Progetto SOL FARM – Corso A e B – a.a. 22/23, Andrea Lepori , matricola : 602620

Relazione del progetto

Questo documento costituisce la relazione sul progetto Farm, contiene una descrizione generale del funzionamento del progetto, una panoramica sulle principali scelte implementative lasciate a discrezione dello studente e informazioni riguardanti i test aggiuntivi effettuati per verificare il corretto funzionamento del programma. Il progetto è stato sviluppato tramite l' IDE Visual Studio Code su una macchina Windows 11 ed è stato testato su WSL Ubuntu, e sulle macchine virtuali xubuntu e Ubuntu 20.04.4 su VirtualBox.

Funzionamento generale ed architettura del progetto

Il programma C "farm" deve ricevere in input una serie di file (o direttamente da linea di comando od eventualmente cercandoli in una directory fornita come valore dell' opzione -d) e delle opzioni. Il programma si suddivide logicamente in 2 processi, il primo è il processo MasterWorker ed il secondo, generato dal primo, è il processo Collector. Questi due processi, comunicando tramite socket AF_UNIX AF_LOCAL cooperano tra loro, il processo MasterWorker si occupa di parsare le opzioni e gli argomenti forniti al programma da linea di comando, avviare un threadpool di thread worker a cui il thread MasterMain via via sottomette i task con i file da elaborare attraverso l' uso di una coda concorrente per la sincronizzazione tra threads. Ciascun worker thread del pool esegue una computazione sui dati del file binario il cui pathname preleva dalla coda concorrente e si occupa attraverso una propria connessione socket stabilita all' attivazione del thread, di comunicare all' altro processo, il Collector, il risultato della computazione ed il file su cui è stata eseguita. Il processo collector agisce da server e mantiene una struttura dati ordinata con i risultati delle computazioni sui vari file che aggiorna via via che i worker del pool che agiscono da client glielo chiedono. Il processo MasterWorker, in particolare il thread MasterMain, una volta esauriti i task da sottomettere al threadpool o perché già sono stati tutti sottomessi o perché è sopraggiunto un segnale di terminazione, ordina una chiusura "dolce" del threadpool che fa si che non siano accettati più task in ingresso ma che siano smaltiti quelli già presi in carico. Una volta terminato lo smaltimento dei task in coda, viene terminato ciascuno thread del pool e chiuse le loro rispettive linee di comunicazione col collector, infine viene distrutto il threadpool. A questo punto thread il MasterMain del processo MasterWorker si incarica di aprire una propria connessione col Collector per notificare la terminazione del lavoro e di dare inizio alle operazioni di terminazione del processo. Il processo collector, che fa da master nella comunicazione interprocesso, si mette in ascolto di richieste di connessione con una listen ed è in grado di gestire più connessioni da client diversi che comunicano secondo un protocollo di comunicazione prestabilito, attraverso il costrutto select. I vari worker del pool possono richiedere con un particolare codice l' inserimento di un nuovo record nella struttura dati mantenuta dal server che via via viene aggiornata e mantenuta ordinata. Una volta ricevuta la notifica di avvio del processo di terminazione e una volta chiuse tutte le connessioni con i vari client, il processo collector stampa ordinatamente i record memorizzati, dealloca le strutture di memorizzazione e termina. Il processo MasterWorker nel thread MasterMain era in attesa della terminazione del processo figlio (che è il collector) e una volta risvegliatosi dalla waitpid termina anc'esso.

