Università di Pisa
Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Progetto di Sistemi Operativi e Laboratorio – Farm2 2023/24
Docente Massimo Torquati

Studente Baldini Enrico, Matricola 597780

Introduzione
Farm2, il programma sviluppato nel corso del progetto, consiste in un programma multiprocessore con multi-threading. Il progetto, scritto in lingua inglese con commenti in italiano, è stato sviluppato per sistemi UNIX su una macchina virtuale Linux Ubuntu 64 bit versione 22.04 LTS, la quale ha due processori a disposizione.
Il contenuto del file .zip che include la presente relazione è disponibile anche alla pagina Github https://github.com/Baldo-University/SOL_Farm2

Struttura del progetto
I file che compongono il progetto sono raccolti in diverse cartelle sulla base del ruolo che svolgono.
La directory root del progetto contiene un *makefile* che permette di semplificare i passaggi di compilazione e test del codice, oltre che la pulizia dei file temporanei creati durante l'elaborazione. Le istruzioni sull'utilizzo del *makefile* sono, oltre che descritte nella presente relazione, presenti all'interno del file *README.md*. Inoltre è all'interno della root directory che viene stampato il file *nworkeratexit.txt*, nel quale è segnato il numero di thread attivi al termine dell'esecuzione di Farm2.
I file in formato .c del codice sorgente sono contenuti all'interno della directory *./src*. All'interno di quest'ultima si trova la sottodirectory *./headers*, la quale contiene file in formato .h con le informazioni essenziali su costanti e struct utilizzate dai file .c.
La directory test contiene al proprio interno i file forniti insieme alla documentazione del progetto, ovvero *generafile.c* e *test.sh*.
Infine la directory *./doc* contiene, oltre che la presente relazione, il file PDF contenente le linee progettuali stabilite dal docente.
Il codice sorgente è composto dai seguenti file .c e .h:

- *main.c*, ovvero il main principale con cui eseguire il Farm2.
- *masterworker.c*, che contiene il funzionamento del thread master di Farm2 e la gestione dei segnali nel processo a cui master appartiene.
- *pool.c*, contiene le funzioni offerte dal threadpool e la funzione lanciata dai singoli worker.
- *workfun.c*, che svolge la sommatoria delineata nella documentazione del progetto.
- *collector.c*, che contiene il codice utilizzato dal processo collector.

Ad eccezione di *main.c* e *collector.c*, ogni file sorgente ha un proprio header che altri file possono includere per avere accesso alle funzioni che essi forniscono.

- *masterworker.h*, possiede la firma della funzione masterworker e le costanti standard del programma.
- *pool.h*, contiene la struct del threadpool e le firme delle funzioni per l'utilizzo dello stesso.
- *workfun.h*, ha solo la firma della funzione centrale di *workfun.c*.
- *utils.h*, viene incluso in tutti i file sorgente poiché contiene le definizioni di macro per la massima lunghezza dei filepath, per il test di controllo di funzioni cruciali e per la stampa di debug.
- *message.h*, viene incluso da *pool.c* e *collector.c* per via della sua struct message, la quale stabilisce il formato dei messaggi scritti dai thread worker e letti dal processo collector.

Istruzioni per l'uso

Le istruzioni rapide per l'esecuzione del programma Farm2 su Linux sono le seguenti.
In seguito al download del file .zip contenente il progetto, aprire un terminale sulla root del progetto.
Per compilare il programma, eseguire il comando "*make*".
Per effettuare il test delineato da test.sh, eseguire "*make test*".
Per pulire i file temporanei creati durante i due passaggi precedenti, eseguire "*make clean*".
Se al posto di "*make*" si esegue "*make debug*", il programma verrà eseguito in versione debug, ovvero facendo stampe su *stderr* per tenere traccia di quale componente stia eseguendo quale azione.

