Relazione progetto FARM

Sistemi Operativi e Laboratorio AA 2022-2023

Maria Celeste Cesario - Mat. 597765

Indice

1- Introduzione

Il progetto consiste in un programma, **farm**, composto da due processi, denominati master_worker e collector, in comunicazione tra loro tramite una socket di tipo AF_UNIX.

Master_worker è un processo multi-threaded composto da un thread Master e da un numero variabile di thread Worker.
Una volta effettuato il parsing degli argomenti ricevuti da linea di comando e il setup della socket di comunicazione e del signal handler, esso effettua una fork( ) che genera il processo collector, crea il thread Master e avvia la Threadpool di Workers.
Il thread **Master** si occupa di passare ai worker i file passati da linea di comando e gli eventuali file nella directory passata tramite l'opzione -d.
I thread **Worker**, alla ricezione del file designato, effettuano la somma ponderata dei valori contenuti all'interno e la inviano attraverso la socket al processo Collector.
Il processo **Collector** si occupa quindi di ricevere e memorizzare le somme e i nomi dei file, per poi restituirle in output in ordine crescente di somma.

2- Struttura del progetto

All'interno della directory /headers sono contenuti gli header file per ognuno dei file .c ad eccezione del main. Per l'implementazione di specifiche parti si rimanda al codice, opportunamente commentato. Si allega una breve descrizione.

collector.* - Contiene la struttura dati del messaggio inviato dal worker thread, implementa la funzione eseguita dal processo Collector e altre funzioni di supporto.

list.* - Contiene le definizioni delle strutture dati double-ended queue e dei nodi, utilizzabili anche per comporre una linked list. Implementa funzioni che consentono di lavorare con queste due strutture.

master.* - Contiene la struttura dati degli argomenti del master e le variabili extern della double-ended queue tra master e worker, e del segnale di uscita. Implementa la funzione eseguita dal master thread e una funzione che permette il parsing della directory ricevuta.

worker.* - Implementa la funzione eseguita dai worker thread e una funzione che permette di scrivere un messaggio sulla socket connessa.

utils.* - Implementa una serie di metodi e funzioni di supporto e di controllo su operazioni. Contiene inoltre due funzioni, readn e writen, fornite nell'assignment 11 nella libreria conn.h.

threadpool.* - Implementa e gestisce una fixedThreadpool. questa libreria è stata presa dalla correzione dell'assignment 11. È stata solo modificata la funzione di gestione dei mutex per uniformarla al resto del progetto.

All'interno della directory principale sono contenuti tutti i file .c delle librerie implementate, il file main.c, il makefile, il file generafile.c e lo script di test test.sh

3- Schema del programma

All'avvio del programma, vengono mascherati tutti i segnali e avviato il thread **sighandler** (*v. gestione dei segnali*), successivamente vengono controllati gli argomenti della linea di comando tramite getopt, in modo da aggiornare i parametri e controllare la validità dell'eventuale directory. Si crea quindi la socket e si effettua l'operazione di bind, per poi effettuare la fork( ). Il processo figlio quindi esegue la funzione **Collector**, mentre il processo padre si occupa della creazione della threadpool di **workers** e di creare il thread **Master**, a cui vengono passati i rispettivi parametri. Il thread main resta quindi in attesa della terminazione del master, del collector, del signal handler e distrugge la threadpool, per poi effettuare le operazioni di pulizia.

Collector - Il collector resta in ascolto di connessioni tramite listen e accetta connessioni in arrivo e legge i messaggi sul socket tramite select (v. gestione della comunicazione). Alla ricezione di un messaggio, nel caso si tratta di un file ricava il relative path del file e inserisce ordinatamente il record nella lista. In caso di ricezione di segnali, il processo stampa la lista fino a quel momento, terminando l'esecuzione o meno a seconda del segnale ricevuto (*v. gestione dei segnali*).

Master - Il master inizializza la queue di comunicazione con il worker, dopodiché controlla se i file ricevuti da linea di comando sono file .dat e in caso affermativo aggiunge l'absolute path alla coda, notificando i worker. Se viene passata una directory viene anche effettuata la ricerca dei file all'interno. Alla fine del lavoro invia un segnale di fine ed esce. In caso di segnale di interruzione (*v. gestione dei segnali*), esce prematuramente senza inviare più file.

Worker - Il worker, alla ricezione del task, effettua la connessione al socket del processo collector e preleva il primo elemento nella queue condivisa con il master. In caso sia un

file, lo apre e effettua la somma ponderata dei valori contenuti all'interno (numero*indice) e invia file e somma al collector, mentre se vi è la ricezione di segnale si comporta adeguatamente (*v. gestione dei segnali*), inoltrandolo al collector e avviando il processo di chiusura.