Guardando più nel dettaglio all'architettura, il programma C "Farm" è composto da due processi, il processo MasterWorker ed il processo Collector. Il programma deve ricevere in input una serie di file (o direttamente da linea di comando oppure eventualmente cercandoli in una directory fornita come valore dell' opzione -d) e delle opzioni. Il processo MasterWorker è un processo multi-threaded, prevede un thread principale, qui denominato masterMain, che si occupa di parsare le opzioni e gli argomenti con la funzione getopt, di avviare il threadpool settando alcune impostazioni tramite i valori passati in input alle opzioni come il numero di thread del pool e la dimensione della coda concorrente e di passare in rassegna sia i file passati al programma direttamente come argomenti a linea di comando, sia quelli eventualmente trovati esaminando la cartella passata come valore dell' opzione –d (se presente), e di sottometterli al threadpool per essere processati. Il processo MasterWorker si duplica attraverso la funzione fork, generando il processo figlio che viene subito differenziato con una exec per mettere in esecuzione il codice che effettivamente implementa il processo collector. Il processo collector agisce da server nella comunicazione, si mette in ascolto di richieste di connessione ed è in grado di gestire connessioni diverse provenienti da client diversi andando a realizzare il protocollo di comunicazione stabilito tra i due processi. Il thread MasterMain del processo MasterWorker e i vari thread worker del threadpool, cooperano attraverso una coda concorrente di task da elaborare. Questa coda ha una dimensione limitata, il processo MasterMain vi inserisce task man mano che esamina l' input e i thread del pool li prelevano. Questo meccanismo è sincronizzato attraverso una lock che garantisce l'accesso in mutua esclusione alle strutture dati condivise tra più processi e due variabili di condizione, notfull su cui si sospende il produttore se la coda è piena in attesa di avere spazio per poter inserire un task e notempty su cui si sospendono i vari thread consumatori in attesa che la coda non sia vuota e che quindi ci siano task da elaborare, andando ad implementare quello che è un vero pattern di cooperazione del tipo produttore/consumatore. Ciascuno thread del pool ha una connessione socket AF_UNIX AF_LOCAL dedicata col

collector, che viene stabilita alla creazione del thread e chiusa quando il thread viene terminato, ogni volta che un thread ha eseguito un task (una computazione sui dati long di un file regolare binario), comunica al collector il risultato del calcolo ed il pathname del file proprio su quella connessione con un protocollo di comunicazione prestabilito che prevede dei codici che identificano il tipo di richiesta che il client manda al server. Quindi le comunicazioni relative a diversi task elaborati da uno stesso thread viaggiano tutte sulla connessione propria di quel thread worker. Il collector, agisce da server, mantiene una linked list ordinata di nodi che memorizzano coppie (risultato della computazione, pathname del file su cui è stata eseguita), l'ordine è crescente sulla base della chiave che è il risultato della computazione ed ogni volta che un client manda una richiesta di inserimento di un nuovo nodo il server è in grado di gestirla andando ad inserire ordinatamente nella lista. Il thread MasterMain del processo masterWorker dispone anch'esso di una connessione col processo collector (quindi le connessioni sono una per ogni thread worker ed una per il thread masterMain ed una riservata al signal handler inizializzata sempre nel MasterMain di cui parleremo successivamente). Per quanto riguarda il pattern seguito per la comunicazione inter-processo tra MasterWorker e Collector si ribadisce che è il collector a fare da master per la comunicazione e in un paradigma di comunicazione client-server rappresenta il server: i thread del processo MasterWorker (thread worker e thread MasterMain) sono i client che vogliono connettersi sui vari socket per instaurare delle connessioni, quindi se al momento della connect il server ha già eseguito la bind e la listen ed è quindi in ascolto sull' indirizzo del socket, la richiesta di connessione viene accettata dal server con una accept e la connessione instaurata, altrimenti il client è costretto ad aspettare un po' e riprovare a connettersi finchè non ci riesce.

Breve descrizione del processo masterWorker

Il processo MasterWorker si deve occupare di gestire l' input, sottomettere i task al threadpool (inserendoli quindi nella coda concorrente dei task da elaborare), gestire il threadpool dei thread worker che eseguono il calcolo dei risultati sul file e comunicano col collector scrivendo sulle proprie socket, gestire la terminazione del programma e gestire i segnali in arrivo.