Funzionamento nel dettaglio

Il programma viene lanciato mediante l'eseguibile "farm" insieme ad una serie di argomenti ed eventuali opzioni. Il file "*main.c*", che contiene il main del suddetto eseguibile, controlla innanzitutto che sia stato passato almeno un argomento oltre che al nome dell'eseguibile. Se tale condizione è rispettata, *main.c* imposta la mask dei segnali intercettati dal programma; la mask è inizialmente "completa", ovvero blocca tutti i segnali ad eccezione di SIGKILL e SIGSTOP. Dopodiché viene invocata la system call *fork()*, dove in seguito a un controllo di errore di quest'ultima funzione, viene creato un processo figlio che, mediante *execl()*, esegue "*collector*". Il processo padre ripristina la maschera presente al momento del lancio e invoca la funzione *masterworker()* descritta in *masterworker.c*, passando argc, argv e il nome della socket come argomenti.
Il primo compito di *masterworker()* è stabilire la mask dei segnali descritta nelle linee progettuali assegnate. Viene così creata una sigset_t, che funge da argomento del pthread_t sighandler_thread, il quale intercetta i segnali inviati al processo in maniera asincrona; per tale thread si è optato per una forma detached al fine di evitare di gestirne la conclusione con una *pthread_join()*.
Successivamente, il thread master svolge lo studio delle opzioni passate mediante chiamate ripetute di *getopt()*. Sono così inizializzate le variabili che indicano il numero iniziale di thread nel threapool (mediante -n), la lunghezza della coda di produzione (-q) e il ritardo di inserimento dei file in quest'ultima (-t). Se tali variabili non sono impostate da opzioni passate, il programma utilizzerà quelle stabilite nell'header di masterworker. Oltre alle suddette opzioni, master usa l'opzione -d per stilare una linked list di directory contenenti file binari da elaborare.
Completata la gestione delle opzioni, masterworker invoca la funzione initialize_pool() offerta da *pool.h,* passando come argomento il numero iniziale di thread, la lunghezza della coda e il nome della socket. Tale funzione, in seguito ad un sanity check, alloca memoria alla struct di tipo threadpool_t che mantiene al proprio interno le informazioni necessarie al funzionamento del threadpool; di particolare importanza sono la lista di thread worker, quella che svolge il ruolo di coda di produzione, le variabili, i mutex e le condition variables delle liste.

Una volta allocata e inizializzata la memoria del threadpool, *initialize_pool()* chiama n volte la funzione add_worker() di pool.c, dove n è il numero di thread iniziali. La funzione *add_worker()* inserisce nella lista di worker un nuovo thread, il quale viene subito lanciato e inizializzato, permettendogli di cominciare subito ad elaborare file. Il funzionamento dei worker sarà descritto in seguito, dopo quello del master. Dopo aver verificato di aver creato almeno un worker, *initialize_pool()* restituisce al master un puntatore alla struct di threadpool creata.
Giunto a questo punto, master utilizza la variabile di ritardo per inizializzare una struct timespec con cui misurare i tempi di attesa per l'inserimento dei file nella coda.
Da questo punto in poi fino alla propria terminazione, il master si comporterà principalmente come produttore in un paradigma produttore-consumatore, fornendo ai worker consumatori i file da elaborare. Master passa in rassegna prima gli argomenti passati direttamente da linea di comando e successivamente, in maniera ricorsiva, le directory passate con -d, alla ricerca di file binari; verificato che ogni file così esaminato sia effettivamente un file binario, invoca per ciascuno di essi la funzione *enqueue_task()*, con threadpool e nome del file come argomenti. Tale funzione, descritta in pool.c, alloca memoria per un nuovo elemento della coda di produzione, assegnandogli il nome passato e inserendolo in fondo alla coda. Tale funzione si blocca con *pthread_cond_wait()* in caso la coda risulti piena e invoca *pthread_cond_signal()* per indicare a un worker la disponibilità di un task.
Prima di ogni inserimento, master controlla se il processo abbia intercettato segnali SIGUSR1 o SIGUSR2. Tali segnali sono gestiti dal *sighandler_thread*, il quale utilizza una variabile globale *thread_num_change* che funge da contatore dei suddetti segnali. Se il contatore segnala un valore diverso da zero, master invoca la funzione *add_worker()* o *remove_worker()*: il primo in caso di valore positivo, il secondo in caso di valore negativo. Tale operazione viene ripetuta in un loop che ad ogni passaggio riduce di 1 il valore assoluto del contatore, fino ad azzerarlo.
La funzione *remove_worker()* si limita a aumentare di uno la variabile intera *remove_threads* del pool, ma solo se la variabile *num_threads*, che mantiene il conto dei worker attivi, è maggiore stretta di uno; ricordiamo infatti che il numero di thread nel pool non può scendere sotto ad uno durante il funzionamento del pool; sarà poi un worker a verificare la variabile *remove_threads* e a terminare il proprio funzionamento.
Esauriti i file da elaborare, o in caso *sighandler_thread* abbia ricevuto SIGTERM, SIGQUIT, SIGHUP o SIGINT, master smette di produrre file in coda, eventualmente chiudendo directory aperte con *readdir() e* liberando la lista di directory memorizzate con -d. È a questo punto che viene invocata la funzione *await_pool_completion()* presente in *pool.c*.
La funzione *await_pool_completion()* innanzitutto salva in una variabile il numero di worker presenti e attivi nel threadpool. Dopo di ciò, essa inserisce nella coda di produzione un task "finale", contraddistinto dagli altri task di coda per la presenza del flag *endtask* settato a 1. Tale task, letto dai worker, è il segnale per questi ultimi di uscire dal loop di esecuzione dei propri compiti. *await_pool_completion()* invoca una *pthread_cond_broadcast()* per avvertire tutti i worker della presenza del task finale ed entra in un while-loop che invoca *pthread_join()* su ciascun worker e ne libera uno ad uno la memoria. La funzione distrugge quindi i mutex e le condition variables del threadpool, libera la memoria allocata dinamicamente per il pool e il nome della variabile socket che esso possiede e infine restituisce al master il numero di thread attivi salvato all'inizio della funzione.
Ricevuto il numero di thread, master crea il file *nworkeratexit.txt*, che contiene solamente tale valore. Master poi invoca *remove()* per rimuovere dal filesystem il file socket creato per stabilire le connessioni client-server. Infine, master invia con *pthread_kill()* il segnale SIGQUIT al *sighandler_thread* per farlo terminare, per poi chiudere a sua volta la funzione masterworker().

