# La comunicazione fra processi

***dr. Andrea E. Naimoli***

Università degli Studi di Trento

Dipartimento di Ingegneria e Scienza dell'Informazione

via Sommarive 14

I - 38050 Trento - Povo, Italy

# La comunicazione fra processi: concetti generali e piping

***dr. Andrea E. Naimoli***

Università degli Studi di Trento

Dipartimento di Ingegneria e Scienza dell'Informazione

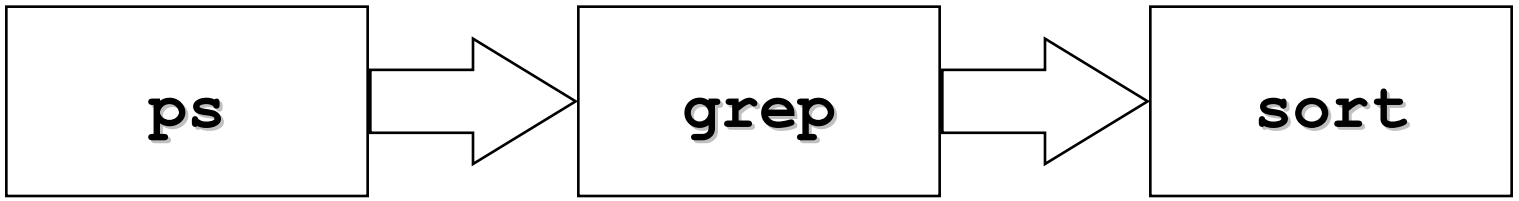
via Sommarive 14

I - 38050 Trento - Povo, Italy

# Comunicazione tradizionale

- Il primo meccanismo di comunicazione tradizionale introdotto nei sistemi Unix sono le *pipe*
- Il suo utilizzo nei sistemi *unix-like* è esteso e quotidiano

