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

## 1. Introduzione al progetto

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

- due file eseguibili, **server** e **client**, che permettono di lanciare i rispettivi programmi;
- 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*" fanno parte della **threadPool** (definita nella libreria omonima); ognuno di essi serve una singola richiesta da parte di uno qualsiasi dei client connessi. Se il numero di richieste ricevute in contemporanea dai client 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; infine, se il segnale è di tipo *SIGPIPE*, esso viene semplicemente ignorato. La comunicazione tra il sigThread e il thread main avviene tramite una pipe denominata *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); tale worker 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 denominata *requestPipe*) che la richiesta è stata servita e che è tornato disponibile.

Alcune operazioni (*lockFile*, *unlockFile*, *closeFile*, *removeFile*) 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*.

## 3. Le librerie: API, fileQueue, threadpool, partialIO

La documentazione completa delle quattro librerie, comprendente la descrizione di tutte le funzioni che esse mettono a disposizione, è presente nei rispettivi *header files*: di seguito verranno descritti solo alcuni dettagli o caratteristiche delle implementazioni che si ritengono degni di nota.

La libreria **API** definisce l'interfaccia di comunicazione tra client e server.

Per tenere traccia dei file aperti dal client, utilizza una struttura dati interna denominata **ofT**: altro non è che una *linked list* avente come campo *"data"* (in questo caso chiamato *filename*) il nome del file, e come campo *"next"* il puntatore al prossimo ofT nella lista. Tale struttura cresce dinamicamente fino a un valore massimo definito nel file header (*MAX_OPEN_FILES*, impostato di default uguale a 50).

La funzione *writeFile* termina con errore se l'operazione immediatamente precedente non è stata una *openFile(O_CREATE | O_LOCK)* terminata con successo.

La funzione *lockFile* opera nel seguente modo: prima di tutto verifica se il file è già stato aperto dal client. Se sì, invoca direttamente la funzione ausiliaria *lockFile_aux*, che tenta di acquisire la lock restando eventualmente in attesa se essa è già posseduta da un client diverso; se no, prova ad aprire il file in modalità *locked* (tramite una *openFile(O_LOCK)*): in caso di successo la funzione termina immediatamente, altrimenti viene controllato il codice di errore (*errno*) ricevuto dal server e, se esso corrisponde a un errore di permessi (significante che la lock è attualmente posseduta da un altro client), chiama la *lockFile_aux*, altrimenti termina con errore.

La procedura *setDirectory* permette, a seconda del parametro *'rw'*, di specificare la cartella dove scrivere il file eventualmente espulso dal server in seguito a un capacity miss provocato da una *openFile(O_CREATE)* (ovvero quando si richiede di creare un nuovo file ma il numero massimo di file nello storage è stato già raggiunto), oppure quella dove scrivere il file letto con la *readFile*. Entrambe le funzionalità non sono previste dalla consegna del progetto, ma sono state implementate poiché ritenute potenzialmente utili e coerenti con la specifica.

La libreria **fileQueue** comprende la definizione di varie strutture dati (*fileT*, *nodeT*, *queueT*) utilizzate dal server per gestire la memorizzazione dei file in memoria principale tramite una coda *FIFO*. Tutte le funzioni che operano sulla coda sono *thread-safe* grazie all'utilizzo di una mutex lock interna, ad eccezione ovviamente della *createQueue* e della *destroyQueue*, che devono essere chiamate da un solo thread (in questo caso il main del server). Per le funzioni che operano direttamente sui fileT (*createFileT*, *writeFileT*, *destroyFile*), invece, non viene garantita l'atomicità: per questo motivo esse vengono chiamate da altre funzioni della libreria solo dopo aver ottenuto la lock sulla coda.

La libreria **threadpool** è stata tratta dalla soluzione dell'esercitazione 11 pubblicata sulla pagina *DidaWiki* del corso, e solo lievemente modificata per adattarla alla struttura del server.

