Sistema Client-Server per Calcolo SHA-256

Università:	Università degli Studi di Verona
Dipartimento:	Informatica
Corso:	Laboratorio di Sistemi Operativi
Anno Accademico:	2024 – 2025
Nome:	Dona' Noura
Matricola:	VR486309
Data:	Luglio, 2025

Indice

1 Introduzione : Descrizione progetto

Questo progetto riguarda la creazione di un'applicazione distribuita in linguaggio C, composta da un **server** e un **client**, che comunicano tra loro tramite tecniche di **Interprocess Communication (IPC)** per calcolare l'hash SHA-256 di file. Le tecniche di IPC utilizzate includono, **code di messaggi (Message Queues)**, **memoria condivisa (Shared Memory)** e **semafori POSIX nominati**, che servono per scambiare dati e sincronizzare correttamente i processi coinvolti. Viene inoltre descritta l'architettura multithreaded (basata su processi figlio), che permette di gestire più richieste contemporaneamente. Per decidere l'ordine in cui gestire le richieste, sono state implementate due politiche di schedulazione: **FCFS** (First-Come. First-Served), che segue l'ordine di arrivo e, **SJF** (Shortest Job First), che dà priorità alle richieste più brevi. Infine, il programma si chiude liberando tutte le risorse e senza lasciare processi attivi.

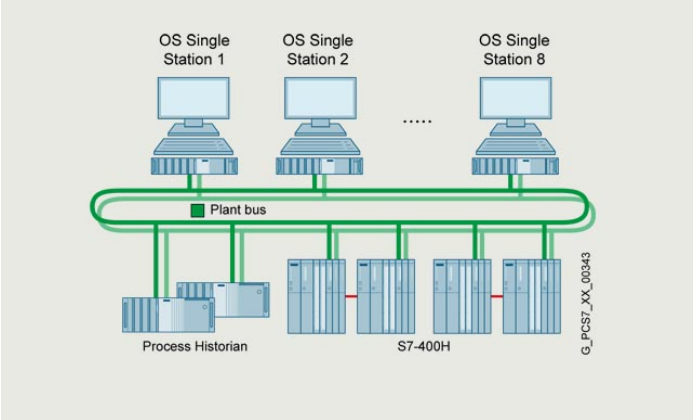

Figura 1: Panoramica dell'Interprocess Communication (IPC) in un'architettura Client-Server.

2 Meccanismi di Interprocess Communication (IPC) Utilizzati

Il funzionamento del sistema client-server si basa su tre metodi principali di comunicazione tra processi indipendenti. Questi metodi permettono ai processi di scambiarsi dati e di interagire in modo coordinato.

2.1 Code di Messaggi (Message Queues)

Le code di messaggi sono il principale canale per la comunicazione tra server e client in modo asincrono. Permettono ai processi di scambiarsi dati anche se non sono attivi nello stesso momento, così mittente e destinatario possono lavorare separatamente.

- **Coda del Server (SERVER_MSG_QUEUE_KEY)**: è il punto dove arrivano tutte le richieste dei client. Il server legge da questa coda per elaborare le richieste.
- **Coda di Risposta del Client (client_pid come chiave)**: per ogni client viene creata una coda personale, identificata tramite il suo PID (Process ID). In questo modo, il server (o i worker) può inviare la risposta direttamente al client corretto.

Il protocollo di comunicazione definisce quattro tipi di messaggi (`mtype`), ognuno con una funzione diversa:

- `MSG_TYPE_REQUEST` (1): Utilizzato dai client per inviare richieste di calcolo dell'hash di un file al server. Contiene informazioni come il PID del client richiedente, il nome del file da elaborare e la sua dimensione.
- `MSG_TYPE_CONTROL` (2): Inviato da un client speciale (client di controllo) al server per modificare parametri operativi in tempo reale, come il limite massimo di processi worker attivi.
- `MSG_TYPE_STATUS_REQ` (3): Utilizzato da un client per chiedere informazioni sullo stato del server, ad esempio quanti worker sono attivi, quante richieste ci sono in attesa e quale politica di schedulazione è attualmente in uso.
- `MSG_TYPE_RESPONSE` (4): È il messaggio che il server (o i worker) manda al client con la risposta. Può contenere l'hash calcolato o un messaggio di errore o stato.

