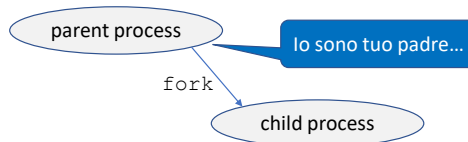


Comunicazione inter-processo

Ing. Giovanni Nardini - University of Pisa - All rights reserved

Comunicazione inter-processo

- Il processo genitore e i processi figli possono scambiarsi informazioni?



- Variabili, puntatori, file descriptor, stream vengono «ereditati» dal processo figlio
- Non realizza una comunicazione completa tra i due processi
- **IPC - InterProcess Communication**
 - Meccanismi per lo scambio di informazioni tra processi

Nel momento della creazione, la memoria del processo figlio è una copia esatta della memoria del processo genitore e include le stesse variabili, gli stessi file descriptor ecc.

Questa caratteristica può essere sfruttata per passare delle informazioni al processo figlio: il processo genitore assegna un valore a una determinata variabile, che può essere letta dal processo figlio dopo la sua creazione.

È comunque un meccanismo uni-direzionale, e che permette lo scambio di informazione solamente alla creazione del processo → ci serve qualcosa di più...

Pipe

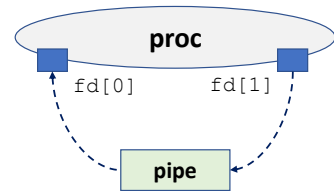
```
#include <unistd.h>
```

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

Restituisce un array di due file descriptor

0 se ok, -1 in caso di errore (setta errno)

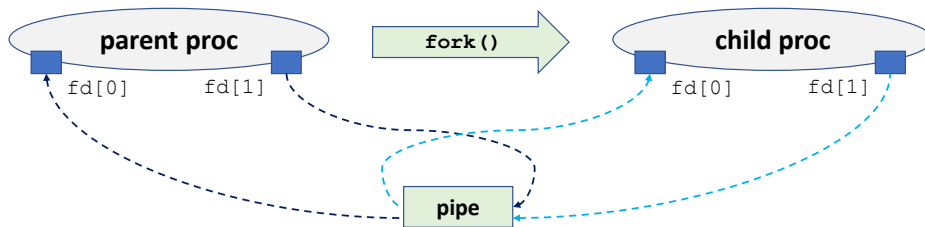
- System call che crea una pipe collegata al processo chiamante
- `fd[0]` aperto in lettura, `fd[1]` aperto in scrittura
 - I dati scritti su `fd[1]` possono essere letti su `fd[0]`
- Scrittura con `write()` e lettura con `read()`
- La pipe ha un buffer di dimensione `PIPE_BUF`
 - Definita in `<limits.h>`, tipicamente 4096 byte



Meccanismo di comunicazione che si realizza tramite file descriptor, sui quali si leggono/scrivono dei byte utilizzando le system call viste per i file descriptor. Una pipe è essenzialmente un buffer allocato in memoria principale (in kernel space) su cui si scrive usando la syscall `write` e si legge usando la syscall `read`. L'allocazione in memoria avviene al momento dell'invocazione della system call `pipe()`.

Pipe e fork

- Con la `fork`, il processo figlio ha una copia dei file descriptor per utilizzare la pipe
- Si è creato un canale di comunicazione tra il processo genitore e il processo figlio



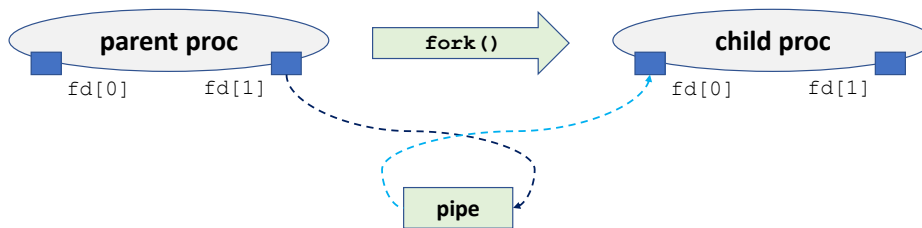
Di per sé, la pipe apre una comunicazione unidirezionale tra due file descriptor di uno stesso processo.

