Esercitazione 4

Gestione dei file in Unix

Primitive fondamentali (1/2)

open	• 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	• **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	• **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** 0 per end-of-file -1 in caso di errore (perror e errno per sapere quale)
write	• **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	• **lseek(fd, offset, origine)** sposta l'I/O pointer di **offset** posizioni rispetto all'***origine***. Possibili valori per origine: 0 per inizio del file (SEEK_SET) 1 per posizione corrente (SEEK_CUR) 2 per fine del file (SEEK_END)

Esempio 1

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

Esempio 1 - 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 1 - Traccia (2/2)

Il programma deve realizzare il seguente comportamento:

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

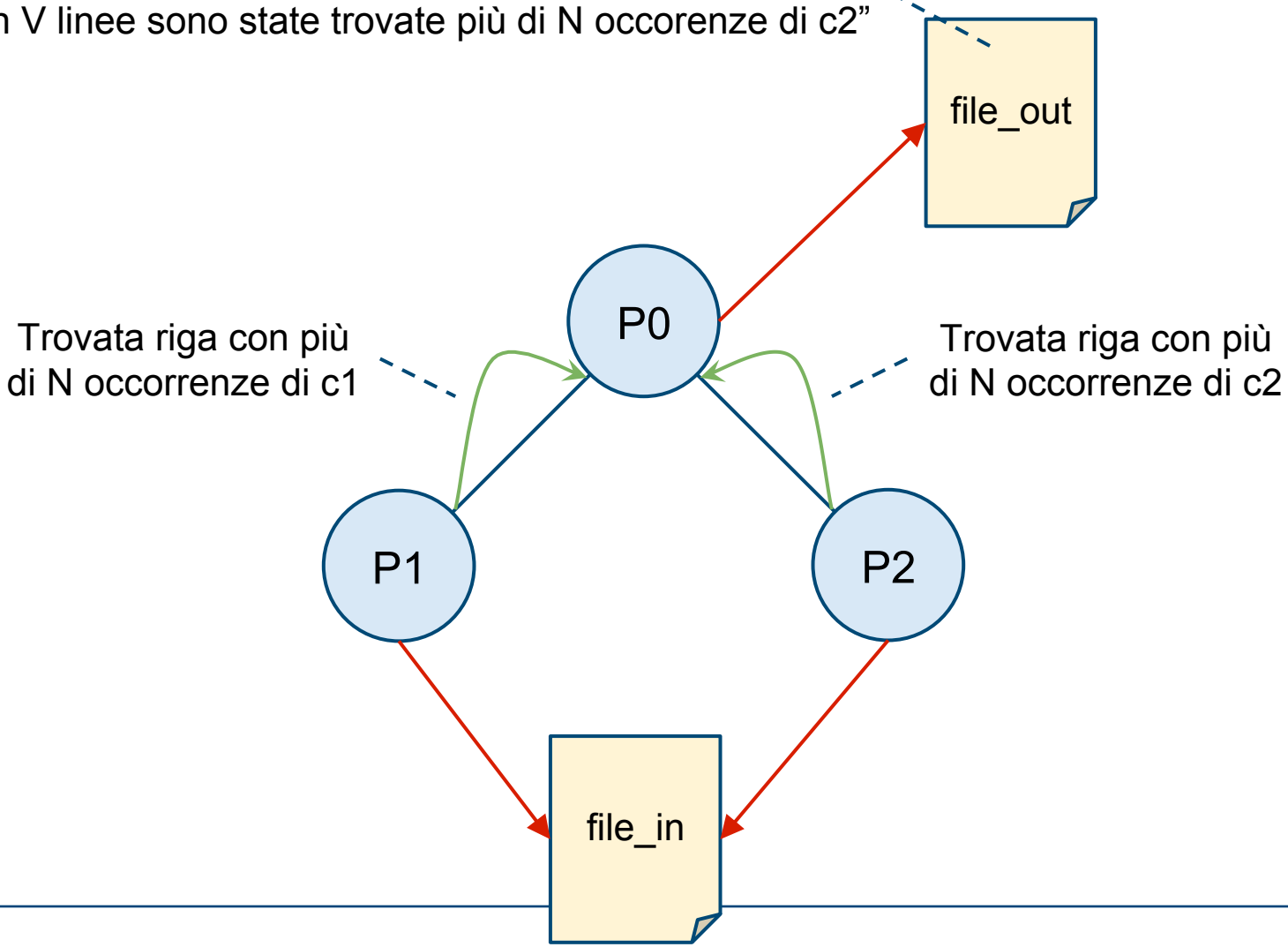
- Leggere **`file_in`** e contare, riga per riga, il numero di occorrenze rispettivamente del carattere **`c1`** (P1) e di **`c2`** (P2)
- **Al termine della lettura di ogni riga**, avvertire P0 se il numero di occorrenze trovate del carattere di competenza è maggiore di **`N`**

