ESERCITAZIONE 9/05/2024

NOTA BENE:

a) Per ognuno dei programmi C che devono essere scritti per questa esercitazione (e per le prossime) deve essere scritto anche il makefile che produce il file eseguibile controllando tutti i possibili WARNING di compilazione (opzione –Wall) che devono sempre essere eliminati, come chiaramente anche gli errori di compilazione e di linking! Si ricorda, inoltre, di fare attenzione al fatto che, se si usa un particolare standard del linguaggio, anche questo va specificato dato che non si può presupporre che chi deve verificare il funzionamento abbia lo stesso standard di default!

b) Una volta ottenuto l'eseguibile, va chiaramente verificato il funzionamento; in caso di passaggio di parametri va verificato anche che i controlli funzionino correttamente!

c) Si ricorda che qualunque stampa (in particolare, usando la printf) che sia aggiunta rispetto a quanto richiesto dalla specifica è opportuno che venga segnalata dalla stringa "DEBUG"!

1. Con un editor, scrivere un programma in C `provaPipe-bis.c` che deve prendere ispirazione da due programmi mostrati a lezione (disponibili su **GITHUB**): i due programmi sono `provaopen.c` e `provaPipe.c`. Tale programma deve avere solo *due parametri di invocazione* che devono essere nomi di file (come in `provaopen.c`). Dopo aver aperto i due file passati come parametri (salvando i valori ritornati dalle open nelle variabili **fd1** e **fd2)**, si proceda alla chiusura di **fd1** (in modo assolutamente analogo a quanto fatto nel programma `provaopen.c`), quindi -invece che aprire un altro file- si deve creare una pipe (in modo assolutamente analogo a quanto fatto nel programma `provaPipe.c`): si stampino, infine, i valori dei due file descriptor della pipe (sempre come fatto nel programma `provaPipe.c`). Verificarne il funzionamento sia con un numero di parametri sbagliato, che giusto, ma file non esistenti che esistenti: cosa si osserva come valore del lato di lettura della pipe?
2. Con un editor, scrivere un programma C `pipe-SenzaClose.c` che, partendo dal programma `pipe.c` mostrato a lezione (disponibile su **GITHUB**), elimini nel figlio la chiusura del lato di lettura della pipe ed elimini nel padre la chiusura del lato di scrittura della pipe. **Verificare, quindi, che la mancanza delle chiusure dei lati delle pipe non utilizzati porta a un comportamento scorretto, cioè ad un DEADLOCK che sarà risolvibile solo abortendo l'esecuzione dei due processi con un ^C!**
3. Con un editor, scrivere un programma C `pipe-Generico1.c` che, partendo sempre dal programma `pipe.c`, deve andare a considerare però un diverso protocollo di comunicazione: in particolare deve essere previsto che il figlio invii al padre stringhe di lunghezza generica e che il figlio, dopo aver letto una linea (*supposta non più lunga di 512 caratteri compreso il terminatore di linea*) dal file passato come parametro e averla trasformata in stringa (come nel programma originale), deve mandare al padre *per prima cosa* la lunghezza della stringa, *compreso il terminatore*, e poi deve mandare la stringa; quindi il padre, per ogni linea letta dal figlio, riceve *per prima cosa* la lunghezza della stringa e poi deve usare questa informazione per leggere il numero di caratteri che costituiscono la stringa; chiaramente, il padre alla fine deve stampare via via le stringhe ricevute, analogamente al programma originale. Verificarne il funzionamento *i)* con un numero di parametri sbagliato, *ii)* con un numero di parametri giusto, ma file non esistente e *iii)* con un numero di parametri giusto e file esistente; in particolare, si verifichi il funzionamento sia con i file usati per la verifica del programma `pipe.c` (`input.txt` e `input1.txt`, entrambi disponibili su **GITHUB**) che, chiaramente, con almeno un file che presenta lunghezze di linee NON fisse e NON note a priori!
   OSSERVAZIONE 1: Si ricorda che il figlio, per leggere una linea da un file, deve leggere un carattere alla volta, dato che si deve cercare il carattere '**\n**', salvando i caratteri nel buffer `mess` (di lunghezza 512, come specificato nel testo) e quindi il modo più furbo per farlo è questo:
   `while (read (fd, &(mess[L]), 1) != 0)` supponendo che `L` sia l'indice, inizializzato a 0, che serve per inserire i singoli caratteri nel buffer `mess`.
   OSSERVAZIONE 2: L'indice `L`, incrementato di 1 (dato che si deve mandare la lunghezza della stringa, *compreso il terminatore*) deve essere la prima informazione inviata al padre.
   OSSERVAZIONE 3: Per l'invio di queste due informazioni (lunghezza della stringa, rappresentata da `L`, e stringa, rappresentata da `mess`) si **DEVE** usare la stessa pipe e NON se ne deve creare un'altra!

