PROGETTO SOL FARM2 – a.a 23/24
studente: Natalia Ceccarini
matricola: 598537

Descrizione generale del progetto
Il programma farm2 consiste in due processi: uno denominato MasterWorker , il processo padre, e l'altro Collector, il processo figlio. Questi due processi comunicano tramite un file socket di tipo AF_UNIXfd AF_LOCAL.

Il MasterWorker ha il compito di gestire un pool di thread per l'esecuzione di task. I task consistono nell'eseguire operazioni su un insieme di file regolari presenti in una directory specificata. Il MasterWorker inserisce i task in una coda concorrente e li elabora in modo graduale, prelevandoli dalla coda concorrente e inviando i risultati al Collector.

Per configurare il threadpool, il MasterWorker accetta una serie di opzioni da linea di comando. Queste opzioni includono:

- `-n`: specifica il numero di thread nel pool;
- `-q`: specifica la dimensione massima della coda dei task;
- `-d`: specifica il percorso della directory radice in cui cercare i file;
- `-t`: specifica il tempo in millisecondi tra l'inserimento di un task e il successivo.

Il programma può inoltre accettare altri file da cercare nella directory corrente.

Il threadpool è una struttura che gestisce n thread worker che possono aumentare e diminuire dinamicamente, i thread worker a cui il master worker sottomette i task lavorano su una struttura dati condivisa la coda dei task in cui ci sono salvati una funzione di computazione da efettuare sui dati del file, il pathname del file e il file descriptor del socket così da poter inviare il risultato della computazione al collector che lo inserirà in una lista ordinata contenente i risultati di altre computazioni effettuate dagli altri thread worker su altri file e la stamperà ogni secondo della sua esecuzione.

Quando tutti i task sono stati completati o quando viene ricevuto un segnale di terminazione, viene invocata la funzione destroy del threadpool. Questa funzione attende il completamento di tutti i task pendenti, nel caso ce ne siano, e l'uscita dei thread worker. Una volta che tutti i task sono stati eseguiti e i thread sono terminati, la funzione destroy procede al rilascio di tutte le risorse e della memoria allocata dal threadpool, garantendo una pulizia completa delle risorse utilizzate durante l'esecuzione.

Dopo la distruzione del ThreadPool, viene inviato un codice di terminazione al Collector per notificarlo che deve terminare il processo. La comunicazione avviene tra il MasterWorker e il Collector attraverso una connessione stabilita dal thread main del Master verso il Collector.

Il Collector resta in ascolto utilizzando la funzione `select()` per monitorare i file descriptor pronti e accettare le varie connessioni dai client. In questo contesto, il client è il MasterWorker. Il Collector è il server.

Il processo MasterWorker si duplica utilizzando la funzione `fork()`, generando un processo figlio che viene immediatamente sostituito con una `exec()` per avviare il codice che implementa il processo Collector. Quest'ultimo agisce come un server nella comunicazione, in ascolto di richieste di connessione.

Il thread masterMain del processo MasterWorker e i vari thread worker del threadpool cooperano attraverso una coda concorrente di task da elaborare. Questa coda è di dimensione limitata: il thread masterMain vi inserisce i task man mano che esamina l'input, mentre i thread del pool li prelevano. Questo meccanismo è

sincronizzato attraverso un meccanismo di locking che garantisce l'accesso in mutua esclusione alle strutture dati condivise.

Inoltre, sono utilizzate due variabili di condizione: `full`, su cui il produttore si sospende se la coda è piena, e isEmpty, su cui i vari thread consumatori si sospendono in attesa che la coda non sia vuota e ci siano task da elaborare. Tale approccio implementa un pattern di cooperazione di tipo produttore/consumatore.

Ogni volta che un thread completa l'esecuzione di un task, che consiste in una computazione sui dati contenuti in un file regolare binario, comunica al Collector il risultato del calcolo, insieme al percorso completo del file e la lunghezza del percorso. Poiché le comunicazioni relative ai diversi task elaborati viaggiano sulla stessa connessione, tutte le scritture sul socket sono protette da mutua esclusione per garantire l'integrità dei dati.

Descrizione del masterWorker

Prima di verificare gli argomenti passati dalla linea di comando, il MasterWorker crea un socket e ottiene il relativo file descriptor. successivamente, si collega al socket del server utilizzando il nome esportato con la bind.

