Progetto Sistemi Operativi UNIPI 2021
Julian Duff

Implementazione server e gestione richieste client:

Il file server e' implementato con un main thread che gestisce le connessioni dei client, creando un socket sul quale verranno accettate nuove connessioni ch sono e gestite attraverso un fd_set e chiamate alla syscall select.
Per gestire le richieste viene utilizzata una threadpool con un numero costante di workers.
Le richieste mandate dai client vengono inserite in una coda protetta da mutex dal MT.
Questa coda permette ai threadworkers di chiamare qualsiasi funzione che abbia un valore di ritorno void* e un parametro void* . Viene usata principalmente per richiedere la lettura di una richiesta client, ma permette anche di svolgere operazioni diverse, come ad esempio l' uscita di tutti i thread.
La presenza di una richiesta nella coda viene segnalata dal MT attraverso una condition signal nel momento in cui viene inserita, ma i WT sono configurati per controllare (sempre in ambiente mutex locked) il contenuto della coda se un determinato lasso di tempo e' trascorso dall'ultima ricezione del segnale, per evitare che un segnale venga perso quando tutti i thread workers sono occupati.
Le richieste client verranno lette dai thread workers in modo tale da prevenire qualsiasi lettura doppia della stessa richiesta. Una richiesta e' definita dal suo valore definito nella enum ReqFunctions, contenuta in API.h. Il valore enum e' passato dall'API al server e utilizzato per accedere alla funzione necessaria per completare la richiesta attraverso un array di puntatori a funzioni definita lato server. Ad esempio, la chiamata di API readFile() mandera al server il valore enum e_fileRead per indicare la richiesta desiderata. Un qualsiasi threadworker leggera' quindi il valore enum della richiesta e successivamente chiamera' la funzione necessaria. Queste funzioni leggeranno quindi dal socket, il cui file descriptor viene passato dal MT ad un WT fino a che la richiesta viene completata. Le informazioni passate saranno il nome che identifica univocamente un file, i suoi contenuti (se idonei alla richiesta) , identificatore client (attraverso il loro PID), o qualsiasi altra informazione utile per la richiesta. Se la richiesta non puo' essere completata, per esempio se viene richiesta la read di un file che non esiste nel server, o la write di un file che esiste gia', viene passato un valore >0 (che equivale al codice ERRNO prodotto dall'operazione) al socket, indicando che l'operazione e' stata annullata, e la funzione API che ha chiamato la richiesta terminera' restituendo -1 e settando ERRNO al valore mandato dal server. Se invece il valore passato e' 0, la richiesta e' valida, e la funzione definita nell'API potra' quindi mandare i dati necessari per completare l'operazione (ad es. write) o ricevere la risposta del server (ad es. read).
I WT indicano al MT se un client socket deve essere nuovamente ascoltato o se non e' piu attivo tramite una pipe (lettura MT scrittura WT).
Il MT e tutti i WT ignorano i segnali SIGINT, SIGQUIT, SIGHUP e SIGPIPE.
SIGINT, SIGQUIT e SIGHUP vengono gestiti da un apposito signal handler thread, il quale avverte il MT tramite pipe se uno di quei segnali e' stato ricevuto.
In base al segnale ricevuto, viene interrotta la ricezione di nuove connessioni client (SIGHUP) o iniziata immediatamente la procedura di chiusura del server.
In caso di segnale SIGHUP, la chiusura del sever viene eseguita quando non ci sono piu' socket client aperti.
Per chiudere il server vengono rimosse tutte le richieste dei client presenti nella coda dei WT e aggiunta una richiesta di terminazione di tutti i WT. Se una richiesta e' in elaborazione da un WT al momento di ricezione di SIGQUIT o SIGINT, essa viene completata prima dell'uscita del thread.
Successivamente alla chiusura di tutti i thread eccetto il MT, Tutta la memoria utilizzata per contenere file e strutture dati di supporto viene liberata.

Stuttura file system:

Un file e' definito dal suo path assoluto, interpretato come stringa. Per ricavare il path assoluto viene utilizzata la funzione realpath(), una richiesta all'API di un file non esistente su disco non e' quindi valida, e viene ignorata.
I file sono organizzati in record, i quali sono contenuti in una tabella hash, in cui sono inseriti in base alla chiave (di tipo unsigned long) generata in base al path assoluto, definito abspath nel codice sorgente del server.

