
# UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Sistemi Operativi
Anno accademico 2022-2023

## Progetto per:
## Laboratorio di Sistemi Operativi

Relazione progetto Farm

Prof. Massimo Torquati

Lorenzo Jori,
607152 – Corso A

# Introduzione:

Il progetto che di seguito viene presentato nei dettagli, ha come idea di base quella di: creare due processi, il processo MasterWorker e il processo Collector. Il processo Collector viene generato dal processo MasterWorker (usando la syscall Fork() ) ed entrambi comunicano reciprocamente attraverso un socket AF_UNIX. Il processo Master prende in input una lista di file binari e delle opzioni da riga di comando che vanno a settare i parametri d'esecuzione del processo. In caso gli argomenti non siano forniti il processo verrà lanciato con dei valori di default.
Il processo master prende file binari che contengono dei numeri interi lunghi (long) e su di essi svolge il seguente calcolo:

$$result = \sum_{i=0}^{N-1} i * file[i]$$

dove "N" è il numero d'interi lunghi (long) contenuti nel file mentre "i" è l'indice di riga del file corrente. "file[i]" rappresenta l'intero lungo, scritto all'interno del file, alla i-esima riga.
I risultati di questa operazione vengono mandati, usando il socket AF_UNIX, al processo Collector che si occupa di memorizzare i risultati in maniera ordinata (per ordine di risultato, crescente) per poi stamparli sullo stdout.

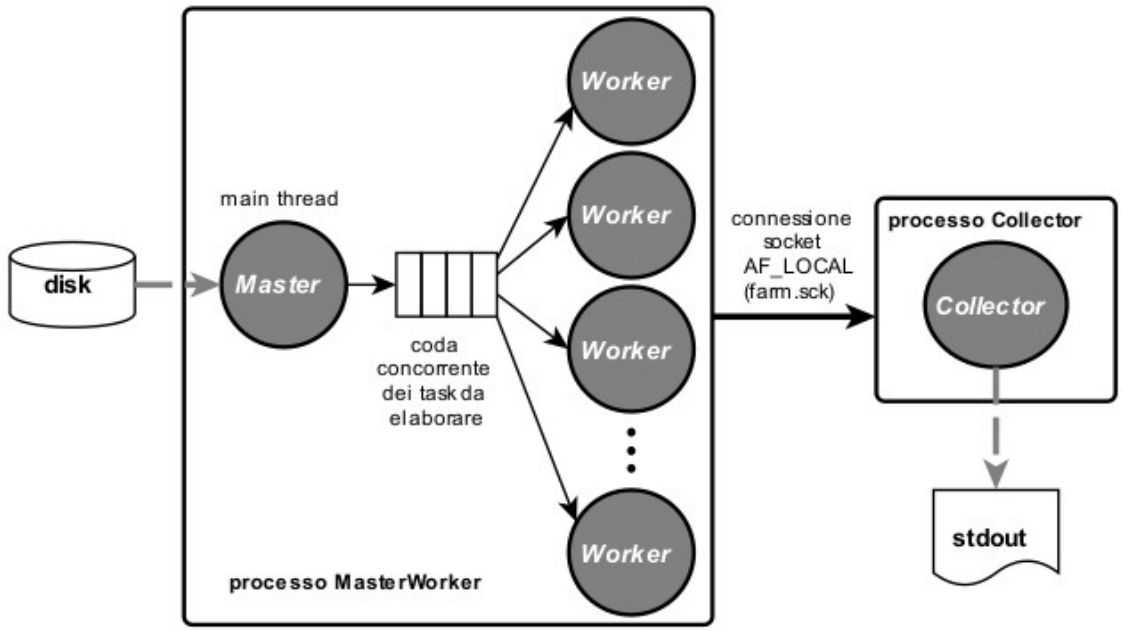


**Figura 1:** architettura logica del progetto "*farm*".

# 1. Processo MasterWorker:

L'architettura del processo MasterWorker è multithread con la seguente struttura: MasterThread che è un singolo thread che si occupa di accodare i task e un ThreadPool che si occupa di svolgere i task. Il processo ha una struttura interna organizzata nel seguente modo:

- il **main** del processo si occupa di leggere i filename, (i filename vengono parsati da argv, in quanto possono essere dati come opzione da command line) controlla che siano dei file regolari ed, in caso affermativo, li passa al MasterThread.
- Il **MasterThread** legge i filename passati dal main per poi andare ad accodarli all'interno di una coda concorrente. Essa contiene nei suoi nodi il task che il ThreadPool deve svolgere (il task è un tipo di dato astratto che è composto da: puntatore alla funzione da svolgere (cioè il task) e il nome del file, su cui svolgere il task, da leggere).
- Il **ThreadPool** si occupa di svolgere il task, cioè il calcolo, sui file binari presenti nella coda. Esso accede alla coda concorrente tirando giù il task ed eseguendolo. I singoli thread worker del pool si occupano anche di comunicare con il processo collector tramite una connessione, una per worker, ad un socket AF_UNIX dove vengono scritti i risultati che, successivamente, vengono letti dal processo collector.

