RELAZIONE: FARM

Progetto SOL - Corso A e B – a.a. 22/23 **05.05.2023**
DEODATI MICHELA 597983

Introduzione:

Il progetto si compone di un totale di otto file su cui viene separato il codice. In linea generale MasterThread.h/.c si occupa di stabilire una connessione con il Collector, implementa e avvia il signal handler, crea e inserisce task nella threadpool e si occupa di liberare la memoria alla fine dell'esecuzione. Workerpool.h/.c si occupa di implementare al threadpool, dalla creazione alla distruzione e liberare la memoria, in più implementa la funzione che legge il contenuto dei file e ne esegue i calcoli. Collector.c/.h si occupa di connettersi tramite la socket farm.sck al MasterThread e ricevere i risultati calcolati dai workerthreads. Main.c si occupa della lettura dell'input. Util.h continente funzioni di uso generale.

WorkerPool:

"WorkerPool.c" e "WorkerPool.h" sono i due file che riguardano la threadpool e il task svolto dai threads. Il task consiste nel leggere una serie di valori **long** maggiori di zero da un file passato come stringa alla funzione "*leggieSomma*" (thread task vero e proprio), calcolare la somma di ciascun valore moltiplicato per la propria 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. Converte il valore void*arg ((leggieeSomma_arg*)arg.path)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, acquisisco la lock sulla socket invio la stringa e aspetto che il Collector risponda dopo la scrittura per rilasciare la lock sulla connessione.
4.Libero la memoria e return.
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 del 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 l'array di workers, inizializza i parametri come il numero massimo di thread, la mutex sulla coda/threadpool e sulla socket, 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, ritorna un *workerpool_t** in caso di successo. I threads creati non vengono inizializzati direttamente con il task *leggieSomma* ma vengono inizializzati con la "generica" funzione "***wpoolWorker***" a cui viene passata la threadpool. La funzione wpoolWorker consiste nel: acquisire la mutex sulla coda per rimuovere un task ed

eseguirlo. Dopo aver acquisito la lock la prima cosa che fa è controllare che la coda dei task non sia vuota altrimenti aspetta un segnale(pthread_cond_wait() in un ciclo while(condizione)per evitare wake up spuri). Se la threadpool fosse in fase di uscita si controlla se ci sono ancora task pendenti ((wpool->exiting == true) && (wpool->pendingQueueCount == 0)), se la condizione è true il thread termina. Altrimenti il thread prende un task dalla coda (FIFO), decrementa la dimensione della dei task da eseguire, incrementa gli active task riposiziona correttamente la testa controllando che non sfori la dimensione della coda stessa altrimenti la riporta a wpool->queueHead=0. Dopo aver sistemato la testa rilascia la lock, il thread esegue leggieSomma sugli argomenti messi in coda, riacquisisce la lock, decrementa le active task e se ci sono ancora task e non si è in stato di uscita 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 due modalità: uscita forzata, uscita con attesa dei task. L'uscita forzata distrugge la threadpool senza preoccuparsi di aspettare i threads. L'uscita con waitTask=true fa la join su tutti i threads della threadpool dopo aver lanciato un segnale in broadcast per svegliarli tutti. Dopo la fase di waitTask viene chiamata la freeWorkPool() che libera le risorse allocate, distrugge le mutex e le cond. Infine abbiamo la funzione **addTask**() che permette di aggiungere una task alla queue. Alla addTask viene passata la threadpool e una stringa (leggieSomma_arg, già castato a void*) che contiene il nome/path del file. Dopo aver controllato che gli argomenti passati siano validi, la addTask controlla che la coda non sia piena o in stato di uscita, altrimenti ritorna uno. Poi esegue la lock sulla coda e inserisce il task in posizione wpoool->queueTail, assicurandosi che il puntatore di fine coda non sfori la dimensione massima altrimenti lo riporta a zero. Viene poi lanciata una signal e rilasciata la lock sulla coda. AddTask ritorna tre valori: zero se è andato tutto a buon fine, uno in caso in cui la coda dei task sia piena o la threadpool è in fase di uscita e non accetta più task, valore minore di zero se ci sono stati degli errori.

