Relazione progetto Farm - SOL

Jacopo Cioni - MAT: 581651

Maggio 2023

Contents

Chapter 1

Introduzione

1.1 Informazioni di macchina e di compilazione

Il progetto Farm è stato implementato su macchina multi-core Linux Ubuntu 20.04 LTS. La compilazione del progetto viene fatta tramite *Makefile* utilizzando il comando **make**. Per eseguire il progetto è necessario utilizzare prima il comando **make generafile**. Per compilare il progetto ed avviare direttamente lo script bash di test basta utilizzare il comando **make test**.

È possibile trovare il progetto al seguente link:

`https://github.com/cionijacopo/Farm-SOL`

NOTA: per utilizzare correttamente il comando **make test** è necessario prima generare tutti i file di test utilizzando il comando **make generafile**.

1.2 Struttura e schema delle dipendenze

La struttura del progetto si divide in:

- **doc**: cartella contenente tutti i documenti di testo relativi al progetto
- **includes**: cartella contenente tutti gli header files del progetto
- **src**: cartella contenente tutti i file sorgente

Più precisamente, la cartella includes contiene tutti gli headers dei file .c presenti nella cartella src. Per comodità, ho deciso di non suddividere questi files in ulteriori sottocartelle. Oltre a contenere la firma di tutti i metodi, alcuni degli header files contengono la definizione dei valori di sistema utilizzati nel progetto. Troviamo la definizione della SOCKETNAME e della

MAX_BACK_LOG nel file *connection.h*, il MAX_PATH_LENGTH nel file *isregular.h* ed i valori MAX_ARGV_LENGTH, DEFAULT_NUM_THREADS, DEFAULT_DELAY_TIME e DEFAULT_QUEUE_LENGTH nel file *utils.h*. La descrizione dei file sorgente verrà fornita in seguito.

Complessivamente, possiamo riassumere tutte le dipendenze del progetto con questo schema:

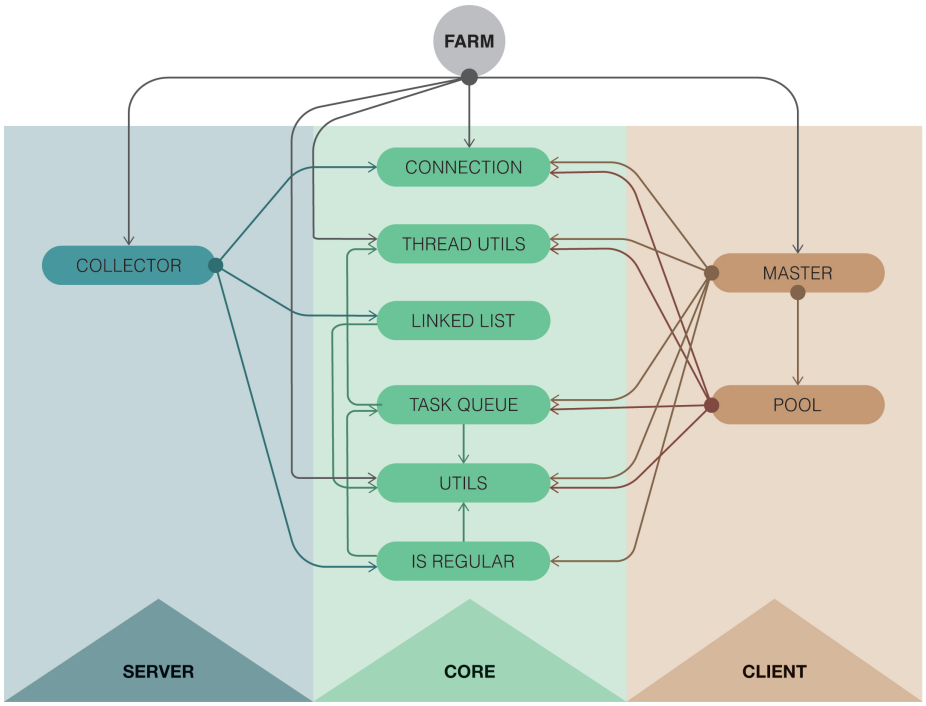

Figure 1.1: Dipendenze

Chapter 2

Sorgente

2.1 farmProject

Il file principale del progetto è **farmProject.c**. Questo file contiene il metodo main e si occupa di generare il processo **master** ed il processo **collector**. All'avvio, farmProject:

1. Esegue il parsing degli argomenti passati da terminale e controlla la loro correttezza. Nel caso in cui ci fosse un parametro errato, il programma sostituisce i valori scelti con quelli di default.
2. Maschera i segnali di interesse ed avvia un thread dedicato per gestirli. Il thread in questione eseguirà la funzione *sigHandler()* che resterà in attesa di segnali attraverso la *sigwait()*.
3. Ignora SIGPIPE per evitare terminazioni inaspettate.
4. Crea i processi *master* e *collector*.
5. Attende la terminazione del signalHandler thread.

