

Quinta Esercitazione

Comunicazione tra
processi Unix: pipe

System Call relative alle pipe

pipe	<ul style="list-style-type: none">• int pipe (int fd[]) crea una pipe e assegna i 2 file descriptor relativi agli estremi di lettura/scrrittura ai primi due elementi dell'array fd.• Restituisce 0 in caso di creazione con successo, -1 in caso di errore
dup	<ul style="list-style-type: none">• fd1=dup (fd) crea una copia del descrittore fd.• La copia viene messa nella prima posizione libera della tabella dei file aperti.• Assegna a fd1 la nuova copia del descrittore, -1 in caso di errore

Primitive di comunicazione

read	<ul style="list-style-type: none">Stessa system call usata per leggere file regolari, ma può essere bloccante: Se la pipe è vuota il processo chiamante si blocca fin quando non ci sono dati disponibili.
write	<ul style="list-style-type: none">Stessa system call usata per scrivere su file regolari, ma può essere bloccante: Se la pipe è piena il processo chiamante si blocca fin quando non c'è spazio per scrivere.
close	<ul style="list-style-type: none">Stessa system call usata per chiudere file descriptor di file regolariNel caso di pipe, usata da un processo per chiudere l'estremità della pipe che non gli interessa

Esempio – comunicazione tra processi mediante pipe

Realizzare un programma C che, utilizzando le *system call* di UNIX, preveda un'interfaccia del tipo:

esame F N C

- **F** rappresenta il nome assoluto di un file
- **N** rappresenta un intero
- **C** rappresenta un carattere.

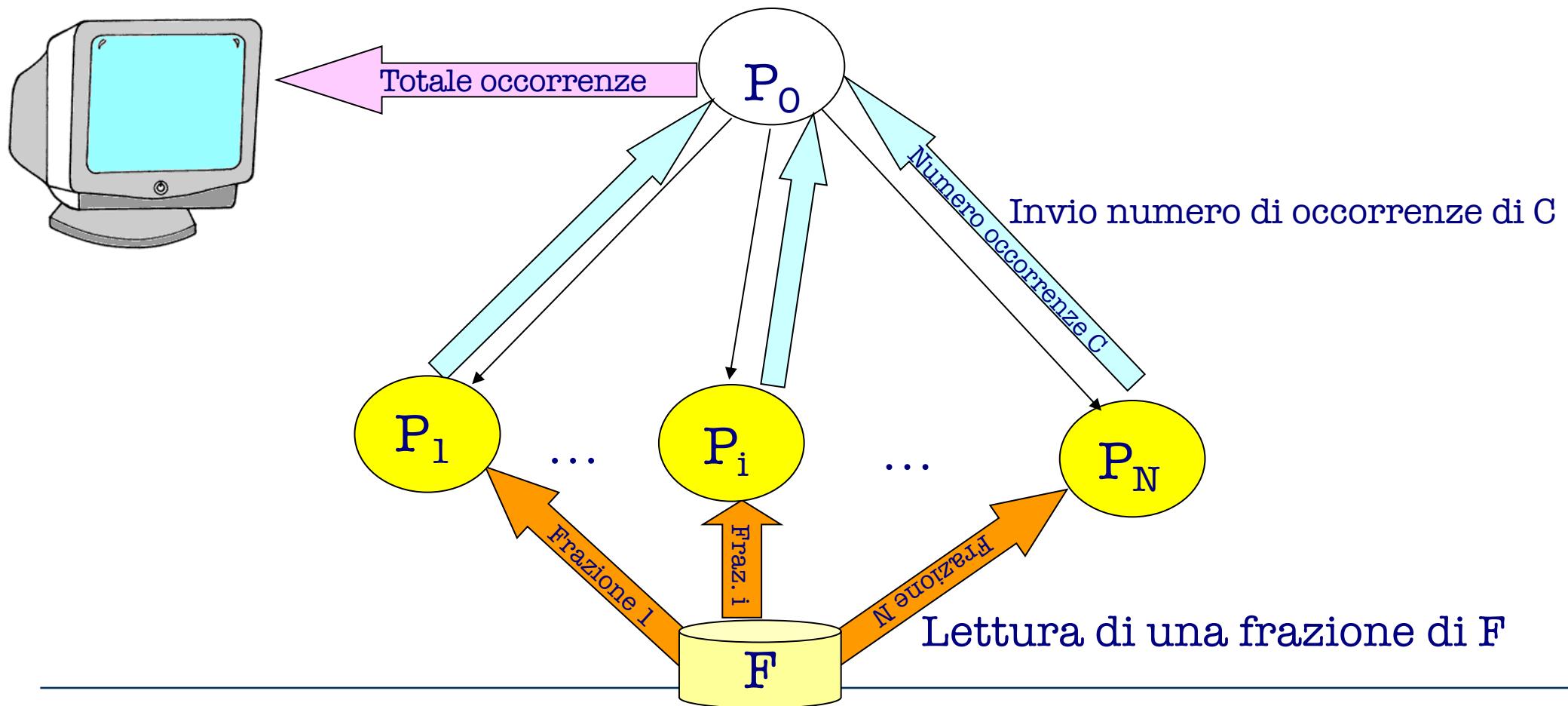
Il processo iniziale P_0 deve creare un numero **N** di processi figli ($P_1, P_2, \dots P_N$).

