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*” che impementano sostanzialmete tutte le funzionalità. 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. Il processo core di tutto il funzionamento è quello implementato da “*MasterThread.c*” e ”*MasterThread.h*” che svolge il compito di controllo dell’input, gestore della threadpool e ricettore dei segnali di terminazione e di stampa. 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 personalizzazione della threadpool come argomenti del main, viene controllato immediatamente che il numero di argomenti sia maggiore di zero, altrimenti errore, poi viene chiamata funzione *runMasterThread*  a cui vengono passati i parametri del main. *RunMasterThread* legge i parametri, eventualmente cerca nella cartella passata con -d, esgue la fork per invocare il processo Collector come figlio, crea e avvia il signal handler e la threadpool, passa i file e la task a ogni worker, attende la terminazione del Collector per terminare.

WorkerPool:

“WorkerPool.c” e “WorkerPool.h” sono i due file che riguardano la threadpool e la task svolta dai thread. La task riguarda leggere dei valori **long** maggiori di zero da un file passato come stringa alla funzione “*leggieSomma”*, calcolare la somma di tutti questi valori moltiplicati per la loro posizione i all’interno del file e comunicare tramite connessione socket AF\_UNIX il valore ottenuto e il nome (o percorso) del file al processo Collector. Nello specifico la funzione “*leggieSomma”* esegue i seguenti passi:  
1. Crea una connessione tra il worker e il collector all’indirizzo SOCKET\_NAME “./farm.sck” controllando che sia svolto tutto correttamente.  
2. Converte il valore void\*arg nella stringa filePath e apre una prima volta il file in modalità “rb”(lettura binaria) calcolandone la dimensione tramite il risultato della funzione “*getFileSize()*”:

long getFileSize(FILE \**file*) {

    long size;

    fseek(*file*, 0, SEEK\_END);

    size = ftell(*file*);

    fseek(*file*, 0, SEEK\_SET);

    return size;

}

Dividendo poi per sizeof(long) in modo da ottenere quanti numeri ci sono esattamente nel file, a questo punto viene allocato un 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 allocato dinamicamente, eseguo la somma alloco il buffer per scrivere e aspetto che la connect sia diversa da -1.  
4. Scrivo sulla socket path e valore del file assicurandomi che venga scritto ogni byte in una sola volta.  
Parliamo ora delle funzioni che gestiscono la threadpool dalla creazione, alla distruzione. Ho definito tre struct: “*workerthread\_t*”, ”*workertask\_t*” e ”*workerpool\_t*” che rappresentano rispettivamente un genrico thread apprtenetente alla pool, la generica task di un worker, la threadpool.

typedef struct workerpool\_t{

    pthread\_mutex\_t lock; *//mutext nell'accesso al'oggetto*

    pthread\_cond\_t  cond; *//usata per notificare un worker*

    workerthread\_t\*      workers; *//array di worker id*

    int             numWorkers; *//size dell'array workers*

workertask\_t\*   pendingQueue; *//coda interna usata per le task pendenti*

    int             queueSize; *//massima size della coda, se settata a -1 non ccetta task pendenti*

int             activeTask; *//numero di task che sono attualmente in esecuzione*

    int             queueHead; *//testa della coda*

    int             queueTail; *//fine della coda*

    int             pendingQueueCount; *//quantità di task pendenti presenti*

    bool            exiting; *//true segnala se viene iniziato il protocollo di uscita*

    bool            waitEndingTask; *//true quando si vuole aspettare che non ci siano più lavori in coda*

}workerpool\_t;

La funzione createWorkerpool alloca la coda delle task e inizializza i parmetri come il numero massimo di thread, la mutex sulla coda, le condizioni, gestisce gli errori nella creazione della threadpool stampando all’occorrenza cosa è andato storto e ritornando NULL in caso di insuccesso, associa ad ogni thread l’indice corrispondente nella threadpool. 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 pescare una task dalla coda ed eseguirla, dopo aver acquisito la mutex la prima cosa che fa è controllare se la pool non è in fase di uscita, altrimenti non vengono accettate nuove task, e controllare che la coda delle task non si vuota. Se la pool fosse in fase di uscita si controlla se è un’uscita forzata o un’uscita con attesa di terminazione delle task già inserite in coda, nel primo caso il thread ritorna immediatamente, nel secondo caso il thread pesca una task, anche se si è in fase di uscita, solo se ce ne sono ancora, in modo da finire le task in coda, altrimenti esce.  
La funzione