UNIVERSITÀ DI PISA

Relazione Progetto SOL

Aprile Filippo
Corso A
580175

a.a 2022/23

Contents

1 Introduzione

Il codice sorgente per l'implementazione di ***farm*** è stato suddiviso nelle seguenti folder:

- **collector**, contenente il codice per l'implementazione del Colletor.
- **core**, contente il codice per l'implementazione delle funzionalità condivise da Master e Collector.
- **master**, contenente il codice per l'implementazione del Master.

Queste folder sono a loro volta suddivise in:

- **include**, contenente i file header utilizzati.
- **src**, contenente i file.c utilizzati.

Inoltre è presente un folder ***build***, contenente:

- **objs**, folder dove vengono salvati i file.o del progetto.
- **generafile.c**, file.c per la generazione dei file binari (fornito con il testo).
- **test.sh**, script bash per testare il progetto (fornito con il testo).
- **testSignal.sh**, script bash per il test dei segnali da gestire nel progetto. Per ogni segnale viene eseguita ***farm*** a cui vengono passati valori diversi per gli argomenti opzionali: -t, -n e -q, oltre a -d testdir/ e i file.dat. Dopo un secondo viene mandato il segnale da gestire e viene controllato il valore di exit: 0 (test passato), altimenti (test fallito).

Infine è presente un ***Makefile***, per la compilzazione e il testing del progetto.

1.1 Makefile

Il ***Makefile*** prevede i seguenti target:

- **farm**: compila l'intero progetto generando l'eseguibile ***farm***.
- **generaFile**: compila generafile.c generando l'eseguibile ***generafile***.
- **test**: esegue i test definiti in ***test.sh***.
- **testSignal**: esegue i test definiti in ***testSignal.sh***.
- **cleanProject**: rimuove tutti i file.o presenti in ***build/objs*** e rimuove l'eseguibile ***farm***.
- **cleanFile**: rimuove tutti i file.dat, ***testdir/*** e l'eseguibile ***generafile***.
- **cleanAll**: esegue ***make cleanProject*** e ***make cleanFile***.

2 Implementazione del progetto

Il codice per l'implementazione del Master e' suddiviso nei seguenti file presenti in ***master/src***:

- **master_config.c**: si occupa della lettura degli argomenti passati da linea di comando.
- **master_farm.c**: crea la threadPool, setta la nuova gestione dei segnali, esegue la produzione dei task e aspetta la terminazione della threadPool.
- **master_task_producer.c**: definisce i metodi per la produzione dei task.
- **master_task_handler.c**: definisce le lock, variabili di condizione e metodi per l'uso della coda concorrente.
- **master_task_worker.c**: definisce il comportamento dei thread worker, i quali hanno il compito di elaborare il risultato dei file binari.

Per quanto riguarda le funzionalità condivise tra Master e Collector, il codice per l'implementazione si trova nei file presenti in ***core/src***:

- **queue_utils.c**: definisce i metodi per la creazione e uso di un coda generica (void*).
- **signal_handler.c**: definisce le varibile globali e i metodi per la gestione dei segnali tramite sigaction.
- **socket_handler.c**: definisce i metodi per la creazione e la gestione della socket AF_UNIX.
- **utils_.c**: definisce macro e metodi per gestione degli errori, controlli, readn e writen.

Viene inoltre utilizzato il ***pthread_utils.h*** contenente macro per gestione degli errori a seguito di: lock, unlock, signal, wait, join e broadcast.

Per l'implemtazione del Collector è stato usato solamente il file **collector.c** presente in ***collector/src***.

Per ognuno di questi file è presente un file header in: ***master/include*** (Master), ***core/include*** (Core), ***collector/include*** (Collector), che contiene le signature dei metodi, macro e eventuali varibili globali da essi definiti in modo da poter essere utilizzati dagli altri file.c.

Il file **master.c** è il **main.c** del Progetto.

2.1 Gestione degli argomenti

Gli argomenti possono essere:

- **opzionali**: letti mediante utilizzo di getopt(), salvati in variabili globali definite extern nel **master_config.h** e inizializzati ai valori di default in **master_config.c**. Vengono effettuati i seguenti controlli:

  -q, -n, viene controllato che l' argomento sia un numero > 0.