Ogni figlio P_i ($i=1, \dots N$) deve leggere una parte diversa del file **F**: in particolare, se L è la lunghezza del file **F**, il figlio dovrà leggere una frazione di L/N caratteri dal file **F**, secondo il seguente criterio:

- P_1 leggerà la prima frazione;
- P_2 leggerà la seconda frazione;
- ...
- P_N leggerà l'ultima frazione **N**;

Ogni processo P_i leggerà quindi una frazione di F allo scopo di calcolare il numero delle occorrenze del carattere C nella parte di file esaminata; al termine della scansione, P_i comunicherà al padre il numero delle occorrenze di C incontrate nella frazione di file assegnatagli.

Il padre P_0 , una volta ottenuti i risultati da tutti i figli, **stamperà il numero totale di occorrenze** di C nel file F e terminerà.



Impostazione

- Accesso parallelo di processi a parti diverse dello stesso file: aperture separate del file da parte dei figli per evitare la condivisione dell'I/O pointer.
- Comunicazione dei figli con il padre: uso di **una sola pipe**.
- Calcolo della dimensione di un file e posizionamento dell'I/O pointer in posizione arbitraria: lseek()

Soluzione dell'esercizio

```
#include <fcntl.h>
...
#define Nmax 50
#define Mdim 8

int fd, pipefd[2], L;
char c, msg[Mdim];
void figlio(int ind, char c, int fra, char filein[]);

main(int argc, char **argv)
{   int pid[Nmax], N, stato, i, tot, K, p, fraz;
    if (argc!=4)
    {   printf("sintassi!\n");
        exit(-1);
    }
```

```
N=atoi(argv[2]);
c=argv[3][0];
if ((fd=open(argv[1],O_RDONLY))<0)
{
    perror("open padre");
    exit(-2);
}
L=lseek(fd, 0, 2); /* calcolo dim. L*/
fraz = (L%N)==0 ? L/N : L/N+1;
close(fd);
if (pipe(pipefd)<0) exit(-3); /* apertura pipe */
/* creazione figli */
for(i=0;i<N;i++)
{
    if ((pid[i]=fork())<0)
    {
        perror("fork error");
        exit(-3);
    }
    else if (pid[i]==0) figlio(i, c, fraz, argv[1]);
```

```
/* padre */
close(pipefd[1]);
for(tot=0,i=0; i<N; i++)
{    read(pipefd[0],&K, sizeof(int)); //lettura binaria
tot+=K;
}
close(pipefd[0]);
printf("\n valore ottenuto: %d\n", tot);
for(i=0; i<N; i++)
{    p=wait(&stato);
    printf("terminato%d con stato %d\n", p, stato>>8);
}
exit(0); /* fine padre */
} /* fine main */
```

```
void figlio(int ind, char c, int fra, char filein[])
{ int i, j, cont=0; char car;
close(pipefd[0]);
fd=open(filein, O_RDONLY); /*I/O pointer privato */
lseek(fd, ind*fra, 0); /*posiz. inizio frazione */
for(j=0; j<fra; j++)
{   i=read(fd, &car, 1);
    if (i==0) break;
    if (car==c)
        cont++;
}
write(pipefd[1],&cont, sizeof(int)); //scrritt. binaria
close(pipefd[1]);
close(fd);
exit(0);
}
```

Esercizio 1 (1/2)

Si realizzi un programma di sistema in C che preveda la seguente interfaccia:

./inverti_classifica class

dove **class** è il pathname di un file di testo esistente, che contiene la classifica finale di una gara di mezzofondo di atletica.

