# Relazione progetto di Laboratorio di Sistemi Operativi

Lorenzo Buonanno 525289, 01/09/2019

## Struttura progetto:

Il progetto è costituito da un server, una libreria di supporto per i client, dei client di test, un makefile e degli script di test che eseguono i client di test e ne elaborano i risultati

L'unico file condiviso tra server e client è "utils.h", che contiene definizioni di funzioni e costanti globali utili sia ai client che al server (`true`, `false`, `MAX_FILENAME_LEN`, `write_all`, ...)

### *Server*

Il codice per il server è suddiviso in cinque moduli: `object_store`, `signal_handler`, `connection_handler`, `server_worker`, `stats`

#### `object_store`

Contiene il `main` del programma, crea la cartella di storage se non esiste, maschera i segnali ricevuti sé e dai thread figli, successivamente crea e fa partire i server per gestire segnali e connessioni, con un pipe per far comunicare il server che gestisce i segnali con quello che gestisce le connessioni (comunicazione nell'altra direzione può avvenire direttamente tramite segnali al processo).
Dopo aver chiamato i due thread gestori, non fa altro che aspettare tramite `join` la loro terminazione

#### `signal_handler`

Si occupa della gestione dei segnali: rimane bloccato in ciclo su `sigwait` finché non riceve un segnale, inviando `'S'` al thread di gestione delle connessioni se il segnale è SIGUSR1, inviando `'T'` e terminando altrimenti

#### `connection_handler`

Si occupa della gestione delle connessioni in arrivo. Crea il `socket` tramite cui avverranno le comunicazioni, crea un `pipe` per mandare il segnale di terminazione ai thread worker e resta in ascolto tramite `poll` in un ciclo di nuove connessioni o nuovi segnali.
Se riceve una nuova connessione crea un nuovo `worker` a cui passa `file descriptor` della connessione appena creata e della `pipe` per il segnale di terminazione, facendolo partire come `detached` in modo da non dover avere una struttura dati che memorizzi i thread creati per liberare le loro risorse attendendo la loro fine con `join`.

Se riceve un segnale di terminazione scrive sul socket (comune a tutti i worker) un byte, esce dal ciclo di `poll` e aspetta che siano terminati tutti tramite attesa su una `condition variable active_workers_CV` legata alla variabile condivisa `active_workers`, che indica il numero di thread worker attivi. In questo progetto è stato scelto di limitare al massimo la quantità di memoria condivisa tra diversi thread, per semplificare il codice, aumentarne la sicurezza e la modularità, e non avere problemi di performance legati all'utilizzo di un alto numero di lock.
È stato tuttavia deciso di mantenere il numero di thread attivi nell'unica variabile condivisa utilizzata, invece di chiamare `join` a momento di terminazione su tutti i thread creati, per non dover avere la complessità aggiuntiva di gestire una struttura dati che salvi in memoria tutti i thread creati nella storia del server, cancellando periodicamente quelli inattivi.
La gestione della terminazione è stata effettuata tramite messaggi su pipe invece che scrittura su variabili condivise anche per evitare di dover fare polling periodico sul valore di queste variabili, rendendo il codice meno modulare, meno performante e meno `event driven`.
Se il `connection_handler` riceve 'S' sulla pipe tramite cui riceve i segnali, crea un thread che raccoglie e stampa le statistiche legate allo store

### `stats`

Modulo che serve solo a isolare la logica per la raccolta e la stampa delle statistiche, usa la funzione POSIX `ftw` invece della più recente `nftw` per essere più portabile, stampa il numero di file, di byte memorizzati, di cartelle (ossia di utenti registrati) e di worker attivi. Per il lock di stdout (necessario se vengono inviati contemporaneamente più segnali di stampa statistiche) usa i lock associati a active_workers, garantendo anche che il numero stampato di worker attivi sia corretto al momento della scrittura

### `server_worker`

Questo modulo contiene la maggior parte del codice del progetto, si occupa di tutta la comunicazione con il client e dell'implementazione della logica legata alle varie operazioni.
Riceve `file descriptor` con una `pipe` in lettura per la richiesta di terminazione e una socket per la comunicazione con il client.
Ascolta in loop tramite `poll` bloccante senza timeout per eventi sui due descriptor. In caso di `POLLIN` o errori sul pipe termina (senza leggere il contenuto del pipe, in modo che arrivi `POLLIN` a tutti i worker), ignorando eventuali contenuti del socket.
Quando arrivano dati sul socket li salva in un buffer, memorizzato come thread local, che gestisce la lettura a blocchi fino all'arrivo di un header. Una volta arrivato un header procede allo svolgere la funzione associata e comunicare la risposta al client.
Per la gestione dello stato associato alla connessione usa diverse variabili dichiarate come `static __thread`: is_done, che termina il loop di poll quando false, per provocare la terminazione del worker dalle varie funzioni, il buffer di lettura per poter leggere i messaggi anche se ricevuti in diversi eventi, informazioni relative al file in cui sta avvenendo una STORE e il nome della cartella in cui opera il client.

Il worker non fa assunzioni sulla suddivisione di parti dei messaggi ricevuti nelle varie `read`, ma assume che il client aspetti la risposta a ogni comando prima di inviarne un altro.

## *Client*

La libreria di supporto per la creazione di client è interamente contenuta nel file access_library.c, segue la specifica della consegna del progetto, con l'aggiunta di una funzione per indicare la dimensione dei dati restituiti dalla retrieve

Le altre parti del progetto seguono le indicazioni nel testo indicato nel pdf