UNIVERSITÀ DI PISA

RELAZIONE PROGETTO FARM

Studente: Stefano Passanante
Matricola: 597415

INDICE

INTRODUZIONE

Descrizione del progetto

Farm è un programma composto da due processi, il primo denominato MasterWorker ed il secondo denominato Collector. MasterWorker, è un processo multi-threaded composto da un thread Master e da 'n' thread Worker (il numero di thread Worker può essere variato utilizzando l'argomento opzionale -n). Il programma prende come argomenti una lista (eventualmente vuota se viene passata l'opzione '-d') di file binari contenenti numeri interi lunghi ed un certo numero di argomenti opzionali (le opzioni sono '-n', '-q', '- t', '-d'). Il processo Collector viene generato dal processo MasterWorker. I due processi comunicano attraverso una connessione socket AF_LOCAL (AF_UNIX). Il processo MasterWorker è il creatore dell'unica connessione esistente per la comunicazione tra i thread worker e il processo collector. Il socket file "farm.sck", associato alla connessione AF_LOCAL, deve essere creato all'interno della directory del progetto e deve essere cancellato alla terminazione del programma.

Specifiche richieste per il processo MasterWorker

Il processo MasterWorker deve per prima cosa gestire i segnali SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGUSR1. Alla ricezione del segnale SIGUSR1 il processo MasterWorker notifica il processo Collector di stampare i risultati ricevuti sino a quel momento, mentre alla ricezione degli altri segnali, il processo deve completare i task eventualmente presenti nella coda dei task da elaborare, non leggendo più eventuali altri file in input. Si deve poi fare il parsing degli argomenti passati nella linea di comando; controllando gli eventuali argomenti opzionali:

- **'-n'** per specificare il numero di thread Worker del processo MasterWorker (default=4);
- **'-q'** per specificare la lunghezza della coda concorrente tra il thread Master e i thread Worker(default=8);
- **'-d'** specifica una directory in cui sono contenuti file binari ed eventualmente altre directory contenente file binari; i file binari dovranno essere utilizzati come file di input per il calcolo;
- **'-t'** specifica un tempo in millisecondi che intercorre tra l'invio di due richieste successive ai thread Worker da parte del thread Master (default=0);

Fatto il parsing degli argomenti il thread Master crea un pool di thread Worker, tanti Worker quanto specificato, il pool comunica con il thread Master tramite la coda concorrente dei task da elaborare.

Specifiche richieste per il processo Collector

Il processo Collector è generato dal processo MasterWorker. Una volta stabilita la connessione con il processo MasterWorker tramite la socket "farm.sck", rimane in attesa dei vari thread Worker che invieranno il nome del file e il suo risultato. Il Processo Collector una volta terminati i file che i thread Worker inviano o all'arrivo del segnale SIGUSR1 deve stampare in maniera ordinata i valori ottenuti, nel formato seguente:

Risultato1 filepath1
Risultato2 filepath2
Risultato3 filepath3

Implementazione Farm

L'implementazione del progetto Farm è contenuta nella cartella **"src"** del progetto. Essa a sua volta contiene il file main.c e le cartelle:

- **Proesso_Master**: Questa cartella contiene l'implementazione di tutto il lavoro che deve eseguire il processo Master. Contiene due file: master_worker.c che implementa il codice del Master thread e worker.c che implementa il codice del task eseguito dai thread Worker.
- **Processo_Collector**: Questa cartella contiene il codice d'implementazione del processo collector, nel file collector.c;
- **coda_lista**: Questa cartella contiene i file utilizzati per implementare le strutture di appoggio per il processo collector (lista.c) e per il processo Master (queue.c);

Descrizione file main.c:

La funzione main è implementata nel file main.c all'interno della cartella **"src"**. Il thread main funge anche da thread Master. Come prima cosa, crea il thread sighandler_thread che andrà a gestire i segnali richiesti nella specifica del progetto e andrà anche ad ingorare il segnale SIGPIPE per evitare che il processo venga interrotto per una tentata scrittura su un descrittore che non ha la read pronta a leggerlo. In seguito eseguirà una fork e il processo figlio eseguirà la funzione collector_process() implementata nel file collettor.c contenuto nella cartella **ProcessoCollector.** Tale funzione eseguirà il lavoro richiesto al Collector, quindi lettura dei file inviati dai thread Worker e alla fine stampa dei risultata in maniera ordinata. Il processo padre, invece creerà una socket sulla quale rimarrà in attesa di una richiesta di connessione da parte del collector. Una volta stabilita la connessione, il thread main, chiama la funzione master_worker(int argc, char* argv[],int fd_c) implementata nel file master_worker.c questo file contiene la vera e proprio implementazione del lavoro che deve compiere il processo MasterWorker. Quindi una volta effettuti: parsing degli argomenti, creazione del pool dei worker, invio dei file da leggere ai vari worker, ecc. la funzione master_worker termina dopo aver pulito tutta la memoria allocata per fare i vari lavori, e si torna nella funzione main dove si rimane in attesa che il processo figlio termini il proprio compito, si esegue un pthread_cancel e in seguito una join sul thread che gestisce i segnali, perchè altrimenti rimarrebbe sempre in attesa passiva sulla sigwait, si cancella la socket creata e il processo termina.

