RELAZIONE: FARM

Progetto SOL - Corso A e B – a.a. 22/23

DEODATI MICHELA 597983

Introduzione:

Il progetto si compone di un totale di 8 file su cui viene separato il codice. I file di maggiore rilevanza sono il “*MasterThread.c*”, “*WorkerPool.c*”, ”*Collector.c*”. In generale i file “*WorkerPool.c*” e “*WorkerPool.h*” implementano la threadpool di workers dalla creazione alla distruzione, così come il codice della task che ogni singolo worker deve eseguire. I file “*Collector.c*” e ”*Collector.h*” implementano il processo che stabilisce la connessione con i workers e attende che gli vengano mandati i risultati delle task per poi stamparli ordinatamente. Il processo core di tutto il funzionamento è quello implementato da “*MasterThread.c*” e ”*MasterThread.h*” che svolge il compito di gestore della threadpool e ricettore dei segnali di terminazione e di stampa tramite un thread signal handler. Il controllo dell’input e la ricerca nella cartella viene lasciato al main. Nel file “*Util.h*” sono presenti macro, funzioni e #define che sono di comune utilizzo. La linea generale di funzionamento è: vengono passati file e opzioni di personalizzazione della threadpool come argomenti del main, viene controllato immediatamente che il numero di argomenti sia maggiore di zero, poi viene fatto un controllo più approfondito vengono esplorate eventuali cartelle, poi viene chiamata funzione ***runMasterThread()*** a cui vengono passati i parametri del main e un array contente i nomi dei file. ***RunMasterThread****()* esegue la fork() per invocare il processo Collector come figlio, crea e avvia il signalhandler e la threadpool, passa i file e la task a ogni worker, attende la terminazione del Collector per terminare, anche in caso di ricezione di segnale. Il return del main è il valore ritornato da ***runMasterThread().***

WorkerPool:

“WorkerPool.c” e “WorkerPool.h” sono i due file che riguardano la threadpool e la task svolta dai thread. La task consiste nel leggere dei valori **long** maggiori di zero da un file passato come stringa alla funzione “*leggieSomma”*(tread task vera e propria), calcolare la somma di tutti questi valori moltiplicati per la loro posizione i all’interno del file e comunicare, in un'unica stringa, tramite connessione socket AF\_UNIX farm.sck il valore ottenuto e il nome (o percorso) del file al processo Collector.   
Nello specifico la funzione “***leggieSomma****”* esegue i seguenti passi:  
1. Crea la socket controllando che sia svolto tutto correttamente.  
2. Converte il valore void\*arg nella stringa filePath e apre il file in modalità “rb”(lettura binaria) calcolandone la dimensione tramite il risultato della funzione “***getFileSize()***”, che impiegando la fseek() sposta l’indice alla fine del file, con la ftell() ricava l’indice che corrisponde alla dimensione in byte del file, sempre tramite la fseek() riporta il puntatore in testa al file e ritorna la dimensione ottenuta. Dividendo poi questa dimensione per sizeof(long) ottengo quanti numeri ci sono esattamente nel file, a questo punto viene allocato dinamicamente un long fileArray che conterrà i valori letti dal file.  
3. Leggo tutti i valori del file con la fread, passandogli direttamente la dimensione del file e il puntatore alla testa del fileArray, eseguo la somma, alloco il buffer per scrivere e aspetto che il Collector faccia l’accept().  
4.Invio filePath e valore del file al Collector assicurandomi che venga scritto ogni byte in una sola volta.  
  
