Progetto Farm Sistemi Operativi

Eva Fiori matricola 597631

e.fiori3@studenti.unipi.it

a.a 2022/2023

1. Introduzione

Il progetto è composto da più file descritti in dettaglio in seguito. Nella cartella compressa consegnata si trovano il Makefile, lo script test.sh e le tre cartelle documenti, headers e source. La cartella documenti contiene la presente relazione e il testo della consegna del progetto. In headers sono presenti tutti i file headers .h e per finire i file .c sono in source. Ad ogni file .c corrisponde un header eccetto per main.c e generafile.c. I file in questione sono collector.c, list.c, master.c, myLibrary.c, threadpool.c e tree.c.
Il codice è suddiviso nei vari file per separare i vari compiti. tree.c e list.c contengono rispettivamente le funzioni e gli struct relativi agli alberi binari e alle liste. collector.c, master.c e threadpool.c servono per separare le funzioni relative al processo Collector, al main thread e ai worker thread. Per finire il main.c, grazie alla fork, implementa entrambi i processi.
In seguito sono riportate le principali scelte implementative e una breve descrizione del funzionamento dei vari codici sorgente. Tutto il codice è dettagliatamente commentato per descrivere il comportamento di ogni funzione e l'utilizzo delle principali variabili.

2. Scelte implementative

Nell'implementazione dei due processi, per quanto riguarda la socket, MasterWorker copre il ruolo di client e Collector quello di server. La scelta è motivata dal fatto che la connessione socket lato client dovrebbe terminare prima di quella lato server. In questo modo il Collector può tranquillamente finire di leggere il contenuto della socket, anche se dall'altro lato sono terminate le scritture. In particolare nel flusso dell'esecuzione del MasterWorker, la chiusura del file descriptor della socket avviene in uscita dal programma, quindi normalmente dopo aver atteso la terminazione del Collector tramite waitpid. Questo non avvantaggia il MasterWorker nella chiusura, che potrebbe avvenire subito dopo il termine delle scritture, ma permette la chiusura anche in caso di terminazione precoce del processo dovuta a un errore di qualsiasi tipo che causa exit(EXIT_FAILURE).
La connessione alla socket utilizzata nel processo MasterWorker è soltanto una, sono però i worker thread a effettuare le scritture accedendo alla socket in mutua esclusione grazie a una mutex.
Infine scelgo di ignorare il segnale di SIGPIPE in entrambi i processi per evitare la terminazione causata da una scrittura sulla socket.

3. Descrizione del codice

1. Makefile

Nel Makefile sono implementati tre metodi principali: test, farm e clean.
test lancia l'esecuzione dei test verificando che la compilazione di generafile e farm sia aggiornata. farm compila l'intero progetto. clean rimuove qualsiasi prodotto dell'esecuzione dei due metodi precedenti, quindi file .dat e .txt generati, la directory dei test, gli eseguibili, i .o e un'eventuale farm.sck, utilizzando l'opzione -f che ignora eventuali file o directory non presenti. L'esecuzione del metodo all genera entrambi gli eseguibili.

2. main.c

Nel codice main come prima cosa vengono scansionati gli argomenti passati. Questo avviene tramite getopt e una lista dove vengono memorizzati gli argomenti non riconosciuti come opzioni, verificando che si tratti di file da elaborare. Leggendo il manuale, mi risulta che la getopt, una volta riconosciute le opzioni e i rispettivi argomenti, lasci l'array argv ordinato in modo che tutti gli argomenti non riconosciuti durante la scansione si trovino in fondo e il primo di questi sia indicato da optind. Nonostante vari test questa proprietà non mi è risultata valere utilizzando l'opzione di compilazione -std=c99, nemmeno includendo il file header getopt.h. Ho optato per isolare i potenziali file binari inserendoli in un'apposita lista, senza includere getopt.h perché non permette di manipolare la variabile optind.
Successivamente alla scansione degli argomenti viene chiamata la funzione mascheraSegnali() definita in master.c. La maschera è ereditata dopo la fork(), perciò varrà per entrambi i processi.
A questo punto viene eseguita la fork() e l'esecuzione del main si divide tra i due processi grazie ai rami dell'if.
Nel ramo Collector si prosegue con la chiamata di tre sole funzioni utili alla creazione della socket, la lettura dalla stessa e la stampa finale dell'albero. In atexit è aggiunta la deallocazione dell'albero.
Il ramo MasterWorker è leggermente più articolato. Per prima cosa lancia il thread gestore dei segnali. Successivamente chiama createThreadPool(), implementata e descritta in threadpool.c, passando come parametri numero di thread, lunghezza della coda e delay, eventualmente letti tra gli argomenti. Chiama creaSocketClient(), in master.c. Adesso si è pronti per svolgere task elaborando i file, quindi, se settata, si scansiona la directory mettendo nella coda i file all'interno, con scanDir() in master.c, e successivamente si fa lo stesso con la lista creata durante la lettura degli argomenti, attraverso la chiamata a pushFiles(), sempre in master.c. Terminate entrambe le funzioni non ci sono più task da elaborare, si setta il flag di terminazione e si svegliano eventuali thread in attesa. Si attendono i thread, poi il processo figlio ed infine il gestore dei file.
In main.c è definita solo la funzione svuotaFileBinari(), eseguita in atexit per deallocare la lista creata durante l'acquisizione degli argomenti.

