

Progetto di Sistemi Operativi e Laboratorio A.A. 2020-2021

Nicola Vetrini – matricola 600199

La repository contenente il codice del progetto è consultabile al seguente indirizzo: [github](#)

Architettura del sistema software

Il filesaver implementato è diviso in tre entità principali: il server, il client e la API di comunicazione tra client e server. La API è implementata come libreria condivisa, linkata sia dal programma server che dal programma client. Inoltre ho realizzato una libreria condivisa di funzioni di utilità (*libutils.so*), tra cui vi è l'implementazione di una coda di nodi (lista concatenata singola).

Il server è organizzato secondo il modello master-worker (con una threadpool di dimensione fissata all'avvio) ed internamente utilizza una coda sincronizzata (un produttore, n consumatori) per lo smistamento delle richieste provenienti dai client ai thread worker. Le richieste sono inserite dal thread manager in coda e servite da uno dei thread worker, che invia un feedback al manager una volta che la richiesta è stata completata (che abbia avuto successo o meno).

Il server mantiene anche una coda sincronizzata dei path dei file inseriti nel filesystem, in modo da rendere agevole l'implementazione della politica di rimpiazzamento FIFO dei file (maggiori dettagli nella sezione successiva).

Per memorizzare i file nel server ho scelto di utilizzare una [hash table](#) la cui chiave di ricerca è il path (assoluto) del file, poiché esso è univoco nel file server. La hash table permette l'accesso in tempo costante al caso medio, perciò è un'ottima struttura dati per ricerca puntuale di dati. Ho preferito non implementare un filesystem ad albero in quanto ciò potrebbe potenzialmente ridurre le prestazioni del server, dato che l'albero delle directory potrebbe crescere in modo non bilanciato.

I valori della hashtable sono dei puntatori a strutture (*struct fs_filedata_t*, definita nel file *server-utils.h*) il cui contenuto è un puntatore ai dati del file, la sua dimensione in byte, i socket dei client che hanno aperto questo file e quale socket (se esiste) ha il lock su questo file.

Politica di rimpiazzamento dei file

La politica di rimpiazzamento dei file è FIFO sia nel caso in cui uno o più file debbano essere rimpiazzati a perché l'inserimento di nuovi dati provocherebbe il superamento della quantità di memoria massima che possono occupare i file all'interno del server che nel caso in cui il numero di file aperti superi quello del massimo numero di file che possono essere memorizzati contemporaneamente nel server. Entrambe le soglie sono lette dal file di configurazione e rimangono costanti per tutta l'esecuzione del server.

I file da espellere sono inviati lungo il socket al client che ne ha provocato l'espulsione. Le funzioni della API che possono provocare espulsioni di file leggono il numero di file espulsi direttamente dalla risposta del server (il cui formato è descritto nel seguito) ed in ragione di ciò leggono, ed eventualmente salvano nella directory fornita, i file espulsi. Tutte le operazioni lato server sono scritte nel file di log: quali file sono espulsi e le loro dimensioni. La funzione che si occupa della lettura e del salvataggio dei file ricevuti lato API è *write_swp*, implementata nel file *fs-api-utils.c*. Tale funzione stampa i path e le dimensioni dei file ricevuti, più eventuali errori, rispettivamente su standard output e standard error.

Formato file di configurazione

È possibile specificare un file di configurazione del server, che deve contenere alcuni o tutti i parametri con cui il server viene eseguito. Tale file va specificato come argomento da riga di comando con l'opzione *-f <conf_path>*. Se il server è lanciato senza specificare *-f* oppure non è possibile aprire e/o leggere il file di configurazione specificato (ad esempio perché non esiste) viene usato il file di configurazione di default "*config.conf*" generato dallo script *makeconf.sh* chiamato con opportuni parametri nel target *all* del Makefile.

Il formato del file configurazione è quello di una serie di linee terminate da '\n' (non sono consentite linee vuote o commenti). Ogni linea contiene due campi, separati da '\t'. Il primo campo contiene una delle seguenti stringhe ed il secondo contiene il valore assegnato.

