
UNIVERSITÀ DI PISA

# Progetto Laboratorio di Sistemi Operativi "File Storage"

## Corso A

## 1. Descrizione del problema:

Il progetto prevede la realizzazione di un ***File Storage Server*** che permette la memorizzazione di file in memoria principale. La capacità dello storage è fissata all'avvio; il sistema ***File Storage*** prevede opportuni meccanismi di rimpiazzamento dei file memorizzati in caso di capacity misses (cioè superamento del numero massimo di file memorizzabili e/o superamento della dimensione massima in Mbytes dello storage).

## 2. Architettura del sistema:

L'architettura del sistema è del tipo client-server; Il server accetta connessioni di tipo AF_UNIX da parte dei client. Ogni client effettuerà una sola connessione verso il server sulla quale invierà delle richieste seguendo un protocollo di comunicazione richiesta-risposta.

### Server

Il processo server ha una struttura multi-threaded che segue lo schema manager-worker; All'avvio viene fissato il numero di thread worker del threadpool. I thread worker estrarranno i file descriptor da servire da una lista di lavoro condivisa opportunamente sincronizzata. All'estrazione del un valore speciale '-1' il thread worker estrattore termina la sua esecuzione.

All'avvio una procedura automatica imposta i parametri di default del server cioè:

- ***a.*** *Numero di thread worker del threapool*
- ***b.*** *Numero massimo di file memorizzabili*
- ***c.*** *Capacità massima dello storage*
- ***d.*** *Nome del socket di comunicazione*
- ***e.*** *Nome del file di log*
- ***f.*** *Politica di rimpiazzamento*

Successivamente verrà effettuata una scansione del file testuale di configurazione per impostare dei valori personalizzati ai parametri sopra riportati. Il file di configurazione dovrà avere il seguente path relativo ***./txt/config_file.txt*** . Il file testuale dovrà avere il seguente formato:

```
NWORKERS numero
MAX_FILE_N numero
MAX_FILE_DIM numero
SOCKET_FILE_PATH path
LOG_FILE_NAME path
REPLACEMENT_POLICY [fifo|lru]
```

```
LOG_FILE_NAME ./txt/journal.txt
NWORKERS 4
MAX_FILE_N 10
MAX_FILE_DIM 1
SOCKET_FILE_PATH ./SocketFileStorage.sock
REPLACEMENT POLICY fifo
```

In caso di errori nella fase di scansione delle opzioni la procedura ripristinerà l'intera configurazione di default del server. *Esempi di errore possono essere una riga del tipo "NWORKERS 5 y" oppure l'inserimento di una politica di rimpiazzamento non supportata.*

```
Creazione del file di log avvenuta con successo!
--------------------
Fri Jan 14 10:05:02 2022
Server acceso!
```

Durante l'attività del server, le informazioni rilevanti riguardo le operazioni e le connessioni vengono riportate in un file di log il cui nome è stato specificato in fase di configurazione. Ogni entry del file di log conterrà il timestamp e la descrizione dell'evento.

Dopo la configurazione iniziale e l'apertura del file di log il server inizializzerà il threadpool e le strutture dati di supporto. Ogni thread worker eseguirà la procedura ***WorkerFun*** che prende come parametro un riferimento alla pipe di comunicazione con il server (che è il componente ***manager*** dell'architettura manager-worker). Ogni worker preleverà dalla coda un file descriptor da servire e leggerà su tale connessione un intero che rappresenta il numero dell'operazione richiesta. Tale intero sarà un parametro della funzione ***ExecuteRequest*** invocata subito dopo.

```
int ExecuteRequest(int fun,int fd){
    switch(fun){
        case (3):{ //Operazione Open File…
        case (4):{ //Operazione Read File…
        case (5):{ //Operazione Read N File…
        case (6):{ //Operazione Write File…
        case (7):{ //Operazione AppendToFile…
        case (8):{ //Operazione Lock File…
        case (9):{ //Operazione Unlock File…
        case (10):{ //Operazione closeFile…
        case (11):{ //Operazione Remove File…
        case (12):{ //Operazione "Terminazione"…
        default:{ //Operazione Sconosciuta…
    }
    return 0;
}
```

