UNIVERSITÀ DI PISA

Facoltà di Informatica

OBJECT STORE

Progetto in C

Anno accademico – 2018/2019

Corso: Sistemi Operativi Laboratorio

Matricola: 561333
Studente: Chenxiang Zhang
Docente: Massimo Torquati

1. Descrizione del problema

Vedi il file *testo_problema.pdf* incluso nella cartella.

2. Architettura della soluzione

Descrizione della soluzione

Un processo chiamato *oclient* manda una richiesta a un altro processo *oserver* attraverso un socket creato per la comunicazione secondo il modello architetturale *client-server.*

La prima richiesta da parte del client al server è la funzione di sistema connect() (fig.1) che stabilisce e instaura la connessione attraverso un socket file descriptor, un indicatore astratto usato per le successive comunicazioni.

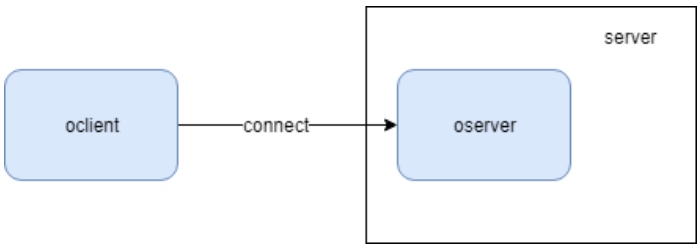

Fig.1 - client esegue il connect

Il server, che era in ascolto con la syscall *accept(…),* una volta accettata la connessione spawna un thread in modalità detached che avrà come compito la gestione di tutte le successive richieste del client. (fig.2)

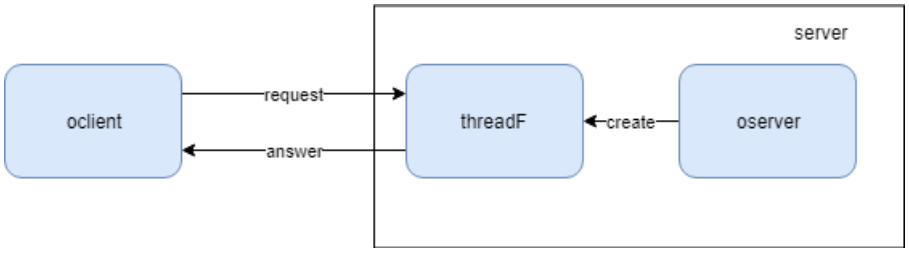

Fig.2 – server accetta la connessione e crea thread worker

Una volta che i clienti si sono connessi, possono iniziare a mandare le request al proprio threadF associato e aspettando di ricevere una risposta per valutare l'esito dell'operazione. (fig.2.1)

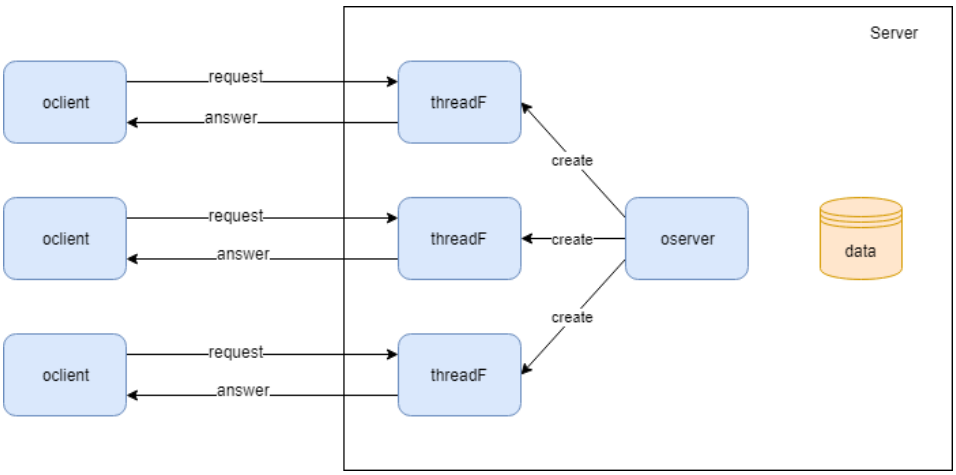

Fig.2.1 – esecuzizone di più client

3. Realizzazione del client

Implementazione

Il programma *oclient* è composto dal linking di oclient.o e libAccess.a. (fig.3)

