# Relazione Progetto di Sistemi Operativi

(Object Store, settembre 2019)

Raffaele Apetino – N.M. 549220
Laboratorio di Sistemi Operativi
Università di Pisa – Dipartimento di Informatica

## Introduzione

Il progetto richiedeva di realizzare un **object store** basato su una struttura *client e server multithread*. Sinteticamente: il client richiede la registrazione, in seguito richiede di effettuare store di oggetti al server, quest'ultimo si occupa del salvataggio in una cartella univoca per ogni client. Il client può inoltre effettuare la richiesta di un file presente sul server ma anche la cancellazione di esso ed infine si disconnette dal server. Il tutto avviene secondo *socket* locali *AF_UNIX*.

## Le librerie

### objectstorelib

La libreria objectstorelib definisce e implementa la libreria utilizzata dal client per la comunicazione con il server, secondo protocollo richiesto nel testo del progetto. Nel file *objectstorelib.h* sono contenute le segnature e informazioni documentate con Doxygen (per maggiori inf. fare riferimento a quelle).

### rwn

La seconda libreria che ho definito contiene due funzioni che si occupano di chiamare read e write, controllando di eseguire lettura e scrittura di esattamente n byte. Queste funzioni vengono utilizzate nel caso in cui si è a conoscenza del numero esatto di byte che devono essere letti o scritti. È inoltre definita una funzione *readcn()* che si occupa di leggere l'*header* fino a che non trova '\n'.

Tutte le librerie sono linkate staticamente, il che significa che sono racchiuse nel programma a tempo di compilazione.

## Struttura del Server

### Comportamento principale e Thread Detached

Il server è destinato a gestire numerosi client contemporaneamente: la mia scelta è ricaduta su una struttura dati largamente utilizzata anche su progetti ampi e

commercializzati, una **hash-table**. Questa mi permette di accedere in tempo costante, *O(1)*, alle informazioni del client per svariati controlli (come la già avvenuta registrazione oppure il fatto di essere già online). Per questo progetto ho deciso di utilizzare una tabella hash ad *indirizzamento diretto*, poiché i client da gestire con i test sono relativamente pochi e con una buona funzione hash si riesce a non avere collisioni. È in ogni caso facilmente implementabile una tabella hash con liste di trabocco in caso di collisione. La **funzione di hashing** che ho deciso di utilizzare è una funzione basata su numeri primi e shifting ciclico di 5 posizioni (per maggiori informazioni il codice è commentato in modo opportuno). L'hashing (quindi il calcolo dell'indice) avviene sul nome di registrazione del client, quest'ultimo poi verrà salvato in una *struct clientinfo* affiancandosi ad una variabile per controllare se il client è già online e una stringa contenente il *path* personale nel server. La struttura principale del server funziona in questo modo: si avvia e viene configurata la gestione dei segnali, crea il socket e dopo varie operazioni standard si mette in ascolto di client sulla *accept().* Quando arriva un client viene creato un thread detached con funzione *myworker().* Il server si occupa di gestire tutti i client in modo parallelo, ciò significa che ogni client ha bisogno di essere gestito da un singolo thread. Ho scelto di utilizzare i thread detached in modo da non dover tener conto di tutti i *file descriptor* di ogni client. Ogni thread detached all'uscita gestisce autonomamente la propria chiusura senza il bisogno di essere atteso da una *pthread_join().* Una volta attivato il thread, inizia la comunicazione con il client.

## Gestione dei segnali

Nel testo veniva richiesta la gestione dei segnali da parte del server. Personalmente ho gestito: SIGINT, SIGQUIT, SIGTERM, SIGPIPE, SIGSEGV e SIGUSR1. Appena riceve un segnale il server termina, accertandosi che i client eseguano l'ultima operazione per mantenere i dati del server in uno stato consistente. Il tutto viene gestito da un thread handler checksignals. Ho preso cura, prima dell'avvio del server, di settare i bit della

maschera a 1 per bloccare i segnali interessati. Grazie alla *pthread_sigmask* ho settato in OR anche i bit dei thread che andrò ad avviare, in modo tale che vengano anche da essi bloccati. Nell'handler ignoro SIGPIPE così da non far terminare il server se un client termina in modo anomalo. Per tutti gli altri, setto una variabile fire_alarm (obbligatoriamente sig_atomic_t) e termino il thread di gestione dei segnali. Nel caso di ricezione di SIGUSR1 stampo su stdout un resoconto finale. Di fondamentale importanza è la creazione della socket in modalità SOCK_NONBLOCK, in modo tale da non interferire (nella gestione) durante l'esecuzione continua della *accept().*

