Interprocess Communications

Franco Maria Nardini

Pipe

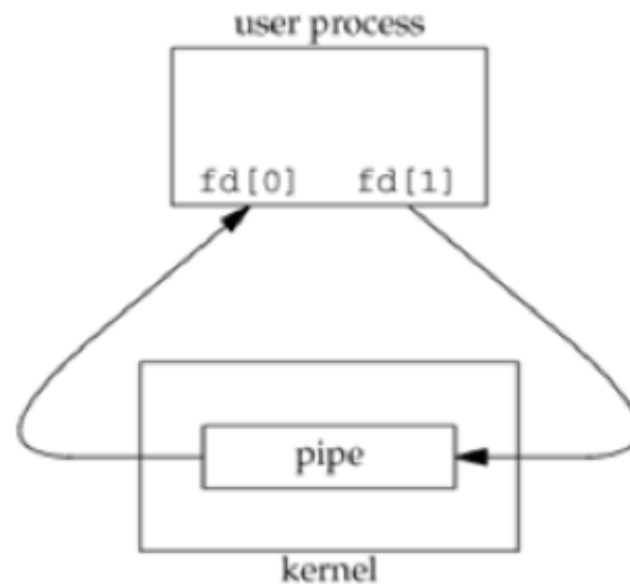
```
cat pippo.txt | less

sort pluto.txt | uniq -c | less
```

- è il metodo più datato di IPC in UNIX
- half-duplex
  - POSIX.1 consente full-duplex
  - half-duplex sempre per massima portabilità
- può esser usata tra processi che hanno ancestor comune

