Laboratorio di Programmazione Sistema
A.A. 2020/21
Luca Lombardo – Mat. 546688

1. Introduzione al progetto

Il programma realizzato consiste in un'applicazione client-server che consente la memorizzazione di file in memoria principale. Esso è costituito dai seguenti file:

- i due eseguibili **server** e **client**;
- quattro librerie, contenute all'interno della cartella */includes*: ***api***, ***threadpool***, ***fileQueue*** e ***partialIO***;
- un **Makefile**, che consente di compilare gli eseguibili e/o effettuare dei test per il corretto funzionamento del software;
- un file **config.txt**, all'interno della cartella omonima, che mantiene le impostazioni di configurazione del server;
- un file di log dal nome configurabile (default: **logs.txt**), contenuto nella cartella *logs*, all'interno del quale vengono scritte tutte le informazioni sulle operazioni effettuate dal server al termine di ogni sua esecuzione;
- quattro script bash, contenuti nella cartella *script*, tre dei quali utilizzati dal Makefile per eseguire i test, e uno **(statistiche.sh)** che effettua il parsing del file di log e produce un output formattato con varie informazioni;
- alcune decine di file di prova (la maggior parte vuoti) contenuti nella cartella *testFiles*, che servono per eseguire i test.

Tutto il codice del progetto è disponibile anche su **GitHub** (con l'history completa dei commit) all'indirizzo: ***https://github.com/Siryus18/File-Storage-Project***

2. Il server

Il server è un processo **multi-threaded**:

- il thread **main** accetta le connessioni in arrivo dai client su un socket di tipo *AF_UNIX* e di nome configurabile (default: *mysock*). Al termine dell'esecuzione produce, inoltre, un sunto delle statistiche contenente informazioni sulle operazioni effettuate;
- un numero configurabile di thread "*worker*", facenti parte della **threadPool**; ognuno di essi serve una singola richiesta da parte di uno qualsiasi dei client connessi. Se il numero di richieste ricevute dai client in contemporanea è maggiore della dimensione della pool, esse vengono inserite in una lista d'attesa (*pending queue*), anch'essa di dimensione configurabile. Se la lista d'attesa si riempie, ulteriori richieste vengono rifiutate;
- il thread **sigThread**, infine, agisce da "*signal handler*", intercettando i segnali in arrivo al server. Se viene ricevuto un segnale di tipo *SIGINT* o *SIGTERM*, il server termina il prima possibile, ossia non accetta più nuove richieste e chiude tutte le connessioni attive (il sunto delle statistiche viene comunque generato); se il segnale ricevuto è di tipo *SIGHUP*, invece, non vengono più accettate nuove connessioni, ma vengono servite tutte le richieste dei client già connessi. La comunicazione tra il sigThread e il thread main avviene tramite la pipe *sigPipe*.

Il server gestisce la memorizzazione dei file in memoria principale tramite la libreria thread-safe **fileQueue**, appositamente sviluppata e descritta in seguito.

Quando un client invia una richiesta, essa viene data in carico a uno dei thread worker disponibili (o inserita nella pending queue); quest'ultimo riceve quindi il comando tramite un protocollo di domanda-risposta, nello specifico utilizzando delle chiamate alla libreria **partialIO** (descritta in seguito), e lo passa come argomento alla funzione **parser**. Questa funzione, come si intuisce dal nome, effettua il parsing del comando: se esso viene riconosciuto come valido, chiama la procedura opportuna che si occupa di svolgere la specifica operazione richiesta dal client (tipicamente scrivendone il risultato sul file di log). Una volta fatto ciò, il worker comunica al main (tramite la pipe *requestPipe*) che la richiesta è stata servita e che è tornato disponibile.

Alcune operazioni richiedono il set o il reset del flag *O_LOCK* su un file; per gestirle, si fa uso di una struttura dati interna (**waitingT**): trattasi di una coda, implementata come una *linked list*, che tiene traccia di quali client sono attualmente in attesa di ottenere la mutua esclusione su un file. Grazie a questo meccanismo è possibile evitare il *polling* o l'utilizzo di *condition*.

Per garantirne l'atomicità e la consistenza, tutte le scritture sul file di log, nonché le modifiche alla struttura dati waitingT precedentemente descritta sono sincronizzate tramite una *mutex lock*.