Descrizione file master_worker.c

Una volta chiamata la funzione master_worker da parte del main, si esegue il parsing degli argomenti passati da linea di comando tramite la funzione di libreria getopt. La variabile optind fornita da getopt, sarà l'indice del prossimo elemento da processare in argv che è stato modificato da getopt in modo tale che gli argomenti non riconosciuti saranno alla fine, quindi una volta usciti dal while optind indicherà l'eventuale lista di file inserita da linea di comando. Terminata la getopt, si andranno ad inizializzare le varie strutture (coda dei task, pool di worker e struttura che contiene gli argomenti da passare ai worker) in base ai valori opzionali o quelli di default. Una volta inizializzata le strutture, all'interno di un ciclo for, tramite una pthread_create si andranno a creare i vari thread worker, che eseguiranno la funzione threadfunc(void* arg) che avrà come argomento una struttura che conterrà la coda dei task su cui i worker dovranno estrarre i file e il descrittore della connessione, per poi

inviare il risultato e il nome del file al collector. La comunicazione tra il Master e i worker avviene grazie alla coda dei task da elaborare (struttura che verrà descritta in seguito) tramite il protocollo produttore consumatore visto a lezione di laboratorio. In questo caso il Master funge da produttore perché inserisce i file (quelli passati da linea di comando o quelli trovati ricercando in maniera ricorsiva nella directory fornita dall'opzione -d) con una push sulla coda, e i thread worker andranno a leggere i file tramite una pop sulla coda e quindi loro saranno i consumatori. Quando saranno finiti i file da immettere nella coda o quando il thread per la gestione dei segnali, ne riceve uno che fa terminare l'inserimento nella coda dei task, si farà una push di stringhe che indicano EOF in modo tale da poter comunicare ai worker che non ci sono più file da leggere. Si faranno tante push di EOF quanti sono i worker creati, così che ogni consumatore leggerà tale stringa e quel thread terminerà il suo lavoro. Fatto ciò si esegue una join su ogni singolo thread del pool creato, si chiude la connessione, e liberiamo la memoria allocata.

Descrizione file worker.c

In questo file è contenuta l'implementazione del task che viene eseguito dai thread worker creati. Come già detto in precedenza i worker fungono da consumatori. Dalla coda fornita dalla struttura Thread_Args, vengono estratte le stringhe dei file da aprire. (quindi ogni thread sostanzialmente farà una pop sulla coda fino a quando non estrae la stringa di EOF, a quel punto il thread esce dal ciclo e termina) se non è un EOF, si apre il file. Da qui in poi si entra in una fase di muta esclusione (descrizione della concorrenza in seguito), si calcola il risultato da inviare al collector, grazie ad una funzione risultato(char* file, FILE *ifp) implementata per rispettare la formula descritta nella specifica. Restituitoci il risultato, si chiude il file e si inizia a comunicare con il processo collector. Come prima cosa si invia la lunghezza del nome del file che abbiamo aperto (per ogni write che viene fatta dai thread worker, si esegue una read per capire se la write è andata a buon fine), in seguito si invia il nome del file e alla fine il risultato calcolato e poi si esce dalla mutua esclusione e si libera l'eventuale memoria allocata.

