Nome: Michael **Cognome:** De Angelis **Matricola:** 560049 **Corso:** Informatica – B

Anno Accademico.2018 / 2019

Sistemi Operativi e Laboratorio

Progetto "Object Store"

"Università di Pisa – UNIPI"

Relazione del: 01.07.2019

Corso: Sistemi Operativi e Laboratorio [SOL]

Tipologia del lavoro svolto	Luogo di svolgimento delle mansioni	Gruppo di lavoro	Arco di tempo in cui si è svolto il progetto	Tempo impiegato per lo svolgimento delle mansioni
Realizzazione di un progetto assegnato e relativa relazione scritta	Propria abitazione	Singolo	Maggio 2019 – Settembre 2019	Circa due settimane

Sommario

Analisi e Progettazione server side

Nella seguente sezione ci occuperemo dell’analisi del testo proposto e delle scelte effettuate durante la fase di progettazione e realizzazione.

Di seguito viene riportato un frammento del testo del progetto:

> *“Lo studente dovrà realizzare un Object store implementato come sistema Client-Server, e destinato a supportare le richieste di memorizzare e recuperare blocchi di dati da parte di un gran numero di applicazioni. La connessione fra clienti e Object Store avviene attraverso socket su dominio locale. In particolare, sarà necessario implementare la parte server (Object Store) come eseguibile autonomo; una libreria destinata a essere incorporata nei client che si interfacci con l'Object Store usando il protocollo definito; e infine un client di esempio che usi la libreria per testare il funzionamento del sistema.”*

Come punto di partenza, analizziamo la struttura della componente server dell’architettura sviluppata; quest’ultima è infatti un eseguibile multi-threaded che resta in attesa di richieste di connessioni da parte di generici client che fanno uso della libreria “client_op”.
Per ogni nuova connessione ricevuta dal server questo si occupa di generare un thread che si occuperà della gestione delle comunicazioni tra i due partner della comunicazione.
Il server mantiene, durante la sua esecuzione, una struttura dati di tipo hash table in cui sono ripotati, sotto forma di entry < nome client, socket client, client successivo in lista >, tutti i client connessi al server in un dato istante in modo da evitare così connessioni multiple da parte di utenti con lo stesso nome.
Oltre alla struttura dedicata ai client il server mantiene le informazioni riguardanti il suo stato interno comprensive di: numero di client connessi, numero di file memorizzati nel server, dimensione (in bytes) complessiva dei dati memorizzati nel server.
Il server dispone di un ulteriore thread dedicato alla gestione dei segnali che vengono ricevuti. Prima della generazione di tale thread viene effettuata un’operazione di mascheramento di tutti i segnali in modo che questi possano essere “catturati” attraverso la system call *sigwait* e, successivamente, gestiti come se ci si trovasse in un normalissimo segmento di codice C non essendo così limitati dalla tipologia di operazioni che è possibile effettuare per la gestione di un determinato segnale (downside che si avrebbe in una gestione con handler registrati attraverso *sigaction*). Tra tutti i segnali mascherati ne vengono gestiti due in particolare: SIGUSR1 e SIGINT. Alla ricezione del segnale SIGUSR1 il thread gestore dei segnali si occupa di stampare su standard output le informazioni descrittive dello stato del server citate poc’anzi. La ricezione del segnale SIGINT invece fa in modo che inizi la procedura di shutdown del server; il thread gestore dei segnali condivide infatti con il principale flusso d’esecuzione del server (thread listener) una *pipe* utilizzata per implementare tale funzione e notificare al thread di accettazione di nuove richieste di interrompere la registrazione di nuovi client; parallelamente quando arriva il segnale di terminazione i thread che si occupano delle comunicazioni con i client terminano l’operazione che stanno gestendo in quell’istante per poi uscire in modo ordinato (liberano la memoria allocata, eliminano l’entry del client dalla hash table e infine chiudono il corrispondente socket dati).
Nel lasso di tempo in cui i thread del server terminano, il flusso principale resta in attesa per poi procedere all’esecuzione di una funzione di cleanup in cui viene deallocata la hash table ed eliminato il file socket dal workspace.
I thread implementati server-side per la gestione dei client vengono creati joinable in modo che liberino il loro descrittore a seguito dell’esecuzione della funzione *join*; tale scelta implementativa è stata dettata dal fatto di voler scongiurare qualsivoglia problema legato alla gestione autonoma della liberazione della memoria tramite un approccio detached dei thread; se da una parte questa scelta comporta il dover mantenere i descrittori dei thread in una struttura dati, per poi utilizzarli nelle operazioni di join, dall’altra ci consente di prestare “minore” attenzione alla generazione di memory leak del tutto inaspettati.
Il thread gestore dei segnali è anch’esso non detached e le sue risorse vengono liberate tramite la funzione join eseguita dal thread main del server (medesimo approccio dei connection handler thread).

Come è possibile intuire da quanto esplicitato fino ad ora, essendo presenti più thread che agiscono sulle stesse strutture dati condivise del server, è stato necessario introdurre un meccanismo di sincronizzazione attraverso l'utilizzo dei mutex offerti dalla libreria pthread.
La struttura dati hash table implementata fa anch'essa utilizzo di meccanismi di sincronizzazione quali mutex per le attività di inserimento, rimozione, ricerca e distruzione. Per ogni indice della hash table (un puntatore ad una lista linkata) viene creato un mutex così da non dover bloccare l'intera struttura quando vi si agisce (inserimento, rimozione e ricerca), se non durante l'operazione di distruzione (deallocazione), ma solo la lista di trabocco interessata garantendo così il principio di mutua esclusione.

