UNIVERSITÀ DI PISA

Relazione SOL-FARM

Alessandra De Lucrezia 565700

Corso A - anno 2022-2023

1 Introduzione

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 processo Collector viene generato dal processo MasterWorker. I due processi comunicano attraverso una connessione socket AFLOCAL (AFUNIX). Il processo MasterWorker legge gli argomenti passati alla funzione main uno alla volta, verificando che siano file regolari. Se viene passata l'opzione '-d' che prevede come argomento un nome di directory, viene navigata la directory passata come argomento e considerando tutti i file e le directory al suo interno. Il nome del generico file di input (unitamente ad altre eventuali informazioni) viene inviato ad uno dei thread Worker del pool tramite una coda concorrente condivisa. Il generico thread Worker si occupa di leggere dal disco il contenuto dell'intero file il cui nome ha ricevuto in input, e di effettuare un calcolo sugli elementi letti e quindi di inviare il risultato ottenuto, unitamente al nome del file, al processo Collector tramite la connessione socket precedentemente stabilita. Il processo Collector attende di ricevere tutti i risultati dai Worker e al termine stampa i valori ottenuti sullo standard output, ordinando la stampa.

2 Struttura generale

Ogni file all'interno del progetto è opportunatamente commentato, per vedere una sezione in particolare si rimanda alla lettura del codice.

2.1 Directory "include"

Nella directory "include" sono contenuti i seguenti header file:

- *collector.h*: contiene la struttura dati del messaggio che riceverà dai thread del threadpool e la dichiarazione del metodo del Collector;
- *tree.h*: contiene la dichiarazione della struttura dati dell'albero binario, e la dichiarazione dei metodi usati per gestirlo;
- *master_worker.h*: contiene la struttura dati del master worker *params_master_worker_t*, la struttura dati da passare al momento della creazione del worker *worker_thread_arg_t*, il metodo del master worker *master_worker* e il metodo del worker *worker*;
- *threadpool.h*: oltre a contenere la dichiarazione dei metodi, implementati nella liberia *threadpoll.c*, contiene due strutture dati *taskfun_t* che rappresenta il *task* di un thread e la struttura *threadpool_t* che mantiene una serie di informazioni come il numero di thread, la lunghezza della coda ed altre variabili come pthreadMutex e varie pthreadCond;

  Questa libreria è stata presa dalla correzione degli assegnamenti, ma sono state apportate delle modifiche per il corretto funzionamento per il caso specifico;
- *util.h*: contiene l'implementazione di una serie di macro per la gestione degli errori oltre alle funzioni di writen e readn per la scrittura e lettura di un numero n di byte su un file descriptor. Tali funzioni sono state fornite come note ausiliarie all'assegnamento 9 sulla pagina del corso di Sistemi Operativi nella libreria *conn.h*;
- *worker_support.h*: contiene la dichiarazione dei metodi usati per processare l'eventuale directory inserita al momento della compilazione, un metodo che controlla se un file è binario, ed i metodi usati per mandare messaggi al Collector.(anche i segnali saranno inviati sulla connessione socket)

2.2 Directory principale

Nella directory principale del progetto sono contenuti i *file.c* delle implementazioni dei *file.h* descritti sopra (ad eccezione di *util.h*), il file contenente il main *farm_main.c*, il makefile, il *test.sh* e *generafile.c*, questi ultimi sono stati forniti insieme al testo del progetto.

3 Architettura generale

All'avvio del programma vengono mascherati tutti i segnali (si rimanda alla lettura dalla sezione Gestione Segnali per vederlo nello specifico) successivamente vengono controllati gli argomenti presi da CLI che vengono salvati all'interno della struttura dati del master worker, *params_master_worker_t*, poi si crea la connesione socket, con SOCKET_NAME *farm.sck* ed una connessione AF_UNIX(AF_LOCAL), successivamente viene fatta una *fork()* da cui avrò il processo padre (MasterWorker) ed il processo figlio (Collector). Analizziamoli sommariamente per analizzarne le funzionalità principali.

