

SHA-256

Opzione 2: Pthreads & FIFO

UniVR - Dipartimento di Informatica

A.A. 2025/26

Mattia Nicolis – VR500356

Indice

| | | |
|----------|------------------------------------------------------------|----------|
| 1 | Introduzione | 1 |
| 2 | Architettura e specifiche implementate | 2 |
| 2.1 | Componenti principali | 2 |
| 2.2 | Specifiche implementate | 2 |
| 3 | Scelte implementative e dettagli tecnici | 3 |
| 3.1 | Gestione della concorrenza | 3 |
| 3.2 | Scheduling e priorità (SJF) | 3 |
| 3.3 | Caching dei risultati | 4 |
| 4 | Difficoltà riscontrate | 5 |
| 4.1 | Sincronizzazione delle FIFO | 5 |
| 4.2 | Gestione dei puntatori nella coda prioritaria | 5 |
| 4.3 | Gestione delle eccezioni I/O e prevenzione crash | 5 |
| 5 | Test e risultati | 6 |
| 5.1 | Verifica scheduling SJF | 6 |
| 5.2 | Verifica del caching | 6 |
| 5.3 | Test di robustezza | 6 |
| 6 | Guida alla compilazione e avvio | 7 |
| 6.1 | Compilazione | 7 |
| 6.2 | Esecuzione | 7 |

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.

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)**: agisce come componente di smistamento.
Si occupa esclusivamente di ascoltare la FIFO pubblica, deserializzare le richieste in arrivo e inserirle nella coda condivisa (`request_queue`) per il successivo processamento.
- **worker threads**: thread pre-allocati (thread pool) che consumano le richieste dalla coda. Ciascun worker è responsabile della verifica in cache, dell'eventuale calcolo CPU-intensivo (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 4) 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
- ▷ **introspezione e monitoraggio**: implementazione di un meccanismo di query (tramite un eseguibile ausiliario `cache_query`) che permette all'amministratore di ispezionare in tempo reale lo stato della cache interna del server senza interrompere il servizio

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 correttezza dei dati (thread safety) e l'efficienza, è stato adottato il pattern **producer-consumer** con le seguenti caratteristiche:

- **mutua esclusione**: un singolo mutex globale (`server_mutex`) protegge l'accesso concorrente alle strutture dati condivise: la `request_queue` e la `hash_cache`.
Questa scelta semplifica la logica di sincronizzazione prevenendo la **race conditions**.
- **granularità del lock**: il mutex viene acquisito solo per le operazioni rapide di manipolazione delle liste.
Durante il calcolo oneroso dell'hash (SHA-256) e l'invio della risposta (I/O), il lock viene rilasciato. Questo permette un parallelismo reale: mentre un thread calcola, gli altri possono accedere alla coda o alla cache.
- **attesa passiva**: Oltre al mutex, è stata utilizzata una **condition variable** (`pthread_cond_t`). I thread worker rimangono in stato di attesa (sleep) quando la coda è vuota, evitando lo spreco di CPU tipico del *busy waiting*.
Vengono risvegliati dal dispatcher solo all'arrivo di una nuova richiesta.

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 gestite in base alla dimensione del file (`file_size`):

- **implementazione**: l'ordinamento avviene tramite un algoritmo di **insertion sort** sulla lista linkata: la funzione `enqueue_request` scansiona la lista e inserisce il nuovo nodo mantenendo l'ordine crescente
- **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)

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 Gestione delle eccezioni I/O e prevenzione crash

L'apertura diretta di un file inesistente da parte di un worker causava comportamenti indefiniti o terminazioni anomale del thread, lasciando potenzialmente il sistema instabile.

Il problema è stato risolto inserendo un controllo preventivo tramite la syscall `access(F_OK)`. Questo funge da guardia (guard clause), permettendo al server di intercettare l'errore e rispondere al client con un codice "ERRORE_FILE_NOT_FOUND" in modo controllato, senza compromettere il thread pool.

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.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, garantendo la stabilità del servizio.

Guida alla compilazione e avvio

6.1 Compilazione

Per generare i file eseguibili del progetto, seguire i seguenti comandi:

Bash

```
mkdir <nome_cartella>
cd <nome_cartella>
cmake ..
make
```

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

6.2 Esecuzione

Il sistema richiede l'apertura di terminali separati.

Bash - terminale 1

```
./server
```

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

In un secondo terminale, lanciare il client indicando il nome del file di cui si vuole calcolare l'hash:

Bash - terminale 2

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

Per verificare lo stato della memoria interna del server:

Bash - terminale 3

```
./cache_query
```