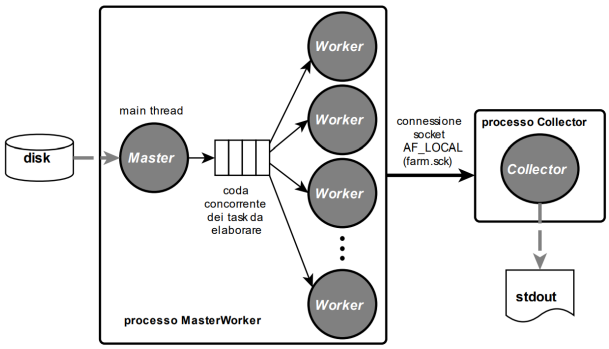
Progetto FARM
Relazione di Davide Marchi
Matricola 602476

Relazione del progetto di Sistemi Operativi e Laboratorio

Consegna e specifiche richieste



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'). 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). 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"***). 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<=risultato2<=risultato3, ...).

Possibili input

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.

File consegnati

Per poter soddisfare tutte le richieste ho deciso di dividere il mio codice all'interno di sei file C distinti, ognuno dei quali con un proprio header. Nello specifico:

MasterWorker.c	File contente il metodo main. Al suo interno è presente il parsing degli argomenti, la gestione dei segnali, della connessione e del cleanup in caso di errori
Master.c	File all'interno del quale sarà presente l'avvio del threadpool e l'esplorazione dell'eventuale directory passata come argomento
Worker.c	File che contiene l'operazione che dovrà essere fatta per ogni file, dunque la lettura, il calcolo e l'invio di tali informazioni al Collector
Collector.c	File con dentro la funzione che dovrà essere eseguita dal processo figlio dopo la fork. Contiene la comunicazione col MasterWorker e una propria funzione di cleanup
list.c	File all'interno del quale sono contenute le funzioni e struct necessarie per poter creare la coda con le informazioni che il collector dovrà memorizzare
threadpool.c	File usato anche nelle esercitazioni durante il corso a cui sono state apportate alcune modifiche. Permette la gestione di un threadpoole e dunque l'esecuzione degli worker

Funzionamento dell'architettura

L'esecuzione dell'eseguibile denominato farm porterà all'avvio dei vari processi e dei vari thread necessari per lo svolgimento della consegna. Nello specifico l'esecuzione partirà con l'avvio di un singolo thread: il main thread del processo MasterWorker.

Processo MasterWorker

Una volta avviato il **main thread** del processo MasterWorker questo si occuperà come prima cosa di bloccare tutti i segnali che dovranno essere gestiti dal signal handler thread e di iniziare ad ignorare SIGPIPE, poiché la sua gestione di default porterebbe solo ad interruzioni forzate non desiderate.
In questo modo d'ora in poi saremo sicuri che nessuno dei segnali citati sia fonte di disturbo all'interno del main thread.
Dopodichè controllerà che vi sia un numero sufficiente di argomenti passati da tastiera, dato che non è prevista un'esecuzione senza argomenti, e nel caso in cui siano sufficienti eseguirà la fork. A questo punto il processo figlio si accorgerà di essere tale ed eseguirà la funzione apposita per lavorare come Collector, mentre il padre continuerà l'esecuzione del main.
I prossimi passi del main saranno l'apertura (in funzione di server, dato che sarà lui ad eseguire la listen) del canale di comunicazione col Collector e l'installazione di una funzione di cleanup nel caso in cui le successive chiamate di sistema falliscano e richiedano una terminazione immediata. Fino ad ora infatti eventuali errori portavano semplicemente ad una terminazione preventiva, ma dopo aver aperto la connessione è buona norma chiuderla, attendere la terminazione del collector ed eliminare i vari collegamenti al socket prima di terminare l'esecuzione.
A questo punto non resta che effettuare il parsing degli argomenti inseriti da tastiera, avviare il **thread SignalHandler**, avviare il **thread Master** e poi attendere la sua terminazione (join).
Una volta avvenuto ciò si procederà con la terminazione del signal handler e la liberazione della memoria con la cleanup. La conclusione di tali operazioni corrisponderà con la terminazione effettiva di tutto il programma.

Thread SignalHandler

Tutti i thread (ma anche i processi) avviati dal main thread erediteranno la sua gestione dei segnali, dunque provvederanno ad ignorarli o mascherarli.
Il SignalHandler è l'unico che invece rimarrà appositamente in attesa per gestirli. Per fare ciò dovrà come prima cosa avere il set dei segnali da ascoltare e a quel punto userà uno while e una sigwait per rimanere in attesa che alcuni di essi vengano bloccati.
Una volta che un segnale sarà bloccato userà uno switch per distinguerli e provvederà a gestirli.
Nello specifico:

- Nel caso di **SIGINT**, **SIGQUIT**, **SIGHUP** e **SIGTERM** setterà una variabile globale (termination), leggibile anche dagli altri thread, che indicherà al thread Master di smettere di inserire file nella coda.
- Nel caso si **SIGUSR1** sfrutterà il canale di comunicazione per inviare al Collector un intero speciale (-1) che il Collector saprà interpretare come richiesta di stampa dei risultati ottenuti fino a quel momento e risponderà con 1 o 0 a seconda che la richiesta sia stata esaudita o meno.

