6. PIPE

La comunicazione tra processi può avvenire:

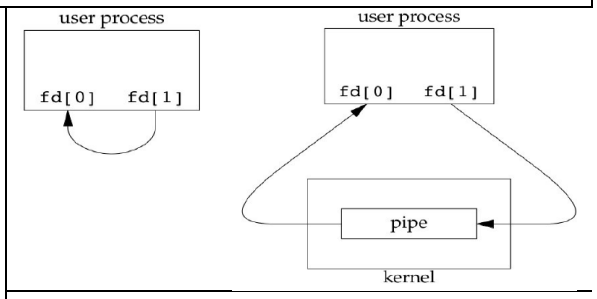
- Passando dei files aperti tramite fork
- Attraverso il filesystem
- Utilizzando le **pipe**
- Utilizzando le **FIFO**
- Utilizzando **IPC di System V**
- Utilizzando **stream** e **socket**

Le **PIPE** sono **half-duplex**, ovvero il flusso di dati è in una sola direzione. Possono essere utilizzate solo tra processi che hanno un antenato in comune.

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Restituisce 0 se OK, -1 in caso di errore.

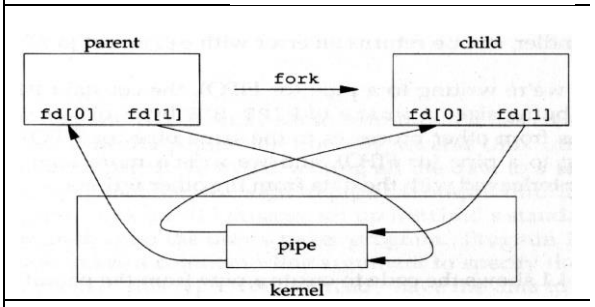
- **filedes[0]** è il file descriptor di un "file" aperto in **lettura**
- **filedes[1]** è il file descriptor di un "file" aperto in **scrittura**
- inoltre, l'output di filedes[1] corrisponde all'input di filedes[0]



L'utilizzo tipico delle pipe è il seguente:

```
int fd[2];
…
pipe(fd);
pid=fork();
…
```

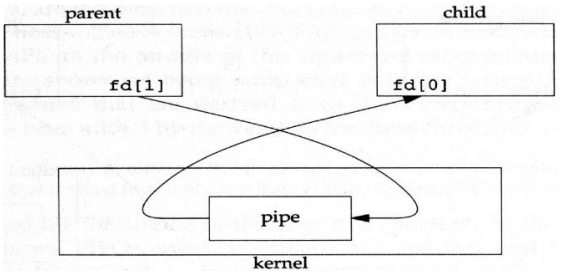
NOTA: il figlio in tal caso eredita dal padre tutto il codice, ma in particolare eredita anche la tabella dei file aperti, quindi sia padre che figlio avranno nella tabella dei file aperti, oltre a STDIN/OUT/ERR, anche 2 file decriptor, uno in cui possono scrivere e uno leggere.



Una delle possibilità dopo la fork è la seguente:

```
if(pid>0) {        // padre
    close(fd[0]);
}else
  if(pid==0) {     // figlio
    close(fd[1]);
}
```

Questo crea un canale dal padre verso il figlio.
NOTA: in questo modo abbiamo creato un canale dal padre verso il figlio.



6.1 I/O SU PIPE

Una volta che è stata creata la pipe e che è stato scelto il verso di comunicazione è possibile utilizzare le funzioni di I/O che lavorano con i file descriptor (*tranne open, creat e lseek*). Una pipe è un canale di comunicazione in cui i dati vengono letti nello stesso ordine in cui vengono scritti, ovvero in ordine FIFO (non si parla più di offset). La semantica di read e write è leggermente modificata.

FUNZIONE WRITE:

Quando la pipe si riempie (la costante ***PIPE_BUF*** specifica la dimensione, questa non è modificabile), la write si blocca fino a che la read non ha rimosso un numero sufficiente di dati. La scrittura è atomica se i dati sono <= PIPE_BUF.

Se il descrittore del file che legge dalla pipe è chiuso, una write genererà un errore (segnale ***SIGPIPE***).

FUNZIONE READ:

Legge i dati dalla pipe nell'ordine in cui sono scritti. Non è possibile rileggere o rimandare indietro i dato letti. Se la pipe è vuota la read si blocca fino a che non vi siano dei dati disponibili. Se il descrittore del file in scrittura è chiuso, la read restituirà un errore dopo aver completato la lettura dei dati.

FUNZIONE CLOSE:

La funzione close sul descrittore del file in scrittura agisce come end-of-file per la read.

La chiusura del descrittore del file in lettura causa un errore nella write.

```
…
int fd[2];
pipe(fd);
pid=fork();
if(pid>0) {         /* padre */
  close(fd[0]);
  write(fd[1],"ciao figlio\n",12);
}else{              /* figlio */
  close(fd[1]);
  n=read(fd[0], line, 12);
  write(STDOUT_FILENO, line, n);
}
…
```

NOTA.1: bisogna decidere a priori il tipo di dato, o struttura dati, usato tra i due processi durante la comunicazione.
NOTA.2: supponiamo parta prima il figlio, una volta che viene fatta la read ma non ha nulla da leggere e in questo caso si blocca, dando vita ad una sorta di *sincronizzazione*.
NOTA.3: se si vuole una comunicazione bidirezionale, bisogna creare 2 pipe, tale operazione va fatta prima di richiamare le fork().
NOTA.4: se si omettono le close(), un processo non saprà mai che l'altro processo ha terminato la scrittura/lettura perché si avrà sempre il fd riferito in entrambi, per questo è meglio chiudere ciò che non servirà.