3.1 Processo padre

Nel processo padre viene creato il thread per la gestione dei segnali, dopo di che si crea il threadpool usando i valori presi da linea di comando o i valori di default. Quando il threadpool viene creato per ogni thread viene stabilita una connessione socket stabile. Si aspetta che tutti i thread siano connessi alla socket e si passa alla creazione del ThreadMaster che si occuperà di scorrere tutti gli argomenti presi dalla CLI per trovare i file, verificare che siano binari e mandarli ad un thread del threadpool che esegue le seguenti azioni:

- apre il file binario;
- esegue il calcolo leggendo un long per volta;
- memorizza il path del file in msg.pathfile e il conto ottenuto in msg.conto;
- scrive con la funzione *writen* sulla socket la struttura dati *msg*.

Dopo aver finito di scorrere tutti gli argomenti presi da CLI, il master processa la directory salvata nella sua struttura dati e verificando ogni file trovato nella directory e nelle sue eventuali sottodirectory esegue gli stessi passi di prima.

Quando tutti i file saranno stati mandati, e non è stato riecevuto nessun segnale, viene inviato un messaggio dal nome "done" e avrà come conto il numero dei file inviati. Se il Collector ha ricevuto tutti i file che doveva ricevere stampa tutti i messagi ricevuti e termina, altrimenti continua il suo flusso. A questo punto si esce dal metodo del thread master e si aspetta la terminazione del figlio, analizzandone lo stato di uscita, successivamente si libera la memoria (distruggendo il threadpool e terminando il thread_signal) e si termina.

3.2 Processo figlio

Nel processo figlio viene chiamato il metodo *collector* all'interno del quale viene effettua la *listen* per mettersi in ascolto sulla socket dove i thread si connetteranno, successivamente viene eseguita la *select* che, in base al file descriptor pronto, effettuerà o una *accept* per accettare la rischiesta di connessione dai thread oppure sarà pronto il canale per lettura e scrittura, nello specifico il collector non scrive mai sulla connessione, ma legge soltanto. Alla ricezione del messaggio il Collector si comporta in modo differente in base al valore di *msg.path*:

- *exit*: messaggio inviato dopo che il thread_signal ha ricevuto un segnale tra quelli gestisti (tranne che per SIGURS1) che obbliga il collector a stampare i file ricevuti fino a quel momento e a terminare;
- *print*: messaggio ricevuto dopo che il thread_signal ha ricevuto SIGURS1, il collector stampa i file ricevuti fino a quel momento e poi continua normalmente il suo flusso (può ricevere più SIGURS1);
- *done*: messaggio ricevuto quando il master worker ha finito di mandare tutti i file, avrà come valore di msg.conto il numero dei file che deve ricevere prima di porter terminare, se li ha ricevuti tutti allora stampa l'albero contentente tutti file ricevuti e termina;
- *"pathfile"*: messaggio generico mandato da un thread che ha eseguito il calcolo di un file binario. Il Collector salva il messaggio inserendolo, in maniera ordinata in modo crescente in base al valore di msg.conto, all'interno di un albero binario.

Nel caso in cui non ci siano stati errori nell'esecuzione dei metodi, prima di terminare si dealloca la memoria usata e si termina restituedo il valore 0. Mentre in caso di errori, si pulisce la memoria e si ritorna -1. Il valore di uscita viene analizzato e se uguale a -1 viene mandato un segnale al processo padre con il metodo *kill(getpid(), SIGINT)*.