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 1 - Schema

“In L linee sono state trovate più di N occorenze di c1”
“In V linee sono state trovate più di N occorenze di c2”



Esempio 1 - Problematiche

Gestione dei file

- Flag di apertura (lettura o scrittura?)
- Gestione della lettura / scrittura
- I/O pointer condiviso o separato?

Realizzazione "comunicazione" figli-padre:

- Invio di segnali al processo P0
- È permesso l'uso della sleep per rallentare il ritmo di invio dei segnali da parte dei figli, onde evitare che segnali multipli vengano "accorpati" in uno

Esempio di soluzione (1/4)

```
int counter1, counter2;

int main(int argc, char* argv[]) {
    int pid, N, i;
    char to_check[NUM_CHARS];
    char *file_in, *file_out;
    ... /* 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 */}
    }
}
```

Esempio di soluzione (2/4)

```
    /* ... Continua main */
    for (i=0; i< 2; i++){
        wait_child();
    }
    print_output(file_out, N, to_check);
    return 0;
}

void father_handler(int signo){
    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);
    counter = 0;
    nread = read(fd, &read_char, sizeof(char));

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

Esempio di soluzione (3/4)

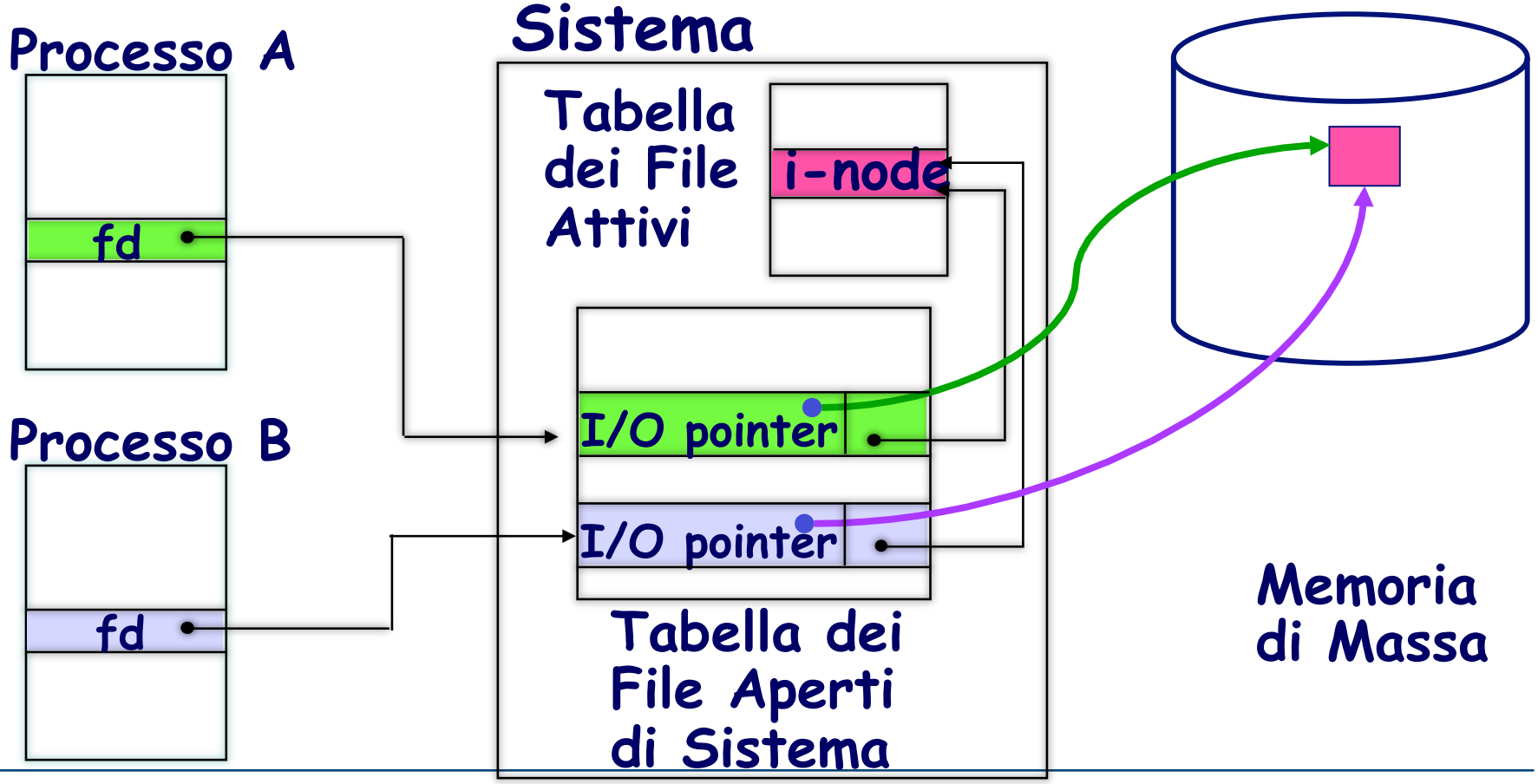
```
void figlio(char *input, char to_check, int limit, int sig_to_send){
    int fd, counter, nread; char read_char;
    fd = open(input, O_RDONLY);
    counter = 0;
    nread = read(fd, &read_char, sizeof(char));

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

Avrei potuto aggiornare qui direttamente i valori delle variabili globali **counter1** e **counter2**?
Le variabili globali sono condivise tra padre e figlio? NO!

