RELAZIONE PROGETTO SISTEMI OPERATIVI E LABORATORIO: FARM

Diego Bernardini

Anno Accademico 2023-2024

SOMMARIO

INTRODUZIONE

Panoramica

farm è un programma composto da due processi, MasterWorker e Collector. Il primo è un processo multi-threaded composto da un thread Master e da un numero 'n' di thread Worker che può variare in base all'opportuno parametro di ingresso [vedi manuale d'uso]. Il processo Collector viene generato dal processo MasterWorker e i due comunicano attraverso una connessione socket AF_UNIX. Il programma prende come argomenti una lista, eventualmente vuota, di file binari contenenti numeri interi lunghi ed un certo numero di argomenti opzionali che andranno ad inizializzare l'ambiente di esecuzione.

Il processo MasterWorker legge gli argomenti passati alla funzione main uno alla volta, verificando che siano file regolari. Dopodiché Il nome del file viene inviato ad uno dei thread Worker del pool tramite una coda concorrente condivisa. A questo punto il generico thread Worker si occupa di leggere dal disco il contenuto del file di cui ha ricevuto il nome, effettuare un calcolo sugli elementi letti al suo interno e, infine, inviare il risultato ottenuto, unitamente al nome del file, al processo Collector che rimarrà in attesa di ricevere tutti i risultati dai Worker prima di stampare i valori ottenuti sullo standard output, ordinando la stampa in ordine crescente di risultato.

Il processo MasterWorker deve 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, sempre in modo ordinato. Invece, alla ricezione degli altri segnali, il processo deve completare i task eventualmente presenti nella coda condivisa, non leggendo più eventuali altri file in input, e quindi terminare dopo aver atteso la terminazione del processo Collector ed effettuato la cancellazione del socket file. Il processo Collector maschera tutti i segnali gestiti dal processo MasterWorker. Il segnale SIGPIPE deve essere gestito opportunamente dai due processi.

Struttura Delle Directories

- documents: contiene questa relazione e il testo del progetto.
- src: contiene i file .c.
- myInclude: contiene i file .h.

SCELTE DI PROGETTAZIONE/IMPLEMENTAZIONE

Makefile

File utilizzato per automatizzare la compilazione e l'esecuzione del progetto. Al suo interno sono implementati quattro metodi: clean, cleanall, farm e test. Il primo elimina i file oggetto derivati dalla compilazione del progetto e il file .sck utilizzato per la comunicazione tra i due processi. Il secondo sfrutta il comando precedente ed elimina altri prodotti generati dell'esecuzione del progetto. Il terzo compila l'intero progetto e l'ultimo viene utilizzato in fase di testing.

Main

La prima azione che viene fatta è inizializzare la maschera per i segnali. La seconda è invocare la *fork()* per andare a creare il processo Collector seguendo la specifica del progetto. A questo punto le azioni dei due processi sono separate all'interno dei rami di un unico if/else.

Masterworker

Le operazioni compiute da questo processo sono implementate nel file *master.c*. Osservando il main si nota l'ordine delle varie chiamate: per prima cosa viene creato il thread *sighandler_thread* che si occuperà della gestione dei segnali. Dopodiché viene creata la lista dei file binari che dovranno essere processati. Poi viene creato il threadpool e infine la socket per la comunicazione.
A questo punto, se non ci sono stati errori che altrimenti avrebbero causato una chiusura anticipata del programma, tutte le risorse necessarie all'esecuzione del programma sono state inizializzate correttamente e viene invocata la funzione *pushList()*, grazie alla quale i file binari che si trovano nella lista precedentemente creata vengono inseriti nella coda concorrente condivisa.
Se, nuovamente, non ci sono stati errori di alcun genere viene avviato il protocollo di terminazione, prima di attendere la terminazione dei vari thread del pool; del processo Collector e del sighandler_thread.

Collector

Le operazioni compiute da questo processo sono implementate nel file *collector.c*. Intuitivamente, il suo comportamento deve essere quello di rimanere in ascolto sulla socket in attesa di ricevere messaggi che, una volta arrivati, dovranno essere memorizzati e ordinati in una sua struttura dati interna per poi essere stampati al momento opportuno. Osservando il main si nota l'implementazione di questo comportamento: come prima cosa viene creata la socket per la comunicazione con la funzione *openServerSocket()*. Fatto ciò, la funzione successiva che viene invocata è *runCollector()*, la quale implementa esattamente il comportamento descritto poco fa: attendere messaggi, memorizzarli e stamparli sullo standard output al momento opportuno.

Threadpool

La struttura e l'interfaccia si trovano in *threadpool.h* mentre l'implementazione in *threadpool.c*. Il threadpool vero e proprio viene creato e inizializzato grazie alla chiamata *createThreadpool()*, che

prende in input il numero di thread worker che dovranno essere creati, la dimensione della coda condivisa e l'intervallo di tempo che dovrà passare tra l'invio di due richieste successive ai thread worker. All'interno di questa funzione vengono anche inizializzate la mutex e le variabili condizione necessarie per la corretta sincronizzazione.