I thread worker svolgono la funzione *thread_func()*, descritta in *pool.c*, prendendo come argomento il threadpool che li ha creati. Ogni worker possiede un proprio intero identificativo id e un buffer di 264 byte capace di contenere un filepath di massimo 255 caratteri ed un intero long. Inoltre, ogni worker stabilisce una propria connessione locale client-server con il processo collector. Inizializzate le variabili e creata la connessione, il worker entra in un loop di esecuzione. Per prima cosa esso controlla se *remove_threads()* sia maggiore di zero e se ciò è vero il worker esce dal loop. Se invece resta nel loop, worker setta a zero la memoria del proprio buffer e consuma un task dalla coda di produzione. Mediante *strncpy()* il worker salva nel buffer il filepath del file prima di liberare la memoria dedicata al task di cui faceva parte. Tale filepath viene passato come argomento per l'invocazione di *workfun()*.
La funzione *workfun()* svolge la seguente sommatoria, richiesta dai requisiti del progetto:

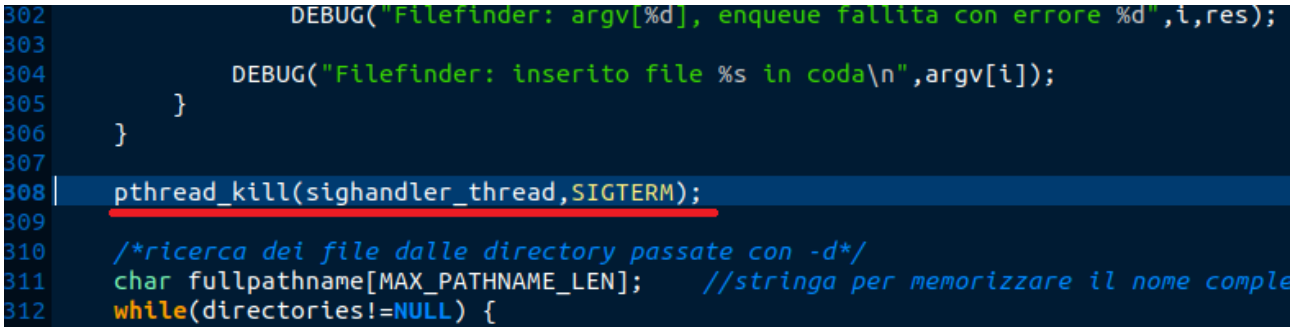
$$\sum_{i=0}^{N-1} (i * file[i])$$

La funzione *workfun()* restituisce al worker il risultato intero della sommatoria come *long int*.
Ottenuto il risultato della sommatoria, il worker usa *memcpy()* per copiare i byte del risultato negli ultimi 8 byte del proprio buffer (ricordiamo che un long in Linux 64 bit occupa appunto 8 byte). Fatto ciò, worker invoca la *write()* per inviare al processo collector il nome del file elaborato e il relativo risultato. A questo punto worker ricomincia il loop di esecuzione; ne uscirà una volta constatato che il task da eseguire è in realtà il task finale inserito da *await_pool_completion()*.
Un worker uscito dal loop termina la connessione ed effettua *pthread_exit()* per concludersi.
Il processo collector viene lanciato con un solo argomento, ovvero la stringa col nome della socket.
La gestione dei segnali del processo *collector* consiste nell'ignorare i segnali che il processo master invece gestisce. Collector dichiara la lista, inizialmente vuota, dei risultati che manterrà durante il proprio funzionamento. Subito dopo esso lancia un thread *printer_thread* che stampa la lista incompleta dei risultati ogni secondo, fino al termine di esecuzione del processo.
In seguito collector crea la socket farm2, la setta a non-blocking con *ioctl()*, per poi eseguire la *bind()* e la *listen()*.
Collector usa un paradigma single-threaded per la gestione delle connessioni client-server stabilite con più client worker. Tale paradigma è implementato con la chiamata di sistema *poll()*, che si appoggia ad un array di *struct pollfd*, il quale viene inizializzato dal collector in seguito alla creazione della socket. Il primo indice dell'array prende come file descriptor quello della socket *farm2.sck* e accetta come eventi solo la ricezione di dati da leggere.
