Università di Pisa -- Dipartimento di Informatica
Corso di Laurea in Informatica
Progetto di Laboratorio di Sistemi Operativi
a.a. 2020-21

Giulia Falchetti
Matricola: 550286
Repository Github: https://github.com/Falchetti/Server_storage
Corso A

1. Introduzione

Il progetto riguarda la realizzazione di un file storage server e di uno o più client che comunicano con esso attraverso un'interfaccia ben definita. Lo storage viene configurato all'avvio del server fissandone la capacità in termini di dimensione e numero di file, i quali vengono identificati unicamente tramite il proprio path assoluto. Lo scopo del server, implementato come un singolo processo multi-threaded, è gestire adeguatamente alcune decine di connessioni contemporanee da parte di più client, i quali inviano richieste riguardo i file memorizzati nello storage e ricevono risposte in accordo al protocollo di comunicazione "richiesta-risposta".

2. Client

Il client è un programma che ha come funzione principale l'invio di richieste al server tramite un'interfaccia prestabilita. Le richieste da inviare sono comunicate al client tramite dei flag passati da riga di comando.

2.1 Ordine e molteplicità dei flag

L'ordine e la ripetizione dei flag influiscono sulla valutazione delle richieste da svolgere.

- Il flag **–f,** il quale specifica il nome del socket AF_UNIX a cui connettersi, deve essere specificato prima dei flag che prevedono richieste dirette al server {-w, -W, -r, -R, -l, -u, -c}. Inoltre, in caso di richieste multiple di tipo –f, quelle successive alla prima provocano la stampa di un messaggio d'errore e vengono ignorate.
- I flag **–d** e **–D**, che indicano su quali cartelle salvare i file inviati dal server, devono essere specificati prima dei flag –r/-R e –w/-W a cui fanno riferimento. Inoltre, qualora i flag –d/-D fossero ripetuti, i file inviati dal server verranno salvati nella cartella specificata più di recente tra quelle indicate fino a quel momento.
- È stata seguita la stessa politica per il flag **–t**, solo le richieste a lui successive avranno quello specifico tempo di attesa.
- In caso di richieste multiple di tipo **–p**, quelle successive alla prima provocano la stampa di un messaggio d'errore e vengono ignorate.

2.2 Protocolli per l'accesso e la modifica dei file dello storage

Il client, prima di inviare richieste al server, si assicura che il path ricevuto da riga di comando non sia relativo, se lo è lo trasforma in assoluto.
In caso di richiesta di scrittura (-w/-W) o di lettura (ad eccezione di –R) di un file viene applicato il protocollo:

- Apertura del file con lock attiva
- Scrittura/lettura del file
- Chiusura del file

Invece, per fare operazioni di lock, unlock e rimozione, il file di riferimento non deve necessariamente essere stato aperto prima. Per leggere, scrivere o rimuovere il file dallo storage inoltre, è necessario che non sia in stato locked da parte di un altro client. Infine, in caso di richiesta di scrittura, se il file non è presente nello storage, questo viene creato (*openFile(O_CREATE|O_LOCK)*) e il suo contenuto viene scritto tramite la

funzione *writeFile()*, altrimenti viene aperto con lock (*openFile(O_LOCK)*) e scritto in append (*appendToFile()*).

2.3 Dettaglio flag più significativi

Per i seguenti flag sono state fatte le scelte progettuali indicate.

-f : Il processo client apre la connessione con il server una sola volta e per farlo ripete un tentativo di connessione al secondo per un minuto.
-w : Nel caso venga passato un valore negativo o un oggetto non numerico come parametro di questo flag, viene usato il valore di default 0. Inoltre, per realizzare la ricorsione nelle cartelle è stata utilizzata, con qualche modifica, la funzione lsR vista a lezione.
-d/-D : Se le cartelle specificate da riga di comando non esistono, vengono create.

2.4 Gestione errori

In caso di esito negativo delle richieste inviate al server, solo per errori gravi come, ad esempio, il fallimento di una malloc il processo client si interrompe, negli altri casi passa semplicemente alla richiesta successiva dopo aver stampato un messaggio d'errore sulla base dell'errno settato dalla funzione che lo ha rilevato.

3. Comunicazione Client – Server

Il client comunica con il server attraverso un protocollo di comunicazione e un'interfaccia ben definiti.

3.1 Protocollo di comunicazione

Il client, attraverso la API, invia un messaggio di richiesta al server nel seguente formato:

[tipo_di_operazione**;**file_di_riferimento**;**size_contenuto]

Se la richiesta è readNFiles al posto del file_di_riferimento ci sarà il numero dei file richiesti.
In seguito, se la taglia del contenuto è maggiore di zero, invia anche il contenuto [contenuto].
Il server riceve il messaggio e se c'è necessità di inviare file al client (file espulsi/letti) ne invia il path (se necessario), la taglia del contenuto e, se questa è maggiore di zero, il contenuto stesso. Al termine dell'invio dei file invia il carattere speciale "**$**".
Infine, per ogni richiesta, in caso di successo invia il messaggio "**Ok**" e in caso di fallimento un messaggio d'errore nel formato [**Err:**-------].
In quest'ultimo caso, il client decodifica il messaggio d'errore e lo stampa a schermo.
Infine, tutti i messaggi scambiati sul canale di comunicazione tra client e server seguono il protocollo:
invio taglia del messaggio, invio del messaggio.

