Documentazione progetto Sistemi Operativi

Servizio di calcolo dell'impronta SHA256 su file multipli

Università di Verona

Imbriani Paolo - VR500437

11 luglio 2025

Indice

1 Struttura del progetto

1.1 Specifiche

Il progetto consiste in un servizio che calcola l'impronta SHA256 di uno o più file. L'obiettivo è anzitutto realizzare un server che permette multiple computazioni di impronte SHA256 in parallelo, e un client che permette di inviare i file al server e ricevere le impronte calcolate. Per sviluppare questo progetto vi erano tre alternative:

- IPC (Inter-Process Communication) con semafori, code e memoria condivisa;
- PThread (Parallel Threads) e FIFO
- IPC e Schedulazione con MentOS

Dopo aver valutato le alternative, si è scelto di sviluppare il progetto utilizzando IPC su Linux / MacOS.

1.2 Obiettivi

Gli obiettivi del progetto sono:

- Implementare un server che riceve richieste ed invia risposte, usando code di messaggi
- Implementare un client che invie richieste e riceve risposte, usando code di messaggi
- Trasferire file al server tramite memoria condivisa
- Instanziare processi distinti per elaborare richieste multiple concorrenti
- Introdurre un limite al numero di processi in esecuzione, modificabile tramite un secondo client
- Schedulare le richieste pendenti in base alla loro dimensione
- Utilizzare almeno un semaforo per sincronizzare il flusso di comunicazione e/o processamento
- Offrire multipli algoritmi di schedulazione delle richieste pendenti (p.e. FCFS), configurabile alla partenza del server

1.3 Struttura

Il progetto è composto da due parti principali: il server (`server.c`) e il client (`client.c`). È stata usata una libreria chiamata `common.h` per condividere le strutture dati e costanti tra il server e il client. Inoltre vi è anche un file chiamato `worker.c` che contiene la logica per calcolare l'impronta SHA256 dei file ricevuti ed è l'eseguibile che verrà chiamato dal server per elaborare le richieste.

1.3.1 Client

L'eseguibile del client richiede come argomento il nome del file da inviare al server e l'algoritmo di schedulazione che si vuole utilizzare. Queste sono le prime righe all'interno del file `client.c`:

```
// Prendi algoritmo e file come argomento del client
if (argc < 3) {
    printf("Uso: %s <algorithm> <file>\n", argv[0]);
    printf("Algorithms: 0=FCFS, 1=Priority\n");
    return 1;
}
```

Il workflow del client è il seguente:

- Ottenere il nome del file e la scheduling policy da utilizzare
- Leggere il file e calcolare la sua dimensione
- Creare la memoria condivisa per il file tramite la chiave `SHM_KEY` definita in `common.h`
- Ottenere la coda dei messaggi per le richieste con `msgget(REQ_MSG_KEY)` anch'essa definita in `common.h`
- I dati del file vengono copiati nella memoria condivisa
- Si prepara il messaggio da inviare al server, che contiene:
  - Il PID del client
  - Il nome del file
  - La dimensione del file
  - L'algoritmo di scheduling da utilizzare
- La richiesta viene inviata al server tramite `msgsnd()`
- Il client attende la risposta dal server tramite `msgrcv()`
- Il client riceve la risposta dal server, che contiene:
  - L'impronta SHA256 del file
  - Il tipo di risposta

1.3.2 Server

Il server è l'eseguibile del progetto che gestisce le richieste dei client. Quest'ultimo si avvia e rimane in attesa di richieste dai client. Il workflow del server è il seguente:

- Per essere process-safe Il server manda segnale SIGCHLD al processo padre quando un figlio termina e raccoglie lo stato di terminazione del processo figlio (questo per evitare processi zombie)
- Il server ottiene la coda dei messaggi per le richieste e anche la memoria condivisa.

