Gael Mouraret matricola : 564777

Progetto Farm

indice

1) Struttura cartelle
La directory del progetto contiene il makefile, il codice sorgente di generafile e lo script test.sh.
Il codice sorgente del progetto si trova nella directory *src*.
Nella directory *obj* invece vengono creati i file oggetto, e nella cartella *headers* sono contenuti tutti gli header files.
Le directory *dat_files** contengono alcuni file con estensione ".dat" per testare il programma.

2) Target makefile
Oltre al comando "make", che compila normalmente il progetto, creando gli eseguibili *farm* e *collector* nella directory principale, ci sono il comando "clean" che elimina tutti i file oggetto dalla cartella "obj", il comando "test" che lancia i test contenuti in "test.sh" e i comandi "debug_master", "debug_*collector*" e "debug_*collector*" che abilitano le stampe in stdout di informazioni sull'esecuzione di *farm* a runtime. L'ultimo comando è "debug_clear" che annulla l'effetto delle precedenti.

3) Files sorgenti
Il codice del progetto è suddiviso in diversi files sorgenti ".c".

- main.c :
Contiene il codice che viene eseguito lanciando l'eseguibile "farm", e quindi tutto il flusso di esecuzione principale.
controllo opts -> lancio processo *collector* -> lancio *workers* -> lancio *master* -> terminazione

- opts.c :
Contiene funzioni utili a fare il parsing delle opzioni di farm a riga di comando.

- utils.c :
Contiene funzioni di utilità varie, utilizzate in molti dei sorgenti del progetto.

- master.c :
Contiene il codice del thread master.

- worker.c :
Contiene il codice dei thread worker.

- *collector*.c :
Contiene il codice (singol thread) del server.

- *rdd.c, cqueue.c, fdtable.c, stringxlong.c* :
Contengono i metodi da utilizzare con le rispettive strutture dati.

4) Strutture dati

- r_dirdesc (rdd.h) :
La struttura vuole rappresentare una directory con path relativi.
char dir_rpath* rappresenta il path relativo della directory (nome della directory incluso).
*char** subdirs* contiene invece il nome di tutte le directory contenute in essa e size_t *subdirs_size* è il numero delle sottodirectory (quindi la dimensione di *subdirs*).
*char** files* contiene invece il nome di tutti i files della directory e size_t *files_size* è la sua size.

Questa struttura viene impigata in *opts.c* per gestire l'opzione "-d" nella funzione *master_directory_opt*, che prende come argomento il nome di una directory, e tramite quello viene costruito un vettore di r_dirdesc, contenenti la directory argomento e tutte le altre sottodirectory. Da questo poi vengono estratti tutti i path relativi dei files ".dat" che saranno usati da farm.

- cqueue :
Rappresenta una coda concorrente contenente i path dei files ".dat" usati da farm.
I campi sono :
*char** *items*, vettore dei path.
size_t *size*, dimensione del vettore dei path.
size_t max_size, massima dimensione della coda specificata con l'opzione "-q".
pthread_mutex_t mtx, pthread_cond_t empty_cond, pthread_cond_t full_cond sono rispettivamente il lock, la variabile di condizione di coda vuota e quella di coda piena.
unsigned int left_pool rappresenta il numero di thread che non operano più sulla coda (informazione utile al thread master per capire se non ci sono più workers al lavoro).

La coda è LIFO, ogni nuovo elemento viene aggiunto in coda, ed ogni elemento prelevato viene prelevato dalla coda.
L'inserimento dell'elemento di terminazione nella coda ignora la necessità di controllare che la coda sia piena.

-fd_entry (fdtable.h) :
Struttura che associa ad un file descriptor (int) un buffer per contenere i messaggi inviati da esso.

- fd_table :
Rappresenta una tavola che associa ad ogni file descriptor, un buffer contenente i messaggi inviati da esso.
Viene usato da *collector* per gestire i file descriptor dei client che vi si connettono ad esso. Così *collector* può memorizzare i messaggi (parziali) inviati dai client.
La struttura contiene un vettore di fd_entry, la sua dimensione e il valore massimo dei file descriptor all'interno del vettore.

- stringxlong :
Struttura che unisce una stringa (char*) e un long. Viene usata da *collector* per memorizzare l'output da stampare alla fine dell'esecuzione.

5) Gestione segnali
I segnali mascherati per *collector* sono SIGINT, SIGERM, SIGHUP, SIQUIT, SIGUSR1 e SIGPIPE.

Lato client :

main.c maschera i segnali SIGINT, SIGERM, SIGHUP, SIQUIT, e SIGUSR1, quindi tutti i thread worker e il master thread ereditano la maschera.