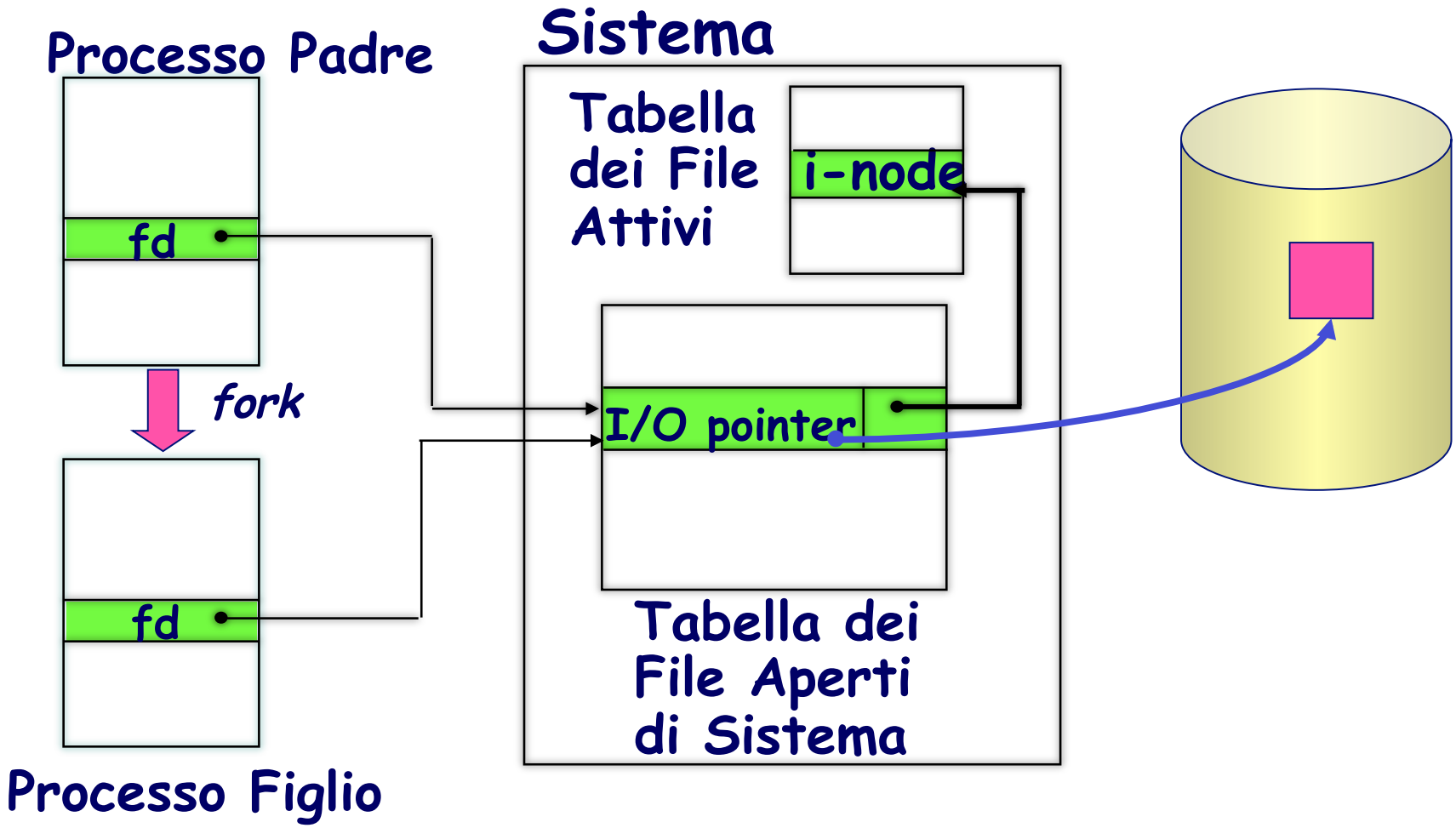
Esempio di soluzione (4/4)

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

Esempio: i processi A e B (indipendenti) accedono allo stesso file, ma con I/O pointer distinti



Esempio: processi *padre* e *figlio* condividono l'I/O pointer di file aperti prima della creazione.



Esercizio 2

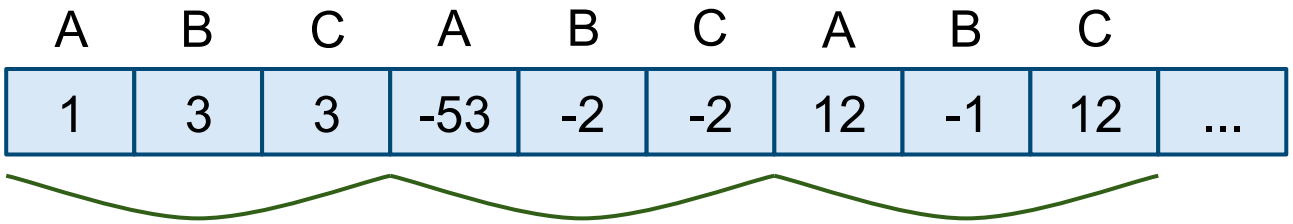
I/O e sincronizzazione avanzata
tra processi