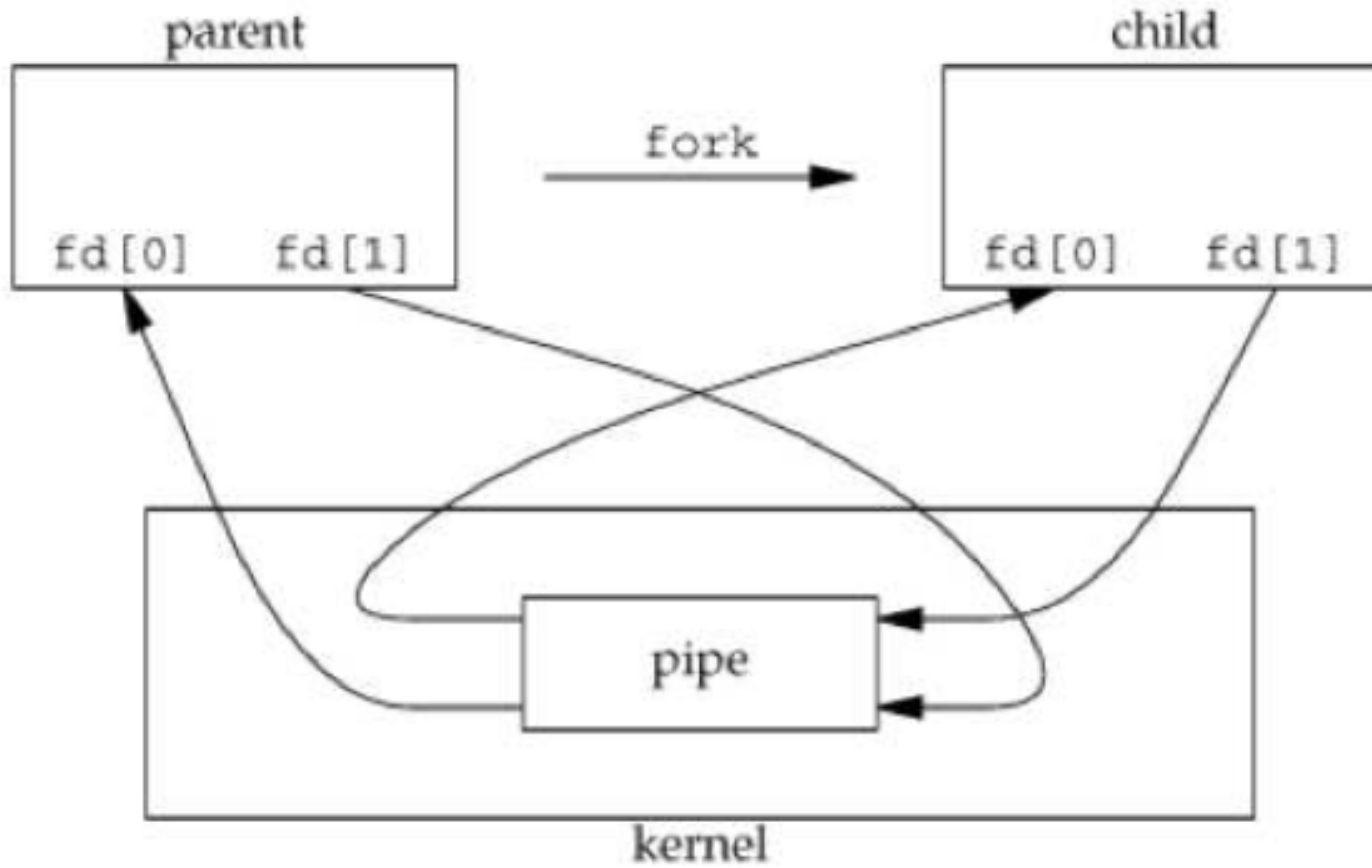
pipe(2)

```
#include <unistd.h>

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

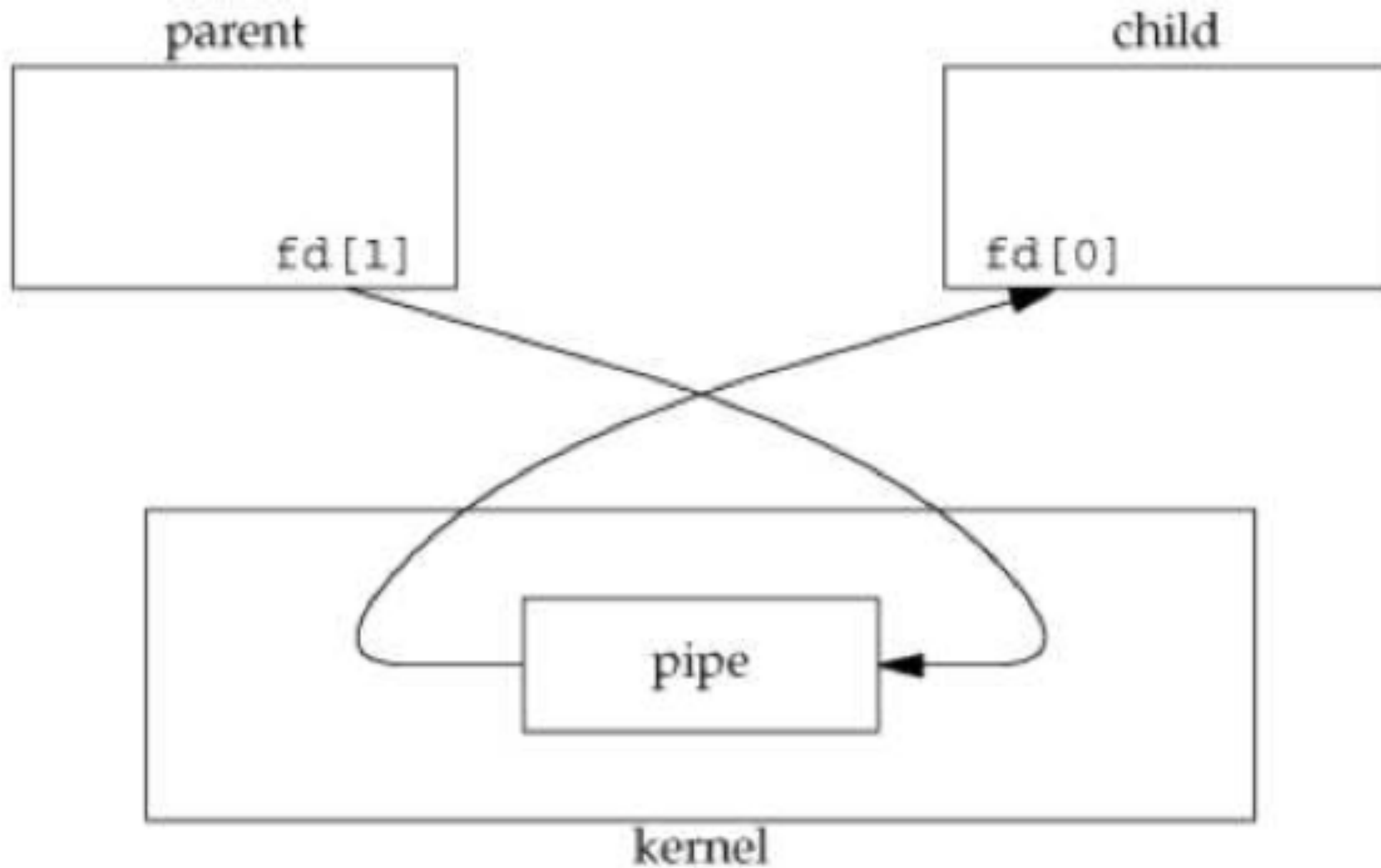
Returns: 0 if OK, -1 otherwise



Pipe



Pipe



Pipe

- Dopo le chiusure:
  - `read(2)` da un pipe il cui *fd[1]* è stato chiuso: ritorna 0 dopo aver letto tutti i dati
  - `write(2)` su un pipe con *fd[0]* chiuso:
    - `SIGPIPE`
    - se segnale ignorato, o catturato con return da signal handler:
      - `write(2)` ritorna -1
      - `errno` settato a `EPIPE`

Pipe

- Quando si scrive su pipe, la costante **`PIPE_BUF`** specifica la dimensione del buffer del kernel associato al pipe

- Se processi multipli scrivono sulla stessa pipe:

  - scritture < **`PIPE_BUF`**: no interleaving con dati di altri processi.

- **`pathconf(2)`**, **`fpathconf(2)`** consentono di determinare **`PIPE_BUF`**

Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/10/pipe1.c

$ cc -Wall ./pipe1.c

$ ./a.out
```

Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/10/pipe2.c

$ cc -Wall ./pipe2.c

$ ./a.out
```

`popen(3)`

- La C standard library ci viene in aiuto! :)
- il lavoro “sporco” visto finora:
    - creare il pipe
    - `fork`are il processo
    - chiudere i terminali non usati del pipe
    - eseguire il comando a shell
    - attendere la terminazione

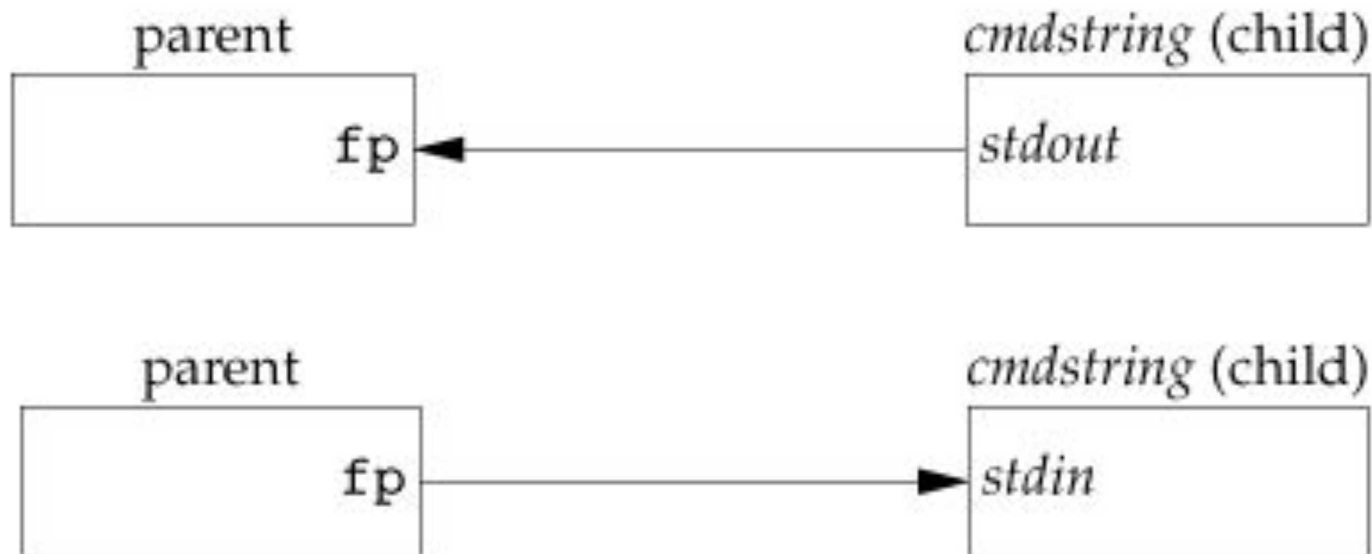
popen(3)