## Struttura del client

Durante lo sviluppo, anche se non richiesto dal testo, la mia decisione è stata di creare un client interattivo in cui inserire i vari comandi REGISTER, STORE ecc… in modo tale da testare approfonditamente con *valgrind* il comportamento del server e del client. Arrivato ad un risultato stabile ho deciso di scrivere il client per i test. Ogni client è un processo, da riga di comando prende il nome ed il numero del test da eseguire. Dopo la registrazione del client, il test 1 si occupa della creazione di 20 file nella directory del client, con dimensione da 100b a poco più di 100kB ripetendo una stringa di 100 caratteri. In seguito viene eseguita la *os_store()* a cui passo come parametro il nome, il puntatore al file e la dimensione del file. Nel test numero 2 invece apro sequenzialmente i file contenuti nella directory del client controllando con il buffer restituito dalla *os_retrieve()* che essi combacino. Il test 3 è molto semplice e si occupa della cancellazione dei file contenuti della directory server del client tramite la *os_delete().* Infine si procede con la *os_disconnect()* e si stampa il riepilogo client (che sfrutta le variabili globali di tipo sig_atomic_t). Il riepilogo finale contiene le informazioni richieste dal testo e in più il tempo passato in *User mode* e quello in *Kernel mode.*

## Makefile - Test - Script

### Makefile e Test

Il makefile fa utilizzo della Shell bash, e non della Shell standard sh. Questo perché utilizzo il comando *declare* per la dichiarazione dell'array di pid del client test 1, da aspettare prima di mandare in esecuzione i client test 2 e 3. Il makefile comprende i target richiesti dal testo, in particolare il target *test* si occupa dell'esecuzione di 50 client sul test 1 contemporaneamente, aspetta che essi siano terminati ed in seguito effettua altri test distribuiti (30/20) su test 2 e 3 simultaneamente. Tutto lo standard output è ridiretto sul file *testout.log*, che viene letto ed utilizzato dallo script *testsum.sh* eseguito dal makefile come ultima operazione.

### Script

Lo script utilizza bash, il ciclo principale si basa sulla lettura e parsing del file testout.log. Mi occupo di tenere conto del numero di client lanciati per test 1, client per test 2 e test3. Calcolo i client totali lanciati e il numero di anomalie riscontrate in ogni singolo test. In caso di anomalie, salvo in un array il nome del client che ha avuto problemi e ne stampo il contenuto a fine rapporto. Per finire, lo script manda il segnale SIGUSR1 facendo utilizzo del comando *killall.*

## Istruzioni utilizzo del programma

- Spostarsi nella directory contenente i file sorgenti scompattati in precedenza.
- Aprire un terminale nella directory
- Eseguire il comando `make` per compilare
- Aprire un secondo terminale ed eseguire il server lanciando `./objectstoreserver.out`
- Dare il comando `make test` nel terminale principale dove abbiamo precedentemente dato il comando `make`
- È possibile con il comando `make clean` ripulire da tutti i file creati ed eseguibili compilati

### Distribuzioni su cui è stato testato il progetto

Il testing è stato eseguito su *Ubuntu 18.04*, *Debian 9* e *Fedora 29* su VM Parallels, infine anche su XUbuntu (vm consigliata sulla pagina del corso).

## Conclusioni

Una struttura dati utilizzabile, in alternativa alla tabella hash, potrebbe essere una lista, aumentando però il tempo di ricerca. Un'altra alternativa riguardante la scelta di settare la socket in modalità NONBLOCK, sarebbe quella di utilizzare la funzione *poll()* sul file descriptor socket del server. Il compito della *poll()* è di aspettare che accada un evento sulla socket così da poter essere poi eseguita la *accept().* Mi ritengo comunque soddisfatto delle scelte di progetto fatte, poiché con pochi cambiamenti si può portare il progetto a funzionare con numerosi client.