Dato che la `fork` effettua copia la memoria del processo genitore nella memoria del processo figlio, anche quest'ultimo ha una copia dei file descriptor da cui poter/leggere scrivere e comunicare con il processo genitore.

Ciascuno dei due processi ha un file descriptor per leggere dalla pipe e un file descriptor per scrivere sulla pipe.

Direzione della pipe

- Dopo la `fork`, si deve decidere la direzione della comunicazione
- Genitore → Figlio
 - Processo genitore chiude `fd[0]`, processo figlio chiude `fd[1]`
- Figlio → Genitore
 - Processo genitore chiude `fd[1]`, processo figlio chiude `fd[0]`



- Quando tutti i file descriptor vengono chiusi, la pipe viene eliminata dal kernel

È raccomandato decidere quale direzione di comunicazione si vuole sfruttare (maggiori dettagli nelle prossime slide)
Se volessi realizzare una comunicazione full-duplex (nelle due direzioni), devo aprire due pipe.

Pipe: esempio

```
#include ...
int main() {
    ...
    int fd[2];
    if (pipe(fd) < 0) {
        /* error */
    }
    if ( (pid = fork()) < 0) {
        /* error */
    }
    else if (pid > 0) {
        close(fd[0]);
        write(fd[1], "Io sono tuo padre\n", 18);
        close(fd[1]);
    }
    else {
        close(fd[1]);
        int n = read(fd[0], buff, BUFFLEN);
        write(STDOUT_FILENO, buff, n);
        close(fd[0]);
    }
    return 0;
}
```

dichiarazione dei due file descriptor

creazione della pipe

processo genitore chiude il file descriptor per la lettura

processo figlio chiude il file descriptor per la scrittura


Casi particolari


- Tentativo di scrittura su una pipe il cui buffer è **pieno**
 - `write()` si blocca in attesa che l'altro processo legga qualche dato
- Tentativo di lettura su una pipe il cui buffer è **vuoto**
 - `read()` si blocca in attesa che l'altro processo scriva qualche dato

Casi particolari

- Tentativo di lettura su una pipe il cui lato di **scrittura** è stato **chiuso**
 - `read()` restituisce `-1`
- Tentativo di scrittura su una pipe il cui lato di **lettura** è stato **chiuso**
 - Genera un segnale `SIGPIPE`
 - Di default, `SIGPIPE` fa terminare il programma
 - Se il segnale viene ignorato, o gestito senza far terminare il programma, la `write()` restituisce `-1` e imposta `errno=EPIPE`

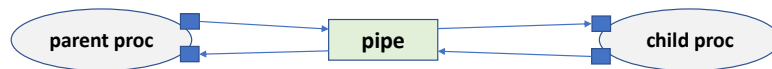
```
...
else if (pid > 0) {
    close(fd[0]);
    write(fd[1], "Io sono tuo padre\n", 18);
    close(fd[1]);
}
else {
    close(fd[0]);
    ...
}
```

 `SIGPIPE`

 processo figlio chiude il file descriptor per la lettura

Problemi di una pipe «bi-direzionale»

- Sincronizzazione



- Chiusura dalla pipe

```
...
else if (pid > 0) {
    close(fd[0]);
    while ( write(fd[1], buff, BUFFLEN) > 0);
    close(fd[1]);
}
else {
    close(fd[1]);
    ...
    close(fd[0]);
}
```

Provoca SIGPIPE sul genitore

```
...
else if (pid > 0) {
    while ( write(fd[1], buff, BUFFLEN) > 0);
    close(fd[1]);
}
else {
    ...
    close(fd[0]);
}
```

