Verifica di apprendimento n.1

Traccia A

Classe 4F informatica

a.s. 2024/25

Prof. D'Alba Luca

Argomento oggetto di verifica: programmazione parallela mediante la creazione e l'utilizzo di processi paralleli utilizzando il linguaggio C++

Traccia dell'esercizio da svolgere:

Creare un programma in C++ (utilizzando il compilatore online) che rispetti le seguenti richieste:

1) Si chiedano in input due numeri interi all'utente (i numeri saranno salvati nelle variabili **a** e **b**)

2) Si creino le variabili intere ***ris_lista_1***, ***ris_lista_2*** e ***ris_lista_3***

3) Si considerino le seguenti liste di istruzioni:

LISTA 1
1. Fare la somma tra **a** e **b**
2. Aggiungere 4 al numero ottenuto
3. Dividere per 3 il nuovo numero ottenuto

LISTA 2
1. Fare la differenza tra **a** e **b**
2. Moltiplicare il numero ottenuto per 2
3. Sottrarre 3 al nuovo numero ottenuto

LISTA 3
1. Fare il prodotto tra **a** e **b**
2. Sottrarre 1 al numero ottenuto
3. Dividere per 2 il nuovo numero ottenuto

Nelle variabili *ris_lista_1*, *ris_lista_2* e *ris_lista_3* devono essere memorizzati i risultati delle operazioni eseguite dalle rispettive liste e **queste tre liste di istruzioni devono essere eseguite in PARALLELO**, non ha importanza quale sia la lista eseguita nel processo padre e quali nei figli

4) CIASCUN PROCESSO FIGLIO dovrà, prima di eseguire le operazioni della sua lista, stampare la scritta "*Sono un figlio con PID* <pid> *e mio padre ha PID* <pidpadre>", dove al posto di <pid> ci dovrà essere il suo PID e al posto di <pidpadre> ci dovrà essere il PID del suo processo padre

5) SOLO IL PROCESSO PADRE dovrà, prima di eseguire le operazioni della sua lista, stampare la scritta "*Sono il padre e ho PID* <pid>", dove al posto di <pid> ci dovrà essere il suo PID

6) Poi bisogna creare una variabile **c** in cui verrà memorizzato il risultato del prodotto tra i tre risultati delle liste

7) Alla fine, deve essere stampato il valore della variabile **c**

Ad esempio, se a=5 e b=3, la variabile **c** dovrebbe essere uguale a 28 alla fine dell'esecuzione. **Ovviamente è necessario che prima di eseguire il prodotto dei risultati delle liste, si ASPETTI che TUTTE le liste siano state eseguite** (con particolare attenzione alle liste eseguite dai processi figli che richiederanno un apposito comando, nel processo padre invece non è necessario un comando aggiuntivo per aspettare la fine della sua lista perché è lo stesso padre a invocare i comandi di attesa dei figli subito dopo aver eseguito la sua lista).

Suggerimenti: si suggerisce di creare due processi figli e ogni processo figlio, dopo aver eseguito la stampa richiesta e la sua lista di operazioni, può terminare subito, restituendo come valore di ritorno il risultato della lista.
SI RICORDI CHE NEL PADRE PER COMPRENDERE IL VALORE DELL'EXIT STATUS DI UN FIGLIO È NECESSARIO USARE UNA FUNZIONE PARTICOLARE.
Si legga passo passo il comando e si cerchi, per quanto possibile, di svolgere l'esercizio "un comando alla volta".

ISTRUZIONI PER CONSEGNARE: PER SICUREZZA SALVARE IL CODICE PREMENDO SUL TASTO DOWNLOAD. Negli ultimi 5 minuti, il docente attiverà il link per la consegna dal quale sarà possibile accedere a Classroom. Nel compilatore online, cliccare il tasto arancione “Share” nella barra superiore del compilatore online, copiare l’intero link che si trova nella voce “Share code” e consegnare tale link sul compito Classroom caricato dal docente (premendo su Aggiungi o crea > Link > SCRIVERE IL CODICE A MANO).