  -t, viene controllato che l' argomento sia un numero ≥ 0.

  -d, viene controllato che l'argomento sia il path di una dir.

- **file binari**: vengono controllati nel **master_producer.c**, in caso di file non binario oppure file non esistente, il file viene ignorato. Lo stesso avviene per i file contenuti nella folder passata come argomento dell' opzione-d. Il controllo viene fatto nel **master_producer.c** al fine di ridurre la complessita' in tempo/spazio iterando una sola volta gli argomenti e non creando altre stutture in memoria per la lore gestione oltre la coda concorrente.

Se non viene passata l'opzione -d e non viene passato nessun file binario come argomento oppure i controllo falliscono, il programma termina con fallimento.

2.2 Calcolo risultati dei file binari

Per il calcolo dei risultati dei file binari viene creata una threadPool e viene utilizzata una coda concorrente (politica LIFO) definita in ***master_task_handler.c***, implementado il paradigma del **produttore/consumatore**. La coda concorrente viene inzializzata prima della creazione della threadPool, con il flag settato a 1 (aperta), in modo che i thread worker si mettino in attesa del producer sulla **condition variable empty**.

2.2.1 Producer

Il main thead del processo Master si occupa di:

- *iterare la folder dell' opzione -d (in caso venga passata)*,
- *eseguire la push dei path relativi dei file binari presenti e di quelli passati come argomento sulla coda concorrente*,
- *chiude la coda concorrente per sengalare la fine della produzione.*

Per eseguire le **push** sulla coda concorrente:

- *aquisisce la lock sulla coda concorrente*,
- *itera finché la coda risulta piena, eseguendo una **wait** sulla **condition variable full** della lock (definita in **master_task_handler.c**), in attesa che un worker esegua una pop sulla coda concorrente*,
- *esegue la **push** del path relativo del file sulla coda, effettua una **signal** sulla **condition variable empty** della lock (definita in **master_task_handler.c**) per risvegliare i thread worker in attesa e rilascia la lock sulla coda concorrente.*

Conclude con la **close** della coda concorrente:

- *setta il flag della coda a 0 (chiusa) (aquisendone e rilasciandone la lock)*,
- *esegue una broadcast sulla condition variable empty per segnalare thread worker in attesa di terminare (aquisendone e rilasciandone la lock).*

2.2.2 Worker

I thread worker si occupano di:

- *eseguire la pop dei path relaviti dei file dalla coda concorrente finche' non riceve NULL*,
- *calcolarne il risultato*,
- *inviare il path relativo dei file e i loro risultati al processo Collector.*

Per eseguire le **pop** dalla coda concorrente, il thread Worker:

- *aquisisce la lock sulla coda concorrente*,
- *itera finche' la coda risulta vuota e aperta, eseguendo una **wait** sulla **condition variable empty** della lock (definita in **master_task_handler.c**), in attesa che il producer esegua una push sulla coda concorrente*,
- *controlla lunghezza e stato della coda concorrente (aperta/chiusa):*

  ***coda concorrente vuota e chiusa**, rilascia la lock sulla coda e restituisce NULL.*

  ***altrimenti**, esegue la **pop** del path relativo del file dalla coda, effettua una **signal** sulla **condition variable full** della lock (definita in **master_task_handler.c**) per risvegliare il producer in attesa e rilascia la lock sulla coda concorrente.*

2.3 Comunicazione con il processo Collector

Per la comunicazione tra il processo Master e il processo Collector, entrambi creano **una socket AF_UNIX**.

Il processo Master fa da client della socket, connettendosi e inviando i risultati tramite i thread Worker mediante 2 scritture per ogni file: con la prima viene scritta la size del buffer e con la seconda viene scritto il buffer, così strutturato:

- **risultato del file**, long (8 bytes),
- **lunghezza path relativo del file**, size_t (8 bytes),
- **path relativo del file**, lunghezza path relativo del file bytes.

Per la gestione della concorrenza viene usata una lock sula socket, la quale viene acquisita precedentemente alla prima scrittura e rilasciata dopo la seconda. Una volta che il processo Master ha terminato chiude la socket.