- **tpool**: un intero > 0 che definisce il numero di thread worker del processo server. Oltre ai thread worker il server ha un thread manager ed un thread che gestisce la terminazione, non conteggiati.

- **maxmem**: un intero > 0 che indica il massimo numero di Mbyte di memoria che possono occupare complessivamente i file nel server in ogni dato istante. Contribuisce alla determinazione della quantità di memoria occupata soltanto la dimensione dei dati del file, non gli altri dati accessori memorizzati o la memoria occupata dalla *struct fs_filedata_t*
- **sock_path**: il path del socket usato per accettare le connessioni dai client. Può essere sia un path relativo che un path assoluto, a patto che sia raggiungibile dalla directory base del progetto
- **log_path**: il path del file di log del server. Per i path relativi vale quanto affermato sopra

I parametri del file di configurazione possono comparire in un ordine qualunque nel file, ma se il valore non viene specificato o è un valore non consentito per il parametro in questione allora il server setta un valore di default noto a tempo di compilazione (quello definito in *server-utils.h*).

Comunicazione tra manager e workers

Tra il thread manager ed i thread worker vi è una coda il cui accesso avviene in mutua esclusione, su cui il server comunica le richieste provenienti dai client (comprehensive del socket sul quale inviare le risposte). Dopo aver inserito la richiesta in coda il server provvede a togliere dal set di file descriptor ascoltati in lettura (tramite select) quello del socket da cui proveniva la richiesta. Quando un worker ha servito una richiesta (che essa abbia avuto successo o meno) scrive il socket servito su una pipe il cui estremo di lettura è ascoltato dal server, che provvede di conseguenza a reinserire il socket tra quelli ascoltati da select. Se la richiesta era di disconnessione dal server allora il worker, se è andata a buon fine, scrive il socket del client cambiato di segno ed il manager, se legge un intero negativo, non reimmette il socket nel set di quelli ascoltati in lettura (infatti non arriverà più alcuna richiesta da parte di quel client. Inoltre, se era stato ricevuto un segnale SIGHUP in precedenza, il manager controlla se il client disconnesso era l'ultimo (sono rimasti connessi 0 clients); in tal caso i thread worker sono tutti terminati e quindi il server termina secondo la procedura descritta nel seguito.

Terminazione del server

Un thread dedicato nel server gestisce la ricezione di segnali, mentre tutti gli altri thread del processo hanno i segnali mascherati (compreso il thread manager) per cui il thread che esegue la funzione *term_thread()* ha il compito di gestire la terminazione. Per fare ciò si serve di una pipe condivisa (dichiarata in *struct fs_ds_t*) il cui estremo di lettura è ascoltato dalla select nel thread manager. Il thread che gestisce la terminazione scrive l'intero FAST_TERM (definito come 1 nell'header del server) se riceve SIGINT o SIGQUIT e SLOW_TERM (2) se riceve SIGHUP. Dopo aver effettuato la scrittura (di 4 bytes, quindi sicuramente atomica anche senza prevedere sincronizzazione esplicita) questo thread termina ed il thread manager, quando andrà a leggere dalla pipe l'intero scritto agirà di conseguenza.

Il server gestisce due tipi di terminazione: la terminazione veloce e la terminazione lenta.

Nella terminazione di tipo veloce (causata dall'invio del segnale SIGINT o SIGQUIT al server) tutte le connessioni del server verso l'esterno sono chiuse: vengono chiusi sia il socket in ascolto di connessioni da nuovi client che i socket dei client che in quel momento erano connessi al server. Il server non può quindi ricevere ulteriori richieste, per cui svuota la coda dalle richieste che non erano ancora state servite dai worker (perché il client che le ha fatte non avrebbe modo di ricevere la risposta) ed inserisce in coda delle nuove richieste dal contenuto diverso da quelle provenienti dalla API. Le richieste di terminazione veloce inserite sono il numero dei thread worker, poiché quando un worker leggerà dalla coda {FAST_TERM,0,0,0} allora terminerà il thread, per cui il thread manager, che eseguirà *pthread_join* su tutti i worker thread deve attendere che ogni worker sia terminato prima di procedere con la procedura di terminazione del server (che comprende la stampa delle statistiche d'uso tramite la funzione *stats* in *server.c*).