Thread Master

Il thread Master riceverà dal main thread che lo ha avviato gli argomenti passati da tastiera dall'utente e dovrà provvedere per prima cosa alla creazione del **threadpool**, il quale avrà n worker e una coda di task pendenti di dimensione q.
A quel punto pusherà una funzione di cleanup che verrà usata per terminare il threadpool nel caso in cui il thread Master avesse problemi ad inserire un task nella coda e si trovasse a dover terminare improvvisamente con una pthread_exit.
A questo punto una volta avviato il threadpool non resta che inserire nella coda dei task pendenti le coppie funzione da eseguire (chiamata Worker) e nome del file su cui eseguirla: per quelli di cui si possiede direttamente il nome basterà ciclcare all'interno di argv, mentre nel caso in cui sia stato passato il nome di una directory usando -d bisognerà ricorrere ad una funzione ricorsiva che esplorerà la directory e quelle sottostanti fino ad avere individuato tutti i file regolari presenti al loro interno.
Degna di nota è una funzione apposita che prima di inserire un task nella coda si assicura di attendere l'intervallo di tempo richiesto tra un inserimento e l'altro e che non sia stata settata a 1 la variabile termination (il Master infatti smette di inserire file nella coda solamente se li finisce o se **termination** diventa 1).
Dopo aver concluso tutti gli inserimenti nella **coda concorrente dei task da elaborare** attende in fine la conclusione del threadpool e poi termina.

Thread Worker

La funzione eseguita materialmente dagli worker appartenenti al threadpool si chiama **workerpool_thread**, ed è contenuta nel threadpool.c. Si occupa di leggere ed eseguire i task che vengono inseriti nella coda fino a che non viene richiesta la terminazione del threadpool.
Ciò che a noi interessa maggiormente è la funzione che viene passata come task da far eseguire a tali thread, e si tratta della **funzione Worker** contenuta dentro Worker.c.
L'esecuzione di un task Worker consiste nell'apertura del file binario il cui nome è stato passato come argomento, e nella lettura dei long contenuti al suo interno per poter calcolare il risultato relativo a tale file.
Una volta fatto ciò si occupa di acquisire la lock su una mutex per garantire l'accesso esclusivo alla connessione per comunicare col Collector, e di inviare lunghezza del nome del file, nome del file e risultato.
Dopodiché attende la risposta del server, che può essere 1 o 0 a seconda che il file sia stato memorizzato correttamente dal Collector o meno. L'ultimo step consiste nel rilasciare la lock sulla mutex.
Importante citare anche la macro **SYSCALL_CLEANUP_RETURN** usata per fare le chiamate writen e readn in sicurezza. Infatti nel caso in cui le chiamate di sistema dovessero fallire la macro chiamerà un'apposita funzione di cleanup e farà una return per terminare l'esecuzione della funzione..

Processo Collector

Il processo Collector sarà avviato dal main thread del processo MasterWorker e per prima cosa aprirà la connessione per comunicare con quest'ultimo. Lo farà in veste di client, ripetendo ciclicamente ogni 200 millisecondi la chiamata connect fino a che il processo MasterWorker non accetterà la sua richiesta.
A questo punto installerà una funzione di cleanup per fare in modo che l'uscita a seguito di chiamate di sistema fallimentari non lasci memoria allocata o il file farm.sck, e dopo aver fatto ciò inizierà a leggere i messaggi che riceve dal processo **MasterWorker**.
L'ordine di lettura è: **lunghezza** del nome del file, **nome** file e **risultato** associato a tale file.
La ricezione dei normali valori per un file ne causa il suo inserimento in coda e l'invio di 1 nel caso in cui tale inserimento sia andato a buon fine (cosa non scontata dato che richiede una malloc).
Valori particolari per quanto riguarda il primo intero comportano azioni diverse:

- La lettura di **EOF** infatti causerà l'uscita dal ciclo di letture e scritture e porterà alla terminazione del Collector dopo aver stampato in modo ordinato i risultati ottenuti fino a quel momento.
- La lettura di **-1** invece indica una richiesta speciale per stampare i valori ottenuti fino a quel momento (sempre in modo ordinato). E dopo aver fatto ciò il collector risponderà con 1 o 0 se tale richiesta è andata a buon fine o meno (anche questa volta cosa non scontata dato che è richiesta una malloc) prima di ritornare al normale ciclo richiesta-risposta.

Strutture utilizzate

Per adempiere alle richieste è stato necessario utilizzare due strutture esterne aggiuntive.