4- Scelte di implementazione

All'interno del progetto si è scelto di utilizzare due strutture dati diverse per la comunicazione master-worker e il salvataggio dei risultati dal collector.

Per la comunicazione master-worker è stata implementata una **double-ended queue** con politica FIFO, in cui il thread master effettua la push dei filename in coda, mentre i worker threads prelevano dalla testa tramite pop.
Per il salvataggio dei dati del collector si è utilizzata una **linked list,** effettuando un inserimento ordinato alla ricezione di un nuovo record da inserire.

Si assume inoltre che la directory passata da linea di comando sia contenuta all'interno della directory principale e che i path assoluti non superano i 255 caratteri.

In fase di sviluppo è stato riscontrato un bug per cui il file di socket non veniva generato con il nome corretto ma troncato, perciò è stato introdotto un fix per accorciare il nome in caso di connect fallita con codice di errore ENOENT.

5- Gestione della concorrenza

L'unica struttura presente che poteva presentare problemi di accesso concorrente è la **queue** tra master e worker. Essa è stata opportunamente protetta applicando un mutex all'intera struttura, rendendola quindi thread-safe. È stata anche utilizzata una pthread_cond_t per identificare se la coda è piena o meno. Se viene effettuato un push su una coda piena, infatti, il thread resta in attesa con pthread_cond_wait. Il pthread_cond_signal viene inviato ogni volta che un worker thread effettua un pop dalla coda.
Si è inoltre deciso di dichiarare il segnale di uscita come variabile sig_atomic_t, in modo da effettuare l'aggiornamento in modo atomico ed evitare così letture e scritture concorrenti.

6- Gestione della comunicazione

La comunicazione con il processo Collector avviene tramite l'utilizzo di **socket** AF_LOCAL (AF_UNIX). La socket viene creata prima di effettuare la fork( ) e viene passato il suo file descriptor al collector per rimanere in ascolto di connessioni. La connessione avviene ogni volta che un thread Worker riceve un task, oppure quando il signal handler invia il segnale di stampa. Il collector accetta le connessione e legge i messaggi in arrivo tramite l'utilizzo di una select.
L'invio di messaggi viene effettuato seguendo un **protocollo** ben stabilito: ogni comunicazione consiste infatti in tre messaggi distinti: il primo messaggio contiene un integer, indicante la lunghezza del filename, il secondo uno stream di char contenente il filename effettivo e il terzo un long contenente la somma ponderata calcolata dal worker.

7- Gestione dei segnali

All'avvio del programma viene creata una **maschera** con i segnali interessati, che viene allegata come argomento al signal handler thread, inoltre viene ignorato SIGPIPE. Il thread creato poi resta in attesa di segnali tramite sigwait.
Alla ricezione di SIGUSR1 stabilisce una connessione con il collector e invia il segnale di stampa, mentre alla ricezione di SIGINT, SIGQUIT, SIGTERM o SIGHUP comunica ai worker l'interruzione del lavoro, aggiornando anche la variabile globale di uscita in modo da consentire la corretta terminazione di master e worker threads.

In caso di assenza di segnali, alla fine del proprio compito il thread master invia un messaggio di fine alla pool di worker e termina. Alla ricezione del segnale, gli altri thread terminano correttamente a loro volta consentendo la chiusura del programma.

8- Gestione degli errori

All'interno del progetto vengono gestiti la maggior parte degli errori in modo da terminare il programma in modo sicuro, liberando la memoria allocata tramite opportune funzioni di cleanup. In pochissimi casi vi è l'uscita diretta dal programma.
Le funzioni di cleanup e la gestione degli errori è visibile nel dettaglio all'interno del codice.

9- Compilazione ed esecuzione

Durante lo sviluppo è stato utilizzato un parametro e una funzione di debug, in modo da generare gli argomenti direttamente nel programma e consentire un test più rapido. Gli argomenti venivano modificati opportunamente per effettuare il test a seconda delle differenti richieste. Si rimanda al codice per il segmento di debug.
È stato poi effettuato testing tramite una combinazione di GDB e valgrind, eseguendo il seguente comando con l'opzione di debug attiva:
valgrind –leak-check=full –show-leak-kinds=all ./farm

Per eseguire e compilare il programma è necessario trovarsi nella directory principale.
A questo punto sono disponibili una serie di opzioni:

- make farm: per compilare il programma
- make test: per eseguire lo script di test.sh, fornito insieme alla traccia del progetto
- make clean: per rimuovere solo gli eseguibili
- make cleanall: per rimuovere tutti i file generati