Si faccia riferimento a *src/thread_pool.c, src/master_thread.c, src/farm.c* per i dettagli implementativi.

## 1.1 Struttura del task:

il task è implementato nel seguente modo:

```
typedef struct task{
    // The function to be executed.
    void * (*taskfunc)(char*);
    // The argument to be passed.
    char *arg;
}task_t;
```

dove abbiamo, come primo argomento, il puntatore alla funzione da svolgere  mentre, come secondo argomento, il filename che verrà letto dal disco. Tale struct è definita all'interno dell'include file *include/task.h* .

## 1.2 Coda concorrente:

La coda che contiene i task da svolgere è implementata usando una lista concatenata (*src/linkedList.c, src/queue.c per dettagli implementativi*) sull'intera struttura è definito un mutex, in modo tale che, l'accesso alla struttura dati sia sicuro, in mutua esclusione, senza rischi di lasciare la struttura dati "coda" in stati inconsistenti. Oltre al mutex, sulla coda, è definita anche una conditional variable in modo tale da

consentire la risoluzione del problema “produttore-consumatore”; il MasterThread produce task da svolgere per il ThreadPool mentre il ThreadPool consuma i task accodati dal MasterThread. La conditional variable consente sia al MasterThread sia al ThreadPool di mettersi in attesa su di essa in caso:

- La coda abbia **raggiunto la sua capacità massima**, in quanto: il MasterThread produce i task più velocemente rispetto alla velocità di consumo di essi da parte del ThreadPool. Conseguentemente il MasterThread dovrà attendere che il ThreadPool consumi qualche task (almeno uno) per accodare il task successivo.
- La coda sia **vuota** in quanto il ThreadPool consuma i task più velocemente di quanto il MasterThread riesca a produrli. Dunque, il ThreadPool dovrà attendere che il MasterThread abbia prodotto almeno un task.

### 1.3 MasterThread:

(*src/master_thread.c, per dettagli implementativi*). È il thread che ha il compito di accodare i task da svolgere. Esso definisce al suo interno anche il task che deve essere svolto dal threadpool: implementando la funzione “*workerTask*”. Incapsulata all’interno del task, essa definisce il lavoro che un singolo worker del pool andrà a svolgere sul file specificato.
[La scelta di mettere l’implementazione del task all’interno del MasterThread è legata, semplicemente, all’idea che sia il master a definire il lavoro per i worker].

### 1.4 ThreadPool:

(*src/thread_pool.c, per dettagli implementativi*). Il ThreadPool è il core dello svolgimento dei task, all’avvio del processo MasterWorker, dopo aver fatto il parsing di argv e settato tutti i parametri, viene creato il threadPool che, consumando la coda, porta a compimento i task in essa immagazzinati. Ogni worker, terminata l’esecuzione del task, apre una connessione socket AF_UNIX, verso il processo Collector, inviando su di esso il risultato ottenuto dallo svolgimento del task. (è stata implementata la soluzione di usare una connessione per ogni worker, invece di una singola connessione persistente). Il ThreadPool si stoppa quando riceve dal MasterThread un task “speciale” di stop: viene settato a 0 il flag “active”, presente nella struct thread_pool, che fa terminare l’esecuzione dei singoli thread nel pool.

### 1.5 Gestione dei segnali:

(*src/farm.c, src/master_thread.c. Cercare ((ctrl + f)) signal handling per identificare tutti i dettagli implementativi).*
I segnali che sono gestiti dal processo MasterWorker sono i seguenti: SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGUSR1, SIGPIPE. La gestione è affidata ad un thread che, partendo all’avvio del processo, si mette in attesa dei segnali in ingresso.