- Entriamo in un loop, dove il server attende le richieste dai client tramite `msgrcv()`.
- In base al tipo di richiesta ricevuta, il server può:
  - Cambiare il numero massimo di processi in esecuzione
  - Schedulare la richiesta e avviare un processo figlio per calcolare l'impronta SHA256 del file

1.3.3 Worker

Il worker è un processo figlio del server che si occupa di calcolare l'impronta SHA256 del file. Il workflow del worker è il seguente:

- Il worker riceve la richiesta dal server tramite la coda dei messaggi delle richieste
- Il worker legge il file dalla memoria condivisa
- Calcola l'impronta SHA256 del file
- Prepara la risposta da inviare al client, che contiene:
  - L'impronta SHA256 del file
  - Il tipo di risposta
- Il worker invia la risposta al client tramite la coda dei messaggi delle risposte

Il worker è definito nel file `worker.c` e richiede:

- Il PID del client che ha inviato la richiesta
- La chiave della memoria condivisa per leggere il file
- La dimensione del file

```
if (argc < 4) {
    fprintf(stderr, "Usage: %s <shm_key> <data_size>
     <client_pid>\n", argv[0]);
    return 1;
}
```

1.3.4 Libreria condivisa `common.h`

La libreria `common.h` contiene le definizioni delle strutture dati e delle costanti utilizzate come:

```
#define REQ_MSG_KEY 0x5678  // Per richieste client->server
#define RESP_MSG_KEY 0x5679 // Per risposte server->client
#define SEM_KEY 0x1111 // chiave per il semaforo
#define SHM_KEY 0x1234 // chiave per la memoria condivisa
#define MAX_FILE_SIZE 4096 // massima dimensione del file da processare
// numero massimo di file concorrenti che il server può gestire
#define MAX_CONCURRENT 5
```

```c
// Tipo di messaggio controllo per modificare il limite dei file concorrenti
#define CTRL_MTYPE 99
// Tipo di messaggio richiesta per client a server
#define REQ_MTYPE 1
// Tipo di messaggio risposta per server a client
#define RESP_MTYPE 2
// Lunghezza massima del nome del file
#define MAX_FILENAME_LEN 256

// Scheduling policies
#define SCHED_FCFS 0
#define SCHED_PRIORITY 1
```

invece le strutture dati sono:

```c
// Messaggio che invia il client al server
struct msg_request {
    long mtype;
    size_t size;
    pid_t pid;
    int scheduling_policy;
};

// Messaggio che il server invia al client
struct msg_response {
    long mtype;
    char hash[65];
};

// Messaggio che permette di modificare il limite di file concorrenti
struct msg_control {
    long mtype;
    int new_limit;
};
```

2 Implementazione degli obiettivi

2.1 Instanziare processi distinti per elaborare richieste multiple concorrenti

Per gestire richieste multiple concorrenti, il server entra in un loop dove attende le richieste dai client. Quando riceve una richiesta, crea un processo figlio per elaborare la richiesta. Questo è stato fatto tramite la funzione `fork()` all'interno del server. Il processo figlio esegue il worker per calcolare l'impronta SHA256 del file. Il server gestisce i processi figli in modo da evitare processi zombie, utilizzando il segnale `SIGCHLD` per raccogliere lo stato di terminazione dei processi figli. Per eseguire il worker, il server chiama l'eseguibile `worker` passando come argomenti la chiave della memoria condivisa, la dimensione del file e il PID del client che ha inviato la richiesta.

2.2 Introdurre un limite al numero di processi in esecuzione, modificabile tramite un secondo client

Questo obiettivo è stato implementato tramite l'uso di un semaforo per sincronizzare l'accesso alla variabile che tiene traccia del numero di processi in esecuzione. In più è stato creato un nuovo tipo di Request per modificare il limite dei processi in esecuzione. Infatti, se un client vuole modificare questo dato può inviare una richiesta al server con il tipo di messaggio `CTRL_MTYPE`. Il server riceve questa richiesta e modifica il limite dei processi in esecuzione.

