Progetto di laboratorio di sistemi operativi

Kamer Cekini: 598793 - corso B

Anno accademico: 2020/2021

Pagina GitHub del progetto: https://github.com/KamerC1/ProgettoSOL-2021

File system

Struttura

Il file system è composto da due strutture dati:

- Uno struct `ServerStorage` che, tra le altre cose, contiene una coda per gestire la politica di rimozione di tipo FIFO, e una tabella hash per memorizzare i `File`.
- Uno struct `File` che viene memorizzato come entry della tabella hash di `ServerStorage`.
  - Per la tabella hash, è stata usata l'implementazione di Jakub Kurzak (reperibile qui).

Mutua esclusione

Il file system usa due livelli di mutex: una mutex globale su tutto il file system (`ServerStorage.globalMutex`) e una mutex locale per ogni `File` (`RwLock_t.mutexFile`).

Mutex globale

La mutex globale viene utilizzata:

- in tutti quei casi in cui non è possibile usare accedere alla mutex locale (creazione e rimozione di un file);
- nelle funzioni che possono invocare l'algoritmo di rimpiazzamento per espellere eventuali `File` (e.g. `appendToFile()` e `writeToFile()`). Questo è necessario per non ritrovarci nel caso in cui, ad esempio, una `writeFile()` determini la rimozione di un `File` ancora in uso da un'altra funzione dell'API.
- nelle funzioni che fanno uso della system call: `chdir()` (e.g. `readNFiles()`).

Mutex locale

La mutex locale è realizzata attraverso la struttura dati `RwLock_t` che permette di implementare una *multi-reader lock* con priorità alla scrittura - questo perché la stragrande maggioranza delle funzioni dell'API modificano la struttura `File`.

Per potere acquisire la mutex locale, il file system deve:

1. acquisire la mutex globale e individuare il `File` su cui operare;
2. acquisire la mutex locale sul `File` del punto 1 e, successivamente, rilasciare la mutex globale.

Gestione concorrenza per `removeFile()`

Oltre alle variabili di condizione e alle mutex citate sopra, è necessario una variabile di condizione sul file system (`ServerStorage.condRemoveFile`) per poter gestire la rimozione di un `File` attraverso la `removeFile()`. A supporto di ciò vengono utilizzate due valori booleani:

- `ServerStorage.isRemovingFile`: *TRUE* $\iff$ si sta eliminando il `File`.
- `ServerStorage.isHandlingAPI`: *TRUE* $\iff$ si sta gestendo un `File` del file system.

Questo è essenziale perché, in quelle funzioni in cui viene acquisita la mutex locale, la mutex globale viene rilasciata e, di conseguenza, vi è il rischio che nel mentre la `removeFile()` venga eseguita. Ad esempio:

1. in `lockFile()`, il Thread `T1` viene *de-schedulato* appena ottiene la *mutex* locale e rilascia quella globale.
2. Entra in esecuzione il thread `T2` che acquisisce la *mutex* globale e rimuove il file.
3. Riparte il thread `T1` che ora deve operare su un `File` rimosso.

Gestione concorrenza per `writeFile()` e `appendToFIle()`

Anche se le funzioni `writeFile()` e `appendToFile()` usano la mutex globale per tutta la loro durata, è obbligatorio usare comunque la mutex locale per ovviare al problema in cui tali funzioni vengano eseguita mentre altre funzioni, che usano la mutex locale, sono in esecuzione. Ad esempio:

1. in `lockFile()`, il Thread `T1` viene *de-schedulato* appena ottiene la *mutex* locale e rilascia quella globale.
2. in `writeFile()`, entra in esecuzione il thread `T2` che acquisisce la *mutex* globale e rimuove il `File` su cui stava operando il thread `T1` (grazie alla politica di rimozione).
3. Riparte il thread `T1` che ora deve operare su un `File` rimosso.

In tal senso, il thread `T2` prima di essere eseguito deve aspettare che non vi sia nessuno lettore o scrittore in esecuzione.

openFile()

Per poter operare su un `File`, è prima necessario aprirlo attraverso la funzione dell'API: `openFile()`. Di quest'ultima, oltre ai flags indicati nella specifica, è stato aggiunte il flag OPEN_FILE che permette, al client chiamante, di aprire un file.

Inoltre, è possibile combinare le flags in questo modo:

- O_CREATE + O_OPEN: apre il `File` se esiste, altrimenti lo crea
- O_CREATE + O_OPEN + O_LOCK: se il `File` non esiste si comporta come O_CREATE + O_LOCK, altrimenti si co

Questa combinazione di flags permette al client di eseguire in modo atomico la `openFile()`, altrimenti sarebbe costretto a controllare prima lo stato del `File` e, successivamente, decidere quale operazione svolgere (e.g. O_CREATE, O_LOCK) - vedere sezione "Considerazioni" per capire perché potrebbe essere un problema.

lockFile()

L'operazione di `closeFile()`, `lockFile()` e `removeFileInfo()` non necessitano di possedere la lock per essere eseguite: questo perché non influenzano lo stato del `File` e degli altri client, né possono leggere dati "riservati".