Il processo Collector assume il ruolo di master della socket, mettondosi in attesa della connessione da parte del processo Master. Una volta avvenuta la connessione legge i risultati inviati dai thread Worker di quest'ultimo, li salva in una struct Data definita in ***collector.h*** e li inserisce in un coda in modo ordinato. La lettura dei risultati termina quando la read restituisce 0, ovvero il processo Master ha chiuso la socket (nessun scrittore sulla connessione). Terminata la lettura dei risultati il Collector chiude la socket, chiude il descrittore del processo Master, rimuove la socket tramite l'unlink e stampa i path relativi dei file ed i loro risultati in modo ordinato.

Le *scritture/letture sulla socket* vengono effettuate utilizzando le funioni **readn/writen** fornite a lezione al fine di evitare *scritture/letture parziali.*

2.4 Gestione dei segnali

I segnali: SIGHUP, SIGTERM, SIGQUIT, SIGINT, SIGPIPE, SIGUSR1 vengono bloccati prima della creazione del processo Collector e della threadPool in modo che, per ereditarieta, rimangano bloccati in entrambi. Una volta creato il processo Collector e la threadPool, il main thread del processo Master ridefinisce la funzione di handler per ognuno di essi tramite sigaction e, in seguito, li sblocca (per far si che vengano gestiti unicamente da lui). Le funzioni di handler vengono ridefinite nel seguente modo:

- **SIGTERM, SIGQUIT, SIGINT, SIGHUP, SIGUSR1**, setta la variabile *volatile sig_atomic_t* signalFlag, definita in ***signal_handler.c***, al valore del segnale ricevuto.
- **SIGPIPE**, ignorato.

Il controllo della ricezione dei segnali viene fatto dal main thread del processo Master ogni volta che deve eseguire una push sulla coda concorrente. Vi sono 3 casi:

- **nessun segnale ricevuto (signalFlag = 0)** , viene eseguita la **push** sulla coda concorrente.
- **segnale SIGUSR1 ricevuto (signalFlag = SIGUSR1)** , viene fatta un scrittura sulla socket di "STAMPA" dal main thread, resettato il signalFlag a 0 e fatta la **push** sulla coda concorrente. Ogni volta che il processo Collector legge un risultato dalla socket controlla se ha ricevuto il messaggio "STAMPA" e in caso stampa i path relativi dei file e il loro risultati fino a quel momento ricevuti in modo ordinato.
- **segnale di terminazione ricevuto (signalFlag = SIGTERM, SIGQUIT, SIGINT, SIGHUP)** , viene eseguita la **close** della coda concorrente, e si ha la terminazione safe del programma.

La ricezione dei segnali comporta la terminazione di un eventuale sleep da parte del main thread. Per far si che una volta ricevuto un SIGUSR1, il segnale venga gestito correttamente e venga completata la sleep rimanente, il main thread itera fino a che la sleep rimanete non è = 0. In caso la sleep venga interrotta, controlla il segnale ricevuto:

- **segnale SIGUSR1 ricevuto (signalFlag = SIGUSR1)** , esegue un scrittura sulla socket di "STAMPA" e resetta il signalFlag a 0.
- **segnale di terminazione ricevuto (signalFlag = SIGTERM, SIGQUIT, SIGINT, SIGHUP)** , viene eseguita la **close** della coda concorrente, e si ha la terminazione safe del programma.
- **infine setta la sleep rimanente al nuovo valore.**

È stata preferita la gestione dei segnali tramite sigaction rispetto a quella tramite sigwait, in quanto quest'ultima sembra confarsi maggiormente alle situazioni in cui si devono gestire i segnali di un server(o un qualsiasi programma che debba loopare "indeterminatamente" soddisfando richieste) prevedendo l'uso di un thread apposito per la loro gestione, che in questo caso andrebbe terminato esplicitamente una volta che il processo Master completa la produzione dei task. Inoltre e' stato deciso di bloccare i segnali prima della creazione del processo Collector e della threadPool in modo da sfruttare l'ereditarieta' della signal mask al costo di ignorarli fino al loro sblocco, a seguito del setup dei loro signal handler.