Relazione di sistemi operativi

Edoardo Poltronieri - VR501425

Progetto di sistemi operativi sulla codifica SHA256

Indice

1 Obiettivi del progetto **3**

2 Organizzazione progetto **3**

3 Fase di sviluppo **4**
3.1 Comunicazione tra client e server 4
3.2 Memoria condivisa 4
3.3 Semafori System V 5
3.4 Architettura Multi-Processo 5
3.5 Pool di Worker 5
3.6 Schedulazione SJF 6
3.7 Algoritmi multipli di schedulazione 6

4 Fase di testing **7**
4.1 Server 7
4.2 Client 7
4.3 Ulteriori test 8

5 Note finali **8**

1 Obiettivi del progetto

Il progetto deve prevedere la creazione di un **Server** e di un **Client** che comunicano tramite *code di messaggi*, mentre i file devono essere trasferiti attraverso una *memoria condivisa* protetta da *semafori System V*. Il server deve ricevere le richieste dei client, deve calcolare l'**hash SHA256** dei file ricevuti e deve convertirli in una stringa esadecimale di 64 caratteri. Il client deve inviare i file e attendere la risposta del server contenente l'hash calcolato.

Il server deve essere in grado di gestire **più richieste in parallelo** utilizzando processi *worker*, limitando il numero di processi attivi e sincronizzando il flusso tramite semafori per evitare conflitti nella memoria condivisa. Un secondo client speciale deve consentire di modificare dinamicamente il numero di worker attivi durante l'esecuzione.

Per garantire **robustezza e affidabilità**, il server deve gestire eventuali errori durante la comunicazione, come file troppo grandi o malfunzionamenti della memoria condivisa, e deve occuparsi di prevenire la creazione di processi zombie. Le richieste devono poter essere schedulate in base alla dimensione dei file, assicurando un'elaborazione ordinata ed efficiente.

2 Organizzazione progetto

Per cominciare, ho deciso di strutturare il progetto in modo chiaro e ordinato, in modo da facilitare lo sviluppo e la manutenzione del codice. Ho creato una cartella generale chiamata **SHA-hashing**, al cui interno ho organizzato le varie componenti del progetto seguendo una logica modulare. La cartella **src** contiene tutti i file sorgente in linguaggio C, suddivisi tra quelli relativi al **server** e al **client**, così da separare chiaramente le responsabilità delle diverse parti del programma. Al suo interno è presente anche il file **protocol.h**, che definisce le strutture dati e i messaggi utilizzati per la comunicazione tra client e server. La cartella **bin** è stata creata per ospitare gli eseguibili generati dalla compilazione, permettendo di mantenere pulita la directory principale e di evitare confusione con i file sorgente. Nella cartella **testfiles** ho inserito una serie di file di esempio da utilizzare per testare il corretto funzionamento del trasferimento e del calcolo dell'hash SHA256. Per permettere la gestione dinamica del numero di processi worker del server, è stato implementato un secondo client, **client_modify_worker**, in grado di inviare richieste speciali per modificare il pool di worker attivi senza interrompere il server. Infine, nella radice del progetto sono presenti i file **README.md**, contenente una breve descrizione del progetto e le istruzioni per la compilazione e l'esecuzione, e **Makefile**, pensato per automatizzare la compilazione dei vari componenti, semplificando il processo e riducendo il rischio di errori manuali. Questa organizzazione permette di distinguere chiaramente le diverse responsabilità, facilita l'espansione futura del progetto e garantisce una maggiore leggibilità e manutenibilità del codice.

3 Fase di sviluppo

Lo sviluppo del progetto è avvenuto seguendo un approccio **graduale e step by step**, in modo da verificare il corretto funzionamento di ciascuna componente prima di passare alla successiva.

Per prima cosa ho configurato **WSL su Windows**, così da poter lavorare in un ambiente Linux completo all'interno del mio sistema operativo principale. Successivamente ho creato la cartella principale del progetto, chiamata **SHA-hashing**, e al suo interno ho predisposto una struttura ordinata con le cartelle **src**, **bin**, **testfiles**, oltre ai file di supporto **Makefile** e **README.md**. Questa organizzazione iniziale ha permesso di avere un progetto chiaro e facilmente gestibile.

La prima scelta progettuale riguardava le tecnologie da utilizzare: ho deciso di implementare la comunicazione tra client e server tramite **code di messaggi**, il trasferimento dei file tramite **memoria condivisa** e la sincronizzazione tramite **semafori System V** (Specifiche opzione 1, IPC su Linux).

3.1 Comunicazione tra client e server

Specifiche 1 - 2

Usando code di messaggi, realizzare un server che attende le richieste ed invia le risposte ed un client che formula richieste ed attende risposte.

