# 

# Fabio Sirugo

# Mat: 533099

RELAZIONE OS\_STORE

# Server.c

# DEFINE Server

#define CONNECT\_MAX 20

#define CONNECT\_MIN 10

#define MAXIMUM\_PATH 4096.

#define SOCKNAME "objstore.sock"

#define SIZE\_HEADER 16

#define MAX\_SIZE\_FILE 255

#define BLOCCO\_DATI 2048

MAXIMUM\_PATH : In questo contesto sto ipotizzando che il path massimo è di 4096 byte

SIZE\_HEADER : la dimensione MASSIMA che posso avere considerando i comandi ( in realtà usato poco).

MAX\_SIZE\_FILE : impongo dei vincoli alla dimensione massima del nome di un file.

BLOCCO\_DATI : ampiamente usato nella funzione myRead() per leggere BLOCCO\_DATI alla volta.

Con CONNECT\_MAX e CONNECT\_MIN stabilisco un range di client che possono essere attivi contemporaneamente.

# Funzionamento Server

Inizia con la funzione start\_server(), questa ha il compito di creare il socket eseguendo i comandi socket(), Bind() e listen(), il tutto controllando se vi sono errori. Come passo successivo viene creata la directory “data” utilizzando il metodo **char \*createData()** che restituisce un percorso fino alla directory data (.../data).createData viene chiamata non solo per creare la Dir data ma anche per inizializzare la variabile globale path\_iniziale. Non avendo incontrato errori in questa fase il server invoca il dispatcher.

**void dispatcher (int fd)**

Come prima cosa il metodo inizializza le variabili globali :

show\_stats = 0;

fine = 0;

numero\_anomalie\_client = 0;

stored\_Data = 0;

client\_connessi = 0;

numero\_file = 0;

numero\_fileRimossi = 0;

che serviranno per stampare le statistiche di os\_store.

Il dispatcher prepara i valori e gli attributi necessari per accettare le richieste dei client e per generare i thread.

Il dispatcher inizia un ciclo che si appresta ad accettare le richieste degli utenti e a creare i thread per gestire i clienti.

All’interno del ciclo inoltre verranno effettuati dei check per vedere se il server sta attualmente gestendo un numero elevato di client.

In tal caso il ciclo viene messo temporaneamente in wait fino a quando il numero di client da gestire sia un numero fattibile.

Il ciclo termina quando fine == 1 ovvero quando arriva il segnale di terminazione.

**static void \*gestore\_Clienti(void\*arg)**

Il metodo riceve il fd passato come argomento e lo utilizza per effettuare la prima read.

Come scelta implementativa ho deciso di imporre una dimensione fissa al numero di caratteri che può avere il nome di un file per considerare solo dei nomi utenti realistici.

**“Linux** has a **maximum** filename length of 255 characters for most filesystems (including EXT4), and a **maximum** path of 4096 characters.”.

(Anche se in realtà è anche meno).

Di conseguenza alla prima read utilizzerò un buffer di MAX\_SIZE\_FILE(== 255Byte) +12.

Una volta effettuata la read, chiamo la funzione char \*\*tokenizer(char \*str , int \*num) che restituisce la stringa str splittata su “ \n” e num contiene il numero di token restituiti.

gestore\_Clienti controlla che il comando ricevuto sia effettivamente una REGISTER, controlla inoltre che la dimensione del nome rispetti la lunghezza.

Una volta effettuati i vari controlli viene creata la Directory del cliente che si è appena registrato.

Il server una volta ricevuta la REGISTER (ovvero la os\_connect) potrà ricevere le operazioni: RETRIEVE, LEAVE, STORE e DELETE.

Inizia un ciclo che termina solo quando il server riceve la os\_disconnect.

All’interno di questo ciclo il thread legge un solo Byte per vedere quale operazione dovrà effettuare.

Se il char letto è una ‘R’ chiama la myRetrieve(), se ‘L’ chiama la myLeave, se ‘D’ chiama la myDelete() e se ‘S’ chiama la myStore().

Ho fatto questa scelta implementativa perchè insieme al vincolo imposto prima (il nome dei file ha una dimensione massima di 255 Byte) riesco a gestire meglio i vari HEADERS.

Nel caso in cui ricevo una LEAVE, DELETE o RETRIEVE so già quanti byte leggo e quindi faccio in totale 3 read, mentre nel caso della STORE non so con certezza quante letture farò.

Una volta uscito dal ciclo termina il thread.

**char \*myRead(int fd)**

La funzione myRead legge a BLOCCO\_DATI byte alla volta e controlla se all’interno è presente il carattere ‘\n’. Una volta incontrato il carattere newline termina e restituisce la stringa inviata dal client fino al newline.

**int myStore(int fd , char \*nome)**

Inizialmente svolge dei controlli per verificare se fd e nome sono validi e legge altri 5 byte per vedere se il comando ricevuto è effettivamente una store.

