# LABORATORIO DI SISTEMI OPERATIVI

# PROGETTO FARM 22/23

Relazione di Raoul Morucci, Matricola 580102

Data consegna 17/05/2023

# DESCRIZIONE GENERALE DEL FUNZIONAMENTO

Farm è un programma C che presi in input dei file, directory e parametri, crea due processi denominati MasterWorker e Collector che comunicano tramite una socket AF_LOCAL.

All' avvio del main() infatti viene eseguita una fork() da cui il padre prende il nome di Masterworker che sarà poi il client, il figlio prende il nome di Collector che rappresenterà il server, è stato scelto questo tipo di paradigma per il seguente scenario:

Il client vuole connettersi sulla socket, se il server è in ascolto sull'indirizzo della socket accetta la connessione, altrimenti il client è costretto ad attendere per connettersi.

Da qui le esecuzioni verranno analizzate separatamente cercando di descrivere i due processi seguendo una linea temporale (facile per il Collector, difficile per il MasterWorker avendo in esecuzione più threads).

# SCELTE IMPLEMENTATIVE DEL PROCESSO MASTERWORKER

Il processo MasterWorker si deve occupare di:

gestire l'inpunt, inserire i file nella coda concorrente, gestire i threads che calcolano i risultati sui file e scrivono sulla socket e gestire i segnali in arrivo.

Adesso andremo a descrivere come ognuna di queste fasi è stata implementata.

## GESTIONDE DELL'INPUT

L'input viene gestito tramite un parser che fa uso della funzione getopt().

Gli argomenti -n, -t, -q vengono passati alla funzione arg_int() che verifica se sono dei numeri interi positivi e restituisce il valore alle rispettive variabili globali Nthread, Tdelay e Qlen.

L'argomento -d viene passato alla funzione arg_chr_dir() che verifica se la stringa passata è una directory esistente, in seguito ritorna un puntatore a Directoryname.

Per la gestione dei file è stato preso in considerazione il fatto che alla terminazione di getopt() argv risulta orinato nel seguente modo:

{argomenti gestiti da getopt() ⇦ , ⇨argomenti non gestiti da getopt()}

quindi si sfrutta questa invariante per leggere solo i file regolari da destra verso sinistra.

## INSERIMETO FILE NELLA CODA CONCORRENTE

Per l'implementazione della coda concorrente di nome Queue è prevista una struct, list, La quale ha all'interno delle variabili per la gestione della concorrenza (cond e mutex) e due puntatori di tipo struct node che rappresentano la testa e la coda di una lista doppiamente linkata al cui interno ha i pathname dei file.

Dopo la gestione dell'input, la creazione dei thread e la connessione sulla socket (che verranno spiegate in seguito) vengono inseriti i file nella coda concorrente leggendo argv da destra verso sinistra e verificando che siano file regolari. Ogni volta che si legge un file regolare il precesso esegue una nanosleep() pari a Tdelay espresso in millisecondi tramite la funzione my_nanosleep(). A questo punto il file viene inserito in testa alla

coda concorrente tramite la funzione head_insertion() che inserisce in testa il pathname del file allocando memoria pari alla lunghezza della stringa +1.

Se l'argomento -d è stato passato in ingresso viene eseguita la funzione list_files_recursively() che ricorsivamente visita Directoryname e le sue eventuali sotto directory inserendo i file regolari nella coda concorrente esattamente come nella gestione dei file in argv.

In ognuno dei due casi di inserimento si gestisce opportunamente la concorrenza sulla coda tramite lock di mutex, e wait, signal sulla condizione seguendo il paradigma produttore consumatore, considerando anche che la coda ha una lunghezza massima pari a Qlen.

## GESTIONE DELLA THREADPOOL

La thredpool chiamata th_pool è descritta da una struct pool che prevede al suo interno un array di thread pthread_t, mutex, cond più due variabili che vanno ad indicare il numero di thread nella threadpool e quanti sono in esecuzione.

