FEDERICO BERNACCA 536683

Il progetto, volto a simulare un sistema client-server, secondo le direttive assegnate, è stato realizzato mediante le scelte implementative che verranno illustrate a seguire.

All'avvio il server si mette in attesa di ricevere connessioni ripetutamente: per ognuna di queste viene creato un thread dispatcher che esaudisce le richieste del client e rimane in esecuzione finché non viene richiesta la disconnessione.
Tutte le richieste, inviate come singola stringa da parte del client, vengono tokenizzate e inserite dal server, parola per parola, in un char **buffer, cosicché ad ogni posizione del buffer, corrisponda esattamente una parola.
Alle funzioni implementate è stata aggiunta **os_register()**, con la quale il client si registra nello store, ovvero viene creata la directory col suo nome all'interno della directory "data". Se la cartella con il nome dell'utente è già presente, l'operazione fallisce e viene segnalato al client.
Segue **os_connect()**, con la quale l'utente può usufruire delle funzionalità che l'object store permette.
Viene restituito "true" se l'utente è registrato, ma non è ancora connesso, altrimenti fallisce. In caso di esito positivo, il suo nome viene memorizzato per tutta la durata della sessione in un char *user e viene inserito in una tabella hash globale che mantiene tutti gli utenti connessi in quel momento.

Per salvare gli utenti connessi al server è stata utilizzata come struttura dati una tabella hash, con gestione delle collisioni tramite linked list.
La struttura dati linked list consta di tre campi:

```
/* linked list */
typedef struct _node {
    int key;
    char *item;
    struct _node *next;
} node;
```

Al char *item viene associata una chiave univoca int key al momento dell'aggiunta nella tabella hash, questo fa si che, una volta avuto accesso alla giusta lista tramite la funzione hash, l'operazione di rimozione di un nodo (utente) risulti molto più efficiente visto che il nodo viene confrontato con la chiave piuttosto che con char *item
La struttuta dati hashtable è così composta:

```
/* struttura in cui salvo gli utenti connessi in sessione */
typedef struct hashtable_ {
   node **list;
   int nElem;
   int size;

   /* avrò un mutex per ogni lista della tabella hash */
   pthread_mutex_t *mutex;

   /* mutex per aumentare la variabile nElem */
   pthread_mutex_t incr;
} hashtable;
```

size è la dimensione dell’”array verticale” di puntatori a linked list nelle quali vengono salvati gli utenti connessi.
nElem è il numero di oggetti totali attualmente presenti nella tabella (in questo caso gli utenti online), la quale ha associato un mutex per essere incrementata e decrementata. è condivisa dalle varie liste e quindi ha bisogno di essere modificata in mutua esclusione quando un utente si connette/disconnette dal server.
***mutex** è un array di mutex di cui si fa uso per gestire gli accessi alle varie linked list in mutua esclusione.
Come chiave univoca associata ai client viene utilizzato il file descriptor che gli viene assegnato al momento della connessione con il server, la quale potrà poi venire riutilizzata al momento della disconnessione da parte di un client, e la connessione di un altro.
La funzione hash che “smista” nella corretta linked list è definita come la somma degli interi associati ai caratteri della parola moltiplicati per il primo 31.

Per quanto riguarda **os_store()**, quando il server riceve la richiesta di memorizzazione del file “filename”, esso crea il path “.../wd/data/user" e controlla se “filename” è gia presente in questo path, se lo è invia al client un messaggio di errore, altrimenti salva il blocco di dati in “.../wd/data/user/filename”. In caso di esito positivo, prima di inviare al client la risposta, controlla che il blocco di dati scritto sul file non sia corrotto, confrontandolo con quello inviato dal client tramite memcmp.
Quando un client richiede **os_retrieve()** di “filename”, come prima cosa viene creato il path “.../wd/data/user" e controlla se il file è presente in questa directory. Se lo è, viene creato il path ".../wd/data/user/filename", viene aperto questo file, viene estratta la sua dimensione, viene allocato un buffer nel quale verranno memorizzati i dati di “filename” e viene inviato al client come risposta, il quale contenuto verrà verificato nel client di test tramite memcmp.
Per **os_delete()**, viene creato il path ".../wd/data/user/filename", successivamente si prova la rimozione, nel caso di fallimento, si testa errno e come risposta al client viene inviato il messaggio di errore, altrimenti il file è rimosso e viene segnalato.
os_disconnect() rimuove char *user dalla tabella hash e fa terminare la connessione col server, chiudendo il file descriptor a lui associato, e facendo terminare il thread dispatcher a lui “assegnato”.

