UNIVERSITÀ DI PISA

Relazione Progetto di Sistemi Operativi e Laboratorio

Leonardo Stoppani

A.A. 2020/21

Introduzione

Il progetto prevede la realizzazione di un **file storage server**, un client e una API per la comunicazione tra client e server. Il server offre tramite API una serie di operazioni che il client utilizza per implementare i comandi della cli fornita all'utente. Le funzioni della API simulano la semantica della libreria di I/O di unix sui file del server, i quali vengono identificati in modo univoco dal loro path assoluto. Client e server comunicano su una sola connessione con il protocollo "richiesta-risposta" specificato piu' avanti.
Al seguente link e' possibile raggiungere la repository github pubblica del progetto :

`https://github.com/lilf4p/file-storage-sol.git`

Server

Architettura Ho implementato il server come un singolo processo multi-threaded secondo lo schema "master-worker". Il file di configurazione puo' essere specificato da linea di comando tramite l'opzione -s "config.txt" e puo' trovarsi in una qualunque directory, se non viene specificato o si presentano errori il server viene avviato con parametri di default. La struttura del file "config.txt" prevede che si possa assegnare il valore val all'attributo att tramite la sintassi "att=val", una coppia per riga, inoltre sono ammesse righe vuote.
I parametri che si possono settare sono il numero di thread del pool (num-thread), il numero massimo di file (max-files), la dimensione in byte massima (max-dim) e il nome del socket (socket-name).

```
1 max_files=10000
2 max_dim=134217728
3 num_thread=1
4 socket_name=./tmp/LSOfilestorage.sk
5
```

Il thread master corrisponde al thread main che sfrutta il multiplexing di canali per poter restare in ascolto di nuove connessioni da parte di client e inviare le richieste dei client ai thread worker. Ho implementato la comunicazione tra master e worker tramite una coda di file descriptor, il thread master inserisce i file descriptor dei client che vengono restituiti dalla select come pronti in lettura, mentre i thread worker li prelevano. Una volta prelevato il file descriptor di un client il worker serve una sua richiesta e lo restituisce al master tramite una pipe a sua volta registrata sulla select. Nella pipe scrivo il file descriptor del client ritornato e un flag per specificare al master se deve registrarlo nuovamente nella select o chiudere la connessione. Ho gestito gli accessi concorrenti dei thread worker con una lock sulla coda e una variabile condizione per evitare attesa attiva quando la coda e' vuota.
I thread worker leggono la richiesta del client e la passano alla funzione **eseguiRichiesta** che interpreta il comando e usa il protocollo di comunicazione con la API per soddisfarlo.

La funzione worker implementa il funzionamento del thread worker, le funzioni insertNode, removeNode, updatemax gestiscono la coda per la comunicazione master-worker.
La funzione gestore-term gestisce la terminazione del server tramite segnali, alla ricezione di SIGINT e SIGQUIT viene settata la variabile globale term a 1, alla ricezione di SIGHUP viene settata a 2. Con il flag a 1 il server termina il prima possibile, con il flag a 2 e il contatore num-client il server gestisce la terminazione soft, termina solo quando il contatore scende a 0, ovvero quando tutti i client connessi si disconnettono. In entrambi i casi il server si assicura di aver liberato la memoria, interrotto i thread e stampato su standard output le statistiche. Ho definito tre macro per la gestione dei fallimenti delle chiamate di sistema e una per le funzioni di libreria pthread. Da citare quella per le system call su socket dove in caso di fallimento comunico al master di chiudere la connessione con il client, sempre utilizzando la pipe e settando il flag fine a -1. Sempre parlando si chiamate di sistema su socket, ho scelto di utilizzare l'implementazione che ci e' stata fornita a lezione di readn e writen (by W. Richard Stevens and Stephen A. Rago, 2013, 3rd Edition, Addison-Wesley).