- **libAccess.a** è una librearia statica che contiene le funzioni con i comandi per accedere al server: os_connect, os_store, os_retrieve, os_deleve, os_disconnect.
  E' composto da access.o e dalla sua interfaccia access.h, quest'ultimo include anche i due header connection.h (contiene le macro per la comunicazione client-server) e utils.h (contiene le macro per controllare i valori di ritorno delle funzioni)

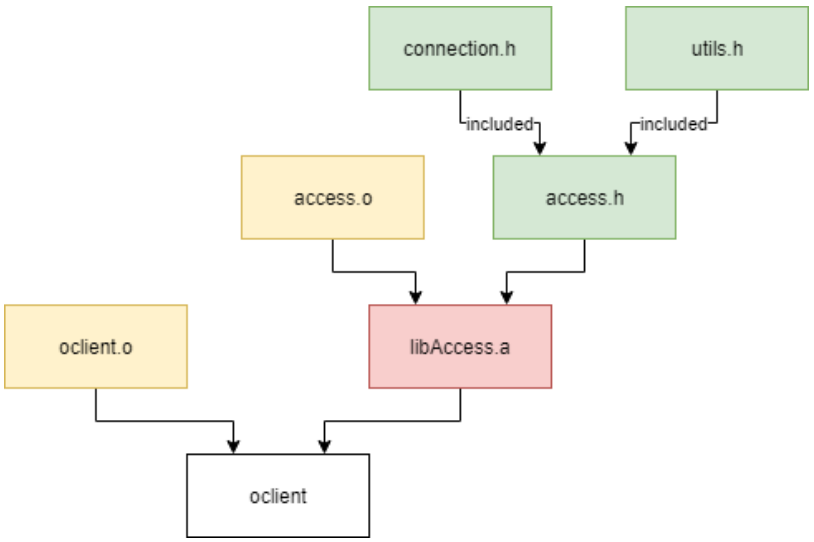

Fig.3 – oclient linking

Funzionalità

Il client inizia la sua esecuzione controllando la validità degli argomenti passati all'avvio con la funzione **check_args(…)**, subito dopo chiama la funzione della os_connect() per stabilire una connessione. Se la richiesta fallisce termina la sua esecuzione altrimenti continua eseguendo il numero del test che gli è stato passato come argomento.
Alla fine printa l'esito delle operazioni e esegue la disconnect.

4. Realizzazione del server

Implementazione

Il programma *oserver* è composto dal linking di oserver.o, structure.o, threadF.o, libUtils.a (fig.4)

- **structure.o** è il file oggetto compilato dal sorgente structure.c che contiene la struttura dati client_t con le relative funzioni per gestirlo come una linked-list.
  Utilizzando anche la pthread_mutex_t gestiamo la mutua esclusività delle funzioni per manipolare la linked-list.
- **threadF.o** è il file oggetto compilato dal sorgente threadF.c il quale contiene tutte le funzioni per gestire il thread: spawn_thread(), manage_request(), send_message().
- **libUtils.a** è una libreria statica creata con lo scopo di contenere le funzionalità generali così come le macro per controllare i valori di ritorno delle syscall e funzioni come mystrdup.