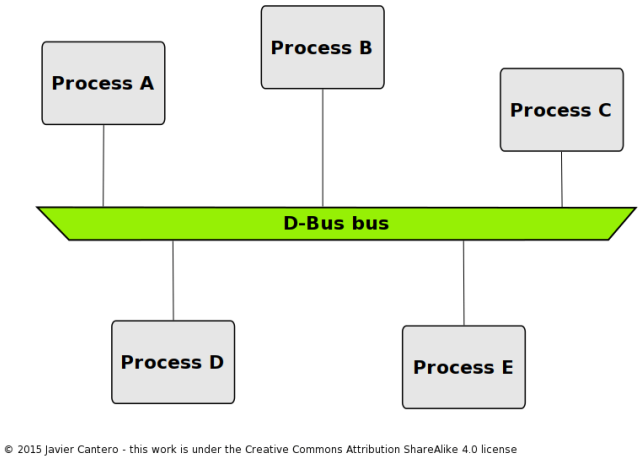

Figura 2: Rappresentazione concettuale di una coda di messaggi per la comunicazione IPC.

2.2 Memoria Condivisa (Shared Memory - SHM)

La memoria condivisa è un metodo IPC molto efficiente per il trasferimento di grandi quantità di dati perché permette a più processi di leggere e scrivere nella stessa area di memoria, evitando di copiare i dati tra i diversi spazi di indirizzamento dei processi.

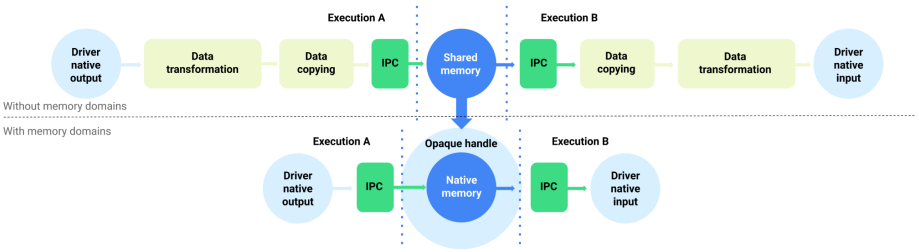

Figura 3: Illustrazione del meccanismo della memoria condivisa tra processi.

- **ID SHM (SHM_KEY)**: Il server crea un'unica area di memoria condivisa, definita dalla struttura `SharedMemoryData`, che viene resa accessibile a tutti i processi coinvolti (server, worker e temporaneamente anche i client quando devono caricare un file). Questa area viene mappata nello spazio di indirizzamento di ciascun processo.
- **S**truttura `SharedMemoryData`: È la struttura principale nella memoria condivisa e contiene tutti i dati necessari per far comunicare e coordinare i processi tra loro:
  - `max_workers`: campo che indica il numero massimo di worker che possono operare contemporaneamente
  - `current_workers`: contatore che indica quanti worker sono attivi in un dato momento.
  - `scheduling_algo`: indica quale politica di schedulazione viene usata (FCFS o SJF).
  - `pending_requests_count`: Un contatore che mostra quante richieste sono in attesa nella coda.
  - `request_queue`: È un buffer circolare, cioè un array speciale, usato per mettere in fila le richieste dei client. Ogni elemento dell'array contiene informazioni sulla richiesta, come il PID del client, il nome del file e la dimensione.
  - `queue_head`, `queue_tail`: servono a gestire l'inizio e la fine della coda circolare
  - `file_data_buffer`: è il buffer dove i client scrivono il contenuto del file da elaborare (fino a 256 KB).
  - `file_data_current_size`, `file_data_client_pid`, `file_data_filename`: contengono le informazioni relative al file inserito nel buffer, per permettere al server di sapere a quale richiesta appartiene

N.B.: Il buffer `file_data_buffer` è fatto per gestire un solo file alla volta. Quando un client invia una richiesta, il contenuto del file viene copiato in questo buffer condiviso. È importante che il worker, appena prende la richiesta dalla coda, copi subito i dati in un suo buffer privato. Così si evita che un altro client sovrascriva il buffer prima che il worker abbia finito di usare i dati

2.3 Semafori POSIX Nominati

I semafori POSIX nominati servono a sincronizzare con precisione il server e i processi worker, controllando l'accesso alle risorse condivise. A differenza di quelli anonimi, questi semafori hanno un nome nel filesystem e restano attivi finché non vengono cancellati con un comando apposito (`sem_unlink`).