**ps | grep | sort**



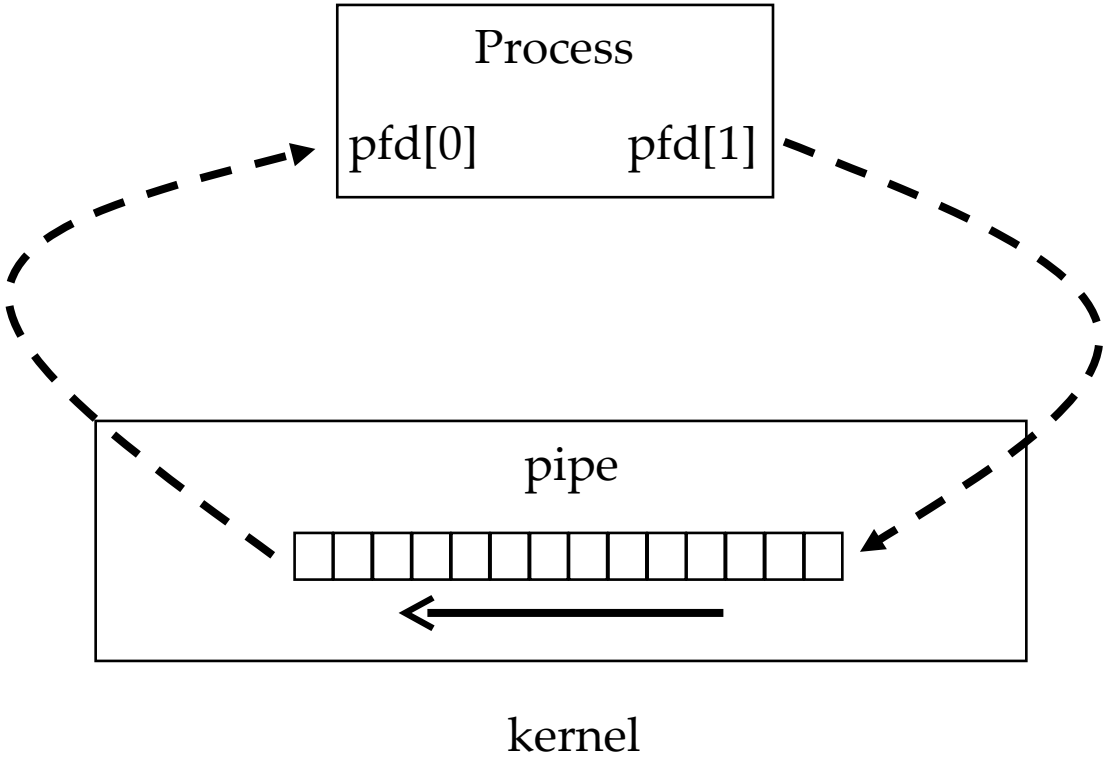
# pipe

**#include <unistd.h>**

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

- Crea un canale di comunicazione rappresentato da 2 file descriptors
  - **pfd[0]** viene aperto in lettura
  - **pfd[1]** viene aperto in scrittura
- Gli *fd* non sono connessi a nessun file reale, ma a un buffer nel kernel

# Schema di funzionamento



# Dimensione e limite

- $ ulimit -a | grep pipe
  - ✓ pipe size (512 bytes, -p) 8 ->512 x 8 = 4096 byte
- Definito in
  - ✓ /usr/src/linux/include/linux/limits.h

# write su pipe

**int write (pfd[1], void *buf, int nbytes)**

- I dati sono scritti in ordine di arrivo
- Quando la pipe è piena write si blocca finché read libera abbastanza spazio
- L'intero ritornato è il numero di byte effettivamente scritti sul file (-1 in caso di errore)

# read su pipe

```
int read (pfd[0], void *buf, int nbytes)
```

- Se ci sono dati, vengono letti nell'ordine in cui sono stati scritti e una volta letti vengono cancellati dalla pipe (non possono essere riletti)
- Quando finiscono i dati (o si raggiunge **`nbytes`**) la read ritorna il numero di byte letti;
- Se la pipe è vuota la lettura si blocca finché arriva qualcosa da leggere.
- Se la pipe è chiusa in scrittura, **`read`** ritorna 0 (EOF)
- L'intero ritornato è il numero di byte effettivamente letti dal file (0 se EOF, -1 in caso di errore)

# close su pipe

```
close(pfd[0]);  close(pfd[1]);
```

- Chiudendo il file descriptor si provoca un EOF
- Se si prova a scrivere su una pipe chiusa in lettura, **write** ritorna -1 per segnalare un errore

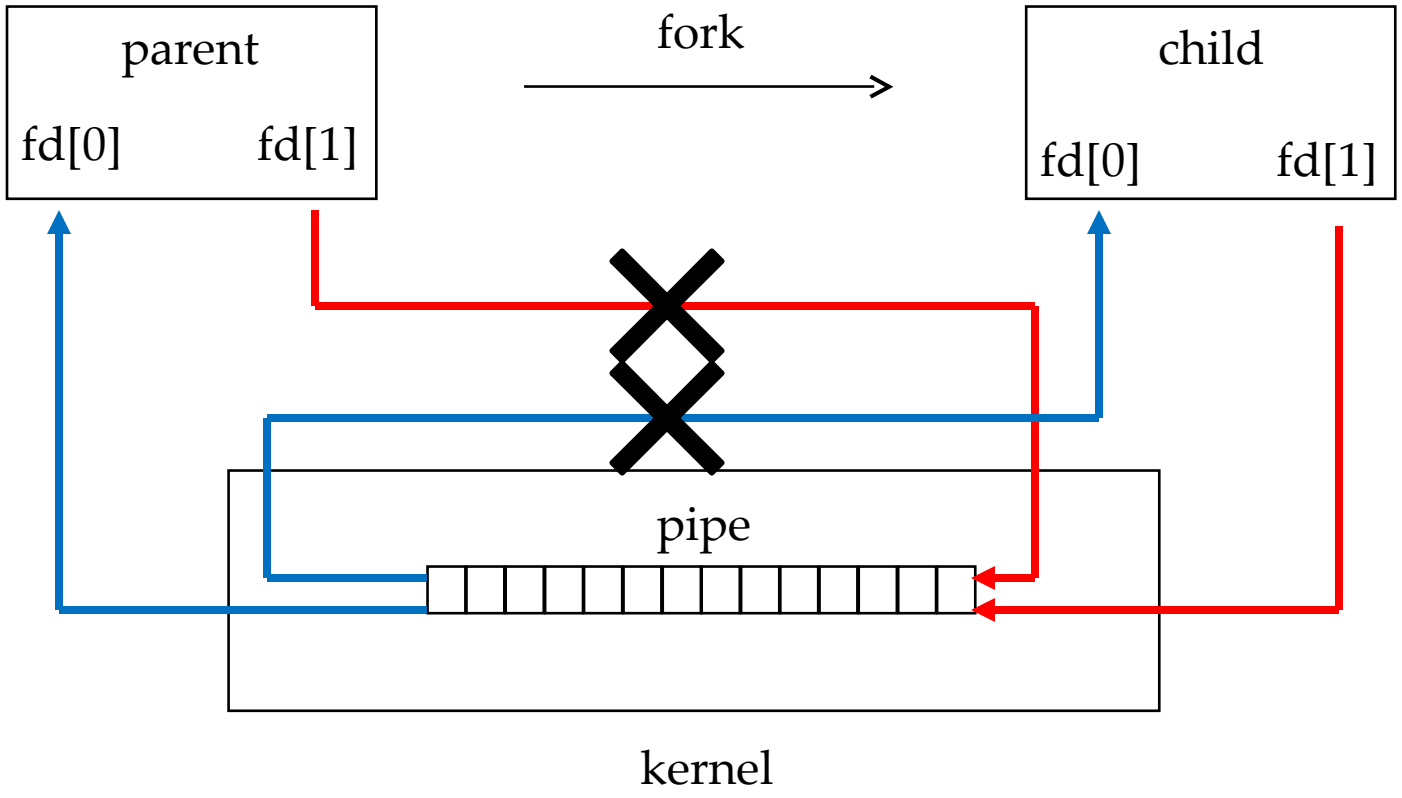
(Esempio "01-pipe.c")

