Quinta Esercitazione

Strumenti di IPC - Pipe Unix

Andrea Reale
andrea.reale@unibo.it

Agenda

- **Esercizio 1**
  - Scambio di informazioni tra processi tramite pipe
- **Esercizio 2**
  - Redirezione di stdin e stdout su pipe

Tabella Riassuntiva

pipe	• **Crea** una pipe e scrive **i file descriptor relativi agli estremi** di lettura/scrittura sui primi due elmenti dell'array passato come argomento • Restituisce 0 in caso di creazione con successo, -1 in caso di errore
dup	• **Crea una copia** del descrittore passato come argomento • Per la copia, **viene usato il descrittore dal numero più basso di quelli disponibili** • Ritorna la nuova copia del descrittore, -1 in caso di errore
read	• Stessa system call usata per leggere file regolari • **Bloccante**: se la pipe è vuota il processo chiamante si blocca fin quando non ci sono dati disponibili
write	• Stessa system call usata per scrivere su file regolari • **Bloccante**: se la pipe è piena il processo chiamante si blocca fin quando non c'è spazio per scrivere
close	• Stessa system call usata per chiudere file descriptor di file regolari • Nel caso di pipe, usata da un processo per chiudere l'estremità della pipe che non gli interessa

Esercizio 1

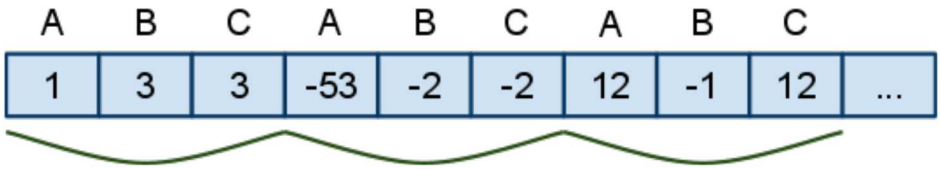
IPC e sincronizzazione tramite pipe

Esercizio 1 - 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 1 - 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, **comunicare a P1** il **valore** del maggiore
  - Letta l'ultima tripletta, segnalare a P1 il termine della sua elaborazione (chiusura lato pipe, aggiungere nota)

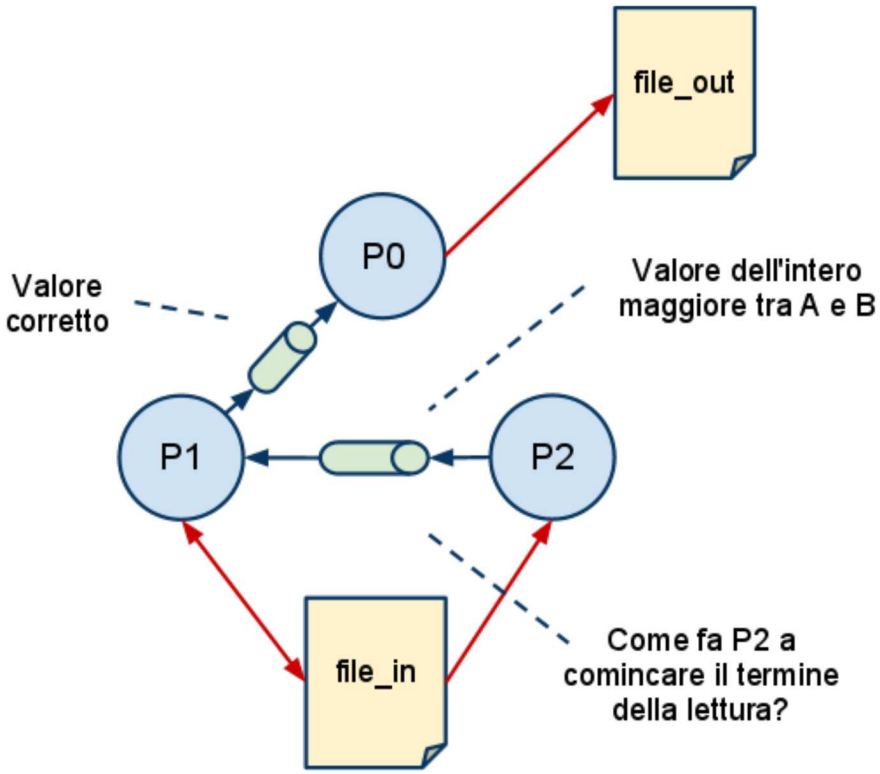
Esercizio 1 - Traccia (3/3)

- Il processo **P1** deve
  - **Leggere** il valore di C e, **nel caso in cui questo risultasse diverso** dal massimo **comunicatogli** da P2
    - Scrivere il valore dell'intero maggiore **al posto** del relativo elemento C della tripletta
    - **Comunicare** a P0 il valore corretto
- Il processo P0 deve
  - **Sommare i valori ricevuti da P1**
  - Al termine dell'elaborazione dei figli, scrivere tale valore su **file_out**

Esercizio 1 - Note alla soluzione

- Uso di **pipe** vs uso di **segnali** per sincronizzare processi
- Come fa P2 a comunicare a P1 il termine della sua elaborazione?
  - Un **segnale di fine**, o è possibile farlo anche con altri strumenti?