Alla ricezione di uno di essi (con l'eccezione di SIGUSR1) si devono completare i task presenti in coda, non riceverne altri e terminare l'esecuzione.
Per quanto riguarda SIGUSR1, alla sua ricezione devono essere stampati sullo stdout i risultati parziali, cioè tutti i quelli ricevuti fino a quel momento.
La gestione dei segnali è implementata usando i flag:

```
volatile sig_atomic_t partial_result;
volatile sig_atomic_t no_more_task;
volatile sig_atomic_t sig_pipe_rise;
```

Alla ricezione di un segnale il flag "partial_result" (caso: SIGUSR1) e "no_more_task" (caso: altri segnali, eccetto SIGPIPE) vengono settati ad 1. Tali flag sono presenti anche nel MasterThread dichiarati come "extern", in modo tale che il MasterThread abbia modo di leggerne il valore settato dal processo farm.
Se i flag sono settati, un task "speciale" viene messo in coda, per notificare sia al ThreadPool che al processo Collector, l'azione che deve compiere in base al segnale ricevuto.
Il segnale SIGPIPE viene gestito sia dal processo MasterWorker che, alla ricezione, termina eliminando tutte le strutture dati, sia dal processo Collector che lo ignora.
Il processo Collector gestisce tutti gli altri segnali ignorandoli.

### 1.6 Flag:

Il proccesso Master prende in ingresso alcuni parametri (flag) da riga di comando; essi specificano:

- -n <num> : numero di thread worker all'interno del ThreadPool.
- -q <num> : è il limite massimo di dimensione della coda.
- -t <num> : specifica il tempo in millisecondi che intercorre tra l'invio di due richieste successive ai thread Worker da parte del thread Master.
- -d <dirname> : specifica una cartella in cui sono contenuti i file binari.

## 2. Processo Collector:

(*src/farm.c dalla riga 210 alla 429, per dettagli implementativi*).
Il processo Collector *non* è multithread. Esso fa multiplexing di più socket usando la system call "select" che monitora i vari socket file descriptor, in base a quale si trova in stato di "ready", si verifica uno dei seguenti comportamenti:

- Se è pronto il socket di connessione, viene accettata una nuova connessione e il nuovo file descriptor viene aggiunto al set.
- Se è pronto uno dei socket di I/O con un client, avviene la lettura dei risultati calcolati dal processo master.

Una volta ricevuti i risultati dal processo MasterWorker, il processo Collector li memorizza in modo ordinato (ordinamento crescente: dall'intero lungo più piccolo al più grande) all'interno di una lista concatenata, per poi stamparli a video.

Il Collector funziona da master per le connessioni (si comporta da server) e viene creato usando la system call “fork()”, dunque è il processo figlio, mentre il padre è il MasterWorker.
Il processo Collector termina il suo ciclo while(true) quando riceve la stringa “exit”. Essa notifica al Collector che i task del processo MasterWorker sono finiti, quindi il Collector può stampare i risultati e terminare.

## 3. Gestione errori:

Gli errori vengono gestiti, quando possibile, anche tramite l’utilizzo di macro definite all’interno dell’include file “*util.h*”. In caso di errori ritenuti fatali la scelta implementativa fatta è quella di terminare il programma.

**Note:**

*Per eseguire il progetto e i test basta entrare nella cartella, dov’è presente il makefile (cartella: ProgettoFarm-SOL22-23), col terminale e lanciare il comando “**make test**”; tale comando **eseguirà lo script “test.sh”** (il test di riferimento) fornito come materiale esterno per il progetto (assieme alle specifiche del progetto e al codice generafile.c per generare l’eseguibile “generafile”).*
*Per **vedere le stampe su stdout** del processo collecotor basta lanciare il comando **“make normalexe”** e verranno visualizzate a video le stampe. (Entrando nello script “normalexe.sh” è possibile cambiare i parametri d’esecuzione cioè i flag -n, -q, -d, -t sopra spiegati). Tale script è stato usato per debugging durante la fase d’implementazione del progetto.*
*Per ripulire dai file di test basta lanciare il comando **make cleanall** e verranno eliminati tutti i file .dat, i file .txt, i file .o e le cartelle usate per il testing.*
*In alternativa si può lanciare **make clean** che non eliminerà i file .o .*
*Se si vuole automatizzare il processo descritto sopra, basta decommentare la riga 108 dello script test.sh.*

***My github link for the project:***

*il link sottostante conduce alla mia pagina github su cui sono stati fatti i commit del progetto.*
https://github.com/lorenzo2508/Laboratorio-di-Sistemi-Operativi-