

Progetto SOL – Corso A e B – a.a. 22/23

Docente Massimo Torquati

massimo.torquati@unipi.it

Si chiede di realizzare un programma C, denominato *farm*, che implementa lo schema di comunicazione tra processi e thread mostrato in Figura 1.

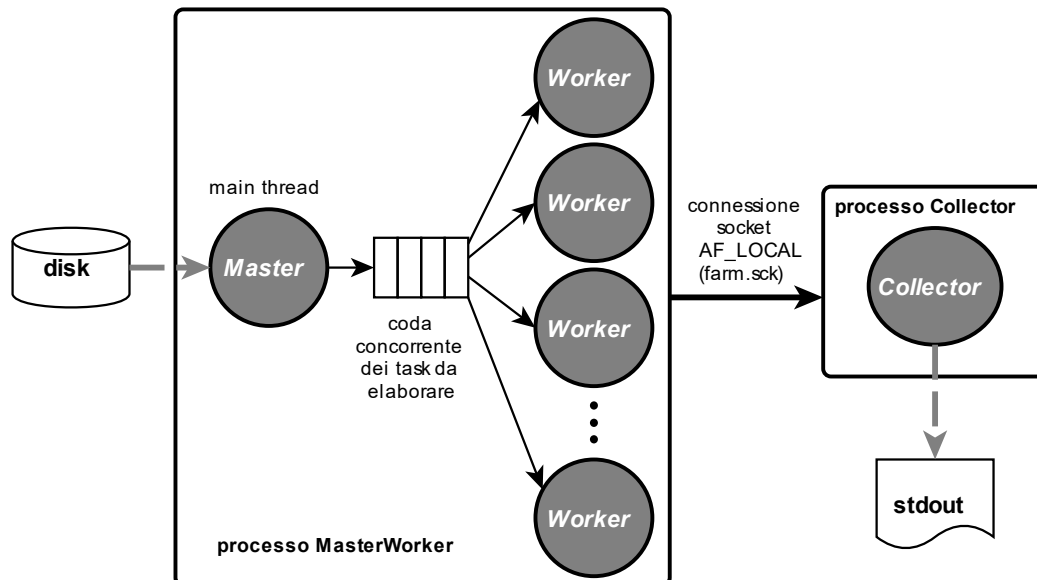


Figura 1 Architettura logica di connessione tra i processi *MasterWorker* e *Collector*

farm è un programma composto da due processi, il primo denominato *MasterWorker* ed il secondo denominato *Collector*. *MasterWorker*, è un processo multi-threaded composto da un thread *Master* e da '*n*' thread *Worker* (il numero di thread *Worker* può essere variato utilizzando l'argomento opzionale '*-n*' – vedere nel seguito). Il programma prende come argomenti una lista (eventualmente vuota se viene passata l'opzione '*-d*') di file binari contenenti numeri interi lunghi ed un certo numero di argomenti opzionali (le opzioni sono '*-n*', '*-q*', '*-t*', '*-d*'). Il processo *Collector* viene generato dal processo *MasterWorker*. I due processi comunicano attraverso una connessione socket AF_LOCAL (AF_UNIX). Viene lasciata allo studente la scelta di quale tra i due processi fa da processo master per la connessione socket, così come la scelta se usare una sola connessione o più connessioni, una per ogni *Worker*. Il socket file "*farm.sck*", associato alla connessione AF_LOCAL, deve essere creato all'interno della directory del progetto e deve essere cancellato alla terminazione del programma.

Il processo *MasterWorker* legge gli argomenti passati alla funzione *main* uno alla volta, verificando che siano file regolari. Se viene passata l'opzione '*-d*' che prevede come argomento un nome di directory, viene navigata la directory passata come argomento e considerando tutti i file e le directory al suo interno.

Il nome del generico file di input (unitamente ad altre eventuali informazioni) viene inviato ad uno dei thread *Worker* del pool tramite una coda concorrente condivisa (denominata "coda concorrente dei task da elaborare" in Figura 1). Il generico thread *Worker* si occupa di leggere dal disco il contenuto dell'intero file il cui nome ha ricevuto in input, e di effettuare un calcolo sugli elementi letti e quindi di inviare il risultato ottenuto, unitamente al nome del file, al processo *Collector* tramite la connessione socket precedentemente stabilita. Il processo *Collector* attende di ricevere tutti i risultati dai *Worker* ed al termine stampa i valori ottenuti sullo standard output, ordinando la stampa, nel formato seguente:

```
risultato1  filepath1
risultato2  filepath2
risultato3  filepath3
```

La stampa viene ordinata sulla base del risultato in modo crescente ($risultato1 \leq risultato2 \leq risultato3, \dots$). Il calcolo che deve essere effettuato su ogni file è il seguente:

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

dove N è il numero di interi lunghi (long) contenuti nel file, e $result$ è l'intero lungo che dovrà essere inviato al *Collector*. Ad esempio, supponendo che il file “mydir/fileX.dat” passato in input come argomento del *main* abbia dimensione 24 bytes, con il seguente contenuto (si ricorda che gli interi lunghi – *long* – sono codificati con 8 bytes in sistemi Linux a 64bit):