2.2 Client

I file contenuti nella cartella *client* sono **farm_master_worker.c** e **pool.c**. Nel primo file si trova la funzione *farm_master()* che si occupa principalmente di inizializzare la coda concorrente dei task, di creare la threadpool e di aggiungere i file che sono regolari alla queue. La funzione utilizzata in questo passaggio è *pushPool()*, che è definita all'interno del file **task_queue.c** situato nella cartella *core*. La funzione *pushPool()* prende in input la coda safe ed il nome del file regolare e si occupa di inserirlo all'interno della coda concorrente. Concluso il lavoro degli worker viene attesa la terminazione dei threads della pool e vengono liberate le risorse precedentemente allocate.
Il file **pool.c** contiene la funzione eseguita dal generico thread worker e la funzione utilizzata per il calcolo dei risultati.

2.3 Server

Il file contenuto nella cartella *server* è **farm_collector.c**. Questo file agisce da master nella connessione con tutti gli workers e si occupa di creare la server socket dove andare a leggere tutti i messaggi. Dopo essere stati letti ed interpretati, i messaggi vengono salvati all'interno di una *linkedlist* ordinata che è definita all'interno dei files **linkedlist.c/.h** situati nelle cartelle *core* ed *includes*. La funzione *handler()* gestisce anche i messaggi speciali END e SIGNAL. Quando viene ricevuto il messaggio END allora il client ha terminato l'invio di tutti i file ed il server può passare alla fase di terminazione. Quando viene ricevuto il messaggio SIGNAL allora è stato sollevato il segnale SIGUSR1 ed il server provvede alla stampa di tutti i file ordinati e registrati fino a quel momento.

2.4 Core

Nella cartella *core* sono contenuti tutti i file sorgente utili al funzionamento del progetto. Nello specifico:

- **connection.c**: contiene le funzioni utili per la lettura *readn()* e la scrittura *writen()* dei dati sulla socket. In questo file si trovano anche le funzioni *clientSocket()*, contenente la **connect** ed utilizzata da tutti coloro che vogliono instaurare una connessione con il server, e *cleanup()*, utilizzata per l'unlink della SOCKETNAME.
- **isregular.c**: contiene le funzioni utili per determinare la regolarità dei file e delle cartelle. La funzione *naviga_cartella()* si occupa di esplorare cartelle e subcartelle alla ricerca di files regolari.
- **linkedlist.c**: contiene le funzioni utili a creare, stampare ed eliminare la lista ordinata utilizzata dal file collector per raccogliere i dati ricevuti. Tutte le funzioni sono ricorsive.
- **task_queue.c**: contiene tutte le funzioni utili alla gestione della coda concorrente dei task.
- **utils.c**: contiene le funzione safe per la malloc e la funzione *isNumber()*.

Chapter 3

Scelte di progettazione

La coda concorrente dei task da elaborare è stata implementata sulla base del problema del Produttore-Consumatore. In questo caso il produttore è il *thread master* presente nel processo **master**, mentre il consumatore è un thread worker generico della threadpool. Per garantire la concorrenza, la coda è stata implementata utilizzando un **mutex** per l'acquisizione della lock e due variabili **cond** per notificare se la coda fosse piena o vuota.
Nello specifico, la funzione di inserimento in coda si trova nel file **task_queue.c** ed è chiamata *pushPool()*. Questo metodo, prima di inserire il nome di un file nella coda, prende la *LOCK()* e controlla in un ciclo while che la coda non sia piena. In caso di risposta negativa il thread master si mette in attesa sulla condizione **pieno** e rilascia la lock. Quando sarà possibile inserire un nuovo elemento, il thread verrà risvegliato e riprenderà automaticamente la lock sull'oggetto. La funzione di rimozione dalla coda si trova nel file **task_queue.c** ed è chiamata *popPool()*. Il generico thread worker, prima di estrarre il nome di un file dalla coda, prende la *LOCK()* sulla coda e controlla in un ciclo while che la coda non sia vuota. In caso di risposta negativa il thread worker si mette in attesa sulla condizione **vuoto** e rilascia la lock. Quando sarà possibile estrarre un nuovo elemento, il thread verrà risvegliato e riprenderà automaticamente la lock sull'oggetto. Per semplicità, ho scelto di utilizzare una coda circolare.

La linked list utilizzata dal processo collector è definita nel file **linkedlist.c**. In questo caso ogni nodo è composto da un intero contenente il risultato calcolato dallo worker e da una stringa contenente il nome del file elaborato. Queste informazioni vengono ricevute dal collector concatenate l'una con l'altra. Viene utilizzata la tokenizer per separare le due stringhe dal carattere "-". Le funzioni di gestione della lista sono tutte ricorsive, compreso l'inserimento ordinato.

La memoria allocata per la coda concorrente viene liberata dal processo **master**, mentre la memoria allocata per la linked list viene liberata dal processo **collector**. Entrambe le *free()* vengono chiamate al termine della funzione principale situata in ogni processo.