Ginevra Maoro 581700

PROGETTO PER IL CORSO “SISTEMI OPERATIVI E LABORATORIO”

ANNO 2022-2023

DESCRIZIONE GENERALE

“farm” è un programma composto da due processi: il primo, che sarà il processo padre, è denominato MasterWorker, il secondo, il figlio, è denominato Collector.

Il MasterWorker è un processo multi-threaded composto da un thread Master (quello che corrisponde al singolo thread che nasce con la creazione del processo MasterWorker) e da ‘n’ thread Worker.

Il thread Master legge gli argomenti passati da linea di comando che sono una lista (eventualmente vuota se viene passata l’opzione ‘-d’) di file binari contenenti numeri interi lunghi ed un certo numero di argomenti opzionali

Il processo MasterWorker e il Collector comunicano attraverso una connessione socket del tipo AF_UNIX. È stato scelto di usare una sola connessione tra i due processi, alla quale accedono in mutua esclusione i thread Worker. In questa connessione il processo Collector ricopre il ruolo di Server mentre il MasterWorker quello di Client.

Quando il thread Master legge i file da input o all’interno della directory passata da input, dopo aver controllato che siano file regolari, inserisce il path all’interno di una coda condivisa a capacità limitata implementata come buffer circolare. I thread Worker e il thread Master accederanno uno alla volta alla coda condivisa facendo uso di mutex per avere un accesso in mutua esclusione, inoltre verranno usate variabili di condizione in modo da non fare attesa attiva quando il buffer è pieno o vuoto.

Una volta che il thread Worker è riuscito a estrarre dalla coda condivisa il nome o il path del file, ne leggerà dal disco il contenuto ed effettuerà il calcolo sui numeri lunghi al suo interno. A questo punto andrà a scrivere, in mutua esclusione rispetto agli altri thread Worker sul socket (facendo uso di un mutex associato al file descrittore della connessione socket). La prima cosa che scriverà è la lunghezza del nome del file, poi il nome del file e infine il risultato del calcolo associato a quel file. Finite queste operazioni il thread Worker proverà a leggere un altro elemento dalla coda condivisa.

Il processo Collector invece leggerà i dati dal socket, il primo dato (lunghezza del nome del file) serve a capire quanti byte dovrà leggere per recuperare il nome del file dal socket, poi legge il nome del file e infine la terza lettura serve per il risultato associato al file. Il Collector man mano che legge “nome-risultato” lo va a inserire in una lista in maniera ordinata rispetto al risultato. Una volta che avrà letto tutti i file dal socket il Collector stampa la lista ordinata

Argomenti opzionali

- -n <nthread>: specifica il numero di thread Worker del processo MasterWorker (oltre al thread base “thread Master” che nasce col processo) se non viene passata questa opzione il valore di default è 4.
- -q <lunghezza-coda >: specifica la lunghezza della coda concorrente tra il thread Master e i thread Worker, se non viene passata questa opzione il valore di default è 8.
- -d <nome-directory> specifica il nome della directory in cui sono presenti altri file da prendere in considerazione come file di input o eventuali altre sottodirectory da scandire alla ricerca di file binari.
- -t <delay> specifica il tempo in millisecondi che intercorre tra l’invio di due richieste successive ai thread Worker da parte del thread Master, se non viene passata questa opzione il valore di default del delay è 0 ms.

Il processo MasterWorker deve inoltre gestire i segnali: SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGUSR1. Alla ricezione di SIGUSR1 il processo MasterWorker setterà a 1, sfruttando un signal-handler, un flag di stampa condiviso con i thread Worker, in modo che ogni worker prima di andare a inserire il file che ha appena letto dalla coda condivisa controlla questo flag e se è a 1 scrive sul socket la stringa “stampa” e successivamente il nome del file che aveva preso in carico. Il Collector quando leggerà dal socket la

stringa “stampa” effettuerà la stampa della lista ordinata che ha costruito fino a quel momento. La ricezione del segnale SIGUR1 non fa terminare il programma.