RICHIESTE AGGIUNTIVE PER POTER OTTENERE IL VOTO MASSIMO
Al fine di poter ottenere il voto massimo, **DOPO AVER FINITO E SALVATO L’ESERCIZIO SVOLTO**, ritornare sul codice appena prodotto nel compilatore online e fare le opportune modifiche al fine di aggiungere ed eseguire in parallelo una quarta lista di istruzioni qui di seguito descritta (oltre alla consueta stampa di pid e pid del padre e a considerare nel prodotto finale nella variabile **c** anche il risultato di questa lista):

LISTA 4
1. Fare la divisione intera tra **a** e **b**
2. Aggiungere 2 al numero ottenuto
3. Moltiplicare il nuovo numero ottenuto per 2

Ad esempio, se a=5 e b=3, la variabile **c** dovrebbe essere uguale a 168 alla fine dell'esecuzione.
Dopo aver apportato tali modifiche al codice nel compilatore online, generare un nuovo link per questo codice e consegnare **ANCHE** questo link (sia il compito originale sia quello con le richieste aggiuntive per il voto massimo) nello stesso compito Classroom caricato dal docente.
Questo codice contribuirà al raggiungimento di massimo **UN** voto aggiuntivo.

Funzioni principali per la generazione e la gestione dei processi UNIX con il linguaggio C++

La definizione pratica di processo e il PID

Un processo è un insieme di istruzioni **ATTUALMENTE IN ESECUZIONE**.
Quindi se noi creiamo un programma (insieme di istruzioni) e lo eseguiamo (lo "apriamo") 5 volte contemporaneamente (se tale programma non blocca tale possibilità) avremo 5 processi che eseguono le stesse istruzioni di quel programma.
Ogni processo ha un codice numerico univoco che lo identifica all'interno del computer, tale codice viene detto *Process IDentifier* e viene abbreviato con la sigla *PID*.
Ogni processo deve avere un PID diverso da quello di qualsiasi altro processo in esecuzione.

Le librerie necessarie

Per l'esecuzione dei programmi in C++ che creano e gestiscono processi in UNIX servono generalmente, per quel che abbiamo trattato, le librerie **`unistd.h`** (per le funzioni fork, getpid, getppid ecc.) e **`sys/wait.h`** (per le funzioni wait, waitpid ecc.)

La funzione `fork()`

È la funzione chiave per la generazione dei processi, permette a un processo (che verrà chiamato PADRE) di creare un nuovo processo identico a sé stesso che verrà chiamato FIGLIO, il figlio sarà eseguito in parallelo al padre (contemporaneamente, se ci sono core della CPU liberi), eseguirà idealmente le STESSE istruzioni del padre **A PARTIRE DALLA FORK**.
Al momento della creazione del figlio (ovvero quando viene eseguita la fork), non solo viene creato un processo figlio che esegue le stesse istruzioni del padre, ma viene fatta anche una **COPIA** DELLE VARIABILI del padre nel figlio. Questo vuol dire che il figlio, al momento della sua creazione, ha le stesse variabili contenenti gli stessi valori del padre, ma DAL MOMENTO DELLA CREAZIONE IN POI PADRE E FIGLIO EVOLVONO **INDIPENDENTEMENTE E SE VIENE CAMBIATO IL VALORE DI UNA VARIABILE SOLO NEL PADRE, QUESTO NON SI RIPERCUOTE NEL FIGLIO (E VICEVERSA)** PERCHÉ ALLA CREAZIONE DEL FIGLIO VIENE FATTA UNA **COPIA** DELLE VARIBILI (che poi possono evolvere indipendentemente tra padre e figlio).
La funzione fork, oltre a creare un processo figlio, restituisce un valore di ritorno che è **DIVERSO NEL PROCESSO PADRE E NEL PROCESSO FIGLIO**, il valore di ritorno è un numero intero che di solito usiamo salvare in una variabile chiamata *pidfork*. Nel caso si abbia la necessità di creare più figli, ma è buona pratica farlo sempre, conviene chiamare *pidfork1* o *pidfork_primofiglio* (o *pidfork2*, a seconda che sia il primo, il secondo o terzo figlio creato ecc.) la variabile in cui memorizziamo il valore di ritorno della funzione fork(), in modo da non generare confusione.
Il valore di ritorno della funzione fork() (e quindi il contenuto della variabile in cui lo salviamo, ad esempio *pidfork1*) è PARI A **0 NEL FIGLIO**.
NEL PADRE È invece PARI AL **PID DEL FIGLIO APPENE CREATO (quindi diverso da 0 nel padre)**.
Essendo diversi in padre e figlio i valori di ritorno della funzione fork(), **sarà diverso tra padre e figlio anche il contenuto della variabile in cui abbiamo salvato tale valore** (ad esempio *pidfork1*), **in particolare questa sarà l'UNICA variabile che, al momento della creazione del figlio, NON sarà una copia IDENTICA tra padre e figlio ma conterrà DA SUBITO DUE VALORI DIVERSI TRA PADRE E FIGLIO**.