Politiche di rimpiazzamento

Il file system implementa 5 diverse politiche di rimpiazzamento: FIFO, LRU, LFU, MRU e MFU. Esclusa la politica FIFO, se vengono individuati un insieme di potenziali vittime (e.g. n elementi che hanno lo stesso numero di accessi nella politica LFU), viene scelto sempre l'ultimo `File` presente nella tabella hash.

- Per implementare la politica di tipo FIFO viene usata una coda che memorizza i `File` in ordine di creazione (attraverso il flag O_CREATE in `openFile`). L'inserimento in coda e la cancellazione in testa hanno complessità $O(1)$.
- Per implementare la politica di tipo LRU e MRU viene usato un campo di tipo `time_t`, specifico per ogni `File`, che memorizza il tempo dell'ultimo accesso al relativo `File`. La ricerca della vittima ha complessità $O(n)$, con n il numero di entry della tabella hash.

- Similmente alla politica LRU e MRU, gli algoritmi LFU e MFU usano un contatore che tiene conto dei numeri di accessi al relativo File.

Anche se `openFile()` può causare l'espulsione dei `File`, solamente ad `appendToFile()` e `writeFile()` vengono inviati i `File` espulsi (questo perché `openFile()` non contiene il parametro `dirname`). In particolare, i `File` espulsi vengono memorizzati in una lista poiché una singola scrittura può causare l'espulsione di più `File`.

Gestione degli errori

La maggior parte degli errori vengono reindirizzati verso il chiamante (tutte le funzioni delle API ritornano -1 in caso di errore, ed *errno* è impostato opportunamente).
Il sistema, invece, termina bruscamente per gli errori fatali e in tutti quei casi in cui rimarrebbe in uno stato inconsistente (e.g. errore in `pthread_mutex_lock()`, `chdir`, ecc.). In questi casi, la memoria allocata non viene liberata e non vengono chiusi tutti i *fd*.

Funzionalità aggiuntive

All'API sono state aggiunte le funzioni:

- `size_t getSizeFileByte(const char pathname[])`: ritorna il numero di bytes occupati dal `File` identificato da pathname.
- `int isPathPresent(const char pathname[])`: verifica se il `File` identificato da pathname è presente nello storage e se è stato aperto dal client che ha chiamato la funzione.

Server

Implementazione

La comunicazione tra master e worker avviene tramite una coda di tipo FIFO, memorizzando in questo ultima l'*fd* dei client da gestire. Il worker, una volta soddisfatta la richiesta, comunica al manager l'esito attraverso una pipe. In particolare, viene ritornato -1 se il client ha chiuso la connessione, altrimenti il *fd* precedentemente gestito. Inoltre, Tutti gli fd ritornati dalla pipe vengono messi nuovamente nella `readset`, la quale è gestita dalla `select()`.

Segnali

Il segnale SIGPIPE viene mascherato dal server per l'intera durata della sua esecuzione, mentre i segnali SIGINT, SIGQUIT e SIGHUP vengono gestiti come richiesto. L'unico particolare degno di nota è che il server non termina immediatamente alla ricezione di segnali SIGINT e SIGQUIT, ma aspetta che il worker-thread finisca di gestire la richiesta in corso.

Chiusura connessione del client

Quando il client chiude la connessione con il server, viene chiamata la funzione `removeClientInfo()` che permette di rimuove l'*fd* e la *lock* del suddetto client da ogni file. Questo è necessario perché lo stesso *fd* può essere assegnato a più client in un determinato lasso di tempo (a patto che più client nello stesso istante non abbiano lo stesso *fd*).

File di configurazione

Il file di configurazione viene passato al server tramite linea di commando e deve presentare le seguenti regole:

- Vengono ignorate tutte le linee che non iniziano con una lettera.
- Ogni istruzione è ha esattamente questa sintassi `<chiave>=<valore>;`, dove le *chiavi* devono appartenere a un insieme predeterminato. Quindi, tra i vari campi non sono concessi: spazi, tab, ecc.

- Ogni istruzione deve terminare con il punto e virgola - tranne per le linee ignorate.
- Ogni istruzione deve essere presente su una linea a sé stante. Questo perché tutto i caratteri presenti dopo il punto e virgola vengono ignorati.
- Se un'istruzione con la stessa chiave compare più volte, allora viene considerata solamente l'ultima occorrenza.

```
MAX_BYTES=<Numero naturale>; #size massima in bytes dello storage
MAX_FILE=<Numero naturale>; #size massima in files dello storage
MAX_WORKER=<Numero naturale>; #size esatto di thread worker
LOGFILE=<pathname>; #(se non esiste viene creato, altrimenti sovrascritto)
SOCKNAME=<pathname>;
REPLACEMENTE_ALG=<val>;
#Val può essere: FIFO, LRU, MRU, MFU, LFU (è case sensitive)
```

Se una delle chiavi sopracitate non rispetta le regole richieste, oppure non è proprio presente, allora viene scelto un valore di default esclusivamente per la chiave mancante (questi valori sono presenti in: `configParser.h`). I valori di default vengono impostati anche se al server non viene passato nessun file a linea di commando oppure se il file passato non è reperibile.