Non e' garantito che file con nome diversi generino sempre la stessa chiave, quindi per identificare un file nella tabella e' utilizzata la chiave per ridurre il numero di ricerche necessarie e abspath per identificare univocamente il file. La tabella e' protetta da mutex per evitare collisioni tra WT.
La memoria del server e' fissa e divisa da un certo numero di pagine (definito dalla dimensione pagina passato per config) ed e' gestita quindi da un array di num pagine elementi , ciascuno di dimesione uguale ad una pagina singola. Per distribuire le pagine tra file viene utilizzata una coda che contiene un elemento per ciascuna pagina non attualmente in uso. Essa e' protetta da mutex per evitare che la stessa pagina venga distribuita a piu' file o altri tipi di comportamenti anomali. Avendo ogni file pagine distinte, e' quindi possibile eseguire piu' operazioni di scrittura o lettura sulla memoria principale del server in contemporanea a patto che vengano eseguite su file diversi.
Per gestire operazioni su file specifici (il cui nome e' passato come argomento) viene utilizzata la tabella, ma operazioni su un certo numero di file (come la readNFiles) o la rimozione di file in caso di raggiungimento capacita' massima nel server (eseguita in ordine FIFO) vengono eseguite attraverso una struttura di supporto, il FileStack.
Esso e' un array dinamico contenente i nomi di tutti i file inseriti nel server (non necessariamente attualmente presenti), dal piu' recente in cima fino al piu' vecchio in fondo. Se un file viene rimosso esplicitamente dal server (chiamando removeFile), esso non e' quindi rimosso dallo stack, per evitare un'operazione potenzialmente costosa.
Ogni accesso a file nel filestack passa quindi attraverso la tabella hash per confermare la presenza del file nel server. Questa procedura permette di eseguire l'eliminazione di file in ordine FIFO a tempo costante.
In base alla frequenza di operazioni removeFile, lo stack potrebbe contenere molti file ormai inesistenti, viene quindi chiamata una operazione di deframmentazione dello stack quando esso raggiunge la sua dimensione massima (definito come una proporzione maggiore di 1 del limite file del server), per evitare un overhead eccessivo su operazione che utilizzano lo stack.
Un operazione di eliminazione file FIFO prende quindi il primo file valido dallo stack, lo elimina (rimuovendo prima il suo record dalla tabella hash) e restituisce le pagine utilizzate dal file alla coda di pagine. Questa operazione viene chiamata ogni volta che viene raggiunto il numero massimo di file nel server o ogni volta che un file non trova pagine libere (ovvero la dimensione massima e' stata raggiunta). Dopo questa chiamata lo stack viene anche parzialmente deframmentato, ovvero la cima della struttura e' impostata alla posizione del file preso, riducendo la frequenza delle chiamate di defrag necessarie.
Le operazioni di lettura e scrittura su file vengono eseguite una pagina per volta.
Il record di un file e' definito dal: nome assoluto (abspath), dimensione file, numero di pagine occupate, valore delle pagine occupate, mutex che protegge il file da operazioni contemporanee da piu' threads (la quale viene bloccata per qualsiasi operazione che legge o scrive file/valori dei record) e lista di client in stato open.

File config :
il file config e' letto dal server prima dell'inizializzazione della memoria per file o la creazione di threads per definire il valore di certi parametri.
Viene chiamata la funzione configGetAll definita in config.c, che esegue il parsing del file passato al server come argomento di linea di comando per settare il valore delle variabili globali definite in config.h, in base al loro token identificativo, definito in configGetAll.
Il file avra una struttura del tipo:
config_variable_str = someword
config_variable_str2 = 5
ecc…
I token sono separati da qualsiasi numero di spazi " " o di "=", compresa una qualsiasi combinazione di essi
Sono valide quindi line del tipo
var_value 4
var_value = 5
var_value == someword
var_value = = = = = 3
Questo permette di formattare il file config in base alle preferenze di chi lo scrive.
Se sono presenti piu' token identici, viene utilizzato il primo valore trovato e ignorati quelli che si trovano piu' avanti nel file.
E' possibile aggiungere un commento alla linea di config utilizzando un ulteriore separatore dopo il valore desiderato, ad esempio:

mem_size = 45 MB | → MB non viene letto dal parser
weird_value = 457295 comment explaining use of weird value | → qualsiasi cosa dopo il secondotoken viene ignorata

Per accedere a variabili condivise tra threads vengono usate le funzioni protette da mutex configReadInt e configReadSizeT.

Scelte implementative dell'API:

openFile/closeFile : l'apertura o chiusura di un file da un client e' definito dal suo PID. Esso viene passato via socket per le operazioni che richiedono che un file sia aperto per terminare con successo (read,write,remove,ecc…). I PID vengono inseriti in una coda di client PID (PID che hanno attualmente il file aperto), contenuta nel file record. La velocita' di verifica che un client abbia un file in modalita' aperta dipende quindi dal numero di client che hanno il file in stato opened, e' quindi preferibile per un programma qualsiasi che utilizzi l'API chiudere un file non appena non ne abbia piu' bisogno.

writeFile : Scritture di file vuoti o sovrascritture a file con contenuto gia' presente sono rifiutate.

readNFile : La lettura di n file viene eseguita creando una copia del FileStack deframmentato, i cui elementi sono messi in ordine casuale. Vengono letti i primi n elementi della copia. Se un file viene eliminato dal server durante l'operazione, esso non viene letto, non e' quindi garantito che la readN legga n file, ma potrebbe restituirne di meno.Questo viene fatto perche' eseguire la lock di tutti i file che richiede la readNFiles rallenterebbe significativamente operazioni come la lettura,open,close,ecc di file singoli.
Il nome del file spedito equivale ad abspath meno qualsiasi riferimento alla directory di provenienza, se un file e' gia presente nella directory passata ad readNFiles esso viene sovrascritto.

Scelte implementative client :

parametro -d dir : operazioni di lettura che chiamano -d devono passare una directory esistente, in caso contrario la funzione API non viene chiamata (la funzione API chiamata con directory non nulla e non valida restituisce errore).

parametro -w dir : se passata una directory inesistente la funzione API chiamata restituisce errore.

parametro -t msec : -t setta un delay a tutte le richieste del client che seguono il parametro -t . ripetere -t equivale a sovrascrivere il valore del delay. Questo permette di variare il tempo di attesa nella stessa lista di richieste client.

Tests:

test1: vengono eseguite ogni operazione supportata dal client su file prodotti dallo script createFiles.sh viene inoltre controllato che i file scritti coincidono con i file letti dal server, che la readNFiles possa leggere ogni file dal server e che sia possibile rimuovere ogni file dal server con -c.

test2: viene eseguita la scrittura di file_num files (definito nello script) al server e parsato l'output del server e dei client per verificare che i file vengano rimossi correttamente e che non siano piu' presenti nel server una volta rimossi.

repository github pubblica : https://github.com/JulianDuff/Progetto_SOL_2021