SISTEMI OPERATIVI - FARM2

Ahmad Shatti 578268

Marzo 2024

Indice

1 Introduzione e struttura del progetto

Farm2 si articola in due principali componenti: il processo **Collector** che agisce da server e il processo multi-threaded **MasterWorker** che funge da client. Il processo MasterWorker utilizza un threadpool per creare uno o più thread Worker, ciascuno dei quali esegue un calcolo sui file binari in ingresso e invia il risultato al processo Collector. Quest'ultimo si occupa di riordinare i risultati e della stampa in ordine crescente. Il progetto è strutturato intorno alla cartella **`src`**, che contiene il codice per l'implementazione di Farm2, e una cartella **`header`** che contiene i file di intestazione per ciascuno dei file di implementazione, ad eccezione di `main.c`. All'interno di `src`, sono presenti i seguenti file:

- **`main.c`**: gestisce gli argomenti da riga di comando, avvia il processo Collector e coordina l'esecuzione del MasterWorker. Quest'ultimo gestisce i segnali, crea un threadpool e inserisce i task al suo interno. Infine libera le risorse prima di terminare l'esecuzione del programma.
- **`master.c`**: inizializza la maschera dei segnali e crea una lista di file binari a partire dagli argomenti in input e da una directory specificata.
- **`list.c`**: gestisce una lista concatenata di file binari, con operazioni come l'inserimento di nuovi nodi in coda, la rimozione della testa della lista, la liberazione della memoria allocata per la lista e la stampa degli elementi.
- **`tpool.c`**: implementa un threadpool per gestire l'esecuzione parallela dei task all'interno. Le funzioni definite comprendono la creazione del threadpool, l'aggiunta di nuovi task alla coda del pool, l'esecuzione dei task da parte dei thread del pool, l'invio dei risultati dei task al processo Collector, la creazione di nuovi thread o la rimozione di un thread in risposta a segnali specifici e la deallocazione del threadpool alla chiusura del programma.

- **`tree.c`**: gestisce un albero binario in cui ogni nodo contiene una stringa e un numero associato al risultato dei task. Le operazioni comprendono l'inserimento di nodi nell'albero, la stampa dei nodi in ordine crescente e la liberazione della memoria allocata dell'albero.
- **`collector.c`** implementa un server che gestisce le connessioni dei client. Costruisce un albero binario in base ai dati ricevuti dal cliente e stampa l'albero dopo ogni secondo di esecuzione. Termina deallocando la memoria e chiudendo la socket.
- **`util.*`** fornisce varie funzioni di utilità utilizzate da MasterWorker e Collector, come la gestione delle operazioni di sincronizzazione.

Inoltre, è presente un **`makefile`** per automatizzare il processo di compilazione e l'esecuzione dei test tramite i file `test.sh` e `generafile.c`.

2 MasterWorker

Il processo MasterWorker è distribuito tra i file `main.c` e `master.c`. Nel primo sono definite 3 variabili globali:

- `pool`, rappresenta il thread pool utilizzato in Farm2.
- `stop`, inizializzata a 0, viene utilizzata per controllare lo stato del processo principale e determinare se deve terminare o continuare l'esecuzione. Quando `stop` viene impostato a 1 da un segnale di terminazione o durante la fase di chiusura del programma, il processo principale interrompe l'inserimento di nuovi task nel thread pool e avvia la procedura di pulizia delle risorse. Viene dichiarata `volatile sig_atomic_t` per garantire che la lettura e scrittura avvengono atomicamente.
- `usr2_rcv`, con la relativa lock `usr2_lock`, vengono utilizzate per gestire la ricezione del segnale SIGUSR2 da parte del gestore dei segnali, rappresentato dalla funzione **`sigHandler()`**. La `usr2_lock` è una mutex utilizzata per garantire la corretta sincronizzazione durante l'accesso e la modifica della variabile `usr2_rcv`. Quando viene ricevuto il segnale SIGUSR2, la variabile `usr2_rcv` viene impostata a 1 e l'effetto è che un thread worker all'interno della funzione **`worker_task()`** nel file `tpool.c` terminerà la sua esecuzione se il numero di thread all'interno del pool è superiore a 1, altrimenti si resetta a 0 il flag `usr2_rcv`.