2.3 Schedulare le richieste pendenti in base alla loro dimensione

È stato creato un semplice array di richieste pendenti, che viene ordinato in base alla dimensione del file. Questo viene gestito tramite le funzioni `enqueue()` e `dequeue()` successivamente evolute in enqueue e dequeue che vengono "cambiate" in base alla scheduling policy scelta.

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

È stato utilizzato un semaforo per sincronizzare l'accesso alla variabile che tiene traccia del numero di processi in esecuzione. Il semaforo deve garantire che solo un processo alla volta possa accedere a questa variabile, anche perché vitale per lo spawn degli worker.

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

Sono stati implementati due algoritmi di schedulazione:

- FCFS (First-Come, First-Served): le richieste vengono elaborate nell'ordine in cui arrivano.
- Priority: le richieste vengono ordinate in base alla loro dimensione, le più piccole vengono elaborate per prime.

Questo è stato fattibile chiedendo al client di specificare l'algoritmo di schedulazione da utilizzare e poi quando si è all'interno del server, si controlla il tipo di richiesta e si applica l'algoritmo di schedulazione scelto. Questo va a cambiare le nostre funzioni `enqueue()` e `dequeue()` che ora in base alla scheduling policy scelta, ordinano le richieste in modo diverso.

3 Difficoltà riscontrate

3.1 Gestione dei semafori e processi figli

È stata più volte consultata la documentazione per gli IPC con "msgrcv" e "msgsnd" per capire come gestire correttamente i processi e le code di messaggi.

Non era molto chiaro inizialmente come gestire i processi figli e i semafori. Inizialmente si era pensato di utilizzare un semaforo per sincronizzare l'accesso alla coda di messaggi, ma si è scoperto che non era necessario, poiché le code di messaggi sono già sincronizzate. Il semaforo è stato usato per evitare che i processi figli accedessero a delle variabili sensibili come il numero di processi in esecuzione. Inoltre i semafori erano stati usati dove non era necessario, causando confusione e problemi di sincronizzazione. In più non era stato settato il tipo di richiesta a `client_pid`:

```
// Usare il PID del client per indirizzare unicamente la risposta
resp.mtype = client_pid;
```

Questo ha causato problemi nel ricevere le risposte dal server, poiché il client non riusciva a distinguere le risposte destinate a lui da quelle destinate ad altri client.

3.2 Code dei messaggi dedicate

Inizialmente si era pensato di utilizzare una singola coda di messaggi per il client e il server, a prescindere che il tipo di messaggio fosse una richiesta o una risposta. Tuttavia, si è riscontrato che questo approccio non funzionava correttamente, in quanto il client non riusciva a distinguere tra le richieste e le risposte, causando confusione nella gestione dei messaggi. Infatti, il server riusciva correttamente a inviare le risposte al client, ma il client aspettava un tempo indefinito per ricevere le risposte, poiché non riusciva a distinguere tra i messaggi di richiesta e quelli di risposta. Per risolvere questo problema, si è deciso di utilizzare due code di messaggi distinte: una per le richieste e una per le risposte. Questa parte di codice è definita in `common.h`:

```
#define REQ_MSG_KEY 0x5678  // Per richieste client->server
#define RESP_MSG_KEY 0x5679 // Per risposte server->client
```

Di conseguenza il client ora invia le richieste alla coda di messaggi delle richieste e attende le risposte dalla coda di messaggi delle risposte. Questo è come viene implementato in `client.c`:

```
// Coda delle richieste: manda le richieste al server
int req_msqid = msgget(REQ_MSG_KEY, 0666);
if (req_msqid == -1) {
    perror("msgget request queue");
    return 1;
}

// ...

// Coda delle risposte: riceve le risposte dal server
int resp_msqid = msgget(RESP_MSG_KEY, 0666 | IPC_CREAT);
if (resp_msqid == -1) {
    perror("msgget response queue");
    return 1;
}
```

Il server di conseguenza avrà come coda di messaggi solo quella delle richieste, e risponderà tramite il worker con i messaggi alla coda delle risposte del client.