Comunicazione tra Processi

Le pipe

Comunicazione tra processi

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**

Caratteristiche delle pipe

- Le pipe sono half-duplex
  - Il flusso di dati è in una sola direzione
- Possono essere utilizzate solo tra processi che hanno un antenato in comune

Funzione *pipe*

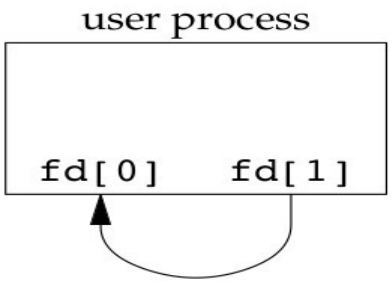
#include <unistd.h>

int pipe(int *filedes[2]*);

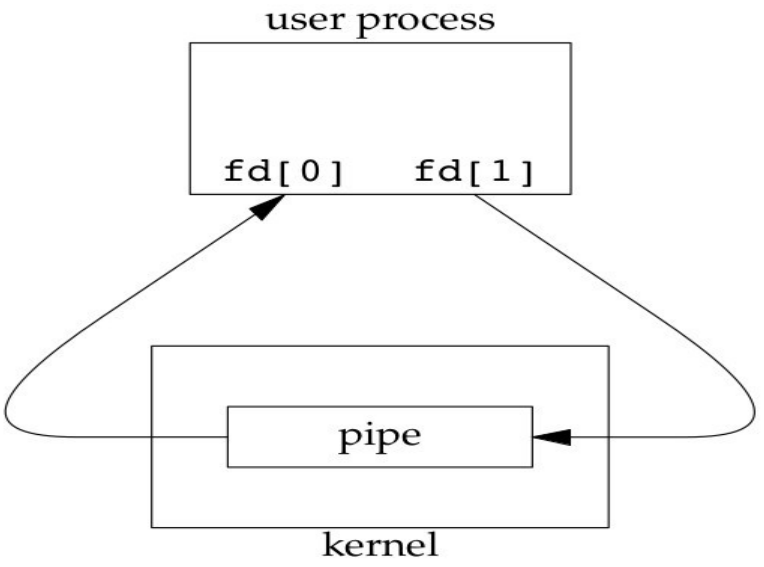
Restituisce 0 se OK, -1 in caso di errore

