Farm 2

Sandro Montesu

Indice

1 Introduzione e struttura del progetto

Il progetto "**farm2**" è strutturato in due processi: **MasterWorker** e **Collector**, che agiscono rispettivamente come client e server, comunicando attraverso una connessione socket di tipo **AF_UNIX**. All'avvio, il processo MasterWorker esegue una chiamata a **fork**, creando il processo Collector e continuando la propria esecuzione. MasterWorker è responsabile della creazione e dell'inizializzazione dei thread necessari per la comunicazione tra i due processi. Questi thread, chiamati "**worker**", processano i file presenti in una coda e li inviano al server.

Il processo Collector, a sua volta, riceve queste informazioni e le stampa periodicamente in modo ordinato.

Il processo MasterWorker è composto da otto file, ciascuno con le relative librerie per l'esportazione delle funzioni:

- **main.c**: si occupa della gestione dei segnali e della creazione del processo Collector tramite la chiamata a fork.
- **MasterThread.c**: si occupa del **parsing** degli argomenti opzionali, e invoca funzioni definite in altri file per gestire la creazione e l'inizializzazione dei thread, oltre all'apertura della connessione socket.
- **Worker.c**: contiene l'implementazione dei thread Worker.
- **TaskQueue.c**: include tutte le funzioni legate alla gestione della coda utilizzata nel programma. .
- **DirectryManager.c**: include tutte le funzioni necessarie per gestire l'interazione con le directory. Se viene specificato l'argomento opzionale -d, è possibile passare una cartella come argomento, e questo file si occupa di gestire tutte le operazioni correlate a tale cartella.
- **Socket.c**: contiene tutte le funzioni relative alla gestione dei socket, tra cui la connessione al server e l'invio dei dati. Questo file è condiviso tra i due processi, anche se ciascun processo utilizza funzioni diverse al suo interno.
- **ThreadPoolWorker.c**: definisce il thread pool, una struttura dati che gestisce tutte le operazioni relative ai thread worker e alle strutture dati associate (come la coda dei task). Questo file si occupa della creazione, terminazione (in casi speciali, poiché i thread sono di tipo detached), e della deallocazione delle strutture dati utilizzate dai worker.
- **Utils.c**: contiene funzioni di utilità generale utilizzate in diverse parti del progetto. L'header di questo file include macro necessarie per la corretta gestione delle variabili di lock. Il file è condiviso tra i processi.

Il processo Collector è suddiviso in quattro file, di cui due condivisi con il processo MasterWorker, come già menzionato.

- **Collector.c**: gestisce i segnali, richiama le funzioni definite in **Socket.c** per creare il server e ricevere le informazioni dal client. Inoltre, crea un thread dedicato alla stampa, che utilizza una struttura ad albero per visualizzare i dati in modo ordinato.
- **Tree.c**: contiene tutte le funzioni relative alla gestione dell'albero, inclusa le funzione di stampa, che viene passata al thread di stampa per organizzare e visualizzare le informazioni.

Il progetto è stato testato su Ubuntu 24.04 LTS e su Xubuntu 14.10

2 Come funziona il progetto

L'utente può passare una lista di file come input al programma. Questi file indicano dei file binari che contengono numeri in formato long, che verranno elaborati dai thread worker e inviati al server insieme al nome del file. È inoltre possibile specificare parametri opzionali come la lunghezza della coda, il numero di thread worker, l'intervallo di tempo tra l'inserimento di un file e l'altro nella coda, e una o più cartelle contenenti altri file da processare.

All'avvio, il programma crea un processo figlio responsabile della creazione di un server locale per ricevere i dati elaborati. Il processo padre, nel frattempo, tenterà per 10 volte consecutive, con intervalli di 1 secondo tra un tentativo e l'altro, di connettersi al server. Dopo aver eseguito il parsing degli argomenti opzionali, il processo padre avvia i worker e inserisce i file nella coda tramite funzioni definite in altri file.

Il main thread attenderà la terminazione dei worker utilizzando una condition wait. Una volta risvegliato, invierà al server un messaggio speciale per segnalare la fine dell'elaborazione, scriverà in un file il numero di thread attivi al momento della terminazione, eliminerà il socket e terminerà l'esecuzione.

Il processo figlio, invece, attenderà in un ciclo infinito di ricevere i dati elaborati e, tramite un thread dedicato, stamperà periodicamente i dati ricevuti fino a quel momento. Uscirà dal ciclo solo al ricevimento del messaggio di fine, a quel punto eseguirà un'ultima stampa prima di terminare.

