Progetto di Sistemi Operativi

Opzione 2: Pthread e FIFO

Matteo Drago
Matricola: VR500241

Anno Accademico 2025/2026

Indice

1 Introduzione

Il progetto ha come obiettivo lo sviluppo di un servizio client-server per il calcolo concorrente di impronte digitali SHA-256 di più file. Il sistema è composto da un server, capace di gestire richieste multiple simultaneamente, e da un client che interroga il servizio.

2 Specifiche implementate

Le specifiche implementate sono state:

- **Comunicazione IPC tramite FIFO**: è stato implementato un canale di comunicazione bidirezionale tra il Client e il Server utilizzando Named Pipes (FIFO). Il protocollo permette l'invio del percorso del file dal Client al Server e la ricezione dell'impronta calcolata.
- **Gestione della concorrenza**: il Server è stato progettato per istanziare thread distinti per ogni richiesta che viene ricevuta, permettendo l'elaborazione di più file in modo parallelo.
- **Limitazione dei Thread (Thread pool)**: è stato introdotto un limite massimo al numero di thread in esecuzione simultanea per evitare il sovraccarico del sistema (un massimo di 6 thread).
- **Schedulazione delle richieste**: le richieste in attesa vengono riordinate in base al criterio *Shortest Job First* (SJF), questo per privilegiare i file più piccoli e ridurre il tempo medio di attesa.
- **Caching in memoria e richieste identiche simultanee**: è stata implementata una Linked List per memorizzare le coppie *<persorso, hash>* . In caso di richiesta ripetuta per uno stesso file il server restituisce il valore in cache andando ad evitare calcoli inutili. Nel caso arrivassero più richieste per lo stesso file contemporaneamente, il calcolo viene fatto una sola volta, andando a restituire il risultato dell'unica computazione fatta.
- **Cache**: è possibile interrogare il server per conoscere le richieste già processate.

3 Architettura del progetto

L'architettura è stata progettata per massimizzare il *throughput*, minimizzare la latenza per i file di piccole dimensioni e ottimizzare l'uso delle risorse tramite caching e coaleshing delle richieste. Il server adotta il pattern Producer - Consumer gestito tramite un Thread pool.

3.1 Struttura

Main Thread (producer) : Ascolta sulla FIFO pubblica, gestisce le `REQ_QUERY` e le `REQ_CALC`. Le prime vengono gestite immediatamente (interrogazione alla cache). Per le seconde viene prima verificata la presenza in cache, se il file non è presente viene inserito un nuovo task nella coda condivisa e viene attivato un worker.

Worker Threads (consumers) : All'avvio viene creato un numero fissato di thread (`NUM_THREADS = 6`). I thread rimangono in attesa finchè non c'è lavoro andando ad eliminare il consumo di CPU a vuoto. Successivamente ogni worker preleva un task, legge il file dal disco, calcola l'hash SHA-256 e aggiorna la cache.

3.2 Protocollo di comunicazione

Il protocollo segue uno schema richiesta-risposta sincrono:

1. **Setup**: il Client crea la propria FIFO privata usando il proprio PID come identificatore univoco
2. **Request**: il Client apre la FIFO pubblica in scrittura (`O_WRONLY`) e invia un binario strutturato (`request_t`) contenente:
   - il PID del mittente (per la risposta)
   - il tipo di richiesta (Hash o Query Cache)
   - il percorso del file. Successivamente la FIFO pubblica viene chiusa.
3. **Wait**: il Client apre la propria FIFO privata in lettura (`O_RDONLY`). Le pipe sono bloccanti e il Client entra in un "blocked state" finchè il Server non scrive qualcosa.
4. **Processing**: il Server riceve la richiesta, la elabora (con Cache o Worker) e apre la FIFO privata del Client specifico.
5. **Response**: Il server scrive il risultato nella FIFO privata e chiude la connessione.
6. **Teardown**: il client si sveglia, legge i dati, chiude la FIFO privata e la cancella dal filesystem.

4 Dettagli implementativi

4.1 Granularità del Locking

Non è stato utilizzato un unico mutex globale, bensì sono stati utilizzati:

- `queue_mutex`: protegge solo l'inserimento o il prelievo dei task
- `cache_mutex`: protegge la lettura o scrittura dei risultati

Questo è fondamentale in quanto permette al Main Thread di accettare e accodare nuove richieste nello stesso istante in cui un Worker sta scrivendo un risultato in cache permettendo di aumentare il parallelismo.