Proprio in virtù di ciò, **possiamo usare un controllo su tale variabile** (che ha valori diversi tra padre e figlio) **per far eseguire NELLA REALTÀ ISTRUZIONI DIVERSE AL PADRE E AL FIGLIO**.
La funzione fork() non richiede l'inserimento di alcun parametro.

Come eseguire in realtà istruzioni diverse tra processo padre e processo figlio

Dopo l'esecuzione dell'istruzione fork, se nel nostro programma abbiamo scritto delle istruzioni (dopo la fork) senza l'uso di particolari accorgimenti, **queste saranno viste sia dal padre sia dal figlio e di conseguenza saranno eseguite in parallelo in modo identico dal padre e dal figlio**; ma evidentemente se creiamo un processo figlio che verrà eseguito in parallelo **probabilmente lo facciamo per fargli eseguire istruzioni diverse contemporaneamente al padre**, potrebbe non avere senso far fare alla CPU lo stesso identico calcolo in parallelo su due core diversi infatti.
E quindi come si fa?
Abbiamo detto che al momento della creazione del figlio, la fork restituisce valori diversi a padre e figlio, salvati ad esempio nella variabile *pidfork1*, **possiamo quindi usare un if controllando proprio il valore di questa variabile in modo da far eseguire istruzioni diverse a padre e figlio**.

```
if ( pidfork1 == 0 ) {

}
else {

}
```

Un if così fatto permette di far eseguire istruzioni solo nel figlio (nell'if) o solo nel padre (nell'else), perché i processi conterranno tutte le righe di codice del programma in maniera identica, ma essendo diverso il valore della variabile *pidfork1* i due processi entreranno in branche diverse dell'if e difatti eseguiranno istruzioni diverse.
Si ricordi che se si vuole creare un secondo figlio, la **seconda** istruzione fork() deve essere posizionata nell'if o nell'else (preferibilmente nell'else in modo da far creare i figli sempre allo stesso padre) perché se la si posiziona al di fuori verrà vista sia dal padre sia dal primo figlio, quindi **entrambi** creeranno un proprio figlio e si avranno DUE NUOVI figli anziché uno solo (può anche essere l'effetto desiderato, ma bisogna essere consapevoli di cosa si sta facendo e generalmente non è l'effetto desiderato).

La terminazione dei processi

Un processo di un programma scritto in C++ generalmente termina quando incontra il *return* **NELLA FUNZIONE MAIN**, spesso però accade che il padre crei dei figli **per far eseguire loro MOMENTANEAMENTE dei calcoli in parallelo, esauriti i quali I FIGLI NON SERVONO PIÙ** ma il padre ha ancora altre istruzioni da eseguire dopo, è bene quindi terminare i processi figli una volta svolto il loro compito e per fare ciò si può utilizzare la funzione `exit(`*numero*`)` dove al posto di *numero* ci sarà un valore di ritorno intero che potrà essere visto dal padre una volta terminato il figlio.
Si consiglia di utilizzare la funzione **exit** per terminare **i FIGLI** (**OVVIAMENTE DEVE ESSERE POSIZIONATA NEL RELATIVO IF DEL FIGLIO**) e **NON il return** perché il return termina un processo SOLO SE SI TROVA NEL MAIN, se è usato in un sottoprogramma non termina il processo, mentre l'exit lo termina sempre.

Le funzioni `getpid()` e `getppid()`

La funzione `getpid()` restituisce il PID (come numero intero) del processo in cui viene eseguita, la funzione `getppid()` (**con DUE p**) restituisce il PID **del PADRE** del processo in cui viene eseguita.

Nessuna delle due funzioni richiede l'inserimento di alcun parametro.

La sincronizzazione tra processi e la comunicazione da figlio a padre

Se creiamo dei figli per far eseguire loro delle istruzioni in parallelo, ad esempio dei calcoli, **è abbastanza logico che il padre abbia bisogno del risultato dei calcoli del figlio e quindi probabilmente a un certo punto della sua esecuzione ha anche bisogno eventualmente di FERMARSI e ASPETTARE che il figlio TERMINI LA SUA ESECUZIONE e restituisca il risultato al padre**.

