

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 del 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 di 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*).

Formato file di log

Il server produce un file di log, il cui path può essere specificato tra i parametri del file di configurazione e che viene troncato se già esistente nel filesystem (cioè al termine di due esecuzioni successive del server con stesso file di log sono presenti solamente le operazioni relative all'ultima esecuzione). Una riga del file di log ha il seguente formato:

```
[<data ed ora>] [CLIENT <pid>] <messaggio>: <esito operazione>\n
```

Data e ora sono quelle ottenute chiamando *ctime_r()* utilizzando come parametro il timestamp (*time_t*) corrente. In realtà il formato della riga nel caso di espulsioni di file dal server viene stampata soltanto la stringa “[SERVER] Capacity miss: espulsi i seguenti file” e sulle righe successive stampa i path dei file espulsi con le relative dimensioni in byte. Il messaggio è una stringa che indica la funzione eseguita dal worker in seguito alla richiesta pervenuta (*api_openFile*, *api_readFile*, ...) con il path del file e le flags o la dimensione del buffer scritto (se pertinenti). L'esito è una stringa il cui contenuto dipende da *errno*: se *errno* == 0 (ovvero si sono verificati errori di uso del fileserver da parte dei client, ma non sono fallite delle system call che settano *errno*) allora è la stringa “OK” in caso di operazione eseguita e “FAILED” altrimenti. Se sono fallite funzioni che settano *errno* allora il messaggio è quello ottenuto con *strerror_r* (ho usato la versione XSI-compliant, ovvero quella con prototipo *int strerror_r(int errnum, char *buf, size_t buflen)*). Se non fosse possibile ottenere la stringa di errore il resto del messaggio di log viene stampato comunque.

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 da 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 thread manager, che provvede di conseguenza a leggerlo 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 reinserisce il socket nel set di quelli ascoltati in lettura. Inoltre, se era già 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 tutti i segnali mascherati (compreso il thread manager) per cui il thread che esegue la funzione *term_thread()* ha il compito di gestire la terminazione del server. Per fare ciò si serve di una pipe condivisa il cui estremo di lettura è ascoltato dalla select nel thread manager. Il thread che gestisce la terminazione scrive l'intero FAST_TERM (definito nell'header *server-utils.h*) se riceve SIGINT o SIGQUIT ed invece scrive SLOW_TERM 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 farà partire la procedura di terminazione.

Nella terminazione di tipo veloce 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 della API. Le richieste sono strutture descritte meglio nel seguito, ma in questo caso il primo campo viene settato come `FAST_TERM` e tutti gli altri a 0. Il thread manager inserisce un numero di queste richieste pari al numero di worker thread creati (la dimensione della threadpool) cosicché ciascun worker, che in seguito alla lettura di una di queste richieste termina immediatamente, possa ricevere una richiesta che lo faccia terminare. Il thread manager, concorrentemente, eseguirà `pthread_join` su tutti i worker thread di cui deve attendere la terminazione. Una volta completato le join il manager procede con la stampa delle statistiche d'uso tramite la funzione `stats()` in `server.c` e poi libera la memoria allocata prima di terminare.

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 analogo a come descritto in precedenza, ma usando in questo caso una variabile condivisa (`slow_term`) per far terminare i worker.

Il server prima di terminare stampa su standard output un sunto delle statistiche di uso ed i parametri di configurazione. Le informazioni stampate sono: path del socket su cui i client si erano connessi e path del file di log generato (letti dal file di configurazione), numero di client connessi al momento della terminazione, massima quantità di memoria occupabile dai file (dal file di configurazione), quantità di memoria occupata al momento della terminazione, quantità massima di memoria occupata durante l'esecuzione del server (tutte le quantità di memoria sono espresse sia in Mbyte che in byte), massimo numero di file che potevano essere presenti contemporaneamente nel server, numero di file contenuti nel server al momento della terminazione, massimo numero di file che sono stati presenti contemporaneamente durante tutta l'esecuzione e, per ultima, la lista dei path dei file presenti nel server al termine dell'esecuzione (con la relativa dimensione in byte, se reperibile). I file sono ordinati dal meno recente (il primo tra quelli presenti ad essere stato creato nel server) al più recente.

API

La API è costituita da un insieme di funzioni che soddisfano la specifica data nel testo del progetto ed una struttura dati condivisa (`struct conn_info`, dichiarata in `fs-api.h`) 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. Essendo una struttura dati condivisa...