Come abbiamo detto il MasterWorker prende argomenti passati da linea di comando, poiché il nome dei file passati non sono considerati argomenti opzionali, si sfrutta l'implementazione GNU della funzione getopt per riordinare argv in modo da disporre le opzioni con i relativi valori all'inizio degli argomenti passati a linea di comando, mentre gli argomenti non opzionali sono collocati alla fine.

Con un ciclo scorriamo il vettore da optind ad argc-1, nel quale si va a verificare che argv[optind] è un file regolare, se lo è lo aggiungiamo alla lista dei task nel threadpool. Essendo la struttura acceduta concorrentemente dal thread produttore e dai worker thread, è protetta da un accesso in mutua esclusione attraverso una lock.

Si utilizzando le condition variables per gestire correttamente la sincronizzazione, in particolare se la coda è piena e non c'è spazio per inserire un nuovo task, il produttore si sospende sulla condition variable coda piena, in attesa di essere svegliato da un worker con una signal non appena viene prelevato un task dalla coda.

Se è stata specificata l'opzione -d con il nome di una cartella, viene chiamata la funzioni find che esplora ricorsivamente il sottoalbero del file system con radice in quella cartella, cercando file regolari da sottomettere al threadpool.

Con l'opzione -t è possibile definire un ritardo in millisecondi che indica l'intervallo di tempo che deve passare tra una sottomissione di task e la successiva.

Threadpool

La struttura è definita con i seguenti campi:

- Puntatore alla testa e alla coda della lista dei task.
- Puntatore alla testa e alla coda della lista dei tid (identificatori) dei thread.
- Numero totale dei thread.
- Tid del thread manager che gestisce l'incremento e il decremento dei threads
- Numero dei thread ancora attivi.
- Dimensione della lista dei task.
- Numero dei task presenti nella lista dei task.
- Contatori usati come parametri nelle funzioni di aggiunta e rimozione dei threads

- Contatore di rimozione dei thread che viene decrementato ogni volta che si esce dalla funzione eseguita dal thread
- Due variabili di lock.
- Tre variabili di condizione (una per svegliare i worker, una per svegliare il produttore e una per svegliare il manager)
- Variabile di uscita

Dopo aver letto e controllato gli argomenti opzionali passati dalla linea di comando, chiamiamo la funzione che alloca e inizializza la struttura threadpool. In questa struttura, vengono aggiunti un numero di thread worker pari all'argomento specificato con l'opzione -n. Questi thread worker rimangono in attesa di un nuovo task nella lista.

Inoltre, viene creato un thread manager per la gestione dinamica del numero dei thread, il quale rimane in attesa finché i contatori dei segnali SIGUSR1 e SIGUSR2 non diventano maggiori di 0. Una volta che ciò accade, il thread manager aggiunge o rimuove thread worker dal thread pool in base al contatore specifico.

La funzione `addJobToThreadpool` prova ad inserire un task nella coda. Se la coda non è piena, il task viene aggiunto in coda, il numero di job nella coda aumenta e viene inviato una signal. Se un thread worker è in attesa su una lista vuota, si sveglierà, preleverà il task dalla testa della coda, lo eseguirà e decrementerà il numero dei task nella coda. Inoltre, può inviare un segnale per svegliare un eventuale produttore in attesa su coda piena. È importante notare che l'inserimento e il prelievo dei task avvengono in mutua esclusione per garantire l'integrità dei dati nella coda.

La funzione addThreadsToPool permette l'aggiunta di n thread al pool.

La funzione `removeThreadFromPool` consente la rimozione di n thread dal threadpool. Modifica il contatore `removeThreads`, impostandolo uguale al parametro num passato. Viene inviato una signal in modo che se c'è un worker che sta dormendo, venga svegliato poiché `removeThreads` è maggiore di 0. Successivamente, il worker esce dalla funzione decrementando il contatore `removeThreads`.

La funzione `destroyPool` gestisce la terminazione dei worker, la terminazione del thread manager, l'esecuzione delle join, la distuzione delle variabili di lock e delle variabili di condizione. Successivamente, libera la memoria di tutte le strutture dati utilizzate dal threadpool.

Gestione segnali
Inizialmente, i segnali SIGINT, SIGTERM, SIGHUP, SIQQUIT, SIGPIPE, SIGUSR1 e SIGUSR2 vengono mascherati nella maschera dei bit. Per il segnale SIGPIPE, viene installato un gestore che lo ignora (SIG_IGN). La maschera dei bit sarà ereditata dal processo Collector quando sarà eseguita l'exec.