4 Protocollo di comunicazione

Il processo Collector e i thread Worker comunicano tra loro attraverso una connessione Socket AF_UNIX. La socket viene creata prima di effettuare la fork e il suo file descriptor viene passato come argomento al Collector. Il Collector, che si comporta da Server, effetua la listen per rimanere in attesa della richiesta di connessione dei thread che rappresentano i client. Quando viene creato il threadpool per ogni thread viene stabilita una connessione stabile sulla socket, che rimane attiva per tutto il tempo dell'esecuzione del programma, il file descriptor della socket viene passato come argomento ad ogni task, che verrà poi utilizzato per mandare i messaggi tra i thread e il Collector. Il Collector accetta le varie connessioni da parte dei Worker e legge i messaggi inviati dai Worker utilizzando la Select.

5 Gestione concorrenza

Per gestire la concorrenza dei thread è stata usata la libreria fornita nella correzione degli assegnamenti dell'ottava lezione di laboratorio, *threadpoll.h*, a cui sono stata apportate delle modifiche come l'aggiunta all'interno della struttura dati del threadpoll di alcune *pthread_cond_t* per segnalare che la coda è piena, o che tutti i thread sono connessi sulla socket. In caso di coda piena, nel metodo *addToThreadPool*, il thread viene messo in attesa con una *pthread_cond_wait* per poi essere risvegliato con una *pthread_cond_signal* lanciata nel metodo *workerpool_thread*.

6 Gestione segnali

All'avvio del programma tutti i segnali vengono mascherati, viene ignorato SIGPIPE e successivamente, in seguito alla fork, nel processo padre viene creato un thread, il thread signal, che si occuperà di ricevere e gestire i seguenti segnali: SIGURS1, SIGQUIT, SIGINT e SIGTEM.

Alla ricezione di tutti i segnali, tranne per che per SIGURS1, il thread signal prepara un messaggio di "exit", che verrà poi assegnato ad un thread del threadpool per essere consegnato al Collector, sempre tramite la connessione socket come se fosse un messaggio normale. Il collector alla ricezione terminerà stampando i risultati ricevuti fino a quel momento, terminerà anche il MasterWorker con il conseguente rilascio di memoria utilizzata.

Alla ricezione di questi segnali, sempre ad eccezione di SIGURS1, il MasterWorker smetterà di inviare i file ai thread, si ritornerà nel processo principale dove era stata fatta la fork, si libera la memoria allocata e si termina. Mentre per SIGURS1 il comportamento è diverso, il thread prepara il messaggio di "print", che verrà inviato sulla connessione socket con lo stesso procedimento usato per gli altri segnali e il collector alla ricezione stamperà l'albero contenente i file ricevuti fino a quel momemento, non terminerà ma continuerà il suo flusso di esecuzione. Il segnale SIGURS1, a differenza degli altri segnali, può essere ripetuto più volte.

7 Gestione errori

Nel progetto vengono gestiti la maggior parte dei possibili errori che possono accadere durante l'esecuzione del programma. Viene testata ogni chiamata di funzione e nel caso di errore viene gestito in modo da riuscire a librerare la memoria prima di terminare, solo in rari casi si esce direttamente con il comando _exit(EXIT_FAILURE). Per vedere nello specifico come vengono gestiti gli errori, si rimanda alla lettura del codice.

8 Scelte progettuali

Per lo sviluppo del progetto sono state assunte le seguenti premesse:

- viene passata solo una directory da CLI, ed è situata all'interno della cartella del progetto;
- la lunghezza complessiva del path del file e della directory non superano i 255 caratteri;
- i segnali non gestiti hanno il comportamento di default.

9 Compilazione ed esecuzione

Durante lo sviluppo il progetto è stato costantemente eseguito usando il comando:

valgrind –leak-check=full –show-leak-kinds=all –track-origins=yes ./farm -n 2 -q 1 -d testdir file*

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

- make: per compilare;
- make cleanall: per rimuovere tutti i file generali (compresi i file binari e la direcory "testdir");
- make clean: per rimuovere solo gli eseguibili (se presenti);
- make test: per eseguire lo script di test.sh, fornito insieme alla traccia del progetto.