Il passo iniziale è stato creare i due file principali: **server.c** e **client.c**. Il server rimane sempre attivo, riceve le richieste dei client e restituisce l'hash calcolato, mentre il client si occupa di inviare i file e attendere la risposta. Per partire in modo controllato, ho inizialmente implementato solo la comunicazione base tra i due, per capire bene il flusso dei messaggi e standardizzare la loro struttura nel file **protocol.h**. Il client invia una richiesta che contiene il proprio **PID** (numero univoco che viene assegnato al processo) e le informazioni sul file. Il server, dopo l'elaborazione, invia una risposta al client utilizzando il suo **PID** come tipo di messaggio, garantendo che ogni client riceva la risposta corretta.

3.2 Memoria condivisa

Specificha 3

Trasferire il file da client a server attraverso memoria condivisa.

Ho introdotto la **memoria condivisa** per il trasferimento dei file, sviluppando l'implementazione in maniera graduale così da poter controllare passo dopo passo la corretta scrittura e lettura dei dati. Questa soluzione permette di superare le limitazioni di dimensione imposte dalle code di messaggi, di gestire in modo più efficiente i file di grandi dimensioni e di migliorare le prestazioni complessive del sistema. In questo schema, il client copia l'intero contenuto del file nella **SHM** (shared memory) e segnala al server l'avvenuta operazione tramite la coda di messaggi.

A questo punto il progetto integra già due tipologie di meccanismi di IPC (Inter-Process Communication): le code di messaggi e la memoria condivisa (SHM), che consentono a processi distinti di comunicare e scambiarsi dati in maniera coordinata.

3.3 Semafori System V

Specifica 7

Utilizzare almeno un semaforo per sincronizzare il flusso di comunicazione e/o processamento.

A questo punto, il server gestiva ancora una sola richiesta alla volta: lanciare più client contemporaneamente poteva provocare conflitti e sovrascrivere i risultati SHA calcolati creando una situazione di **race condition**. (Ho deciso che logicamente aveva piu senso risolvere il problema delle race condition prima di continuare con lo sviluppo).

Per risolvere questo problema, il passo successivo è stato l'introduzione dei **semafori System V**, che permettono di sincronizzare l'accesso alla memoria condivisa e prevenire conflitti tra client concorrenti. Il semaforo agisce come un meccanismo di lock e unlock che obbliga client e server a mettersi in coda per accedere alla memoria condivisa. Un processo (o thread) può scrivere o leggere solo dopo aver acquisito il semaforo, rilasciandolo subito dopo per permettere al successivo di procedere.

3.4 Architettura Multi-Processo

Specifica 4

Istanziare processi distinti per elaborare richieste multiple concorrenti.

Tuttavia, anche con i semafori, l'approccio era ancora limitato: il server gestiva le richieste in sequenza, creando una coda di attesa per le richieste successive, risultando poco efficiente.

Per incrementare l'efficienza ed abilitare l'elaborazione concorrente, ho introdotto un sistema basato su fork(), grazie al quale il server può generare processi figli incaricati del calcolo dell'hash dei file. È stato inoltre previsto un meccanismo di gestione dei **processi zombie**, così che il server principale possa continuare a ricevere richieste senza interruzioni. Questo problema è stato risolto mediante un gestore di segnali che consente al server di "mietere" i processi figli terminati, evitando che rimangano in memoria e occupino inutilmente risorse di sistema.

3.5 Pool di Worker

Specifica 5

Introdurre un limite al numero di processi in esecuzione, modificabile dinamicamente da un secondo client.

Per ottimizzare ulteriormente il sistema ed imporre un limite controllato ai processi in esecuzione, sono passato da un modello a processi fork() a un pool di **thread worker**. Creare un processo per ogni richiesta ha un overhead elevato; i thread, invece, sono più leggeri e condividono lo stesso spazio di memoria del server. Ho stabilito un limite al numero di worker attivi, che agiscono come una risorsa fissa. Ho sviluppato un client dedicato alla gestione del numero di worker, capace di inviare al server un messaggio speciale per modificare dinamicamente i processi in esecuzione. In questo modo il server può adeguare le proprie risorse al carico di lavoro.

Nota riguarda i worker: attualmente non è possibile ridurre il numero di worker in modo sicuro, perché non è stato implementato un meccanismo di terminazione dei cicli infiniti, che continuano a essere eseguiti fino a chiusura del server. Questa limitazione è stata annotata come punto di possibile sviluppo futuro.

3.6 Schedulazione SJF

Specifica 6

Schedulare le richieste pendenti in ordine di dimensione del file

Per ridurre il tempo di attesa medio delle richieste, ho introdotto un algoritmo di **schedulazione SJF** (Shortest Job First). Un thread manager interno raccoglie le richieste e le ordina in una coda in base alla dimensione del file, dando priorità a quelli più piccoli. In questo modo i file che richiedono meno tempo per il calcolo dell'hash vengono processati per primi, con il risultato di migliorare il flusso di lavoro complessivo e diminuire i tempi di attesa.

