Relazione progetto di Sistemi Operativi, appello estivo, a.a.. 2020-21
Yuri Andriaccio 599329

Come richiesto dal progetto, sono state sviluppate due applicazioni, una per il client e un'altra per il server.
Il server risponde a tutte le richieste inviate tramite l'API sviluppata, ridirigendo le richieste in arrivo a dei thread worker. I thread eseguono la richiesta, inviando il messaggio di ritorno al client, e segnalano al server di aver terminato il proprio lavoro, rimettendosi in attesa di nuove richieste.

MAKEFILE:
Il makefile utilizza delle regole per autogenerare i file da compilare guardando ricorsivamente le directory in cerca di file sorgente. La funzione recorsiva utilizzata e' stata scritta da **larskholte**, su StackOverflow.com, domanda 2483182. Il make presenta vari comandi eseguibili, si possono visualizzare lanciando il comando **make help** che fornisce la lista di comandi e una descrizione di cosa fanno. Da sottolineare il comando **make convert_sh** che rendere eseguibili tutti i file sh del progetto, e il comando **make stats** che esegue lo script statistiche.sh sul file di log autogenerato da test1, test2 e test3 (o di default). Come richiesto i target **test1**, **test2** e **test3** eseguono i vari test richiesti. Sono forniti anche i target **test2lru** e **test2lfu** che utilizzano le politiche di rimpiazzamento lru e lfu.

TEST3:
Lo stresstest **test3** richiede una moltitudine di file che non sono inclusi nell'archivio per semplicita' e spazio (sono circa 64mb) ma sono autogenerati (una sola volta) con uno script **dummy_gen.sh**, che genera 200 file con dati random nella cartella relativa al test3. La dimensione dei file generati garantisce che il test3 venga eseguito secondo gli scopi prefissati, cioe' di eseguire uno stresstest testando tutte le operazioni possibili, e garantiscono che venga eseguito piu' volte anche l'algoritmo di rimpiazzamento. E' possibile rigenerare i file chiamando il target **dummy_clear** e poi rieseguendo il **test3**. Per come e' costruito il test, la maggior parte delle richieste dei client fallisce, mentre una piccola parte viene realmente eseguita. Infatti i client eseguono 5 operazioni casuali scegliendo 5 file casualmente. La numerosita' delle operazioni pero' garantisce che operazioni su file uguali possano accadere, dimostrando quindi stabilita' anche in caso di accessi concorrenti.

FILE DI CONFIGURAZIONE:
Il server utilizza un file di configurazione specificato tramite l'opzione [-c] (o di default il file config.txt nella cartella corrente, se c'e'). La lettura del file utilizza un parser specifico che implementa l'interfaccia indicata nel file config.h, facilmente intercambiabile con altri formati. Il formato corrente e' [**chiave=valore**] senza le parentesi, uno per linea. Le linee vuote sono saltate. E' possibile inserire commenti a singola linea utilizzando un [#] come delimitatore. Il file config.txt contiene tutte le opzioni specificate. Non c'e' un controllo sulle opzioni passate, quelle non riconosciute sono semplicemente saltate.
Le opzioni attualmente accettate sono:
- socket_name - [stringa] nome del file di socket a cui si collega il server. (default: ./socket.sk)
- log_name - [stringa] nome del file di log delle operazioni del server. (default: ./log.txt)
- worker_threads - [intero] numero di worker thread. (def: 4)
- storage_size - [intero] dimensione massima dello storage in bytes. (def: 100Mb = 104857600b)
- max_files - [intero] massimo numero di file memorizzabili. (def: 100)
- max_connections - [intero] massimo numero di connessioni contemporanee. (def: 10)
- algorithm - [stringa] nome dell'algoritmo di cachemiss. Possibili valori: fifo, lru, lfu. (def: fifo)

SERVER (POLL):
Per la gestione dei thread e delle connessioni, il server utilizza una moltitudine di file descriptor gestiti da una chiamata della funzione **poll**. I segnali sono ascoltati solo dal main thread, mentre sono ignorati dai worker thread, inoltre non vengono realmente ascoltati tramite signal handler ma tramite file descriptor, utilizzando la chiamata **signalfd**, funzione Linux-specific, che permette di attendere l'arrivo di un segnale attraverso un file descriptor, e quindi di incorporarlo con gli altri file descriptor nella **poll** generale. La poll gestisce i file descriptor delle connessioni attive e del file socket su cui e' connesso il server. In caso arrivino dati su una connessione a un client, la poll attiva un thread in attesa e gli passa il file descriptor della connessione in attesa di risposta. In caso di dati in arrivo sul file socket, viene invece gestita l'accettazione di nuove connessioni. Il signal file invece viene innescato da segnali esterni di chiusura del programma (SIGINT, SIGQUIT, SIGHUP) o da segnali interni (SIGUSR1) mandati dai thread per comunicare il termine dell'operazione che stava eseguendo.

