SOL: PROGETTO FARM 2022

GIORDANO SCERRA

Architettura:

il progetto è costituito da due processi: **MASTERWORKER** e **COLLECTOR**. si è scelto di optare per un modello **una connessione per worker**, con questi come "*client*" e il collector come "*server*" dotato di un selector *sequenziale*, in ascolto anche su una **pipe senza nome** intra-processo. questa scelta è stata guidata dalla volontà di implementare più tecnologie possibili viste a lezione, e non necessariamente dalla ricerca del modello più efficiente. si poteva anche infatti implementare il programma con un solo socket, passato agli worker e acceduto in maniera protetta con mutex ed eventualmente condition variables. si sarebbe anche potuto implementare un collector multithreaded per un selector non sequenziale, ma si è reputata una scelta non significativa giudicato l'esiguo contributo ipoteticamente apportato da un threadpool di supporto. inoltre come anche spiegato più avanti, non sarebbe stato in accordo con le mire di questa implementazione. analizziamo ora i due processi.

MASTERWORKER è costituito da:

1) un **thread main** (*Masterworker.c*)
2) un **thread master** (*Master.c*)
3) un **thread signal handler** (SignalHandler in *Masterworker.c*)
4) un **threadpool di workers** (costituito da *Worker.c*)

analizziamo questi 4 punti in dettaglio:

1) **il thread main** (Masterworker.c) si occupa di spawnare il thread master, il processo collector e il thread signal handler. si occupa quindi anche di joinare questi due thread e aspettare il segnale SIGCHLD del collector, prima di chiudere. prima di chiamare la fork() per spawnare il processo collector blocca i segnali SIGINT SIGQUIT SIGTERM SIGHUP SIGUSR1 e ignora SIGPIPE, in questa maniera il processo figlio erediterà questa gestione dei segnali. contrariamente, l'installazione del cleanup con atexit non viene ereditata, poiché installato dopo la fork(). il thread main crea anche la pipe senza nome per la comunicazione intra processo tra masterworker e collector, passandola al thread signal handler. al thread master passa i propri riferimenti ad argc e argv, per il parsing. si mette quindi immediatamente in attesa che i thread spawnati ritornino.

2) **il thread master** (Master.c) si occupa di effettuare correttamente il parsing dalla linea di comando, eventualmente esplorare la cartella passata con l'opzione -d e verificare la regolarità dei file parsati. dopodiché spawna un threadpool di workers, di cui si occuperà anche di fare la join tramite un cleanup installato dopo la creazione. tutti i file parsati da elaborare sono salvati in una lista non ordinata. il Master insieme agli Worker implementa un protocollo PRODUTTORE-CONSUMATORE con lock e condition variables sulla coda

concorrente dei task da elaborare. oltre alla gestione standard del protocollo si sono aggiunti due controlli: uno è per controllare che ci sia almeno un worker online in grado di prelevare elementi dalla coda: se così non fosse, il master si chiude e verrà joinato dal main. il secondo è sui segnali di interruzione: se se ne è ricevuto uno, il signal handler setta la variabile sig_interrupted che il master controlla ad ogni ciclo in maniera thread safe, ed eventualmente uscirà dal ciclo e notificherà agli workers di chiudere. infatti dopo aver concluso di inviare i task nella coda, il master aspetta che la coda sia vuota e setta la variabile end_workers in maniera thread safe ed effettua una broadcast, svegliando tutti i worker online in attesa. dopodiché chiude e joina gli workers.

3) **il thread signal handler** (SignalHandler in *Masterworker.c*) riceve in input la maschera dei segnali da ascoltare (tutti quelli bloccati dal main) e l'end di scrittura della pipe. tramite la chiamata sigwait, questo thread intercetta i segnali di terminazione e SIGUSR1 gestendoli rispettivamente cambiando in maniera thread safe la variabile sig_interrupted (citata sopra per notificare al master di smettere di inviare task ed iniziare la fase di chiusura), e scrivendo sulla pipe il segnale SIGUSR1 (il collector leggerà dalla pipe e stamperà la lista ordinata). alla ricezione di un qualsiasi segnale di terminazione, il thread ritorna, venendo joinato dal main. alternativamente, verrà cancellato dal main tramite phtread_cancel, con cancellation point su sigwait.

4) **il threadpool di workers** (costituito da *Worker.c*) costituisce un pool di "client", che si connetteranno al "server" collector. infatti, dopo aver incrementato in maniera thread safe un contatore che segnala al master il numero di worker attivi, il worker tenterà di connettersi al collector, già spawnato: si creerà quindi un proprio socket e tenterà di connettersi un massimo volontariamente esagerato di 1000 volte, per non rimanere bloccati in un ciclo di tentata connessione. in caso non riesca a connettersi, il worker chiama la exit poiché si tratta di uno dei pochi errori fatali del programma. dopo essersi connesso il worker avrà installato una routine di cleanup che notifica al collector della sua disconnessione e chiude il socket. dopodiché il worker implementa il protocollo PRODUTTORE-CONSUMATORE e preleva elementi dalla coda concorrente. dopo aver prelevato un elemento, apre il file, esegue il calcolo e invia lunghezza del filename, risultato e filename al collector. in caso di fallimento di queste operazioni di read e write, si è optato per una chiusura soft che prevede la disconnessione e la chiusura del singolo thread. nel caso in cui la variabile end_workers venga settata da parte del master, il worker si disconnetterà dal collector e prima di ritornare chiuderà il suo socket.

