

Esercitazione 4

Gestione dei file in Unix



Primitive fondamentali (1/2)

open	<ul style="list-style-type: none">• Apre il file specificato e restituisce il suo file descriptor (fd)• Crea una nuova entry nella tabella dei file aperti di sistema (nuovo I/O pointer)• fd è l'indice dell'elemento che rappresenta il file aperto nella tabella dei file aperti del processo (contenuta nella user structure del processo)• possibili diversi flag di apertura, combinabili con OR bit a bit (operatore)
close	<ul style="list-style-type: none">• Chiude il file aperto• Libera il file descriptor nella tabella dei file aperti del processo• Eventualmente elimina elementi dalle tabelle di sistema

Primitive fondamentali (2/2)

read	<ul style="list-style-type: none">• read(fd, buff, n) legge al più n bytes a partire dalla posizione dell'I/O pointer e li memorizza in buff• Restituisce il numero di byte effettivamente letti<ul style="list-style-type: none">0 per end-of-file-1 in caso di errore
write	<ul style="list-style-type: none">• write(fd, buff, n) scrive al più n bytes dal buffer buff nel file a partire dalla posizione dell'I/O pointer• Restituisce il numero di byte effettivamente scritti o -1 in caso di errore
lseek	<ul style="list-style-type: none">• lseek(fd, offset, origine) sposta l'I/O pointer di offset posizioni rispetto all'origine. Possibili valori per origine:<ul style="list-style-type: none">0 per inizio del file (SEEK_SET)1 per posizione corrente (SEEK_CUR)2 per fine del file (SEEK_END)

Esempio

Primi passi con operazioni di I/O
e sincronizzazione tra processi

Esempio - Traccia (1/2)

Si realizzi un programma C che usi le opportune System Call Unix e realizzi la seguente interfaccia

```
./conta_caratteri c1 c2 N file_in file_out
```

- **c1** e **c2** sono caratteri ASCII
- **N** è un numero intero
- **file_in**: path di un file di testo esistente nel filesystem, composto di righe di lunghezza non nota a priori
- **file_out**: path di un file di testo **non esistente** nel filesystem

Esempio - Traccia (2/2)

Il programma deve realizzare il seguente comportamento:

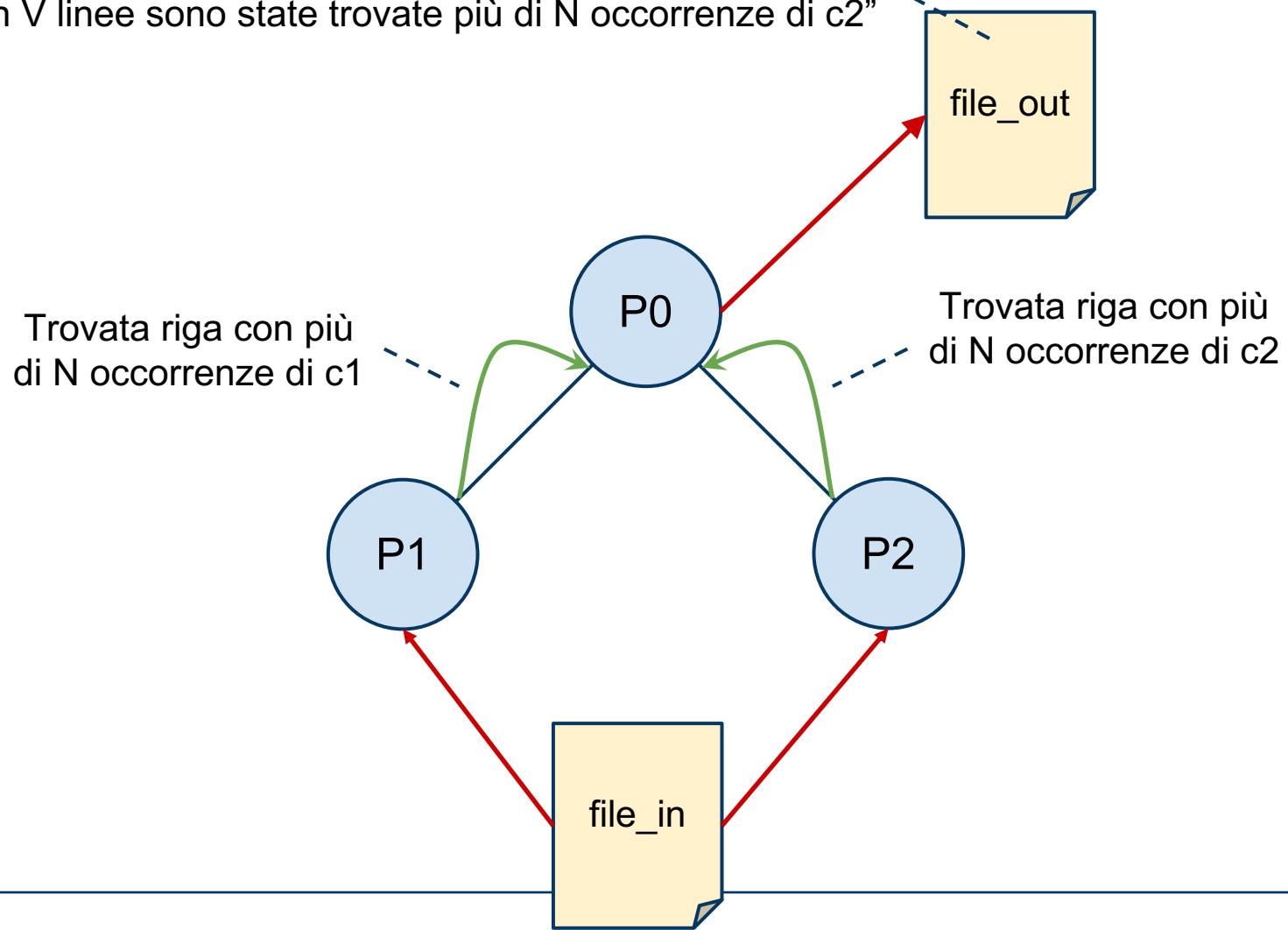
Il processo padre P0 genera **P1** e **P2**, **ognuno dei quali** deve:

- Leggere **file_in** e contare, riga per riga, il numero di occorrenze rispettivamente del carattere **c1** (P1) e **c2** (P2)
 - **Al termine della lettura di ogni riga**, avvertire P0 se il numero di occorrenze trovate del carattere di competenza è maggiore di **N**
- **P0** deve:
- Tenere traccia, **separatamente per ciascun figlio**, del numero di righe con più di **N** occorrenze dei caratteri cercati
 - **Una volta terminate tutte le letture dei figli**, scrivere su **file_out** tale informazione

Esempio - Schema

“In L linee sono state trovate più di N occorrenze di c1”

“In V linee sono state trovate più di N occorrenze di c2”



Esempio - Problematiche

Gestione dei file:

- Gestione della lettura / scrittura
- P1 e P2 usano lo stesso file: **I/O pointer condiviso o separato?**

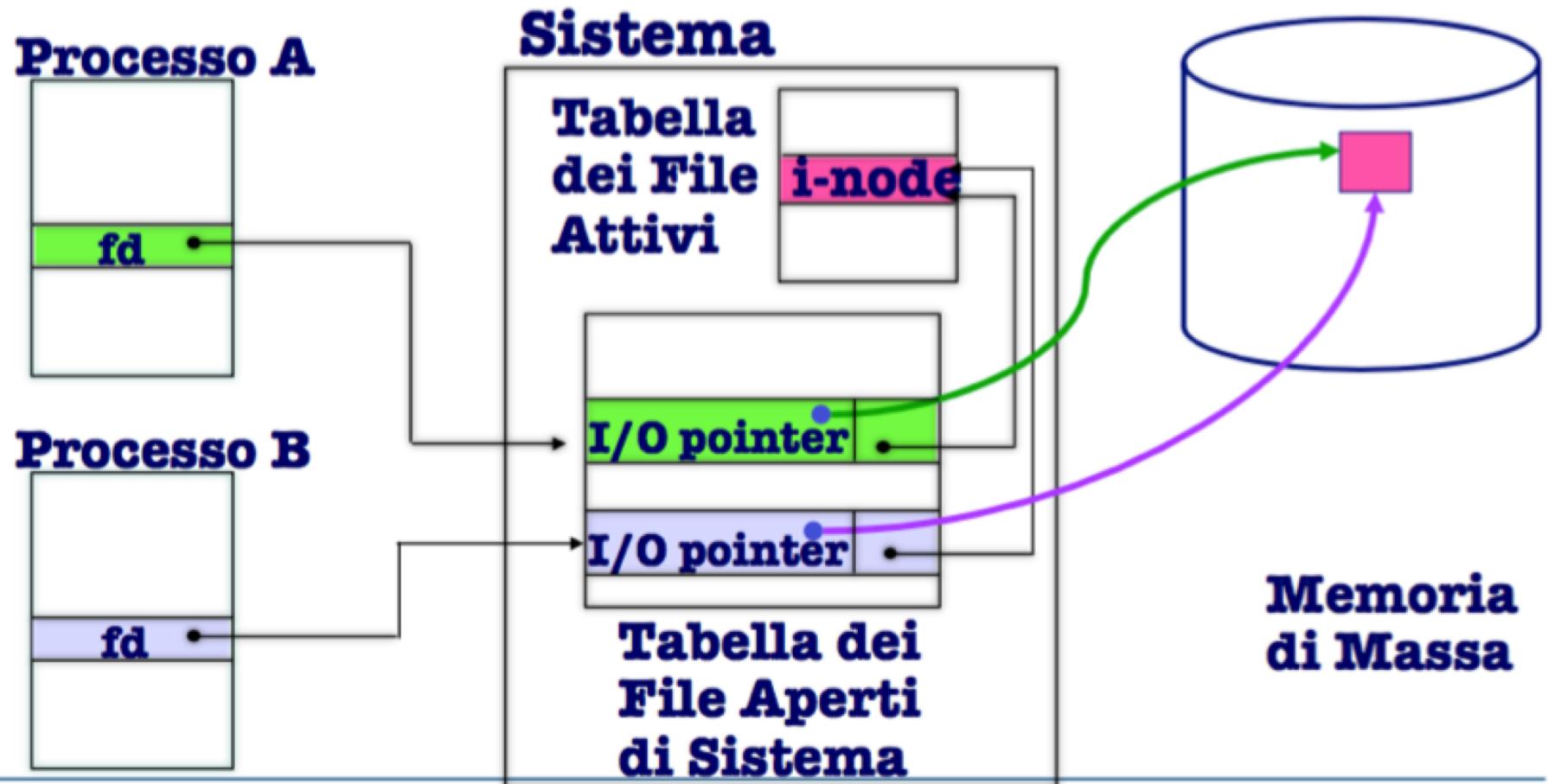
Realizzazione “comunicazione” figli-padre:

