Università degli Studi di Verona
Dipartimento di Informatica

RELAZIONE FINALE

Progetto di Sistemi Operativi

Prof. del Corso: Masini Andrea
Geretti Luca

Studente: Darie Alexandru – VR500116

INDICE

INTRODUZIONE

Data la seguente specifica:

> *"L'obiettivo è realizzare un server che permetta multiple computazioni di impronte SHA-256. Il tempo di calcolo per impronta è proporzionale al numero di byte dell'ingresso (ovvero il file), dipendente dalla piattaforma e dalla implementazione dell'algoritmo.*
>
> *Successivamente va realizzato un client che invii l'informazione di file di input al server e riceva l'impronta risultante appena computata."*

Si potevano scegliere tre implementazioni differenti:

1. Semafori, code di messaggi e memoria condivisa in Linux / macOS
2. Pthread con monitor e FIFO in Linux / macOS
3. Code di messaggi in MentOS orientato ad aspetti di schedulazione

È stata scelta la seconda opzione per l'implementazione.

- **Approfondimento sul SHA-256:**

  SHA (Secure Hash Algorithm) è una famiglia di algoritmi che trasformano una stringa di byte di qualsiasi lunghezza in un'impronta (digest) di lunghezza fissa.

  Nel caso di SHA-256, l'impronta è di 256 bit (32 byte), rappresentata da 64 caratteri esadecimali.
  È usata per rappresentare contenuti complessi in forma compatta, rendendo molto poco probabile che due input diversi producano la stessa impronta. I principali usi sono:

  - indicizzazione tramite hashing;
  - verifica di integrità dei dati (es. controllo di file scaricati confrontando l'hash ufficiale con quello calcolato).

- **Approfondimento sulla scelta d'implementazione:**

  Si è scelto di utilizzare strumenti come Monitor e FIFO**:**

  - I **monitor**, che sono meccanismi di sincronizzazione, permettono di gestire l'accesso concorrente a risorse condivise, garantendone mutua esclusione.
  - Le **FIFO** (First-In, First-Out), sono canali di comunicazione tra processi che utilizzano code per assicurare uno scambio ordinato di dati, senza alcuna ambiguità.

FUNZIONAMENTO DEL CLIENT

Il client ha la funzione di comunicare con il server attraverso l'uso delle FIFO. Leggendo da tastiera il percorso di un file viene creata una struttura Request (definita in *request_handler.h*), è fondamentale per sapere:

- Il percorso del file su cui operare.
- Il tipo di operazione *(ad esempio 0 = CALC_HASH, 1 = QUERY_CACHE)*.
- Il nome della FIFO privata che il client ha creato per ricevere la risposta.

Senza la Request il server non saprebbe quale file processare, quale operazione eseguire e dove mandare il risultato.

Questa struttura viene poi inviata e il server in ascolto su tale FIFO, la riceve e ne estrae i dati. Il client apre la sua FIFO in modalità lettura e riceve la risposta del server – hash SHA-256 del file se l'operazione è andata a buon fine, altrimenti un messaggio d'errore.

FUNZIONAMENTO DEL SERVER

Il server ha il compito di calcolare l'hash SHA-256 dei file richiesti dai client; affinché il codice risulti più chiaro e modulare, le funzionalità del server sono state suddivise in più file, ognuno con un ruolo ben preciso:

- In *main.c* si trova la parte principale dell'applicazione: viene creata la FIFO principale usata dai client per inviare le loro richieste, e dove inoltre viene avviato il thread pool per gestire l'elaborazione parallela.
  Ogni volta che arriva una nuova richiesta, essa viene inserita in una coda di elaborazione per i thread che si occuperanno di processarla.

- In *cache.c* è stata implementata la gestione della cache: essa si occupa di memorizzare i risultati già calcolati per evitare ricalcoli inutili.
  Se più thread chiedono lo stesso file allo stesso momento, la cache permette di sincronizzarli: uno solo calcola l'hash, mentre gli altri aspettano che il risultato sia pronto.

- In *queue.c* si trova la parte relativa alla coda delle richieste: implementa una coda ordinata in base alla dimensione dei file. In questo file vengono racchiuse anche le funzioni per accodare e prelevare richieste, cosi da garantire la sincronizzazione tra i thread.

- In *request_handler.c* sono definite le funzioni del thread pool: ogni thread rimane in attesa di richieste, le prende dalla coda e le elabora.
  In base al tipo di richiesta, tramite *QUERY_CACHE* si controlla se un hash è già disponibile, mentre tramite *CALC_HASH* si recupera dalla cache o si fa il calcolo quando necessario.