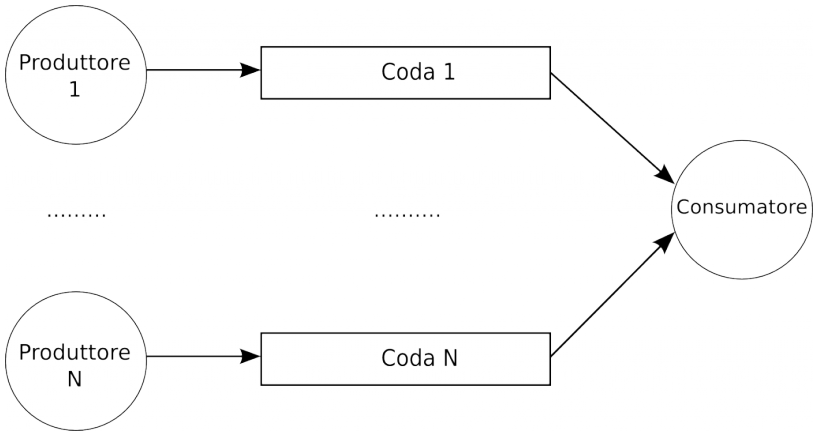

Figura 4: Funzionamento di un semaforo per la sincronizzazione tra processi concorrenti.

- `/worker_limit_sem`: semaforo contatore, inizializzato a `MAX_WORKERS_DEFAULT`, serve a limitare quanti processi worker possono essere attivi contemporaneamente. Prima di creare un nuovo worker, il server chiama `sem_wait()` e, se il limite è stato raggiunto, si blocca in attesa. Quando un worker termina, esegue `sem_post()` per liberare uno slot e permettere al server di crearne un altro.
- `/queue_mutex_sem`: semaforo binario (mutex), inizializzato a 1, che protegge l'accesso alla parte critica della memoria condivisa. In particolare, controlla l'accesso alla coda delle richieste (`request_queue`), i contatori globali come (`current_workers e pending_requests_count`), e i campi informativi del `file_data_buffer`. Serve a garantire che solo un processo per volta possa modificare queste risorse condivise.
- `/queue_fill_sem`: semaforo contatore, inizializzato a 0, che indica quando ci sono nuove richieste nella coda. Ogni volta che un client manda una richiesta, il server esegue `sem_post()` per segnalarla. I processi worker eseguono `sem_wait()` per attendere nuove richieste: se la coda è vuota, si bloccano finché non ne arriva una.

- `/shm_init_sem`: semaforo binario, inizializzato a 1, usato dal server per essere sicuro che la memoria condivisa e gli altri semafori vengano inizializzati solo una volta. Questo è utile quando il server si riavvia, per evitare che si creino doppie inizializzazioni o errori nello stato.

3 Architettura del Server

Il server è la parte principale del sistema: riceve le richieste, le organizza e le assegna ai worker per farle eseguire.

3.1 Inizializzazione del Server (`main`)

Il processo di inizializzazione del server è progettato per garantire uno stato operativo stabile e pulito:

- **Pulizia Iniziale**: All'avvio, il server rimuove eventuali risorse IPC rimaste da esecuzioni precedenti (semafori, coda di messaggi, memoria condivisa). Questo serve a evitare problemi dovuti a chiusure anomale e garantisce che il server inizi sempre in uno stato pulito e controllato.
- **Gestori di Segnali**: Il server usa dei gestori per gestire segnali di sistema che possono cambiare il suo comportamento:
  - `SIGINT` (`Ctrl+C`): Quando arriva questo segnale, `sigint_handler` si occupa di pulire tutte le risorse IPC usate, come semafori, memoria condivisa e code di messaggi, prima di chiudere il server.
  - `SIGCHLD`: Questo segnale arriva quando un processo figlio termina. `sigchld_handler` utilizza `waitpid` con l'opzione `WNOHANG` per recuperare lo stato dei worker terminati, evitando processi zombie e aggiornando il contatore `current_workers` nella memoria condivisa per permettere nuovi worker.
- **Configurazione Schedulazione**: Il server può scegliere l'algoritmo di schedulazione tramite un argomento passato dalla riga di comando (`"sjf"` per Shortest Job First o `"fcfs"` per First-Come, First-Served), permettendo flessibilità nella gestione delle richieste.
- **Creazione/Attacco IPC**: Durante l'avvio, il server crea o si collega alle risorse IPC principali:
  - Viene creata o aperta la coda di messaggi del server, identificata da `server_msqid`, che serve per ricevere le richieste dai client.
  - La memoria condivisa (`shmid`) viene creata o collegata e poi attaccata allo spazio di indirizzamento del server (`shm_ptr`), , così che il server possa leggere e scrivere direttamente in quella memoria condivisa.

- **Inizializzazione SHM e Semafori**: Il semaforo `shm_init_sem` serve a garantire che la struttura `SharedMemoryData` e i semafori nominati vengano inizializzati una sola volta in modo corretto. I semafori `worker_limit_sem`, `queue_mutex_sem` e `queue_fill_sem` vengono aperti o creati con i loro valori iniziali predefiniti.

3.2 Ciclo Principale del Server

Una volta inizializzato, il server entra in un ciclo infinito (`while(1)`) per gestire ininterrottamente le richieste in arrivo:

- **Ricezione Messaggi**: Il server attende e riceve messaggi dalla sua coda principale utilizzando `msgrcv` con `mtype = 0`, il che significa che accetta messaggi di qualsiasi tipo.
- **Gestione** `MSG_TYPE_REQUEST`: Quando riceve una richiesta di calcolo hash da un client:
  1. Riceve la `RequestMessage` dal client.
  2. Prima di accedere ai dati condivisi, il server prende il semaforo `queue_mutex_ptr` per evitare che altri processi li modifichino nello stesso momento.
  3. Inserisce in memoria condivisa (`SharedMemoryData`) le info del file: dimensione, PID del client e nome del file. Questo serve al worker per sapere cosa elaborare.
  4. Aggiunge la richiesta `request_queue` tramite `enqueue_request`. Se la coda è piena, invia un messaggio (`ResponseMessage`) di errore al client.
  5. Rilascia il semaforo `queue_mutex_ptr` e segnala la disponibilità di una nuova richiesta tramite `sem_post(queue_fill_sem_ptr)`.
  6. Esegue `sem_wait(worker_limit_sem_ptr)`per verificare se può creare un nuovo worker. Se il limite è stato raggiunto, attende.
  7. Se c'è spazio, incrementa `current_workers` (protetto dal mutex) e crea un nuovo processo figlio tramite `fork()`.
  8. Il processo figlio avvia `worker_process()`, mentre il processo padre continua il suo ciclo per ricevere nuove richieste.
- **Gestione** `MSG_TYPE_CONTROL`: Quando il server riceve un messaggio di controllo:
  1. Riceve una `ControlMessage` che contiene il nuovo valore del limite massimo di worker.
  2. Prende il semaforo `queue_mutex_ptr` per accedere in modo sicuro alla memoria condivisa e aggiornare il campo `shm_ptr->max_workers`.
  3. Se il nuovo limite è più alto di quello attuale, esegue `sem_post(worker_limit_sem_ptr)` tante volte quante servono per rendere disponibili i nuovi slot per i worker.
  4. Rilascia il semaforo `queue_mutex_ptr` dopo aver finito l'aggiornamento.