COLLECTOR è costituito da un singolo thread:

il processo collector, dopo aver chiuso l'end di scrittura della pipe, crea il socket di ascolto, e dopo la bind e la listen è pronto ad accettare connessioni. se una di queste operazioni fallisce, il processo chiama inevitabilmente exit, essendo un errore fatale. si è scelto di gestire le connessioni degli worker tramite un selector, pur non sfruttando a pieno la sua potenza con un ulteriore threadpool, che si è giudicato ininfluente ai fini della scelta stessa di non implementare una singola connessione: l'intenzione era infatti quella di implementare quante più tecnologie incontrate a lezione. si tratta quindi di un semplice selector sequenziale in un while con guardia booleana. il set dei file descriptor da ascoltare, inizialmente con solo listen socket e read end della pipe, viene manipolato a seconda della caduta delle connessioni degli worker. infatti, alla ricezione della lunghezza filename di un

carattere speciale (-1), il collector smetterà di considerare quel socket azzerandone il bit annesso al fd nel set master. il selector è dotato di un timeout, semplice meccanismo di autodifesa da eventuali malfunzionamenti: infatti essendo il timer spropositatamente di 10 secondi, periodo di sospetta inattività del collector, la chiusura grazie a questo è praticamente impossibile interferisca con la normale esecuzione del programma. alla ricezione di una nuova connessione, si incrementa il contatore di client attivi. alla ricezione di una richiesta di chiusura connessione, si controlla se questo contatore non sia 0. in tal caso il collector ha terminato correttamente il suo lavoro e può chiudere, dopo aver chiuso il listen socket e il read end della pipe. se sulla pipe c'è scritto qualcosa, leggo: se si tratta di EOF disconnetto la pipe chiudendone il file descriptor. se si tratta del segnale SIGUSR1 posso stampare il contenuto della lista ordinata fino a quel momento. prima di ritornare il programma stampa la lista dei risultati in maniera ordinata e ne libera la memoria allocata.

Strutture utilizzate

per la realizzazione del programma si sono utilizzate due code: *boundedqueue.c* e *orderedqueue.c* . la prima è una versione di un codice fornito dal professore modificata personalmente in maniera da poter implementare un protocollo produttore consumatore esterno personalizzato in Master.c e Worker.c. la seconda proviene da un vecchio assignment di Programmazione I con il professor Priami, in cui si può scegliere se inserire in maniera ordinata o semplicemente in testa (diminuendo così la complessità). boundedqueue è quindi diventata una semplice lista limitata, un array circolare. queste due code si trovano nella libreria condivisa e linkata dal processo farm (libQueues.so). si è scelto inoltre di abbandonare l'utilizzo di threadpool.c (anch'esso fornito dal professore) per poter implementare una versione del threadpool più scarna, con pending queue gestita manualmente. per l'ispirazione nella realizzazione degli altri componenti del programma si è consultato con più frequenza siti come IBM.com (es. select) e discussioni su stackoverflow.com (es. parsing). gli header file conn.h e util.h sono stati forniti dal professore e modificati personalmente. Si possono trovare altri esempi di codice fornito dal professore, come le funzioni per sfogliare le directories o la stessa struttura del thread signal handler.

Considerazioni

il programma è stato testato in diverse situazioni di errore, rispondendo sempre in maniera positiva. infatti si è cercato di realizzare un protocollo **uniforme** di terminazione bug-free ,con cleanup handlers e timers. si è cercato dove possibile di evitare chiusure troppo brusche e si è preferito un approccio più guidato verso una chiusura con rilascio di risorse. un esempio di questo è riportato nel target "make mytest", con particolare attenzione all'ultimo test, il test 6. si noti infatti che in Worker.c si trovano delle righe commentate come TEST (da riga 61 a riga 73) , queste servono infatti a chiudere un worker alla ricezione di un determinato file. si vuole infatti in questo test sottolineare come risponda il programma nella situazione in cui il master si ritrova con nessun worker online nel mezzo della sua esecuzione. come si potrà controllare, entrambi i processi terminano **correttamente** e

rilasciando tutte le risorse (valgrind). la possibilità di deadlock è stata studiata nel file schemalock.txt nella cartella RELAZIONE, ed è quindi stata giudicata impossibile l'eventualità. il progetto è stato sviluppato su windows WSL e al momento della consegna passa correttamente tutti i test sottoposti dal professore.

```
giordano@LAPTOP-9AO39JKJ:~/FARM_OK/FARM$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.5 LTS
Release:        20.04
Codename:       focal
giordano@LAPTOP-9AO39JKJ:~/FARM_OK/FARM$ lsb_release -d
Description:    Ubuntu 20.04.5 LTS
```

```
giordano@LAPTOP-9AO39JKJ:~/FARM_OK/FARM$ make test
chmod +x test.sh
./test.sh
test1 passed
test2 passed
test3 passed
test4 passed
test5 passed
giordano@LAPTOP-9AO39JKJ:~/FARM_OK/FARM$
```

26/10/2022 - GIORDANO SCERRA