

Laboratorio di Sistemi Operativi

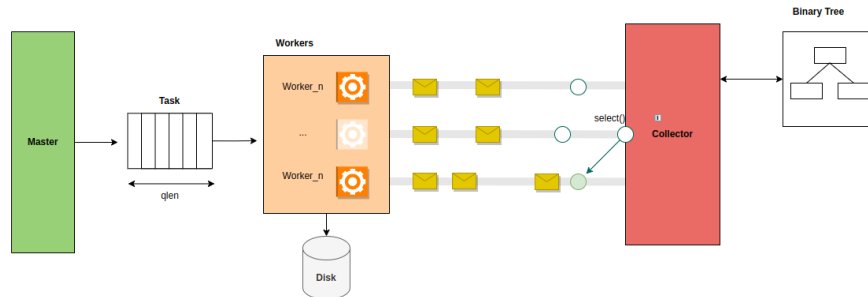
Relazione progetto Farm

Francesco Salvatori
8 Mar 2023

Contents

1	Struttura del progetto	2
1.1	Master-Worker	2
1.2	Collector	3
2	Libreria Statica	3
2.1	Struttura e compilazione	4
2.2	lib/src/threadmaster.c	4
2.3	lib/src/threadworker.c	4
2.4	lib/src/queue.c	4
2.5	lib/src/tree.c	4
2.6	Altri file .c	4
3	Compilazione ed esecuzione dei test	5
3.1	Makefile	5
3.2	Test	5

1 Struttura del progetto



Architettura del progetto

Il progetto è composto da due processi principali: Master-Worker e Collector. Il thread master comunica con i thread worker tramite una coda di file da elaborare, i thread worker comunicano con il Collector tramite una socket generata da Collector. Collector comunica con una struttura dati ad albero per immagazzinare i risultati ottenuti dai thread worker.

1.1 Master-Worker

Il processo Master-Worker fa il parsing degli argomenti a riga di comando e carica la lista di files da elaborare, genera i thread master, il threadpool dei thread worker, e si mette in attesa della terminazione (e in ascolto dei segnali). Master-Worker è un processo multi-threaded: crea un thread per eseguire la routine del thread Master ed un numero variabile di thread (threadpool): uno per ogni Worker specificato dall'opzione -n.

Il thread Master comunica con i thread Worker attraverso una lista di task, implementata con una coda thread-safe. L'unica funzione del Master thread è quella di immettere sulla coda dei file da elaborare (attendendo, qual ora la lista fosse piena) fino a terminarli tutti o finché non viene interrotto dal flag "Master_running", che può essere impostato solo alla ricezione di un segnale di terminazione.

Una volta raggiunto lo stato di terminazione, il Master Thread inserisce nella coda dei Task tanti files dummy "QUIT" quanti sono i thread che fanno parte del threadpool, causando la loro terminazione.

La funzione `recursive_file_walk_insert(char* dir, list * l)` prende in input il nome della directory da visitare ed una lista di caratteri, l.

L'accesso alla cartella viene fatto con `opendir()` e `readdir()`. Dato che gli unici campi POSIX-compliant di una struct dirent sono `d_name` e `d_ino`, per interpretare la tipologia di file è stato utilizzata la funzione `stat()` e le macro `S_ISDIR()` e `S_ISREG()`.

1.2 Collector

Collector è un processo "single-threaded".

La comunicazione tra Master-Worker e Collector è stata realizzata designando quest'ultimo come processo incaricato di creare la socket "server". Dovendo gestire più richieste contemporanee da parte dei worker, ho deciso di ricorrere alla funzione `select()`.