Descrizione del file collector.c

Il file collector.c contiene l'implementazione della funzione collector_process() che esegue il lavoro che deve fare il processo collector, inoltre utilizza anche una lista ordinata, che fornisce al collector le funzioni insert, printList e libera rispettivamente per: inserire la stringa del file e il risultato in una lista ordinata, stampare gli elementi all'interno della lista, liberare la memoria (tale struttura utilizzata sarà implementata nel file lista.c che sarà descritto in seguito). Una volta stabilita la connessione con il processo Master, si effettuano una serie di read per leggere: la lunghezza della stringa che si deve ricevere (**read 1**), la stringa (**read 2**), e il risultato (**read 3**). Per ogni read si effettua una write per comunicare al processo Master che la lettura è avvenuta con successo. Avendo sia il path del file che il suo risultato, tramite la insert si inseriscono nella struttura lista ordinata. Questo si fa fino a quando la **read 1** ci restituisce un valora maggiore di 0, in caso contrario vuol dire che il processo Master ha chiuso la connessione e che non c'è più nulla da leggere, si esegue la funzione printList(), si libera la memoria e si cancella la socket. Inoltre se la **read 1** legge il valore **-1** esso rappresenta una lettura speciale, per cui si devono stampare gli elementi letti fino a quel momento. Leggerà **-1** quando il thread che gestisce i segnali, riceve SIGUSR1.

IMPLEMENTAZIONE

Descrizione file queue.c

In questo file è contenuta l'implementazione della struttura utilizzata dal processo Master, per gestire la comunicazione tra il thread Master i thread Worker. Questo file è stato utilizzato in una lezione di laboratorio, per risolvere un esercizio su un protocollo produttore, consumatore. Ho apportato alcune modifiche a quel file in quanto, per quella esercitazione la coda non prevedeva una dimensione massima. Quindi è stato aggiunto l'attributo maxlen che rappresenta la lunghezza massima della coda e la condition variable coda_piena per effettuare una wait quando il Producer cerca di inserire qualcosa nella coda, ma è stata raggiunta la dimensione massima. Quando un consumer effettuerà un pop si farà la signal su coda_piena, in modo tale da poter permettere al Producer di tornare ad inserire elementi nella coda se era in uno stato di attesa.

Descrizione file lista.c

Questo file contiene l'implementazione della struttura utilizzata dal collector per mantenere i file e i risultati che arrivavano dai thread worker. Si avvale di una struttura di tipo ListNode che rappresenta un nodo della lista, il nodo contiene: il risultato, la stringa del nome del file e un puntatore al prossimo nodo. Tramite la funzione insert(…) si vanno ad inserire i nodi nella lista in maniera ordinata rispetto al valore del risultato. La funzione printList(…) scorre la lista e ne stampa il risultato e la stringa che rappresenta il path del file. La funzione libera(…) scorre anche essa la lista e libera la memoria allocata.

Gestione della concorrenza

Le sezioni critiche del progetto sono tre: **1)** la gestione della coda concorrente dei task da elaborare, perché ci sono più worker che accedono in contemporanea a quella struttura, ma come già detto, questo caso è stato gestito tramite il protocollo produttore-consumatore. **2)** quando i worker devono utilizzare la connessione per inviare i dati al collector, perché appunto più worker potrebbero fare delle write in contemporanea e compromettere la comunicazione con il collector. **3)** quando si cerca di leggere o modificare la variabile globale termina. Viene utilizzata dal thread sighandler_thread per comunicare che è arrivato un segnale che per la gestione prevede l'interruzione delle push dei file sulla coda di comunicazione con i worker. Questa variabile globale se venisse acceduta in contemporanea: sia dal thread che gestisce i segnali che dal thread Master, la gestione dei segnali sarebbe compromessa. Per la gestione dei casi **2) e 3)** sono state utilizzate due mutex, rispettivamente: **lock_conn e lock_term.** Quando i worker dovranno inviare i nomi dei file e il risultato al collector prenderanno la lock sulla mutex **lock_conn** prima di avviare la comunicazione e la lasceranno al termine di essa. Quando invece il thread che gestisce i segnali o il thread Master cercherà di leggere o modificare la variabile termina dovrà prima acquisire la lock su **lock_term.**

Gestione dei segnali

La gestione dei segnali richiesti dalla specifica viene demandata al thread sighandler_thread quindi se arriva uno dei segnali specificati, verrà indirizzato a quel thread. La gestione di: SIGINT, SIGQUIT, SIGHUP, SIGTERM è la medesima, si accede con mutua esclusione alla variabile globale termina e si setta a **-1**, in modo tale che il thread Master si accorga che non deve più inserire dati nella coda, i worker terminano il loro lavoro prelevando gli ultimi task rimasti nella coda e inviando il risultato al collector, che stamperà tutto. La ricezione del segnale SIGUSR1 invece, deve solamente far stampare al collector i dati ricevuti fino a quel momento, senza interrompere nulla. Per fare ciò quando si verifica la ricezione del segnale SIGUSR1 si prende la lock sulla mutex **lock_conn** e si invia al collector un numero speciale **-1** che permetta al collector di stampare i dati ricevuti fino a quel momento e mettersi nuovamente in ascolto.