# Comunicare con le pipe

- Forma più vecchia di *Inter Process Communication* in UNIX
- Sono half-duplex: i dati possono scorrere solo in una direzione
- Possono essere usate **SOLO** fra processi che hanno un antenato comune
- Normalmente una pipe è creata dal processo padre il quale poi chiama la **fork**
- La pipe viene dunque *copiata* dal padre al figlio
- In questo modo se uno dei due processi scrive su un capo della pipe, l'altro può leggere

(Esempio "02-pipe-fork.c")

# Comunicare con le pipe



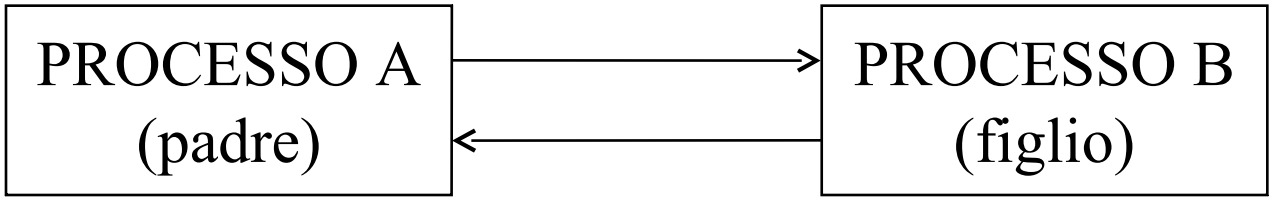
# pipe - esempio

Voglio scrivere un programma che si comporti come la riga di comando **`who|wc`** (che permette di contare gli utenti collegati)