La libreria **partialIO**, infine, permette di gestire i casi di scritture/letture parziali tramite due funzioni, *readn* e *writen*, tratte dal testo "*Advanced Programming In the UNIX Environment*", di Richard Stevens e Stephen. A. Rago, 2013, 3rd Edition, Addison-Wesley. La versione originale comprendeva delle operazioni di aritmetica su puntatori di tipo *void**: tale comportamento è previsto da GCC come estensione GNU, funzionante trattando il tipo *void* come avente dimensione uguale a 1 byte. Tuttavia, allo scopo di rendere il progetto interamente conforme POSIX (ed evitare qualsiasi tipo di *warning* a tempo di compilazione), si è deciso di effettuare, esclusivamente nelle suddette operazioni aritmetiche, il casting dei puntatori a *char**: ciò non modifica in alcun modo la semantica del codice, permettendo allo stesso tempo di rispettare le specifiche degli standard.

## 4. Il client

Il **client** è un processo **single-threaded** che comunica con il server esclusivamente tramite l'API descritta precedentemente.

L'utente può richiedere di effettuare una o più operazioni sullo storage tramite i seguenti **comandi**, specificabili come argomenti al momento dell'esecuzione del client (le parentesi quadre indicano porzioni opzionali del comando):

**-h**: stampa il messaggio d'aiuto, comprendente la lista di tutti i comandi disponibili e la loro sintassi, e poi termina immediatamente (i comandi specificati successivamente vengono ignorati);
**-f** *filename*: stabilisce una connessione al socket AF_UNIX "*filename*", tramite una chiamata alla funzione *openConnection* dell'API. I parametri di tale chiamata fanno sì che la connessione venga tentata ogni cento millisecondi, e che fallisca definitivamente se non è riuscita entro due secondi;
**-W** *file1[,file2]*: invia allo storage una lista di file da scrivere, separati da virgole. Internamente, per ogni file ne viene tentata la creazione tramite la funzione *openFile* dell'API (con flag *O_CREATE | O_LOCK*) alla quale, se quest'ultima ha avuto successo, seguono una *writeFile* e una *closeFile*. Se il file è stato creato con successo ma la scrittura è terminata con errore, significa che il contenuto del file supera la dimensione massima (in bytes) dello storage: in questo caso, si effettua una *removeFile* per ripulire il server dal file (vuoto) appena creato;
**-w** *dirname[,n=0]*: invia allo storage '*n*' file da scrivere, contenuti nella cartella "*dirname*" (o nelle sue sottocartelle); se '*n'* è uguale a zero o non è specificato, si richiede la scrittura di tutti i file possibili. L'operazione avviene internamente tramite una chiamata alla funzione "*ftw*" (conforme POSIX), che visita ricorsivamente la cartella specificata e su ogni file trovato invoca una funzione ausiliaria (*cmd_w_aux*), che a sua volta eseguirà il comando *'-W'*;
**-D** *dirname*: specifica la cartella nella quale scrivere i file espulsi dal server in seguito a *capacity misses* causati dai comandi *'-w'* o *'-W'*. Se quest'opzione non è usata come comando immediatamente successivo a una scrittura, viene generato un errore. Se l'opzione non è specificata, i file espulsi dallo storage non vengono memorizzati sul disco. Se "*dirname*" è uguale alla stringa vuota o a '.' (singolo punto), i file vengono scritti nella stessa cartella dell'eseguibile *client*;
**-r** *file1[,file2]*: legge dallo storage una lista di file separati da virgole. Internamente, per prima cosa richiede l'apertura del file che si vuole leggere (tramite una chiamata alla *openFile* dell'API), e poi invoca in successione una *readFile* e una *closeFile*;
**-R** *[n=0]*: legge *'n'* file qualsiasi dallo storage chiamando la funzione *readNFiles* dell'API. Se *'n'* è uguale a zero o non è specificato, vengono letti tutti i file presenti nel server per i quali si hanno i permessi necessari;
**-d** *dirname*: specifica la cartella nella quale scrivere i file letti dal server con l'opzione *'-r'* o *'-R'*. Se quest'opzione non è usata come comando immediatamente successivo a una lettura, viene generato un errore. Se l'opzione non è specificata, i file letti dallo storage non vengono memorizzati sul disco. Se "*dirname*" è uguale alla stringa vuota o a '.' (singolo punto), i file vengono scritti nella stessa cartella dell'eseguibile *client*;
**-t** *time*: imposta il tempo d'attesa (in millisecondi) tra una richiesta e l'altra. Se non specificata, si presume uguale a zero (nessun ritardo);
**-l** *file1[,file2]*: tenta di acquisire la mutua esclusione su una lista di file, separati da virgole, tramite invocazioni della funzione *lockFile* dell'API. Se un file è stato già messo in modalità *"locked"* da un altro client, l'operazione non termina fino a che la lock su di esso non viene rilasciata (o si verifica un evento diverso, come la cancellazione del file);
**-u** *file1[,file2]*: rilascia la mutua esclusione su una lista di file, separati da virgole, tramite invocazioni della funzione *unlockFile* dell'API;
**-c** *file1[,file2]*: tenta di rimuovere dal server una lista di file, separati da virgole. Per far ciò, necessita di ottenere la mutua esclusione, per cui esegue innanzitutto una *lockFile* dell'API (che, come nel caso del comando *'-l'*, può determinare un'attesa fintanto che la lock sul file è già posseduta da un client differente), e poi una *removeFile*;
**-p**: abilita le stampe sullo standard output di tutte le informazioni utili associate a ogni operazione effettuata: tipo di comando, esito (con eventuale codice di errore), file di riferimento, numero di bytes letti/scritti, ecc. Alcune di queste stampe avvengono internamente alle funzioni dell'API: per attivarle (o disattivarle), viene invocata la procedura *printInfo* dell'API.