Per l'implementazione del salvataggio dei file in seguito a espulsioni di file o letture ho deciso di creare file nella directory specificata con `-d` o `-D` il cui nome è formato soltanto dal nome del file, non da tutto il path. Quindi non ricostruisco la struttura ad albero di directory che i file letti avevano nel filesystem all'interno della directory di salvataggio. La funzione che implementa il salvataggio è `write_swp()`, in `fs-api-utils.c` ed è chiamata internamente dalle funzioni della API.

Protocollo di comunicazione tra API e server

Le funzioni della API interagiscono con il server tramite il socket assegnato a quel client, sul quale scrivono le richieste ed i relativi parametri, attendendo poi la risposta da parte del server prima di inviare successive richieste dallo stesso client.

Il client deve chiamare `openConnection` per aprire una connessione con il server; la API chiama `connect()` a tale scopo, ritentando ogni msec millisecondi in caso di fallimento, fino al raggiungimento di `abstime`. L'implementazione data del client prima di aprire la connessione salva il timestamp corrente (tramite `time(0)`) e setta `abstime` come quell'istante a cui aggiungo 5 (cinque secondi), per cui dopo circa cinque secondi di tentativi di connessione falliti la `openConnection` ritorna -1 al client.

Al momento dell'accettazione della connessione da parte del server la API invia sul socket appena aperto un intero contenente il PID del client chiamante. Tale valore verrà memorizzato dal server in una struttura dati

interna al fine di identificare il client all'interno del server. Sia il PID che il socket di un determinato client rimangono costanti durante tutta la durata della connessione, ma il socket non è utilizzabile per identificare i client in quanto potrebbe essere riassegnato lo stesso descrittore di socket a client connessi in momenti diversi. Potenzialmente anche processi connessi in tempi diversi al server possono avere lo stesso PID, per cui la soluzione adottata non è perfetta, ma è in generale meno probabile che ciò accada. Soluzioni alternative esplorate sono state l'utilizzo di `rand_r`, o in generale funzioni che producessero numeri pseudocasuali, ma le cui sequenze fossero riproducibili a partire dai due dati che il server conosce di ogni richiesta: PID e socket del client. Tuttavia, proprio il requisito della riproducibilità vanifica l'adozione di un identificatore intero pseudocasuale, in quanto se vi fossero client connessi in momenti diversi con stesso PID (e potenzialmente sul medesimo descrittore) il problema si ripresenterebbe. Possono essere trovate soluzioni più raffinate al problema, comprendenti ad esempio il tempo di inizio della connessione come parametro, che richiederebbero un'implementazione leggermente più elaborata per concordare uno stato comune tra API e server da cui partire per derivare gli identificatori ad ogni richiesta, ma nel progetto ho implementato soltanto l'identificazione tramite PID.

Le richieste effettuate dai client tramite la API possono essere di apertura ('O'), di lock ('L'), di unlock ('U'), di lettura ('R'), di lettura di `n` files qualsiasi nel server ('N'), di scrittura ('W'), di scrittura in coda ('A'), o di rimozione ('X') di un file. Un file è identificato univocamente dal suo path (assoluto) all'interno del server, per cui i path forniti all'interno delle richieste ed inviati come file espulsi sono tutti assoluti.

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. A differenza delle altre richieste nella richiesta di disconnessione, mandata dalla funzione della API `closeConnection()`, la API non riceve una risposta dal server.

In tutti gli altri casi il server, una volta ricevuta e servita una richiesta invierà una risposta. Se la funzione della API prevede l'invio di file o si sono verificate espulsioni a causa di capacity misses allora read successive provvederanno a far ricevere alla API i file. Il numero e la dimensione in byte dei buffer inviati è specificata nella risposta (`struct reply_t`), con il formato descritto in seguito.