Cache File Ho implementato la cache di file in memoria del server con una lista di struct file che comprende il path del file (path), il contenuto (data), una lista di file descriptor dei client che hanno aperto il file (client-open), il file descriptor del client che puo' effettuare una writeFile su quel file (client-write).

```
typedef struct file {
    char path[PATH_MAX];
    char * data; //conte
    node * client_open;
    int client_write; //
    struct file * next;
} file;
```

La lista client-open e il flag client-write sono informazioni che mi servono per realizzare la semantica open-write/read-close, client-write viene settato quando viene chiamata la API openFile con flag impostato, viene poi resettato quando lo stesso client esegue una qualsiasi altra operazione.
Ho gestito gli accessi concorrenti dei thread worker con una lock sulla lista che viene acquisita e rilasciata dalle funzioni che operano su di essa. Tra le piu' importanti c'e' la addFile che se chiamata con il flag settato a 1 prova a creare il file aggiungendolo alla lista e impostando il client per la scrittura, se settato a 0 prova ad aprirlo in lettura e scrittura aggiungendo il file descriptor del client alla lista del file identificato da path. La funzione getFile restituisce il conenuto del file identificato da path se esiste e se e' stato aperto dal client, caratteristica individuata grazie alla funzione fileOpen. La funzione writeData scrive il contenuto nel file solamente se client-write e' uguale al file descriptor del client. La removeFile rimuove il nodo dalla lista con il path uguale a quello richiesto. La removeClient rimuove il client dalla lista della struct file con path corrispondente. La funzione freeCache libera tutta la memoria allocta per la cache, scorrendo i nodi file della lista e liberando i vari campi della struct.
Ho scelto una lista come struttura dati per la cache per la sua dinamicita', infatti la dimensione della cache e' sempre uguale alla dimensione totale dei file contenuti (piu' metadati) in quel momento, con 0 file la dimensione e' pari a zero, incrementa ad ogni file scritto e decresce ad ogni file eliminato.
Ho implementato una politica di sostituzione in cache di tipo FIFO, dato che nella lista aggiungo i file nuovi in testa, vado a rimuoverli in coda. Rimuovo nella addFile quando supero il numero massimo di file, mentre chiamo la funzione resize-cache quando nella writeData e appendData supero la dimensione massima. La funzione resize-cache

elimina l'ultimo file della lista finche' non c'e' abbastanza spazio per il nuovo file da scrivere.

Client

Il client offre all'utente un'interfaccia a riga di comando per utilizzare i servizi del server, ho implementato questa interfaccia con getopt e una lista di coppie <cmd,arg> dove cmd e' l'opzione e arg il suo relativo argomento se presente. Per prima cosa faccio il parsing di argv con getopt e inserisco le coppie nella lista con la funzione addLast poi ,una volta finiito il parsing, eseguo prima le opzioni -h,-p,-t,-d,-f se presenti, in questo modo non conta l'ordine in cui l'utente le inserisce. Successivamente passo ad eseguire le opzioni rimaste -w,-W,-r,-R,-c nell'ordine in cui le ha inserite l'utente, anche ripetute. Finite le opzioni libero la memoria e chiudo la connessione con la API closeConnection.
Ho scelto di costruire il path assoluto con la funzione realpath nelle opzioni -w e -W, mentre per le altre opzioni e' il client che lo deve specificare, questo perche' nel primo caso si stanno creando file nuovi sul server, mentre negli altri stiamo richiedeno una sorta di download al server e nel server potrebbero esserci piu' file con path realativo uguale, ma mai con path assoluto uguale.
L'opzione -W per ogni file passato come argomento esegue le API openFile, writeFile e closeFile solo se tutte hanno successo, altimenti se una fallisce le altre non vengono eseguite. L'opzione -w ha un funzionamento simile alla -W ma lo fa per n file della directory chiamando listDir, funzione che esplora ricorsivamente tutta la directory.
L'opzione -r mantiene lo schema open-read-close, mentre l'opzione -R chiama solo la API readNFiles e -c chiama solo removeFile.
In caso di fallimento di una funzione della API il client stampa con perror l'errno opportunamente settato dalla relativa API.
Il flag-stampa viene settato con l'opzione -p e permette di abilitare le stampe su standard output per ogni operazione nel formato *Operazione : op File : f Esito : positivo/negativo*.
La variabile globale statica num-files viene utilizzata nella funzione listDir per limitare il numero di file da inviare nell'opzione -w.
La variabile globale tms contiene il ritardo in millisecondi tra una richiesta al server e l'altra specificato con l'opzione -t, 0 altrimenti.

