Progetto di laboratorio di Sistemi Operativi

Gemma Ragadini

1 Server

1.1 Configurazione

La configurazione del server all'avvio viene eseguita grazie alla funzione **Configura** che, leggendo il file di configurazione assegna il valore ad alcune delle variabili globali presenti nella struct **Parametri**: il numero di workers, lo spazio totale dello storage, lo spazio attualmente occupato, il nome della socket, il numero massimo di file che possono essere caricati sullo storage, il numero di file attualmente presenti, le variabili intere *isClosing* e *closed* (una usata per la terminazione immediata del server e una per la terminazione più lenta, come descritto in seguito).

1.2 File di Configurazione

Il file di configurazione è strutturato in modo che su ogni riga ci sia la coppia *nomevariabile* = *valore*. Non è rilevante l'ordine delle righe. Ho scritto due diversi file di configurazione, per i due test. Il server, se non specificato un nome diverso del file da riga di comando, leggerà dal file di configurazione del test1.

1.3 Struttura Dati

Come struttura dati per lo storage dei file ho scelto di utilizzare una coda di struct **File** ciascuna delle quali è composta dal nome del file (che sarà poi il pathname assoluto) , il contenuto del file , la dimensione dello spazio allocato per il contenuto del file (*size*) e il puntatore alla testa della lista degli fd dei client che hanno in quel determinato momento il file aperto. Nel file **file.h**, oltre a ciò che ho appena descritto sono presenti le funzioni per maneggiare la struttura dati (per l'inserimento di un file, l'apertura di un file da parte di un client, la rimozione di un file per liberare spazio, la scrittura e la scrittura in append. In particolare durante l'esecuzione delle funzioni che aggiungono contenuto ai file dello storage si controlla che la memoria residua sia sufficiente a contenerlo e , nel caso ciò non si verificasse , vengono eliminati file dallo storage (secondo la politica FIFO) fino a che non sia soddisfatta la condizione. La funzione che elimina un file dallo storage viene chiamata anche nel caso in cui si fosse raggiunto il numero massimo di file presenti. È presente anche la funzione per l'eliminazione dello storage al momento della chiusura del server.

1.4 Coda dei lavori

I lavori che i worker dovranno effettuare, richiesti dai client, sono rappresentati da una struct **JobElement** che contiene un carattere che indica il tipo di operazione richiesta al server, il pathname del file su cui deve essere effettuata, e l'intero fd della socket per comunicare con quel client. Questi lavori sono organizzati in una coda **JobList** e queste strutture sono contenute nel file **works.h**. **Comunicazione con i client** La comunicazione con i client avviene tramite socket. Il processo principale tramite un ciclo infinito attende nuove connessioni da parte dei client. Ogni volta che si connette un nuovo client viene creato un nuovo elemento di tipo JobElement che avrà assegnato il campo dell'*fd* con l'fd del client ottenuto dalla accept e il tipo di operazione richiesta che sarà ancora sconosciuto e per questo inizializzato con il carattere *X*. A questo punto il lavoro verrà inserito nella JobList e poi da lì estratto dai vari thread workers che, leggeranno dalla socket le informazioni necessarie per eseguire l'operazione richiesta (nel caso che l'operazione sia 'X' leggeranno anche il tipo di operazione da eseguire). Una volta completata la singola operazione e mandate le risposte al client, il worker che la aveva presa in carico leggerà dalla socket la prossima operazione richiesta da quello stesso client (nel caso quella appena eseguita non fosse l'ultima e quindi il client si fosse disconnesso) e inserirà nella lista dei lavori un prossimo lavoro da eseguire, ricominciando il suo ciclo. In questo modo ogni worker porta a termine una singola operazione ma un determinato client non verrrà servito dallo stesso worker per ogni sua richiesta.

1.5 Operazioni eseguibili dal server

Open : aggiunge un nuovo file con contenuto vuoto allo storage e lo rende aperto da parte del client che ha richiesto l'operazione , oppure apre semplicemente il file per il client. **Read** : legge il contenuto del file di cui è stata richiesta la lettura e lo restituisce. **readN** : legge e restituisce il contenuto del numero di file richiesto (scelti in ordine dalla struttura dati) e restituisce il numero di file effettivamente letti. Questo e quello della prossima funzione sono gli unici casi in cui parte della comunicazione sulla socket viene fatta in una funzione chiamata da quella che esegue il worker, e non direttamente da quest'ultima. **Write**: legge un buffer dalla socket e lo scrive nel contenuto del file (se presente sullo storage) corrispondente al pathanme passato. Se il file è già stato scritto, ne sovrascrive il contenuto. **Append**: scrive in append al contenuto del file il buffer passato **Close**: chiude il file al client (ovvero elimina l'fd del client dalla lista dei client che lo hanno aperto).
In tutti i casi viene restituito al client un intero che gli permetterà di sapere se l'operazione che ha richiesto è andata a buon fine o meno.

1.6 Mutua Esclusione

Potendo non implementare un meccanismo di LOCK sui singoli file, ho implementato la mutua esclusione sulla lista dei lavori da effettuare e sulla intera struttura dati.