- Invio di segnali al processo PO
- È permesso l'uso della sleep per rallentare il ritmo di invio dei segnali da parte dei figli, per cercare di evitare che segnali multipli vengano "accorpati" in uno

Esempio: processi indipendenti

I processi A e B fanno ognuno la propria `open()`

=> accedono allo stesso file, ma con I/O pointer distinti

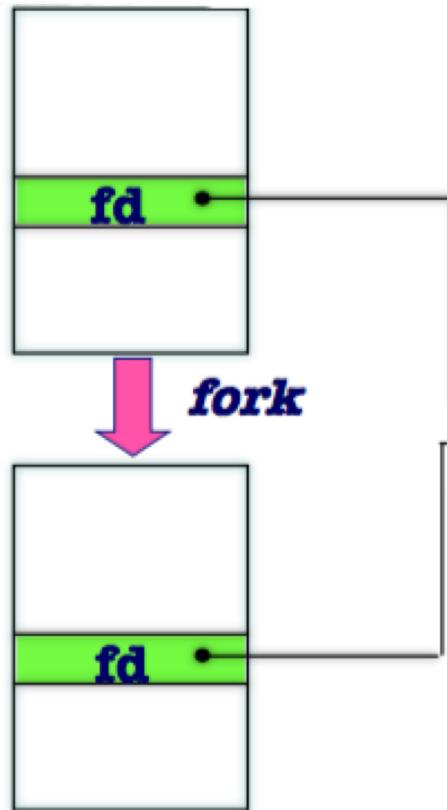


Esempio: processi in gerarchia

Il padre fa una `open()`, poi crea il figlio

=> accedono allo stesso file con I/O pointer condiviso

Processo Padre



Sistema



Processo Figlio

Esempio di soluzione (1/4)

```
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
...
int counter1, counter2; /* Variabili globali */

int main(int argc, char* argv[]) {
    int pid, N, i;
    char to_check[2]; /* per memorizzare c1 e c2 */
    char file_in[maxstr], file_out[maxstr];
    ... /* Controllo e recupero argomenti */

    /* Gestore dei segnali dai due figli */
    signal(SIGUSR1, &father_handler);
    signal(SIGUSR2, &father_handler);

    for (i=0; i<2; i++) {
        pid = fork();
        if (pid < 0) { /* Gestione errore e uscita */ }
        else if (pid == 0) { /* Figli */
            int sig_to_send;
            sig_to_send = (i == 0) ? SIGUSR1 : SIGUSR2
            figlio(file_in, to_check[i], N, sig_to_send);
            exit(EXIT_SUCCESS);
        } else { /* Codice Padre */ }
    } /* continua ... */
}
```

Esempio di soluzione (2/4)

```
/* ... Continua main */
for (i=0; i<2; i++)
    wait_child();
/* Quando i figli han finito posso stampare su file_out.
 * In questo caso non servono altri segnali */
print_output(file_out, N, to_check);
return 0;
}

void father_handler(int signo){
    /* P0 aggiorna le sue variabili globali : */
    switch(signo) {
        case SIGUSR1: /*from P1*/
            counter1++;
            break;
        case SIGUSR2: /*from P2*/
            counter2++;
            break;
        default:
            fprintf(stderr, "Segnale inaspettato\n");
            exit(EXIT_FAILURE);
    }
}
```