Modello di soluzione




```
int main(int argc, char *argv[]) {
   int pid, i, sum;
   int pipe_p0_p1[2], pipe_p1_p2[2];

   /* Creo le pipe prima di generare i figli così da condividerle
      con loro (e tra di loro) */
   i = pipe(pipe_p0_p1); /* .. manca il controllo errori */
   i = pipe(pipe_p1_p2);  /* .. manca il controllo errori */

   for (i=0; i<2; i++) {
      pid = fork();
      if (pid == 0) {
         if (i==0) { /*P1*/
            /*Chiudo gli estremi che non interessano a P1*/
            close(pipe_p0_p1[0]); close(pipe_p1_p2[1]);
            processo_p1(pipe_p1_p2[0], pipe_p0_p1[1], argv[1]);
            exit(EXIT_SUCCESS);
         }
         else if (i==1) { /*P2*/
            close(pipe_p0_p1[0]); close(pipe_p0_p1[1]);
            close(pipe_p1_p2[0]);
            processo_p2(pipe_p1_p2[1], argv[1]);
            exit(EXIT_SUCCESS);
         }
      }
   }
```

```
        /* ... CONTINUA IL MAIN ... */

       /* Entrambi i figli usciranno prima che arrivino ad eseguire
        * le istruzione che seguno,
        * che sarnno dunque eseguite solo dal padre */

      /* Chiusura dei lata delle pipe non di interesse per P0*/
      close(pipe_p1_p2[0]); close(pipe_p1_p2[1]); close(pipe_p0_p1[1]);

      /* P0: procedura di lettura dei dati da P1. Dopo aver letto tutti
       * i dati, restituisce la loro somma */
      sum = p0_read_from_p1(pipe_p0_p1[0]);
      /* P0: stampa della somma sullo standard output */
      print_output(sum, argv[2]);

      /* Attesa della fine dei figli */
      for ( i=0; i<2; i++) wait_child();
      return 0;
  }
```

```
int p0_read_from_p1( int pipe_read_end ) { /* P0 */
    int value, nread, sum = 0;
    /* Quando P1 chiuderà il suo lato di scrittura nread == 0 */
    while ( (nread = read(pipe_read_end, &value, sizeof(int))) > 0 ) {
        sum += value;
    }

    if (nread < 0) {
        perror("P0: errore di lettura nella pipe con P1");
        close(pipe_read_end);
        exit(EXIT_FAILURE);
    }
    close(pipe_read_end);
    return sum;
}
void print_output(int sum, char *outputpath) { /* P0 */
    int fd, written;
    char buf[MAX_STRING_LENGTH];
    fd = creat(outputpath, 00640);
    if (fd < 0){ /* Gestione errori */ }

    written = write(fd, &sum, sizeof(int));
    if (written < 0) { /* Gestione errore di scrittura */ }

    return;
}
```

```c
void processo_p2(int pipe_wr_end, char *inputfile) { /* P2 */
    int fd, nread, read_buf[2], written;
    fd = open(inputfile, O_RDONLY);
    if ( fd < 0 ) {
        perror("P2: errore nell'apertura del file di input");
        /* Segnalo a p1 la fine della lettura */
        close(fd); close(pipe_wr_end);
        exit(EXIT_FAILURE);
    }

    /*Due interi alla volta, fino a EOF*/
    while ( (nread = read(fd, read_buf, 2*sizeof(int))) > 0 ) {
        /* ... */
        /* Scrivo sulla pipe "destinata" a P1 il maggiore */
        written = write(pipe_wr_end, &max, sizeof(int));
        if (written < 0) {
            perror("P2: Errore nella scrittura sulla pipe");
            close(fd); close(pipe_wr_end);
            exit(EXIT_FAILURE);
        }
        /*Devo saltare la entry corrispondente a C */
        lseek(fd, sizeof(int), SEEK_CUR);
    }

    /* Fine elaborazione: lo segnalo a P1 chiudendo l'estremo della pipe*/
    close(fd);
    close(pipe_wr_end);
}


void processo_p1(int pipe_p1p2_rd, int pipe_p0p1_wr, char *inputfile) {
    int fd, nrw, sk, nread, max_value, c_value;

    fd = open(inputfile, O_RDWR);
    if ( fd < 0 ) { /* Chiusura delle pipe, gestione errori */ }
    /* Fin quando P2 non chiude il suo estremo */
    while ( (nread = read(pipe_p1p2_rd, &max_value, sizeof(int))) > 0 ) {
        /* Spostamento su C */
        lseek(fd, 2*sizeof(int), SEEK_CUR);
        nrw = read(fd, &c_value, sizeof(int));
        if(nrw < 0) { /* Chiusura file e pipe, gestione errori */ }

        if ( max_value != c_value) {
            /* Sposto indietro il cursore*/
            lseek(fd, -sizeof(int), SEEK_CUR);
            /* Srivo il primo elemento di buf su C*/
            nrw = write(fd, &max_value, sizeof(int));
            if(nrw < 0) { /*Chiusura file e pipe, gestione errori */ }
            /*Comunico a P0 il valore della correzione*/
            nrw = write(pipe_p0p1_wr, &max_value, sizeof(int));
            if (nrw < 0) { /*Chiusura file e pipe, gestione errori */}
        }
    }
    if (nread < 0) { /* Chiusura file e pipe, gestione errori */ }

    close(fd); close(pipe_p1p2_rd); close(pipe_p0p1_wr);
}
```