API

La API permette al client di comunicare on il server, ho scelto di includerla nel client sotto forma di libreria statica **libapi.a** e si trova nella directory lib.
Il flag connesso viene settato quando la openConnection ha successo e indica se il client ha aperto la connessione con il server. La variabile globale sc contiene il file decsriptor del socket a cui il client si e' connesso e viene utilizzato dalle varie funzioni per le readn e writen. Il buffer socket-name contiene il nome del socket con cui e'

stata chiamata la openConnection che ha avuto successo, in questo modo la closeConnection controlla che venga chiamata sullo stesso socket.
La funzione di utility msleep implementa una sleep in millisecondi sfruttando nanosleep.
La funzione di utility compare-time confronta due struct timespec e viene utilizzata per implementare il timer della openConnection, se la connect sul soket fallisce viene riprovata ogni msec specificati fino al raggiungimento del tempo assouto abstime.
La funzione di utility mkdir-p dato un path di directory, se non esiste la crea insieme a tutte le directory intermedie.
Tutte le funzioni della API restituiscono 0 in caso di successo, -1 e settano errno in caso di fallimento, si trasmette l'errore al client. Anche in questo caso ho scelto di utilizzare l'implementazione della readne writen per le chiamate di sistema su socket.
Il protocollo base prevede che le richieste inviate dalla API abbiano la sintassi **<op,arg>** dove per op si intende una delle funzioni della API e arg una lista di argomenti separati da ",", inoltre i messaggi hanno dimensione fissata di DIM-MSG. La risposta del server ha la sintassi **<res[,errno]>** se res vale 0 l'operazione e' andata a buon fine, se res vale -1 l'operazione e' fallita con errore errno, in questo modo le funzioni della API possono settare facilmente errno. Il protocollo base utilizzato per le API openFile, closeFile e removeFile, non e' sufficiente quando e' necessario inviare o ricevere un file, poiche' non si puo' stabilire una dimensione a priori, in questi casi server e API si scambiano una serie di messaggi in modo tale da concordare la dimensione del messsaggio che conterra' il file e allocare lo spazio necessario. Nella writeFile e appendToFile (buffer invece di file) la API a invia un messaggio con comando e path , ricevuta conferma dal server invia la size del file, ricevuta la seconda conferma invia il file e aspetta il risultato dell'operazione dal server. Nella readFile il server dopo aver ricevuto il comando dalla API risponde con la size del file o con errore e dopo la conferma il server invia il file. Nella readNFiles la API invia la richiesta con il numero di file e il server risponde con il numero n di file disponibili dopodiché' ha un funzionamento simile alla readFile ma ripetuto per n volte.

Makefile

Il Makefile che ho realizzato presenta i target **all** di default per costruire gli eseguibili server e client, **clean** per eliminare gli eseguibili, **cleanall** per eliminare tutti i file generati dal Makefile (eseguibili,oggetto,temporanei,librerie,...), **test1** per lanciare il primo test con valgrind, **test2** per lanciare il secondo test. Il target test1 dopo aver avviato il server con valgrind in background e il file di config apposito, lancia lo script bash **test1.sh** che a sua volta lancia un client per testare tutte le opzioni. Il target test2 lancia il server e lo script bash **test2.sh** che lancia 5 client per testare la sostituzione in cache. Entrambi gli script bash aspettano che il server si sia avviato e alla fine inviano il segnale SIGHUP al server.