Per permettere al thread principale di gestire sia il descrittore del socket dedicato alla ricezione delle nuove connessioni che il descrittore della pipe per la segnalazione di inizio della procedura di terminazione, si fa utilizzo della system call *select.*

Protocollo di comunicazione

Riportiamo brevemente il formato del protocollo di comunicazione:

"Tutti i messaggi scambiati fra clienti e Object Store hanno un formato basato su una riga di testo (header) codificato in ASCII, terminata da un carattere newline ("\n"); la riga di header può essere in alcuni casi seguita da un numero prefissato di dati binari. La lunghezza di questi dati è espressa, in ASCII, nella riga di header."

Il primo problema che è stato affrontato è stato quello di stabilire con che modalità leggere l'header e i possibili dati in modo preciso ma allo stesso tempo efficiente; si riportano di seguito le soluzioni ideate con le motivazioni per cui sono state scartate e infine la soluzione effettivamente adottata:

- Lettura tramite read / revc di un byte alla volta fino a '\n': altissimo overhead dovuto alle molteplici system call;
- Utilizzo della funzione *ioctl()* per quantificare la quantità di bytes presenti nel socket e fare una lettura precisa: non standard POSIX, implementazione dipendente dall'architettura;
- Lettura di chunk di 1024 bytes alla volta: in caso di dati molto grandi si ha un alto overhead mentre in caso di dati molto contenuti vi è spreco di memoria;
- *recv* attraverso il flag *MSG_PEEK*: soluzione effettivamente adottata: sapendo che l'header (considerato in questa istanza fino a '\n') non può, per costituzione superare la dimensione di 280 bytes (ca.), viene effettuata una lettura tramite il flag MSG_PEEK (che permette di leggere dal socket senza che vengano prelevati dati) in modo da calcolare il numero esatto di bytes da leggere con la successiva recv così da prelevare il solo header fino a '\n' e successivamente processarlo.

Ciononostante, è importante notificare che tutte le letture dal socket, per coerenza di scrittura del codice, sono state effettuate attraverso la system call recv la quale, quando ha il parametro flag impostato a 0, è equivalente alla system call read.
Per quanto concerne invece le scritture è stata usata la system call *send*, anch'essa corrispondente alla system call write quando ha flag impostato a 0.
Per le system call send e recv sono state definite, rispettivamente, le funzioni send_all e recv_all che permettono di scrivere e leggere l'intera quantità di dati specificata, cosa che send/write e recv/read non assicurano.
L'operazione di recupero dell'header fa inoltre uso della system call select per imporre un timeout sull'operazione di lettura; se un determinato client non invia alcun header per un certo periodo di tempo quest'ultimo viene espulso per inattività.
La system call select è stata utilizzata anche nella funzione os_connect (collegamento di un client) per permettere di inserire un timeout alla connessione di un client; per implementare tale feature è stato inoltre necessario rendere momentaneamente non bloccante il socket del client attraverso la system call *fcnt()*.

I comandi di request inviati dal client attraverso la libreria sviluppata seguono il protocollo e non contengono il terminatore di stringa a seguito dello ‘\n’; cionondimeno la funzione che si occupa del recupero degli header può gestire sia il caso classico che il caso in cui l’header sia nul-terminato.

Test e Script

Object Store permette la memorizzazione, dato il puntatore e la dimensione, di qualsiasi tipo di dato (prototipo void* block della funzione os_store); nei test sviluppati vengono memorizzate, recuperate e cancellate delle stringhe (char* opportunamente nul-terminate ‘\0’) contenenti, in numero pari alla dimensione del file meno uno, caratteri da ‘a’ a ‘t’: uno tipologia di carattere per ognuno dei venti file memorizzati da ogni singolo client utilizzato negli esempi (in riferimento al primo test in questo caso).
Sono stati sviluppati, al fine di eseguire e verificare il risultato dei test, due script bash; il primo si occupa semplicemente di mettere in esecuzione il numero richiesto (dal testo del progetto) di client, mentre il secondo si occupa di eseguire delle operazioni di filtraggio sul file di log risultante in modo da recuperare informazioni quali: client eseguiti e numero di client per ogni batteria di test, totalità delle anomalie verificatesi e numero di anomalie per ogni batteria di test.
Si pone inoltre l’attenzione al fatto che l’esecuzione dello script di test necessita la presenza di un’istanza di Object Store in esecuzione; lo script di analisi invece necessita della presenza di un file di log e di un’istanza di Object Store, quest’ultima unicamente per l’invio del segnale di SIGUSR1.
Si riporta di seguito un esempio del formato di output generato dall’esecuzione di un client:

```
####################### usertest15 ##########################

Test battery :                      1
Operations executed :               20
Operations terminated w/ OK :       20
Operations terminated w/o OK [KO] : 0

#############################################################
```

Riportiamo inoltre il risultato di un’esecuzione dello script di analisi:

```
########################################

#Clients executed  : 100

#Clients battery 1 : 50
#Clients battery 2 : 30
#Clients battery 3 : 20

#Clients that reported anomalies : 0
Anomalies battery 1 : 0
Anomalies battery 2 : 0
Anomalies battery 3 : 0

########################################
```

Appendice A – Sistemi Operativi

Per il testing dell’architettura sviluppata sono stati utilizzati i seguenti sistemi operativi:

- Ubuntu 18.04.02 LTS
- Xubuntu 14.10
- Windows w/ Subsystem Linux