MasterThread:

Il processo MasterThread è il core del progetto. Implementa il signal handler che riceve e gestisce i segnali SIGINT, SIGTERM, SIGHUP, SIGUSR1, alla ricezione di uno dei tre segnali di terminazione il signal handler controlla che il valore di int*stop (condizione del while della *runMasterThread()*) sia ==1 in questo caso non è stato ricevuto un segnale esterno, ma il signal handler viene chiuso con *pthread_kill(signal_handler_id, SIG***), altrimenti viene inviato un messaggio al Collector composto da un solo carattere ("t") sulla socket che è salvata nella threadpool. Nel caso di ricezione di SIGUSR1 viene mandata al Collector una stringa di due caratteri. Dopo l'invio di entrambi i messaggi viene cambiato il valore della variabile globale "flag": uno per terminazione due per stampa. Nel caso di segnale di terminazione viene rilasciata la lock e viene cambiato il valore di int*stop da zero a uno. La funzione **runMasterThread()** è il cuore di tutte le funzionalità del MasterThread, si occupa innanzitutto di creare una socket che si metterà in ascolto della richiesta di connessione del Collector, successivamente esegue la fork() assicurandosi che vada a buon fine. Nel processo figlio (pid==0) viene fatto partire il Collector (*runCollector()*). Nel processo padre viene invece accettata la connessione con il Collector, creata la threadpool a cui vengono passati i parametri letti tramite getopt() dal main e la collectorSocket. Inseguito viene avviato il signal_handler a cui viene passata la struct sigHarg che contiene un int*, sigset_t*, workerpool_t*. Prima di entrare nel while viene creato un array di lerggieSomma_arg (struct di argomenti da passare alla funzione *leggieSomma()*) che poi sarà passato, di indice in indice, alla coda della threadpool tramite la addTask. La condizione di iterazione del while si compone di due parti messe in AND, un int*stop che serve per fermare il ciclo quando

vengono messi in coda tutti i file passati al main o viene ricevuto un segnale di terminazione. La seconda condizione è il return di un *waitpid()* non bloccante. Se collectorTerminato==process_id (child_id) significa che il Collector è terminato, quindi è inutile che il MasterThread continui a lavorare. Le varianti di interruzione del ciclo while vengono gestite differentemente: se il ciclo è stato interrotto per la terminazione anticipata del Collector viene distrutta la threadpool, e viene chiuso il signal_handler, viene liberata la memoria e return EXIT_FAILURE. Nel caso di uscita con *stop =1 termino la threadpool aspettando che vengano elaborate tutti i tasks ancora presenti in coda termino il sig_handler, se la destroy non va a buon fine, libero la memoria e return EXIT_FAILURE, altrimenti return EXIT_SUCCESS, ma solo dopo aver eseguito una waitpid bloccante per aspettare che termini il Collector prima di eseguire la return.

Collector:

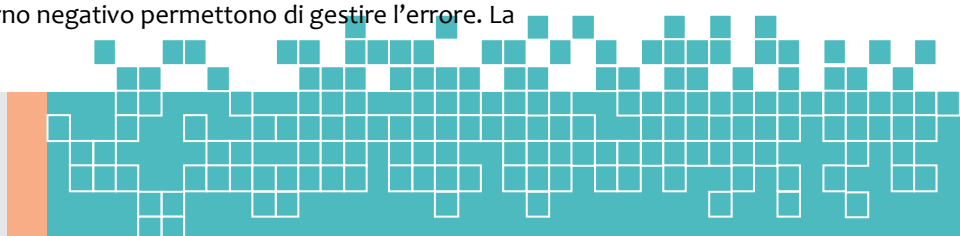
I file Collector.c/.h implementano le funzionalità del Collector, la funzione **runCollector()** è il "main". Alla funzione runCollector() viene passato il numero totale di file che devono essere elaborati, viene allocato uno string[] (= char*[]) per inserire i risultati calcolati dai workers, successivamente viene mandata la richiesta di *connect()* al MasterThread, se va a buon fine inizia il vero e proprio compito del Collector: viene allocato un buffer e si entra nel while(true). Ad ogni iterazione del while viene controllato se siano stati inseriti tutti i file altrimenti il Collector si mette in lettura della socket, controlla che il risultato della *read()* sia >zero, in quel caso controlla ciò che è stato letto: se la lunghezza è uno o due significa che ho ricevuto un segnale e quindi stampo o termino a seconda del segnale ricevuto, altrimenti salvo in dataArray (l'array di stringhe allocato all'inizio). La stampa,che deve avvenire in maniera ordinata, sfrutta il **qsort()** della <stdlib.h> per ordinare gli elementi del dataArray prima di stamparli, la funzione **compare()** passata converte gli "*n*" caratteri prima dello spazio di ogni stringa dell'array in long e fa il confronto su quale dei due sia il maggiore.

main.c:

Nel file main.c vengono eseguite le operazioni di controllo dell'input e di ricerca dei file che poi verranno passati al MasterThread. 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():* EXIT_SUCCESS se è andato tutto bene, EXIT_FALIRE altrimenti e si occupa di liberare 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. Contiene anche le macro che cambiano il valore della variabile globale flag nel MasterThread.

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** si 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.