Quarta Esercitazione

Gestione dei file in Unix

Andrea Reale
andrea.reale@unibo.it

Agenda

- **Esercizio 1**
  - Primi passi con le system call per l'I/O in Unix
- **Esercizio 2**
  - Sincronizzazione avanzata tra processi ed I/O condiviso

Tabella Riassuntiva

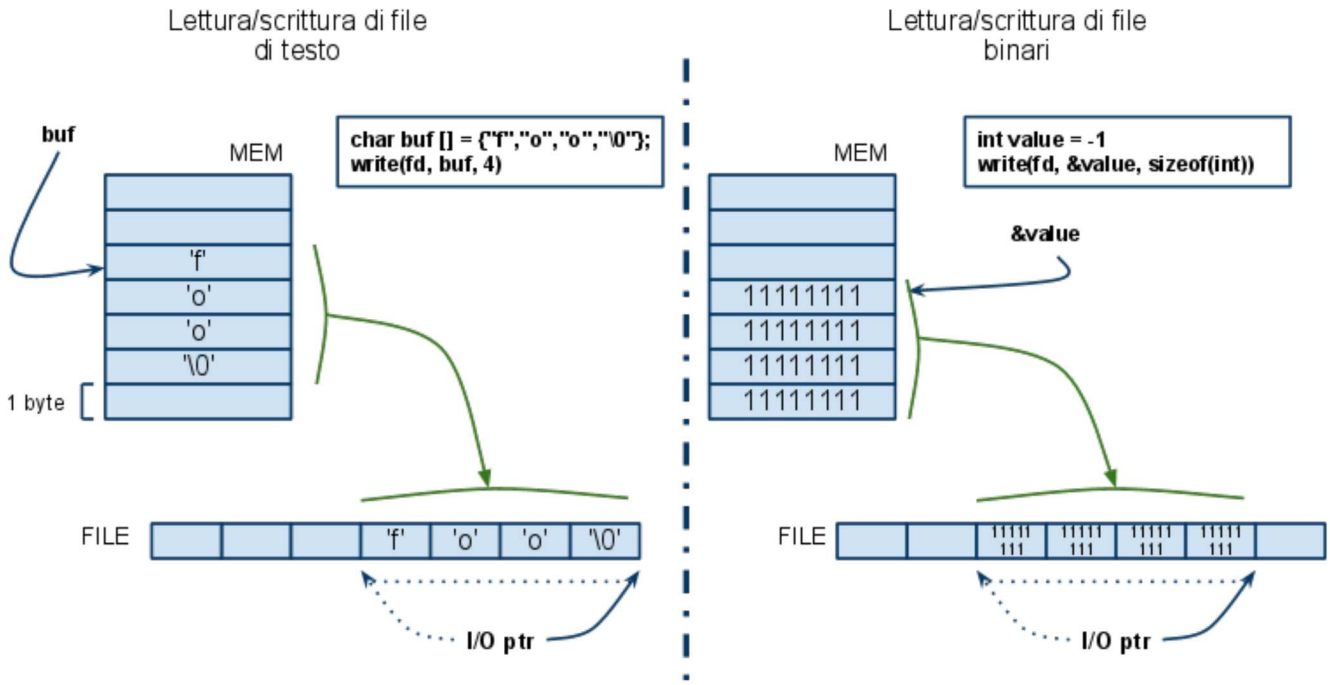
open	• Apre il file specificato e restituisce il suo **file descriptor** (fd) • Crea una nuova entry nella tabella dei file apert di sistema (**nuovo I/O pointer**) • fd è nello **spazio di memoria del processo** • possibili diverse **flag di apertura,** combinabili con OR bit a bit (operatore |)
close	• **Chiude** il file aperto • Libera il file descriptor nello spazio del processo • **Eventualmente** libera elimina elementi dalle tabelle di sistema

read	• Legge **al più** n bytes a partire dalla posizione dell'I/O pointer
	• Restituisce il **numero di byte effettivamente letti**
	  ○ 0 per end-of-file
	  ○ -1 in caso di errore (perror e errno per sapere quale)

write	• **Scrive n bytes** (in casi particolare <= n bytes) a partire dall'I/O pointer • Restituisce il numero di caratteri effettivamente scritti ○ -1 in caso di errore

Un breve richiamo su read e write

int read (int fd, char *buf, int n);
int write (int fd, char *buf, int n);



Note: sleep() e sincronizzazione

- Usare la **sleep()** come meccanismo di sincronizzazione, in generale, **non è una buona idea**.
  - Chi mi **assicura** che il tempo di "riposo" sia sufficiente perché un altro processo compia le azioni che mi aspetto faccia?
  - Se, invece, il tempo di riposo è eccessivo rispetto a quello necessario, grande perdita in velocità. (spesso è così: **granularità di sleep al secondo**)