La threadpool viene inizializzata tramite la funzione thread_create() che crea la threadpool ed inizializza le variabili all'interno di th_pool. Ogni viene thread creato in modalità joinable a cui viene passata l'esecuzione della funzione thread_work() che si occupa di gestire la concorrenza sulla coda concorrente e quando possibile eseguire operazioni su Queue e scrittura sulla socket. Ogni qual volta un thread riprende l'esecuzione dopo una wait (quindi ha in possesso la mutex) va ad eseguire delete_last() che elimina l' ultimo elemento della lista restituendo un pathname del file su cui deve essere eseguito il calcolo "risultato = ∑(i * file[i])" aprendo il file in binario e leggendo a chunk di 8 byte, questa ultima operazione viene eseguita dalla funzione result_from_file() che quindi prende un pathname e restituisce un long come risultato. A questo punto acquisisce la lock per la scrittura sulla socket, verificando prima che non ci siano richieste di stampa da parte del signalhandler, questa parte verrà approfondita dopo. Avviene a questo punto la scrittura sulla socket tramite la funzione write_on_socket() che scrive pahtname, lunghezza pathname e risultato tramite la funzione write_n(), è stata usata questa funzione per evitare le scritture parziali sul canale.

Le condizioni di terminazione dei threads sono 2:

1) Se il master ha finito di inserire nella coda concorrente e non ci sono altri file da leggere nella coda la variabile cond_term = FALSE, a questo punto un thread sveglia tutti gli altri thread tramite broadcast.
2) Il signalhandler ha ricevuto un segnale di terminazione, quindi i thread devono prima finire di consumare la coda e dopo terminare.

Al momento della terminazione ogni thread modifica la variabile n_th_on_work che indica il numero di thread in esecuzione che stanno lavorando, quando questa variabile raggiunge lo 0 si notifica al Collector tramite la socket che non ci saranno più scritture da ora in poi, a questo punto tutti i thread fanno la join.

## GESTIONE DEI SEGNALI

La gestione dei segnali viene implementata tramite la funzione signal_handler_master() che ignora SIGPIPE e maschera i segnali SIGINT, SIGQUIT, SIGTERM, SIGHUP, SIGUSR1 e

SIGUSR2. Dopo crea un thread con attributi detached che rappresenterà i signalhandler a cui viene fatta eseguire la funzione signal_handler_thread_work() che si occupa della gestione dei segnali, in particolare:

- Alla ricezione dei segnali SIGINT, SIGQUIT, SIGTERM e SIGHUP setta la variabile globale term_for_sig = FALSE in modo che i thread in esecuzione inizino la routine di terminazione, quindi viene eseguita la funzione write_on_socket_finish() che scrive sulla socket come long -SIGTERM (-15) per notificare la terminazione del processo MasterWorker .
- Alla ricezione di SIGUSR1 si va a modificare la variabile globale _sigusr1 incrementandola di 1 (parte da 0), in modo che i thread la confrontino con la variabile count_sigusr1 e finché le due variabili non hanno lo stesso valore, un thread worker scrive sulla socket tramite la funzione write_on_socket_sigusr1() che scrive come long -SIGUSR1 (-10) per notificare al Collector la richiesta di stampa.
- Alla ricezione di SIGUSR2 il signalhandler termina la sua esecuzione, e non essendo joinable rilascia le risorse.

# SCELTE IMPLEMENTATIVE DEL PROCESSO COLLECTOR

Il Collector si occupa di ignorare tutti i segnali gestiti dal MasterWorker, aprire la connessione della socket, leggere sulla socket e stampare i file ed i rispettivi risultati in modo ordinato.

## GESTIONE SEGNALI

All'avvio il Collector si occupa gestire opportunamente SIGPIPE, SIGINT, SIGQUIT, SIGTERM, SIGHUP, SIGUSR1 e SIGUSR2 tramite la funzione sigaction() che permette di settare un determinato comportamento alla ricezione di un segnale, in questo caso è stato scelto SIG_IGN, quindi tutti i segnali specificati sopra saranno ignorati.

## APERTURA DELLA CONNESIONE SOCKET

In questo caso è stata scelta una singola connessione, quindi una sola socket su cui tuui i thread scrivono e su coi il Collector legge. Quinidi in ordine vengono eseguite socket(), bind(),listen(),accept(), dopo l esecuzione con successo dell' accept il canale è attivo e possono essere effettuate letture e scritture.

## LETTURA SU SOCKET E GESTIONE DELLA STAMPA

La lettura sulla socket è eseguita tramite la funzione read_from_socket() che legge in ordine risultato, lunghezza pathname e pathname, ogni lettura viene eseguita tramite read_n() che evita letture parziali su canale.

La funzione read_from_socket() gestisce tre casi in particolare:

1) Il risultato è un numero >= 0 il pathname viene letto correttamente e la lunghezza è un numero >=0 a questo punto viene creato un nuovo nodo nell' albero binario di ricerca di nome root, il quale è descritto da una struct abr che prevede delle variabili per contenere risultato e pathname di lunghezza letta sulla socket. La creazione di un nuovo nodo viene fatta tramite la funzione create_tree() che naviga l' abr usando come metodo di confronto il risultato.