Nella terminazione di tipo lento (causata dall'invio al server di SIGHUP) il thread manager non accetta nuove connessioni (quindi quel socket viene chiuso), ma i client connessi possono continuare ad inviare richieste finché non si disconnettono; quando non vi è più alcun client connesso allora il server termina liberando le risorse, in modo simile a come descritto in precedenza, ma usando in questo caso una variabile condivisa (*slow_term*) scritta soltanto dal thread che gestisce la terminazione.

TODO: REVIEW

API

La API è costituita da un insieme di funzioni che soddisfano la specifica data nel testo del progetto ed una struttura dati condivisa che consente alla API di associare al pathname del socket (usato come parametro nelle call della API) al socket di tipo AF_UNIX usato per la comunicazione con il server. Dato che ogni client connesso ha un socket diverso la API mantiene internamente l'associazione biunivoca tra client e socket utilizzando un array di coppie (PID del client, descrittore del socket) aggiornato ogni volta che un client si connette o si disconnette dal server.

Protocollo di comunicazione tra API e server

La API implementa le funzioni descritte dalla specifica (nel file fs-api.c) ed inserisce le apposite richieste sul socket creato per la comunicazione con il server. Dal momento che possono essere connessi più client contemporaneamente la API deve mantenere uno stato interno tramite una variabile globale (struct conn_info dichiarata in fs-api.h) che principalmente memorizza le coppie (PID, socket) per determinare, grazie ad una chiamata a getpid(), quale client abbia richiesto una certa operazione e quindi determinare univocamente il socket sul quale inviare la richiesta (o anche negare a priori l'operazione, se ad esempio il client non si era connesso al server chiamando openConnection).

Le richieste dei client tramite la API possono essere di apertura, di ~~lock (L)~~, di ~~unlock (U)~~, di lettura, di scrittura, di scrittura in coda (append), ~~o di rimozione (X)~~ di un file, il quale è identificato univocamente dal suo path all'interno del server. Vi è un'ulteriore tipo di richiesta, quella di disconnessione (!) che serve essenzialmente al server per rimuovere il client da quelli da cui riceve richieste. Una volta inviata la richiesta al server esso invierà una risposta ed (opzionalmente) dei file alla api, che poi provvederà a gestirli opportunamente.

Le richieste sono delle strutture (struct request t) contenenti i seguenti campi:

- char type: il tipo della richiesta, ovvero $type \in \{ '!', 'O', 'L', 'U', 'R', 'W', 'A', 'X' \}$
- int flags: le flag della richiesta (O_CREATEFILE, ~~O_LOCKFILE~~)
- size_t path_len: la lunghezza del pathname
- size_t buf_len: la lunghezza del buffer che passo al server (in richieste di append)

Le risposte del server hanno un formato simile (struct reply_t):

- char status: l'esito dell'operazione richiesta. I valori possibili sono 'Y' o 'N'
- int nbufs: il numero di file che saranno inviati sul socket in risposta della richiesta
- size_t paths_sz: la lunghezza dei path dei file inviati

Se il server deve inviare dei file allora nbufs è > 0 e le dimensioni di essi sono inviati in una write successiva come array di size_t (quindi il ricevente conosce quanti byte leggere a priori e come interpretarli, dato che saranno nbufs * sizeof(size_t), da memorizzare in un array. I path dei file sono poi inviati nella successiva write, concatenati in una stringa (terminata quindi da '\0'), ma tra loro separati da '\n'. Il valore del campo paths_sz è calcolato in modo da essere uguale al numero di byte della stringa di path generata (compreso il terminatore).