Gestione dell' input : il parser incaricato per la gestione dell' input fa uso della funzione getopt() ed è fondamentale che se ne utilizzi la sua implementazione GNU. Gli argomenti delle opzioni –n, -t, -q e –d, se queste sono presenti, sono passati a delle funzioni che verificano ed eseguono la conversione a long andando a modificare i valori delle variabili che contengono i valori di default di questi parametri. Per la lettura dei nomi dei file passati in input è fondamentale considerare che l'implementazione GNU della funzione getopt esegue una permutazione di argv in modo da mettere prima le opzioni con rispettivi valori se queste sono presenti tra gli argomenti passati a linea di comando e mettere in fondo al vettore gli eventuali argomenti non opzionali : [opzioni gestite da getopt ←,→argomenti non opzionali]. Getopt si arresta una volta incontrato il primo argomento non opzionale quindi per leggere i file sarà sufficiente scorrere argv dall'indice in cui si arresta getopt fino alla fine del vettore.

Sottomissione dei task al threadpool (inserimento nella coda concorrente)

Dopo aver terminato il parsing con la funzione getopt, sfruttando il particolare ordinamento di argv descritto sopra, con un ciclo for viene scorso tutto questo vettore per i che va da optind ad argc-1, all' interno del corpo del loop prima si dorme per il tempo di distanziamento nella sottomissione dei task specificato con l' opzione –t (oppure 0 di default), utilizziamo per dormire una funzione wrapper che utilizza nanosleep(), poi si controlla se il file è un file regolare e in caso positivo si chiama la funzione addToThreadpool che aggiunge il task alla coda concorrente dei task da eseguire. Essendo questa struttura acceduta in concorrenza dal thread produttore e dai worker thread è protetta con l'accesso in mutua esclusione attraverso una lock ed è presente un meccanismo di sincronizzazione di tipo produttore/consumatore più elaborato che fa uso di due condition variable. In particolare se la coda è piena e non c'è spazio per l' inserimento il produttore si sospende sulla variable di condizione coda piena e attende di essere risvegliato da un worker con una signal non appena viene prelevato un task , liberando così un posto. Se è stata fornita l' opzione –d con valore il path di una cartella viene chiamata la funzione find che esplora ricorsivamente il sottoalbero del file system con radice quella cartella in cerca di file regolari da sottomettere nelle medesime modalità al threadpool.

Gestione del threadpool

Il threadpool viene creato subito dopo il parsing delle opzioni dal thread MasterMain attraverso la funzione createThreadpool() a cui vengono passati come parametri il numero dei thread del pool e la lunghezza della coda concorrente dei task. La struttura threadpool_t incapsula al suo interno tutti i meccanismi necessari ad implementare la sincronizzazione della coda condivisa dai worler e MasterMain e la mutua esclusione nell' accesso alle risorse condivise fra più threads, qualsiasi essi siano, se vogliono accedere ad una risorsa condivisa del pool devono acquisire la lock per accedere una sezione critica in mutua esclusione. I threads del pool sono rappresentati con l' array pthread_t * threads

di dimensione num_threads, la coda dei task pendenti è taskfun_t * pending_queue ed è realizzata come una array circolare di task di dimensione limitata queue_size, poi ci sono una serie di variabili come taskonthefly che rappresenta i task in esecuzione al momento, i puntatori alla testa e al fondo della coda per inserimenti e prelevamenti e una variabile exiting che se maggiore di 0 indica che è iniziato il protocollo di uscita, in particolare se uguale ad 1 un thread prima di terminare aspetta che non ci siano più lavori in coda. Una volta creata ed instaurata la connessione col collector, fino alla sua terminazione un thread del pool esegue un ciclo in cui acquisisce la lock, finchè la coda è vuota e non deve uscire si sospende sulla variabile di condizione not_empty, una volta che è stato risvegliato dalla signal di un produttore, ora che ha la certezza che la coda non sia vuota, preleva un task dalla coda, invia una signal al produttore per notificare il verificarsi della condizione coda non piena e rilascia la lock per eseguire la funzione compute sul file prelevato dalla coda. L' esecuzione della funzione compute che implementa il cuore del lavoro del worker esegue la computazione di interesse sui dati del file, e questa computazione non richiede la mutua esclusione e può andare in parallelo vero. Una volta eseguito il calcolo il worker scrive il risultato e il nome del file su cui è stato eseguito sulla socket che identifica la connessione col collector. Infine richiede la lock per aggiornare delle variabili interne e riesegue il ciclo.