Nel server ho utilizzato quattro variabili globali:

```
/* lock per controllare la directory data, alla quale devo accedere in mutua esclusione*/
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/* la setto a true quando ricevo un segnale */
volatile bool KILL_SERVER = false;

/* la setto a true quando ricevo SIGUSR1 */
volatile bool _SIGUSR1_ = false;

/*  salva gli utenti connessi in sessione */
hashtable *ht;
```

Il primo mutex viene utilizzato quando un utente desidera registrarsi. Si sceglie di eseguire l'operazione in modo atomico perché se due client con lo stesso nome volessero registrarsi, potrebbero sorgere problemi.

Per la gestione dei segnali ci si avvale di due handler:
uno viene "chiamato" quando il segnale è SIGUSR1 ed entrambe le variabili vengono settate a true, l'altro invece entra in funzione per gli altri segnali e setta a "true" solo KILL_SERVER.
I segnali vengono catturati solo dal main, spetta a lui il compito di gestirli mentre nei vari threads vengono ignorati tramite maschera.
Quando arriva un segnale, vengono rifiutate ulteriori connessioni in arrivo, mentre i client connessi finiscono le proprie operazioni, una volta disconnessisi (la hashtable ha il campo nElem == 0) viene chiuso il server. Se il segnale era SIGUSR1, prima di chiudere, stampo le informazioni sullo STORE, tra cui:
nomi di tutti i client registrati, numero e size dei suoi file (le dir "." e ".." non vengono conteggiate), utenti attualmente connessi, utenti registrati, oggetti totali degli utenti e size totale dello store.

Nel client di test ho effettuato le seguenti scelte:
come prima cosa, l'utente passato da riga di comando si registra e si connette.
Per i test di tipo 1, i suoi dati verranno salvati in file che hanno per nome le lettere dell'alfabeto da "a" fino a "t" (20 come richiesto). I dati salvati saranno "questa_e'_una_stringa_di_prova_" se il nome del file è pari, altrimenti saranno una sequenza crescente di interi da 0 fino a 9, tutto incrementato ad ogni iterazione.
os_store() può fallire se il file da memorizzare è già presente oppure se è corrotto.
Per i test di tipo 2, si richiede os_retrieve() dei file, per ogni utente, con nomi da "a" fino a "z", se il blocco dati viene restituito, ci si assicura che sia valido tramite memcmp.
os_retrieve() può fallire se il file richiesto non è presente oppure se è corrotto.
Per i test di tipo 3, si richiede os_delete() dei file, per ogni utente, con nomi da "a" fino a "z".

Per salvare il report delle operazioni degli utenti, e scriverla su testout.log ho utilizzato un'ulteriore struttura dati:

```
/* struttura in cui salvo l'esito delle operazioni */
typedef struct outcomeOp_ {
   /* numero op  */
   int nOperations;

   /*numero op fallite */
   int failedOp;

   /* numero op a buon fine */
   int succeededOp;

   /* lista che salva le op fallite */
   node *failedOpName;

   /* lista che salva le op a buon fine */
   node *succeededOpName;
} outcomeOp;
```

Sul file di log verranno scritti i seguenti risultati:
batteria di test effettuata, utente che ha eseguito la batteria, operazioni effettuate, operazioni effettuate con successo, operazioni fallite e nome di queste operazioni.
Per eseguire i test, i client vengono presi dal file di testo “nomi.txt”, contenente 50 nomi che verranno utilizzati da “test.sh”.
Una volta terminato “test.sh” verrà eseguito “testsum.sh”, che stamperà su schermo il file “testout.log” con in aggiunta i seguenti report:
client lanciati in totale, operazioni effettuate per batteria in totale, operazioni eseguite con successo per batteria e operazioni fallite per batteria.
Eseguendo “memorytest”, verrà lanciato lo script “memorytest.sh”, che equivale a “test.sh”, con la differenza che i client verranno lanciati con valgrind. Una volta terminata l’esecuzione verrà lanciato uno script che, analizzando il report di valgrind, stamperà a schermo se si hanno avuto memory leak o meno.

Le funzioni utilizzate sia da client sia dal server si trovano in “commons.c”, le strutture dati in “struct.c”, la libreria richiesta ha file c “requests.c” e per finire, nel file header “error.h” sono presenti alcune macro per la gestione e stampa degli errori.