Il MasterWorker gestisce i parametri forniti dalla riga di comando consentendo all'utente di definire opzioni come il numero di thread worker, la lunghezza della coda concorrente, la directory contenente i file binari da elaborare e il ritardo tra l'inserimento di due task consecutivi. Questo viene fatto attraverso la funzione **`getopt()`** che analizza le opzioni passate al programma. Successivamente, il MasterWorker setta la maschera dei segnali e crea il processo figlio Collector che si occupa di gestire le connessioni dei client worker. A questo punto il MasterWorker inizializza un thread per la gestione dei segnali tramite la funzione **`sigHandler()`**. Quest'ultima alla ricezione dei segnali di terminazione SIGHUP, SIGINT, SIGQUIT e SIGTERM imposta la variabile globale `stop` a 1, bloccando l'inserimento di nuovi task nel pool e mandando il programma in fase di uscita. Quando invece viene ricevuto il segnale SIGUSR1 viene chiamata la funzione **`createNewThread()`**, definita in `tpool.c`, che crea un nuovo thread nel pool. Se viene ricevuto il segnale SIGUSR2 un thread worker nel pool terminerà la sua esecuzione. In seguito il MasterWorker crea un thread pool ed esplora ricorsivamente i file passati da riga di comando e la directory se specificata, inserendoli in una lista. Ogni elemento di questa lista viene poi passato al thread pool eventualmente con un ritardo se specificato. Infine il programma attende la terminazione del processo Collector e pulisce le risorse prima di terminare l'esecuzione del programma.

2.1 ThreadPool

Tramite il file `tpool.c` si implementa un thread pool con le seguenti variabili:

- `lock`, garantisce la mutua esclusione nell'accesso al pool da parte dei vari thread.
- `full` e `empty` variabili di condizione per sincronizzare l'accesso alla coda del pool.
- `threads` e `numthread` rappresentano rispettivamente l'array di thread che costituiscono il pool e il numero attuale di thread nel pool.
- `queue, queue_max_size, queue_task, head` e `tail` che rappresentano rispettivamente la coda del pool, la dimensione massima della coda, il numero di task all'interno della coda e i riferimenti alla coda.
- `exiting`, flag per indicare se è iniziato il protocollo di uscita del thread pool.

La funzione **`createThreadPool()`** inizializza il thread pool allocando memoria per la struttura del pool, l'array dei thread e la coda dei task. Inoltre inizializza la lock, le variabili di condizione e crea i thread del pool. La funzione **`addTaskToThreadPool()`** aggiunge un nuovo task alla coda del thread pool a condizione che non sia iniziato il protocollo di uscita, risvegliando nel caso un thread in attesa che la coda non fosse vuota. Se la coda è piena, il thread attende finché non è disponibile spazio. Il cuore del pool è rappresentato dalla funzione **`worker_task()`** che viene eseguita da ogni thread del pool. Questa funzione, all'interno di un loop infinito, verifica che la coda non è vuota e il pool non è in fase di uscita (altrimenti attende), estrae un task dalla coda e lo esegue utilizzando **`execution_task()`**. Quest'ultima funzione apre il file passatogli in input, in sola lettura, ed esegue il seguente calcolo:

$$\texttt{result} = \sum_{i=0}^{N-1} (i \times file[i])$$

Dopo l'esecuzione del task, viene chiamata la funzione **`sendToCollector()`** per inviare il nome del file e il risultato del task al processo Collector. La funzione `sendToCollector()` crea una socket e invia i risultati tramite essa dopo aver verificato che la connessione con il server è stata stabilita. Se viene ricevuto il segnale SIGUSR2, viene controllato all'interno della funzione **`worker_task()`** che il numero di thread nel pool è superiore a 1 e in questo caso uno dei thread viene terminato. Quando invece il processo principale riceve il segnale SIGUSR1, viene chiamata la funzione **`createNewThread()`** che crea un nuovo thread e lo aggiunge al pool. Infine la funzione **`destroyThreadPool()`** imposta il flag di uscita `exiting` a 1, risveglia tutti i thread in attesa, invia un segnale al Collector per indicare la fine dell'esecuzione e attende che tutti i thread nel pool terminino. Successivamente dealloca la memoria utilizzata nel pool e scrive il numero di thread su un file di output tramite la funzione **`writeNumber()`**.

