Relazione Progetto Lab II

Yuriy Rymarchuk - 614484 - yura.ita.com@gmail.com

06-05-2022

Panoramica

Il lavoro del thread **Master** è stato implementato dentro al thread main del programma, che rispettivamente:

- Controlla gli argomenti opzionali, usando la funzione `getopt()`, e assegna a ciascuna variabile il suo valore. Se non è stato passato il flag opzionale la variabile viene istanziata con valore di default definite dalle costanti.
- Crea e imposta i segnali con opportuno set di maschera e avviando il thread `Signal_Handler` che intercetta e gestisce i segnali settati con `sigwait()`
- Crea e imposta la struttura del socket address specificando il nome del socket e la famiglia `AF_UNIX`
- Apre i semafori per la sincronizzazione e esegue il `fork()` da cui parte il processo **Collector** a cui viene passato `struct sockaddr_un sa` coi cui aprirà la connessione socket per comunicare con i thread **Worker**
- Crea la struttura `th_struct` e inizializza la coda bounded per la comunicazione con i thread **Worker**
- Crea **N** thread **Worker** passandogli la struttura `th_struct`
- Cicla in loop for sui nomi dei file passati, controllando che siano dei file regolari, per poi inserire i rispetti nomi e la dimensione dei file nella coda `q` utilizzando la struttura `f_struct`
- Aspetta la terminazione di tutti thread con la join, e del processo con la `waitpid()`
- Fa la pulizia della memoria liberando tutte le strutture, chiude il socket della comunicazione, i descrittori di file e i relativi semafori

---

Strutture create

`f_struct`

- Puntatore al nome del file `char *filename`;
- Dimensione del file `size_t filesize`;

`th_struct`

- Descrittore del file socket della connessione server `int fd_skt`;

- Puntatore al semaforo `sem_t *semS`;
- Puntatore al semaforo `sem_t *semC`;
- Puntatore alla coda di comunicazione `BQueue_t *q`;

Segnali

I rispettivi segnali `SIGHUP, SIGINT, SIGQUIT, SIGTERM` vengono aggiunti alla maschera `sigset_t set` che verrà gestito dal thread `sig_handler`, il quale nel caso di ricezione di uno di questi segnali aggiorna la variabile `sig_term` di tipo `volatile sig_atomic_t` a 1. Il thread **Master** controlla nella condizione del for loop se la variabile `sig_term != 1`. Nel caso positivo esce dal loop, quindi smette di inviare i messaggi sulla coda di comunicazione con i thread e comincia a eseguire la chiusura normale del programma e la rispettiva pulizia della memoria. Nel caso di una terminazione normale il thread Master invia un segnale `SIGUSR1` al thread `sig_handler` usando la `pthread_kill()` per poi proseguire con la solita routine di pulizia della memoria.

Thread Worker

Gli worker avviano un ciclo while del fetch dei `f_struct` dalla coda, controllando che non sia il messaggio di terminazione dello stream chiamato `EOS`, nel caso positivo esce dal while, rimette nella coda il messaggio `EOS` e termina la sua routine. Nel caso invece in cui l'elemento ricevuto è un effettivo file da processare, il Worker usa la funzione:
→ `mmap_file(const char *file_name, long **content_ptr, size_t size);`
che salva nel puntatore `long *content` il file mappato in memoria trattandolo come un effettivo array di interi long per poi calcolare il resultato finale e creare la stringa di stampa da inviare al processo **Collector** tramite la socket. L'accesso alla scrittura sul socket viene sincronizzato attraverso due semafori, usando l'unica connessione client aperta, dal main, sul descrittore `th_struct->fd_skt`. Al termine del l'invio del messaggio il thread libera le strutture utilizzate per contenere i dati del file e fa `munmap()` della porzione di memoria dove era contenuto array di interi long.

Collector

Al processo **Collector** viene passato `struct sockaddr_un sa` che rappresenta l'indirizzo con cui aprire il socket di comunicazione coi thread, e il segmento `shmsegment_t *shmptr` che contiene due semafori necessari per sincronizzare la lettura dal socket. La prima cosa che fa il processo è impostare la socket per accettare la connessione, che crea il thread main , mettendosi in ascolto sul socket utilizzando descrittore `fd_skt` che viene legato all'indirizzo passato . La routine principale consiste nella lettura, dal socket, delle stringhe passate, salvate in un buffer locale, e la loro stampa sul `stdout`.

Tutta la routine viene sincronizzata con il metodo dei due semafori `semS` e `semC` situati nel segmento condiviso `shmptr`. Dato che i tutti e due semafori sono stati inizializzati a **1** la prima chiamata `V(semS)` sblocca il thread Worker che è stato il primo a mettersi in attesa con `P(semS)`. Finché il thread non fa finito la scrittura sul socket il Collector rimane bloccato sulla chiamata `P(semC)` dentro al while. Appena il processo ha letto, utilizzando la `read()`, il messaggio lo stampa e sblocca il prossimo thread Worker con la chiamata `V(semS)` alla fine del while. Lo stesso sistema speculare viene usato anche nei Worker per le scritture.

Semafori e Memoria

La scelta di sincronizzare la scrittura sul socket utilizzando i semafori è venuta con l'idea di scrivere un programma che utilizzasse pochissimi byte allocati per creare, gestire e organizzare la connessione socket senza limitare il parallelismo dell'esecuzione del programma. Infatti lanciando il comando per visualizzare il riepilogo sull'uso del heap :

→ `valgrind -s --leak-check=full --show-leak-kinds=all ./farm -n 8 -q 16 file*`

```
==1430==
==1430== HEAP SUMMARY:
==1430==     in use at exit: 0 bytes in 0 blocks
==1430==   total heap usage: 1 allocs, 1 frees, 272 bytes allocated
==1430==
==1430== All heap blocks were freed -- no leaks are possible
==1430==
==1430== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==1428==
==1428== HEAP SUMMARY:
==1428==     in use at exit: 0 bytes in 0 blocks
==1428==   total heap usage: 54 allocs, 54 frees, 3,350 bytes allocated
==1428==
==1428== All heap blocks were freed -- no leaks are possible
==1428==
==1428== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

possiamo notare che, anche con con 8 thread, vengono utilizzati 3600 bytes di memoria, neanche 4K di byte. Se avessimo dovuto creare una connessione socket per ogni thread sicuramente ci sarebbe costato di più in termini di uso della memoria.