Esempio di soluzione (3/4)

```
void figlio(char *input, char to_check, int limit, int sig_to_send){
    int fd, counter, nread; char read_char;
    fd = open(input, O_RDONLY); /* Apre il lettura */
    counter = 0;

    while(nread = read(fd, &read_char, sizeof(char)) > 0)
    {
        if ( read_char == to_check )
            counter++;
        if ( read_char == '\n' ){ /* Linea terminata */
            if (counter > limit){
                kill(getppid(), sig_to_send);
                sleep(1);
            }
            counter = 0;
        }
    }
    close(fd);
}
```

Esempio di soluzione (3/4)

```
void figlio(char *input, char to_check, int limit, int sig_to_send){
    int fd, counter, nread; char read_char;
    fd = open(input, O_RDONLY); /* Apre il lettura */
    counter = 0;

    while(nread = read(fd, &read_char, sizeof(char)) != 0)
    {
        if ( read_char == to
            counter++;
        if ( read_char == '\n' ){ /* Linea terminata */
            if (counter > limit){
                kill(getppid(), sig_to_send);
                sleep(1);
            }
            counter = 0;
        }
    }
    close(fd);
}
```

ogni figlio fa la sua open()
=> I/O pointer condiviso o distinto?

Esempio di soluzione (4/4)

```
/* P0 scrive il risultato sul file di output: */

void print_output(char *pathname, int n, char *c) {
    /* apro il file in scrittura, se non esiste lo creo,
     * e se esiste cancello tutto il suo contenuto prima
     * di iniziare a scriverci (NB: equivalente a creat() ) */
    fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, 00640);
    if (fd < 0) { /* ...errore... */}

    sprintf(buf,"In %d linee sono state trovate piu' di %d
        occorrenze di %c\n", counter1, n, c[0]);
    bytes_to_write = strlen(buf);
    written = write(fd, buf, bytes_to_write);
    if (written < 0) { /* ... errore ... */}

    sprintf(buf,"In %d linee sono state trovate piu' di %d
        occorrenze di %c\n", counter2, n, c[1]);
    bytes_to_write = strlen(buf);
    written = write(fd, buf, bytes_to_write);
    if (written < 0) { /*... errore ...*/}

    close(fd); /* Chiusura del descrittore! */
}
```

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 assume 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 scrivere in un file di nome **SPEAKER** la classifica ordinata in senso inverso (dall'ultima riga alla prima), per consentire allo speaker della manifestazione la comunicazione dei risultati dall'ultimo al primo classificato. A tal fine il processo PO crea un figlio P1 e si mette in attesa della sua terminazione.

Il figlio P1 legge le righe contenute in **class** in senso inverso (cioè dall'ultima alla prima) e scrive ogni riga letta nel file **SPEAKER** (nel quale, pertanto, le righe saranno ordinate in ordine decrescente di tempo). Una volta conclusa questa operazione, P1 termina.

Il padre PO, una volta terminato il figlio, stampa sullo standard output il contenuto di **SPEAKER**, e successivamente termina.

Note

- Il file **class** è un file di **testo** strutturato in **righe**, ognuna contenente 9 caratteri significativi;
- ogni riga termina con il carattere '\n'

☞ **ogni riga occupa in totale 10 bytes**

Note

- Come leggere un file a ritroso? (ricorda: il metodo di accesso è **sequenziale**) -> uso di **lseek!**