```
#include <stdio.h>

FILE *popen(const char *cmd, const char *type);
                                Returns: file pointer if OK, NULL otherwise

int pclose(FILE *fp);
                                Returns: termination status cmd or -1 on error
```

- storicamente implementata con pipe unidirezionale, ora usa socket o full-duplex pipe (quando disponibile)
- *type*: 'r' oppure 'w' ('r+' per comunicazioni bidirezionali)
- *cmd* passato a `/bin/sh -c`

`popen(3)`

- **`popen(3)`**: **`fork`**, **`exec`** (*cmdstring*) e ritorna uno standard I/O pointer



pclose(3)

- **`pclose(3)`**: chiude lo standard I/O stream, attende la terminazione del comando e ritorna l'exit status della shell
  - se errore di esecuzione della shell: **`exit(127)`**
- esecuzione da Bourne shell (**`/bin/sh -c`** cmd)
  - espansione di caratteri speciali

```
fp = popen("ls *.c", "r");
```

Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/10/popen.c

$ cc -Wall ./popen.c

$ ./a.out
```

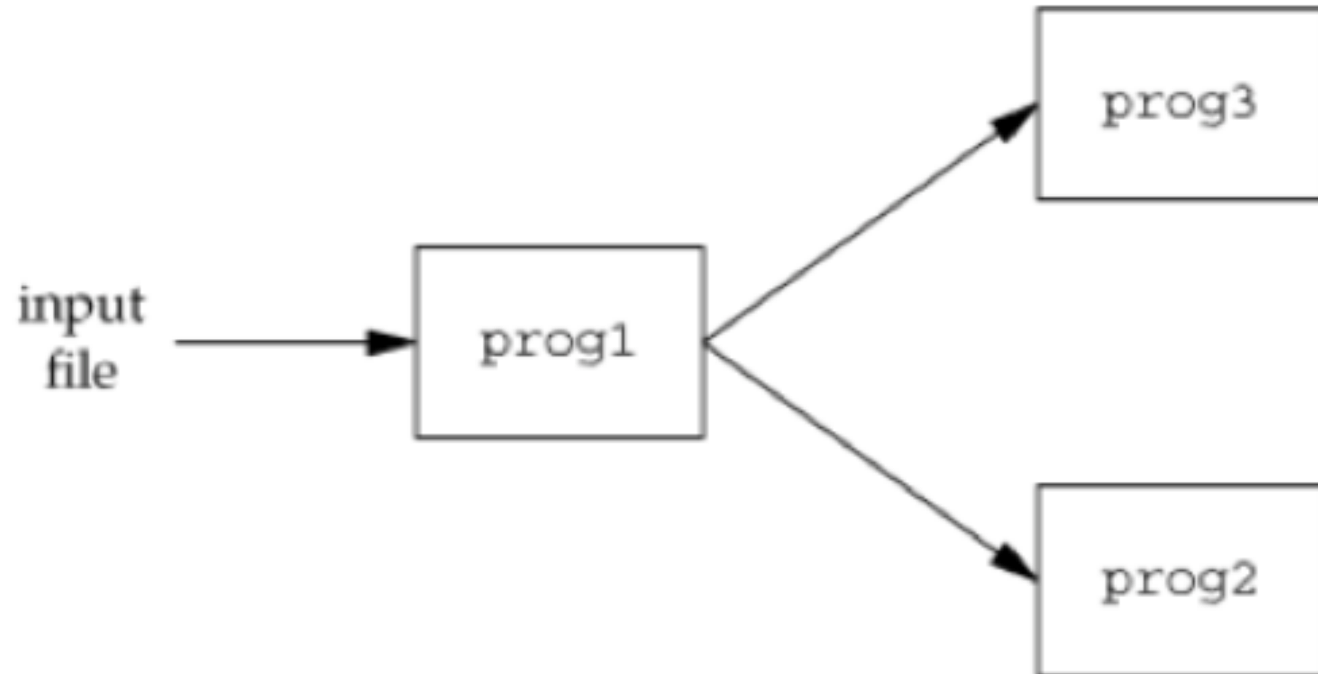
Hands on! :)

