Progetto SOL – Corso A e B – a.a. 22/23

Studente: Giovanni Guerrazzi Corso A matricola 580384

Questo progetto è formato da 3 file.c e da vari include di supporto.

I file sono i seguenti:

farm. c: nonché il programma principale. (andrà a creare i processi MasterWorker e Collector)

threadPoolWorker.c: contiene l'implementazione del threadpool dei worker.

calculateResult: funzione che calcola il risultato come richiesto dalla specifica del progetto sui file binari

farm.c:

questo programma esegue al suo interno una *fork()* andando a creare il processo Collector (processo figlio) mentre il processo padre svolgerà il ruolo di MasterWorker.

All'inizio del programma si trovano 5 variabili globali che mi permettono di gestire la comunicazione tra i 2 processi mediante l'ausilio di un signalHandler che va a settare queste variabili a seconda del segnale che riceve.

Il MasterWorker è un processo multithread che ho scelto faccia da master (eseguirà le chiamate socket, bind, listen, accept)

Il main thread si occupa di creare la connessione socket AF_UNIX (connessione locale) a cui andrà successivamente a connettersi il processo Collector con la system call connect

Inoltre, il main thread effettua tutte le operazioni di parsing della linea di comando (grazie alla funzione *getopt()*) che sono necessarie a stabilire i parametri del programma (numero dei thread del thread pool, dimensione della coda dei task …). I file binari possono essere letti direttamente da terminale oppure ricercati all'interno di una directory passata sempre da terminale. Per la lettura di file binari da directory utilizzo la funzione ricorsiva ***exploreDirectory():***

Il MasterWorker è composto anche da un poll di un certo numero di thread (di default questo numero è 4). A questi threads, detti workers, sono passati i file sui quali svolgono un conto particolare mediante l'utilizzo della funzione ***calculateResult()***, ottenendo un risultato che inviano al processo Collector mediante scritture su socket

Se è inserito da terminale un ritardo (<-t delay>), ci deve essere del tempo di delay che intercorre tra l'invio di due richieste successive ai worker thread da parte del main thread. Questo ritardo l' ho simulato tramite l'invocazione della funzione *nanosleep()* una volta che il task è stato inserito nel threadpool

Il processo **Collector, è un processo single thread**, che una volta stabilita la connessione con Il MasterWorker attende di leggere i risultati che gli vengono inviati dai worker thread. I risultati devono essere stampati in maniera ordinata. Per questo ho fatto sì che il Collector inserisse i risultati letti con tanto di filepath all'interno di una ***lista unidirezionale*** tramite la funzione ***InsertOrder()*** in maniera ordinata. Per far ciò mi sono definito un tipo particolare, *struct*

collection_t, in modo da avere in un'unica struttura sia il risultato del calcolo sul file che il suo filepath. Al termine della connessione, la lista viene stampata tramite la funzione ***printList()*** e successivamente viene deallocata la memoria degli elementi che la componevano con la funzione ***freeList()***

Il Collector, per ogni worker thread che ha terminato il suo task, legge il risultato dal file descriptor del socket mediante l'utilizzo di 3 readn per le seguenti informazioni:

-il risultato

-la lunghezza del filepath

-Il filepath

threadPoolWorker:

Contiene l'implementazione dell'interfaccia threadPoolWorker.h

È caratterizzato da una struttura di tipo *struct threadpool_t* con un certo numero di argomenti quali il numero di threads, dimensione della coda di task, indice dell'elemento all'inizio della coda, indice dell' elemento alla fine della coda …)

In sostanza un task è composto dalla "coppia" funzione ***calulateResult()*** e dal filepath del file binario su cui il worker thread deve calcolare il risultato che successivamente invierà al Collector

La coda dei task ho deciso di implementarla come un *array circolare*:

- l'indice che punta alla testa dell'array *indica la posizione in cui il* ***worker thread*** *andrà a prelevare il prossimo task da eseguire*
- l'indice che punta alla coda dell'array *indica la posizione in cui il* ***main thread*** *andrà a inserire il nuovo task*

Le funzioni che sono implementate all'interno del threadpool sono:

-createThreadPool():

crea il *threadpoo*l andando a inizializzare i valori della struttura

-addTaskFileToThreadPool():

questa funzione permette al main thread di aggiungere un nuovo task alla coda dei task. La coda è gestita come nel classico caso di produttore-consumatore, andando a utilizzare un mutex per la mutua esclusione e una variabile condizione in modo tale da gestire i thread in maniera efficiente

I task vengono prelevati dalla coda ed eseguiti dalla funzione ***threadWorkerFun()*** (funzione che esegue ciascun worker thread)

-destroyThreadPool():

elimina il threadpool facendo utilizzo di una funzione ausiliaria per la deallocazione della memoria: ***freePoolResources()***.

Il modo in cui il threadpool può essere eliminato varia a seconda del parametro che gli viene passato come secondo argomento (int mode):

se quest'ultimo è 0 il threadpool viene liberato aspettando prima che i task in coda vengano eseguiti (non vengono più accettati altri task)

se invece vale 1 il threadpool viene liberato subito non eseguendo i task in coda (non vengono più accettati altri task)

calculateResult:

Questa è la funzione che permette ai worker thread di effettuare il calcolo richiesto sui file binari.

Una volta effettuato il calcolo viene inviata la risposta al processo Collector mediante tre scritture sul socket:

-il risultato

-la lunghezza del filepath

-Il filepath

Terminazione del programma:

Il programma può terminare in *due* modi differenti:

-perché ha ricevuto un segnale di terminazione (SIGINT,SIGQUIT…) che viene appositamente gestito dal signalHandler con eventuale stampa parziale dei risultati calcolati sui file letti fino a quel momento.

-in assenza di segnali, una volta finito di leggere tutti i file binari in input, la variabile globale END viene messa a 1 e questo comporta la terminazione standard del programma (normale flusso di esecuzione)

Gestione segnali:

Per quanto riguarda i segnali ho scelto di gestirli in maniera asincrona mediante l'utilizzo della funzione *sigaction*. Mi sono andato a definire un signalHandler che ho chiamato banalmente *sigHandler,* assicurandomi che accedesse solamente a variabili globali che ho definito come volatile sigatimic_t.

Ho scelto di far sempre terminare il programma farm alla ricezione di un segnale fra quelli sottoelencati con però delle differenze a livello di gestione della coda dei task del thread pool:

- Se il sigHandler cattura uno fra i segnali SIGINT/SIGQUIT/SIGTERM/SIGHUP allora la variabile globale QUIT viene messa a 1 e ciò comporta la terminazione del programma con il completamento dei task presenti in coda nel threadpool con successiva stampa parziale (file binari analizzati fino a quel momento)

- Se il sigHandler cattura il segnale SIGUSR1 allora la variabile globale STOP viene messa a 1 e ciò comporta la terminazione del programma **senza** il completamento dei task presenti in coda nel threadpool con successiva stampa parziale (file binari analizzati fino a quel momento)

- Se il *sigHanlder* cattura il segnale SIGPIPE semplicemente viene ignorato.

Makefile:

Una volta decompressa la cartella contenente il progetto per compilare il tutto eseguire il comando:

$make all

per generare sia l'eseguibile *farm* che l'eseguibile *generafile* per la generazione dei file binari oltre che a tutti i file oggetti necessari per la compilazione

$make cleanall

per eliminare sia gli eseguibili che i file oggetto (ed altri file)

$make test

per eseguire lo script test.sh. (anche solo eseguendo questo comando genererà gli eseguibili necessari all'esecuzione del progetto)