□ Per posizionare l'I/O pointer a fine file:

```
lseek(fd_in, 0, SEEK_END);
```

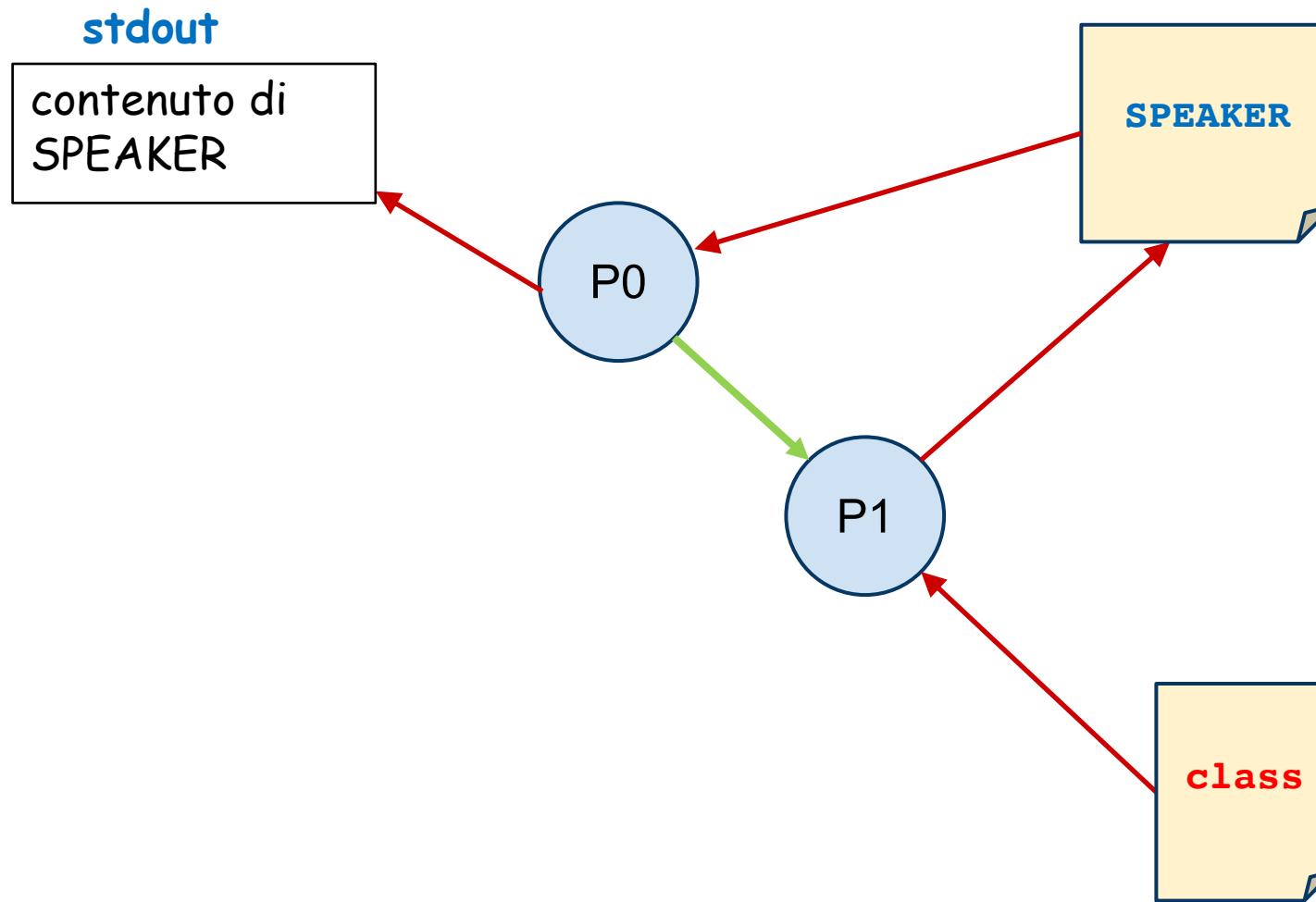
□ Per spostare l'I/O pointer sul byte precedente:

```
lseek(fd_in, -1 , SEEK_CUR);
```

□ Per spostare l'I/O pointer indietro di N byte:

```
lseek(fd_in, -N , SEEK_CUR);
```

Modello di soluzione



Esercizio 2 (1/2)

Si realizzi un programma C che, usando le opportune system call unix, assegna i premi ai partecipanti di un concorso.

A tale scopo il programma deve realizzare la seguente interfaccia:

`./assegna_premi punteggi premi`

dove **punteggi** e **premi** sono pathname di file di esistenti. In particolare:

- **punteggi** è un file di **testo** contenente una sequenza di righe ognuna contenente **il cognome e il punteggio** di un partecipante al concorso. **Si assume che il file sia ordinato in ordine crescente di punteggio** (cioè l'ultima riga contenga i dati relativi al partecipante con il punteggio massimo, la penultima quelli del candidato con il secondo punteggio ecc.)
- **premi** è un file **binario** contenente una sequenza di **N interi**, che rappresentano gli **N premi in denaro** del concorso, secondo la logica seguente: il primo intero rappresenta il primo premio, il secondo è il secondo premio, ecc.

Esercizio 2 (2/2)

Il regolamento del concorso stabilisce che gli N premi disponibili debbano essere assegnati agli N partecipanti con il punteggio maggiore, con la seguente politica: il primo premio è assegnato al partecipante con il punteggio massimo, il secondo al partecipante con il secondo punteggio, ecc.

A questo scopo P0 deve creare due figli P1 e P2 e attendere la loro terminazione.

- P2 legge le righe contenute in **punteggi** in senso inverso (cioè dall'ultima alla prima) e scrive ogni riga letta in un file di appoggio **fileINV** (nel quale, pertanto, le righe saranno ordinate in ordine decrescente di punteggio). Una volta conclusa questa operazione, manda un segnale a P1.
- P1, alla ricezione del segnale da P2, unisce il contenuto di **premi** e **fileINV** per scrivere in un nuovo file di testo **VINCITORI** (di N righe) il risultato dell'assegnazione, attribuendo ad ogni riga i-sima il seguente formato: “il sig. <cognome> ha vinto il premio n.<i> (euro <premio>) con il punteggio <punti>”.