3. collector

Sono implementate le funzioni riguardanti creazione, chiusura e unlink della socket, commentate dettagliatamente all'interno del codice. Le funzioni di chiusura e unlink sono messe in pila in atexit non appena creato un fd o la bind, in modo da essere deallocate ovunque venga provocata l'uscita dal programma.
Inoltre è implementata la funzione di lettura dalla socket che distingue i tre casi di terminazione, richiesta di stampa e scrittura di un messaggio completo di pathname e risultato del calcolo. Si occupa di chiedere la stampa dell'albero alla ricezione della richiesta e di mettere nell'albero i nodi per ogni messaggio letto, tramite opportune chiamate alle funzioni di tree.c. Ritorna al main.c quando riceve il messaggio di terminazione.
Il sockname `"./farm.sck"` è definito in collector.h.

4. tree

Implementa quattro funzioni relative agli alberi binari e contiene la dichiarazione globale dell'albero utilizzato. È necessaria una dichiarazione globale per l'utilizzo in atexit.
Sempre per la deallocazione in atexit sono presenti due funzioni, la prima senza parametri che chiama la seconda ricorsiva. Le altre due funzioni sono l'inserimento e la stampa.
Gli struct relativi al messaggio e al nodo dell'albero sono definiti in tree.h.

5. master

In master.c sono implementati tre gruppi principali di funzioni: per la gestione dei segnali, per la creazione e chiusura della socket lato client e per l'inserimento in coda dei task.
La gestione dei segnali prevede una funzione che maschera i segnali, mascheraSegnali(). Come detto in precedenza tale maschera viene ereditata anche dal processo Collector. La vera è propria gestione è eseguita dall'apposito thread implementato proprio in questo file sorgente, lanciato dal main.c. Il gestore si limita a settare le flag corrette alla ricezione di un segnale.
La creazione della socket è chiamata dal main ed è subito dopo la creazione del legame tra file descriptor e socket che viene messa in atexit la funzione di chiusura. Al termine della creazione c'è la chiamata alla funzione myConnect implementata in myLibrary.c. Questa funzione in particolare gestisce alcuni casi di fallimento della chiamata a connect(): ENOENT (il socket ancora non esiste), EAGAIN (la richiesta non può essere risolta immediatamente), EINTR (la chiamata è non bloccante e viene interrotta da un segnale), ECONNREFUSED (nessuno è in ascolto). L'ultimo caso citato può voler dire sia che sono stati riscontrati problemi nel collector, che semplicemente il collector non è ancora stato in esecuzione a sufficienza da arrivare a una accept(), per cui si ritenta la connessione altre due volte. In ogni caso di errore gestito si attende sempre mezzo secondo prima del tentativo successivo.
Per finire sono presenti le due funzioni che inseriscono i task in coda. scanDir esegue la scansione della directory, completa di creazione dei pathname, controllo su directory e file regolari e inserimento in coda tramite la chiamata alla push descritta in seguito. pushFiles() scorre la lista e chiama la push su ogni occorrenza.

6. threadpool

In threadpool.h è definito lo struct BQueue e il tipo relativo.
Nel file sorgente sono definite le funzioni pop e push utili rispettivamente all'estrazione e all'inserimento di task nella coda concorrente. La chiamata a pop() si assume che avvenga in possesso della lock, mentre la funzione push() si occupa anche dell'acquisizione.
La funzione risultato() elabora il file passato come argomento dall'apertura alla chiusura, calcolando la sommatoria richiesta.
La funzione worker_thread() che svolge l'intero lavoro del worker. Chiama pop() per estrarre il task e risultato() per elaborarlo. Per finire si occupa della scrittura dei messaggi sulla socket, compresi messaggi di stampa e terminazione, quest'ultimo scritto dall'ultimo thread prima della sua terminazione.
Sono inoltre presenti le funzioni relative a creazione del threadpool e cancellazione dello stesso. La createThreadPool() setta correttamente tutti i valori dello struct, allocando la memoria necessaria e inizializzando mutex e condition variable, e lancia i thread. Prima di ritornare al chiamante mette in atexit la funzione destroyThreadPool() che libera tutte le risorse dello struct del threadpool, eccetto per i thread che sono joinati dal main.

7. list

Implementa tre funzioni relative all'uso della lista: inserisciTesta(), stampaLista() e svuotaLista().
Lo struct relativo al nodo della lista è definito in list.h.

8. myLibrary

myLibrary.h implementa delle macro utili in tutti i file sorgente precedentemente descritti per il controllo sui valori di ritorno e la chiamata di funzioni usate di frequente, inoltre definisce `PATHNAME_MAX_DIM`.
myLibrary.c implementa delle funzioni utili all'interno del progetto, tra cui readn, writen e myConnect descritta in precedenza.

4. Test

Per la fase di testing del codice ho utilizzato i file generati da generafile.c tramite la chiamata di tesh.sh.
Per testare i segnali invece ho utilizzato un secondo terminale aperto dove ho eseguito "ps aux | grep farm" per conoscere il pid dei processi e "kill -<signal> <pid>" per mandare i segnali. I segnali che ho utilizzato sono:
- i segnali gestiti: SIGHUP (1), SIGINT (2), SIGQUIT (3), SIGUSR1 (10) , SIGTERM(15)
- il segnale SIGPIPE (13) ignorato
- i segnali SIGKILL(9) e SIGSTOP(19) non mascherati per terminare l'esecuzione in modo non gestito