IMPLEMENTAZIONE DELLE RICHIESTE

- **Comunicazione tra Server e Client tramite FIFO**

  Per implementare la comunicazione tra server e client e lo scambio di richieste e risposte, sono state utilizzate le FIFO.

  Lato server:

```
int serverFIFO = open(SERVER_FIFO, O_RDONLY);
int serverFIFO_extra = open(SERVER_FIFO, O_WRONLY);

if (serverFIFO == -1)
    errExit(msg: "open server_fifo failed");
if (serverFIFO_extra == -1)
    errExit(msg: "open serverFd_extra failed");

printf("<Server> In ascolto su %s...\n", SERVER_FIFO);
```

  Lato client:

```
if (mkfifo(req.clientFifo, 0660) == -1)
    perror("<Client> FIFO privata già esistente (ok)");

// apro la FIFO del server per inviare la richiesta
int serverFIFO = open(SERVER_FIFO, O_WRONLY);
if (serverFIFO == -1)
    errExit(msg: "open server_fifo failed");

if (write(serverFIFO, &req, nbyte: sizeof(Request)) != sizeof(Request))
    errExit(msg: "write request failed");

close(serverFIFO);

// apro la FIFO del client per ricevere la risposta
int clientFIFO = open(req.clientFifo, O_RDONLY);
if (clientFIFO == -1)
    errExit(msg: "open client_fifo failed");
```

- **Istanziamento di thread distinti per l'elaborazione di richieste multiple in modo concorrente**

  All'avvio del server viene invocata la funzione *init_ThreadPool()* per creare un numero fisso di thread, ciascuno dei quali chiama la funzione *worker_Thread()* che rimane in un ciclo infinito in attesa di ulteriori richieste.

Il main, infatti, legge le richieste in arrivo dalla FIFO per poi inserirle nella coda condivisa. Da quest'ultima i thread prelevano le richieste che vengono gestite tramite la funzione *handle_Request()*; se arrivano più richieste, il lavoro viene suddiviso tra diversi thread che le elaborano in parallelo.

La creazione dei thread avviene nel *request_handler.c*.

- **Schedulazione di richieste pendenti in ordine di dimensione del file**

  Nel momento in cui il client invia una richiesta al server, quest'ultimo la inserisce in una coda d'attesa tramite la funzione *enqueue_by_size()* che, attraverso *stat(),* ottiene la dimensione del file richiesto così da poter inserire la richiesta nella posizione corretta, in ordine di dimensione.

  L'accodamento si trova nel *queue.c*.

- **Introduzione di un limite al numero di thread in esecuzione**

  Si è utilizzato un thread pool di dimensione statica, il cui limite viene definito nella costante *THREAD_POOL_SIZE*, infatti, la funzione *init_ThreadPool()* ha lo scopo di creare un numero *THREAD_POOL_SIZE* di thread.
  Se arrivano più richieste contemporaneamente, non vengono creati thread aggiuntivi ma le richieste rimangono in coda fino a quando uno dei thread diventa disponibile.

- **Caching in memoria delle coppie percorso-hash per ottimizzare richieste ripetute**

  Prima di calcolare l'hash di un file, il thread controlla se quest'ultimo è già presente in hash tramite la funzione *cache_lookup()*. Se già presente, ne restituisce il valore così da evitare la rilettura e la rielaborazione del file.
  La struttura della cache è una lista collegata di coppie *filePath-hash,* protetta da un mutex così da garantirne l'accesso concorrente.

  L'implementazione si trova nel file *cache.c* e *request_handler.c*.

- **Gestione di più richieste simultanee, elaborandone una alla volta e attendendo i risultati nelle restanti**

  Nel caso in cui la funzione *cache_lookup()* prima citata dia esito negativo (cache non presente) il thread inserisce il file nella lista delle richieste pendenti e inizia il calcolo.
  Se nel frattempo un altro thread riceve una richiesta per lo stesso file, invece di calcolare anch'esso l'hash, si mette in attesa e si risveglia quando l'altro thread completa il calcolo del hash.

  La verifica e la gestione delle richieste pendenti si trova in *cache.c*.

- **Implementazione di un thread pool**

  Come spiegato precedentemente, all'avvio del server la funzione *init_ThreadPool()* crea un numero fisso di thread che eseguono *worker_Thread()* processando ciascuna richiesta tramite *handle_Request()*.

- **Implementazione di un meccanismo per l'interrogazione della cache delle richieste già processate**

  Il client ha la possibilità di scegliere se calcolare l'hash oppure interrogare la cache riguardo a uno specifico file, e vedere se tale richiesta è già stata eseguita.
  Infatti, in caso di *QUERY_CACHE*, il thread richiama la funzione *cache_query()* per cercare l'hash in memoria.