esercizio 2 : esempio

premi

2000

1500

1000

punteggi

neri 10

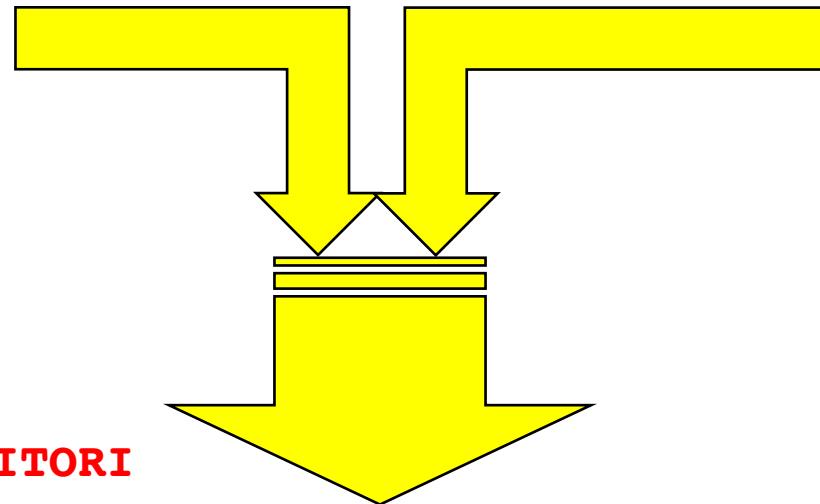
|-----|

lenzi 145

bassi 149

rossi 150

VINCITORI



Il sig. rossi ha vinto il premio n.1 (euro 2000) con il punteggio 150

Il sig. bassi ha vinto il premio n.2 (euro 1500) con il punteggio 149

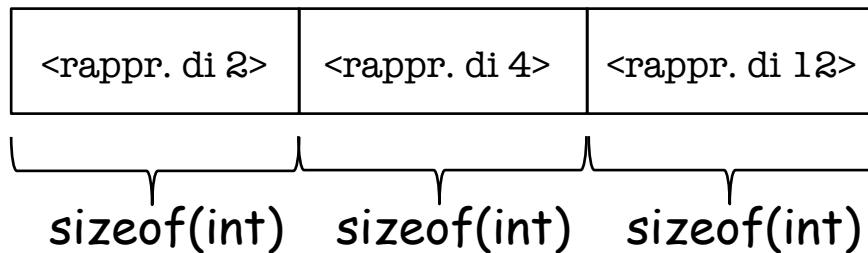
Il sig. lenzi ha vinto il premio n.3 (euro 1000) con il punteggio 145

Note

File Binario: ogni elemento è una sequenza di byte che contiene la **rappresentazione binaria** di un tipo di dato arbitrario.

Esempio:

file binario contenente la sequenza di interi [2,4,12]:



Lettura di file **binario** contenente una sequenza di **int**:

```
int VAR;  
read(fd, &VAR, sizeof(int));
```

Note: Come creare un File Binario?

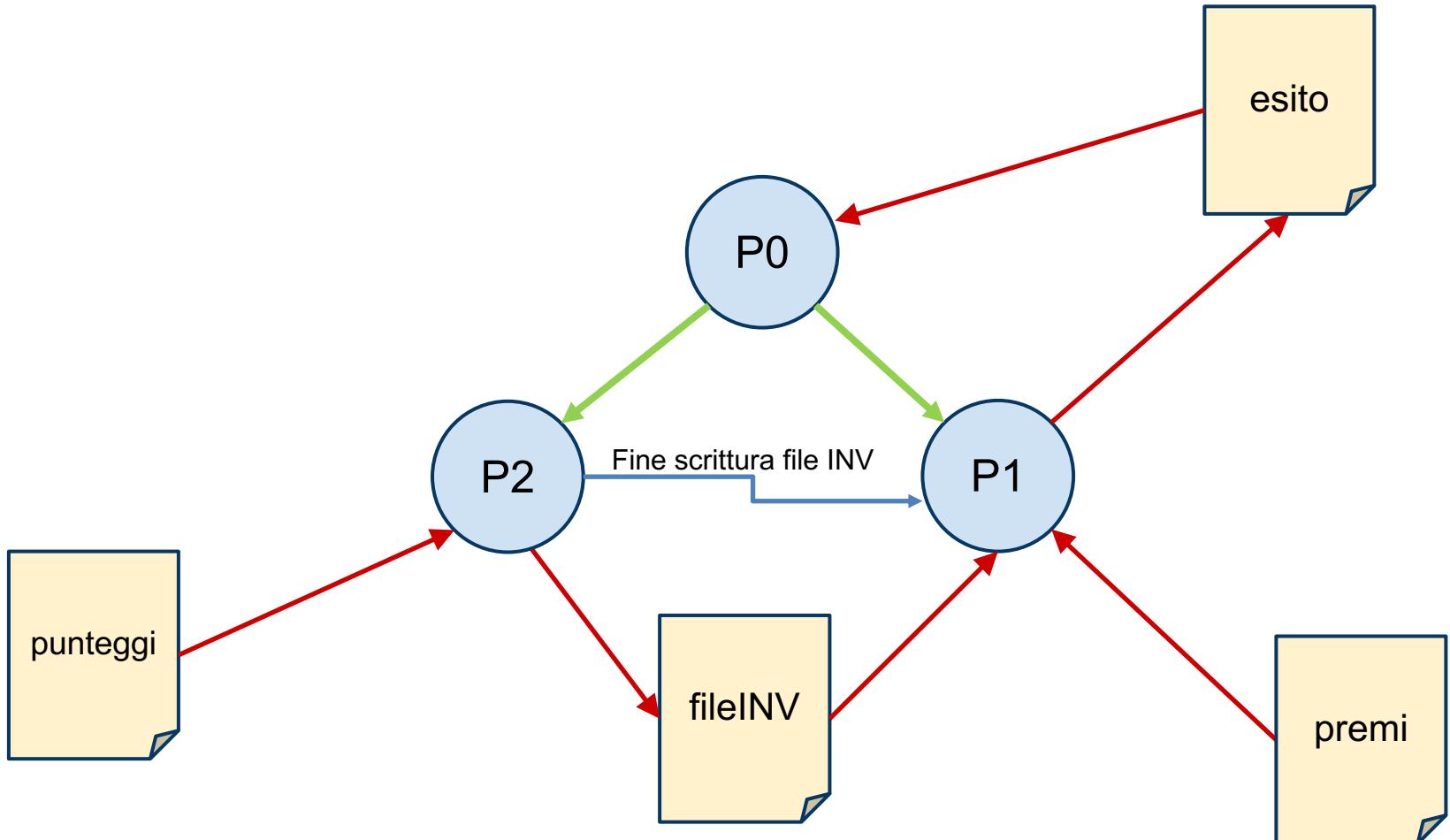
Esempio:

file binario contenente una sequenza di interi dati da standard input (v. crea_bin_int.c):

```
#define dims 25
int VAR, k;
int fd;
char buff[dims] = "";

fd=creat("premi", 0777);
printf("immetti una sequenza di interi (uno per riga),
terminata da ^D:\n"); // cntrl + D fornisce l'EOF a stdin
while (k=read(0, buff, dims)>0)
{
    VAR=atoi(buff);
    write(fd,&VAR, sizeof(int));
}
close(fd);
```

Modello di soluzione

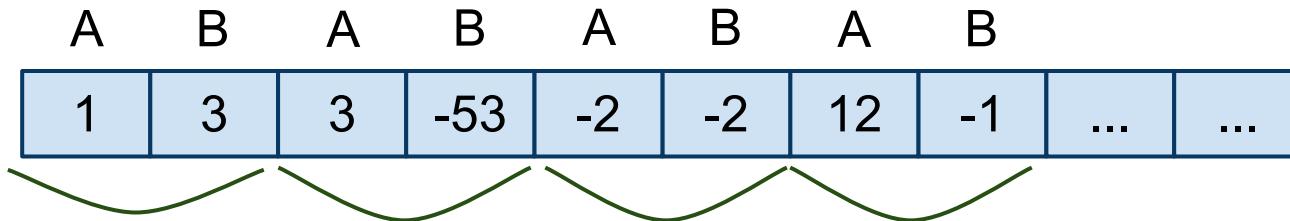


Esercizio 3 - Traccia (1/3)

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

`./correggi f_in f_out`

- **f_in**: path di un file **binario** esistente contenente N coppie (A,B) di numeri interi, con N non noto a priori.
- **f_out**: path di un file non esistente nel filesystem



L'obiettivo del programma è modificare il file **f_in** in modo che tutte le coppie siano ordinate in senso **non decrescente** (ovvero $B \geq A$)

Esercizio 3 - Traccia (2/3)

A tal fine, il programma deve realizzare il seguente comportamento:

- Il processo padre (**P0**) deve generare due figli P1 e P2
- Il processo **P2** deve
 - **Leggere** i due interi (A,B) di ogni coppia in **file_in**
 - Al termine di ogni lettura deve **segnalare a P1** se la coppia è ordinata in senso non decrescente (cioè $B \geq A$) oppure no (cioè $A > B$)
 - Letta l'ultima coppia, P2 deve **notificare** a P1 il termine della sua elaborazione

Esercizio 3 - Traccia (3/3)