Tutti i comandi, ad esclusione di '*-h'*, '*-f'* e '*-p'*, possono essere ripetuti più volte; inoltre, essi vengono eseguiti dal client preservandone l'ordine: questo significa che opzioni quali '*-f*' e '*-t*' hanno effetto solamente sulle richieste ad essi successive. L'unica eccezione a questo ordine riguarda il comando '*-p*': esso, infatti, abilita le stampe informative di tutte le opzioni, indipendentemente se specificate prima o dopo. Tale meccanismo è possibile grazie a una struttura dati interna al client, denominata **cmdT**, che funge da lista ordinata per i comandi e gli argomenti ad essi associati.

Al termine dell'esecuzione del client, viene sempre invocata la funzione *closeConnection* dell'API che si occupa di chiudere la connessione con il server.

## 5. Istruzioni d'uso

Per compilare tutte le componenti del programma allo stesso tempo, è sufficiente aprire un terminale nella *root directory* del progetto ed eseguire equivalentemente il comando ***make*** o ***make all***.

È anche possibile la compilazione separata di client e server utilizzando rispettivamente i comandi ***make client*** o ***make server***.

Con i comandi ***make test1***, ***make test2*** e ***make test3*** vengono lanciati i test per il corretto funzionamento del software. In alcune occasioni si è notato che, alla fine dell'esecuzione dei suddetti test, il terminale può sembrare ancora in attesa della chiusura del programma: in realtà, in questi casi si può facilmente verificare che sia il processo server che i processi client sono già terminati correttamente, ed è sufficiente forzare la comparsa del prompt (per esempio ridimensionando la finestra del terminale) per risolvere immediatamente il problema.

Nota: all'interno dei test si tenta di effettuare anche alcune richieste non valide (come la creazione di file già presenti nello storage o la cancellazione di file già rimossi), quindi l'esito "errore" stampato per le suddette richieste rientra nel comportamento atteso.

Se si desidera ripulire la directory del progetto, è possibile eseguire il comando ***make clean***, che rimuove gli eseguibili di client e server, o il comando ***make cleanall*** che rimuove anche tutti gli altri file con estensione *'.o'* e *'.a'*.

I tre script **test1.sh**, **test2.sh** e **test3.sh** presenti nella cartella *script* dovrebbero essere eseguiti solo tramite i comandi del Makefile descritti in precedenza; lo script **statistiche.sh**, invece, può essere eseguito indistintamente sia dalla root directory del progetto che dalla cartella nella quale risiede.

Di default, è supportato l'invio e la ricezione di file aventi una dimensione massima di 10 kB: è possibile aumentare o diminuire liberamente questo limite modificando le *define* di **BUFSIZE** nei file *server.c* e *api.h*.

*Luca Lombardo*

Mat. 546688

16/12/2021