Gestione dei segnali

Inizialmente i segnali di interesse vengono tutti mascherati fino a prima dell' installazione del signal handler, che viene fatta dopo l' instaurazione di una connessione col collector che verrà utilizzata dal signal handler stesso. Per il segnale SIG_PIPE viene installato come gestore SIG_IGN che ignora il segnale, mentre per gli altri segnali di interesse viene installato un apposito signal_handler. I segnali che richiedono come gestione la terminazione del programma come SIGINT, SIGHUP, SIGQUIT, SIGTERM vanno a settare ad un variabile volatile sig_atomic termina , sig_atomic_t è un tipo integer che può essere acceduto come un' entità atomica in presenza di interruzioni asincrone fatte da segnali. Questa variabile sarà poi controllata dal thread MasterMain per terminare non appena settata, evitando così di sottomettere al pool nuovi task se in fase di terminazione. Alla ricezione del segnale SIGUSR1 invece il signal handler deve notificare il collector di stampare la lista dei risultati inseriti fino a quel momento, per questo verrà scritto il codice di richiesta di stampa sul socket della connessione riservata al signalhandler col collector tramite la funzione write che è asynchronous signal safe.

Gestione della terminazione del processo

Una volta terminati i task da inserire nella coda, o perché sono stati già tutti inseriti o perché è sopraggiunto un segnale di terminazione, viene ordinata la chiusura "dolce" del threadpool ovvero, non sono accettati più nuovi task ma prima di terminare e chiudere il threadpool si smaltiscono i task accumulati nella coda concorrente. Una volta che la distruzione del threadpool è avvenuta e quindi si è attesa la terminazione di tutti i thread del pool, viene aperta una connessione col collector per notificare la terminazione del programma, si chiude questa connessione e la connessione riservata al signal handler e si attende la terminazione del processo figlio che è il collector, una volta arrivata si termina con successo.

Breve descrizione del processo Collector

Il processo collector per prima cosa maschera tutti i segnali gestiti dal processo MasterWorker, si occupa di mettersi in ascolto di richieste di connessione e gestire le connessioni aperte, infine terminare la sua esecuzione stampando i risultati dei calcoli sui file in modo ordinato.

Gestione dei segnali

Il collector maschera tutti i segnali che sono gestiti dal processo masterWorker ed ignora SIGPIPE.

Gestione delle richieste di connessione e delle connessioni

Il collector è un processo single threaded, quando viene lanciato, dopo aver mascherato tutti i segnali inizializza la lista linkata che sarà utilizzata per memorizzare i risultati ordinatamente ed inizializza alcune sue variabili interne tra le quali è importante elencare un flag di terminazione e un contatore delle connessioni gestite correntemente, entrambe inizializzate a 0. Dopo aver eseguito le op. di bind e di listen il collector è pronto ad accettare richieste di connessione. Dato che il collector deve essere in grado di gestire richieste provenienti da più client finchè non sopraggiungono le condizioni di terminazione il collector esegue in un while il costrutto select per monitorare se ci sono file descriptor pronti per essere processati tra quelli che via via si sono registrati. Se il file descriptor è quello del listening socket vengono accettate nuove richieste di connessione con la accept e i file descriptor delle connessioni col client verranno registrati al selector, se i file descriptor pronti sono quelli di socket di input/output si eseguono delle read per capire il tipo di richiesta del client e trattarla opportunamente. Se si legge un end-of-file la connessione è chiusa e il socket deregistrato.

Gestione della terminazione del processo

Una volta ricevuta la richiesta di terminazione dal processo MasterWorker si attende la chiusura di tutte le connessioni intavolate dal server coi vari client, quindi si stampa la lista ordinata con le coppie risultato-pathname_file, si dealloca la struttura/lista che le memorizza, viene cancellato il socket file con la funzione unlink() e si termina.