3.2 Dettaglio delle funzioni dell'API più significative

Per le seguenti funzioni sono state fatte le scelte progettuali indicate.

- **closeConnection()** : se la connessione è stata già chiusa, la richiesta viene ignorata.
- **closeFile()** : alla chiusura di un file, la lock viene rilasciata, se se ne è in possesso. Inoltre, le operazioni che necessitano di avere il file aperto quali, closeF(), writeF(), appendToF(), readF(), readNF(), in seguito alla sua chiusura falliscono.
- **removeFile()** : prima di rimuovere il file le lock vengono rilasciate.

Ad eccezione di openConnection(), nelle funzioni che implementano la API si controlla sempre che la connessione con il server sia stata stabilita in precedenza.

4. Server

Il server è implementato come un singolo processo multi-threaded secondo lo schema master-workers.

4.1 Strutture Dati

Si elencano le strutture dati utilizzate e la loro funzione.

- **type** : struttura dati utilizzata per rendere generica la funzione di cleanup del server.
- **icl_hash_t**: hashtable utilizzata per realizzare il file storage. Per farlo è stata usata l'implementazione fornita a lezione "*icl_hash.c*" di "*Jakub Kurzak*". Le sue entry sono così strutturate:
    - Chiave : path del file
    - Valore : struttura di tipo file_info
- **file_info** : campo valore dell'hashtable, contiene tutte le informazioni di rilievo su un file.
    - Detentore della lock (-1 se non esiste)
    - Lista dei client che tengono il file aperto (un file può essere aperto da più client)
    - Flag per memorizzare se l'ultima operazione sul file è stata openCL (booleano)
    - Taglia del contenuto del file (se vuoto, 0 o valore negativo)
    - Contenuto del file (file con taglia limitata)
    - Variabile di condizione (permette di notificare se il lock owner del file ha rilasciato la lock)
- **queue_t** : utilizzata l'implementazione fornita a lezione, con qualche modifica, per realizzare le seguenti due code.
    - *Task_queue*: utilizzata per affidare ai thread le richieste dei client.
        - Entry: puntatore al descrittore di un client.
    - *File_queue*: utilizzata per realizzare l'algoritmo di rimpiazzamento con politica FIFO.
        - Entry: struttura di tipo file_repl.

      Ogni volta che un file viene creato viene messo nella coda. Tuttavia, quando viene eliminato dallo storage non viene rimosso da file_queue. Per questo ogni volta che un file viene estratto dalla coda è necessario controllare sia ancora presente nello storage.
- **file_repl** : entry utilizzata nella coda file_queue, contiene i seguenti campi.
    - Path del file di riferimento
    - Puntatore al valore associato a quel path nello storage
- **th_sig_arg** : argomento passato al Signal-Handler Thread. I cui campi sono:
    - Maschera delle interruzioni
    - Descrittore della pipe per comunicare con il Master Thread
- **th_w_arg** : argomento passato al Signal-Handler Thread. I cui campi sono:
    - Coda sulla quale vengono caricati i task da eseguire
    - Descrittore della pipe per comunicare con il Master Thread

4.2 File di configurazione

La configurazione del server avviene tramite il parsing di un file di configurazione che prevede il seguente formato:

DIM:val_dim
NFILES:num_files
NWORKERS:num_workers
SOCKET:path_socket
LOG:path_file_di_log

Non vengono accettati valori minori o uguali a zero per NFILES e DIM. Infine, in caso vi siano errori durante il parsing del file, verranno utilizzati i seguenti valori di default:

DIM:1024
NFILES:100
NWORKERS:4
SOCKET:./mysock.sk
LOG:./log.txt

4.3 Avvio del server

Il server dopo aver inizializzato variabili e strutture, aperto i file e creato i thread necessari, inizializza la comunicazione con i client aprendo il socket e mettendosi in ascolto.
Il server attende quattro tipi di descrittori:

- *Descrittore di nuova connessione*: il server tramite l'accept crea il canale di comunicazione con il nuovo client, ne inserisce il descrittore nella maschera e aumenta il contatore delle connessioni attive.
- *Descrittore della pipe condivisa con i thread workers*: il server viene avvisato del fatto che il worker ha compiuto la richiesta di quel client. Se il descrittore letto nella pipe è negativo significa che quel client si è disconnesso, mentre se è positivo significa che è pronto per nuove richieste, e viene così riaggiunto alla maschera.
- *Descrittore della pipe condivisa con il thread signal-handler*: il server viene avvisato del fatto che è sopraggiunta un'interruzione. Se si tratta di SIGINT/SIGQUIT il valore immesso nella pipe dal signal-handler thread è positivo e il server si interrompe. Se il valore è negativo invece, significa che è avvenuta una SIGHUP, quindi il server si ricorda di non accettare più nuove connessioni e attende che quelle attive finiscano prima di interrompersi.
- *Descrittore di una connessione attiva*: il server affida ad uno dei workers la richiesta del client disponibile inserendone il descrittore nella task_queue e togliendolo dalla maschera.