- **Gestione** `MSG_TYPE_STATUS_REQ`: Quando il server riceve una richiesta di stato da un client:

  1. Riceve una `StatusRequestMessage` da un client.
  2. Acquisisce `queue_mutex_ptr` per leggere in sicurezza i contatori e l'algoritmo di schedulazione dalla SHM.
  3. Costruisce una stringa informativa sullo stato corrente (numero di richieste in coda, worker attivi rispetto al limite massimo, algoritmo di schedulazione in uso).
  4. Invia questa stringa al client come `ResponseMessage` (`MSG_TYPE_RESPONSE`) , tramite la sua coda di risposta.

3.3 Funzione `enqueue_request`

Viene chiamata dal server per inserire una nuova richiesta nella coda `request_queue`, che si trova nella memoria condivisa. Se l'algoritmo di schedulazione è SJF (Shortest Job First), la funzione cerca di mantenere la coda ordinata in base alla dimensione del file, mettendo i file più piccoli prima. L'ordinamento viene fatto con un insertion sort, che sposta le richieste nella coda finché trova il punto giusto. Per code molto grandi, sarebbe meglio usare una struttura più efficiente, come una coda di priorità (heap).

3.4 Funzione `dequeue_request`

Usata dai worker per leggere la prossima richiesta dalla coda. Viene chiamata solo dopo aver preso il mutex per evitare accessi simultanei.

—

4 Processo Worker (`worker_process`)

Il processo worker è un figlio del server creato per elaborare una singola richiesta di calcolo hash. Termina subito dopo aver completato il lavoro e restituito il risultato.

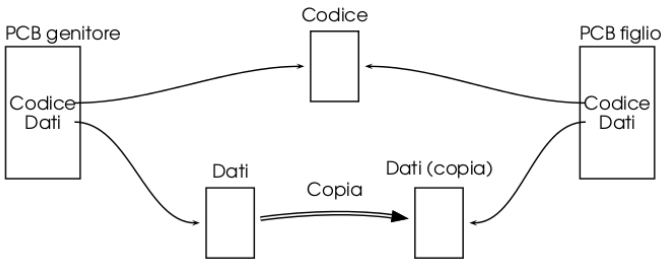

Figura 5: Illustrazione del processo di fork che crea un nuovo processo figlio (worker) dal processo padre (server).

- **Attacco IPC Locale**: All'avvio, ogni worker si collega alla memoria condivisa e apre i semafori nominati già creati dal server. I semafori sono aperti per leggere il loro stato, non per crearli, perché li ha già inizializzati il server padre.
- **Attesa Richiesta**: Il worker esegue `sem_wait(local_queue_fill_sem_ptr)` e resta in attesa finché il server segnala che ci sono richieste disponibili tramite `queue_fill_sem`.
- **Accesso Coda e Copia Dati**: Quando il semaforo `queue_fill_sem` segnala la disponibilità di una richiesta, il worker acquisisce il mutex `local_queue_mutex_ptr` per ottenere l'accesso esclusivo alla coda e prelevare la richiesta tramite `dequeue_request`. Successivamente, copia immediatamente il contenuto del file dal buffer condiviso in un buffer locale, verificando che i metadati (PID, nome file, dimensione) corrispondano. In caso di discrepanze, il worker segnala un errore e termina. Infine, rilascia il mutex per consentire ad altri processi di accedere alla coda.
- **Calcolo Hash**: Il worker calcola l'hash SHA-256 del file copiato nel suo buffer locale, usando le funzioni OpenSSL `digest_buffer` e `bytes_to_hex` per garantire un'elaborazione sicura ed efficiente.
- **Invio Risposta**: Dopo il calcolo, il worker recupera l'ID della coda di messaggi del client (ottenuto dalla richiesta dequeued) e invia l'hash calcolato (o un messaggio di errore in caso di fallimento) come `ResponseMessage` con `MSG_TYPE_RESPONSE`.
- **Pulizia e Terminazione**:
  1. Scollega la memoria condivisa dal proprio spazio di indirizzamento (`shmdt`).
  2. Rilascia il semaforo `worker_limit_sem_ptr` (`sem_post`), segnalando al server che uno slot worker è nuovamente disponibile per nuove creazioni.
  3. Chiude i descrittori dei semafori nominati (`sem_close`) che aveva aperto.
  4. Termina il processo worker (`exit(0)` in caso di successo, `exit(1)` in caso di errore).