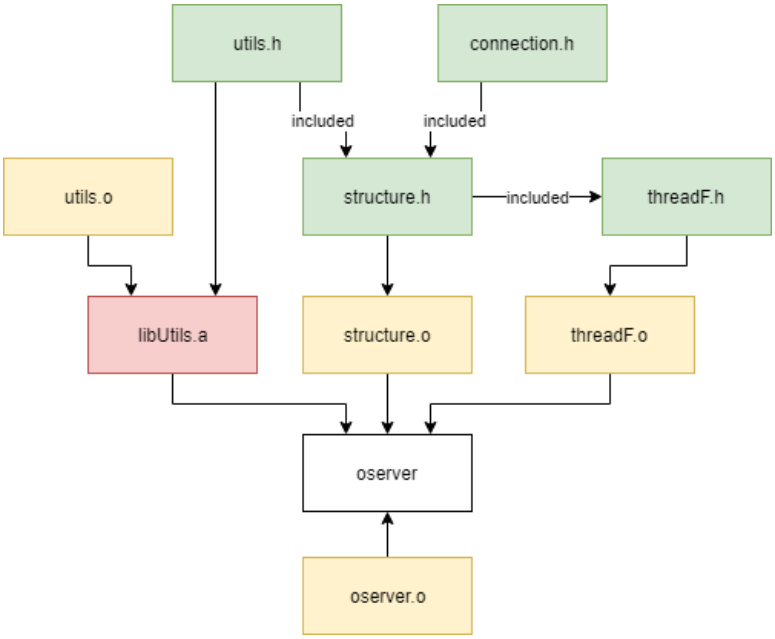

Fig.4 – oserver linking

Funzionalità

Il server inizia la sua esecuzione creando la cartella **data** dove all'interno verranno create le cartelle degli utenti registrati con i relativi oggetti salvati.
Successivamente attiva la funzione **signal_manager()** per gestire il segnale SIGUSR1, una volta completato crea il socket e si mette in ascolto per ricevere le richieste di connessione da parte del client, spawnando un nuovo **thread** in modalità detached per ogni richiesta accettata.
Il server termina solo se riceve un segnale di terminazione, altrimenti rimane sempre attivo.

Segnali

Il segnale **SIGUSR1** è gestito da un signal handler che setta una variabile globale a 1. Questa variabile quando attivata, attiva anche la funzione **print_status()** che mostra alcune informazioni del server: il numero di clienti online, il numero di oggetti salvati nel server e la dimensione totale di tutti gli oggetti salvati. Una volta printato lo status il server riprende ciò che facevo prima.
La gestione di tutti gli altri segnali sono lasciati di default al sistema operativo.

ThreadF

Ogni thread serve unicamente un client. Un thread inizia la sua esecuzione inizializzando una variabile struttura client_t e un buffer da 512byte. Entra poi in un ciclo while dove ogni volta che un client invia una richiesta, il thread lo gestisce usando la funzione **manage_request().**

La funzione manage_request usa la funzione rientrate **strtok_r(…)** per tokenizzare il messaggio dentro il buffer elaborandolo in base al tipo di richiesta (REGISTER, STORE, RETRIEVE, DELETE, LEAVE). Alla fine della esecuzione manda un messaggio di risposta al client per informarlo dell'esito dell'operazione usando la funzione **send_message()** (OK or KO erromessage) e ritorna il client se tutto è andato a buon fine, altrimenti ritorna NULL che farà uscire il threadF dal suo ciclo while per poi terminarlo facendo le opportune free e fclose per evitare eventuali memory leaks.

5. Testing

Funzioni per il testing

Sono state implementate 3 funzioni richieste nel client per testare il funzionamento del server.

- **Test1()**: La funzione esegue un ciclo for per 20 oggetti da inviare con dimensioni crescenti da 100byte a 100.000byte. Il primo file è di 100byte e i successivi partono da 10.000 con un incremento di 5.000 ad ogni iterazione.
  Ho scelto di utilizzare un singolo messaggio di 5 byte per rendere più facile il calcolo da fare per incrementare la size degli oggetti da inviare.
- **Test2()**: La funzione testa la funzionalità di os_store() inviando un dato e successivamente eseguendo la os_retrieve() verifica che il dato sia uguale a quello inviato.
- **Test3()**: La funzione esegue a os_store() e poi la os_delete() del dato salvato.

Bash files

Sono state create I due file bash testclients.sh e testsum.sh.
Testclients.sh lancia tutte le istanze di oclient richiesti (50xtest1, 30xtest2, 20xtest3) e reindirizza l'output nel file testout.log.
Testsum.sh prende il file testout.log precedentemente creato e controlla tutte le linee di righe aggiornando il contatore dei numero di test eseguiti. Alla fine printa il risultato finale di tutti i 3 tipi di test, mostrando quanti hanno avuto successo e quanti sono falliti.

Memory leaks

Utilizzando il programma **Valgrind** con Memcheck è stato verificato l'assenza di memory leaks sia durante l'esecuzione del server che del client. Tutte le malloc sono state liberate con free con successo, e le invalid write/read sono state risolte gestendo la terminazione delle stringhe con mystrdup.

6. Osservazioni

Scalabilità

Possiamo scalare il programma per ospitare un numero molto più grande di utenti usando le **tabelle hash** come struttura per contenere i clienti in quanto sono molto più veloci.

Atomicità

L'atomicità delle operazione dell'invio dei dati, che coinvolgono solo le operazioni di os_store e os_retrieve, è garantito dalle funzioni **writen** e **readn** dal file connection.h.

Altri segnali

E' possibile gestire anche altri segnali in più aggiornando la maschera e creando gli eventuali funzioni handler associati per ogni segnale.

Client Debug Menù

Questa modalità ti permette di testare le singole funzioni offerte dalla libreria libAccess.a
E' possibile attivarla eseguendolo come ***./client name*** omettendo quindi il numero di test che si vuole provare.