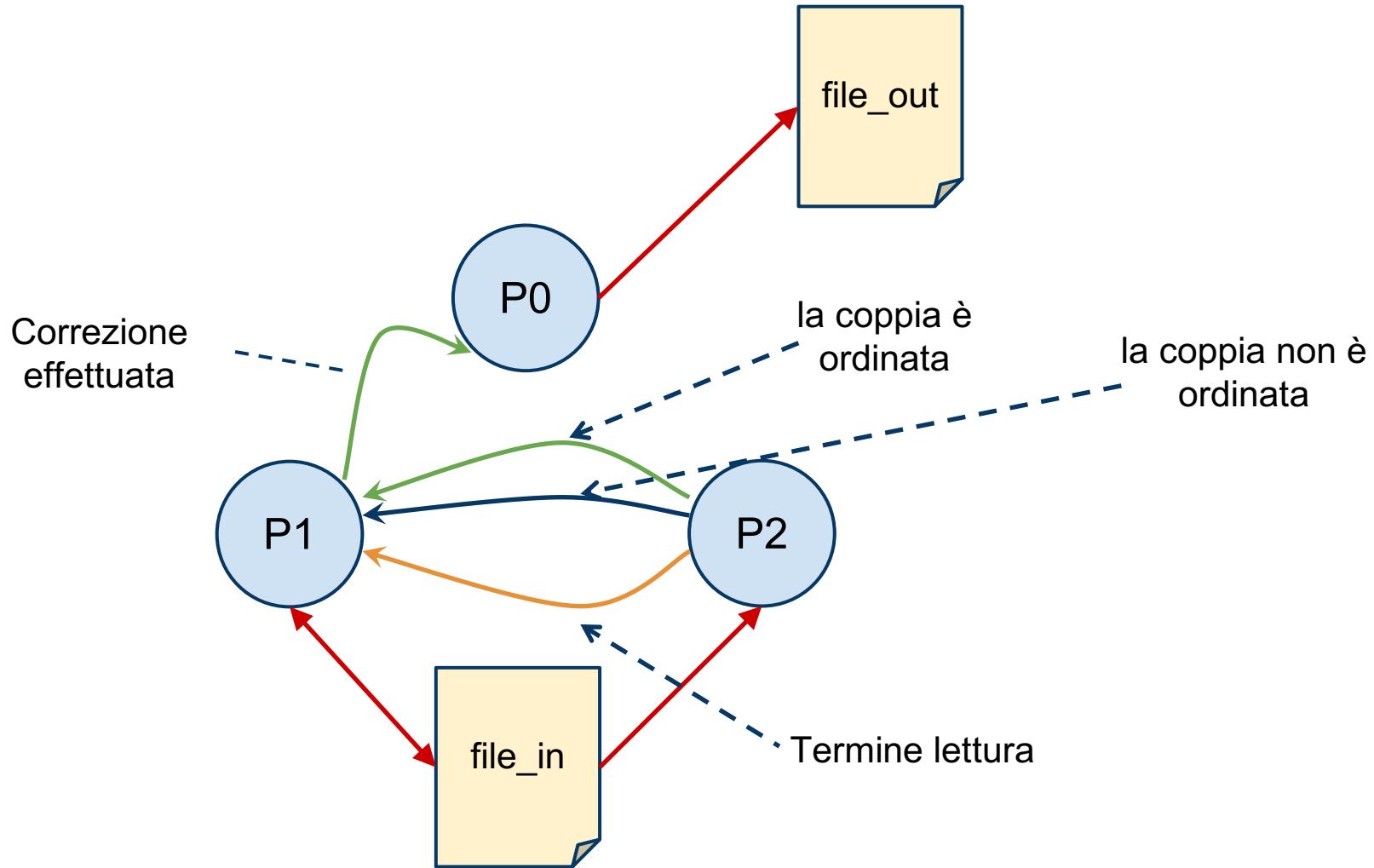
Il processo **P1** deve:

1. a seconda della segnalazione ricevuta da P2:
 - ❑ se la coppia corrente non è ordinata ($A > B$), provvederà a scambiare tra di loro i valori della coppia in `f_in` e notificherà a P0 l'avvenuta correzione.
 - ❑ se la coppia corrente è ordinata, non farà nulla.

Il processo **P0** deve

- Tener traccia del numero di correzioni effettuate
 - Al termine dell'elaborazione dei figli, scrivere tale valore su `file_out`
-

Modello di soluzione



Note

1. Come generare il file **f_in** ? (v. generate.c)
2. Il processo P1 deve leggere e scrivere sullo stesso file.
 - si può aprire lo stesso file due volte (**O_RDONLY** **O_WRONLY**), oppure
 - si può usare il flag **O_RDWR**
 - cosa conviene?
3. P1 e P2 lavorano sullo stesso file:
 - I/O pointer separati o condivisi ? cosa conviene fare?

Note

3. Le **lettura/scritture** di P1 e P2 avvengono **concorrentemente**
 - Come gestire il fatto che, ad esempio, P2 può eseguire più velocemente di P1?
 4. Si assume un **modello affidabile** dei segnali, in cui (contrariamente a quanto accade in linux) tutti i segnali ricevuti da un processo siano opportunamente accodati e non vengano mai accorpati
 - Si può rallentare opportunamente il ritmo di invio dei segnali (con opportune **sleep()**) per simulare questa assunzione
 5. P1 deve gestire tre tipi di segnali provenienti da P2
 - Si usino **SIGUSR1**, **SIGUSR2**, e **SIGALRM**
-