pipe

- *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]*



or

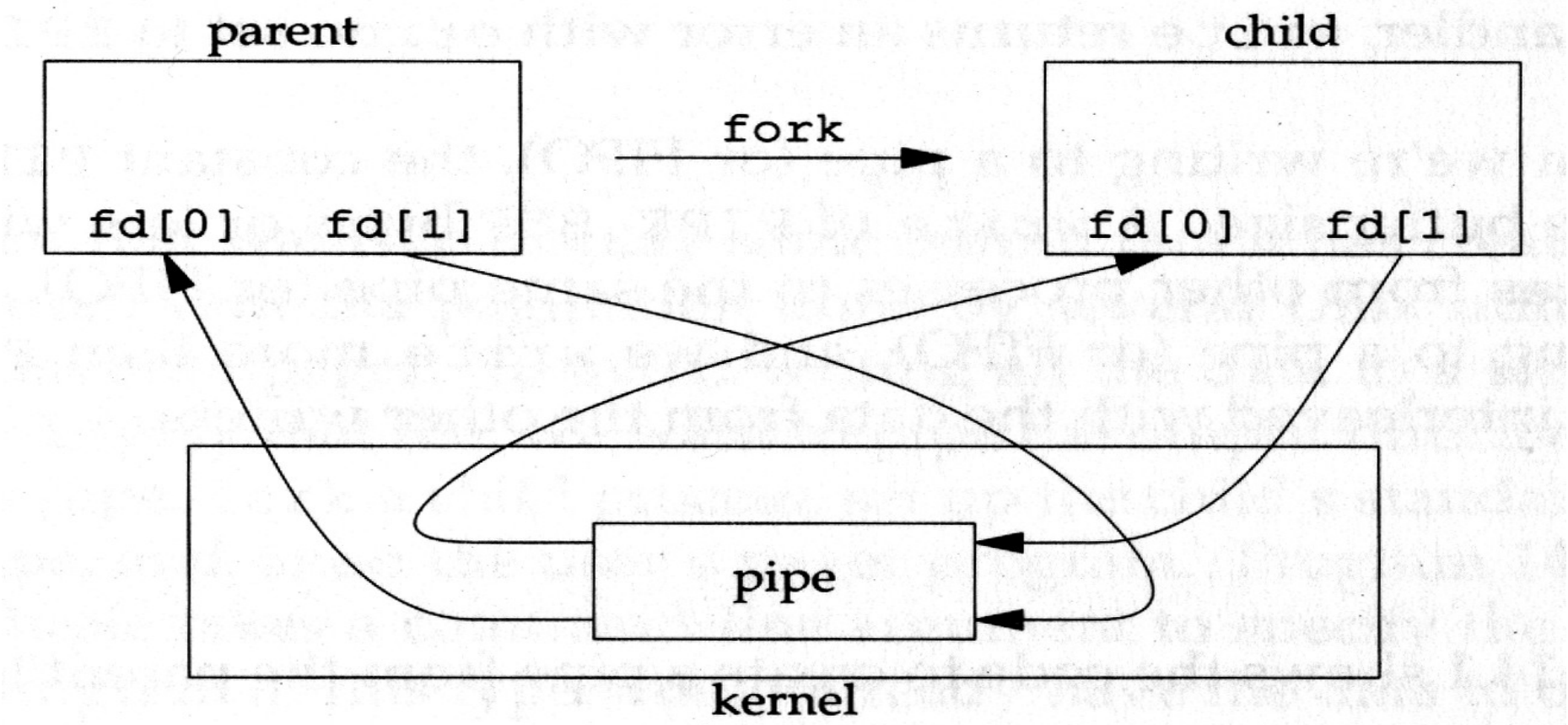


Utilizzo delle pipe (1)