- In esercizio "giocattolo" torna comodo per affrontare problemi pratici, e **concettualmente irrilevanti** agli scopi dell'esercizio.
  - es. aspettare "un po'" affinché un processo abbia il tempo di impostare gli handler di gestione ai segnali di interesse

Esercizio 1

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

Esercizio 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 rige di lunghezza non nota a priori
- **file_out**: path di un file di testo **non esistente** nel filesystem

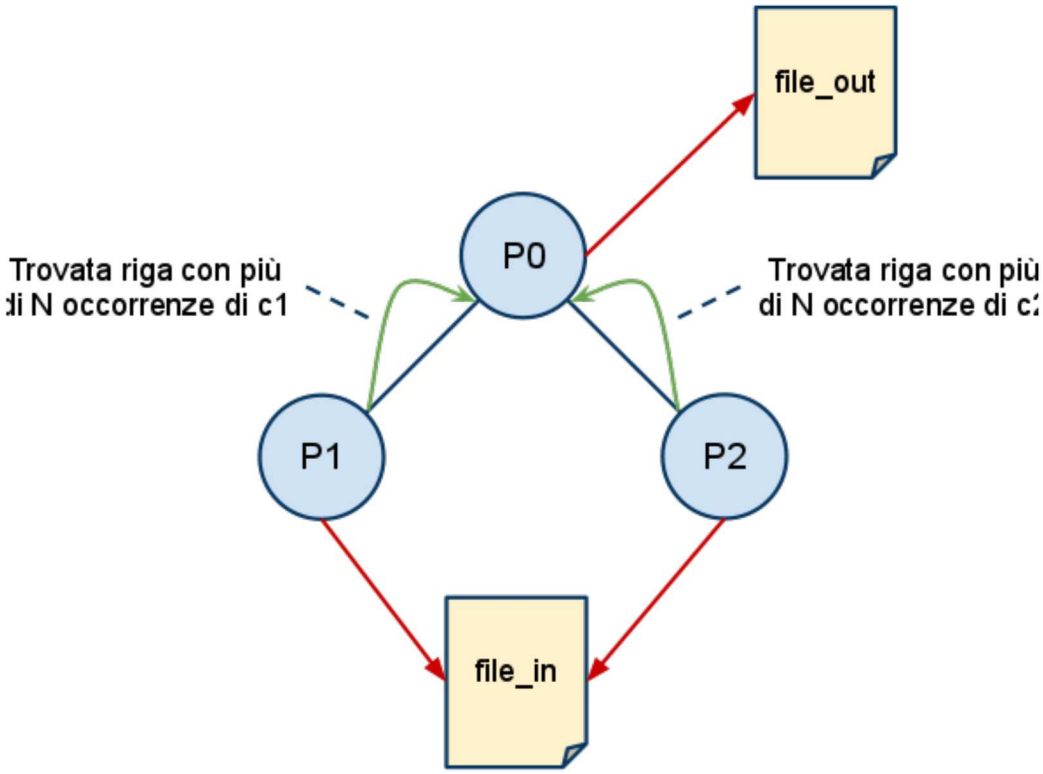
Esercizio 1 - Traccia (2/2)

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, rispettivamente il numer di occorrenze 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, s**eparatamente per ciascun figlio**, del numero di righe con più di **N** occorrenze dei caratteri cercati
  - **Una volta termine tutte le letture dei figli**, scrivere su **file_out** tale informazione

Esercizio 1 - Schema



Esercizio 1 - Problematiche

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

- Gestione IPC (Inter-process communication)
  - Quali strumenti?

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
         codice_figlio(file_in, to_check[i], N, sig_to_send);
         exit(EXIT_SUCCESS);
      }
      else
      { /* Codice Padre */}
   }
}
```

Esempio di soluzione (2/4)

```
   /* ... Countinua 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 codice_figlio(char *input, char to_check, int limit, int sig_to_send) {
   /* Dichiarazione variabili */
   /* Apertura separata file dell'input: I/O Pointer distinti. Sola lettura! */
   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);
            /* Evita di mandare altri sengali al padre mentre
             * e` ancora nel suo handler. Cosa succede se arrivano N segnali
             * uguali al padre mentre e` nel gestore? */
            sleep(1);
         }
         /* Resetta il contatore per la prossima linea */
         counter = 0;
      }
      /* Leggo il prossimo carattere per continuare il ciclo*/
      nread = read(fd, &read_char, sizeof(char));
      if (nread < 0) { /* Gestione errori */ }
   }
   /* Potrei essere arrivato ad EOF senza trovare il terminatore /n
    * cosa succede se intanto avevo trovato piu` caratteri di limit??*/
   close(fd);
}
```

Esempio di soluzione (4/4)