Dopo di che chiama myRead() poichè ha bisogno di leggere il carattere \n.

A questo punto myStore possiede 3 dati importanti

1)nome\_file : il nome del file da salvare.

2)Lunghezza\_letta: la lunghezza len del file.

3)split\_data : parte iniziale del dato da salvare.

myStore() inizializza int dif;

dif conterrà la differenza tra Lunghezza\_letta e strlen(split\_data).

int dif = numero\_caratteri\_totali - numero\_caratteri\_letti.

La variabile dif serve a capire se la myRead ha già letto tutto il blocco dopo \n.

Se dif == 0 allora ha letto tutto il blocco.

Se dif != 0 allora myStore legge dif byte e concatena la parte residua appena letta alla parte iniziale per ottenere il blocco finale.

(Nota: questo è uno dei pochi casi in cui posso utilizzare la readn, perchè so esattamente quanto devo leggere.)

A questo punto myStore ha tutti i dati necessari per salvare il blocco con il nome di “nome\_file”.

(Come scelta implementativa) Se nome\_file è già presente in directory questo verrà cancellato per far posto a quello nuovo.

**int myDelete(int fd , char \*nome)**

Inizialmente svolge dei controlli per verificare se fd e nome sono validi e legge altri 6 byte per vedere se il comando ricevuto è effettivamente una Delete.

Dopo di che esegue una read per ricevere il nome del file da eliminare.

Chiama tokenizer per splittare su “ \n” ed avere la conferma di aver ricevuto il nome file per intero.

Costruisce il path ed effettua una fopen per aprire il file (se esiste).

Se la fopen fallisce vuol dire che il file non è presente nella directory e manda al client un segnale di KO, altrimenti elimina il file.

**int myRetrieve(int fd , char \*nome)**

Inizialmente svolge dei controlli per verificare se fd e nome sono validi.

Successivamente esegue una read di (MAX\_SIZE\_FILE +8).

La stringa ottenuta dalla read viene splittata mediante tokenizer e verifica che il comando ricevuto sia effettivamente una retrieve e che il token successivo sia un nome\_file accettabile.

Verifica se nome\_file è presente o meno, se presente calcola la lunghezza del file nome\_file e salva la lunghezza in len .

Effettua poi una read del file richiesto dal client e salva il risultato in data.

A questo punto il server invia al client “DATA len \n data”.

**int myLeave(int fd , char \*nome)**

Controlla che il comando ricevuto sia effettivamente LEAVE dopo di che chiude il fd.

# Client.c

#include "utils\_client.h"

#define BASE\_STRINGA "0101010101"

Il client fondamentalmente chiama la libreria “utils\_client.h" per poter effettuare i test.

I test usano BASE\_STRINGA per generare dei blocchi fissi da concatenare per poi usarli nei test.

I file hanno nome File\_1.txt, File\_2.txt…..File\_20.txt.

I blocchi vengono generati concatenando di volta in volta BASE\_STRINGA, precisamente accade che all’iterazione ‘N’ la stringa risultante sarà il risultato delle strcat delle vecchie iterazioni ‘N-1-2….’ di BASE\_STRINGA.

Test1 verifica la store.

Test2 verifica la retrieve.

Test3 verifica la delete.

Il main inizializza la variabile globale failure a zero.

Controlla che vengano passati bene i parametri da linea di comando :

nome=argv[1]

tipotest=argv[2]

Effettua inizialmente la connect con nome e se va a buon fine effettua uno switch case in cui controlla il tipo di test da effettuare.

Stampa gli eventuali fallimenti ed esegue la disconnect.

# utils\_client.c

#include "utils\_client.h"

#define BLOCCO\_DATI 2048

int fd\_skt;

Contiene l’implementazione dei metodi :

int os\_connect(char \* name)

void \* os\_retrieve(char \* name)

int os\_store(char \* name, void \* blo, size\_t len)

int os\_delete(char \* name)

int os\_disconnect()

**int os\_connect(char\*name)**

Per prima cosa controlla name != NULL e verifica se la lunghezza del nome rispetta i vincoli di lunghezza.

Una volta passati i controlli effettua la connect al socket inizializzando la variabile globale che conterrà il file descriptor del cliente fd\_skt.

Effettuata la connessione e nel caso in cui non ci siano errori, costruisce la stringa ed invia al server la stringa “REGISTER name \n”, dopo di che resta in attesa dell’esito.

**void \* os\_retrieve(char \* name)**

Controlla name != NULL e verifica se la lunghezza del nome rispetta i vincoli di lunghezza.

Invia al server la stringa “RETRIEVE nome \n” dopo di che aspetta la risposta del server.