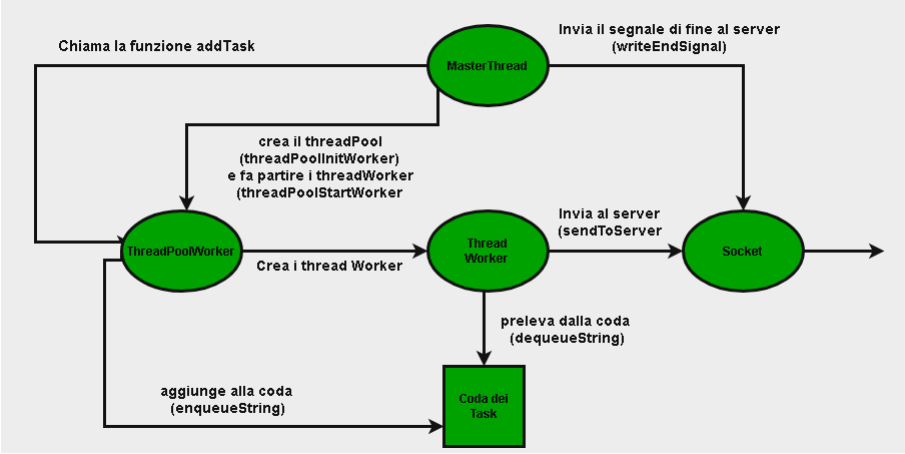
2.1 Gestione dei segnali

Il processo padre gestisce i segnali utilizzando la funzione **sigwait**, in combinazione con un thread dedicato che li gestisce in ciclo. Questo thread utilizza opportune lock per garantire una corretta gestione anche in caso di ricezione di più segnali in rapida successione.

I segnali vengono suddivisi in tre categorie principali:

- **SIGHUP SIGTERM SIGINT SIGQUIT** : una volta ricevuto uno di questi segnali si attende che i worker completino l'elaborazione dei file già estratti dalla coda prima di terminare il programma.
- **SIGUSR1** : viene interpretato come un comando per aggiungere un nuovo thread al pool. Questo avviene tramite una **realloc** dell'array dei thread, incrementando la dimensione del pool.
- **SIGUSR2** alla ricezione di tale segnale si deve gestire la terminazione di un thread. Tuttavia, questa terminazione **non avviene tramite** operazioni come **pthread_cancel**. Invece, il thread stesso si autogestisce, monitorando dei flag e contatori che indicano quando è il momento di terminare, garantendo una terminazione ordinata e controllata.

Di seguito un'immagine rappresentante una struttura semplificata del processo MasterWorker:



3 MasterWorker

Il funzionamento del processo MasterWorker è suddiviso tra i vari file che lo compongono di seguito verranno citati i file principali:

3.1 main.c

Il file main svolge due funzioni principali:

1. **Creazione del processo figlio:** Attraverso la chiamata a fork, viene creato un nuovo processo che esegue la funzione Collector, responsabile della gestione del lato server della comunicazione. Il processo padre, oltre a inizializzare le strutture per il signal handler, avvia la funzione MasterThread, passando gli argomenti argc e argv ricevuti in input. MasterThread gestisce la logica principale del programma, richiamando le funzioni necessarie per la gestione dei thread worker. Una volta conclusa l'esecuzione di MasterThread, il processo padre si occupa di rimuovere il socket utilizzato per la comunicazione tra client e server, assicurando una chiusura corretta e pulita, e attende la terminazione del processo figlio.
2. **Gestione dei segnali:** Il processo padre gestisce i segnali utilizzando la funzione sigwait e un thread dedicato all'esecuzione del signal handler. I segnali che devono essere gestiti sono i seguenti: **SIGHUP SIGTERM SIGINT SIGQUIT SIGUSR1 SIGUSR2**

   Inoltre, il segnale **SIGPIPE** viene ignorato per evitare che il processo termini nel caso in cui si tenti di scrivere su un socket chiuso.
3. **Comportamento alla ricezione dei segnali:**
   - **SIGUSR1**: Quando viene ricevuto, viene impostato il flag **THREADFLAG1**, indicando che il numero di thread worker deve essere aumentato.
   - **SIGUSR2**: Quando viene ricevuto, viene impostato il flag **THREADFLAG2**, segnalando che il numero di thread worker deve essere diminuito.

- **SIGHUP, SIGTERM, SIGINT, SIGQUIT**: Alla ricezione di uno di questi segnali, viene impostato il flag **terminate**, il quale informa i thread di iniziare la procedura di terminazione. I thread completeranno la loro esecuzione in modo controllato, inviando eventuali dati in sospeso prima di concludere.