5 Funzioni di Utilità SHA-256

Per il calcolo dell'hash SHA-256, il sistema utilizza due funzioni di utilità che includono le operazioni crittografiche:

- `digest_buffer`:è una funzione che utilizza le API `EVP` per calcolare l'hash SHA-256 di un buffer di dati, restituendo il risultato in formato binario.
- `bytes_to_hex`: converte l'output binario dell'hash in una stringa esadecimale, rendendo più semplice la visualizzazione e l'utilizzo del risultato.

—

6 Architettura del Client

Il codice client supporta diverse modalità di funzionamento, scelte tramite argomenti da linea di comando. In questo modo, può inviare richieste di hashing, modificare le impostazioni del server o chiedere lo stato del sistema.

6.1 Modalità Hash Client (`run_hash_client`)

Questa modalità è la principale per il client e serve a inviare al server la richiesta di calcolo dell'hash SHA-256 di un file.

1. **Connessione IPC**: Il client si connette alle risorse IPC già esistenti, ottenendo gli identificatori della coda di messaggi del server e della memoria condivisa.
2. **Creazione Coda di Risposta**: Crea una propria coda messaggi di risposta, usando il proprio PID (`getpid()`) come chiave, per ricevere la risposta specifica dal server o dai worker.
3. **Lettura File e Scrittura in SHM**:
   (a) Il client apre il file in modalità binaria per la lettura del contenuto.
   (b) Controlla che la dimensione non superi il limite massimo previsto.
   (c) Scrive il contenuto nel buffer della memoria condivisa e aggiorna i metadati (dimensione, PID client, nome file).
   (d) **Nota sulla Sincronizzazione**: La scrittura nel buffer della memoria condivisa non è protetta da un mutex lato client. Si presume che il server elabori rapidamente la richiesta per evitare conflitti di accesso. Tuttavia, in ambienti con molti client che scrivono contemporaneamente, potrebbe essere necessario un meccanismo di sincronizzazione aggiuntivo per garantire che la scrittura sia completata prima che il server o un worker leggano i dati.
4. **Invio Richiesta**: Una volta che il file è nel buffer condiviso e i metadati sono aggiornati, il client invia una `RequestMessage` alla coda del server con il proprio PID e la dimensione del file.
5. **Ricezione Risposta**: Il client rimane in attesa di una `ResponseMessage` sulla propria coda, contenente l'hash calcolato o un errore.
6. **Pulizia**: Al termine dell'operazione, il client scollega la memoria condivisa e rimuove la propria coda di messaggi temporanea, garantendo una terminazione pulita.

6.2 Modalità Control Client (`run_control_client`)

Questa modalità consente agli amministratori o ai client autorizzati di modificare il comportamento del server in tempo reale, in particolare il limite massimo di processi worker che possono essere attivi.

1. Recupera l'ID della coda di messaggi del server.
2. Crea una `ControlMessage` contenente il nuovo limite desiderato per i worker e la invia alla coda del server, che aggiornerà la sua configurazione in base a questa richiesta.

6.3 Modalità Status Client (`run_status_client`)

Questa modalità permette di interrogare il server per ottenere informazioni sul suo stato operativo.

1. Recupera l'ID della coda di messaggi del server e crea una coda di risposta temporanea.
2. Invia una `StatusRequestMessage` al server, includendo il proprio PID.
3. Attende e riceve la `ResponseMessage` dalla coda di risposta, contenente una stringa con lo stato corrente del server (es. numero di richieste pendenti, worker attivi e limite, algoritmo di schedulazione).
4. Effettua la pulizia rimuovendo la coda di messaggi temporanea dopo aver mostrato lo stato.

7 Considerazioni sulla Sincronizzazione e Potenziale Miglioramento

Nei sistemi che usano IPC è importante gestire bene la concorrenza per evitare errori di accesso ai dati e mantenere le informazioni corrette.

7.1 Problema `SHM_DATA_MISMATCH`

Il messaggio di errore "Worker PID X: Attenzione: i dati del file in SHM non corrispondono alla richiesta dequeued..." indica una potenziale **race condition** critica che può verificarsi sull'unico buffer `file_data_buffer` nella memoria condivisa.

