Relazione Farm

Simone Passèra

I. Introduzione

Questo documento descrive in maniera generale il funzionamento e la struttura complessiva di Farm, un programma C che prende come argomenti una lista di file binari contenenti interi lunghi e per ognuno di essi, effettua un calcolo sugli elementi letti e stampa i risultati ottenuti ordinati in modo crescente. Per un approfondimento maggiore si consiglia la visione dei file sorgenti.

II. Architettura

MasterWorker

Processo multi-threaded composto da un thread Master (main thread), da un numero variabile di thread worker e da un thread che gestisce l'eventuale arrivo del segnale di stampa (SIGUSR1).

Principali operazioni:

- Maschera i segnali di uscita (SIGHUP, SIGINT, SIGQUIT, SIGTERM) ed il segnale di stampa (SIGUSR1).
- Installa l'handler per i segnali di uscita ed ignora il segnale SIGPIPE.
- Crea un socket di tipo AF_UNIX in grado di accettare richieste di connessione.
- Tramite la chiamata di sistema *fork* genera il processo Collector, il quale eseguirà la funzione *exec_collector*, e si mette in attesa della richiesta di connessione da parte del figlio sul socket creato precedentemente.
- Analizza gli argomenti dalla riga di comando. Controlla la correttezza di eventuali opzioni e si aspetta di ricevere nomi di file binari. Considera un file con un formato invalido se non è regolare oppure se la dimensione non è un multiplo dello spazio occupato da un intero lungo.
- Se l'utente ha specificato il nome di una directory tramite l'opzione d, viene eseguita la funzione *read_dir* che naviga ricorsivamente la directory e le directory al suo interno alla ricerca di file validi.
- I file passati come argomento e quelli presenti eventualmente nella directory sono salvati all'interno della struttura dati coda requests.

- Il numero totale di file che deve essere elaborato dal processo MasterWorker viene notificato al processo Collector, il quale può inizializzare la struttura dati al fine di contenere i vari risultati.
- Inizializza una variabile di mutua esclusione per sincronizzare la comunicazione con il processo Collector. Questa variabile è necessaria perché i vari thread scrivono sull' unico endpoint di connessione attivo con il Collector.
- Crea il thread pool attraverso la chiamata di libreria *initThreadPool*.
- Crea il Signal Handler Thread che gestisce il segnale di stampa (SIGUSR1).
- Sblocca i segnali di uscita precedentemente mascherati.
- Preleva dalla coda precedente requests un nome di file e lo invia al thread pool per essere elaborato. Questa operazione viene ripetuta fino ad esaurire la coda se precedentemente non avviene alcuna richiesta di terminazione, effettuata dall'utente inviando uno dei possibili segnali. Tra l'invio di due richieste successive al thread pool, il thread Master si sospende per un tempo in millisecondi specificato dall'utente. La sospensione è implementata attraverso la chiamata di sistema *nanosleep*.
- Avvia il processo di terminazione del thread pool attraverso la chiamata di libreria *shutdownThreadPool*.
- Attende la terminazione del Signal Handler Thread.
- Notifica al processo Collector che può iniziare la procedura di terminazione, inviando il codice 0.
- Termina il processo.

Collector

Processo single-threaded generato dal processo Master. Eredita la gestione dei segnali dal padre, esegue la funzione *exec_collector* con i segnali di terminazione e di stampa bloccati ed ignora il segnale SIGPIPE. Attende di ricevere tutti i risultati dai thread Worker ed al termine stampa i valori ottenuti in ordine crescente. La stampa può essere richiesta prima della terminazione se l'utente invia il segnale di stampa al processo Master.

Principali operazioni della chiamata *exec_collector*:

- Crea un socket di tipo AF_UNIX al quale connettere il processo Master.
- Si connette al processo Master tramite il file socket 'Farm.sck' all'interno della directory corrente creato dal processo Master.
- Attende di ricevere il numero massimo di risultati che gli saranno inviati.
- Alloca un array della lunghezza appena letta contenente puntatori ad una struttura capace di contenere il nome di un file ed il relativo risultato.
- All'interno di un ciclo infinito, il processo legge un numero intero. Se legge 0 il processo stampa i risultati contenuti nell' array ordinandoli con la chiamata di libreria *qsort*, e termina il processo. Se legge -1 stampa i risultati ricevuti fino a

quel momento sempre il modo ordinato. Altrimenti il numero intero indica la lunghezza in byte della stringa relativa ad un nome di file. Dunque il processo esegue due letture, una relativa al nome di file, conoscendo la lunghezza in byte ed una per il risultato associato che sappiamo essere un intero lungo. Memorizza file e risultato nella struttura precedentemente descritta e la aggiunge in fondo all'array. Il processo ripete il ciclo fino alla corretta terminazione data dal codice 0.