- L’utilizzo tipico delle pipe è il seguente

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

Situazione dopo pipe+fork



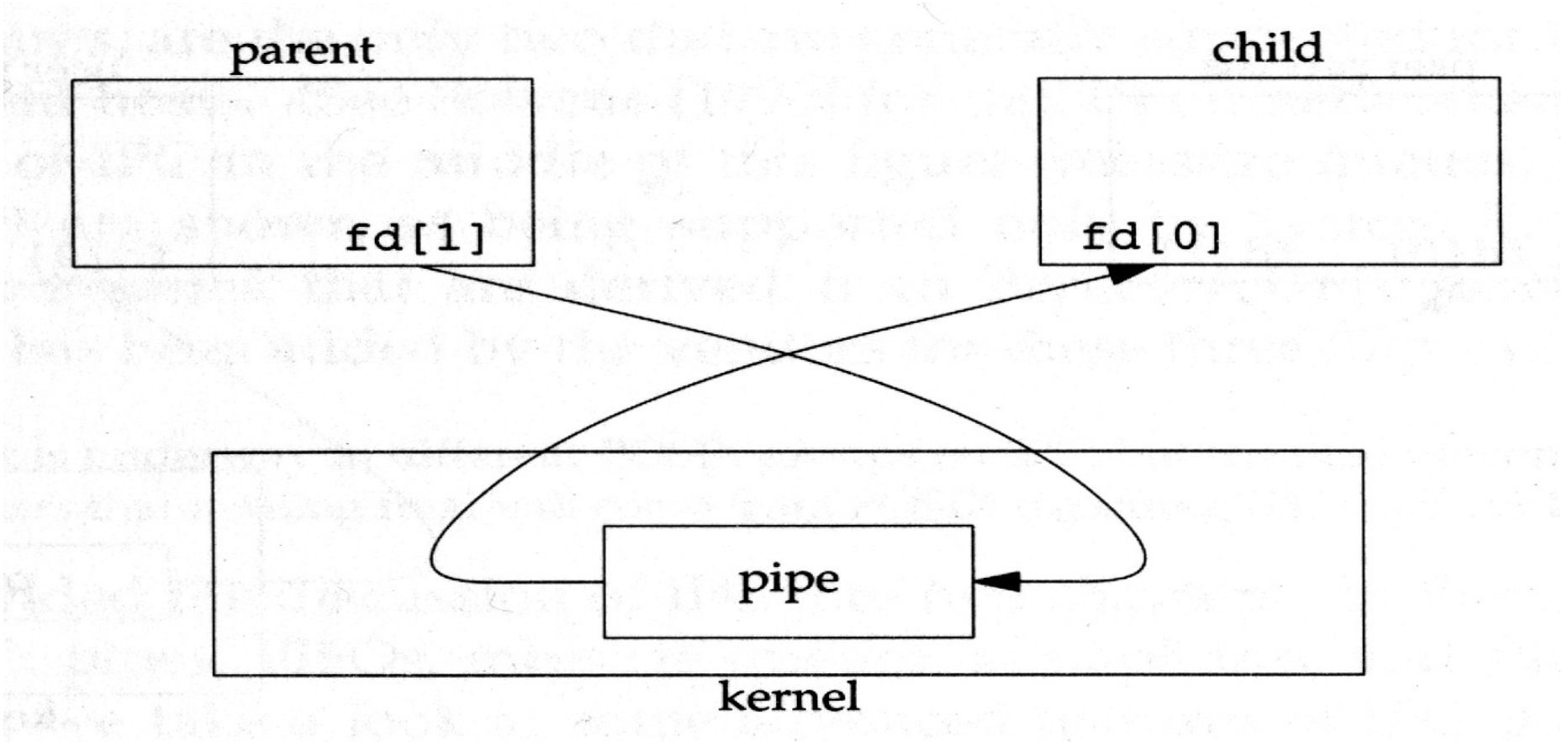
Utilizzo delle pipe (2)

- 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

Pipe da padre a figlio (2)



Utilizzo delle pipe (3)

- 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
- La semantica di ***read*** e ***write*** è leggermente modificata

I/O su una pipe (1)

- Funzione ***write***

Quando la pipe si riempie (la costante PIPE_BUF specifica la dimensione), 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)

I/O su una pipe (2)

- 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.

I/O su una pipe (3)

- 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**.

Esempio

```
...
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);
}
…
```

Utilizzo delle pipe

- una cosa interessante è duplicare i descrittori della pipe su ***standard input*** e ***standard output***
- a questo punto il figlio esegue, con una ***exec***, un programma che può leggere da standard I/O
- ad esempio *more* è uno di questi programmi

Esempio

cat file | more

Come vengono eseguiti i due processi relativi all'esecuzione dei due comandi?

Il comando **cat** ha come input il file e come output STDOUT.

Il comando **more** ha come input un file e come output STDOUT.

Come vengono modificati STDIN e STDOUT di questi comandi?

Esempio

Nel processo **shell** viene creata la **pipe**

pipe(fd);

0	STDIN
1	STDOUT
2	STDERR
3	PIPE_R
4	PIPE_W
5	
6	… … …

Esempio

Dopo l'esecuzione delle **fork**, nel primo figlio viene chiuso il descrittore in lettura

```
close(fd[0]);
```

