

Progetto del corso di Sistemi operativi e Laboratorio presso unipisa – 2023/2024

Enrico Guerra, Matricola 580334

Come indicato, il programma sviluppato, noto come 'Farm', è un programma multiprocessore multithreading sviluppato per lavorare su sistemi UNIX.

Il progetto è stato sviluppato usando github come strumento di supporto al salvataggio, al 'versionamento' ed allo sviluppo su due macchine distinte: una macchina unix ed una macchina windows.

La repository di github è visibile al seguente indirizzo: <https://github.com/Ceorin/SOL-ProjectFarm2> dove è presente anche un file README scritto in lingua inglese che contiene una breve descrizione sulle funzioni e gli scopi del progetto ed una guida all'installazione ed all'uso.

Struttura del Progetto.

I file del progetto sono stati divisi secondo la seguente struttura:

- gli eseguibili [Farm, collector e generafile] vengono generati nella cartella radice del progetto, dove sono presenti anche due file .md descrittivi e dove è presente un makefile.
- i file sorgente sono contenuti tutti all'interno della cartella 'src' con l'eccezione di generafile, fornito dal docente, che insieme allo script in bash fornito sono invece nella cartella 'test'.
- gli headers sono invece tutti contenuti all'interno della cartella 'src/headers'
- i file oggetto vengono compilati invece nella cartella build

Sono poi presenti cartelle adibite ad altre funzioni che non hanno particolare uso attuale: 'log' e 'tmp'. Tmp contiene i file temporanei del progetto durante l'esecuzione, ma l'unico file temporaneo che viene creato consiste nel socket di comunicazione. Log non ha alcun utilizzo nella versione corrente, se non per ridirezionare manualmente esecuzioni del programma su file testuali.

Riguardo il Makefile, In aggiunta alla essenziale funzione di compilare i file e controllarne le versioni, questi mette a disposizione vari target fittizi, tra cui in particolare '**debug**':

Eseguire make debug compilerà i sorgenti passando al preprocessore la costante DEBUG, che viene usata per eseguire ulteriori stampe e monitorare attivamente gli eventi del programma.

Il target debug non effettua una pulizia, per tanto non ricompila file aggiornati. In questo modo è però possibile rimuovere singoli file oggetto per ricompilare in modalità debug solo alcune parti del progetto.

Componenti del Progetto.

Le componenti del progetto sono suddivise tra i file sorgenti come segue

- **main.c** -> contiene il main del programma farm e si occupa esclusivamente di creare un processo figlio su cui eseguire Collector, e di passare poi all'esecuzione di MasterWorker
- **master.c** -> contiene le funzionalità principali del processo MasterWorker, ovvero accetta l'input del programma ed inoltra i task ai thread worker.
 - **worker_pool.c** -> contiene le funzioni per monitorare i thread worker, crearne di nuovi e cancellarne.
 - **thread_task.c** -> contiene la definizione del worker thread e le funzioni per accedere concorrentemente alle risorse condivise.

+ **signal_handlers_master.c** -> contiene le funzioni alternative di gestione dei segnali per master, e l'inizializzazione delle stesse.

+ **sumfun.c** -> contiene la definizione della funzione di somma richiesta che i thread elaborano sui file. Include due versioni, una che utilizza le funzioni di libreria per i FILE ed una che utilizza i file descriptor.

- **collector.c** -> contiene il codice principale di Collector. È stato implementato come un programma a sé, per poter essere eseguito anche separatamente da Farm. Questo permette di suddividere i test di debug, sia di utilizzare Collector come un programma server indipendente. Tuttavia, tale funzionalità non è stata pienamente implementata e rimane unicamente atta all'effettuare test.

+ **collector_print.c** -> contiene il codice utilizzato per le stampe continue di collector. Tale funzione è stata inoltre delegata ad un thread che viene qui definito.

+ **signal_utils.c** -> contiene semplici funzioni che mascherano i segnali; utilizzate da main, collector e signal_handlers_master.

+ **myList.c** -> contiene una mia personale definizione di funzioni e strutture per creare e gestire una lista concatenata generica in c. Questo tipo di liste tengono traccia sia della testa che della coda, ed hanno funzionalità aggiuntive che non sono state usate per il progetto.