I flag impostati, ad eccezione di **terminate**, verranno gestiti in una funzione chiamata **handlerFlag**, eseguita all'interno di un lock. Questa funzione svolge le seguenti operazioni:

Se il flag **THREADFLAG1** è stato impostato, l'array dei thread verrà riallocato e verrà creato un nuovo thread worker in modalità detached. Se il flag **THREADFLAG2** è stato impostato, il contatore **countWorkerTerminate**, inizialmente uguale a zero, verrà incrementato all'interno della lock. Questo contatore dovrà essere decrementato dal thread che termina, e verrà inoltre risvegliato ogni thread bloccato su una condition wait tramite la chiamata a **pthread_cond_broadcast**.

3.2 MasterThread.c

Il file MasterThread ha diverse responsabilità:

3.2.1 Parsing degli argomenti:

L'utente può specificare alcuni argomenti opzionali, che vengono analizzati tramite la funzione getopt. Eventuali errori nella loro specifica vengono gestiti attraverso la funzione **safeStrtol**, presente nel file **Utils.c**, che esegue la conversione tramite strtol e controlla la presenza di eventuali errori.

I parametri che possono essere specificati sono i seguenti:

- Se viene passato l'argomento **-n**, l'utente deve indicare il numero di thread worker da creare.
- Se viene passato l'argomento **-q**, l'utente deve specificare la capacità massima della coda **TaskQueue**.
- Se viene passato l'argomento **-t**, l'utente deve indicare un intervallo di tempo (in millisecondi) che deve intercorrere tra l'inserimento di un elemento e l'altro nella coda.
- Se viene passato l'argomento **-d**, l'utente deve specificare una cartella contenente file binari o altre cartelle. Tramite una funzione **countFile**, vengono contati i file presenti nella cartella e il conteggio viene accumulato in una variabile chiamata **sum** per tenere traccia del numero totale di elementi da processare. Successivamente, la cartella viene aggiunta a un array allocato dinamicamente chiamato **folderFlag**.

3.2.2 Connessione al server:

La connessione al server viene gestita tramite una chiamata alla funzione **connectToServer**, presente nel file **Socket.c**. Come già accennato, il processo padre tenta di stabilire la connessione per un massimo di 10 volte consecutive. Durante ciascun tentativo, viene inserita una pausa di 1 secondo per consentire una riconnessione in caso di esito negativo, garantendo così una gestione efficace delle operazioni di connessione.

3.2.3 Gestione dei thread:

MasterThread richiama le funzioni presenti nel file **threadpool.c**, come **threadpoolInitWorker**, per inizializzare i worker e le relative strutture dati, passando anche la lunghezza della coda, che può essere modificata dall'utente tramite l'opzione -q. Viene quindi impostato un flag chiamato **signalFlag**, il quale ha lo scopo di evitare che i segnali ricevuti e gestiti influenzino la coda e i thread che non sono ancora operativi. Se il flag non è impostato, la gestione dei segnali è disabilitata.

Successivamente, MasterThread avvia i thread worker tramite la funzione **ThreadpoolStartWorker**. Utilizzando la funzione **AddTask**, e passando argc, argv e l'array di cartelle, vengono aggiunti gli elementi alla coda condivisa, permettendo così l'elaborazione dei file da parte dei worker.

3.2.4 Terminazione

Dopo aver aggiunto tutti gli elementi alla coda (o, in caso il flag di terminazione sia stato impostato, facendo terminare prima **AddTask**), MasterThread si mette in attesa della conclusione dei worker tramite una chiamata a **pthread_cond_wait**. Questa condizione viene monitorata in un ciclo che verifica sia il numero di worker attivi sia un flag denominato **atLeastWorkerHadStarted**, il quale ha lo scopo di garantire che MasterThread non arrivi a questo punto del programma prima che i worker siano effettivamente stati avviati. In questo modo, si evita qualsiasi accesso non sincronizzato alle risorse condivise durante la fase di avvio dei worker.

Una volta uscito dal ciclo, MasterWorker reimposta il flag **signalFlag** e invia al server un messaggio di terminazione tramite la funzione **writeEndSignal**, presente in **Socket.c**. Successivamente, procede alla deallocazione di tutte le risorse allocate dinamicamente.

Prima di terminare definitivamente, MasterThread scrive in un file il numero di thread attivi al momento della conclusione del programma.

