SHA-256
Opzione 2: Phtread & FIFO

UniVR - Dipartimento di Informatica

A.A. 2025/26

Mattia Nicolis – VR500356

Indice

1

Introduzione

Il progetto consiste nella realizzazione di un sistema client-server per il calcolo concorrente dell'hash SHA-256.
L'obiettivo principale è massimizzare il throughput tramite un'architettura multithread, garantendo al contempo l'uso efficiente delle risorse e la robustezza del sistema.

2

Architettura e specifiche implementate

Il sistema è strutturato secondo un modello Master-Worker, dove un processo Server centrale gestisce le richieste provenienti da molteplici Client.

2.1 Componenti principali

- **main thread (dispatcher)**: è il cuore del server.
  Si occupa esclusivamente di ascoltare la FIFO pubblica, deserializzare le richieste in arrivo e inserirle nella coda condivisa (`request_queue`).
- **worker threads**: thread pre-allocati (thread pool) che prelevano le richieste dalla coda.
  Ciascun worker è responsabile della verifica in cache, dell'eventuale calcolo CPU-intensive (SHA-256), dell'aggiornamento della cache e dell'invio della risposta sulla FIFO privata del client.

2.2 Specifiche implementate

Le specifiche principali implementate includono:

- ▷ **comunicazione inter-processo (IPC)**: utilizzo di una FIFO pubblica (`/tmp/sha256_server_fifo`) per le richieste e FIFO private univoche per le risposte, prevenendo deadlock comunicativi
- ▷ **gestione della concorrenza (thread pool)**: un main thread agisce da dispatcher, mentre un pool dinamico di worker threads (massimo 3) esegue i calcoli CPU-intensive
- ▷ **scheduling shortest job first (SJF)**: implementazione di una coda prioritaria che ordina le richieste in base alla dimensione del file, minimizzando il tempo di attesa medio
- ▷ **caching (O(1))**: memorizzazione dei risultati calcolati per evitare ricalcoli inutili su file già processati

3

Scelte implementative e dettagli tecnici

In questa sezione si analizzano le soluzioni adottate per soddisfare i requisiti.

3.1 Gestione della concorrenza

Per garantire la sicurezza dei dati (thread safety) evitando complessità eccessive e rischi di deadlock, si è scelto di utilizzare un approccio basato su un mutex globale (`server_mutex`):

- **funzionamento**: questo singolo mutex protegge l'accesso a tutte le risorse condivise critiche: la coda delle richieste (`request_queue`) e la cache dei risultati (`hash_cache`)
- **motivazione**: sebbene teoricamente meno performante di lock granulari, questa scelta semplifica drasticamente la logica di sincronizzazione, garantendo che non si verifichino mai interblocchi tra thread che tentano di accedere a risorse diverse simultaneamente

3.2 Scheduling e priorità (SJF)

Il cuore dell'ottimizzazione risiede nello scheduler. Invece di una classica coda FIFO (First-In First-Out), le richieste vengono inserite nella lista in modo ordinato in base alla dimensione del file (`file_size`):

- **implementazione**: la funzione `enqueue_request` scansiona la lista e inserisce il nuovo nodo nella posizione corretta
- **vantaggio**: questo implementa la politica Shortest Job First (SJF), assicurando che i file piccoli non rimangano bloccati dietro a file di grandi dimensioni, migliorando il tempo di risposta medio del sistema

3.3 Caching dei risultati

Per ridurre il carico sulla CPU, è stata implementata una cache in memoria (Linked List):

- **funzionamento**: Prima di avviare il calcolo SHA-256, ogni worker thread controlla se il percorso del file è già presente nella lista `hash_cache`

- **cache hit**: se il file è stato già elaborato in precedenza, il risultato viene restituito immediatamente, riducendo il tempo di servizio da O(N) (dove N è la dimensione del file) a O(1) (costante)

4

Difficoltà riscontrate

Durante lo sviluppo sono emerse sfide critiche legate alla natura concorrente del progetto.

4.1 Sincronizzazione delle FIFO

Inizialmente, il server completava il calcolo ma il client rimaneva bloccato in attesa indefinita.
Il problema risiedeva nella gestione delle `open()` bloccanti.
È stato necessario sincronizzare l'apertura in lettura del client con l'apertura in scrittura del Server, garantendo che il canale fosse pronto prima dell'invio dei dati.

4.2 Gestione dei puntatori nella coda prioritaria

L'inserimento ordinato (SJF) in una lista linkata condivisa causava race conditions e corruzione della memoria.
L'implementazione ha richiesto una protezione rigorosa tramite mutex durante la scansione lineare (O(N)) per l'inserimento in testa o nel mezzo della lista.

4.3 Deadlock logici in caso di errori

Se un file non esisteva, il thread terminava senza rimuovere il file dalla lista "pending", bloccando per sempre eventuali richieste future per lo stesso nome file.
È stata introdotta una routine di pulizia (`remove_from_pending`) che viene invocata tassativamente anche nei rami di errore (es. file non trovato)

5

Test e risultati

Il sistema è stato collaudato verificando i casi d'uso principali e la gestione degli errori.

5.1 Verifica scheduling SJF

Lanciando in sequenza un file di grandi dimensioni (es. 100MB) seguito immediatamente da un file piccolo (es. 1MB), è stato verificato dai log del server che il file da 1MB viene processato e completato prima di quello più grande, confermando che la coda viene riordinata correttamente in base alla dimensione (`file_size`).

5.2 Verifica del caching

Per verificare l'efficacia del caching, è stato effettuato un test sequenziale:

1. **prima richiesta**: il server calcola l'hash (tempo: proporzionale alla dimensione)
2. **seconda richiesta (stesso file)**: il server rileva il file nella lista `hash_cache` e restituisce il risultato istantaneamente (tempo costante O(1)), senza rileggere il file da disco

5.3 Test di robustezza

È stato simulato l'invio di una richiesta per un file non presente sul server:

- **comando**: `./client file_fantasma.txt`
- **risultato**: il server utilizza la funzione access() per verificare l'esistenza del file prima di tentare l'apertura.
  Invece di terminare in modo anomalo (crash), invia correttamente al client una stringa contenente l'errore (`"ERRORE_FILE_NOT_FOUND"`), garantendo la stabilità del servizio

6

Guida alla compilazione e avvio

6.1 Prerequisiti

Il sistema richiede la presenza delle librerie OpenSSL per il calcolo dell'hash e del tool CMake per la gestione della compilazione:

- sistemi Debian/Ubuntu: sudo apt install libssl-dev cmake
- sistemi macOS: brew install openssl cmake

6.2 Compilazione

Per garantire una compilazione pulita, è stato utilizzato CMake.

Seguire i seguenti passaggi dalla root del progetto:

Bash

```
mkdir build
cd build
cmake ..
make
```

Al termine dell'operazione, verranno generati tre eseguibili nella cartella `build/`: `server`, `client` e `cache_query`.

6.3 Esecuzione

Per un corretto funzionamento, i componenti devono essere avviati in terminali separati seguendo quest'ordine: Seguire i seguenti passaggi dalla root del progetto:

Bash

```
./server
```

Il server inizializzerà la FIFO pubblica e si metterà in ascolto.

In un secondo terminale, lanciare il client indicando un file esistente:

Bash

```
./client percorso/nome_file.txt
```

Per verificare lo stato della memoria interna del server:

Bash

```
./cache_query
```