Logging

La scrittura sul file di log avviene all'interno di ogni funziona che implementa le funzioni dell'API (tali funzioni sono presenti in `serverAPI.c`).

Nella cartella del progetto è presente un file chiamato: `Esempio_fileLog.txt` che mostra un esempio logging per ogni funzione dell'API.

Protocollo di comunicazione

Per ogni funzione dell'API, esclusa al `openConnection()` e la `closeConnection()`, il client invia un intero al server che identifica la richiesta da gestire e, a seguire, viene eseguita una `writen` per ogni parametro della funzione richiesta.

Lato server, vengono inviati eventuali informazioni a seconda della richiesta eseguita (e.g. numero di `File` letti) e, in qualunque caso, un codice di errore che identifica l'esito dell'operazione (se `errno = 0`, allora la richiesta è stata eseguita con successo).

Al posto delle syscall `write()` e `read()` sono state usate le funzioni `readn()` e `writen()` (per maggiori informazioni, guardare questa pagine). Perciò, per ogni messaggio inviato, vi precede una `writen` che specifica la lunghezza del suddetto messaggio.

Client

Pathname assoluto e relativo. Le operazioni che operano con i `File` presenti su disco possono accettare anche pathname relativi; ma sul server, i file, devono essere memorizzati con il loro pathname assoluto - questo è necessario per evitare problemi di collisione.
Tuttavia, se un `File` non è presente su disco, queste operazioni accettano solamente pathname assoluti. Ad esempio, ciò può accadere se si tenta di fare la `readFile()` di un file precedentemente memorizzato sullo storage, ma rimosso dalla memoria secondaria.

Opzione -D e -d. L'opzione $-D$ e $-d$ sono valide solamente se le operazioni precedenti sono, rispettivamente, $-a, -w, -W$ e $-r, -R$. Inoltre, $-D$ e $-d$ accettano anche pathname relativi.

Opzione -a. Oltre alle opzioni richieste, è stato aggiunto l'opzione `-a file1,string` che permette di scrivere in append la stringa `string` sul file `file1` (per potere passare una stringa composta da spazi, bisogna racchiudere `string` tra le

doppie virgolette: e.g. `-a file.txt,"Hello World!"`)

Gestione errore. Quando le operazioni a linea di commando causano un errore lato server, il client deve terminare immediatamente: è stata decisa queste scelte per evitare eventuali anomalie.
Ad esempio, con `-a file1,string -D dir1 -r dir1/file1` l'utente potrebbe voler memorizzare in `dir1/file1` solamente il `File` aggiornato con l'opzione `-a`, ma se quest'ultima fallisse, il contenuto `dir1/file1` verrebbe sovrascritto con un valore indesiderato.

Lock. Se un client tenta di ottenere la lock già acquisita da un altro, esso rimane in attesa finché non può possedere la lock (la notifica avviene dentro la funzione `setUnlockFile()`, in `serverAPI.c`)

Considerazione sulle opzioni

- Le opzioni: $-f, -p$ e $-t$ possono essere collocate in qualsiasi posizione e vengono eseguite prima delle altre operazioni.
- L'opzione $-t$ non può essere ripetuta più volte, e il tempo in millisecondi che deve intercorre tra l'invio di due richieste vale per qualsiasi istruzione (tranne per $-f, -p$ e $-h$). E' stata implementata questa scelta poiché sarebbe stato confusionario e ridondante ripetere più volte l'opzione $-t$ per ogni richiesta - se l'utente desidera avere un time-out diverso, può comunque invocare il client più volte.
- Tutte le operazioni antecedenti all'opzione $-h$ vengono eseguite comunque - ma non quelle successive, poiché il client termina.
- Per l'opzione $-R$ non vi devono essere spazi tra la "`R`" e l'argomento opzionale (e.g. `-R10` è corretto; mentre `-R 10` si comporta come `-R0`)

Test

Oltre ai 3 test richiesti, è stato creato un'estensione del *test 2* che permette di testare tutte le politiche di rimozione in contemporanea per scovare eventuali errori di concorrenza. Tale test non è presente nei target del makefile ed esegue solamente le istruzioni lato client (è stato scelto questo approccio per controllare meglio l'output dei due processi).

Considerazioni

Dal punto di vista dell'utente ci potrebbero essere dei problemi di "concorrenza" dovuti alla presenza della `openFile()`.
Ad esempio, se vengono eseguiti quasi contemporaneamente le seguenti istruzioni:

1. `-f cs_sock -a file1,ciao`
2. `-f cs_sock -a file1,mondo`

Non è detto che il contenuto di *file1* sia "ciao mondo", ma potrebbe essere "mondo ciao". Questo accade perché le istruzioni sono composte da due funzioni dell'API: `openFile()` e `appendToFile()`, e di conseguenza si potrebbe verificare il seguente scenario:

1. Client 1 esegue: `openFile()`;
2. Client 2 esegue: `openFile()`, `appendToFile()`, `closeFile()`;
3. Client 1 esegue: `appendToFile()`, `closeFile()`.