3.3 Worker.c

il file Worker.c contiene l'implementazione dei thread:

3.3.1 Thread Worker

I thread worker, una volta creati, incrementano un contatore condiviso che indica il numero di worker attivi contemporaneamente e impostano il flag **atLeastWorkerHadStarted**. Questi thread sono progettati per elaborare un numero limitato di file, specificato dalla variabile **NUMBER_OF_ELEMENTS** dichiarata nel MasterThread. Ogni thread worker continua a processare i file fino a quando non viene impostato il flag terminate o non ci sono più file da elaborare.

Per ottenere il nome del file da processare, il thread worker utilizza la funzione **dequeueString**, dichiarata nel file **TaskQueue.c**.

Una volta ottenuto il nome del file, il thread worker verifica se la funzione ha modificato un flag chiamato **exitResult**. Se tale flag è impostato, indica la terminazione del thread, che pertanto richiama la funzione **decreaseWorker**, passando il proprio **PID**. Altrimenti, il thread apre il file in modalità lettura e calcola una sommatoria utilizzando i dati contenuti nel file. La formula della sommatoria è: **result** $= \sum_{N=0}^{N-1}(i * file[i])$ dove N è il numero di interi (di tipo **long**) presenti nel file e i è l'indice dell'intero corrente. Questo calcolo viene effettuato per ciascun intero presente nel file.

Dopo aver calcolato la sommatoria, il nome del file e il risultato corrispondente vengono inviati al server tramite la funzione **sendToServer**, presente in **Socket.c**. Infine, il thread incrementa un contatore che indica il numero degli elementi processati e verifica se sono stati elaborati tutti gli elementi. In tal caso, il thread imposta il flag terminate e risveglia gli altri thread con una chiamata a **pthread_cond_broadcast.**

Prima di terminare, ogni thread decrementa il contatore dei thread attivi e l'ultimo thread a terminare risveglia il MasterThread con una chiamata a **pthread_cond_signal**. Questa sezione, che include l'incremento e decremento dei contatori dei thread attivi, il controllo del numero di elementi e l'invio dei risultati al server, è eseguita in mutua esclusione tramite l'utilizzo di opportune lock, garantendo così la sincronizzazione corretta nell'accesso alle risorse condivise.

3.4 TaskQueue.c

Il file TaskQueue.c contiene tutte le funzioni necessarie per la gestione della coda utilizzata nel progetto. Di seguito, una descrizione delle principali funzioni:

- **queueInit**: inizializza la coda dei task, allocando la memoria necessaria per il campo dati, impostando la capacità massima e inizializzando le mutex e le variabili di condizione necessarie per la sincronizzazione.
- **queueDestroy**: dealloca tutte le risorse associate alla coda, inclusa la memoria e gli eventuali meccanismi di sincronizzazione.
- **dequeueString**: prima di estrarre un elemento dalla coda, verifica se ci sono thread da terminare (controllando se il contatore **countWorkerTerminate** è maggiore di 0). In tal caso, imposta il flag **exitResult** e restituisce **NULL**, segnalando la necessità di terminare il thread. Se ci sono elementi disponibili e il flag terminate non è impostato, la funzione restituisce il primo elemento della coda. Se la coda è vuota, il thread worker attende sulla variabile di condizione **notEmpty**.
- **enqueueString**: inserisce un nuovo elemento nella coda. Se la coda è piena, il thread chiamante viene messo in attesa sulla variabile di condizione **notFull**. Se il flag terminate è impostato, la funzione termina l'inserimento. Se l'utente ha specificato un intervallo di tempo tra un inserimento e l'altro (tramite il parametro opzionale -t), la funzione attende per il tempo indicato utilizzando **nanosleep**.

3.4.1 Sincronizzazione

L'accesso alla coda è regolato da meccanismi di sincronizzazione basati su mutex e condition variables. Quando la coda è piena, il thread principale (responsabile dell'inserimento degli elementi) viene messo in attesa finché non si libera spazio. Allo stesso modo, quando la coda è vuota, i thread worker che tentano di estrarre elementi vengono bloccati finché non vengono inseriti nuovi dati.

Per evitare che i worker rimangano bloccati nelle variabili di condizione durante la fase di terminazione del programma, ogni ciclo di controllo delle condizioni include una verifica del flag terminate. Quando questo flag viene impostato, tutti i thread bloccati vengono risvegliati tramite una chiamata a **pthread_cond_broadcast**, consentendo loro di uscire dalle attese e di terminare in modo sicuro.