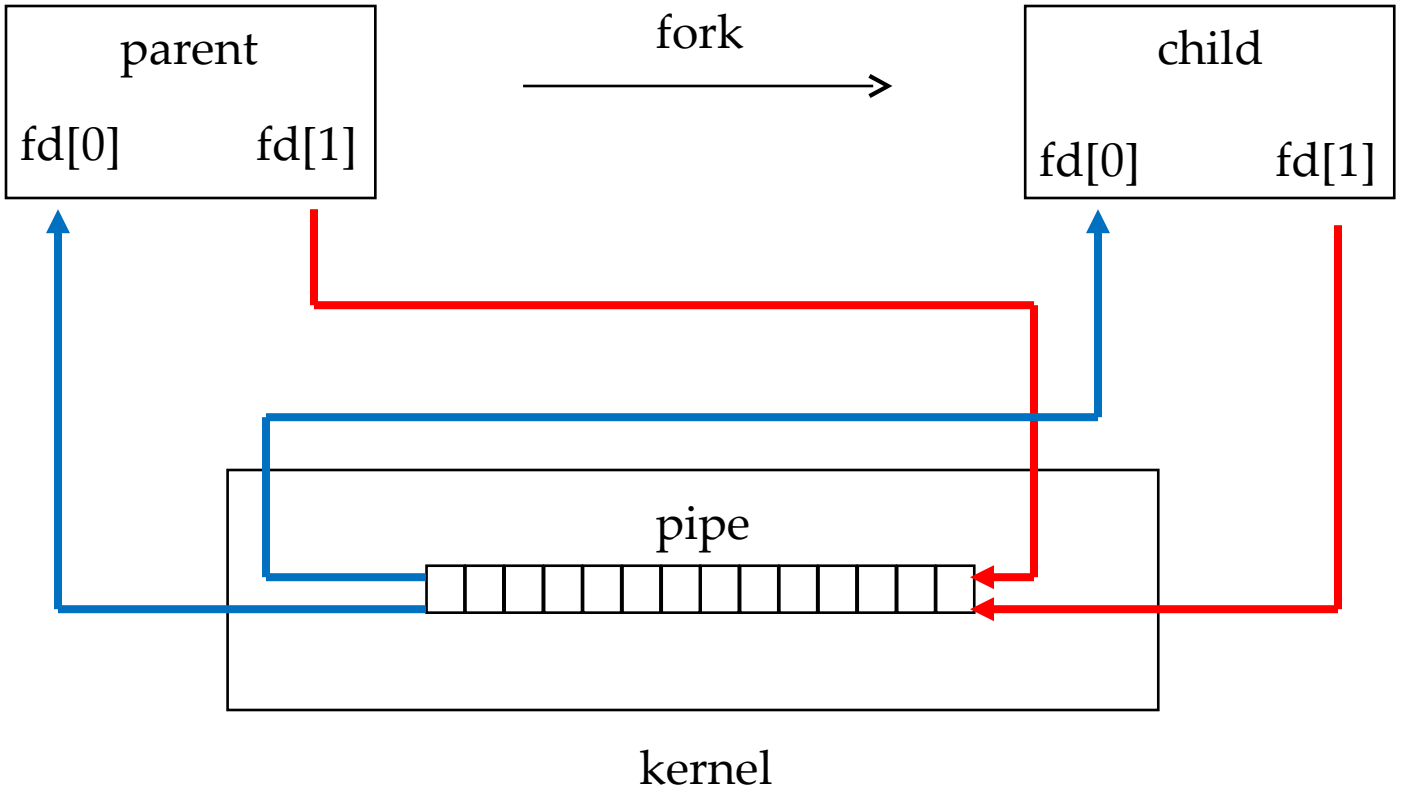
- Creare una pipe
- Creare un processo figlio
- Collegare lo *stdout* del figlio al *write* della pipe
- Collegare lo *stdin* del padre al *read* della pipe
- Il figlio esegue **`who`**
- Il padre esegue **`wc`**

(Esempio "03-whowc.c")