3 Collector

La funzione **`serverCollector()`** è il punto di ingresso del processo Collector. Inizia creando un socket per ascoltare le connessioni dei client. Successivamente entra in un loop in cui utilizza la funzione `select()` per gestire nuove connessioni in arrivo. Quando una nuova connessione viene accettata, il Collector riceve il nome del file e il risultato del calcolo del task dal client. Quindi inserisce questi dati in un albero binario e avvia il thread di stampa se non è ancora stato avviato. Se riceve il valore di terminazione *end* interrompe il thread di stampa e termina il processo. La funzione **`printTreePeriodically()`** eseguita dal thread incaricato della stampa periodica dell'albero dei risultati. Questa funzione riceve come argomento il puntatore dell'albero dei risultati e continua a stampare l'albero a intervalli regolari finché il processo non viene terminato. Una volta terminato il processo, il Collector stampa l'albero dei risultati una volta in modo finale, dealloca la memoria utilizzata per l'albero e chiude la socket.

4 File di supporto: list.c, tree.c e util.*

All'interno del progetto Farm2 vengono utilizzati i seguenti file di supporto:

- **list.c**, contiene l'implementazione per la gestione di una lista concatenata `BinList`. La funzione **insTail()** inserisce un nuovo nodo in coda alla lista, allocando la memoria necessaria per il nuovo nodo e copiando il valore `s` nel campo `file` del nodo. La funzione **removeHead()** rimuove il primo nodo della lista e restituisce un puntatore al secondo nodo, mentre **freeList()** dealloca la memoria occupata dalla lista. Infine **printList()** stampa i contenuti della lista a scopo di debug.
- **tree.c** contiene l'implementazione delle funzioni per la gestione di un albero binario `Node`. La funzione **insNode()** inserisce un nuovo nodo nell'albero, allocando la memoria necessaria per il nuovo nodo e copiando il valore `str` nel campo `file` del nodo. La funzione decide dove inserire il nuovo nodo basandosi sul valore `num` del nodo. Le funzioni **printTree()** e **freeTree()** rispettivamente stampa l'albero e dealloca la memoria occupata da esso.
- **util.h** e **util.c** contengono una serie di macro e funzioni per varie operazioni. Le seguenti macro definite in `util.h` se non riescono ad eseguire correttamente le operazioni di sincronizzazione, emettono un messaggio di errore e terminano il programma.
  - `LOCK()` e `UNLOCK()` acquisiscono e rilasciano il mutex specificato.
  - `WAIT()` sospende il thread corrente sulla variabile di condizione specificata.
  - `SIGNAL()` invia un segnale alla variabile di condizione specificata per risvegliare un thread in attesa
  - `BROADCAST()` invia un segnale broadcast alla variabile di condizione specificata per risvegliare tutti i thread in attesa.

  Altre macro definite in `util.h` sono: `MAXFILENAME` per settare la lunghezza massima del nome del file in input, `SOCKNAME` è il percorso del socket utilizzato per le comunicazioni tra il processo MasterWorker e Collector e `USAGE()` che stampa un messaggio per utilizzare il programma Farm2 correttamente. Il file `util.c` contiene l'implementazione di **check_number()** che verifica se la stringa in input rappresenta un numero intero valido e positivo; **check_file** controlla se un percorso è associato a un file regolare o a una directory; **millisecondSleep()** sospende l'esecuzione per un numero specificato di millisecondi. Le funzioni **readn()** e **writen()** leggono e scrivono esattamente un numero specificato di byte. Infine **cleanupLock()** dealloca un mutex se è stato inizializzato correttamente, mentre **cleanupThread()** termina un thread e ne dealloca le risorse.

