

Progetto ”farm” - Appello 17/05/2023

Tiziano Calvani 579839

14 aprile 2023

1 Introduzione

Il progetto farm implementa lo schema di comunicazione tra processi e thread. Farm main genera dei thread Worker che comunicano con un altro processo denominato Collector che si occupa di raccogliere i messaggi, ordinare i risultati e stamparli sul canale standard output.

Il progetto è stato diviso in diversi file che in fase di compilazioni vengono linkati per creare un codice monolitico che si occupa di effettuare le operazioni richieste.

In particolare, abbiamo:

- generic: in questo file possiamo trovare funzioni utili che vengono usate da tutti i processi;
- queue_lib: dove è presente l'implementazione della coda concorrente, quindi la sua dichiarazione e funzioni che possono essere usate per gestirla;
- signal_handler: contiene tutte le funzioni utili per la gestione dei segnali e la funzione eseguita dal *Signal Thread*
- master_worker: si occupa della gestione del thread Pool e dell'inizializzazione della coda concorrente;
- worker: contiene le funzioni utili per la gestione dei thread concorrenti;
- collector: contiene la parte implementativa del processo che farà da server per la parte corrispondente alla connessione;

2 Struttura

2.1 Introduzione

La funzione main crea attraverso la *fork()* due processi che comunicano tra di loro attraverso delle socket.

La prima socket creata è quella del thread Signal, ovvero un thread che si occupa della gestione dei segnali. La ricezione dei segnali, attesa attraverso la funzione *sigwait()*, fa sì che per ogni segnale ricevuto ci sia un handler. Più in particolare per il segnale SIGUSR1 il thread Signal dopo aver creato una connessione attraverso la funzione *create_Socket()* manda un messaggio al Collector (la stringa ”SIGUSR1”) che fa sì che il Collector stampi tutti i file e i risultati momentaneamente ricevuti su *stdout* senza terminare.

La seconda socket viene creata dal Master Worker. Appena creata la connessione essa sarà passata tramite variabile globale (file descriptor client *fd_c*) ai vari Worker a cui accederanno in mutua esclusione.

2.2 Main

Il Main, contenuto in *farm_main.c*, per prima cosa gestisce, attraverso la funzione *getopt()* gli argomenti passati al programma, più in particolare memorizza i parametri che poi verranno utilizzati per l'intera esecuzione del programma.

Dopo aver memorizzato i vari parametri opzionali la funzione `Main` creerà una maschera per i segnali (`sigset_t mask`) in modo da usarla attraverso la funzione `pthread_sigmask` per bloccare quelli contenuti in `mask(SIGHUP, SIGINT, SIGQUIT, SIGTERM e SIGUSR1)`.

Dopo aver bloccato i segnali, il `Main` creerà un `Thread` che si occuperà dell'opportuna gestione (`signal_thread`).

Attraverso la funzione `fork()` si creerà un altro processo: il `Collector` sarà il figlio del processo padre `Main` che eseguirà la funzione `Master Worker`.

Dopo aver atteso la conclusione dei thread gestiti dalla funzione `master_worker`, il `main` si occupa della chiusura del thread adibito alla gestione dei segnali. La chiusura di questo thread avviene in maniera "forzata" attraverso la funzione `pthread_cancel()`, all'interno del thread però viene installata, all'inizio della sua esecuzione, una procedura di cleanup che si occupa di chiudere la connessione precedentemente stabilita con il processo `Collector`. Si attenderà l'avvenuta chiusura del thread attraverso la funzione `Pthread_join`.

2.3 Signal Thread

Il thread `Signal Thread` si occupa della ricezione dei segnali `mask(SIGHUP, SIGINT, SIGQUIT, SIGTERM e SIGUSR1)`.

Esegue la funzione `signal_thread_handler` contenuta nel file `signal_installer`.

Per prima cosa, il thread installa una funzione di cleanup che viene chiamata nel caso esso venga interrotto da una funzione `pthread_cancel()`. La funzione `cleanup_routine` chiude la connessione stabilita con il descrittore `fd_sig_thread_`. Descrittore usato per creare la socket utilizzata per comunicare la ricezione del segnale "`SIGUSR1`" al processo `Collector`.

Il thread, quindi, inizierà un ciclo infinito dove attenderà la ricezioni dei segnali `SIGHUP, SIGINT, SIGQUIT, SIGTERM e SIGUSR1` attraverso la funzione `sigwait()`.

Non appena verrà ricevuto un segnale, il codice del segnale verrà memorizzato nella variabile `sig` e, attraverso un controllo di questa variabile gestirà in maniera diversa i segnali arrivati.