Non provoca SIGPIPE sul genitore

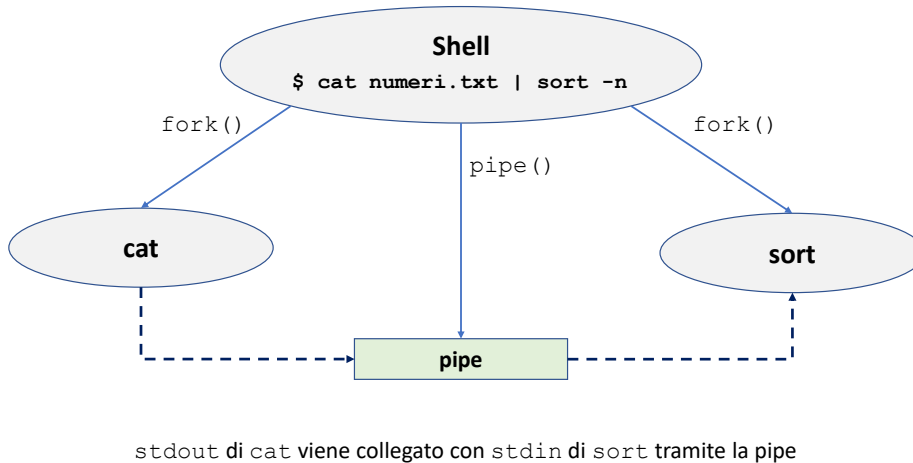
Banalmente, se entrambi processi genitore e figlio potessero leggere e scrivere sulla pipe, ci vorrebbe un meccanismo per sincronizzarli: le scritture di genitore e figlio potrebbero mescolarsi e non ci sarebbe modo di distinguerle.

C'è anche un'altra motivazione: assumiamo che si voglia realizzare una comunicazione dal genitore verso il figlio, in cui il genitore usa un ciclo while infinito per inviare messaggi al processo figlio fino a che il processo figlio chiude il lato di lettura e termina.

Nel primo caso, appena il figlio chiude il file descriptor, la write del genitore genera SIGPIPE perché la pipe non ha più alcun fd di lettura aperto: a seguito di SIGPIPE il processo genitore interrompere il ciclo infinito.

Nel secondo caso, anche se il figlio chiude il file descriptor, la pipe ha ancora un fd di lettura aperto (quello non chiuso del genitore), perciò la write del genitore non genera SIGPIPE e il processo rimane bloccato.

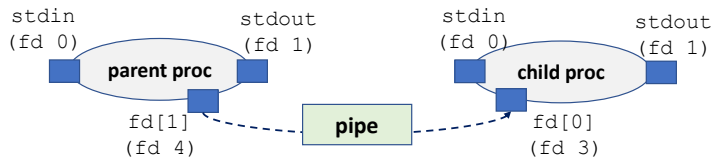
Comando pipe



Il comando `|` della shell usa il meccanismo delle pipe: possibile perché i due processi sono entrambi figli del processo shell che li lancia. In più, mette in comunicazione la pipe con i fd dello stdout e dello stdin dei due processi.

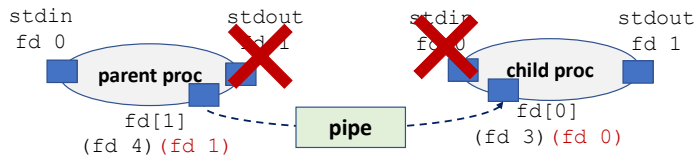
Redirezione di stdin e stdout

Uso comune della pipe → Collegare stdin/stdout di un processo figlio al processo padre



`dup2 (fd[1], STDOUT_FILENO);`

`dup2 (fd[0], STDIN_FILENO);`



? man dup2

L'invocazione della system call `dup2` provoca una modifica della tabella dei file descriptor del processo.

Nel processo genitore, il puntatore del file descriptor 1 (stdout) viene fatto puntare alla stessa riga della kernel file table del file descriptor 4 (fd[1]). Da quel momento in avanti, ogni scrittura su stdout è come se venisse fatta su fd[1] (ovvero sulla pipe).

Lo stesso meccanismo avviene per il processo figlio con lo stdin.

Redirezione di stdin e stdout

Uso comune della pipe → Collegare `stdin/stdout` di un processo figlio al processo padre

- Operazioni da eseguire:
 1. Creare la pipe con system call `pipe()`
 2. Creare il processo figlio con system call `fork()`
 3. Chiudere i file descriptor relativi alla direzione di comunicazione che non ci interessa
 4. Sul figlio, collegare `stdin/stdout` al rispettivo file descriptor della pipe
 - Usando syscall `dup2()` -man `dup2`
 5. Leggere/scrivere dalla/sulla pipe utilizzando le primitive per unbuffered I/O
 - Dobbiamo gestire allocazione e dimensionamento del buffer
 6. Attendere che il processo figlio abbia terminato, prima di terminare il processo genitore, usando `wait()`
- Esistono funzioni della libreria standard del C che fanno tutte queste cose al posto nostro
 - Cosa ricorda?