Le richieste e le risposte sono delle strutture (`struct request_t` e `struct reply_t` rispettivamente) dichiarate in `fs-api.h` e contenenti i seguenti campi:

- **char type:** il tipo della richiesta, ovvero `type ∈ {'!', 'O', 'Q', 'R', 'N', 'A', 'W', 'L', 'U', 'X'}`
- **int pid:** il PID del client che invia la richiesta, usato nel server, ove necessario, per determinare se consentire o meno un'operazione sul file o sulla connessione
- **int flags:** le flag della richiesta (`O_CREATFILE = 0x1 = 1`, `O_LOCKFILE = 0x2 = 2`)
- **size_t path_len:** la lunghezza del path del file sul quale effettuo la richiesta
- **size_t buf_len:** la lunghezza del buffer che passo al server (settato se necessario, altrimenti ignorato)

Le risposte del server hanno un formato simile:

- **char status:** l'esito dell'operazione richiesta. I valori possibili sono 'Y' o 'N'
- **int nbuffers:** il numero di file che saranno inviati sul socket in risposta della richiesta
- **size_t paths_sz:** la lunghezza dei path dei file inviati concatenati. Rilevante solo se `nbuffers > 0`

Se il server deve inviare dei file allora `nbuffers` è `> 0` e le dimensioni di essi sono inviati in una write successiva come array di `size_t` (quindi il ricevente deve conoscere quanti byte leggere a priori e come interpretarli, dato che saranno `nbuffers * sizeof(size_t)`). 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 risposta avrebbe il campo `nbuffers = 4` e la api leggerebbe la stringa `"file1\nfile2\nfile3\nfile4\0"` che può poi tokenizzare opportunamente per ricavarne i singoli path

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

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

Naturalmente non tutte le operazioni hanno la necessità di leggere dei file in risposta, 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 reagire 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 specificato dalla configurazione. Per poter

inserire un nuovo file è necessario quindi rimuovere un altro file, oppure provocare lo swapout di altri 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 client (quindi ricorda anche il PID) 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|O_LOCKFILE)`.

Un'operazione particolare implementata dalla API è la `lockFile()`: tale funzione infatti prova ripetutamente ad acquisire la mutua esclusione sul path dato come parametro inviando ogni 100 millisecondi circa richieste di lock al server. Finché l'operazione non viene consentita dal server la funzione non ritorna al client chiamante, di fatto bloccandone l'esecuzione (i client in questo caso sono single-threaded). L'attesa attiva in questo caso è necessaria poiché devo inviare una richiesta e verificare il contenuto della risposta del server, per cui non è possibile utilizzare variabili di condizione per ridurre l'overhead dato dall'intenso scambio di messaggi tra client e server, anche se, operando in locale, la latenza dovrebbe essere comunque accettabile.

Client

Il client è un programma che riceve da riga di comando le operazioni da effettuare una volta connesso al server, usando le apposite funzioni fornite dalla API. Le opzioni possono essere ripetute più volte e sono eseguite nell'ordine in cui compaiono in `argv`, ad eccezione di `-w` e `-R`, `-D` e `-d`. Le ultime quattro opzioni menzionate sono implementate, ma possono comparire al più una volta e vengono eseguite le chiamate alla API corrispondenti soltanto al termine di tutte le altre richieste (prima le richieste associate a `-w` e poi quelle associate a `-R`). Se è passata `-h` al client tale opzione fa stampare il messaggio di uso (definito in `client.h`) e termina il programma client senza effettuare alcuna operazione eventualmente specificata prima o dopo in `argv`; lo stesso comportamento si ha se non viene specificato con `-f` alcun path del socket su cui il client deve connettersi o se non è stata passata alcuna opzione al programma (`argc == 1`).

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

La hashtable usata per conservare i file è quella fornita nell'esercitazione 7 (`icl_hash`), da cui ho ricavato una libreria condivisa (`./libs/libicl_hash.so`)

(didawiki.cli.di.unipi.it/lib/exe/fetch.php/informatica/sol/laboratorio21/esercitazioniib/icl_hash.tgz).

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>), modificate leggermente per gestire anche `EINTR` senza dover ricorrere a codice aggiuntivo ad ogni `readn/written`. L'implementazione è il `utils.c` ed il prototipo in `utils.h`.

I file usati per effettuare il testing, contenuti nella directory `testcases`, sono stati scaricati dalla seguente pagina web dell'Università di Canterbury ([Homepage](#), [files](#)) che mette a disposizione una serie di file di vario tipo (testo e binari) per il testing di algoritmi di compressione; non ho trovato da nessuna parte menzione di restrizioni nell'uso dei file scaricabili, tuttavia se ritenuto necessario posso sostituire quei file con altri. Gli altri file presenti in `testcases` (nelle sottodirectory) sono stati scritti da me.