4.2 Prevenzione dei DeadLock

L'architettura evita i DeadLock strutturalmente imponendo un ordine di acquisizione rigoroso (Lock Ordering). Non si verifica mai la condizione in cui un thread possiede il secondo mutex e attende il primo, rendendo impossibile l'attesa circolare tra il Main e il Worker.

4.3 Gestione delle risorse

4.3.1 Tempo di calcolo

Per prevenire il fenomeno del thrashing è stato fissato un numero massimo di thread concorrenti(`NUM_THREADS 6`). In questo modo anzichè creare un numero inifnito di thread, che causerebbe un eccessivo context switching, il sistema accoda le richieste in eccesso. L'uso poi dell'algoritmo **SJF** porta alla gestione della coda, privilegiando i file più piccoli per ridurre il tempo medio di risposta.

4.3.2 Throughput

Per evitare un accesso lento alle risorse, prima vi avviare una lettura il server controlla se il file è già in fase di elaborazione, se cosi fosse non viene avviato un nuovo thread, bensì il client viene agganciato al worker esistente. Questo permette di trasformare la velocitità di elaborazione da *O(N)* a *O(1)*.

4.3.3 Uso della memoria e sincronizzazione

Per la memorizzazione dei risultati (**caching**) è stata utilizzata una lista concatenata che memorizza i path e gli hash calcolati precedentemente permettendo di non dover spendere CPU e I/O per i file già processati.

L'accesso alla coda dei task e alla lista della cache è regolata tramite Mutex (`pthread_mutex_t`), che garantiscono mutua esclusione prevenendo la *race conditions*.

5 Verifica e Test

La verifica del sistema è stata fatta manualemente testando diversi casi limite.

5.1 Verifica capacità limite e scheduling SJF

Per verificare l'algoritmo **SJF** e la gestione delle priorità sono state inviate 6 richieste simultanee per file di grandi dimensioni (circa 130 MB). Queste richieste hanno occupato istantaneamente tutti i worker disponibili. Sono poi state inviate 2 ulteriori richieste:

- un file medio (110 MB)
- un file piccolo (101 MB)

Nonostante la richiesta del file medio fosse arrivata prima di quello piccolo, i log hanno mostrato che non appena un thread si è liberato, il sistema ha eseguito il file piccolo

```
[SCHEDULER] Accodato file block_6.bin (Size: 136314880 bytes)
[WORKER] Elaborazione file: block_6.bin
[MAIN] Richiesta calcolo: test_110.bin (PID 8750)
[SCHEDULER] Accodato file test_110.bin (Size: 115343360 bytes)
[MAIN] Richiesta calcolo: test_101.bin (PID 8757)
[SCHEDULER] Accodato file test_101.bin (Size: 105906176 bytes)
[CACHE] Risposto a PID 8733 (Cache Hit/Coalescing)
[WORKER] Elaborazione file: test_101.bin
[CACHE] Risposto a PID 8736 (Cache Hit/Coalescing)
[WORKER] Elaborazione file: test_110.bin
[CACHE] Risposto a PID 8734 (Cache Hit/Coalescing)
```

Figura 1: meccanismo di accodamento e schedulazione.

5.2 Verifica del caching (Cache Hit)

È stata verificata l'efficacia del caching in memoria rieseguendo una richiesta per un file di grandi dimensioni già elaborato. Il tempo di risposta è stato misurato tramite l'utility time ed è passato da circa 2-3 secondi per la prima esecuzione a pochi millisecondi per la seconda.

```
drachoo@drachoo-VirtualBox:~/progettoSistemiOperativi$ time ./build/client block_1.bin
time ./build/client block_1.bin
2d665b3d82bc27e12ad1e3f47ddaf8fa5b6e81b6fd4816f11f6e84857444f1b1

real    0m2,761s
user    0m0,003s
sys     0m0,002s
2d665b3d82bc27e12ad1e3f47ddaf8fa5b6e81b6fd4816f11f6e84857444f1b1

real    0m0,003s
user    0m0,000s
sys     0m0,002s
```

Figura 2: caching in memoria.