Se riceve KO ritorna NULL, altrimenti chiama myRead ed ottiene la una stringa contenente \n.

os\_retrieve inizializza int dif;

dif conterrà la differenza tra Lunghezza\_letta e strlen(dato\_letto\_dopo\_newline).

int dif = numero\_caratteri\_totali - numero\_caratteri\_letti.

La variabile dif serve a capire se la myRead ha già letto tutto il blocco dopo \n.

Se dif == 0 allora ha letto tutto il blocco.

Se dif != 0 allora os\_retrieve legge dif byte e concatena la parte residua appena letta alla parte iniziale per ottenere il blocco finale.

(Nota: questo è uno dei pochi casi in cui posso utilizzare la readn, perchè so esattamente quanto devo leggere.)

A questo punto os\_retrieve ha tutti i dati necessari per restituire il blocco .

**int os\_store(char \* name, void \* blo, size\_t len)**

Come prima cosa controlla i parametri della funzione.

Dopo di che prepara la stringa da inviare allocando lo spazio necessario ed invia "STORE name len \n blo".Dopo di che aspetta la risposta del server.

**int os\_disconnect()**

La disconnect semplicemente invia la stringa “LEAVE \n” ed aspetta l’esito dal client per vedere se la disconnessione è avvenuta con successo o meno.

Anche se il server gestisce in modo tale da chiudere sempre la connessione, altrimenti il client resterebbe in attesa di finire senza niente da fare.

# Testing

Il server può essere testato tramite numerosi Script.

IMPORTANTE :

Tuttavia per un testing corretto l’utente è invitato ad effettuare i seguenti comandi tra un test e l’altro:

make clean -> make -> make testscript.sh

Altrimenti le statistiche mostrate dal server potrebbero non essere consistenti perché per calcolare la dimensione totale dei Byte non utilizzo delle funzioni di libreria, ma utilizzo una variabile globale che viene decrementate o incrementata.

Esempio di dati non consistenti:

1) Avvio il server eseguo tot store e chiudo il server.

2) Avvio il server senza aver fatto la clean dopo il passo 1) ed eseguo tot Delete.

Le statistiche relative al numero di file presenti nella directory data o al numero di byte totali del punto 2) mostreranno dei valori negativi.

# Makefile

Target all : utilizzato per creare l’eseguibile di client e server

Target clean : utilizzato per pulire i file generati da all e rimuovere la directory data ed i suoi file interni

Target test: chiama “make clean -> make -> make test\_base”

Target test\_base: Test base di 100 client < 50 Test1 , 30 Test2 , 20 Test3 >

Target bigtest: 1500 client <500 Test1 , 500 Test2 , 500 Test3>

Target someErrors: (fare la clean prima) lancia 30 Test2 e 20 Test3 e mostra la gestione err.

Target nometroppolungo : lancia dei client con nomi non validi

# Segnali

Globale :

int self\_pipe[2] : che viene inizializzata con una chiamata pipe() nel main().

volatile sig\_atomic\_t show\_stats: inizializzata nel dispatcher.

volatile sig\_atomic\_t fine: inizializzata nel dispatcher.

Entrambe inizializzate nel dispatcher a zero.

Nel main indico quali segnali verranno gestiti :

struct sigaction s;

memset(&s , 0 , sizeof(s));

s.sa\_handler = gestore\_segnali;

if(sigaction(SIGUSR1 , &s ,NULL) == -1) perror("Errore sigaction SIGUSR1");

if(sigaction(SIGQUIT , &s ,NULL) == -1) perror("Errore sigaction SIGQUIT");

if(sigaction(SIGINT , &s ,NULL) == -1) perror("Errore sigaction SIGINT");

if(sigaction(SIGPIPE , &s ,NULL) == -1) perror("Errore sigaction SIGPIPE");

if(sigaction(SIGTERM , &s ,NULL) == -1) perror("Errore sigaction SIGTERM");

La gestione vera e propria dei segnali avviene mediante un thread dedicato.

**signal\_handler\_thread()**

Questo thread utilizza un meccanismo di “Self\_pipe” associato all’utilizzo di una select.

Il thread una volta avviato, entra in un ciclo while che ha come guardia la variabile fine.

Il ciclo si blocca su

esito\_select = select(self\_pipe[0]+1, &rd\_set, NULL, NULL, NULL); che resta bloccato in attesa di ricevere una scrittura dal metodo gestore\_segnali().

Se la select dovesse sbloccarsi, a quel punto il thread verifica esito\_select.

Se esito\_select > 0 allora effettua una readn, altrimenti cicla di nuovo e se necessario si blocca di nuovo sulla select.

Il thread main blocca i segnali così tutti i thread che genera ereditano i segnali bloccati.

**dispatcher (Segnali)**

Nel dispatcher contemporaneamente avremo un’altra select con un time-out di 1 decimo di secondo per la accept.

Questa select avrà il compito di capire se la accept si bloccherà.