Se il segnale arrivato è di tipo `SIGUSR1` allora il thread provvederà a notificare l'evento al processo `Collector` attraverso la connessione precedentemente stabilita. Altrimenti, se il segnale è di tipo `SIGHUP, SIGINT, SIGQUIT`, oppure `SIGTERM` esso setterà il flag `sig_term_received` a `true`.

Questo flag verrà utilizzato per impedire al processo `Master Worker` di inserire file nella coda concorrente, quindi alla ricezione di uno dei segnali elencati sopra, il processo smetterà di aggiungere files all'interno della struttura dati.

Il thread contiene un ciclo `while(1)` che verrà interrotto dal `Main` attraverso la funzione `pthread_cancel()`. In seguito alla chiamata della funzione, il thread eseguirà la funzione di cleanup che chiuderà la connessione al processo `Collector`.

2.4 Master Worker

Il processo `Master Worker` si occupa della gestione del thread Pool e dell'inizializzazione della coda concorrente.

La coda concorrente è una struttura dati strutturata in questo modo:

```
typedef struct {
    char **items;
    int front;
    int rear;
    int size;
    int done;
    pthread_mutex_t q_lock;
} _queue;
}
```

Per poter accedere alla struttura sarà necessario acquisire precedentemente la lock attraverso la funzione `Pthread_mutex_lock()`. L'accesso alla coda viene eseguito solitamente attraverso le funzioni `enqueue()` e `dequeue` che si occupano di controllare che la coda rispettivamente sia non piena e non vuota per poi mettere un item all'interno della coda oppure prelevarelo.

Il processo *Master Worker* dopo aver inizializzato la struttura dati concorrente crea una connessione con il processo *Collector* dove avverrà lo scambio di messaggi tra i due processi.

Successivamente, *Master Worker* crea un numero di thread, stabilita in precedenza (di default 1), che eseguono la funzione *worker()* che si occupa di prelevare dalla coda un file name attraverso la funzione *dequeue*, aprire il file, calcolare il risultato e inviare il risultato con in nome del file nel formato *12345;file_namej* al *Collector* tramite la connessione precedentemente stabilita.

Per poter mandare un messaggio un thread *worker* deve prima acquisire la lock relativa alla socket della connessione. In questa maniera, garantendo la mutua esclusione sull'accesso al canale di comunicazione i messaggi saranno consistenti.

Dopo aver terminato di inserire i nomi dei file attraverso la funzione *enqueue* il *Master Worker* setta il valore del flag *queue_idone* a 1, così da notificare ai *worker* che non trovano file.name da elaborare che sono finiti e che possono terminare.

In seguito, il processo attenderà la chiusura di tutti i thread *worker* attraverso la funzione *Pthread_join()*. Dopo aver atteso la terminazione di tutti i thread il processo manda un messaggio al *Collector*, attraverso la connessione precedentemente stabilita, per notificare la terminazione dell'invio di tutti i file da elaborare.

Infine, il *Master Worker* eseguirà le procedure per la terminazione: distruggerà la struttura *lock* utilizzata per l'accesso concorrente alla connessione, libererà la memoria allocata dinamicamente dalla struttura dati *queue* e rimuoverà il file usato per la connessione *farm.sck*.

2.5 Collector

La creazione dal processo *Collector* viene eseguita dal *Main* attraverso la *fork()*.

La gestione delle connessioni viene eseguita attraverso i canali, quindi il *Collector* è un processo single threaded che gestisce le varie connessioni attraverso il meccanismo della *select()*.

Il processo *Collector* si occupa di gestire i messaggi che arrivano dai vari *Client* in questo caso avremo due *Clients*.

Il primo *Client* è il *Signal Thread* mentre l'altro *Client* è il processo *Master Worker*.

Il processo *Collector* controlla ogni volta la ricezione dei messaggi che arrivano più in particolare controlla se sono arrivati messaggi di terminazioni o segnali. Infatti, se il *Signal Thread* manda il messaggio di avvenuta ricezione del segnale *SIGUSR1* il *Collector* manderà il messaggio di avvenuta ricezione del messaggio (*ACK*) e stamperà la lista dei file che sono arrivati in quel momento. La lista dei file sarà sempre ordinata poiché l'inserimento nella lista avviene in maniera ordinata. Se, invece, il *Collector* riceverà il messaggio di terminazione "*DONE*" da parte del processo *Master Worker* setterà il flag *stop* a 1 che farà terminare il ciclo e stamperà i risultati su *stdout*.

Il *Collector* quando riceve un messaggio manderà a sua volta un messaggio di acknowledge ("*ACK*") per confermare l'avvenuta ricezione del messaggio.

Infine, il processo *Collector* si occuperà della gestione della terminazione: chiuderà la *master_socket*, stamperà i risultati, libererà la memoria dinamica allocata per la lista e eseguirà l'unlink dal file *farm.sck*.