5.3 Verifica del Coalescing (richieste simultanee)

Per validare il requisito di gestione di richieste multiple simultanee sono stati lanciati due client concorrenti che richiedevano lo stesso file non ancora presente in cache. Il server ha istanziato un singolo task di calcolo (un solo worker era attivo), mettendo il secondo client in attesa e servendo entrambi al termina dell'unica operazione I/O.

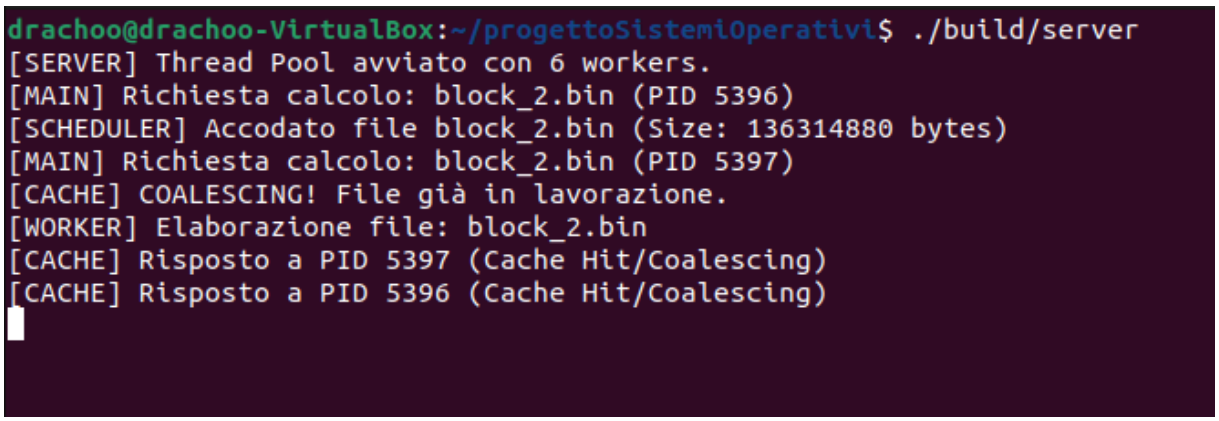

Figura 3: gestione coalescing.

5.4 Verifica della Query

È stato infine implementato il flag `-q` che permette di interrogare lo stato del server mostrando i file memorizzati nella cache e i relativi hash calcolati.

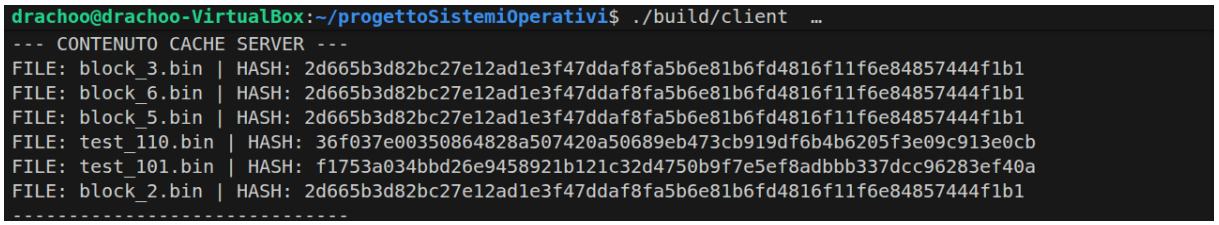

Figura 4: interrogazione al server.

6 Difficoltà affrontate

6.1 Gestione canale di risposta

A differenza della richiesta, la risposta viaggia su un canale dedicato per evitare *race conditions* in lettura tra client diversi. È stato necessario sincronizzare l'apertura della FIFO privata lato server (che deve avvenire solo dopo che il client l'ha creata) e gestire correttamente la scrittura bloccante, assicurando che ogni PID ricevesse esclusivamente il proprio hash calcolato.

6.2 Sincronizzazione delle richieste duplicate

L'architettura del sistema gestisce le richieste duplicate accodando i PID in una lista dinamica protetta da Mutex. La difficoltà principale è stata quella di garantire l'integrità di tale struttura condivisa tra il Main Thread (che accoda le richieste) e i Worker Threads (che la evadono), organizzando lo sblocco dei client tramite IPC in modo perfettamente sincrono con la disponibilità finale dell'hash in cache.