0	STDIN
1	STDOUT
2	STDERR
3	PIPE_R
4	PIPE_W
5	
6	... ...

Esempio

Successivamente viene chiuso lo standard output

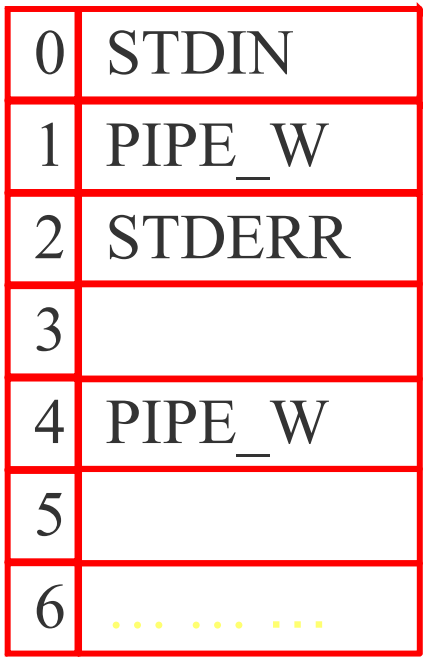
```
close(1);
```

0	STDIN
1	STDOUT
2	STDERR
3	
4	PIPE_W
5	
6	... ...

Esempio

Con la funzione **dup** lo standard output coinciderà con il descrittore del file in scrittura

```
dup(fd[1]);
```



Esempio

Si chiude il descrittore del file in scrittura

```
close(fd[1]);
```

0	STDIN
1	PIPE_W
2	STDERR
3	
4	PIPE_W
5	
6	… … …

Esempio

Si esegue la **exec** del comando **cat**

execlp(“cat”, ...);

0	STDIN
1	PIPE_W
2	STDERR
3	
4	
5	
6	... ...

Esempio

Nel secondo figlio avremo la stessa tabella dei file descriptor

0	STDIN
1	STDOUT
2	STDERR
3	PIPE_R
4	PIPE_W
5	
6	... ...

Esempio

Viene chiuso il descrittore del file in scrittura

```
close(fd[1]);
```

0	STDIN
1	STDOUT
2	STDERR
3	PIPE_R
4	PIPE_W
5	
6	... ...

Esempio

Viene chiuso lo standard input

close(0);

0	STDIN
1	STDOUT
2	STDERR
3	PIPE_R
4	
5	
6	... ...

Esempio

Con la **dup** lo standard input coinciderà con il descrittore del file in lettura

dup(fd[0]);

0	PIPE_R
1	STDOUT
2	STDERR
3	PIPE_R
4	
5	
6	... ...

Esempio

Viene chiuso il descrittore del file in lettura

close(fd[0]);

0	PIPE_R
1	STDOUT
2	STDERR
3	PIPE_R
4	
5	
6	... ...

Esempio

Viene eseguita la **exec** del comando **more**

execlp(“more”, ...);

0	PIPE_R
1	STDOUT
2	STDERR
3	
4	
5	
6	... ...

Esercizio 08_1

Scrivere un programma che:

- prenda in input il nome di un file di testo
- crei due figli che comunicano tramite pipe
- il primo figlio esegue ***cat file*** e manda l'output al fratello tramite la pipe
- il secondo figlio visualizza a video le informazioni ricevute dalla pipe con il comando ***more***

Esercizio 08_2

Un processo P1 crea una pipe e un figlio F1. Un secondo processo P2 comunicherà con P1 tramite un file TEMP. P2 ogni secondo genera un numero casuale da 1 a 100 e lo scrive nel FILE seguito dal proprio pid. P1 dopo 20 secondi dalla creazione del figlio scrive nella pipe il pid di P2, seguito dal numero -1, poi stampa un messaggio sullo schermo e termina la sua esecuzione. Durante questi 20 secondi P1 leggerà i numeri nel file TEMP e scriverà sulla pipe il suo pid con il numero che ha letto. F1 leggerà dalla pipe i pid seguiti dal numero. Se il numero è -1 ucciderà P2 e poi terminerà; altrimenti stamperà al terminale il proprio pid seguito dal numero che ha letto.

Esercizio 08_3

Si supponga di avere un file **ELENCO.TXT** contenente dei record nel formato

Cognome\tNome\n

Scrivere un programma in C che crei il file **ORDINATO.TXT**, versione ordinata del file **ELENCO.TXT**.

Non è possibile modificare il file ELENCO.TXT, non è possibile utilizzare file temporanei.

Per semplicità considerare Cognomi e Nomi di una sola stringa.