

Progetto SOL (ObjStore)

a.a. 2018-2019

Lorenzo Rasoini

Indice

Scelte progettuali	2
1 Strutture dati d'appoggio	2
1.1 linkedlist	2
2 Strutturazione del codice	3
 Funzionamento generale	 4
1 Interazione intra-processo	4
1.1 Comunicazione intra-processo	4
1.2 Gestione della terminazione	4
2 Gestione dei segnali	4
3 Gestione della memoria	5
 Appendices	 5
Appendice A SO su cui è stato testato il programma	5

Scelte progettuali

Di seguito vengono presentate le varie scelte progettuali effettuate durante la realizzazione del progetto

1 Strutture dati d'appoggio

L'unica struttura dati utilizzata all'intero del progetto è `linkedlist` (lista double-linked concorrente) implementata all'interno di `linkedlist.c`

Come per tutte le componenti del progetto la implementazione viene linkata all'eseguibile principale come file oggetto.

1.1 `linkedlist`

```
typedef struct linkedlist_elem {
    void *ptr;
    struct linkedlist_elem *prev;
    pthread_mutex_t *prevmtx;
    pthread_mutex_t mtx;
    pthread_mutex_t *nextmtx;
    struct linkedlist_elem *next;
} linkedlist_elem;
```

Listing 1: Elemento della lista

La lista double-linked è stata scelta per memorizzare i client connessi al server in un dato istante. La scelta è stata dovuta alla facilità di implementazione e all'interesse che mi ha suscitato una sua implementazione in un contesto di programmazione concorrente. Le procedure di ricerca `linkedlist_search` e cancellazione `linkedlist_delete` sono state dotate di un puntatore a una procedura esterna che permette di definire, rispettivamente, quale oggetto trovare e quale oggetto rimuovere.

Gestione della concorrenza

Per gestire accessi e scritture all'interno della lista è stato implementato un meccanismo di "*fine-grained locking*" ritenuto più efficiente in relazione all'utilizzo che viene fatto della lista all'interno del server.

Specificatamente, questo meccanismo è implementato tramite l'utilizzo di una lock: `mtx`, che blocca un singolo elemento della lista, e due puntatori a lock: `prevmtx`, che punta alla lock dell'elemento precedente della lista (se esiste), e `nextmtx`, che punta alla lock dell'elemento successivo alla lista (se esiste).

Per evitare situazioni di deadlock le lock vengono sempre acquisite nello stesso ordine.

L'unica accortezza da avere nell'utilizzo di questa implementazione è l'allocare i dati puntati da `ptr` nell'heap del processo.

2 Strutturazione del codice

Il codice del server è strutturato in moduli che corrispondono a un gruppo di funzionalità separate richieste dal server:

- `os_server.c`

Main del server e i meccanismi di gestione delle interruzioni.

- `dispatcher.c`

Codice relativo al thread dispatcher, e meccanismi di gestione per il segnale `SIGUSR1`.

- `worker.c`

Codice relativo ai worker threads assieme ai meccanismi di ricezione e preparazione dei messaggi inviati.

- `os_client.c`

Parsing dei messaggi e definizione di procedure per la gestione dei comandi individuati dal parser.

- `fs.c`

Definizione di procedure inerenti la scrittura e lettura dei dati inviati e ricevuti dai client.

- `linkedlist.c`

Implementazione di una lista double-linked concorrente.

Il codice dell'interfaccia è interamente contenuto all'interno di `objstore.c`.

L'header file `errormacros.h` contiene varie macro utilizzate per la gestione di eventuali errori lanciati da parte di chiamate di sistema.

Funzionamento generale

1 Interazione intra-processo

Il thread principale, dopo aver creato la socket e aver messo in piedi i meccanismi di gestione dei segnali, crea un thread dispatcher, il quale si occupa di gestire le connessioni che arrivano alla socket. Ogni qualvolta si presenti una connessione il dispatcher controlla che il numero di client connessi non superi 500, in caso positivo accetta la connessione e dopodiché crea un thread worker destinato a gestire tutti i messaggi inviati dal client fino alla disconnessione di quest'ultimo.

1.1 Comunicazione intra-processo

La comunicazione intra-processo è basata su l'utilizzo di due variabili condivise: `OS_RUNNING`, che segnala a tutti i thread del processo la terminazione dello stesso, e `worker_num`, dove al suo interno è memorizzato il numero di client connessi.

1.2 Gestione della terminazione

Il processo di gestione della terminazione ha inizio dopo l'arrivo di un segnale `SIGTERM` o `SIGINT`; in seguito alla ricezione di tale segnale il thread adibito alla gestione dei segnali (il thread principale) setta `OS_RUNNING` a 0.

A questo punto ogni thread worker rileverà il cambiamento della variabile, uscirà dal suo loop e inizierà la procedura di cleanup, notificando al thread dispatcher l'avvenuta terminazione tramite la variabile di condizione `worker_num_cond`.

Appena `worker_num` diventa 0 il thread dispatcher esegue la propria procedura di cleanup, termina e restituisce il controllo al thread principale che farà terminare il processo.

2 Gestione dei segnali

La gestione dei segnali `SIGINT` e `SIGTERM` avviene interamente all'interno del thread principale, il quale, dopo aver creato il dispatcher, si mette in attesa dei due segnali.

Per quanto riguarda la gestione di `SIGUSR1` è stata usata la tecnica del *self-pipe trick*: il thread principale scrive su una pipe letta dal thread dispatcher; quest'ultimo una volta rilevata la presenza di dati nella pipe la svuota ed esegue la procedura `stats`.

3 Gestione della memoria

Per quanto concerne la gestione della memoria si è cercato di usare il più possibile lo stack di ciascun thread e usare le allocazioni sullo heap solo per dati di natura dinamica o di grandi dimensioni (in modo da evitare situazioni di stack overflow).

A SO su cui è stato testato il programma

AppendiceAppendice A

- Arch Linux con kernel versione 5.1.15 e GCC versione 9.1.0