Si assuma che class abbia il seguente formato: ogni riga del file rappresenta il risultato ottenuto da un partecipante alla gara e contiene 2 informazioni: il codice identificativo del partecipante e il tempo ottenuto.

In particolare, tali informazioni sono memorizzate in un formato prefissato, nel quale ogni riga è composta da 9 caratteri ed ha la seguente struttura

<cod-mm:ss>

ovvero: 3 caratteri per il codice, ' - ', 2 caratteri per i minuti, ':', 2 caratteri per i secondi'

Il file è ordinato in **ordine crescente di tempo** (cioè la prima riga contiene i dati del vincitore e l'ultima riga i dati relativi al partecipante arrivato ultimo).

Esercizio 1 (2/2)

Il programma deve stampare sullo standard output la classifica ordinata in senso inverso (dall'ultima riga alla prima), per comunicare i risultati dall'ultimo al primo classificato.

A tal fine il processo PO crea un figlio P1.

Il figlio P1 legge le righe contenute in **class** in senso inverso (cioè dall'ultima alla prima) e invia ogni riga letta al padre PO. Una volta conclusa questa operazione, P1 termina.

Il padre PO riceve ogni riga letta dal figlio P1 e la stampa sullo standard output.

Una volta terminata la comunicazione , il padre attende la terminazione del figlio e successivamente termina.

Note

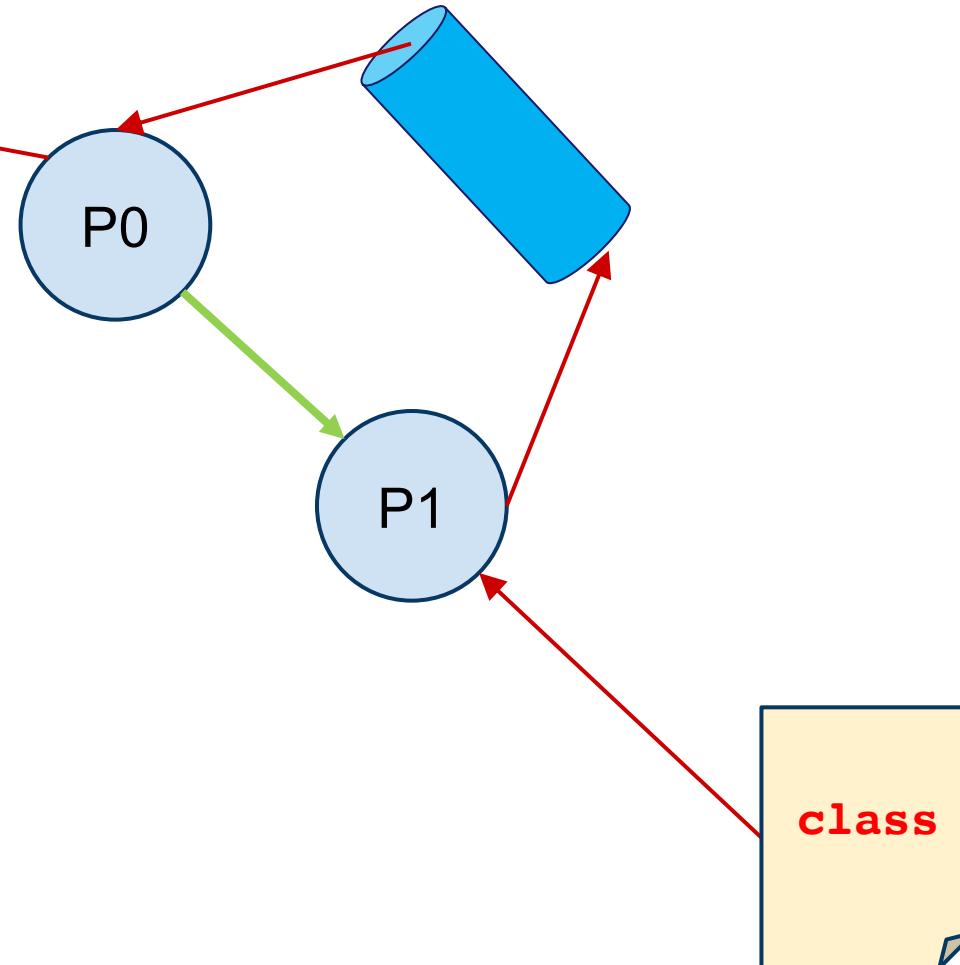
- Vedi es. 4.1
- Come realizzare la **comunicazione** tra P1 e PO?
-> creazione di una **pipe**



Modello di soluzione

stdout

Linee del file class in
ordine inverso



Esercizio 2 - Redirezione su pipe

Si realizzi un programma C che, utilizzando le system call di Unix, preveda la sintassi di invocazione:

./cerca file1 file2 ..fileN parola

dove:

- **file1, file2 .. fileN**: sono pathname di file di testo esistenti nel file system.
- **parola**: una stringa di caratteri alfabetici.