3
2
4

il risultato calcolato dal *Worker* sarà:

$N=3$, $result = \sum_{i=0}^{N-1} (i * file[i]) = (0 * 3 + 1 * 2 + 2 * 4) = 10$, quindi il processo *Collector* stamperà:

10 mydir/fileX.dat

Gli argomenti che opzionalmente possono essere passati al processo *MasterWorker* sono i seguenti:

- -n <nthread> specifica il numero di thread *Worker* del processo *MasterWorker* (valore di default 4)
- -q <qlen> specifica la lunghezza della coda concorrente tra il thread *Master* ed i thread *Worker* (valore di default 8)
- -d <directory-name> specifica una directory in cui sono contenuti file binari ed eventualmente altre directory contenente file binari; i file binari dovranno essere utilizzati come file di input per il calcolo;
- -t <delay> specifica un tempo in millisecondi che intercorre tra l'invio di due richieste successive ai thread *Worker* da parte del thread *Master* (valore di default 0)

Il processo *MasterWorker* deve gestire i segnali SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGUSR1. Alla ricezione del segnale SIGUSR1 il processo *MasterWorker* notifica il processo *Collector* di stampare i risultati ricevuti sino a quel momento (sempre in modo ordinato), mentre alla ricezione degli altri segnali, il processo deve completare i task eventualmente presenti nella coda dei task da elaborare, non leggendo più eventuali altri file in input, e quindi terminare dopo aver atteso la terminazione del processo *Collector* ed effettuato la cancellazione del socket file. Il processo *Collector* maschera tutti i segnali gestiti dal processo *MasterWorker*. Il segnale SIGPIPE deve essere gestito opportunamente dai due processi.

Note

La dimensione dei file in input non è limitata ad un valore massimo. Si supponga che la lunghezza del nome dei file (compreso il pathname) sia al massimo 255 caratteri.

Materiale fornito per il progetto

Il materiale fornito è il seguente:

- Testo del progetto (file *progettoSOLFarm_22-23.pdf*)
- Un programma *generafile.c* per generare i file per i tests
- Uno script Bash (test.sh) contenente alcuni semplici test che il programma deve superare (non consegnare il progetto se i test contenuti nello script non vengono superati).

Consegna del progetto

Il progetto deve essere consegnato sul portale Moodle <https://elearning.di.unipi.it/course/view.php?id=126> facendo l'upload di un file zip (o tgz o tar.gz) avente il seguente nome:

NomeCognome-Matricola.zip (o *NomeCognome-Matricola.tgz*).

Il docente verificherà la funzionalità del progetto su una macchina multi-core Linux Ubuntu 20.04 LTS. eseguendo lo script Bash test.sh.

Lo studente dovrà implementare tutto il codice del programma separando in file diversi il codice contenente la funziona *main*, il codice che implementa il Master thread, il codice che implementa il pool dei Worker thread ed il codice che implementa il Collector. Dovrà essere fornito il *Makefile* per la compilazione del progetto. Il Makefile dovrà avere almeno un target test per poter lanciare l'esecuzione del test.sh. Infine dovrà essere fornita una breve relazione (massimo 5 pagine) in formato PDF che descrive le principali scelte implementative ed eventuali test aggiuntivi fatti dallo studente. Il progetto deve essere svolto da un singolo studente.

Esempi di possibili esecuzioni

```
> ./farm -n 4 -q 4 file1.dat file2.dat file3.dat file4.dat file5.dat -d testdir
```

```
103453975 file2.dat
112546319 testdir/testdir/file7.dat
153259244 file1.dat
293718900 file3.dat
380867448 file5.dat
584164283 file4.dat
672594110 testdir/file6.dat
```

```
> valgrind --leak-check=full ./farm -n 8 -q 4 -t 200 file*
```

(dopo circa 1 secondo viene inviato **SIGINT** al processo MasterWorker)

```
==37245== Memcheck, a memory error detector
==37245== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==37245== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==37245== Command: ./farm -n 8 -q 4 -t 200 file1.dat file2.dat file3.dat file4.dat file5.dat file6.dat file7.dat
==37245==
```

```
64834211 file100.dat
1146505381 file10.dat
1884778221 file111.dat
258119464 file116.dat
```

```
^C 380867448 file5.dat
```

```
==37246==
==37246== HEAP SUMMARY:
==37246==   in use at exit: 0 bytes in 0 blocks
==37246== total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==37246==
==37246== All heap blocks were freed -- no leaks are possible
==37246==
==37246== For lists of detected and suppressed errors, rerun with: -s
==37246== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==37245==
==37245== HEAP SUMMARY:
==37245==   in use at exit: 0 bytes in 0 blocks
==37245== total heap usage: 18 allocs, 18 frees, 2,888 bytes allocated
==37245==
==37245== All heap blocks were freed -- no leaks are possible
==37245==
==37245== For lists of detected and suppressed errors, rerun with: -s
==37245== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```