- **Scenario del Problema**: Un client C1 scrive il file F1 nel buffer condiviso e invia la richiesta. Prima che un worker legga F1, un secondo client C2 scrive il file F2 nello stesso buffer e manda la sua richiesta. Se il worker per C1 legge il buffer dopo la scrittura di C2, trova dati sbagliati, causando errore.
- **Mitigazione Attuale**: Il server aggiorna i metadati (PID, nome file, dimensione) prima di mettere la richiesta in coda. Il worker, appena prende la richiesta, controlla che i metadati corrispondano a quelli nel buffer. Se non combaciano, segnala errore. Questa soluzione funziona solo se il worker legge subito dopo aver prelevato la richiesta.
- **Soluzione**: Per eliminare completamente questa *race condition* e garantire l'integrità dei dati, sarebbero necessari meccanismi di sincronizzazione più granulari o una diversa organizzazione della SHM:

- **Buffer Multipli in SHM**: Allocare più buffer nella memoria condivisa, uno per ogni richiesta. Ogni buffer è protetto da semafori. Il client sceglie un buffer libero per scrivere il file, poi il server mette in coda l'indice di quel buffer. Così più client possono scrivere contemporaneamente senza conflitti.
- **Semaforo per SHM Client-Server**: Introdurre un semaforo binario che protegge il buffer condiviso. Il client lo acquisisce prima di scrivere e lo rilascia dopo. Il server o worker acquisisce lo stesso semaforo prima di leggere. In questo modo si evita che dati vengano sovrascritti mentre sono in uso.

7.2 Algoritmo di Schedulazione SJF Semplificato

L'algoritmo SJF usato in `enqueue_request` ordina le richieste con un metodo semplice che sposta solo elementi vicini, mantenendo la coda approssimativamente ordinata per dimensione del file. Questo funziona bene con poche richieste, ma diventa lento quando sono molte. Per sistemi con tanto carico, è meglio usare una coda di priorità con struttura heap, che inserisce e rimuove richieste in modo più veloce ed efficiente.

7.3 Gestione Errori e Chiusura Semafori

Nel server, i semafori aperti con `sem_open` nella funzione `main` (ad eccezione di `shm_init_sem_ptr` che non viene chiuso lì) non vengono chiusi subito, ma solo nella funzione`cleanup_ipc_resources`, chiamata dal gestore del segnale `SIGINT` (ad esempio, quando l'utente preme `Ctrl+C`). In un sistema di produzione, è importante chiudere sempre tutti i semafori in ogni punto di uscita del programma, sia in casi di terminazione normale che di errori critici. Questo evita perdite di risorse e garantisce una corretta pulizia del sistema operativo.

—

8 Compilazione ed Esecuzione

Per compilare ed eseguire il sistema client-server, sono necessarie le librerie `OpenSSL` (per il calcolo dell'hash SHA-256) e `librt` (per i semafori POSIX nominati).

8.1 Compilazione con Makefile

```
Il progetto include un Makefile per compilare facilmente server e client con le corrette
opzioni per OpenSSL 3.0.15.

Per compilare sia server che client:

make

Per compilare solo il server:
```

```
make server

Per compilare solo il client:

make client

Per pulire gli eseguibili compilati:

make clean
```

8.2 Esecuzione

```
1. Avvio del Server

./server fcfs    # oppure ./server sjf

2. Utilizzo del Client
a) Calcolo Hash

./client hash nome_file

b) Modifica Limite Worker

./client control <numero_worker>

c) Stato Server

./client status
```

Nota Importante: È fondamentale avviare prima il server e solo successivamente i client. Per terminare il server in modo controllato, utilizzare i tasti `Ctrl+C`.

9 Conclusione

Questo progetto mostra come far comunicare diversi processi in un sistema client-server. Usa code di messaggi, memoria condivisa e semafori per far lavorare insieme i processi in modo sicuro e veloce. Vengono anche spiegati alcuni problemi legati al lavoro contemporaneo di più processi e come si potrebbe migliorare il sistema per farlo funzionare meglio. In generale, è una buona base per sviluppi futuri e per sistemi più complessi.