Ad esempio, se il server inviasse i file con path file1, file2, file3 e file4, la api leggerebbe la stringa

"file1\nfile2\nfile3\nfile4\0" che può poi tokenizzare opportunamente.

La generica implementazione di una funzione della api quindi è riassumibile con il seguente pseudocodice:

```
clientConnesso();
writeRichiesta();
readRisposta();
readBufsize();
readPaths();
readFiles();
storeFiles();
```

Naturalmente non tutte le operazioni provocano l'espulsione di file (readFile ad esempio), per cui alcune funzioni della API si limitano a leggere la risposta per vedere se l'operazione è andata a buon fine oppure no e agire di conseguenza. La openFile non provoca alcun rimpiazzamento per numero di file, dato che il server

rifiuta l'operazione (di creazione del file, dato che l'apertura di file già presenti non presenta problemi di questo genere) se sono già presenti nel server il massimo numero di file. Per poter inserire un nuovo file è necessario quindi rimuovere un altro file aperto dal client, oppure provocarne lo swapout attraverso una append opportuna.

Per garantire il particolare modo la correttezza della semantica della `writeFile` il server mantiene anche una struttura dati che associa ad ogni socket client l'ultima operazione completata con successo (il tipo dell'operazione, le flags usate e il path del file sul quale ha operato), così da poter determinare se l'ultima operazione effettuata da quel socket sia stata `openFile(path, O_CREATEFILE)`

Opzione `-w <path> [,n=0]`: La semantica è quella data nel testo del progetto, tuttavia nell'implementazione il parametro del client che contiene l'intero `n` di default è illimitato, ma con valore `-1` per discriminare più facilmente tra il caso illimitato e l'avvenuta lettura di esattamente `n` files di cui effettuare la scrittura. Le directory sono

Script di testing

Nel makefile principale vi sono tre target di test: `test1`, `test2` e `test3`. I file di configurazione per i test sono generati durante l'esecuzione del corrispondente target dallo script `makeconf.sh`, che prende i parametri da scrivere rispettivamente in `test1.conf`, `test2.conf` e `test3.conf`.

Nel `test1` il lancio del server e dei client avviene tramite lo script `test1.sh`, il quale esegue il server con `valgrind` (in background) ed esegue dei client che effettuano delle richieste. Al termine dell'esecuzione dell'ultimo client viene inviato il segnale `SIGHUP` al server (siccome il server è lanciato con `valgrind` il suo PID sarà quello di `valgrind`, per cui `pidof` ha argomento `valgrind.bin`).

Il target `test2` testa invece l'algoritmo di rimpiazzamento dei file, per cui crea, se non esiste, la directory `save_writes` nella directory base del progetto e tenta di scrivere tutti i file nella directory `testcases`, passando al client `-D save_writes`. L'algoritmo di rimpiazzamento dovrebbe risultare al termine dell'esecuzione del server eseguito 6 volte ed i file espulsi (oltre alle stampe effettuate da `write_swap`) sono registrati anche nel file di log del server (`test2.log` in questo caso)

Altre librerie o codice terzo utilizzato

Per la hashtable ho usato quella fornita nell'esercitazione 7 (`icl_hash`) e ne ho ricavato una libreria condivisa (didawiki.cli.di.unipi.it/lib/exe/fetch.php/informatica/sol/laboratorio21/esercitazioniib/icl_hash.tgz) e per la lettura/scrittura di esattamente `n` byte su un file descriptor ho usato le funzioni fornite durante il corso (<http://didawiki.cli.di.unipi.it/doku.php/informatica/sol/laboratorio21/esercitazioniib/readnwritten>). I file usati per effettuare il testing, contenuti nella directory `testcases`, sono stati scaricati dalla seguente pagina web ([Canterbury Corpus](http://www.canterburycorpus.org/)). Gli altri file presenti in `testcases` (nelle sottodirectory) sono stati scritti da me.