2) Il risultato letto è uguale -SIGTERM (-15) quindi non si eseguono le read_n() di pathlen e lunghezza pathlen, la funzione ritorna 2 e si procede alla terminazione del Collector effettuando prima la stampa dell' albero e dopo la chiusure delle socket e la unnlink().
3) Il risultato letto è uguale a -SIGUSR1 (-10), viene invocata la funzione print_tree() che stampa l' albero in modo ordinato facendo una visita simmetrica di root, non esegue altre read_n() come nel caso (2) ma termina la funzione che viene ripetuta finché non si verifica la condizione di terminazione.

# SUDDIVISIONE IN FILE

Andremo adesso descrivere come sono stati suddivisi i file tra i due processi e a quale scopo.

## FILE COMUNI

Qui descriveremo i file comuni ai due processi:

- [datastruct.h]: contiene le descrizioni delle strutture dati implementate.
- [utils.c , utils.h]: contengono macro, le funzioni e le definizioni di funzione che vengono usate per la lettura, scrittura e allocazione di memoria.
- [main.c]: contiene la fork() dei due processi dopo il quale coincide con il processo MasterWorker e funzioni usate dal processo padre.

## MASTERWORKER

- [list.c , list.h]: contengono le funzioni e le definizioni per l' inserimenti in testa, l'eliminazione in coda per la coda concorrente Queue.
- [thread.c , thread.h]: contengono le funzioni e le definizioni per la gestione di threadpool th_pool ,thread worker e del signal handler thread.

## COLLECTOR

- [tree.c , tree.h]: contengono le funzioni e le definizioni per l'inserimento ,la stampa e la liberazione di spazio per l'abr root.
- [collector.c , collector.h]: contengono le definizione e le funzioni per la lettura sulla socket e la funzione principale eseguita dal Colector.

# COME COMPILARE ED ESEGUIRE FARM

La compilazione è prevista usando il makefile, dopo il quale viene generato un eseguibile "./farm".

Descrizione comandi make file per:

Compilare farm: "make".

Compilare generafile: "make generafile".

Eseguire i test forniti dal professore: "make test".

Eseguire i test forniti dal professore più altri test forniti dallo studente: "make mytest".

Eseguire con impostazioni di default: “make exec”.

Eseguire con impostazioni di default usando valgrind: “make valg”.

Ripulire la cartella dai file generati: “make clean”

Riportare la cartella allo stato iniziale: “make cleanall”

# TEST FATTI DALLO STUDENTE

Oltre ai test {1,2,3,4,5} forniti dal professore lo studente ha eseguito ulteriori test presenti in mytest.sh.

## Test 6

lancia 4 volte SIGUSR1

```
./farm -n 1 -d testdir -q 1 -t 1000 file* 2>&1 &
```

## Test 7

lancia 2 volte il segnale SIGUSR1 e dopo lancia SIGINT

```
valgrind --leak-check=full --log-file=/dev/null ./farm -n 1 -d testdir -q 1 -t 1000 file* 2>&1 &
```

## Test 8

verifica che alla ricezione di due segnali quasi nello stesso momento, vengano gestiti entrambi

```
valgrind --leak-check=full --log-file=/dev/null ./farm -n 1 -d testdir -q 4 -t 1000 file* 2>&1 &
```

## Test 9

come il test 3 ma eseguito con valgrind per verificare memory leaks

```
valgrind --leak-check=full --error-exitcode=1 --log-file=/dev/null ./farm -n 1 -d testdir -q 4 -t 1000 file* 2>&1 > /dev/null &
```

## Test 10

passa solo i file generati

```
valgrind --leak-check=full --error-exitcode=1 --log-file=/dev/null ./farm file* 2>&1 > /dev/null &
```

## Test 11

passa solo l'argomento -d testdir

```
valgrind --leak-check=full --error-exitcode=1 --log-file=/dev/null ./farm -d testdir 2>&1 > /dev/null &
```

# CONCLUSIONI

Il codice è stato svolto e testato su una macchina multi-core Linux Ubuntu 20.04 LTS e testato ulteriormente su una VM Xubuntu fornita dal professore su didawiki. I test hanno dato buon esito, tutto il codice è presente al seguente link [IrColligiano (Morucci Raoul) · GitHub](). .