# Comunicazione full-duplex tra processi padre-figlio



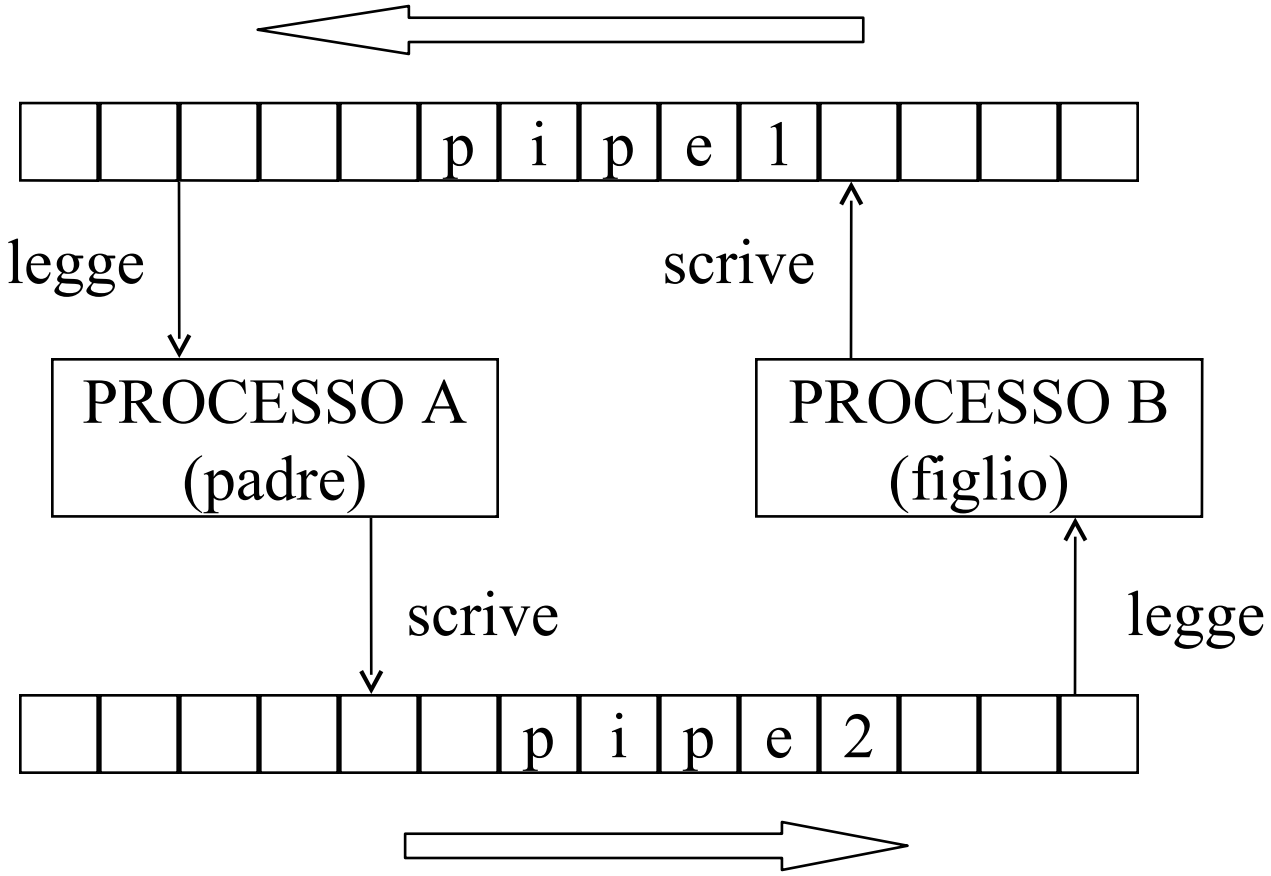
# La soluzione sbagliata



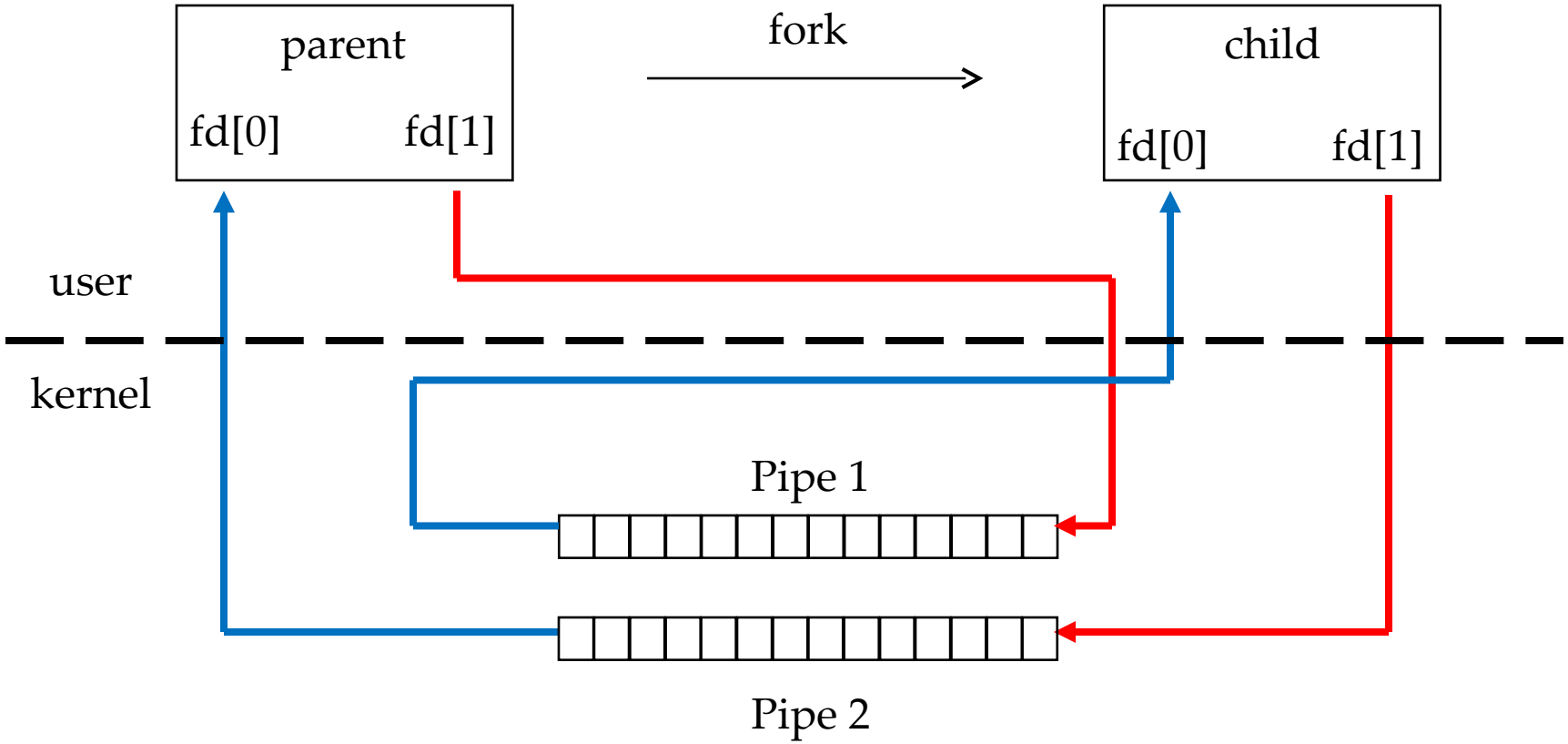
# Cosa c’è che non va?

- I byte scritti dal padre non sono distinguibili da quelli scritti dal figlio
- In queste condizioni il padre legge tutto quello che trova, compreso quanto ha scritto lui
- Si può creare una situazione di stallo

# La soluzione corretta



# Implementazione



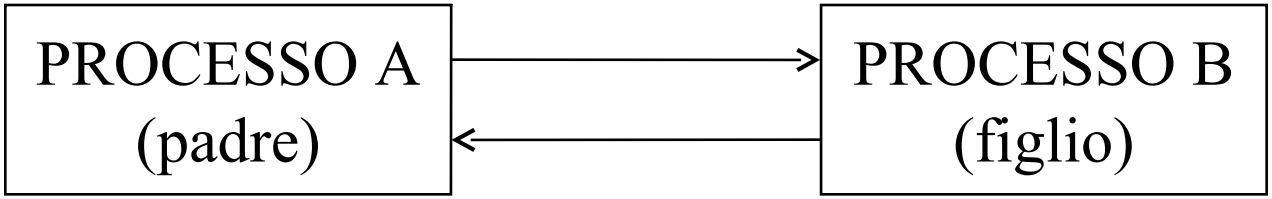
# Esercizio (20 min.)

Scrivere un programma che permette una comunicazione full-duplex tra processo padre e processo figlio

(Esempio “04-pipe-full-duplex.c”)

# Un problema delle pipe

- Possono comunicare attraverso le pipe soltanto processi con un antenato comune che deve essersi preoccupato di aprirle.




- In alcuni casi questo è limitativo. Occorre utilizzare uno strumento diverso: le **FIFO**

# Esempio di applicazione

- minishell in grado di interpretare comandi con parametri dando significato corretto ai caratteri speciali:
  - & background
  - >, < ridirezione
  - | pipe [singole/multiple]

# Bibliografia


- *GaPiL. Guida alla Programmazione in Linux*.. Simone Piccardi
- *Unix. Network Programming. Interprocess Communications*. Richard Stevens
- Operating System: Design and Implementation 3/e by Andrew Tanenbaum and Albert S. Woodhull
- *Modern Operating Systems*. Andrew. S. Tanenbaum, Prentice-Hall 1992, Edizione italiana *Moderni sistemi operativi*, Jackson Libri.