1.7 Terminazione

Per la gestione di SIGHUP viene chiusa la socket e messo a 1 il campo *isClosed* della struct dei parametri. Da quel momento il ciclo del thread principale che fa la accept si interromperà e non verranno più accettate connessioni. I workers finiranno di servire i client che si erano già connessi ma quando troveranno la coda di lavori vuota termineranno. Per quanto riguarda invece SIGINT e SIGQUIT viene sempre chiusa la socket ma poi viene messo a 1 il campo *closed* e da quel momento i worker termineranno appena avranno terminato la singola operazione che stavano eseguendo. In entrambi i caso poi viene fatta la join su tutti i thread e vengono rilasciate le strutture dati utilizzate.

1.8 Gestione degli errori

Ho gestito gli errori tramite propagazione al chiamante, in generale tutte le funioni restituiscono -1 e stampano un messaggio in caso di errore. Le funzioni dell'API settano *errno*.

2 Client

2.1 Configurazione

Il client viene configurato allo stesso modo del server, ma gli argomenti letti vengono inseriti in una struct **Options** che contiene tutti i parametri necessari alla connessione e alle richieste da fare al server. Il file di configurazione ha la stessa struttura di quello del server, ma è al momento vi è presente un solo parametro, ovvero il nome della socket. Se non specificato il nome della socket da riga di comando, il client utilizzerà quello letto da file di configurazione, altrimenti utilizzerà quello che è stato passato dall'utente.

2.2 Opzioni Client

Il client prima legge da riga di comando le opzioni passate le memorizza nella struttura **Options**, successivamente per ogni opzione chiama le funzioni corrispondenti dell'API. Nel caso venissero passati opzioni non supportate, le ignorerà stampando un messaggio che avvisi l'utente.

3 API e Comunicazione

Per ogni scambio di messaggi tra client e server che non sia di lunghezza conosciuta viene inviato prima un intero che rappresenta la lunghezza, e il messaggio immediatamente dopo.

Se vengono chiamate le funzioni non supportate queste ritornano sempre -1 e avvisano l'utente con un messaggio di errore.

La **readFile** restituisce 0 (se la lettura è andata a buon fine) anche nel caso in cui non sia specificata la directory in cui salvare il file letto, però lo salva nel buffer, che però poi verrà rilasciato alla terminazione del programma client senza che sia salvato da nessuna parte.

La **openFile** non supporta il flag di lock.

4 Test

Nel Makefile sono presenti il test1 e il test2.
Il test1 fa partire l'esecuzione del server senza specificare il file di configurazione , ovvero utilizzando quindi quello di default $config\_server\_test1.txt$, per il test2 invece specifica l'altro file di configurazione $config\_server\_test2.txt$.
Il test1 esegue le principali operazioni che si possono fare, lettura, scrittura, lettura da una directory, scrittura in una directory, lettura di N file...

Allo scopo di distinguere meglio il risultato di una lettura di alcuni file dalla lettura di tutti i file ho creato tue cartelle *dir1* e *dir2*. Dopo l'esercuzione del test1 all'interno di dir1 ci saranno i 4 file risultati della lettura prima di un singolo file e poi di tre file, mentre in dir2 ci saranno tutti i file che inizialmente erano in *provafile* risultati della lettura di tutti i file presenti nello storage. Il test2 invece ha lo scopo di provare l'algoritmo di rimpiazzamento dei file nella struttura dati del server, per questo esegue due operazioni: la scrittura di tutti i file presenti nella cartella *provafile*, che scateneranno alcune sostituzioni di file nella struttura dati, e successivamente la lettura di tutti i file presenti sullo storage nella cartella *dir3*, al fine di verificare che alcuni presenti nella cartella *provafile* non erano più presenti alla fine dell'esecuzione per aver fatto spazio ad altri.
Quando un file viene eliminato dallo storage viene stampata la frase *"Elimino un file"*.
Il client stampa una frase per ogni operazione con indicato il suo esito.

5 Note Finali

Tutti i file sono identificati dal loro **pathname assoluto**, questo è sia il loro nome identificativo all'interno del server, ma deve essere anche usato da riga di comando dall'utente che utilizza il client (stessa cosa per le directory).

Il client, se lanciato con l'opzione -p, stampa una stringa per ogni funzione dell'API chiamata che indicherà all'utente se l'operazione è andata a buon fine o meno.

Leggendo il nome della socket dal file di configurazione, il client è in grado di connettersi anche senza che gli sia passata l'opzione -f, nel caso invece ciò avvenisse, verrà utilizzato il nome inserito.

Ogni worker, per ogni lavoro che svolge, stampa un carattere per indicare che tipo di operazione gli è stata richiesta, questa funzione ha lo scopo di seguire più chiaramente il funzionamento del server.

Nel caso di fallimento di un'operazione del client, questo termina e sarà il server a chiudere il file che eventualmente fosse rimasto aperto da parte del client.

Nell'archivio contenente il mio progetto si trova una cartella "provafile" e una "dir". Nella prima è presente una dozzina di file di testo pensati per provare il server ed utilizzati nei test, la seconda è pensata come directory su cui salvare i file letti dal server.