Ogni file sorgente (fatta eccezione per main.c) ha un file header ad esso relativo.

La maggior parte di questi è una basilare interfaccia all'uso delle funzioni disponibili del sorgente per le altre componenti del programma, con le seguenti aggiunte o eccezioni:

- **utils.h** è utilizzato da ogni file e definisce macro di uso comune e la dimensione massima delle stringhe del sistema [255 per nomi di file, 108 per nomi di socket].

- **collector.h** definisce unicamente costanti per la comunicazione con esso.

- **master.h** include i valori predefiniti delle opzioni del programma

- **sumfun.h** include la definizione della struttura dei risultati, utilizzata sia dai thread worker sia da collector.

(Se i programmi agissero su sistemi diversi, sarebbe possibile definire qui una funzione di serializzazione, ma questa caratteristica non è necessaria fintanto che si lavora in locale)

Comportamento del programma.

Dopo aver compilato il programma attraverso il Makefile, si può eseguire l'eseguibile **farm** nella cartella sorgente del progetto. Il programma così eseguito va a creare un processo figlio che esegue il programma **Collector** per poi eseguire la funzione MasterThread, chiamato in seguito semplicemente 'Master'.

Master gestisce un insieme di thread, denominati 'Workers', il cui numero può cambiare durante l'esecuzione. Master legge gli argomenti del programma, venendo eventualmente configurato attraverso le opzioni -n N, -q Q e -t T per definire rispettivamente un numero iniziale di thread, la dimensione della coda concorrente di produzione tra lui ed i worker, ed il delay tra l'inserimento di un item nella coda.

Gli altri argomenti del programma sono -d <DIR> e <FILE>: ogni argomento passato con opzione -d verrà interpretato come possibile directory da elaborare mentre ogni argomento passato senza opzioni verrà considerato come un possibile file. Questi argomenti vengono inseriti in una lista, nota come 'maybe_files', per essere elaborati uno alla volta al termine della configurazione.

L'uso di nanosleep e pthread ha evidenziato la necessità di usare la versione di C POSIX, che ha portato ad un comportamento inatteso della funzione getopt. Questa è stata per tanto utilizzata con il flag '-' per permettere il riconoscimento di argomenti privi di opzione e far sì che ogni argomento possa venir elaborato indipendentemente dall'ordine con cui è stato passato al programma.

Si nota inoltre che l'invio di molteplici opzioni -n, -q e -t non causa interruzioni ma verrà utilizzato l'ultimo dei valori inviato.

Inoltre, se uno (o più) di questi valori passato come argomento non è valido (ad esempio dimensioni negative), verrà utilizzato un valore di default definito come segue: [n: 4 | q: 8 | t: 0]. Per l'opzione -t è stato scelto di definire anche un tempo massimo di 3 secondi di attesa; ogni valore a questo superiore verrà arrotondato a tale limite.

Dopo aver elaborato gli argomenti, Master inizializza il pool di thread e procede ad elaborare la propria lista di possibili-file. Data la definizione non-standard di d_type come valore di dirent (ed assenza sulla mia macchina), è stato necessario effettuare della compilazione opzionale ed utilizzare, in assenza di una definizione di DT_DIR, la funzione POSIX stat per verificare la validità di file e directory.

L'azione di inizializzazione del **thread pool** consiste nel creare la coda concorrente, inizializzare alcune variabili e creare un numero iniziale di thread passato come argomento.

È stata presa la decisione di gestire i thread worker in modalità *detached*. Questo è stato preferito ragionando su una continua richiesta di aumento e riduzione del numero di thread che avrebbe causato un maggior overhead ad una versione sincronizzata, la quale avrebbe dovuto cercare ogni volta quali thread avrebbero catturato richieste di terminazione.

Tuttavia, la scelta ha rivelato complicazioni dovute alla pulizia della memoria all'uscita dal programma (variabili mutex e condizioni) che ha causato problemi a cascata risultanti in una terminazione incerta riguardo il caso dell'ultimo thread in esecuzione o di thread che hanno fallito a connettersi al socket.