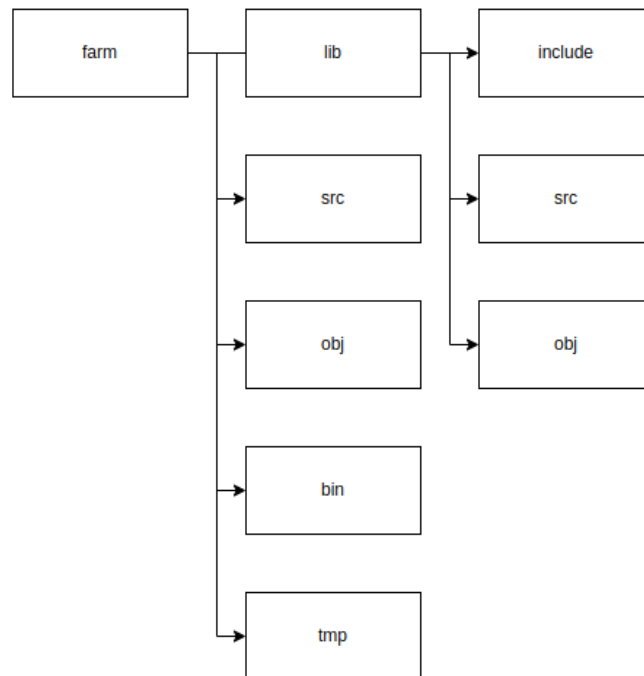
Collector riceve da parte dei worker il nome del file e la dimensione in un'unica stringa. Usando `strtol()` vengono divisi i singoli contributi. Il protocollo di comunicazione è di tipo "best effort", non c'è ritrasmissione né segnale di ACK da parte del server verso i client.

Collector può ricevere da Master-Worker un segnale `USR2`, ed in quel caso, l'handler imposta ad 1 un flag che provocherà la stampa istantanea dei file memorizzati nella struttura dati.

Per evitare di perdere la ricezione di segnali `USR2` durante la stampa, viene incrementato un contatore che tiene traccia di quante stampe devono ancora essere gestite.

2 Libreria Statica

I files `masterworker.c` e `collector.c` vengono compilati usando la libreria statica `libutils.a`, generata a partire da tutti gli altri file `.c` in cui è stato diviso il codice delle varie funzioni.



Struttura delle directory

2.1 Struttura e compilazione

I file sono divisi per sottocartelle in base alla tipologia.

I file `.c` si trovano nella cartella **src**.

I file `.h` si trovano nella cartella **include**.

I file `.o` si trovano nella cartella **obj**.

I file eseguibili si trovano nella cartella **bin**.

2.2 lib/src/threadmaster.c

In questo file è racchiuso il codice della funzione svolta dal thread master e la funzione che permette di accedere ricorsivamente ai file dentro le directory.

La funzione del thread master semplicemente fa un'operazione di 'pop' ad intervalli di X msec dalla lista di files e la accoda sulla coda condivisa con i thread worker. Al termine della sua esecuzione accoda tanti task di terminazione quanti sono i thread worker generati precedentemente.

2.3 lib/src/threadworker.c

In questo file vi è il codice della funzione svolta dai thread worker e la funzione che permette di eseguire il calcolo sui file `.dat` generati da `generafile`.

2.4 lib/src/queue.c

In questo file vi è il codice che realizza la coda sincrona usata per la comunicazione tra thread master e thread worker. Una coda queue al suo interno ha una mutex per sincronizzare l'accesso alla struttura dati e due condition variables: una per segnalare che la coda non è piena e l'altra per segnalare che la coda ha almeno un elemento al suo interno.

La prima condizione serve per permettere al thread master di continuare ad inserire file da elaborare, la seconda per far iniziare a lavorare i thread worker.

2.5 lib/src/tree.c

La gestione dei file lato Collector è affidata ad una struttura dati simile ad un albero binario di ricerca, garantendo, al caso medio, in un albero bilanciato, un tempo di inserimento di $O(\log(n))$, con il vantaggio di poter traversare l'albero *in order*, ed averlo già ordinato in tempo $O(n)$.

I nodi dell'albero binario di ricerca sono delle coppie (chiave, valore), in cui la chiave è il valore calcolato ed il valore è una lista di file, in cui ogni file ha lo stesso risultato della computazione dei thread worker.