```
void print_output(char *pathname, int n, char *c) {
   /* Apro il file in sola lettura, se non esiste lo creo,
    * e se esiste cancello tutto il suo contenuto prima
    * di iniziarci a scrivere (n.b. equivalente a creat() )/*
   fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, 00640);
   if (fd < 0) { /* ..errore... /*}

   sprintf(buf,"In %d linee sono strate 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);
}
```

Esercizio 2

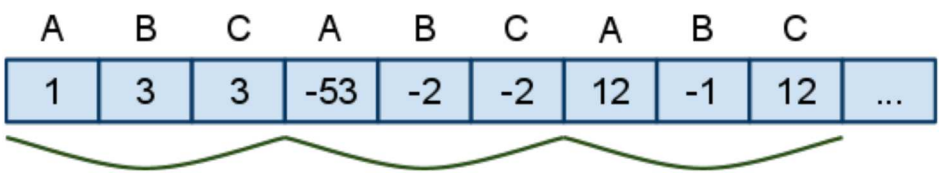
I/O e sincronizzazione avanzata tra processi

Esercizio 2 - Traccia (1/3)

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

./correggi file_in file_out

- **file_out**: path di un file non esistente nel filesystem
- **file_in**: path di un file binario esistente contenente N triplette di numeri interi, con N non noto a priori.
  - Esempio (**Attenzione**: non è un file di testo)



Esercizio 2 - Traccia (2/3)

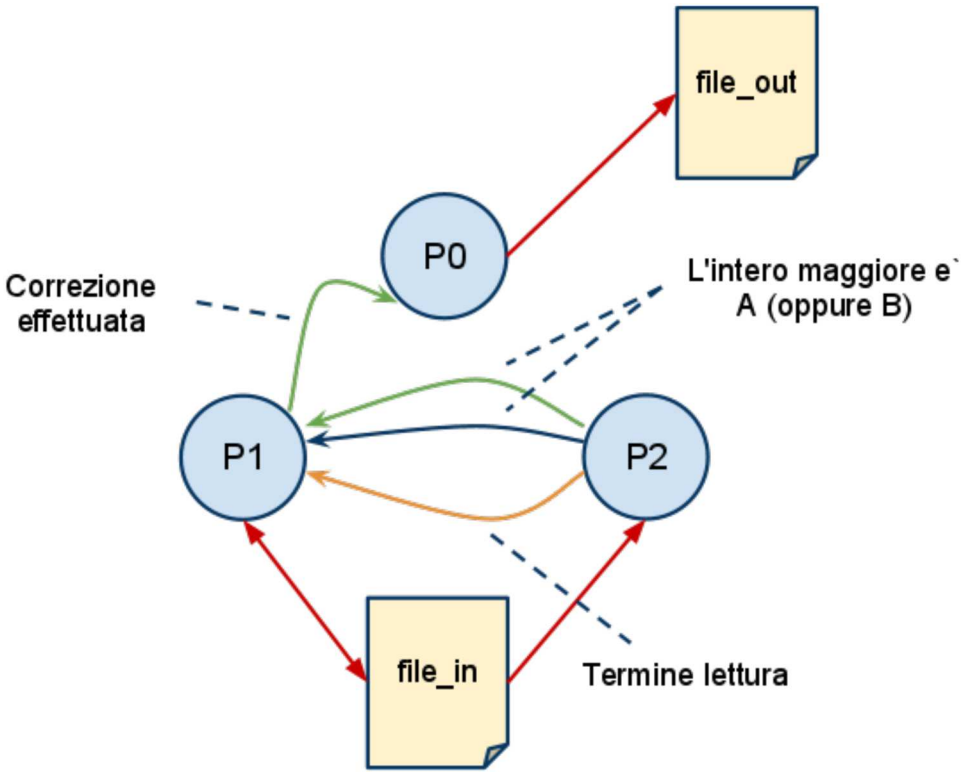
Il programma deve realizzare il seguente comportamento

- 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 della lettura dei primi due elementi di ogni tripletta, **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 alla soluzione

- Si assuma un **modello affidabile** dei segnali
  - Tutti i segnali ricevuti da un processo sono opportunamente accodati e non vengono mai persi
- 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 hndl(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 hndl(int signo) {
11.       condizione = 0;
12.    }
```

Cosa succede se la sequenza di esecuzione è:

- linea 3 (condizione == 1)
- **arriva il segnale gestito da hndl**
- linea 11 (condizione <-- 0)
- linea 4