La funzione si occuperà di leggere dal client tutti i parametri necessari per le varie operazioni ed invocherà le funzioni che effettivamente eseguono le operazioni richieste.
Al termine dell'operazione richiesta il thread worker invia al server una stringa (di dimensione 7 bytes di cui 4 riservati alla memorizzazione del fd) con un particolare formato che contiene il file descriptor servito e l'indicazione riguardante la sua terminazione. Se il primo carattere della stringa è ***F*** significa che il client connesso sul file descriptor riportato all'interno della stringa ha terminato le operazioni; il server quindi non dovrà re-inserire nuovamente il file descriptor nella lista di lavoro del threadpool. In caso contrario il client non ha terminato le richieste perciò il server inserirà nuovamente il suo file descriptor nella lista di lavoro; in questo modo tale client può essere servito da un thread worker per la successiva richiesta. Da notare che con lo schema master-worker ogni worker esegue richieste di client diversi.

| F/C | : | fd | fd | fd | fd | \0 |
|---|---|---|---|---|---|---|

Il server avvia inoltre un thread gestore dei segnali che gestirà in modo opportuno i segnali SIGQUIT, SIGHUP e SIGINT. Il thread gestore dei segnali esegue la procedura ***SignalHandlerFun*** la quale prende per input una ***struct signal_handler_thread_arg*** che incapsula il signal set ed una pipe di comunicazione con il server. Per mezzo di quest'ultima pipe il signal handler invia al server un intero che codifica la tipologia di segnale ricevuta: **1** per il segnale di terminazione immediata (SIGQUIT e SIGINT), **2** per il segnale di terminazione graceful (SIGHUP) cioè al termine dell'esecuzione di tutti i client connessi al momento della ricezione del segnale.
Il server al momento della ricezione dell'intero sulla pipe inserirà dei file descriptor terminatori nella lista di lavoro dei thread worker in modo da interromperne l'esecuzione. In caso di ricezione dell'intero '1' il server inserirà immediatamente i terminatori e passerà alla fase di chiusura deallocando le risorse e effettuando la stampa delle statistiche finali. In caso di ricezione dell'intero '2' il server respingerà ogni connessione effettuata successivamente ed inserirà i terminatori solamente quando non ci saranno più client connessi.

## 3. Stored File

Il tipo ***stored_file*** rappresenta il file che sarà memorizzato in memoria principale; contiene al suo interno una serie di attributi descrittivi e sistemi per l'accesso concorrente che saranno utilizzati e/o modificati dalle varie operazioni sul file.

```c
/*-----File-----*/
typedef struct stored_file{
    char* content; //contenuto
    size_t size; //dimensione

    Node* opened_by; //lista di client che hanno aperto il file

    struct timeval creation_time; //tempo di creazione del file
    struct timeval last_operation; //tempo dell'ultima operazione sul file

    int fd_holder; //fd della connessione del proprietario del file; Se -1 il file e' sbloccato
    pthread_mutex_t mutex_file; //lock del file
    pthread_cond_t is_unlocked; //variabile di condizione per l'attesa dello sblocco del file

    int clients_waiting; //numero di client in attesa di acquisire la lock sul client
    int to_delete; //flag che indica che il file deve essere eliminato
    pthread_cond_t is_deletable; //variabile di condizione per attesa delle condizioni necessarie per l'eliminazione
}stored_file;
```

L'attributo ***content*** conterrà un puntatore al contenuto del file e ***size*** la dimensione di esso. La lista ***opened_by*** conterrà i file descriptor che hanno effettuato l'apertura del file per compiere operazioni su di esso. Le due ***struct timeval*** memorizzeranno i tempi di creazione ed ultima modifica che saranno utilizzati dagli algoritmi di rimpiazzamento. L'intero ***fd_holder*** conterrà il file descriptor del client che ha effettuato l'operazione di lock sul file. Vi è anche un ***mutex*** per il corretto accesso al file ed una variabile di condizione ***is_unlocked*** che permette ai client che vogliono bloccare un file già bloccato da altro client di mettersi in attesa. L'intero ***clients_waiting*** indica il numero di client in attesa di bloccare quel file. Il flag ***to_delete*** è settato ad 1 se il file deve essere eliminato, a 0 altrimenti. Qualora il flag ***to_delete*** sia impostato ad 1 non è possibile effettuare su tale file operazioni di lock, in modo da evitare il problema per cui alcuni client possano mettersi in attesa di acquisire la lock di un file che poi successivamente sarà eliminato dal proprietario causando attesa indefinita.

Gli ***stored_file*** saranno memorizzati in una tabella hash ***storage*** le cui entries avranno come chiave il path assoluto del file memorizzato. Non è possibile quindi avere più ***stored_file*** con lo stesso pathname. L'utilizzo di una tabella hash permette un accesso più veloce ai file memorizzati rispetto ad un'implementazione dello storage con una lista monodirezionale. La tabella hash è opportunamente sincronizzata per mezzo del ***mutex_storage*** in modo che più thread worker possano operare su di essa correttamente. Per le operazioni sulla tabella hash ho elaborato la libreria *"Dependency free hash table implementation - icl_hash"* di Jakub Kurzak fornita durante le esercitazioni del corso.