Scelte implementative lasciate a discrezione dello studente

Quale dei due processi MasterWorker e Collector fa da processo master per la connessione socket?

Il processo Collector esegue la bind e la listen quindi fa da master per la connessione socket, mentre ad agire da client cercando di connettersi facendo la connect sono i thread del processo MasterWorker. Dato che il processo collector è generato dal processo masterWorker con una fork e con una exec e un client nel MasteterWorker potrebbe fare la connect quando il listening socket lato server non è ancora pronto tutte le operazioni di connect sono state inserite in un ciclo while in cui si testa se la connect è fallita per questo motivo e in tal caso si apetta un po' e poi si ritenta di connettersi.

Viene usata una sola connessione o più connessioni, una per ogni worker?

Viene usata una connessione per ogni thread worker del threadpool, una connessione per il thread MasterMain ed un' altra connessione, instaurata sempre all' inizio del thread MasterMain, che però è stata riservata al signal handler. Questo modello di comunicazione permette a ciascun thread worker di poter comunicare subito, una volta calcolato, il risultato al collector, senza bisogno di dover passare dal thread MasterMain, cosa che averebbe implicato il dover definire una mutex per l'accesso controllato ad una connessione condivisa, formando così un collo di bottiglia le cui prestazioni sarebbero state limitate dal thread più lento. Questo modello fa si che le prestazioni non siano dettate dal thread più lento tuttavia più connessioni aperte hanno un impatto maggiore sull' pressione in memoria rispetto ad una soluzione con una sola connessione : si è cercato un compromesso, non viene aperta e chiusa una connessione per ogni task ma una connessione persiste per tutta la vita di un thread quindi viene riutilizzata più volte, inoltre la connessione non è unica ma non sono nemmeno esageratamente numerose dato che i worker solitamente sono molti meno dei task. La connessione del thread MasterMain è creata prima della terminazione, una volta chiuse tutte le altre connessioni del pool per notificare la terminazione al collector. L' altra connessione del MasterMain è instaurata all' inizio prima della registrazione del signal handler ed è riservata a quest' ultimo per comunicare i messaggi di stampa al collector. Sono necessarie due connessioni separate (quella per notificare la terminazione e quella del signal handler), nonostante la connessione per notificare la terminazione sia aperta e utilizzata solo per l' invio di un messaggio e poi chiusa subito, la fusione di queste due connessioni in una sola avrebbe potuto dar luogo a race condition che avrebbero richiesto per essere evitate l'uso di una lock che non può essere inserita nel signal handler.

Protocollo di comunicazione tra i due processi

Il protocollo seguito per la comunicazione tra i client e il server prevede il solo scambio di messaggi unidirezionale da client a server: per prima cosa viene inviato un long che rappresenta il codice del messaggio e ne descrive il tipo (0 → richiesta di inserimento in coda di un nuovo record, 1 → messaggio di stampa della lista ordinata dei risultati e del rispettivo nome del file, 2 → notifica di terminazione). Se il codice inviato è 1 o 2 il server non si aspetta altri messaggi e provvede a soddisfare la richiesta del client, se invece il codice era a 0 seguono il codice un altro long che rappresenta il risultato della computazione, un long che identifica la lunghezza della stringa con il pathname del file ed infine la stringa con il pathaname del file.

Suddivisione in file

In questa sezione è illustrata la strutturazione in file del programma.

FILE COMUNI

Di seguito i file comuni ai due processi :

- [util.h, util.c] : header file e rispettiva implementazione, il primo contiene alcune define e dichiarazioni di funzioni di utilità generale, il secondo ne contiene l'implementazione.

- [communication.h] : header file contenente alcune define e include necessari alla realizzazione della comunicazione tramite socket AF_UNIX AF_LOCAL e la definizione delle funzioni wrapper writen e readn che evitano rispettivamente scritture e letture parziali.