Alla ricezione degli altri segnali il MasterWorker setterà a 1, sfruttando un signal-handler, un falg di stop che andrà a controllare ogni volta che leggerà un file da input in modo che se il flag è a 1 non inserisce il nome del file nella coda concorrente, impedendo così che venga elaborato dai thread Worker. Questo perché voglio che alla ricezione di questi segnali, il processo MasterWorker completi i task presenti nella coda condivisa, senza però leggere altri file da input e terminare quando il Collector avrà stampato la lista ordinata dei file letti fino al momento prima della ricezione del segnale. Il processo Collector maschera tutti i segnali gestiti dal MasterWorker. Il segnale SIGPIPE viene ignorato dal MasterWorker e questa gestione viene ereditata anche dal Collector.

SUDDIVISIONE IN FILE DEL PROGETTO

- **main.c:** all’interno di questo file come prima cosa vengono bloccati tutti i segnali che voglio gestire, così da poter installare in modo “safe” i signal-handler per i segnali che deve gestire il processo MasterWorker. Dopo di che viene effettuata una fork() per creare il processo figlio Collector. Il padre eseguirà il codice contenuto nel file Master_Thread.c mentre il figlio eseguirà il codice contenuto nel file Collector.c.
- **Master_Thread.c**: si occupa di fare il parsing della linea di comando, riconoscendo le opzioni. Crea il socket lato Client per la comunicazione con il Collector. Crea il pool di thread Worker e la coda concorrente a capacità limitata. Naviga l’eventuale directory passata come argomento ed inserisce i file che trova al suo interno nella coda concorrente. Prima di navigare la directory controlla che lo sia effettivamente con la funzione ”controllo_directory” e poi la naviga ricorsivamente alla ricerca di file e altre sotto-directory, man mano che trova file regolari (controllati con la funzione “controllo_file_regolare”) li inserisce nella coda concorrente. Poi controlla che i file passati da linea di comando siano file regolari e inserisce anche essi in coda concorrente.
  Una volta che ho inserito tutti i file in coda, scrivo in essa tante stringhe “finish” (che fungono da EOS: end of stream), quanti sono i thread Worker, i quali, leggendo tale stringa, termineranno finendo in uno stato zombie, dal quale verranno revocati grazie alla funzione “pthrad_join”. Una volta fatta la join di tutti i thread terminati vado a scrivere sul socket, senza il bisogno della mutua esclusione perché ormai il thread Master è rimasto l’unico che può scrivere sul socket (i thread Worker non esistono più), la stringa “finish” che quando verrà letta dal Collector, farà capire a quest’ultimo che non deve più leggere dal socket e che quindi può stampare la lista ordinata di tutti i file. Infine, il thread Master può chiudere la connessione socket.
- **Boundedqueue.c**: contiene il codice che implementa la coda concorrente. Questa coda è stata implementata come buffer circolare, di tipo FIFO, con accesso concorrente che segue il paradigma “Produttore-Consumatore” dove il thread Master ha il ruolo di produttore e i thread Worker hanno quello dei consumatori. La coda viene implementata con una struttura dati che contiene:
  1. Un buffer/array di ‘q’ posizioni.
  2. Una variabile che memorizza l’indice dell’array, in cui è contenuta la testa/inizio della coda, dal quale verranno estratti i dati.
  3. Una variabile che memorizza l’indice dell’array, in cui è contenuta la fine della coda, dal quale verranno inseriti i dati.
  4. Una variabile che memorizza la lunghezza attuale della coda (quanti elementi ho inserito fino a quel momento).
  5. La capacità ‘q’ dell’array, in modo da capire se la coda è piena o vuota confrontando gli ultimi due ultimi valori citati.
  6. Un mutex per garantire l’accesso in mutua esclusione alla coda.
  7. Due variabili di condizione: una per indicare quando la coda è piena e quindi il produttore evita di fare attesa attiva per inserire i dati, l’altra per indicare quando la coda è vuota e quindi i consumatori evitano di fare attesa attiva quando la coda è vuota.