Per fare entrambe queste cose useremo le funzioni `wait(&variabile_intera)` e `waitpid(piddelfigliodaaspettare, &variabile_intera_figlioX, 0)`.

La funzione `wait(&variabile_intera)` fa **FERMARE** l'esecuzione del processo in cui si trova **FINCHÉ NON VIENE TERMINATO IL FIGLIO PIÙ VELOCE**, se il figlio più veloce è già terminato o il processo in cui è usata la wait non ha figli, l'esecuzione non viene fermata perché non c'è niente da aspettare.

Ovviamente **CONVIENE USARE LA WAIT (O LA WAITPID) SOLO NEL PADRE PERCHÉ DOVREBBE ESSERE IL PADRE AD ASPETTARE I FIGLI**.

Il **PARAMETRO** della funzione wait è **MOLTO IMPORTANTE** perché è il **PUNTATORE (che si indica con una &) a una VARIABILE INTERA (CHE DOVREMO CREARE PRIMA DI ESEGUIRE LA wait) IN CUI VERRÀ MEMORIZZATO L'EXIT STATUS DEL FIGLIO APPENA TERMINATO**.

VA NOTATO SUBITO CHE L'EXIT STATUS CHE VERRÀ MEMORIZZATO NELLA `variabile intera` CHE PASSIAMO ALLA WAIT COME PUNTATORE NON È IL VALORE DI RITORNO DEL FIGLIO CHE PASSIAMO TRAMITE LA FUNZIONE EXIT NEL FIGLIO: IL VALORE CHE VIENE MEMORIZZATO NELLA `variabile_intera` **È UN VALORE NON DIRETTAMENTE LEGGIBILE E BISOGNA USARE UNA FUNZIONA PARTICOLARE (WEXITSTAUS(variabile_intera)) PER POTER "convertire" IL CONTENUTO DELLA variabile_intera NEL VERO VALORE CHE HA RESTITUITO IL FIGLIO**.

Come si può intuire **CONVIENE USARE LA FUNZIONE wait QUANDO ABBIAMO UN SOLO FIGLIO perché, visto che permette di aspettare IL FIGLIO PIÙ VELOCE, SE ABBIAMO PIÙ FIGLI DA ASPETTARE NON POTREMO FARLO CON LA FUNZIONE WAIT** MA **DOVREMO USARE LA FUNZIONE `waitpid(piddelfigliodaaspettare, &variabile_intera_figlioX, 0)` CHE CI PERMETTE DI INDICARE, COME PRIMO PARAMETRO, IL PID DEL FIGLIO DA ASPETTARE** e come secondo parametro la variabile intera in cui vogliamo salvare l'exit status di tale figlio (il terzo parametro lo metteremo sempre pari a 0 per adesso).

Come si può vedere questa seconda funzione ci permette di **SCEGLIERE il figlio da aspettare ed è QUESTA (waitpid) che DOBBIAMO utilizzare quando dobbiamo aspettare PIÙ FIGLI, avendo cura di utilizzare tante istruzioni waitpid quanti sono i figli da aspettare (e indicando in ogni istruzione waitpid il corretto PID del figlio da aspettare)**.

Lettura dell'exit status dei figli e estrazione del valore di ritorno

Come abbiamo detto **il valore di ritorno dei figli** (che nel padre salviamo in una `variabile_intera_figlio_X`), **NON È DIRETTAMENTE LEGGIBILE, ma per "convertirlo" nel valore di ritorno che ha restituito il figlio USEREMO LA FUNZIONE `WEXITSTATUS(variabile_intera_figlio_X)`** che **RESTITUISCE COME VALORE DI RITORNO IL VERO VALORE DI RITORNO DEL FIGLIO X CHE È CONTENUTO NEL PARAMETRO CHE GLI PASSIAMO (variabile_intera_figlio_X, variabile che come già detto contiene un valore non immediatamente leggibile)**.

Essendo che la funzione WEXISTATUS RESTITUISCE UN VALORE, PER MEMORIZZARLO DOVREMO SALVARLO IN UNA VARIABILE QUINDI USEREMO UN COSTRUTTO DEL TIPO:

```
int verovaloredirritorno_figlioX = WEXITSTATUS(variabile_intera_figlio_X);
```

WEXITSTATUS SI SCRIVE TUTTO MAIUSCOLO!!