All'interno dello stesso file è presente anche l'implementazione del comportamento del thread worker tramite la funzione *workerpool_thread()*, la quale, dipendentemente dallo stato del programma, estrae il nome di un file dalla coda condivisa, calcola il risultato e invia le informazioni al Collector. Se la coda è vuota, si mette in attesa oppure, se è il caso, segnala al Collector di stampare i risultati raccolti fino a quel momento.

Siccome la coda condivisa si trova all'interno della struttura del threadpool, nell'implementazione dell'interfaccia si trovano anche le funzioni *pop()* e *push()*. Quest'ultima viene chiamata all'interno della *pushList()*, utilizzata dal master per inserire i task dopo aver acquisito la mutex. La *pop()*, invece, viene chiamata all'interno di *workerpool_thread()* per estrarre un task dalla coda e non si occupa dell'acquisizione della mutex.

List

Per gestire l'elenco dei file binari che dovranno essere processati e l'elenco dei risultati raccolti dal Collector ho deciso di utilizzare come strutture dati due semplici liste che differiscono solo per la struttura del singolo nodo che le compone.
Il generico nodo della lista dei file binari (chiamato *nodo*) contiene al suo interno due informazioni: il nome del file in questione e il riferimento al nodo successivo della lista.
Il generico nodo della lista utilizzata dal Collector (chiamato *nodoF*), invece, contiene tre informazioni: il nome del file che è stato processato, il valore calcolato sullo specifico file e il riferimento al nodo successivo della lista.

Per quanto riguardo la gestione di queste strutture dati, la prima lista viene creata tramite l'inserimento dei nodi in testa alla lista, visto che i file binari non dovranno essere processati in un ordine preciso; mentre l'altra attraverso una funzione che inserisce subito il nuovo nodo nella posizione opportuna, ordinando la struttura in ordine crescente in base al valore calcolato.

CmdLineParser

File creato per avere una maggiore chiarezza nel codice. Contiene le funzioni con cui il programma elabora gli input che vengono passati al momento dell'esecuzione. La più importante è *cmdParse()*, chiamata dal MasterWorker, che basa il suo comportamento sulla funzione *getopt()* con la quale è possibile processare le varie opzioni e i relativi argomenti passati a riga di comando .

Segnali

Per quanto riguarda la gestione dei segnali ho deciso di non utilizzare handler e creare un thread, chiamato *sighandler_thread*, il cui unico comportamento è quello di invocare la funzione *sigwait()* e rispondere in modo opportuno al segnale ricevuto. Questa scelta è motivata dal fatto che non ho

alcun tipo di restrizione per quanto riguarda l'accesso alle variabili globali e l'utilizzo di funzioni che possono essere chiamate durante la risposta al segnale appena ricevuto. Tutto questo facendo attesa sincrona.

Gestione concorrenza

Il progetto sfrutta due lock: una interna al threadpool che permette l'accesso alla coda condivisa e una esterna ad esso che permette di scrivere sulla socket. In questo modo è stato possibile da una parte garantire mutua esclusione sulla coda dei task evitando di creare inconsistenze durante le varie operazioni di pop e di push e dall'altra permettere ad un solo thread worker alla volta di scrivere direttamente sulla socket, cosicché i vari messaggi vengano inviati e ricevuti in un modo e in un ordine ben preciso.

Client/Server

Ho deciso di far svolgere il ruolo del server al processo Collector e quello del client al processo MasterWorker. La scelta è stata fatta sulla base della seguente osservazione: da specifica il MasterWorker è colui che crea e invia le informazioni e il Collector colui che invece le attende. Durante un'esecuzione pulita succede che il MasterWorker smetterà di inviare messaggi al Collector sicuramente prima che quest'ultimo abbia terminato la propria esecuzione o la lettura delle informazioni sulla propria socket. Teoricamente la connessione del client viene chiusa prima di quella del server e per questo ho preso questa decisione.

Il fatto che nell'implementazione, però, è il MasterWorker che attende la terminazione del Collector è dato dalla specifica che stabilisce che il primo genera il secondo.

TEST EFFETTUATI

Il progetto è stato realizzato utilizzando i seguenti strumenti:

- OS (Host): Windows
- Sistema di virtualizzazione: VirtualBox 6.1.38
- OS (Guest): XUbuntu
- Editor di testo: Sublime text 3.2.2

Durante la fase di testing ho utilizzato i file test.sh e il file generafile.c per verificare la corretta esecuzione del programma sfruttando alcuni file generati sul momento.

Per testare la gestione dei segnali, come prima cosa, ho digitato sul terminale il comando

<<kill -l >> per visualizzare l'elenco dei segnali che possono essere inviati con il relativo numero identificativo.
Poi ho modificato, all'interno del makefile il comando test, assegnando al parametro "t" il valore 5000 per rallentare l'esecuzione del programma. Infine, ho utilizzato un secondo terminale dove ho eseguito il comando

<<ps -A | grep farm>> per conoscere il pid dei processi durante l'esecuzione del programma

e successivamente

<<kill -<signal> <pid> >> per inviare i segnali al processo opportuno.