A questo punto dell'esecuzione, il processo collector entra in un loop potenzialmente infinito che gli consente di accettare nuove connessioni client e ricevere i risultati elaborati da questi ultimi. Il loop ha inizio con una chiamata a *poll()*, che indica se siano stati riscontrati eventi sui file descriptor dell'array pollfd. Se così è stato, collector entra in un nested loop che esamina gli indici dell'array contenenti file descriptor delle socket client, alla ricerca di eventi. Eventi sul primo indice significano richieste di connessione da parte di nuovi client, mentre eventi sui restanti indici indicano l'invio di nuovi risultati o disconnessione in caso la lettura non restituisca alcun byte. Uscito dal nested loop, collector "pulisce" l'array di pollfd rimuovendo le connessioni terminate, riordinando gli indici assegnati di conseguenza e aggiornando il numero di connessioni. Se, terminata la pulizia, collector si rende conto di aver solo la socket di *farm2.sck* aperta, il processo esce dal loop infinito.
Infine, il collector chiede la *pthread_join()* del printer_thread, stampa la lista completa di risultati, ne libera la memoria, chiude la socket e termina con successo.

Criticità osservate

Come indicato dal mandato progettuale, ho testato il corretto funzionamento del progetto facendo uso dei file “generafile.c” e della script “test.sh”.
A seguito di ripetute esecuzioni del comando “make test”, è stato osservato che talvolta il primo test non risulta superato a causa di discrepanze con il file “expected.txt”. Eppure, quando il programma viene eseguito al di fuori della batteria di test delineati da test.sh con lo stesso comando eseguito dal primo test, il risultato corrisponde sempre a quello atteso.
Test aggiuntivi svolti con “*valgrind –leak-check=full –show-leak-kinds=all make test*” rivelano che il programma soffre di memory leak. Non essendo riuscito a replicare ogni test al di fuori della script non sono stato in grado di individuare la causa di tali leak.
Non sono riuscito ad implementare completamente il funzionamento del debug. Nella presente struttura il progetto possiede macro per la stampa approfondita dei singoli passaggi svolti dal programma per verificarne la correttezza; tuttavia tali funzioni non sono implementate.
Infine, un limite emerso dalle scelte di progettazione è il rischio che vengano passate una quantità tale di opzioni -d che superino la capacità della memoria per una corretta allocazione ancora prima che il programma possa lanciare il threadpool per studiarne i contenuti.

Aggiornamento
Questa sezione descrive i provvedimenti adottati per risolvere i problemi riscontrati dopo il primo invio del progetto:
- Ho individuato che la versione inviata del progetto conteneva un refuso non rimosso in seguito ad una fase finale di test. Si tratta di una system call pthread_kill che invia SIGTERM al processo master a metà fra l’invio dei file passati da linea di comando e quelli passati da directory -d (vedi figura sottostante)



- Test aggiuntivi rivelano che talvolta il programma entra in deadlock con test/test.sh. Non è stato possibile riprodurre tale deadlock né compilando con “make debug” né con prove che non usassero test.sh
- Il test 1 che fallisce è probabilmente legato al fatto che il collector deve stampare ogni secondo sullo stdout. Quando tale stampa “forzata” viene effettuata sullo stderr infatti, il test viene sempre passato