- Prendere il file primi100numeri.txt e dividerlo in due:
    - tutte le righe che matchano “77392” in pluto.txt
    - il resto in paperino.txt

Prima soluzione

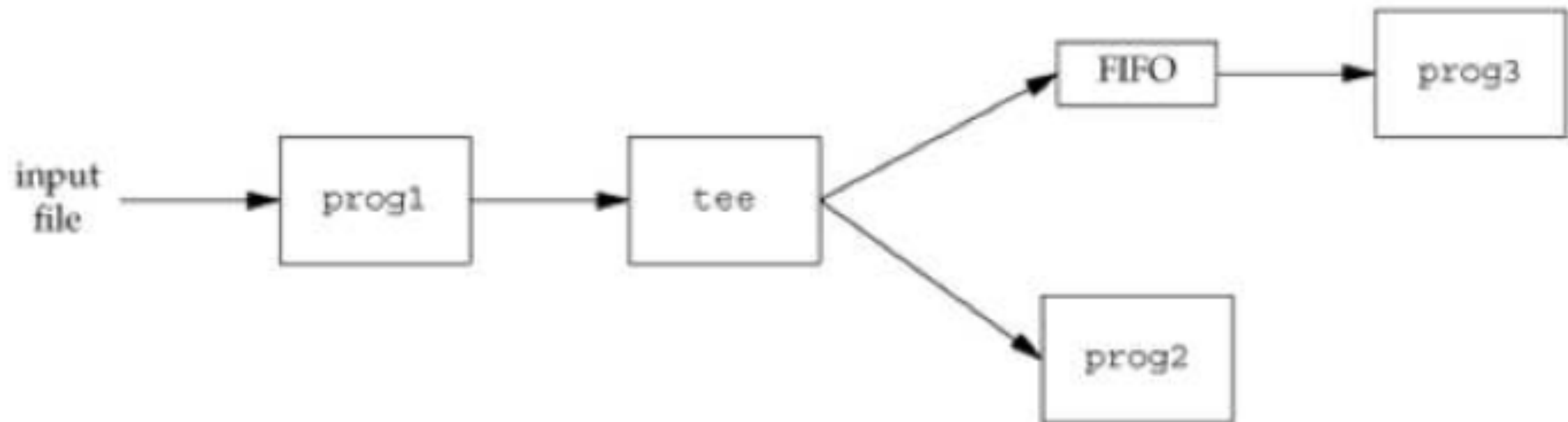
```
$ cat primi100numeri.txt | grep “77392” > pluto.txt

$ cat primi100numeri.txt | grep -v “77392” > paperino.txt
```

Si può far meglio? :)



Si può far meglio? :)

- si, usando `tee(1)` e `mkfifo(1)`




```
$ mkfifo pippo

$ grep 77392 pippo > pluto.txt &

$ cat primi100numeri.txt | tee pippo | grep -v 77392 > paperino.txt
```

FIFOs

- anche conosciute come “named pipes”
- consentono di far comunicare processi scorrelati tra loro
- è un tipo di file (**?**)
  - `st_mode` in `stat`, check con `S_ISFIFO`
- usate dalla shell per duplicare contenuti
  - connessioni non-lineari tra processi (?)
  - nessun file temporaneo!
- usate per comunicazioni client-server

FIFOs

```
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```

Returns: 0 if OK, -1 otherwise

- *mode* stesso di `open(2)`
- consentono l'uso delle normali operazioni di I/O
  - `open(2)`, `read(2)`, `write(2)`, `unlink(2)`

Homework

- Esercizio 9:
  - scrivere esercizio 6 e 7 usando **`pipe(2)`**
- Esercizio 10:
  - scrivere esercizio 6 e 7 usando **`popen(3)`**