Dopo la fork nel processo padre, viene utilizzato un thread handler come gestore dei segnali. Questo thread continua a eseguire sigwait, e ogni volta che ritorna gestisce il segnale corrispondente. Se uno tra i segnali SIGINT, SIGTERM, SIGHUP o SIQQUIT arriva, il programma deve terminare, pertanto si imposta una variabile di terminazione a true. Questa variabile sarà poi controllata dal thread MasterMain per terminare non appena settata, evitando di sottomettere nuovi task al pool se in fase di terminazione.

Se arriva il segnale SIGUSR1, viene tentato di acquisire la lock sulla variabile contatore add, in modo da inviare una signal al thread manager. Quest'ultimo chiamerà addThreadsToPool, aggiungendo altri n thread worker al threadpool e reimpostando il contatore a 0.

Anche se arriva SIGUSR2, viene tentato di acquisire la lock sulla variabile contatore remove, in modo da inviare una signal al thread manager. Quest'ultimo chiamerà la funzione removeThreadsFromPool per svegliare un thread worker, per farlo terminare. Se il contatore `removeThreads` è maggiore di 1, quando gli altri thread worker osserveranno che `pool->removeThreads > 0`, termineranno finché il contatore non ritorna a 0.

È importante notare che il contatore che incremento quando arriva il segnale (ad esempio, `pool->ctrRmoveThreads`) non deve essere confuso con il contatore utilizzato come variabile di condizione su cui un thread worker può svegliarsi dall'attesa (ad esempio, `pool->removeThreads`).

Il primo contatore è utilizzato per gestire l'arrivo dei segnali e per avere una gestione il più veloce possibile, utilizzando una variabile di lock diversa da quella che usano i worker per non rimanere troppo in attesa nell'acquisizione della lock. Il secondo contatore è utilizzato per sincronizzare i thread worker e farli svegliare quando le condizioni specifiche sono soddisfatte (ad esempio, quando pool→removeThreads è maggiore di zero).

Terminazione del processo

Il processo termina in diverse circostanze:

1. Quando sono stati sottomessi tutti i task da eseguire.
2. Quando viene ricevuto uno dei segnali: SIGINT, SIGTERM, SIGQUIT, SIGHUP.
3. Quando si verifica un errore in una chiamata di funzione, come ad esempio l'esaurimento della memoria durante una chiamata a malloc.

Alla terminazione del programma, la funzione atexit chiama la funzione exitHandler. Quest'ultima distrugge il threadPool utilizzando la funzione destroyPool, la quale restituisce il numero di thread ancora attivi. Viene inviato un messaggio di terminazione al collector e il numero di thread restituito da destroyPool viene scritto sul file nworkeratexit.txt. Infine, viene cancellato il file socket.

Processo Collector

Il processo Collector agisce come un server, creando un file "ricevitore" (socket) e allocando un file descriptor ad esso associato con l'indirizzo AF_UNIX. Successivamente, esegue una listen, diventando pronto ad accettare nuove connessioni. Entra in un ciclo while, in attesa che arrivi un messaggio di terminazione.

Durante l'attesa, utilizza la funzione select, la quale rimane in attesa finché uno dei descrittori specificati è pronto. Se il file descriptor del socket è pronto, il processo Collector può accettare nuove richieste di connessione tramite la funzione accept, la quale restituisce al server il descrittore del nuovo socket utilizzabile per il resto della comunicazione con il client.

Quando è stabilita una connessione, il processo Collector legge i messaggi inviati dal MasterWorker. Se legge un messaggio di terminazione, esce dal ciclo ed esegue il processo di terminazione. Altrimenti, inserisce i messaggi (risultato del calcolo, nome file) in una lista ordinata in ordine crescente in base al risultato del calcolo che stampa ogni secondo durante la sua esecuzione.

Compilazione ed esecuzione del progetto

Per compilare ed eseguire il progetto, è stato previsto l'uso di un makefile. Ecco i passaggi da seguire:

1. Posizionarsi nella cartella del progetto.

2. Eseguire il comando `make` per compilare il codice e generare i file oggetto.
3. Per eseguire il file di test test.sh, utilizzare il comando `make test`.
4. È stato incluso un set di test nel file my_test.sh, che è possibile eseguire con il comando `make mytest`.
5. Per eseguire il test con le impostazioni predefinite, utilizzare `make exec`.
6. Se si desidera testare con Valgrind, eseguire `make valg`.
7. Per ripulire la cartella dai file generati digitare make clean
8. Per riportare la cartella allo stato iniziale digitare make cleanall