2.6 Altri file .c

Gli altri file realizzano altre utility, come le funzioni che permettono di inviare e ricevere messaggi sulla socket e le strutture dati per manipolare i file.

3 Compilazione ed esecuzione dei test

Per ottenere i source files vi sono due modi: si può scompattare l'archivio consegnato in una nuova directory, oppure seguire la procedura seguente: Aprire un terminale e digitare:

```
$ git clone https://github.com/fsalva/farm
```

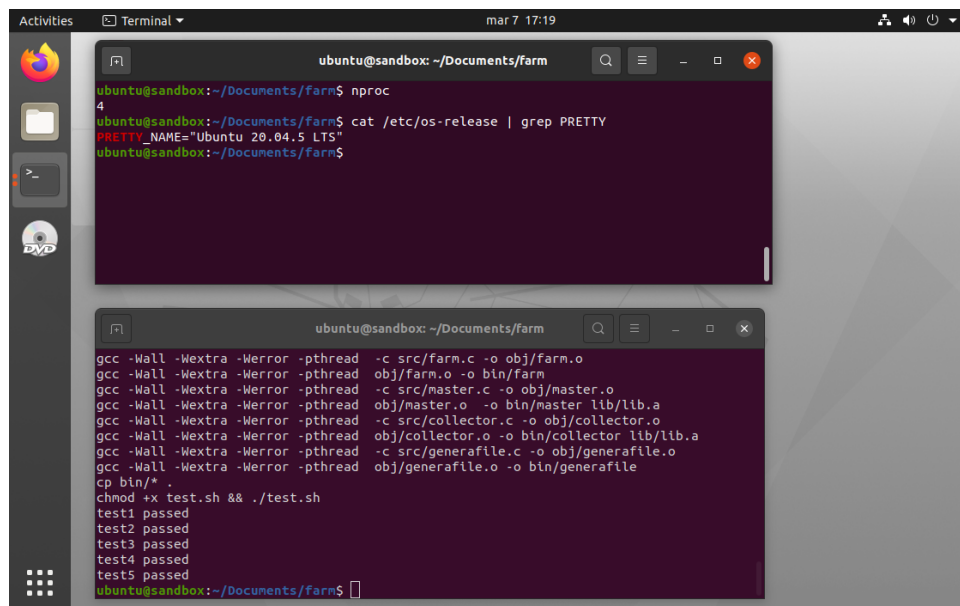
3.1 Makefile

Per testare compilazione ed esecuzione dei test, una volta spostati nella directory, basterà digitare il comando:

```
$ make test
```

3.2 Test

Il codice è stato testato su sistema operativo Fedora Linux 36 e su Ubuntu LTS 20.04 su cui sono stati installati i pacchetti gcc, valgrind e make. In fase di testing sono stati provati i segnali di terminazione, il segnale USR1 (per il trigger della stampa automatica), la duplicazione di files (con nome diverso), l'input di due file identici.



```
ubuntu@sandbox: ~/Documents/farm
ubuntu@sandbox:~/Documents/farm$ nproc
4
ubuntu@sandbox:~/Documents/farm$ cat /etc/os-release | grep PRETTY
PRETTY_NAME="Ubuntu 20.04.5 LTS"
ubuntu@sandbox:~/Documents/farm$

gcc -Wall -Wextra -Werror -pthread -c src/farm.c -o obj/farm.o
gcc -Wall -Wextra -Werror -pthread -c src/master.c -o obj/master.o
gcc -Wall -Wextra -Werror -pthread -c src/collector.c -o obj/collector.o
gcc -Wall -Wextra -Werror -pthread -c src/generafille.c -o obj/generafille.o
cp bin/* .
chmod +x test.sh && ./test.sh
test1 passed
test2 passed
test3 passed
test4 passed
test5 passed
ubuntu@sandbox:~/Documents/farm$
```

Test su macchina virtuale multi-core Ubuntu 20.04 LTS