Risultati ottenuti

```
xubuntu@xubuntu-VirtualBox:~/Scrivania/zero$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Figura 1. Elenco dei segnali

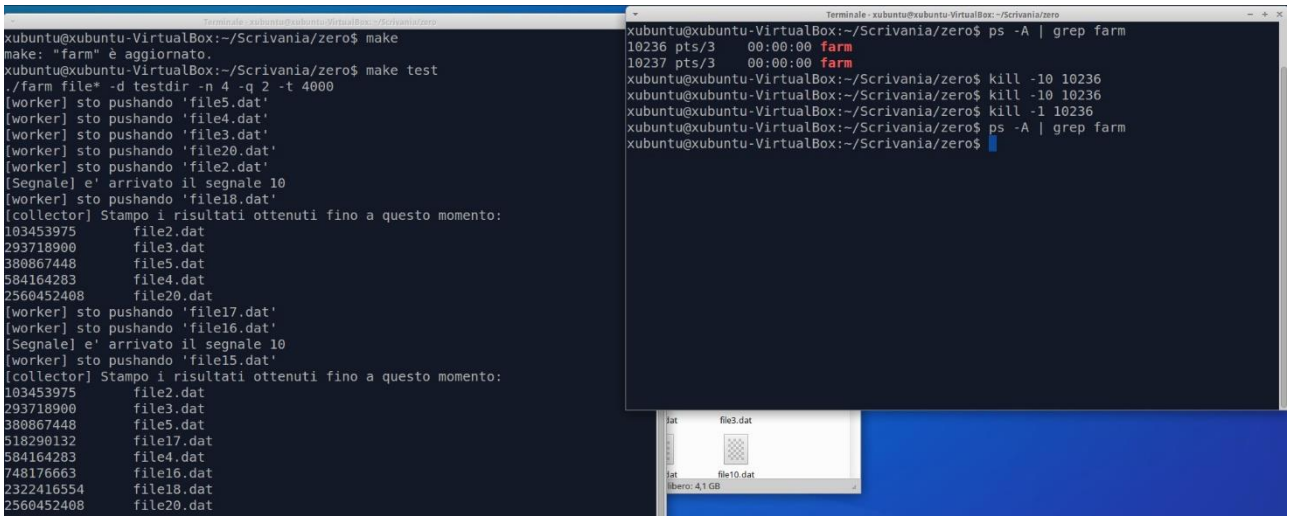

Figura 2. Test SIGUSR1 (10) inviato 2 volte

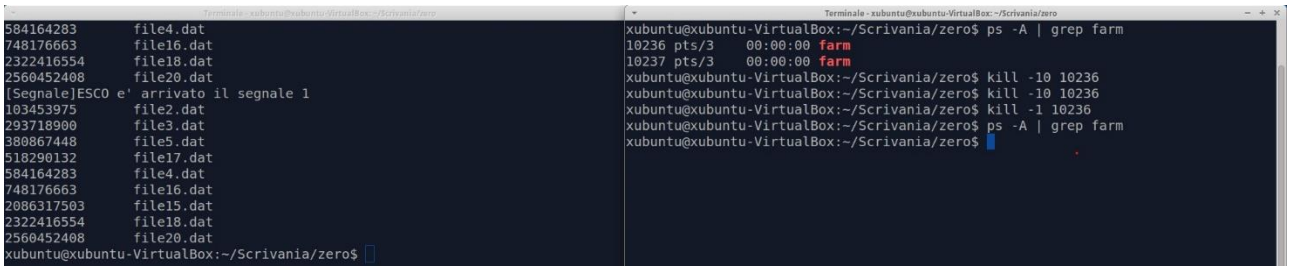

Figura 3. Test SIGHUP (1)

Queste figure testimoniano i risultati derivati dai vari test.

Si riassume che:

- SIGHUP (1), SIGINT (2), SIGQUIT (3), SIGTERM (15) terminano “prematuramente” il programma come da specifica.
- SIGPIPE (19) viene ignorato completamente.
- SIGUSR1 (10) stampa i risultati raccolti dal processo Collector come da specifica.
- SIGKILL (9) e SIGSTOP (19) funzionano correttamente.

CONCLUSIONI

Miglioramenti

Modificare la struttura dati utilizzata dal Collector cercando di ottenere performance migliori.

MANUALE D'USO

Per compilare ed eseguire il programma utilizzando il Makefile è sufficiente posizionarsi nella cartella che contiene i file .c e il Makefile, aprire un terminale e digitare:

<<make>> per compilare opportunamente e successivamente

<<make test>> per eseguire l'eseguibile fornendo in input le informazioni generate dal file test.sh.

Nel caso in cui si volesse eseguire il programma con altri parametri si ricorda che gli argomenti che opzionalmente possono essere passati al processo MasterWorker sono i seguenti:

- -n specifica il numero di thread Worker del processo MasterWorker (valore di default 4)
- -q specifica la lunghezza della coda concorrente condivisa tra il thread Master ed i thread Worker (valore di default 8)
- -d specifica una directory in cui sono contenuti file binari ed eventualmente altre directory contenenti file binari che 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 (valore di default 0)