Parliamo ora delle funzioni che gestiscono la threadpool dalla creazione, alla distruzione. Sono definite tre struct: “*workerthread\_t*”, ”*workertask\_t*” e ”*workerpool\_t*” che rappresentano rispettivamente un generico thread appartenente alla pool, la funzione e gli argomenti della task di ogni worker, la threadpool con tutti i suoi parametri.  
La funzione **createWorkerpool** crea in toto la threadpool: alloca la coda delle task e inizializza i parametri come il numero massimo di thread, la mutex sulla coda, le condizioni di accesso alla stessa, gestisce gli errori nella creazione della threadpool stampando all’occorrenza cosa è andato storto e ritornando NULL in caso di insuccesso. I thread creati non vengono inizializzati direttamente con la task *leggieSomma* ma vengono inizializzati con la generica task “***wpoolWorker***” a cui viene passata la threadpool, nella quale il thread tenta di acquisire la mutex sulla coda per rimuovere una task ed eseguirla, ma solo se la coda ha task da eseguire. Dopo aver acquisito la lock la prima cosa che fa è controllare che la coda delle task non sia vuota altrimenti aspetta un segnale(pthread\_cond\_wait()). Se la pool fosse in fase di uscita si controlla se ci sono ancora task pendenti. Quando finiscono le task in coda (pendingQueueCount=0) il thread termina. Ogni thread prende una task dalla coda (FIFO), incrementa le active task riposiziona correttamente la testa avendo cura che non sfori la dimensione della coda stessa. Dopo aver sistemato la testa invia un signal rilascia la lock, esegue leggieSomma sugli argomenti messi in coda, riacquisisce la lock, decrementa le active task e se ci sono ancora task in coda aspetta una signal per essere svegliato.  
La funzione **destroyWorkerpool** è la funzione che si usa per chiudere la threadpool e liberare le risorse allocate, è in 2 modalità: uscita forzata, uscita con attesa delle task. L’uscita forzata distrugge la pool senza preoccuparsi di aspettare i thread. L’uscita con waitTask=true fa la join su tutti i thread della pool dopo aver lanciato un segnale in broadcast per svegliarli tutti. Dopo la fase di waitTask viene chiamata la freeWorkPool() che libera le risorse allocate. Infine abbiamo la funzione addTask che permette di aggiungere una task alla queue. Alla addTask viene passata la threadpool  
e una stringa già castata a void\* che contiene il nome/path del file. La addTask esegue la lock sulla coda e inserisce la task alla fine, assicurandosi che il puntatore di fine coda non sfori la dimensione massima. Viene poi lanciata una signal e rilasciata la lock sulla coda.

MasterThread:

Il processo MasterThread è il core del progetto implementa la gestione dei segnali e la gestione della threadpool. La funzione principale è ***“runMasterThread()”*** che come prima cosa alloca le risorse per la threadpool, gestisce poi il segnale SIGPIPE per evitare di terminare da una scrittura su una socket. Dopo di che crea una pipe senza nome che serve da comunicazione tra il signal handler e il Collector principalmente, ma anche come mezzo di comunicazione tra il MasterThread stesso e il Collector in caso di errore. Esegue la fork() controllando che non ritorni un valore negativo, altrimenti termina. Nel processo figlio(pid=0) chiama la funzione che avvia il Collector. Nel ramo “parent” crea il avvia il signal handler in modalità detach in modo che una volta terminato le risorse vengano retituite direttamente al sistema senza che ci sia bisogno di fare la join. Se fino a qui non ci sono stati errori inizia la fase di passaggio delle task una alla volta alla threadpool tramite la funzione **addTask().**

Collector:

Nelle specifiche del progetto era stata lasciata libera la scelta se usare il MasterThread o il Collector a fare da server per la comunicazione dei risultati calcolati dai worker. In questo caso ho implementato il Collector come il server della connessione principalmente per due motivi: i workers non devono condividere una singola socket su cui scrivere ma ciascuno ha la propria che gestisce da sè dall’apertura alla chiusura. Secondo motivo è la terminazione e la stampa tramite ricezione di segnale: realizzando il Collector come un selectserver ho potuto aggiungere al master set della select la pipe *signal\_pipe* precedentemente allocata nel MasterThread sulla quale il thread signalHandler può scrivere “s” in caso di ricezione di segnale SIGUSR1 per notificare al Collector che deve stampare e “t” in caso di ricezioni di segnali di terminazione per notificare al Collector di terminare l’esecuzione.  
Al processo Collector vengono passati come argomento il numero di file che sono stati passati al main più quelli contenuti nella cartella, in più viene passata la pipe che permette di comunicare col signal handler in caso di ricezione segnale, sia di terminazione che di stampa. Il funzionamento del Collector consiste nel creare una socket AF\_UNIX sulla quale si mette in ascolto dei risultati che gli vengono inviati dai workers. I risultati vengono allocati in un array di stringhe, nessuno dei client viene salvato nel set perché i worker aprono e chiudono la connessione a ogni task, quindi, pur registrandoli nel set, nessuno comunicherebbe nuovamente su quei file descriptor. La stampa viene eseguita tramite la funzione *stampaRisultati()* la quale ordina tramite il qsort() gli elementi dell’array prima di stamparli. Il qsort() lavora tramite la funzione compare() che sfrutta il carattere ‘spazio’ che si trova tra il valore calcolato del file e la path per ricavare il long calcolato e ordinare l’array in modo crescente.

main.c:

Nel file main.c vengono eseguite le operazioni di controllo dell’input e di ricerca dei file che poi verranno passati al master thread. L’operazione di ricerca nelle cartelle viene eseguita tramite la findFileDir() a cui viene passato il nome di una cartella, il puntatore ad un array di stringhe e un indice. Nel puntatore all’array di stringhe verrà allocato, nella posizione corrispondente all’indice, i file trovati nella cartella, ma solo se sarà conforme per la funzione isFile() che serve a controllare se quello che gli viene passato è un regular file oppure no. FindFileDir() esplora l’albero della directory tramite una simil-BFS cioè procede per livelli salvando in un array di stringhe creato all’inizio della funzione i nomi della sottocartelle trovate in quella passata, le quali poi verranno esplorate ricorsivamente.  
Nel main.c sono inoltre presenti funzioni che controllano i parametri letti tramite la getopt() che se sono negativi o riportano qualche errore saranno settati al valore di default corrispondente a ciascun parametro. Nel main vengono inoltre contati quanti file vengono passati, inclusi quelli nella cartella. Il main termina con il valore di ritorno della funzione runMasterThread(): 0 se è andato tutto bene 1 altrimenti e si occupa di libeare la memoria che aveva allocato per salvare i file.

Util.h:

In Util.h vengono implementate delle funzioni di comune utilizzo a tutti i processi. Di particolare rilevanza sono le funzioni ***writen()*** e ***readn()***, (tratto da “Advanced Programming In the UNIX Environment” by W. Richard Stevens and Stephen A. Rago, 2013, 3rd Edition, Addison-Wesley), che servono per assicurarsi che venga scritto e letto tutto il buffer e in caso di valore di ritorno negativo permettono di gestire l’errore. La macro REMOVE\_SOCKET() che esegue l’unlik della socket. Util.h contiene inoltre le #define di elementi comuni a tutti i file come il nome della socket, UNIX\_PATH\_MAX, PATH\_LEN. La funzione StringToNumber() serve invece per trasformare correttamente una stringa in un valore intero, controllando che non vi siano errori, altrimenti ritorna valore negativo, usata principalmente per convertire i valori ‘optarg’ di ogni opzione del getopt(), tranne la directory.

Informazioni sulla compilazione:

Con **make** crea gli eseguibili *farm* e *generafile.*  
Con **make clean** cancella i file .o e le lib.a.  
Con **make Soket** esegue una rm -r ./farm.sck, da usare in caso di errore non previsto.  
Con **make runtest** fa partire lo script test.sh che esegue i test, da utilizzare solo dopo aver generato gli eseguibili.  
Con **make cleanall** rimuove tutti i file generati dal make e tutti i test lasciando solo i .h .c .sh.