Università di Pisa -- Dipartimento di Informatica
Corso di Laurea in Informatica
Progetto di Laboratorio di Sistemi Operativi
a.a. 2020-21
Michele Velardita
N° matricola 578770

L'applicazione server consiste in un programma multi-threaded che gestisce più richieste contemporaneamente di connessione da parte di client diversi. I client comunicano col server tramite un'API, la cui implementazione si trova in ServerAPI.c e dopodiché vengono serviti da un worker del server il quale esegue un'operazione per poi rimettersi in attesa.

Compilazione ed esecuzione
Una volta scompattato l'archivio, sarà sufficiente spostarsi dentro la root del progetto e compilare il progetto con il comando make. I file eseguibili si troveranno all'interno della stessa cartella in cui si trova il makefile e in cui è stato eseguito il comando di compilazione.

Makefile
è possibile compilare con i comandi:
$make test1 $make test2 $make test3
rispettivamente passano al server i file di configurazione config1.txt, config2.txt e config3.txt, contenuti nella cartella tests, insieme ai 3 script test1.sh test2.sh, test3.sh (insieme a test3support.sh).
Si può utilizzare da terminale il comando make cleanall per eliminare i file compilati e svuotare le cartelle "FileLetti" e "FileEvict" ed eliminare i file di log nella cartella tests.
(A volte il cleanall non elimina il file “mysock” per cui una compilazione successiva può generare errore. Basta rimuovere manualmente il file).

File di configurazione
I file di configurazione contenenti i dati per l'avvio del Server sono generici file di testo contenuti nella cartella tests e contenenti i seguenti campi: socket_name, threads_number, max_storage_size, max_file_number, logger_path.
Viene parsato al server tramite la funzione int parse(..) (riga 35, server.c).

Strutture dati
Nel progetto utilizzo quattro strutture dati principali: SharedQueue_t e List_t contenute in datastructures.h, File_t e FileSystem_t contenute in Filemanager.h.
File_t contiene informazioni sul file quali: il nome del percorso, la dimensione, il contenuto (void* cont), un puntatore al file precedente e un puntatore al file successivo, una Linked list dei client che hanno il file aperto e il client che tiene la lock. Il FileSystem_t tiene traccia di tutti i File_t conservando il puntatore al primo e all’ultimo file. Contiene informazioni sullo storage del server.
SharedQueue contiene i ready client.

Client
Il client consiste in un programma single-threaded che si interfaccia con il server. Contiene un ciclo while in cui setto le variabili relative alle opzioni che può ricevere ad 1, ad esempio all'opzione -p

corrisponde la variabile foundp (=1 se è stata letta correttamente l'opzione -p, =0 altrimenti). In seguito il main del client esegue un controllo sul corretto passaggio dei parametri relativi alle opzioni e sul corretto utilizzo. Nel secondo ciclo while eseguo le opzioni passate da terminale mandando le richieste al server.

Comunicazione Client-Server
Il client riconosce l'argomento passato da terminale ed esegue la rispettiva funziona, ad esempio leggendo l'argomento "-l" chiamerà la lock_file() che ha al suo interno la chiamata della lockFile della server API. Il server gestisce la richiesta tramite la fs_request_manager (filemanager.c). Invia la risposta al client o lancia un eventuale errore (vedere Macro error).

Macro request
All'interno dell'API ho dichiarato delle macro che rappresentano gli argomenti possibili da linea di comando del client:
OPEN_F 1
READ_F 2
READ_N_F 3
WRITE_F 4
APPEND_T_F 5
LOCK_F 6
UNLOCK_F 7
CLOSE_F 8
REMOVE_F 9

Macro error
Sempre all'interno dell'API ho dichiarato delle macro per gli errori possibili che può generare la gestione di una richiesta del client:
E_INV_FLG 140 -> Invalid flags
E_INV_PTH 141 -> Invalid path
E_LOCK 142 -> File locked
E_NOT_EX 143 -> File not exists
E_ALR_EX 144 -> File already exists
E_BAD_RQ 145 -> Invalid request
E_ALR_LK 146 -> File already locked
E_NO_SPACE 147 -> No enough space (in server memory)
E_NOT_OPN 148 ->File not open
E_INV_SCK 149 -> Invalid socket path
E_NOT_CON 150 -> Not connected

Sincronizzazione
Le funzioni f_startRead, f_doneRead, f_startWrite, f_doneWrite mi permettono di gestire le operazioni di lettura/scrittura su file in mutua esclusione. Utilizzo i mutex f_order (per mettermi in coda) e f_mutex(sezione critica), e la condition variable f_go(per segnalare la fine di un'operazione di lettura o scrittura). righe (791-827).

Cache Evict
Per gestire la memoria del server utilizzo una politica Fifo. L'algoritmo di cache è contenuto nella funzione File_t* cacheEvict, che ritorna il puntatore al File_t scartato. Dato che gestisco i file come

una linked list, al raggiungimento della capacità massima dello storage del server rimuovo il file meno recente (in testa alla coda), spostando adeguatamente i puntatori prev e next del file da scartare. (filemanager.c righe 645-700).