POLITICHE DI RIMPIAZZAMENTO:
Sono state implementate tre politiche di rimpiazzamento: FIFO, LRU e LFU. La politica e' selezionabile all'avvio del server tramite il file di configurazione. I dati relativi alle politiche sono salvati nelle strutture dati dei file stessi. Gli algoritmi di rimpiazzamento sono implementati in maniera non bloccante, cioe' non bloccano tutti i file ma cercano di espellerne almeno uno senza bloccare gli altri thread. Puo' succedere che non ne espellano nessuno poiche' un altro thread ha gia' espulso o cancellato l'eventuale file scelto dall'algoritmo. In questo caso, si e' liberato spazio nel server nonostante il fallimento, e quindi i comandi che chiedevano piu' spazio possono continuare la loro esecuzione.
La politica FIFO e' implementata comparando il timestamp della creazione del file in memoria principale. La politica LRU e' invece implementa come tramite l'approssimazione second chance. La politica LFU invece espelle in file riferito di meno.

FILE DI LOG e STATISTICHE:
Il file di log e' della forma "**[Timestamp] – Messaggio**". Ogni print sul file di log apre il file in modalita' append e scrive la nuova linea. Non e' bufferizzato cosi' che, in caso di crash, non si perdano informazioni sulle ultime operazioni avvenute (eventualmente anche la causa del crash stesso). I thread utilizzano una macro che stampa le operazioni in un certo formato, sempre utilizzando l'interfaccia definita nel file log.h.
Lo script **statistiche.sh** esegue il parsing del file di log generato dal server. Come primo e unico argomento richiede il nome del file di log. Lo script utilizza comandi di **grep** e **cut** per ottenere le informazioni richieste, ed e' in grado di gestire facilmente anche file di modeste dimensioni (250000+ linee in pochi secondi). Nel caso ci metta troppo, l'invio del segnale **SIGINT** (Ctrl-C su terminale bash), blocca l'esecuzione dello script e stampa i dati che ha gia' raccolto.

STRUTTURA FILE:
I file sono salvati in memoria temporanea, e sono gestiti unicamente dalle operazioni fornite nel relativo header file. Ogni file ha una suo **mutex** che assicura l'**atomicita** di ogni operazione. Le operazioni di lock e unlock sono implementate utilizzando un descrittore che indica all'esecutore di una qualsiasi operazione se ha i permessi di eseguirla o meno. Un file e' in stato di **lock** solo quando e' **aperto** e il client che lo ha bloccato e' **ancora connesso**. Quando un client si disconnette, il file viene sbloccato e chiuso automaticamente (se non e' stato gia' richiesto dal client stesso). Le operazioni sui file possono ritornare vari codici di errore, utilizzando errno. Questi errori non sono fatali nella maggiorparte dei casi. Di conseguenza, il thread che riscontra l'errore non fatale notifica il client e continua la sua esecuzione normalmente.

STRUTTURA MESSAGGIO:
La comunicazione server-client avviene con un protocollo richiesta-risposta. Ogni messaggio e' composto da una testa e da un corpo. La testa contiene informazioni quali tipo, lunghezza del corpo e checksum. Il checksum serve a verificare che il corpo sia stato ricevuto correttamente, ed e' implementato con la funzione hash **djb2** creata da **Daniel J. Bernstein**, leggermente modificata per evitare che vengano eseguite operazioni in overflow (comportamento indefinito secondo lo standard C). Il protocollo richiesta-risposta vuole che il client prepari il messaggio, lo invii e aspetti una risposta dal server.

READN e WRITEN:
Sono state utilizzate per l'invio e la ricezione dei messaggi, in modo non bloccante da parte di segnali, le funzioni **readn** e **writen** implementate da **W. Richard Stevens** e **Stephen A. Rago**, con piccole modifiche per far felice il compilatore.

CLIENT:
Il client parsa da riga di comando le operazioni da eseguire richieste dal progetto, con l'aggiunta dell'operazione [**-a**] che appende un timestamp ai file specificati (l'operazione di append era l'unica non utilizzata dal client). Una volta parsati i comandi, espande il comando [**-w]** che utilizza una directory in tanti comandi [**-W]** che invece inviano singoli file. I comandi aventi piu' di un file sono invece divisi in tanti comandi atomici. Infine sono inserite opportunamente le richieste di apertura, creazione e chiusura dei file. Quindi il client semplicemente invia in sequenza i comandi e termina.

OPZIONE -R:
L'opzione -R lato client utilizza un'estensione GNU poiche' il suo argomento e' opzionale. Al contrario le altre opzioni che hanno argomenti opzionali richiedono in ogni caso un argomento, piu' eventuali altri opzionali, rientrando cosi' nello standard POSIX della funzione **getopt**.

LOCK BLOCCANTE nell'API:
Nell'API richiesto la lock e' bloccante, cioe' non ritorna finche' non si riscontra un errore o non e' effettivamente bloccato il file richiesto. Tuttavia lato server non e' cosi', anzi il server ritorna subito al client che non e' possibile effettuare la lock poiche' il file e' gia' bloccato. L'API dunque, per soddisfare i requisiti, attende un certo tempo e ritenta di bloccare il file sul server. La soluzione scelta garantisce che lato server non ci siano deadlock e che nessun thread aspetti a vuoto che un file si liberi.