## 4. Operazioni sui file

Le operazioni sui file implementate sono quelle riportate dalla specifica del progetto.
É da segnalare che, nel momento in cui il client invia il comando di terminazione della connessione (CloseConnection), il server si occuperà di sbloccare tutti i file da lui precedentemente bloccati in modo da permettere la continuazione delle operazioni sui file da parte di altri client. Tale funzionalità è implementata dalla funzione ***UnlockAllMyFiles.*** Tutte le operazioni restituiscono un oggetto di tipo ***response*** che al suo interno incapsula un intero ***code*** con l'esito dell'operazione ed un ***messaggio*** di massimo 256 caratteri descrittivo dell'operazione o del problema riscontrato.
In merito alle operazioni WriteFile, AppendToFile il progetto prevede che in caso di capacity misses, se il client ha specificato una cartella di destinazione, i file espulsi dagli algoritmi di rimpiazzamento vengano inviati indietro al client. Sono stati implementati gli algoritmi di rimpiazzamento FIFO (che considera il tempo di creazione di un certo file) e LRU (che considera il tempo dell'ultima operazione sul file). In caso di superamento del numero massimo di file memorizzabili (OpenFile con flag O_CREATE) il file espulso non viene inviato indietro al client.

## 5. Client

Il client invia richieste al server File Storage sulla base delle opzioni che riceve dalla linea di comando. Le opzioni ***-h, -f, -D, -d, -t*** e ***-p*** vengono elaborate direttamente in fase di parsing delle opzioni con i relativi argomenti lette da argv. Le restanti operazioni vengono inserite in una lista monodirezionale affinchè possano essere eseguite successivamente considerando anche il delay artificiale eventualmente specificato con -t. In particolare il client manterrà una lista di ***pending_operation***: una struttura che incapsula al suo interno il codice dell'operazione da eseguire, il numero di argomenti dell'opzione e l'array contenente gli argomenti dell'operazione. La correttezza delle operazioni inserite viene controllata in fase di parsing dalla procedura ***getopt*** e dalla funzione ***operation_check.*** *Il* file ***client_utils.c*** contiene una serie di funzioni e procedure di utilità utilizzate dal client.

```
typedef struct operation{
    char option; //codice dell'operazione
    int argc;
    char** args; //argomenti dell'operazione
}pending_operation;

typedef struct Node{
    pending_operation* op;
    struct Node *next;
}operation_node;
```

Dopo aver aperto la connessione con il server mediante l'operazione ***OpenConnection***, il client invoca l'esecuzione della procedura ***Execute_Requests*** che prende in input la lista di comandi. Tale procedura estrarrà le varie operazioni dalla lista ed invocherà correttamente le ***funzioni di API*** contenute nella libreria ***FileStorageAPILib.a*** creata dal ***Makefile***. Ogni file oggetto di una operazione verrà aperto (con gli opportuni flag) prima dell'esecuzione e chiuso al completamento dell'operazione. Eccezione a tale regola è l'operazione ReadNFiles che non prevede l'apertura a priori dei file da leggere in quanto il client non è ancora a conoscenza di quanti e quali file riceverà dal server. Tutti i protocolli di comunicazione delle varie operazioni sono presenti come commento al relativo codice.
Al completamento di tutte le operazioni richieste il client chiude la connessione con il server mediante l'operazione ***CloseConnection***.

## 6. Makefile

Il Makefile contiene i target phony previsti dalla specifica cioè ***all*** per generare gli eseguibili e ***clean*** per ripulire la directory di lavoro dai file/cartelle generati, moduli oggetto, logs e librerie. Il target phony ***result_clean*** permette di eliminare le cartelle "Espulsi" e "Letti" generate dai tests. I target phony ***test1***, ***test2*** e ***test3*** permettono di eseguire i relativi test. Al termine di ogni test il target farà partire anche lo script statistiche.sh per il parsing del file di log.

## 7. Statistiche

Lo script statistiche.sh si occuperà di effettuare il parsing del file di log prodotto dall'esecuzione del server. Inizialmente lo script verifica l'esistenza del file di log cercando il file ***txt/logfile.txt*** *(*è il nome di default che il server assegna al file di log). Qualora il file di log con nome standard non venga trovato, lo script leggerà il file di configurazione  per trovare il nome corretto del file di log. Se anche questa ricerca non va a buon fine lo script termina con un errore.

## 8. Repository

Link al repository Github: https://github.com/SimoneSchiavone/FILE_STORAGE