Il server smette di ascoltare quando ha finito le connessioni attive (nel caso di SIGHUP) o dopo aver ricevuto i segnali SIGINT/SIGQUIT.

4.4 Thread workers

Il thread worker estrae, non appena è possibile, un descrittore per la comunicazione con un client dalla task_queue. Se il descrittore ha valore speciale 0x1 il thread termina, altrimenti si mette in attesa della richiesta da parte del client. In quest'ultimo caso, una volta ricevuta la richiesta, la decodifica, la esegue sulla base dell'interfaccia dello storage e ne comunica al client l'esito.
Se la funzione dedicata alla gestione delle richieste restituisce il valore –2, significa che il client si è disconnesso. Di conseguenza il thread, invece di inviare al master thread il descrittore del client così come lo ha estratto dalla coda, lo invia rendendolo negativo, per notificare per l'appunto l'avvenuta disconnessione.

4.5 Interfaccia storage

Per le seguenti funzioni sono state fatte le scelte progettuali indicate.

- **openFileL()/openFileCL()/openFileO()/openFileC()** : in caso si tenti di aprire un file che è già stato aperto la richiesta viene ignorata senza generare errori.
- **openFL()/openFCL()/lockF()** : dopo l'attesa per l'acquisizione della lock_ownership sul file, è necessario controllare nuovamente che il file non sia stato eliminato.
- **readNF()** : questa funzione non necessita di avere i file aperti.

4.6 Algoritmo di rimpiazzamento

Lo storage possiede una capacità limitata, dunque a seguito di operazioni di creazione e scrittura può necessitare di espellere dei file per fare posto ai nuovi. È stata implementata una politica di rimpiazzamento di tipo FIFO, supportata da una coda (*file_queue*), nella quale i file vengono aggiunti al momento dell'inserimento nello storage ed espulsi, qualora ancora esistenti, nel caso ci sia la necessità di liberare spazio. I file espulsi a seguito di operazioni di scrittura (*writeF()*, *appendToF()*) vengono inviati al client, mentre quelli rimossi a seguito di operazioni di inserimento (*openC()*, *openCL()*) vengono semplicemente eliminati a causa della segnatura della funzione *openFile()* nella API che non prevede come parametro una cartella di salvataggio.

4.7 Gestione errori e pulizia storage

Il server si interrompe in caso di errori gravi come il fallimento di una malloc o l'impossibilità di inizializzare o aggiornare strutture dati fondamentali per il funzionamento e la correttezza dello storage o del server. Altrimenti, se ad esempio una richiesta di un client non va a buon fine viene notificato un messaggio d'errore e si passa alla successiva.
Per quanto riguada la pulizia della memoria e delle strutture dati invece, il server alla chiusura si dedica alla liberazione della memoria, anche tramite la funzione cleanup_server(), mentre per quanto riguarda lo storage, ad ogni disconnessione di un client viene navigata l'hashtable per rimuovere ogni traccia di quel client nelle liste di apertura dei file e nei campi lock_owner.

5. File di log

Durante l'esecuzione il server effettua sul file di log specificato in fase di configurazione il logging di tutte le operazioni con esito positivo svolte.

5.1 Struttura del file di log

Il file di log presenta il seguente formato:

[Tipo_di_operazione file_di_riferimento bytes_scritti bytes_letti thread_di_riferimento]

Nel caso delle operazioni di *open_connection* e *close_connection* nel campo "file_di_riferimento" è inserito il descrittore della connessione, mentre nello stesso campo nel caso dell'operazione di *rimpiazzamento* è presente il file selezionato come vittima.
Nel campo "bytes_letti" viene inserita la taglia del contenuto dei file rimossi nel caso di operazioni di *remove* o *rimpiazzamento*. I campi non significativi vengono riempiti con il valore –1.

6. Makefile

6.1 Flag di compilazione

Nel makefile sono presenti due possibili flag di compilazione.
Il primo [CFLAGS = -Wall -pedantic -I $(INC)] è adatto a macchine con compilatori recenti, mentre il secondo [CFLAGS = -Wall -pedantic -std=c99 -D_GNU_SOURCE=1 -I $(INC)] è da utilizzare su macchine con compilatori che necessitano la specifica di –std=c99. Quest'ultimo fa sì che non vengano riconosciute alcune funzioni e rende necessaria l'ulteriore specifica di -D_GNU_SOURCE=1. Il codice in questo modo viene compilato correttamente, viene tuttavia influenzata negativamente la stampa degli errori tramite strerror_r.

6.2 Targets

Viene riportato il dettaglio dei seguenti target:

- **statistiche** : permette di eseguire lo script *statistiche.sh* che analizza il file di log prodotto.
- **clean** : permette di rimuovere i socket files.
- **cleanall** : permette di rimuovere socket files, eseguibili, file oggetto, file di log, librerie e le cartelle dove sono stati salvati i file dello storage generati dall'esecuzione del programma.