MASTERWORKER

Di seguito i file del processo masterworker :

- [threadpool.h, threadpool.c] : il primo contiene l' interfaccia per il threadpool il secondo ne contiene l'implementazione.
- [worker.h, worker.c] : il primo è l'header file che contiene la dichiarazione della funzione compute che implementa la computazione che un thread worker esegue sul file binario, l' implementazione della funzione compute è contenuta nel secondo file.
- [masterWorkerMain.c] : file attraverso cui viene fatto partire il programma, attraverso una fork viene duplicato il processo, il processo padre (masterWorker) si occupa di parsare gli argomenti del programma, istanziare il threadpool e sottomettere allo stesso i vari task, il processo figlio viene subito differenziato con una exec che manda in esecuzione il collector.

COLLECTOR

Di seguito i file del processo collector :

- [sortedlist.h] : contiene la definizione della struttura nodo di una lista e la definizione di tutte le funzioni necessarie ad implementare una linked list con inserimento ordinato. L' inserimento di un nuovo nodo nella lista avviene in ordine crescente sulla base del campo long data che contiene il risultato della computazione di interesse eseguita sul file file_name.
- [collector.c] : questo file contiene l'implementazione del processo collector che si mette subito in ascolto in attesa di richieste di connessione e una volta instaurate delle connessioni è in grado di gestirle implementando il protocollo di comunicazione tra i due processi.

Istruzioni per la compilazione e per l'esecuzione del progetto FARM.

La compilazione e l' esecuzione sono state previste per essere effettuate tramite il makefile scritto appositamente. Per prima cosa posizionarsi all' interno della cartella del progetto ed eseguire il seguente comando per la generazione dei file e della cartelle di test: "make generafile". Per compilare il progetto eseguire il comando : "make farm" e successivamente eseguire il comando "make collector". Per eseguire lo script di test fornito dal professore "test.sh" eseguire il seguente comando: "make test". E' possibile eseguire subito il comando "make test" dopo aver generato i file di test senza aver precedentemente esguito la compilazione esplicitamente con il comando del make. Per eseguire lo script loop.sh che esegue per 50 volte in sequenza lo script di test digitare il comando "make looptest". Per eseguire lo script "my_test.sh" contenente dei test aggiuntivi digitare il comando "make mytest". Per eseguire il programma con impostazioni di default digitare "make exec", per eseguire il programma con impostazioni di default e utilizzando valgrind digitare : "make valg", per ripulire la cartella dai file generati digitare "make clean", per riportare la cartella allo stato iniziale digitare: "make cleanall". In sistesi per eseguire i test digitare nella cartella del progetto → "make generafile" e in seguito "make test".

Test aggiuntivi effettuati

Il progetto è stato testato sui test presenti nel file "test.sh" fornito dal professore con esito positivo. Inoltre sono stati eseguiti ulteriori test che si possono visionare nel file "my_test.sh" in cui sono stati variati i valori delle opzioni di –n, -q e -t per verificare il funzionamento con differenti dimensioni del threadpool, differenti lunghezze della coda concorrente e differenti ritardi di sottomissione dei task al pool, inoltre in questa batteria di test aggiuntivi sono stati testati anche gli altri segnali che non erano stati coperti nella prima batteria di test. Per alcuni segnali si è testato anche il comportamento alla ricezione in sequenza di segnali differenti, mentre per il segnale SIGUSR1 si è testato il comportamento del programma alla ricezione immediatamente consecutiva di due di questi segnali. Il programma ha dato esito positivo anche su questa batteria di test. Inoltre sono stati scritti due script, il file loop.sh esgue per 50 volte la batteria di test forntita dal professore "test.sh" e il file "benchmark.sh" è stato scritto per misurare il tempo di esecuzione del programma sotto condizioni fissate (4 worker , coda lunga 8 e nessun ritardo di sottomissione). Tutti i test sono stati eseguiti con esito positivo sia su WSL Ubuntu, e due macchine virtuali su Virtualbox xubuntu (presente nella pagina ufficiale del corso) e Ubuntu 20.04.4. .