Esercizio 2 (2/3)

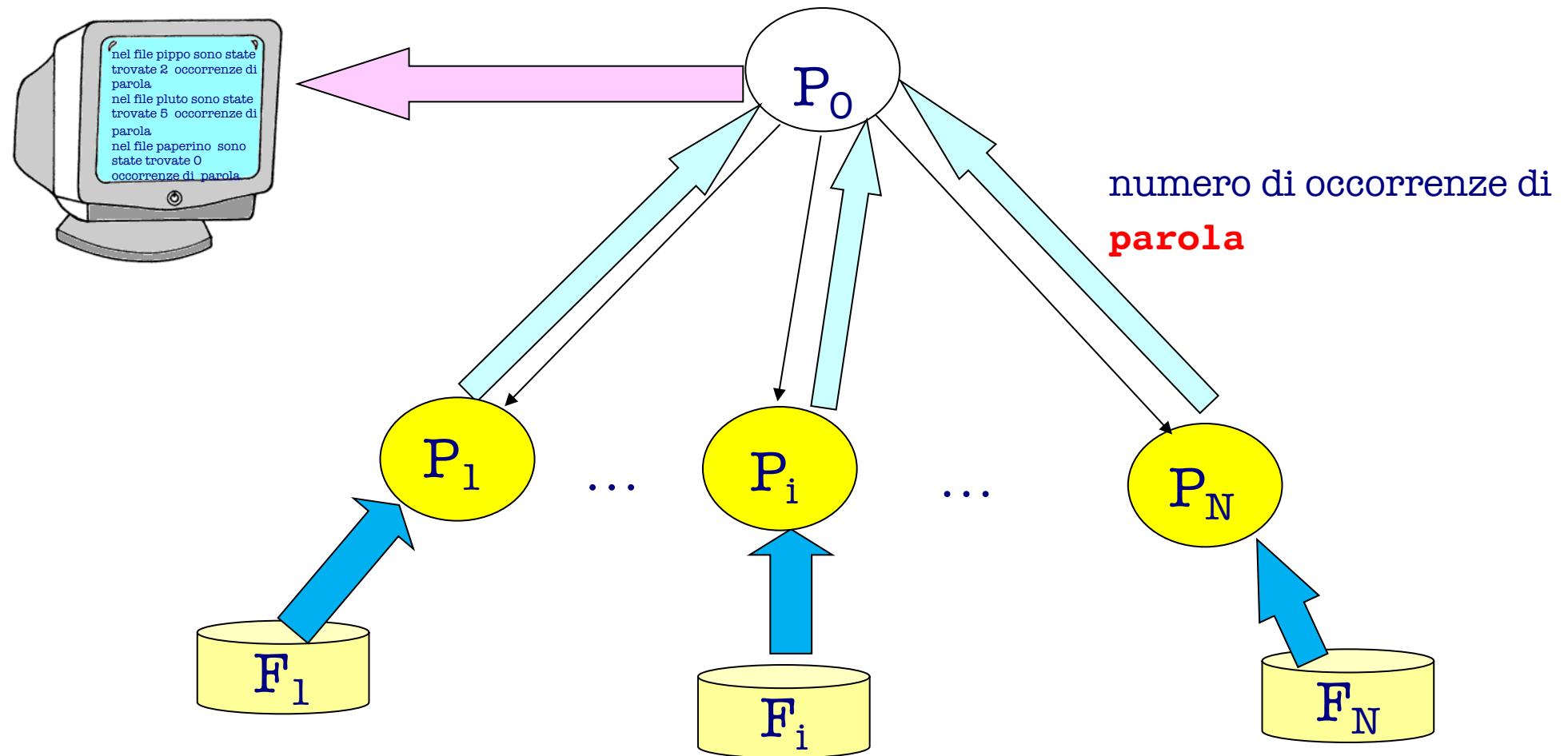
Il processo padre **PO** genera N figli (tanti quanti sono i filename dati come argomenti) P1, P2, .. PN:

Ogni figlio P_i , **tramite il comando grep**, deve **contare** le righe di $file_i$ nelle quali la stringa **parola** compare almeno una volta, **e comunicare a PO il valore risultante da tale conteggio**.

