

# Relazione progetto SOL a.a. 2022/2023


Lorenzo Pernigoni


12/01/2023


## Contents


<b>1</b>	<b>Directory e contenuto</b>	<b>2</b>
<b>2</b>	<b>Makefile</b>	<b>2</b>
<b>3</b>	<b>Test</b>	<b>2</b>
<b>4</b>	<b>Librerie</b>	<b>3</b>
<b>5</b>	<b>Main - MasterWorker</b>	<b>3</b>
5.1	Main . . . . .	3
5.2	Worker . . . . .	3
<b>6</b>	<b>Collector</b>	<b>4</b>


# 1 Directory e contenuto


 `bin` → eseguibili (`farm` e `generafile`)

 `includes` → include file

 `libs` → librerie

 `my_testdir` → file per `my_test.sh`

 `objs` → file oggetto

 `src` → file sorgente

`Makefile`, `my_test.sh`, `test.sh`

`relazione.pdf`, `testo.pdf`

## 2 Makefile

- `make` → Esegue la regola `all` che compila tutto il progetto.
- `make debug` → Fa le stesse cose di `make`, in più definisce la macro `DEBUG`. Durante l'esecuzione del programma `farm` verranno mostrate a video le stampe di debug.
- `make test` → Esegue lo script `test.sh`.
- `make testloop` → Esegue lo script `test.sh` per 10 volte.
- `make mytest` → Esegue lo script `my_test.sh`.
- `make clean` → Elimina gli eseguibili, i file oggetto, le librerie e tutti i file creati dallo script `test.sh` (se presenti).

## 3 Test

Quando si vuole eseguire lo script `test.sh`, assicurarsi, eventualmente con `make clean` e `make`, che il programma non sia stato precedentemente compilato con le stampe di debug attive.

Oltre a `test.sh` è stato aggiunto `my_test.sh` che esegue 3 volte `farm` nei seguenti modi:

- **Test SIGUSR1** → Esecuzione con 6 thread, coda lunga 3, delay di 500 ms e tutti i file della directory corrente (`-d .` ).  
Viene inviato il segnale `SIGUSR1` per tre volte (dopo circa 1, 2, 0.1 secondi), infine `SIGQUIT` (dopo circa 7 secondi).
- **Test Default** → Esecuzione con argomenti di default e directory `my_testdir`.
- **Test Valgrind** → Esecuzione con Valgrind, con 8 thread, coda lunga 4, delay di 0 ms e tutti i file della directory corrente (`-d .` ).

Lo script `my_test.sh` può essere eseguito con le stampe di debug attive.

## 4 Librerie

- **Coda concorrente** → `libs/libboundedqueue.a`

Per la coda concorrente è stata usata l'implementazione che ci è stata fornita a lezione. Vedere `includes/boundedqueue.h` e `src/boundedqueue.c`.

- **Thread pool** → `libs/libthreadpool.a`

Fornisce due strutture, una che rappresenta il thread pool e una che contiene le informazioni da passare ai thread del pool. Inoltre fornisce una funzione per creare il thread pool e una per distruggerlo.

Vedere `includes/threadpool.h` e `src/threadpool.c`.

- **Utils** → `libs/libutils.a`

Definisce macro, tipi e funzioni di utilità. Ad esempio sono presenti macro che controllano il valore di ritorno di una system call, funzioni che controllano il valore di ritorno di altre funzioni (`malloc`, `pthread_mutex_lock`, ...) e l'implementazione di una semplice lista di stringhe.

Vedere `includes/utils.h` e `src/utils.c`.

## 5 Main - MasterWorker

Il codice si trova nel file `src/farm.c`.

### 5.1 Main

Inizialmente viene fatto il parsing degli argomenti, i path dei file regolari passati al programma e quelli contenuti nella directory specificata con l'opzione `-d` vengono inseriti in una lista di stringhe eliminando eventuali duplicati (controllando i-node e device dei file con la `stat`).

In seguito si crea il processo Collector con una fork. Dopo la fork il padre (ovvero il processo MasterWorker) si occupa di creare: un thread in modalità detached per gestire i segnali, la coda concorrente dei task da elaborare e il pool dei Worker thread.

A questo punto gli elementi della lista precedentemente citata vengono inseriti uno ad uno nella coda concorrente con un delay in millisecondi se l'opzione `-t` è settata.

Se viene ricevuto il segnale `SIGHUP`, `SIGINT`, `SIGQUIT` oppure `SIGTERM` esso verrà gestito dal signal handler thread e si smetterà anticipatamente di inserire i path nella coda, avviandosi verso la terminazione.

Una volta scorsa tutta la lista vengono inseriti nella coda anche `n` caratteri particolari, dove `n` è il numero dei thread, in modo da notificare ai Worker che non ci sono altri elementi.

Successivamente il thread main notifica al processo Collector che deve uscire scrivendo il carattere `'1'` sul socket e ne raccoglie l'exit status. Poi termina anche MasterWorker.

### 5.2 Worker

Il generico thread Worker fa da client per la connessione socket.

Dentro a un ciclo `while(1)`: estrae un path dalla coda (se path coincide con il carattere particolare che notifica che non ci sono altri elementi esce dal ciclo), si connette al socket, esegue quattro scritture su di esso e chiude la connessione.

Le scritture sul socket sono, nell'ordine, le seguenti:

1. Il carattere '2' se bisogna notificare al processo Collector che deve stampare (quindi è stato precedentemente ricevuto e gestito il segnale `SIGUSR1`), il carattere '0' altrimenti.
2. La lunghezza del path estratto dalla coda.
3. Il path estratto dalla coda.
4. Il risultato calcolato con la funzione `calculate_result`.

## 6 Collector

Il codice si trova nel file `src/collector.c`.

Il processo Collector fa da server per la connessione socket.

Dentro a un ciclo `while(1)`: accetta una connessione, legge quattro volte dal socket e chiude il descrittore di file ritornato dalla `accept`.

Le letture dal socket sono, nell'ordine, le seguenti:

1. Un carattere: se è '1' esce dal ciclo, se è '2' stampa in modo formattato e in ordine crescente di risultato i risultati e i path ricevuti fino a quel momento chiamando `print_ordered_result`, altrimenti '0' continua senza uscire o stampare.
2. La lunghezza del path.
3. Il path.
4. Il risultato.

Alla fine, una volta uscito dal ciclo, viene fatta la stampa finale con `print_ordered_result` e il processo termina.