Riguardo la **coda concorrente**, questa è stata implementata come uno stack di stringhe di 255 caratteri. Si nota inoltre che le richieste di terminazione ai thread sono gestite in contemporanea allo stack: il thread pool può produrre sia un valore da consumare, *sia una richiesta di uscita*. Entrambe le produzioni avvertono i thread in attesa sulla coda, che si bloccano *esclusivamente* quando la coda è vuota e non vi sono richieste di terminazione. Appena una di queste condizioni cambia, la priorità è gestita come segue:

- Se la coda ha valori e siamo in stato di chiusura, consuma sempre i valori dalla coda
- Se vi sono richieste di terminazione, consuma la richiesta invece di un item della coda
- Altrimenti, consuma l'ultimo item messo sullo stack.

Quando un thread consuma un valore dalla coda, esegue poi la funzione di calcolo e la invia al collector attraverso una connessione aperta all'inizio della propria esecuzione.

Quando un thread consuma una richiesta di terminazione, invia al collector un risultato con delle costanti predefinite di terminazione in collector.h *"/"* e *"/"*

Siccome i thread non possono consumare cartelle (master si occupa di elaborare le cartelle ed aggiungere i file al loro interno alla lista dei file da controllare, ricorsivamente) questi nomi non verrebbero mai valutati per un calcolo ed è noto non siano risultati possibili.

In particolare, Collector considererà *"/"* come richiesta di chiusura della trasmissione del singolo socket, e *"/"* come informazione di fine dell'esecuzione e di assenza di futuri valori. Tuttavia, è possibile che più di un thread invii tale valore (che viene inviato in fase di terminazione), così come è possibile che tale valore ritardi sullo stream.

Collector è per tanto abilitato a ricevere molteplici istanze di “fine esecuzione”, ed aspettare che ogni socket che ha aperto termini di mandare i propri messaggi prima di chiudere la connessione.

Quando Master termina di consumare la propria coda di possibili file, o quando riceve un segnale apposito, egli chiederà attraverso la threadpool la terminazione ad ogni thread per poter eliminare le risorse. Siccome i thread sono detached, egli invierà una richiesta di cancellazione di un numero di thread pari al numero di thread attualmente in esecuzione.

Questa funzionalità, ancora caratterizzata da alcuni difetti, si basa comunque sulla consistenza dello stato interno di creazione e rimozione di thread gestita dal thread pool. Tuttavia, potrebbe dover essere modificata in caso di refactoring del thread pool.

Collector funziona invece come processo dual-threaded server. Dopo aver inizializzato la gestione dei segnali, collector crea una lista generica dove contenere puntatori di result_value, e poi crea un thread in modalità cancellabile. Questo thread, definito come printing_thread ed inseguito chiamato **‘thread di stampa’**, riceve il puntatore alla lista come argomento e rimane in un loop infinito di stampa ogni secondo di tempo **relativo**. Al termine dell’esecuzione, collector cancellerà il thread che, come funzione di uscita, effettuerà un’ultima stampa.

L’ordinamento e la stampa della lista sono stati oggetto di analisi e sono stati ottimizzati come segue: la mia lista generica è dotata sia di puntatore a testa che a coda. Inserendo ogni elemento sempre in coda all’interno del main, è possibile quindi escludere dalla race condition i valori che vanno dal puntatore di testa al puntatore immediatamente precedente a quello di coda.

Per tanto, l’inserimento nella lista da parte del main viene fatto in mutex solamente con **l’inizio** della funzione di stampa, in cui si salvano il valore del puntatore a coda e la dimensione della lista (per stampare fino a quel punto).

Viene poi eseguito un algoritmo di mergesort iterativo sulla porzione della lista definita, in maniera non dissimile da come si effettuerebbe su una lista circolare, passando alla funzione il puntatore di fine “da trattare come null” e concatenare al termine dell’esecuzione alla porzione di lista ordinata. L’unico prezzo da pagare per tutto questo è non poter stampare l’ultimo elemento, che si può risolvere con un elemento fittizio aggiunto al termine dell’esecuzione.

Tutto questo garantisce il beneficio di mantenere un inserimento $O(1)$ nella lista, un ordinamento $O(n \cdot \log n)$ e di non bloccare il programma per la stampa.