- **Worker_consumer.c**: contiene il codice che viene eseguito da ogni thread Worker. Come prima cosa maschero tutti i segnali che vengono gestiti dal thread Master e che quindi non devono essere gestiti dai thread Worker. Il singolo Worker estrae, sempre in mutua esclusione e facendo uso di una variabile di condizione che mi indica quando la coda è vuota, evitando busy wait, il nome del file dalla coda concorrente. Apre il file e fa il calcolo sui numeri lunghi contenuti al suo interno, a questo punto avendo nome del file e risultato va a scrivere in mutua esclusione rispetto a gli altri worker sul socket i 3 parametri: lunghezza nome del file, nome del file e risultato. Prima di scrivere questi tre valori però controlla che il flag di stampa (che viene settato quando il master thread riceve il segnale SIGUSR1) sia a 1, se lo è va a scrivere sul socket la striga "stampa" in modo che il Collector quando la leggerà stamperà la lista dei file ricevuti fino a quel momento. Controllo il flag di stampa in questo punto, all'interno di una zona di codice a cui accedo in mutua esclusione, perché così il flag è acceduto/modificato (riportato a 0 dopo che ho scritto "stampa" sul socket) da un thread Worker alla volta e non ho rischio di interferenza.
- **Collector.c**: contiene il codice che corrisponde al processo figlio Collector. Come prima cosa maschera tutti i segnali che vengono gestiti dal processo MasterWorker eccetto il segnale SIGPIPE, la cui gestione (viene ignorato) la eredita dal padre MasterWorker, e il segnale SIGUSR2. Per il segnale SIGUSR2 viene installato un signal-handler che setta il flag "sig_sigur2" che se è settato a 1 fa terminare prematuramente il Collector (vedi dopo a cosa serve SIGUSR2). Crea il socket lato Server per la comunicazione con il MasterWorker, legge tutti i file e il loro risultato dal socket e li inserisce in una lista in maniera ordinata secondo il valore del risultato. Il Collector quando legge dal socket controlla se ha letto la stringa "stampa", in caso affermativo allora effettua una stampa della lista ordinata che ha costruito fino a quel momento, continuando poi le letture da socket. Il Collector capirà di dover smettere di leggere dal socket quando legge la stringa "finish", a questo punto chiuderà la connessione e stamperà la lista finale.
- **lista_ordinata.c**: contiene le funzioni per inserire l'elemento <risultato, nome file> in modo ordinato nella lista bidirezionale, per stamparla e per liberarla.

ALCUNE PRECISAZIONI

Il MasterWorker può terminare in maniera anticipata se è avvenuto un errore in fase di esecuzione di una system call oppure se riceve uno dei segnali SIGINT, SIGQUIT, SIGHUP, SIGTERM, la cui ricezione è segnata dal settaggio del flag sig_stop_flag/stopflag a 1. In questi casi viene usata la funzione "chiudo_worker" (in Master_thread.c) che si occupa di scrivere le stringhe "finish" per i thread worker, di fare la join di quest'ultimi, di liberare le strutture dati allocate e di chiudere il socket. In questa situazioni di terminazione anticipata la funzione "Master_thread.c" ritornerà nella funzione "main.c" con valore -1 che farà attivare la chiamata di sistema "kill(pid,SIGUSR2)" che invierà al processo Collector il segnale SIGUSR2 per far terminare in maniera anticipata anche lui.

Una delle cause della terminazione anticipata del processo MasterWorker è quando per esempio passo da linea di comando il nome di un file o di una directory che non esiste.

MAKEFILE

All'interno del Makefile il target "farm" serve alla compilazione dell'intero progetto, il target "generafile" serve alla compilazione di "generafile.c" infine ho il target "test" che esegue lo sript "test.sh"

Posizionarsi nella cartella "nome della cartella" e digitare da linea di comando:

1. make farm
2. make generafile
3. make test