Threadpool

Una delle strutture fornita durante le ore di laboratorio è proprio il threadpool.
Permette, come già anticipato, l'esecuzione multithreading di task (coppie funzione - argomento) attraverso il loro inserimento all'interno di una coda di task. Questo inserimento non può essere fatto liberamente, ma solo attraverso un'apposita funzione **addToThreadPool** che permette di inserire all'interno della coda dopo aver controllato che ciò sia possibile per questioni di spazio e dopo avere acquisito le apposite lock. Inoltre, tale funzione deve svegliare tramite segnali gli worker che sono in attesa dell'arrivo di un task.
Il threadpool si prestava perfettamente per il problema richiesto, tuttavia c'era un comportamento non desiderato: provare ad inserire nella coda dei task quando questa era piena causava un respingimento della richiesta. Normalmente tale comportamento è desiderabile, ma in questo caso ho preferito cambiarlo rendendo la addToThreadPool una chiamata bloccante, la quale attende che si liberi posto nella coda.
Per fare ciò è stato necessario aggiungere una **condition variable** sulla quale un thread potesse rimanere in attesa, e fare in modo che i thread worker del threadpool inviassero un segnale ad uno dei thread in attesa nel momento in cui si è liberato un posto nella coda.

List

Il processo Collector necessita la memorizzazione dei contenuti ricevuti da parte del processo MasterWorker e per fare ciò ha bisogno di una struttura dati apposita. Le opzioni erano diverse tra hashTable, liste ordinate, array che cambiavano la dimensione con realloc… Tuttavia la soluzione che ho deciso di implementare è una **lista non ordinata**. Il fatto che la lista non sia ordinata fornisce un vantaggio in termini di prestazioni, infatti dato che gli worker lavorano in parallelo il Collector rischia di diventare il collo di bottiglia. Usando una lista con inserimento in testa la complessità di tale operazione è costante (O(1)), e permette di aumentare l'efficienza durante lo svolgimento del protocollo richiesta - risposta.
La stampa ordinata consiste invece nell'istanziazione di un array di puntatori ai nodi che sarà ordinato usando **qsort** prima di essere stampato. Potrebbe sembrare inefficiente, tuttavia bisogna considerare che un ordinamento con quickSort ha complessità O(n logn), mentre n inserimenti in una lista ordinata $O(n^2)$.

Schema dei threads attivi

Durante l'esecuzione coesisteranno più threads distinti (i quali accederanno concorrentemente a risorse globali condivise) quindi è importante tener conto di quali e quanti siano attivi nello stesso momento:

MasterWorker	1. Avvio del thread main del processo MasterWorker a. Avvio del processo Collector b. Avvio del thread SignalHandler c. Avvio del thread Master i. Avvio di n thread Worker ii. Join di n thread Worker d. Join del thread Master e. Join del SignalHandler f. Wait del processo Collector
Collector	Il collector non necessita di uno schema poiché si tratta di un processo single-threaded

Gestione della concorrenza

Le risorse condivise tra i vari thread del processo MasterWorker che rischiano accessi concorrenti sono la variabile globale **termination** (controllata dal Master ma anche modificabile da SignalHandler e Worker) e il canale di **comunicazione** col Collector (infatti invii contemporanei in ordine sparso da parte di più worker comprometterebbero il protocollo richiesta - risposta). Per gestire la concorrenza entrambe vengono lette e modificate solo dopo avere acquisito la lock sulle apposite **mutex**, e grazie alle funzioni di cleanup è sicuro che prima di terminare verranno sempre rilasciate le varie lock per evitare situazioni di deadlock.

Istruzioni per l'esecuzione del progetto

All'interno della directory è presente un makefile che consente di generare agevolmente gli eseguibili e di lanciare lo script bash per testare. L'ordine per l'esecuzione del test è il seguente:

1. **make** Per la creazioni degli eseguibili farm e generafile
2. **make test** Lancia il file bash.sh, il quale notificherà le corrette esecuzioni
3. **make cleantest** Per rimuovere i file creati con generafile
4. (**make cleanall**) Se si desidera rimuovere anche i file oggetto e gli eseguibili

Test di terminazioni indesiderate

L'esecuzione con Valgrind e l'invio dei vari segnali non costituisce un problema poiché la memoria viene sempre tutta liberata e farm.sck sempre eliminato. Inoltre controllando anche con ps -A si può avere la conferma che entrambi i processi termino correttamente senza rimanere in background.
Caso particolare è quando si interrompe forzatamente uno dei due processi usando SIGKILL per simulare una cancellazione improvvisa di uno dei due. Seppur non sia un problema aggirabile o ignorabile il codice è scritto in modo da potersi accorgere di tale eventualità per terminare senza troppe complicazioni:

- **SIGKILL** al MasterWorker: Il Collector noterà l'impossibilità di comunicare con l'altro processo tramite i risultati delle chiamate di sistema e provvederà a stampare (ordinatamente) i file ottenuti fino a quel momento, ad eliminare farm.sck, a liberare la memoria e a terminare regolarmente.
- **SIGKILL** al Collector: Uno Worker noterà l'impossibilità di comunicare con l'altro processo tramite i risultati delle chiamate di sistema e provvederà a fermare il Master, velocizzando la conclusione standard del MasterWorker (con liberamento di memoria allocata ed eliminazione di farm.sck).

Entrambe le eventualità sono state verificate con Valgrind e controllando la terminazione con ps -A.