Il thread master controllerà i segnali mascherati una volta per ciclo di inserimento di elementi nella coda.
Per garantire che i segnali verranno controllati periodicamente, l'attesa del master thread sulla coda piena è fatta con pthread_mutex_timedwait.
Se si è usata l'opzione "-t", e quindi il thread master si sospende su nanosleep una volta per ciclo di inserimento, per controllare l'arrivo dei segnali questi vengono temporaneamente smascherati, e si gestiscono con un handler che incrementa una variabile volatile sig_atomic_t. Così facendo non si deve attendere tutta la durata della nanosleep prima di controllare il segnale.

E' comunque possibile che tra l'invio di un segnale (esempio SIGINT) e la sua gestione, trascorra qualche secondo, a casua della pthread_timedwait, che in ogni caso attenderà lo scadere del timeout prima di ritornare.

Per quanto riguarda gli worker, le write non controllano EINTR, perchè i segnali sono mascherati per esse.

SIGPIPE è mascherato per tutti i thread di *farm*.

Lato server :
collector eredita la maschera dei segnali di farm (main.c), e quindi deve creare una maschera opportuna per lui, ovvero smascherando SIGUSR1.

collector gestisce SIGUSR2 incrementando la variabile *volatile_sigatomic_t termination*, che notifica la richiesta di chiusura da parte di *master*. Quindi *collector* procede non accettando nuove connessioni dagli worker, e finendo di gestire quelle attualmente aperte.

Alla ricezione di SIGUSR1 invece, viene incrementato un altra variabile *volatie_sigatomic_t print_output_request*, che, al prossimo ciclo di *select*, richiederà la stampa dell'output parziale.

La *read* effettuata per leggere i messaggi degli *worker* potrebbe essere interrota dai segnali, ma non è un problema in quanto alla successiva *select*, sarà notificata nuovamente la presenza di bytes da leggere dai rispettivi file descriptors.

La *select* può essere interrotta dai segnali. Quindi si procede a controllare che l'errore sia EINTR e che i segnali siano stati gestiti opportunamente, quindi si ricomincia il ciclo di *select*.

La *select* è impostata con timeout.

SIGPIPE è gestito controllando il valore di ritorno della *read* e *errno*. Se *read* ritorna EPIPE, allora si gestisce come una chiusura della comunicazione da parte del client.

6) Concorrenza
La parte *collector* è single thread.
La parte *master worker* è multi thread.
I thread *worker* ed il (main) thread *master* comunicano usando una coda concorrente (veci cqueue.h).

La coda implementa un singolo *lock* e due variabili di condizione (una per coda piena e uno per coda vuota).

Se un thread *worker* fallisce, notifica incrementando (in mutua esclusione) la variabile interna alla coda *left_pool*, e ritorna (void*) 1.

Il thread *master* controlla ad ogni iterazione che il valore di *left_pool* sia minore della dimensione iniziale del pool, altrimenti significa che per qualche motivo tutti i thread sono terminati con fallimento, e procede quindi a terminare l'esecuzione (il controllo è assicurato poichè *master* non si blocca mai indefinitamente su *cqueue_append*, che usa la *pthread_cond_timedwait*).

Al termine dell'esecuzione, vengono inseriti nella coda gli elementi di terminazione, ovvero elementi con stringa "\0". Questo notifica agli *worker* che possono smettere di attendere nuovi inserimenti da parte di *master* nella coda.
Questi inserimenti sono effettuati in **testa alla coda**, in contraddizione con il suo essere LIFO, e non viene controllata la dimensione massima della coda.

Se *master* fallisce per qualche motivo, allora notifica la terminazione a tutti gli *worker* inserendo l'elemento di terminazione **in coda a cqueue**, così che essi terminino il prima possibile, poi tenta la *join* con ogniuno di essi, e termina.

Se si incontra un'errore prima che siano stati lanciati i thread, (come ad esempio se c'è un errore nella *fork* che manda in esecuzione *collector*) allora viene inviato il segnale SIGINT al processo stesso, che verrà poi gestito dal thread *master*.

Il processo *collector*, se fallisce, non notifica *master worker*, ma semplicemente termina. Sia il thread *master* che gli *worker* si accorgeranno della terminazione di *collector* grazie alle *write* o alla *connect* e falliranno di conseguenza.

7) Gestione errori
La maggior parte degli errori è gestito con la terminazione di *farm*. Quando possibile (in situazioni single thread) gli errori sono notificati con stampe di messaggi espliciti sulla natura dell'errore con *perror*. In situazioni con molteplici thread, la stampa è diretta a *stderror* e specifica il nome della funzione che ha causato il fallimento.
In seguito a fallimenti di chiamate di sistema o errori gravi, il valore di ritorno delle successive funzioni viene testato al massimo per produrre stampe su *stderr*, poichè il comportamento intrapreso è già quello di uscita con fallimento.