# La comunicazione fra processi: PIPE e FIFO

***dr. Andrea E. Naimoli***

Università degli Studi di Trento

Dipartimento di Ingegneria e Scienza dell'Informazione

via Sommarive 14

I - 38050 Trento - Povo, Italy

# FIFO

- FIFO = First In, First Out (come una coda)
- Sono dette anche named pipes
- Le FIFO non sono strutture all'interno del kernel
- Si comportano per certi aspetti come i file, per altri come le pipe

# FIFO - caratteristiche

- Come i file:
  - ✓ Hanno un nome
  - ✓ Tutti i processi con i permessi adeguati possono accedervi (non soltanto i discendenti del processo che le ha aperte)
- Come le pipe:
  - ✓ I primi dati scritti sono i primi ad essere letti
  - ✓ Non si può usare lseek
  - ✓ I dati possono essere letti una sola volta

# Utilizzo

- Si devono innanzitutto creare, poi vanno aperte in lettura o in scrittura.

- Quando una FIFO viene aperta in scrittura, il programma si blocca fino a quando qualcuno non la apre in lettura (e viceversa)
  - *Blocking I/O: questo permette il loro uso per una sincronizzazione tra diversi processi*.

- Le FIFO come tubi
  - *FIFO = un tubo in cui l'acqua che scorre è rappresentata dai dati.*
  - *Aprire la FIFO è come aprire un rubinetto: non ha senso immettere dati (acqua) finché entrambe le estremità sono state collegate.*

# Una nuova system call

int mkfifo(char *pathname, mode_t mode);

- pathname è il nome della nuova FIFO
- mode sono i permessi della FIFO

Esempio:

```
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
…
mkfifo("myFIFO",FILE_MODE)
```

# Come usare una FIFO

- Una volta creata, la FIFO va aperta in lettura ed in scrittura
  - ✓ N.B. **`mkfifo`** si limita a creare la FIFO, ma il processo non può ancora accedervi.

- Si può usare la system call **`open`**, oppure le funzioni della libreria **`<stdio.h>`**

```
lettura   = open("myFIFO",O_RDONLY | O_NONBLOCK);
scrittura = open("myFIFO",O_WRONLY | O_NONBLOCK);
```

solo se non si vuole bloccare fino all'apertura dell'altro capo della FIFO

# Come cancellare una FIFO

- Nota Bene
  - una pipe viene cancellata dal sistema (kernel) una volta che vengono chiusi tutti i suoi canali di I/O oppure con la chiamata **exit()**
  - una FIFO rimane nel file-system anche dopo una **exit()**
  - per cancellarla occorre esplicitare da programma la sua cancellazione con la system call:

    **unlink ("myFIFO");**

# read FIFO

- Utilizzo analogo alle pipe:
  - ✓**int read (fifo_fd, char *buf, int nbytes)**
  - ✓Se ci sono dati, vengono letti nell'ordine in cui sono stati scritti e una volta letti vengono cancellati dalla FIFO (non possono essere riletti); quando finiscono i dati (o si raggiunge **nbytes**) la **read** ritorna il numero di byte letti
  - ✓Se si prova a leggere da una FIFO vuota, il programma si blocca fino a quando qualcuno ci scrive qualcosa

# write FIFO

```
int write (fifo_fd, char *buf, int nbytes)
```

- Se si prova a scrivere su una FIFO piena, il programma si blocca fino a quando qualche altro processo non libera del posto.
- La **write** è un'operazione *atomica*
- Se si deve scrivere un numero di byte che non supera la dimensione della FIFO, questi vengono scritti tutti in una volta (Non si hanno quindi processi diversi che mescolano parte dei loro dati nella FIFO).