È stato infatti valutato che l’uso di un semplice inserimento ordinato avrebbe causato un inserimento di $O(n)$ n volte, per un’esecuzione al caso pessimo di $O(n^2)$, mentre l’uso di un array avrebbe sì mantenuto l’inserimento $O(1)$ e l’ordinamento efficiente, ma sarebbe stato altresì costoso o la necessità di riallocare l’array in termini di tempo, o la necessità di definirlo di una dimensione eccessiva inutilizzata in termini di spazio. Inoltre, entrambe queste possibilità avrebbero reso impossibile effettuare l’ordinamento in parallelo.

Il **main di collector** si occupa quindi di agire come server per i worker thread che si connettono, attraverso l’uso della funzione poll, preferita alla select per la maggiore chiarezza nell’uso e migliore efficacia nel gestire un numero maggiore di connessioni. (Non al livello di funzioni della famiglia di poll successive, ma sufficiente per gli obiettivi del programma)

Considerando un numero di thread relativamente limitato, e con l’idea che la maggior parte delle connessioni si generino in blocco (all’inizio dell’esecuzione o durante momenti in cui un utente manda una

grande quantità di segnali sequenzialmente), il socket server di collector accetta un gran numero di richieste in sequenza fintanto che può evitare di bloccarsi, ma sui descrittori rimanenti – clienti che mandano valori risultato – elabora una singola richiesta per cliente per poll.

La struttura dei file descriptor di poll è un array che viene riallocato dinamicamente. È stata presa la decisione di effettuare un'estensione **lineare** dell'array, invece che esponenziale, poiché è comunque considerato un caso limite e non ideale quello in cui il numero di connessioni diverse cresca particolarmente fuori misura all'interno di un processo locale come questo. La velocità dell'incremento è definita in collector.h.

Diversamente dall'estensione, la compressione della poll nell'implementazione corrente effettua un dimezzamento della dimensione dell'array quando più del 75% del suo spazio è inutilizzato, fino comunque un limite inferiore pari a quello iniziale.

Problemi noti

Come riportato nel README di github, è noto che il programma non è funzionante al 100%.

È stato osservato empiricamente che più del 90% delle volte il programma supera con successo tutti e 6 i test dello script shell, ma altre volte manca di inviare l'ultimo dei valori risultato e addirittura **è possibile che vada in deadlock o loop infinito** intorno la chiusura del thread pool.

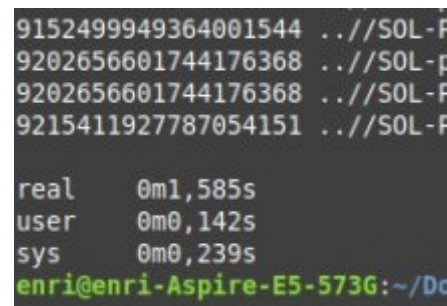
Sfortunatamente non sono riuscito a risolvere il problema prima di questa data di scadenza, ed i tentativi di aggiustare il difetto sembrano averlo portato in risalto.

Test Aggiuntivi

Sono stati effettuati dei test sia con valgrind che non, passando come parametro grandi numeri di file.

L'esecuzione è solitamente andata a buon fine e con successo. In seguito, un esempio dell'output di valgrind ed un esempio del tempo di esecuzione dello stesso comando.

```
==6970== Memcheck, a memory error detector
==6970== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6970== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6970== Command: ./farm -n 4 -q 10 -d ./ -t 1
==6970==
==6970== HEAP SUMMARY:
==6970==   in use at exit: 0 bytes in 0 blocks
==6970== total heap usage: 4,656 allocs, 4,656 frees, 18,255,572 bytes allocated
==6970==
==6970== All heap blocks were freed -- no leaks are possible
==6970==
==6970== For lists of detected and suppressed errors, rerun with: -s
==6970== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```



```
9152499949364001544 ../SOL-F
9202656601744176368 ../SOL-p
9202656601744176368 ../SOL-F
9215411927787054151 ../SOL-F

real    0m1,585s
user    0m0,142s
sys     0m0,239s
enri@enri-Aspire-E5-573G:~/Dc
```