OSSERVAZIONE 4: Per l'invio della stringa rappresentata da `mess` si DEVE fare attenzione ad usare nella *write* il valore di `L` che rappresenta appunto la lunghezza della stringa, *compreso il terminatore*!

4. Con un editor, scrivere un programma C `pipe-Generico2.c` che, partendo sempre dal programma `pipe.c`, deve andare a considerare però un ulteriore protocollo di comunicazione: in particolare, il figlio, dopo aver letto una linea (sempre *supposta non più lunga di 512 caratteri compreso il terminatore di linea*) dal file passato come parametro e averla trasformata in stringa (come nel programma originale), deve mandare *direttamente* la stringa al padre (come nell'esercizio originale - si consideri però l'**OSSERVAZIONE 4** precedente!); quindi il padre, poiché diversamente dall'esercizio originale e dal precedente esercizio, **NON** conosce la lunghezza delle singole stringhe inviate dal figlio, deve ricevere via via i singoli caratteri inviati dal figlio fino al terminatore di stringa e poi (*solo dopo aver ricevuto TUTTA la stringa*, dato che deve incrementare il contatore delle stringhe ricevute) deve stampare la stringa corrente e così via per ogni stringa inviata dal figlio, analogamente al programma originale.
   OSSERVAZIONE: Si segnala che lo stesso schema usato per leggere una linea da un file da parte del figlio si deve analogamente usare nel padre per andare a leggere un carattere alla volta dalla pipe, dato che si deve cercare il carattere '**\0**', salvando i caratteri nel buffer `mess` (di lunghezza 512, come specificato nel testo) e quindi il modo più furbo per farlo è questo:
   `while (read(piped[0], &(mess[L]), 1) != 0)` supponendo sempre che `L` sia l'indice, inizializzato a 0, che serve per inserire i singoli caratteri nel buffer `mess`.

NOTA: I due esercizi che seguono traggono ispirazione da due vecchi testi di esame.

5. Con un editor, scrivere un programma C `provaEsame1.c` che deve prevedere *un numero variabile* **N** *di parametri* (*con* **N** *maggiore o uguale a 2, da controllare*) che rappresentano nomi di file (**F1**, **F2**. ... **FN**). Il processo padre deve generare **N** processi figli (**P0, P1, … PN-1**): i processi figli **Pi** (con **i** che varia da 0 a N-1) sono associati agli **N** file **Ff** (**con f = i+1**). Ogni processo figlio **Pi** esegue in modo concorrente e deve leggere dal proprio file associato, comunicando al padre una selezione dei caratteri letti: in particolare, i processi associati ai file dispari (**F1, F3, ...**) devono selezionare *solo* i caratteri alfabetici, mentre quelli associati ai file pari (**F2, F4, ...**) *solo* i caratteri numerici. Il padre deve scrivere i caratteri ricevuti dai figli su standard output, *alternando un carattere alfabetico e uno numerico*, chiaramente fino a che l'alternanza può essere mantenuta, ma comunque tutti i caratteri ricevuti devono essere scritti sullo standard output. *Una volta ricevuti tutti i caratteri*, il padre deve stampare su standard output il numero totale di caratteri ricevuti e, quindi, scritti sullo standard output. Al termine, ogni processo figlio **Pi** deve ritornare al padre *l'ultimo carattere* letto dal proprio file associato. Il padre deve stampare, su standard output, i pid di ogni figlio con il corrispondente valore ritornato.
   OSSERVAZIONE: Vista la specifica di questo testo, la cosa più semplice è definire **due** pipe: per ottenere un codice 'compatto', si suggerisce di usare un array statico di 2 pipe e quindi `int p[2][2];` la pipe `p[0]` può essere usata per la comunicazione fra i processi (dispari) che leggono dai file pari ed inviano caratteri numerici al padre, mentre la pipe `p[1]` può essere usata per la comunicazione fra i processi (pari) che leggono dai file dispari e inviano caratteri alfabetici al padre. **N.B.** Si può scegliere anche la semantica invertita: l'importante è selezionare la pipe giusta quando si devono chiudere i lati che non si usano e si deve inviare al padre!
   ATTENZIONE: la stampa fatta dal padre del valore tornato dal figlio deve considerare che il figlio torna un char! Si consiglia di aggiungere anche la stampa del valore tornato come int per intercettare eventuali errori.
6. Con un editor, scrivere un programma C che deve prevedere *un numero variabile* **N+1** *di parametri* (*con* **N** *maggiore o uguale a 2, da controllare*): il primo parametro rappresenta il nome di file (**F**) e gli altri N rappresentano *singoli* caratteri (**C1** ...**CN**, *da controllare*). Il processo padre deve generare **N** processi figli (**P0, P1, … PN-1**): i processi figli **Pi** (con **i** che varia da 0 a N-1) sono associati agli **N** caratteri **Cf** (**con f = i+2**). Ogni processo figlio **Pi** esegue in modo concorrente ed esamina il file **F** contando le occorrenze del carattere associato **Cf**: per il conteggio deve essere usata una variabile di tipo *long int*; terminata la lettura del file **F**, ogni figlio **Pi** deve comunicare al padre quante occorrenze di **Cf** ha trovato. Il padre deve ricevere i conteggi inviati dai figli e li deve riportare sullo standard output specificando a quale carattere **Cf** si riferisce ogni singolo conteggio. Al termine, ogni processo figlio **Pi** deve ritornare al padre il carattere **Cf** associato. Il padre deve stampare su standard output il pid di ogni figlio e il valore ritornato.