Pipe - Riflessioni post-esercizio

- Le **pipe** sono uno strumento di comunicazione tra processi
  - Consentono a processi in gerarchia di scambiarsi dati
  - Primitive di accesso **omogenee** rispetto a quelle per file regolari

- Alcune differenze
  - Read e write **bloccanti** (se la pipe è risp. vuota/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 non utilizzate di una pipe?

- Queste proprietà rendono la pipe anche un possibile **strumento di sincronizzazione** tra processi
  - **Quando conviene utilizzare pipe e quando segnali?**

Esercizio 2

Redirezione di stdin e stdout su pipe

Esercizio 2 - Traccia (1/3)

Si realizzi un programma C che, utilizzando le system call di Unix, abbia interfaccia di invocazione:

./cerca file1 file2 string

- **file1** e **file2**: path di file di testo esistenti nel file system. Ciascuno ha un numero variabile di righe, ognuna delle quali ha lunghezza massima di 100 caratteri.

- **string:** stringa non vuota

Esercizio 2 - Traccia (2/3)

Il processo padre **P0** genera due figli (P1 e P2)

- **P1** deve:
  - Leggere file1 e contare il numero **M** di righe in cui string compare almeno una volta
  - Ricevere da **P2** il numero **N** di righe contenenenti string in file2

Una volta ottenuti sia **M** che **N**, comunicarli a **P0**

- **P2** deve:
  - Contare in file2 il numero **N** di righe in cui string compare almeno una volta tramite il comando **grep**
  - Comunicare tale valore a **P1** tramite opportuna redirezione dello **stdout** di **grep**

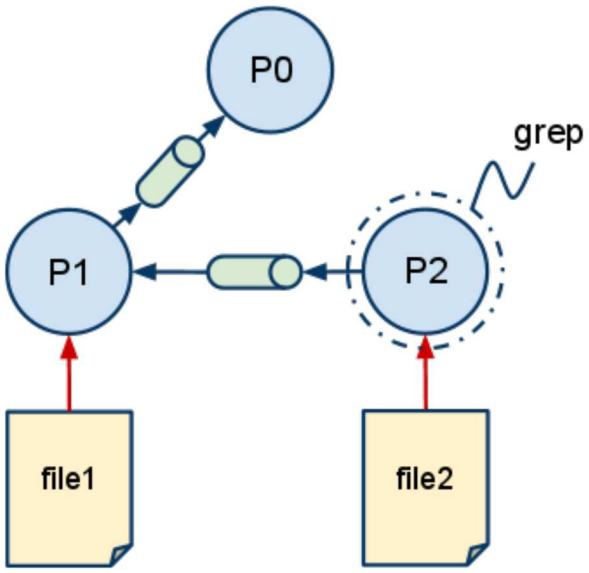
- **P0** deve:
  - Stampare a video il numero totale di occorrenze di string in file1 e file2

Suggerimenti

- P2 deve contare tramite l'esecuzione del comando **grep**
  - **grep -c string file2**
  - **man grep**
  - **Attenzione:** di che tipo è l'output di grep? Caratteri o binario?

- P1 deve contare le righe contenenti string
  - Usare la funzione di libreria strstr in <string.h>
  - **man strstr**

Modello di soluzione



Variante

- Si supponga di non poter usare l'opzione **-c** per **grep**
  - **grep string file2** stampa in output le righe di file2 contenenti string

wc -l conta il numero di linee nel suo standard input

  - **Idea**: piping di grep e wc

Un ulteriore fratello **P3** lancia **grep in pipe** verso P2

- **P2** esegue il comando wc