5 Gestione dei segnali

La maschera dei segnali è configurata per catturare segnali specifici e indirizzare l'esecuzione del programma verso le funzioni di gestione dei segnali appropriate. Questa configurazione avviene tramite la funzione `setSignalMask()`, che crea e imposta la maschera dei segnali necessaria. Successivamente, la gestione effettiva dei segnali è affidata alla funzione `sigHandler()`, che avvia un thread dedicato per gestire i segnali ricevuti. La maschera dei segnali è ereditata dopo la chiamata a **fork()**, il che significa che sarà valida per entrambi i processi MasterWorker e Collector.

6 Gestione degli errori

Gli errori all'interno del progetto sono stati gestiti utilizzando la funzione **perror** per stampare messaggi di errore significativi seguiti dalla descrizione dell'errore ottenuta da **errno**. Inoltre quest'ultima variabile viene gestita manualmente in diverse parti del codice per garantire una gestione precisa degli errori. Quando si sono verificati errori critici che non potevano essere gestiti, il programma è stato terminato immediatamente utilizzando **exit(EXIT_FAILURE)**. Questo approccio ha impedito che il programma continuasse a eseguire operazioni non valide.

7 Makefile e Test

Tramite `makefile` viene compilato il codice sorgente di Farm2. Con il target `test` viene eseguito lo script di test `test.sh`. I test sono stati eseguiti sia sulla mia macchina:

```
ashatti@AShatti:~/Desktop/AhmadShatti_578268$ nproc
4
ashatti@AShatti:~/Desktop/AhmadShatti_578268$ uname -a
Linux AShatti 5.15.0-97-generic #107-Ubuntu SMP Wed Feb 7 13:26:48 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
ashatti@AShatti:~/Desktop/AhmadShatti_578268$ gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

ashatti@AShatti:~/Desktop/AhmadShatti_578268$ ld --version
GNU ld (GNU Binutils for Ubuntu) 2.38
Copyright (C) 2022 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or (at your option) a later version.
This program has absolutely no warranty.
ashatti@AShatti:~/Desktop/AhmadShatti_578268$ make test
running test suite...
./test.sh
test1 passed
test2 passed
test3 passed
test4 passed
test5 passed
test6 passed
```

e sia sulla VM Xubuntu:

```
xubuntu@xubuntu-VirtualBox:~/Scrivania/AhmadShatti_578268$ nproc
4
xubuntu@xubuntu-VirtualBox:~/Scrivania/AhmadShatti_578268$ uname -a
Linux xubuntu-VirtualBox 3.16.0-29-generic #39-Ubuntu SMP Mon Dec 15 22:27:29 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
xubuntu@xubuntu-VirtualBox:~/Scrivania/AhmadShatti_578268$ gcc --version
gcc (Ubuntu 4.9.1-16ubuntu6) 4.9.1
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

xubuntu@xubuntu-VirtualBox:~/Scrivania/AhmadShatti_578268$ ld --version
ld di GNU (GNU Binutils for Ubuntu) 2.24.90.20141014
Copyright (C) 2014 Free Software Foundation, Inc.
Questo programma è software libero; siete liberi di ridistribuirlo secondo i termini
della GNU General Public License versione 3 o (a scelta) una versione più recente.
Questo programma non ha assolutamente alcuna garanzia.
xubuntu@xubuntu-VirtualBox:~/Scrivania/AhmadShatti_578268$ make test
running test suite...
./test.sh
test1 passed
test2 passed
test3 passed
test4 passed
test5 passed
test6 passed
```

Con i target `clean` e `clear` si eliminano i file generati e i file temporanei. Per testare il codice, sono stati principalmente utilizzati i test case forniti. Per verificare che i test fossero sempre affidabili e producessero risultati consistenti in tutte le esecuzioni è stato creato un altro script che eseguiva tutti i test di `test.sh` un numero arbitrario di volte.

```
#!/bin/bash
for i in {1..10}
do
    echo "---------------------- TEST #$i --------------------------"
    make clear
    make
    make test
    echo "------------------------------------------------------------"
done
make clear
```

Questo approccio ha contribuito a garantire che i test fossero sempre veritieri e che il programma mantenesse una stabilità constante.