ATTENZIONE: rispetto alla stampa fatta dal padre del valore tornato dal figlio si consideri quanto detto per l'esercizio precedente!

Per risolvere questo problema si possono usare due protocolli di comunicazione; quindi si devono scrivere DUE programmi C, uno di nome `provaEsame2-a.c` **e uno di nome** `provaEsame2-b.c`**:**

6.a Nel programma `provaEsame2-a.c` si devono usare N pipe (una per ogni processo figlio su cui viene scritto il numero di occorrenze del carattere **Cf** associato): in tale modo, il padre può individuare a quale carattere si riferisce il conteggio inviato ricavandolo dall'indice della pipe!
OSSERVAZIONE: Vista la specifica di questa versione per ottenere N pipe si DEVE seguire quanto mostrato nella lezione di Mercoledì 9 Maggio e che si riporta per l'esercizio corrente qui nel seguito!

6.b Nel programma `provaEsame2-b.c` si deve usare una sola pipe su cui ogni figlio scrive una *struct* con due campi (il carattere **Cf** e il numero *long int* di occorrenze); questa soluzione può essere accettabile visto che il testo non richiede che le informazioni inviate dai figli vengano recuperate dal padre in un ordine specifico.

OSSERVAZIONE: ***nei testi recenti viene specificato se va usata una struct o altra struttura dati!***

GLI OUTPUT PRODOTTI DALLE DUE VERSIONI SONO SEMPRE UGUALI COME ORDINE?

Come ottenere N pipe, con N non noto staticamente:

1) DEFINIZIONE ARRAY DINAMICO piped

```
typedef int pipe_t[2];      /* definizione del TIPO pipe_t come array di 2 interi */
…
pipe_t *piped;  /* array dinamico di pipe descriptors per comunicazioni figli-padre */
```

2) ALLOCAZIONE ARRAY DINAMICO piped DI DIMENSIONE N – ATTENZIONE: DA CONTROLLARE SEMPRE!

```
/* Allocazione dell'array di N pipe descriptors */
piped = (pipe_t *) malloc (N*sizeof(pipe_t));
if (piped == NULL)
{     printf("Errore nella allocazione della memoria\n");
      exit(3);    }
```

3) CREAZIONE DELLE N PIPE PER COMUNICAZIONE FIGLI-PADRE

```
/* Creazione delle N pipe figli-padre */
for (j=0; j < N; j++)
{     if(pipe(piped[j]) < 0)
        {   printf("Errore nella creazione della pipe\n");
            exit();     }
}
```

4)

a) CHIUSURE LATI PIPE NON USATI DA FIGLIO Pi➔ ogni figlio eredita per copia dal padre l'array di pipe (piped)

```
/* Chiusura delle pipe non usate nella comunicazione con il padre */
for (k=0; k < N; k++)
{     close(piped[k][0]);
      if (k != i)
            close(piped[k][1]);
}
```

b) CHIUSURE LATI PIPE NON USATI DAL PADRE dell'array di pipe (piped)

```
/* padre chiude tutte le pipe che non usa */
for (k=0; k < N; k++)
{
     close(piped[k][1]);
}
```