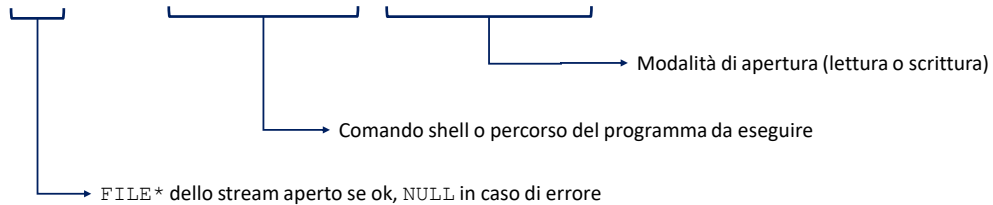


man `dup2`

popen

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

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



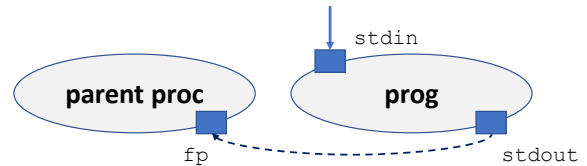
- Crea un processo che esegue il comando o il file eseguibile specificato dal primo argomento
 - `fork + exec`
- Restituisce uno stream per scambiare dati con il nuovo processo
 - Utilizzando funzioni per l'I/O standard

Con una sola funzione, posso creare un processo che esegue il programma specificato come primo argomento, e aprire una pipe tra i due processi collegando gli stream `stdin` e `stdout` (in una delle due direzioni).

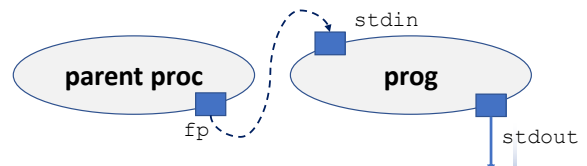
popen

- Argomento `type` specifica la direzione
 - Se `type=="r"`, apre uno stream in lettura collegato a `stdout` del processo creato
 - Se `type=="w"`, apre uno stream in scrittura collegato a `stdin` del processo creato

```
FILE* fp = popen("./prog", "r");
```



```
FILE* fp = popen("./prog", "w");
```

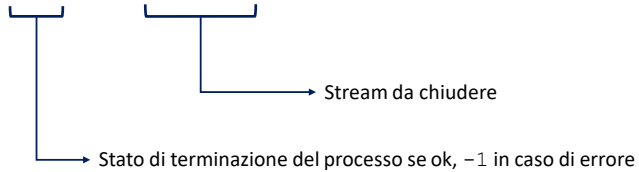


Il processo genitore può usare `fp` come un qualunque altro stream, leggendo/scrivendo per linea/carattere/formattato (vedere slide su standard I/O)

pclose

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

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



- Aspetta la terminazione del processo figlio creato con la `popen` e collegato allo stream `fp`
 - come `wait()` / `waitpid()`
- Chiude lo stream

Serve per sincronizzare il genitore col figlio, aspettandone la terminazione e recuperandone il codice di terminazione (come `wait`).

FIFO

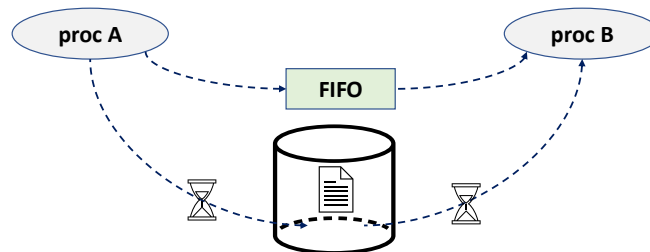
- Una **FIFO** è una pipe che permette la comunicazione tra processi “non imparentati”
- Creazione di una FIFO da shell: `$ mkfifo <fifo_name>`
- Visibile come un file all’interno del file system, di tipo FIFO (cosa mostra `ls -l?`)
- Un processo apre, legge e scrive su una FIFO con le system call per l’unbuffered I/O
 - `open, read, write, close`
 - Basta che ne conosca il percorso (ovvero, il suo nome) → anche note come *named pipe*
- Una FIFO *non* è “seek-abile”
 - Non si può usare la `lseek`
 - Uno o più processi scrivono sulla FIFO, uno o più processi leggono dalla FIFO nello stesso ordine
→ FIFO = First In First Out