# Esercizio (30 min.)

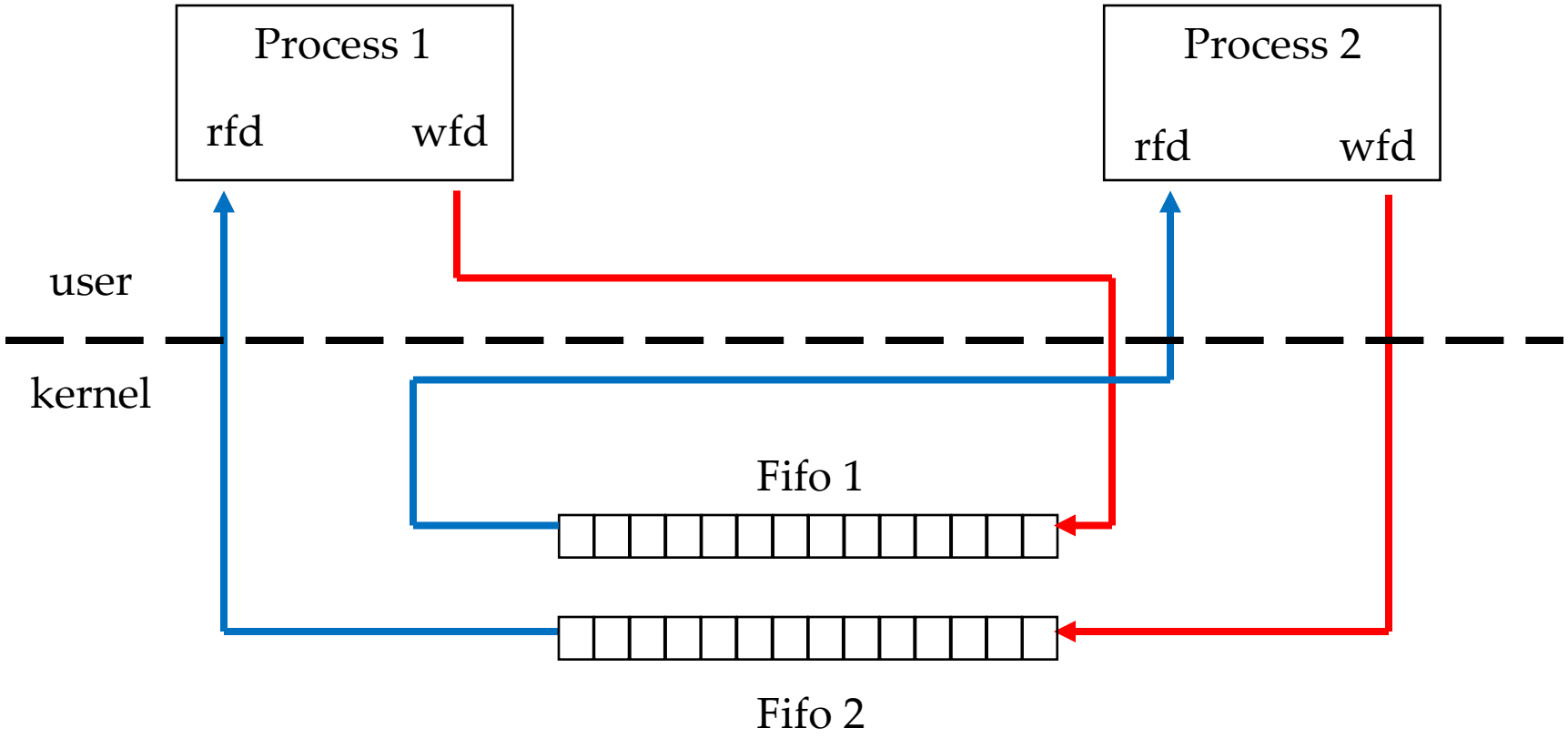
- Scrivere un programma che crea la FIFO con nome "**myFIFO**"
- Da questa FIFO il figlio legge mentre il padre scrive un semplice messaggio

(Esempio "05-fifo.c")

# Comunicazione nei 2 sensi (full-duplex)

Poiché le FIFO, una volta aperte, si comportano come le pipe, per realizzare una comunicazione nei 2 sensi tra 2 processi sono necessarie 2 FIFO

# Implementazione



# FIFO come pipe

**Vantaggi**

- Permettono la comunicazione tra processi che non siano legati da vincoli di parentela

**Svantaggi**

- Servono 5 system call diverse per stabilire il canale di comunicazione
  - Create, write/read per PIPE
  - Create, open, write/read, close, unlink per FIFO
- Posso avere dei problemi coi nomi dei file (ad es. collisioni)