3.7 Algoritmi multipli di schedulazione

Specifica 8

Offrire multipli algoritmi di schedulazione delle richieste pendenti (p.e. FCFS), configurabile alla partenza del server

Come ultimo obiettivo ho reso la politica di schedulazione configurabile all'avvio del server. L'utente può scegliere tra due algoritmi, **FCFS** (First-Come, First-Served) e **SJF** (Shortest Job First), passando un argomento da riga di comando (es. ./bin/server FCFS). Questa scelta viene gestita da una variabile e una funzione di comparazione, che permette al thread manager di ordinare la coda interna in base alla politica selezionata.

4 Fase di testing

Per verificare il corretto funzionamento del progetto, il processo di testing è stato organizzato in più step sequenziali.

Inizialmente è stata eseguita la compilazione dei file sorgente tramite **make**, mentre la pulizia dei file compilati è stata effettuata con **make clean**.

4.1 Server

Inizialmente, per avviare il server era sufficiente eseguire il comando ./bin/server. Una volta avviato, la console segnalava che il server era attivo e pronto a ricevere richieste da parte dei client, sia per la codifica SHA sia per la modifica del numero di worker, impostati di **default** a 5 ma adattabili in base al carico di lavoro. Successivamente, con l'introduzione di più algoritmi di schedulazione delle richieste pendenti, il comando è stato esteso a ./bin/server FCFS—SJF, in modo da consentire la scelta della **politica di schedulazione iniziale**.

4.2 Client

Successivamente è stato testato il funzionamento del client con un singolo file, ad esempio:

./bin/client testfiles/prova.txt

Per simulare l'elaborazione concorrente da parte del server, sono stati lanciati più client in background inviando diversi file contemporaneamente:

./bin/client testfiles/a.txt &
./bin/client testfiles/b.txt &
./bin/client testfiles/c.txt &
./bin/client testfiles/d.txt &
./bin/client testfiles/e.txt &
./bin/client testfiles/f.txt &
wait

Infine, per verificare la correttezza dell'hash calcolato dal server, è stato utilizzato il comando standard:

sha256sum testfiles/prova.txt

Per poter usare questo comando, è necessario installare sul sistema la libreria dedicata tramite il gestore di pacchetti della distribuzione Linux in uso, ad esempio su Ubuntu è sufficiente eseguire:

sudo apt update && sudo apt install coreutils

Questo permette di confrontare facilmente l'hash calcolato dal server con quello generato da strumenti esterni standard, garantendo la correttezza dei risultati.

4.3 Ulteriori test

Per verificare la scalabilità dinamica e le diverse politiche di schedulazione, sono stati eseguiti due ulteriori test.

Test di **Scalabilità Dinamica**: È stato avviato il server con un numero ridotto di worker e sono stati inviati più file per creare una coda di richieste in attesa. Successivamente, è stato utilizzato il client di gestione con il comando ./bin/client_modify_worker "numero" per aumentare il numero di worker attivi. L'osservazione dell'output del server ha confermato l'attivazione immediata dei nuovi worker, che hanno iniziato a processare le richieste in coda, dimostrando la corretta implementazione della scalabilità dinamica.

Test di **Schedulazione**: Sono stati eseguiti due test separati. Per l'algoritmo FCFS (First-Come, First-Served), il server è stato avviato con il comando ./bin/server FCFS, e i file sono stati elaborati nell'esatto ordine di invio. Per l'algoritmo SJF (Shortest Job First), il server è stato avviato con ./bin/server SJF, e l'analisi dell'output ha confermato che i file con le dimensioni minori sono stati processati per primi, dimostrando la corretta implementazione della logica di schedulazione basata sulla dimensione del file.

5 Note finali

In conclusione il progetto è stato organizzando seguendo delle regole ben precise adottate fin dall'inzio e mantendno un approccio step-by-step che mi ha permesso di raggiungere gli obiettivi richiesti uno per volta.

Durante tutto lo sviluppo ho mantenuto un approccio graduale, testando ogni passaggio con **file di prova** e verificando gli hash calcolati tramite la libreria **OpenSSL** e il comando **sha256sum** su Linux. Il **Makefile** ha permesso una compilazione rapida e ordinata, semplificando il testing. Ho cercato di mettere sempre al centro dello sviluppo il problema da risolvere, comprendendo la motivazione di determinate scelte e affrontandolo in modo da trarre conclusioni concrete sui benefici che tali soluzioni hanno portato alla mia applicazione.

Il sistema è stato progettato in modo robusto: la gestione dei segnali evita processi zombie e garantisce il corretto rilascio delle risorse IPC anche in caso di interruzioni improvvise, e le richieste di file troppo grandi rispetto alla memoria condivisa vengono gestite senza compromettere la stabilità del server.

Sicuramente in futuro sarebbe utile implementare una logica per ridurre il numero di worker in maniera "safe"; momentaneamente è possibile solo aumentarli, ma sarebbe possibile introdurre questa funzionalità senza compromettere la stabilità del server.