Thread pool

Oggetto software che permette di semplificare l' utilizzo dei thread Worker all' interno del processo Master. Il thread pool viene gestito dal thread Master attraverso le seguenti funzioni:

initThreadPool: Crea il thread pool con un numero di thread Worker pari a 4 se l' utente non fornisce l' opzione n da riga di comando. Inizializza una coda concorrente interna, la cui dimensione è sempre possibile modificare dall' utente. La coda viene utilizzata dai thread Worker per estrarre un nome di file da elaborare.

shutdownThreadPool: Elimina il thread pool attendendo la terminazione dei thread Worker e cancellando le strutture dati utilizzate. La terminazione dei thread Worker è implementata inserendo all' interno della coda concorrente valori NULL tanti quanti sono i Worker. Quando un thread Worker legge il valore NULL dalla coda termina. In questo modo quando viene invocata la chiamata *shutdownThreadPool* i file che si trovano ancora all' interno della coda concorrente, saranno processati dai vari Worker prima di terminare.

submitThreadPool: Inserisce all' interno della coda concorrente il nome di file passato come argomento, per essere elaborato da uno dei thread Worker. Il thread Master si può sospendere a causa delle variabili di sincronizzazione interne alla coda concorrente, in attesa di inserire il file.

Principali operazioni del thread Worker:

- Salva localmente gli argomenti, composti, dall' identificatore del thread, dal socket relativo alla connessione con il Collector ed associata la variabile di mutua esclusione, dalla coda concorrente da cui estrarre i nomi di file.
- All' interno di un loop, estrae dalla coda concorrente un nome di file. Se il valore è NULL termina con successo. Altrimenti apre e legge il file utilizzando le chiamate della libreria *libc*, calcolando il risultato che dipende dagli interi lunghi contenuti nel file. Dopo aver preso la Lock sulla variabile di mutua esclusione, invia al Collector il file con il relativo risultato. Continua ad estrarre nomi di file da elaborare in attesa di leggere il valore di terminazione.

Signal Handler

Funzione che viene eseguita dal sistema operativo se uno dei segnali di terminazione diventa pendente. La funzione semplicemente imposta ad 1 la variabile *sigexit* per notificare al Signal Handler Thread ed al thread Master di iniziare le operazioni di terminazione. La funzione viene eseguita dal thread che non ha i segnali mascherati ovvero il thread Master. In questo modo se il thread era sospeso sulla chiamata *nanosleep* viene interrotto e può controllare se deve terminare.

Signal Handler Thread

Thread che gestiste il segnale di stampa SIGUSR1, creato dal thread Master. Esegue il codice con i segnali di terminazione e stampa mascherati. Prende come argomento la variabile di mutua esclusione per comunicare con il Collector.
Il thread si sospende, attraverso la chiamata di sistema *sigtimedwait* a cui è possibile passare un timeout inizializzato ad 1 millisecondo, in attesa che il segnale SIGUSR1 sia pendente. Se il segnale viene catturato la chiamata ritorna e prendendo la Lock sulla variabile di mutua esclusione, invia al processo Collector il valore -1 per stampare i risultati. Invece se il timeout scade la chiamata di sistema ritorna ed in entrambi i casi prima di sospendersi nuovamente in attesa, controlla se la variabile globale di terminazione è stata impostata. Questo permette al thread di uscire se l'utente invia un segnale di terminazione oppure se il thread Master vuole far terminare il thread impostando la variabile globale.

III. Compilazione

Per compilare il progetto, posizionarsi all' interno della directory principale dove è presente il Makefile ed eseguire il seguente comando:

```
make
```

L'eseguibile con nome farm sarà creato all' interno della directory **bin**.
I file oggetto sono creati nella directory **obj**.

Il progetto è conforme allo standard POSIX, ad eccezione della funzione *getopt* che utilizza l'implementazione fornita dall' estensione GNU.

Il Makefile fornisce i seguenti comandi per eliminare i file:

```
make clean      # elimina gli eseguibili nella directory bin
```

```
make cleanall   # elimina tutti i file e directory generati dal Makefile
```

IV. Test

All' interno della directory **test** è presente uno script che permette di verificare il corretto funzionamento del programma. Posizionarsi all' interno della directory principale dove è presente il Makefile ed eseguire il seguente comando:

```
make test
```

Lo script esegue 5 test e stampa l' esito per ognuno di essi su standard output. Lo script continua eseguendo una serie di esempi di utilizzo del programma farm, stampando su standard output ogni esempio nel seguente formato:

```
commento:
comando eseguito
stampa del comando

commento:
...
```