Esercizio 2 - 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 triplette di numeri interi, con N non noto a priori.
- **`f_out`**: path di un file non esistente nel filesystem



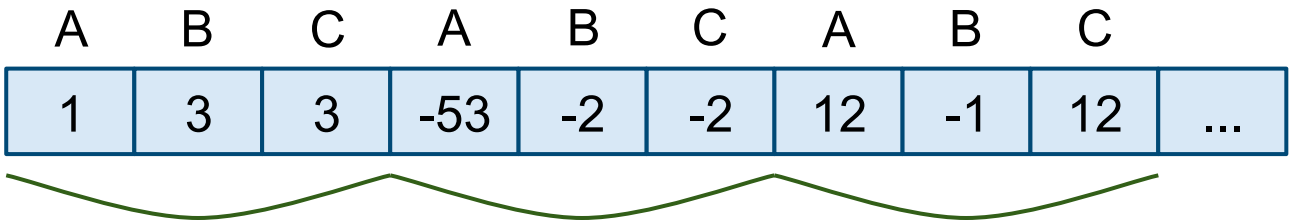
NB: file binario ≠ file di testo.

Quanti byte occupa "800667" su file binario? E su file di testo?

Esercizio 2 - 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
  - **Leggere** i primi due interi (A,B) di ogni tripletta in **`file_in`**
  - Al termine di ogni lettura deve **segnalare a P1** quale (A o B) è il maggiore
  - Letta l'ultima tripletta, comunicare a P1 il termine della sua elaborazione



Esercizio 2 - Traccia (3/3)

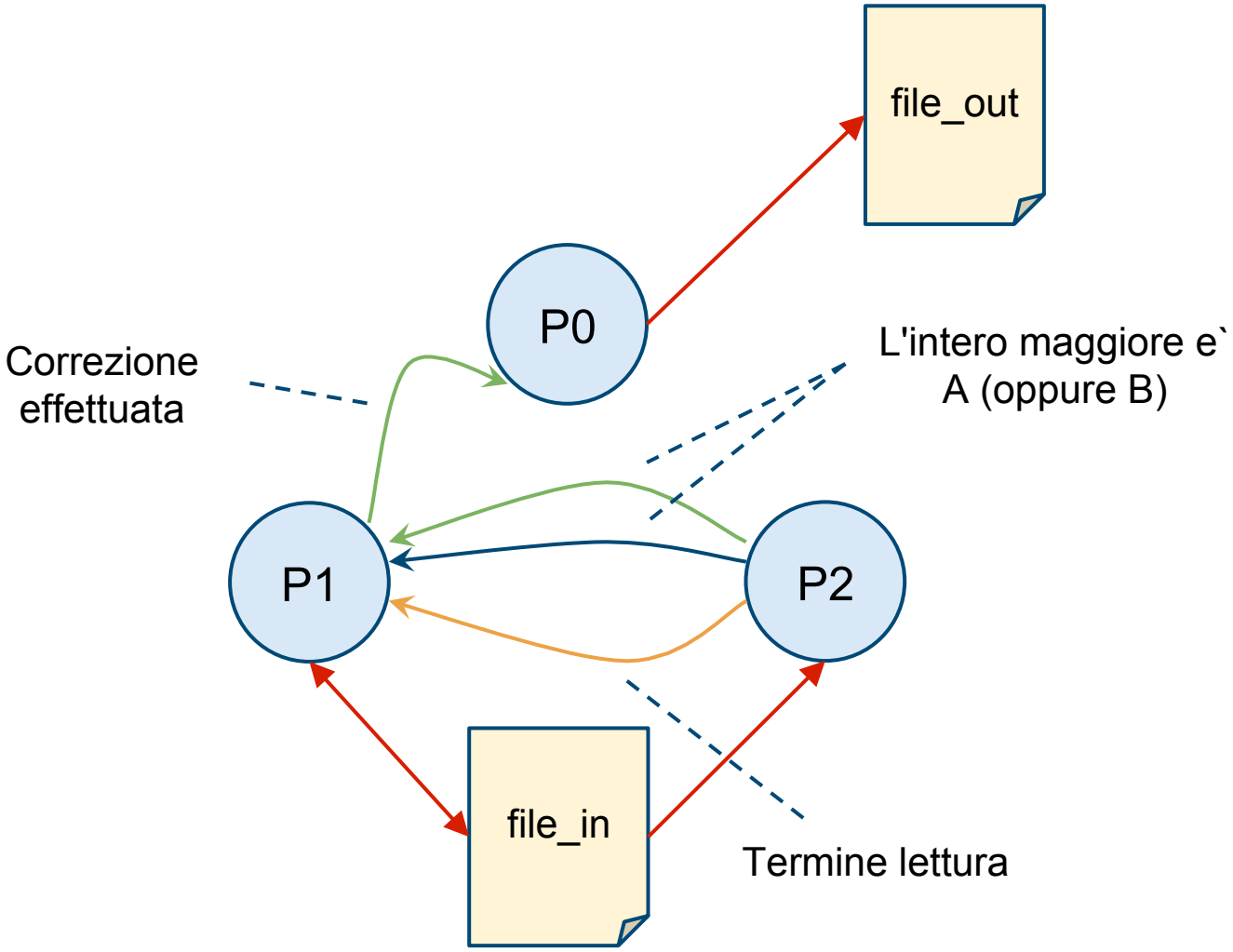
Il processo **P1** deve:

- **Reperire dal file l'intero maggiore** (A o B) a seconda della segnalazione ricevuta da P2
- **Leggere** il valore di C e, **nel caso in cui questo risultasse diverso** dal massimo appena letto
- Scrivere il valore dell'intero maggiore **al posto** del relativo elemento C della tripletta
- **Comunicare** a P0 l'avvenuta correzione

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

Lettura di file binario contenente una sequenza di interi:

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

Si assuma un **modello affidabile** dei segnali

- Tutti i segnali ricevuti da un processo sono opportunamente accodati e non vengono mai persi
- Si può rallentare opportunamente il ritmo di invio dei segnali (con opportune **`sleep()`**) per simulare questa assunzione

P1 deve gestire tre tipi diversi di segnali provenienti da P2

- Si usino **`SIGUSR1`**, **`SIGUSR2`**, e **`SIGALRM`**

Le letture/scritture di P1 e P2 avvengono **concorrentemente**

Come gestire il fatto che, ad esempio, P2 puo` eseguire piu` velocemente di P1?

Nota: eventi asincroni e sincronizzazione

```
1. {
2.     ...
3.   while ( condizione) {
4.     pause();
5.     do_something();
6.   }
7.       ...
8. }
9.
10.void handler(int signo){
11.   condizione = 0;
12.}
```

Nota: eventi asincroni e sincronizzazione

```
1.  {
2.      ...
3.    while ( condizione) {
4.      pause();
5.      do_something();
6.    }
7.        ...
8.  }
9.
10. void handler(int signo){
11.    condizione = 0;
12. }
```

Cosa succede se la sequenza di esecuzione è:

- linea 3: (condizione == 1)
- **arriva il segnale gestito da handler**
- linea 11: (condizione←0)
- linea 4: …

Nota: eventi asincroni e sincronizzazione

```
1.  {
2.      ...
3.    while ( condizione) {
4.      pause();
5.      do_something();
6.    }
7.        ...
8.  }
9.
10. void handler(int signo){
11.    condizione = 0;
12. }
```

Occorrerebbe rendere queste due istruzioni non interrompibili

Abbiamo bisogno di sincronizzare l'accesso dei processi alla variabile **condizione**.
Gli strumenti visti fin ora non sono sufficienti per risolvere il problema: servono meccanismi per l'**accesso atomico** a sezioni di codice.