Una limitazione del meccanismo delle pipe è che permette la comunicazione solo tra processi con un antenato in comune: tipicamente, genitore-figlio.

La FIFO si può creare direttamente dalla shell, quindi è un oggetto che esiste indipendentemente dall’esistenza di processi che la stanno utilizzando. Creando una FIFO dalla shell appare il suo nome nell’elenco dei file della directory (digitando `ls`). Tuttavia, non ho creato nessun file nella memoria secondaria, ho solo creato un nome che può essere riferito dai processi che la vorranno utilizzare.

FIFO vs file regolari

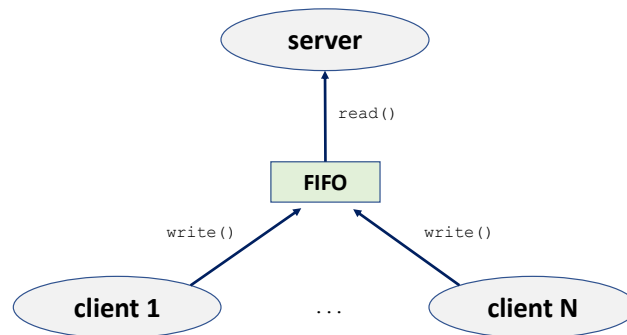
- Una FIFO risiede sulla memoria principale, mentre un file su memoria secondaria
 - Il buffer FIFO viene allocato in memoria principale quando un processo lo apre
 - Quando tutti i processi che utilizzano la FIFO terminano, la FIFO viene eliminata dal kernel
 -
- Quando un processo scrive sulla FIFO
 - i dati vengono memorizzati all'interno di un buffer in memoria principale
 - un secondo processo legge i dati direttamente da quel buffer



Se lo posso aprire e usare come un file regolare, perché non usare direttamente un file regolare per far “parlare” due processi?

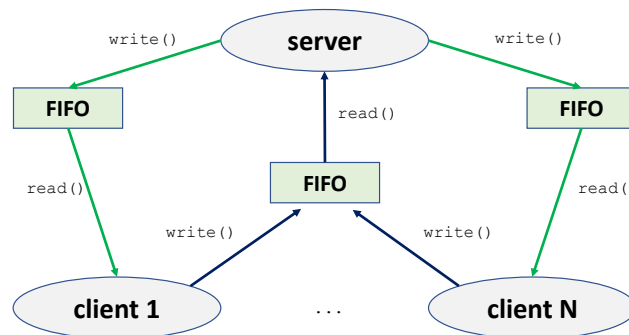
Implementazione di un servizio client-server

- N processi (client) vogliono inviare messaggi a un processo (server)
- Il nome della FIFO deve essere noto a tutti i processi in gioco



Implementazione di un servizio client-server

- Per rispondere alle richieste, il server deve aprire una nuova FIFO per ogni client
 - `int mkfifo(const char* path, mode_t mode);`
- Come accordarsi sul nome delle nuove FIFO?



Per rispondere, il server non può usare la stessa FIFO altrimenti i messaggi si confonderebbero con quelli inviati dai client. Inoltre, qualsiasi degli N client potrebbe leggere quella risposta.

Allora il server deve creare una FIFO dedicata a ogni processo client a cui vuole rispondere. In questo caso, il client e il server devono essere d'accordo sul nome della FIFO. Inoltre, tale nome deve essere univoco e non confondersi con quello delle FIFO per le risposte ad altri client.

Una possibile soluzione è far inviare dal client il proprio PID, in modo che il server possa creare una FIFO dedicata (usando la syscall `mkfifo`) assegnandogli un nome che contiene il PID del processo client.