).
- **Istruzioni per l'alunno**: Aprire una seconda shell e usare il comando `kill -SIGUSR1 [PID]` per inviare il segnale.

3. **Segnali e Processi Figli (`SIGCHLD` e `wait()`)** Scrivere un programma che:
   - Crea un processo figlio usando `fork()`.
   - Il Padre imposta un handler per `SIGCHLD` (segnale di terminazione figlio).
   - L'handler deve chiamare `wait(NULL)` per raccogliere lo stato del figlio ed evitare processi "zombie", stampando un messaggio di conferma.
   - Il Figlio esegue una `sleep(3)` e poi termina.
   - Il Padre entra in un ciclo infinito. Verificare che l'handler venga eseguito dopo 3 secondi e che non si creino zombie.
4. **Mascheramento dei Segnali con `sa_mask`** Scrivere un programma che definisce un handler per `SIGINT`.
   - Configurare la struttura `sigaction` in modo che, mentre l'handler di `SIGINT` è in esecuzione, il segnale `SIGQUIT` (`CTRL+\`) venga mascherato (bloccato) aggiungendolo a `sa_mask`.
   - L'handler di `SIGINT` deve contenere una `sleep(5)`.
   - **Istruzioni per l'alunno**: Lanciare il programma, premere `CTRL+C` e, durante i 5 secondi di sleep dell'handler, premere `CTRL+\`. Cosa succede al segnale `SIGQUIT`? Spiegare perché.
5. **Segnalazione di Errore (`SIGFPE` e `SIGSEGV`)** Scrivere un programma per:
   - Tentare una **divisione per zero** (genera `SIGFPE`).
   - Tentare di **accedere a un indirizzo nullo** (genera `SIGSEGV`).
   - Modificare il programma per implementare un handler per `SIGFPE` che stampa un avviso e permette l'uscita pulita (senza core dump).

- **Domanda teorica**: È saggio o possibile intercettare `SIGSEGV` e continuare l'esecuzione? Spiegare.