PO deve:

- Acquisire il valore K_i calcolato da ciascun figlio P_i , e per ognuno stampare sullo standard output la stringa: «nel file < F_i > sono state trovate $<K_i>$ occorrenze di <parola>».
- Successivamente PO attenderà la terminazione di ogni figlio e terminerà.

Esercizio 2 - Modello di soluzione



Esercizio 2 - Suggerimenti

Ogni figlio Pi deve contare tramite l'esecuzione del comando **grep**.

```
daniela$ man grep  
daniela$ grep -c C file2
```

Attenzione:

L'output di **grep -c C** è una **stringa** che rappresenta il numero di occorrenze di **C**. **Non è di tipo intero!**

Comunicazione figli->padre: quante pipe?

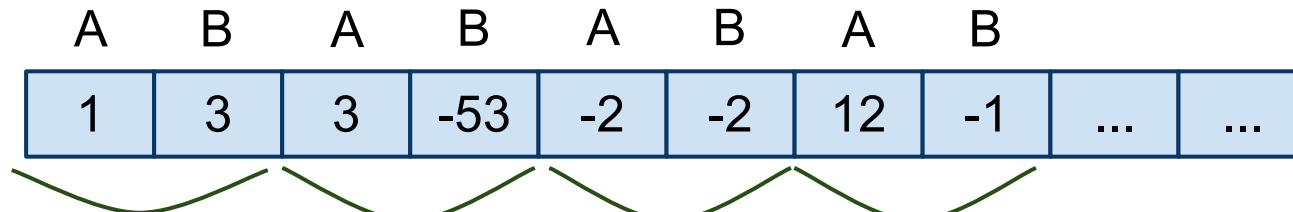
(**NB:** il padre deve poter distinguere il mittente di ogni messaggio ricevuto)

Esercizio 3 - IPC e sincronizzazione tramite pipe

- Si realizzi un programma C che, usando le opportune system call unix, realizzi la seguente interfaccia:

./correggi file_in file_out

- **file_out**: nome di un file non esistente nel filesystem
- **file_in**: nome di un file **binario** esistente contenente **N copie di numeri interi**, con N non noto a priori.



- NB: file binario ≠ file di testo

Esercizio 3 - Traccia (2/3)

Il programma deve realizzare il seguente comportamento:

- Il processo padre (**P0**) deve generare due figli **P1** e **P2**.
- Il processo **P2** deve accedere a **file_in** per
 - **Leggere** i due interi (A,B) di ogni coppia in **file_in** e **comunicare a P1 il valore del maggiore tra i due**;
 - Terminata l'elaborazione dell'ultima coppia, **P2 deve notificare a P1** il termine della sua elaborazione.

Esercizio 3 - Traccia (3/3)