3.5 ThreadPoolWorker

Il file threadpool.c gestisce tutte le operazioni relative al thread pool attraverso la struttura dati **ThreadPool**, che contiene: Un puntatore all'array di thread, Un puntatore alla coda dei task, Il numero di thread attivi.

Le principali funzionalità fornite da threadpool.c includono l'inizializzazione del thread pool, l'avvio dei thread in modalità **detached**, e la deallocazione delle risorse associate. È anche responsabile della terminazione controllata dei thread in caso di

ricezione del segnale **SIGUSR2**. nello specifico la funzione **decreaseWorker** gestisce la terminazione controllata di un thread worker quando viene ricevuto tale segnale. La funzione prende in input il **PID** del thread worker da terminare e lo cerca all'interno dell'array di thread. Una volta trovato, salva l'indice corrispondente, sostituisce il thread da terminare con l'ultimo thread presente nell'array e aggiorna il numero di thread attivi. Infine, l'array di thread viene riallocato con una dimensione ridotta, in modo da escludere il thread che deve terminare. Questo metodo permette di rimuovere un thread dal pool senza interruzioni o corruzione delle strutture dati, garantendo una terminazione sicura del thread.

3.6 DirectoryManager.c

Il file DirectoryManager gestisce tutte le operazioni relative alle cartelle, come l'esplorazione ricorsiva e il conteggio degli elementi presenti. Le sue funzioni consentono di esplorare il filesystem, individuare file e cartelle all'interno di una directory e calcolare il numero di file che dovranno essere processati; inoltre per ogni file trovato si occupa di richiamare la funzione **enqueueString** e aggiungere quindi il file alla coda.

3.7 Socket.c (lato client)

Il file Socket.c contiene una serie di funzioni utilizzate sia dal client che dal server per gestire la comunicazione tramite socket. Lato Client, le funzioni usate permettono di connettersi al server tramite la funzione **connectToServer** e di inviare i dati elaborati dai Worker, infine è presente la funzione **writeEndSignal** che invierà al server il messaggio speciale -1 che verrà interpretato come un segnale di terminazione.

4 Collector

Il processo Collector è implementato nei file **Collector.c, Tree.c** e utilizza funzionalità fornite da **Socket.c** e **Utils.c**.

4.1 Collector.c

Il file Collector.c viene eseguito dal processo figlio, creato tramite la chiamata a **fork**. Questo processo gestisce la creazione di un server locale mediante socket **AF _UNIX**, utilizzando le funzioni fornite da **Socket.c** per ricevere i file dal client. I file ricevuti vengono inseriti in una struttura ad albero, gestita dalle funzioni presenti in **Tree.c**. Un thread dedicato visita l'albero periodicamente, stampandone il contenuto ogni secondo.

Alla ricezione del segnale di terminazione, il processo chiude la connessioni e dealloca le risorse utilizzate prima di terminare l'esecuzione.

4.2 Tree.c

Il file Tree.c contiene tutte le funzioni necessarie per la gestione dell'albero (allocazione e deallocazione, inserimento in maniera ordinata e funzioni di stampa usate dal thread)

4.3 Socket.c (lato Server)

Il file Socket.c gestisce la creazione di un server utilizzando un socket di tipo **AF_UNIX**, permettendo di creare un server locale e gestendo la ricezione dei file inviati dal client. Alla ricezione del messaggio speciale -1 il server verrà terminato.

4.4 Utils.c

Il file Utils.c, insieme al suo header **Utils.h**, fornisce funzioni e macro di supporto per la gestione degli errori in diverse operazioni critiche quali malloc, utilizzo di lock... tale file è condiviso tra Client e Server

5 Compilazione

Il progetto include un Makefile che facilita la gestione della compilazione e la pulizia dei file generati. Il Makefile fornisce tre comandi principali:

5.1 Compilazione (make):

Il comando make compila tutti i file sorgente necessari per il progetto. Utilizza le regole definite nel Makefile per gestire le dipendenze tra i file sorgente e gli oggetti, producendo l'eseguibile finale.

5.2 Pulizia (make clean):

Con il comando make clean, tutti i file generati dalla compilazione, come gli eseguibili e i file oggetto (.o), vengono rimossi dal progetto, insieme a eventuali file temporanei. Questo è utile per ripulire l'ambiente e prepararlo per una nuova compilazione.

5.3 Test (make test):

Il comando make test esegue uno script di test chiamato test.sh, il quale esegue automaticamente una serie di verifiche sul progetto per assicurarsi che tutte le funzionalità implementate funzionino correttamente.