9.1 Esempi di esecuzione

9.1.1 Esecuzione make test

```
alessandra@Hyrule:~/Scrivania/FARM$ make test
chmod +x test.sh
./test.sh
test1 passed
test2 passed
test3 passed
test4 passed
test5 passed
alessandra@Hyrule:~/Scrivania/FARM$
```

9.1.2 Esecuzione con segnale SIGQUIT

```
alessandra@Hyrule:~/Scrivania/FARM_$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./farm -n 2 -q 1 -d testdir file* -t 100
0
==8252== Memcheck, a memory error detector
==8252== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8252== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==8252== Command: ./farm -n 2 -q 1 -d testdir file100.dat file10.dat file116.dat file117.dat file12.dat file13.dat file14.dat file15.dat file16.da
t file17.dat file18.dat file1.dat file20.dat file2.dat file3.dat file4.dat file5.dat -t 1000
==8252==
^\
Ricevuto SIGQUIT
64834211 file100.dat
258119464 file116.dat
532900090 file117.dat
748176663 file16.dat
1146505381 file10.dat
1485251680 file12.dat
1674267050 file13.dat
1878386755 file14.dat
2086317503 file15.dat
==8256==
==8256== HEAP SUMMARY:
==8256==     in use at exit: 0 bytes in 0 blocks
==8256==   total heap usage: 22 allocs, 22 frees, 6,855 bytes allocated
==8256==
==8256== All heap blocks were freed -- no leaks are possible
==8256==
==8256== For counts of detected and suppressed errors, rerun with: -v
==8256== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==8252==
==8252== HEAP SUMMARY:
==8252==     in use at exit: 0 bytes in 0 blocks
==8252==   total heap usage: 48 allocs, 48 frees, 51,502 bytes allocated
==8252==
==8252== All heap blocks were freed -- no leaks are possible
==8252==
==8252== For counts of detected and suppressed errors, rerun with: -v
==8252== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

9.1.3 Esecuzione standard

```
alessandra@Hyrule:~/Scrivania/FARM_$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./farm -n 2 -q 1 -d testdir file*
==7726== Memcheck, a memory error detector
==7726== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7726== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7726== Command: ./farm -n 2 -q 1 -d testdir file100.dat file10.dat file116.dat file117.dat file12.dat file13.dat file14.dat file15.dat file16.dat file
17.dat file18.dat file1.dat file20.dat file2.dat file3.dat file4.dat file5.dat
==7726==
64834211 file100.dat
103453975 file2.dat
153259244 file1.dat
193031985 testdir/testdir2/file150.dat
258119464 file116.dat
293718900 file3.dat
380867448 file5.dat
518290132 file17.dat
532900090 file117.dat
584164283 file4.dat
748176663 file16.dat
890024017 testdir/file19.dat
985077644 testdir/file8.dat
1146505381 file10.dat
1485251680 file12.dat
1674267050 file13.dat
1878386755 file14.dat
1884778221 testdir/testdir2/file111.dat
2086317503 file15.dat
2322416554 file18.dat
2560452408 file20.dat
==7727==
==7727== HEAP SUMMARY:
==7727==     in use at exit: 0 bytes in 0 blocks
==7727==   total heap usage: 46 allocs, 46 frees, 11,831 bytes allocated
==7727==
==7727== All heap blocks were freed -- no leaks are possible
==7727==
==7727== For counts of detected and suppressed errors, rerun with: -v
==7727== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==7726==
==7726== HEAP SUMMARY:
==7726==     in use at exit: 0 bytes in 0 blocks
==7726==   total heap usage: 97 allocs, 97 frees, 178,066 bytes allocated
==7726==
==7726== All heap blocks were freed -- no leaks are possible
==7726==
==7726== For counts of detected and suppressed errors, rerun with: -v
==7726== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
alessandra@Hyrule:~/Scrivania/FARM_$
```