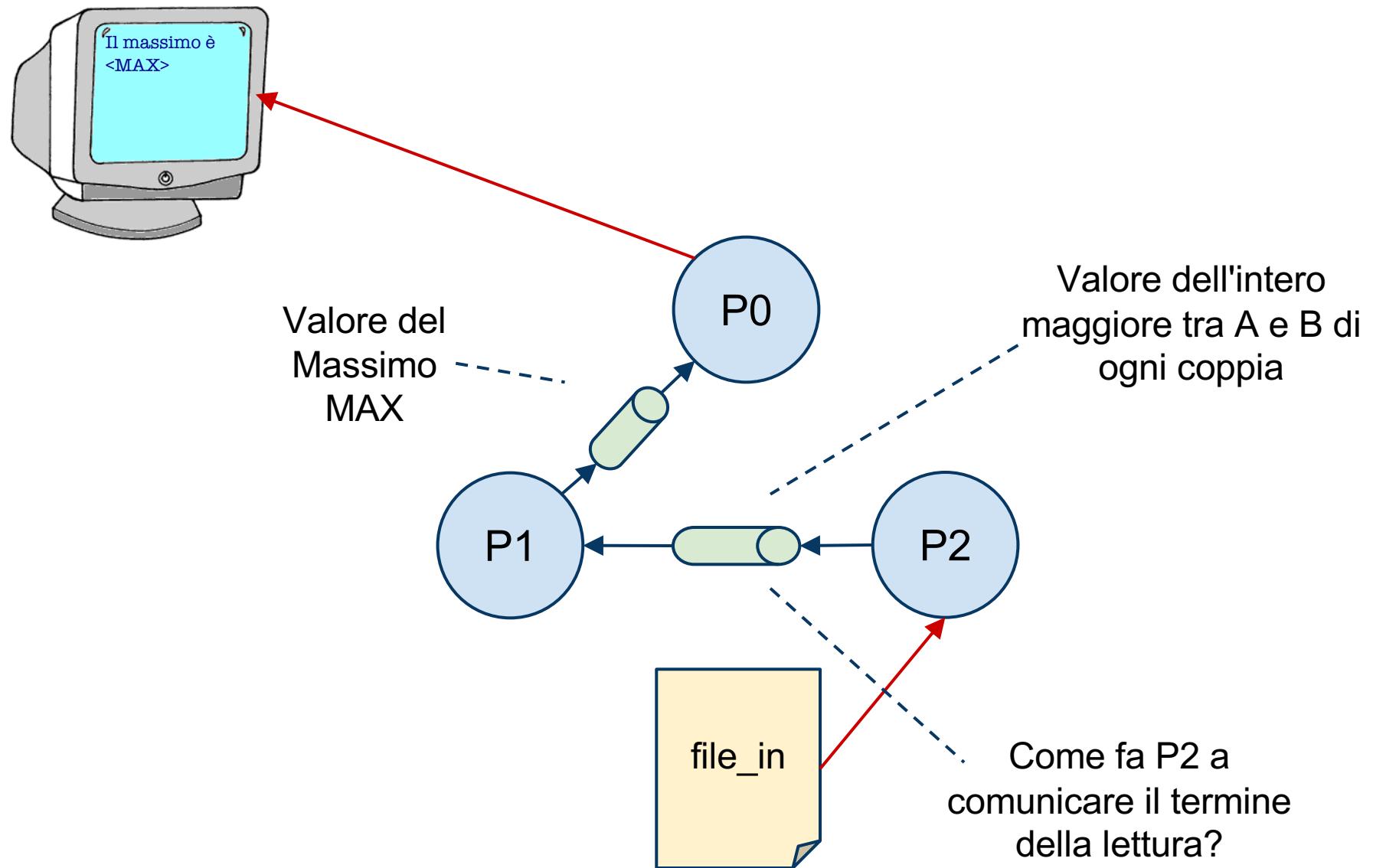
Il processo **P1** deve calcolare il massimo MAX tra tutti i valori ricevuti da P2 e, al termine della comunicazione, dovrà **comunicare a P0** il valore di **MAX**.

Il processo **P0** dovrà stampare sullo standard output il valore ricevuto da P1, attendere la terminazione dei figli, e terminare.

Esercizio 3 - Note alla soluzione

- Vedi esercizio 4.3
 - Uso di **pipe** VS uso di **segnali** per sincronizzare processi
 - **Quante pipe** occorre creare?
 - Comunicazione tra P2 e P1
 - Comunicazione tra P1 e P0
 - **Chi le deve creare?**
 - Dipenderà da chi le deve usare...
 - La pipe tra P0 e P1 può esser creata da P1?!
 - E quella tra P1 e P2?
-

Esercizio 3 - Modello di soluzione



Esercizio 3 - Note alla soluzione

- **Chiudere subito** gli estremi che non mi interessano delle pipe!
- Come fa **P2** a comunicare a **P1** il termine della sua elaborazione?
 - Un **segnale di fine** ? O è possibile farlo anche con altri strumenti?
 - **Chiusura lato scrittura pipe?**

Pipe - Riflessioni post-esercizio

Le **pipe** sono uno strumento di **comunicazione** tra processi

- Consentono a processi in gerarchia di scambiarsi dati

Alcune **differenze** rispetto a read-write su file:

- **read e write bloccanti** (se la pipe è rispet. vuota o piena)
- Una **read** ritorna zero se e solo se **tutti** i fd relativi al lato di scrittura sono chiusi

→ Perché è importante NON LASCIARE APERTE ESTREMITÀ INUTILIZZATE DELLE PIPE?

Le pipe possono essere uno **strumento di sincronizzazione** tra processi. **Quando conviene usare pipe e quando segnali?**

- Se devo comunicare dei dati tra processi, sono più comode le pipe,
- ma se un processo deve **fare delle operazioni intanto** che aspetta di ricevere qualcosa da un altro, **devo ricorrere ai segnali!**